TESIS DOCTORAL

Depuración declarativa y verificación heterogénea en Maude

Declarative debugging and heterogeneous verification in Maude

MEMORIA PARA OPTAR AL GRADO DE DOCTOR

PRESENTADA POR

Adrián Riesco Rodríguez

Directores

José Alberto Verdejo López
Narciso Martí Oliert

Madrid, 2011

ISBN: 978-84-695-1107-7

Depuración Declarativa y Verificación Heterogénea en Maude

TESIS DOCTORAL

Memoria presentada para obtener el grado de
Doctor en Informática
Adrián Riesco Rodríguez

Dirigida por los profesores
José Alberto Verdejo López
Narciso Martí Oliet

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

Abril 2011

Declarative Debugging and Heterogeneous Verification in Maude

PhD Thesis

Adrián Riesco Rodríguez

Advisors
José Alberto Verdejo López
Narciso Martí Oliet

Departamento de Sistemas Informáticos y Computación
Facultad de Informática
Universidad Complutense de Madrid

April, 2011

Acknowledgments

I am finishing this PhD thesis being 28 years old. Throughout these years I have met several people that have been important for me and, since I believe that the decisions one takes in his life are strongly influenced by the people surrounding him, I think that they are one of the main reasons I am here.[1] Some of the people listed below could be added in different categories; I have assigned them to the one I consider most fitting.

I would not be here without my parents, Gloria and Juan Julián, that have supported me my whole life (literally). I also thank my sister Virginia, my uncles Antonio, Agustín, Salva, Julín, and Chete my aunts Encarna, Marlena, and Nena and my cousins Rosana, Guillermo, Quique, and Marleni. I specially thank my grandparents Guadalupe, Martín, Juan Julián, and Loli, that cannot share this moment with us.

I started school when I was 2 years old (I was born in October), and I stayed in the same school—Nuestra Señora de las Delicias—until I was 16, so I was there for approximately half my life thus far. I experienced in this period plenty of difficult situations but I still remember lots of anecdotes, and it was there where I met some of my best (and of course oldest) friends. First of all, I have to mention Charlie, with whom I have shared uncountable hours in uncountable places and situations. In addition to him, I am also happy to have shared my time with Remacha, Vincento, Pablé, Carro, Fran, Marisa, Raquel, Lorena, and Paloma.

Being young and idealistic, some of the people listed above decided to create a football team—Palacios A.D.—and see each other when we were not bound by school. We soon realized that we were not enough to endure a whole match, much less to win it. Therefore, we "bought" several players that have become our friends after 13 years of football and "business meals," with Polan, Pablo, Edu, Pesca, Juanito, and Miguelo being the most important of them. As a matter of fact, we are still looking for a sponsor: do not hesitate to contact me if you want to fund us.

I have spent a lot of time in Abánades, a small village in Guadalajara. There, I am part of a two-person guild with Jorge; when we are not playing videogames, running, eating (we are specially proud of our *torrijas*), or reading, we interact with the rest of the world, being the rest of the world in this case Mario, el Rober, Sonia, Alberto, Alfon, el Chini, el Muñeco, and el Matis.

The most important person I met during my years at the university as an undergraduate is Aida. She has always supported and helped me every time I have needed it, and I must remark that our friendship survived even the construction of an AM transmitter in LTC. During these years I can say that she has become part of my family, and I am sure that I owe several of my virtues (if any), as engineer and as person, to her. As a side effect, I am also happy to have met her mother, Sole, who has taken care of me several times, and her friend Maite. The second most important being that I met during these years is la Pelotita. Although it is only a tennis ball, thanks to it a number of people, later known

[1] I just hope I would not be in a better place without them! $\ddot\smile$

as *The Buffer Crowd*, was connected, and by playing with it the difficult moments were easier. These players are Tomás, DaGo, Javi, Fran, Edu, Adam, Luis, Lucas, Raúl, and Bea. I also want to thank two of my professors during this period: Joaquín and Antonio.

Once I started my PhD studies, I saw the world from the dark side: the professors' side. First of all I thank my advisors, Alberto and Narciso, and thank them for their advice, their patience, their books, and their DVDs. I have very much enjoyed the lunches with them and Ricardo, Miguel, Fernando, and Clara. I was also introduced to people working around the world, and I want to highlight Santi, Salva, Paco, Gustavo, and José.

One year after starting my PhD studies I was assigned an office. There (here) I met several people that make the daily life bearable, or even fun. They are Nacho, Enrique, Rober, Juan, dr. Álvez, Manu, Carlos, Quique Grande, and Dani. While working here I have attended many conferences and schools, and I have got to meet very interesting and kind people, that have both worked and partied with me. I would like to mention Mihai, Till, Christian, Sebastian, Remy, Sasha, Dominik, Ewaryst, Berthold, and Kostas.

My research has been supported by the Ministerio de Educación y Ciencia project *DESAFIOS* (TIN2006-15660-C02-01), the Ministerio de Ciencia e Innovación project *DESAFIOS10* (TIN2009-14599-C03-01), the Comunidad de Madrid projects *PROMESAS* (S0505/TIC0407) and *PROMETIDOS* (S2009/TIC-1465), the Ministerio de Educación y Ciencia grants AP2005-007 and AP2005-0076.

Agradecimientos

He acabado esta tesis doctoral con 28 años. A lo largo de estos años he conocido mucha gente que ha sido importante para mi y, dado que pienso que las decisiones que uno toma en la vida se ven influenciadas por quienes le rodean, creo que ellos son unas de las principales razones por las que estoy hoy aquí.[2] Algunas de las personas enumeradas a continuación se podrían haber añadido en diferentes categorías; en estos casos, las he situado en la que he considerado más adecuada.

No estaría aquí sin mis padres, Gloria y Juan Julián, que me han apoyado toda mi vida (literalmente). También quiero mostrar mi agradecimiento a mi hermana Virginia, mis tíos Antonio, Agustín, Encarna, Salva, Marlena, Nena, Julín y Chete y mis primos Rosana, Guillermo, Quique y Marleni. Dedico un agradecimiento especial a mis abuelos Guadalupe, Martín, Juan Julián y Loli, que no pueden compartir este momento conmigo.

Empecé el colegio con 2 años (nací en Octubre), y estuve en el mismo colegio—Nuestra Señora de las Delicias—hasta los 16, lo que hace aproximadamente la mitad de mi vida. Como en todos lados, tuve buenos y malos momentos, pero todavía recuerdo con mucho cariño esos días, y allí es donde conocí a algunos de mis mejores (y obviamente más antiguos) amigos. Dedico un agradecimiento especial a Charlie, con el que he pasado innumerables momentos en los lugares y situaciones más diversos. Además, también me alegra haber conocido a Remacha, Vincento, Pablé, Carro, Fran, Marisa, Raquel, Lorena y Paloma.

Cuando eramos jóvenes y entusiastas unos cuantos amigos decidimos crear un equipo de fútbol—el Palacios A.D.—para seguir viéndonos cuando dejásemos de estudiar juntos. Pronto nos dimos cuenta de que no eramos capaces de ganar un solo partido por nosotros mismos, por lo que decidimos "fichar" a algunos jugadores que, después de 13 años de fútbol y "concentraciones" han acabado siendo amigos, como Polan, Pablo, Edu, Pesca, Juanito y Miguelo. Por cierto, seguimos buscando patrocinador: no dudes en contactar conmigo si quieres serlo.

He pasado mucho tiempo en Abánades, un pequeño pueblo de Guadalajara. Allí soy parte de una peña de dos personas con Jorge; cuando no estamos jugando a la play, corriendo, comiendo (estamos especialmente orgullosos de nuestras torrijas) o leyendo, interactuamos con el resto del mundo, siendo en este caso el resto del mundo Mario, el Rober, Sonia, Alberto, Alfon, el Chini, el Muñeco y el Matis.

La persona más importante que conocí como estudiante de ingeniería es Aida. Siempre me ha apoyado y ayudado cuando lo he necesitado; baste decir como prueba de nuestra amistad que superamos la construcción de un emisor de AM en LTC, lo cual incluye la quema de transistores, que casi supone la pérdida de nuestras huellas dactilares, y un nocivo y gigantesco cristal de cuarzo. Durante estos años he acabado considerándola parte de la familia, y estoy seguro que muchas de mis virtudes (¡en el caso de tener alguna!), como ingeniero y como persona, se las debo a ella. Además, conciéndola he tenido la suerte

[2]¡Solo espero que sin ellos no estuviese hoy en un lugar mejor! $\ddot\smile$

de conocer a su madre, Sole, que siempre se ha preocupado por mí, y a su amiga Maite. El segundo “ser” más importante de los que conocí en la facultad durante aquellos años es *la Pelotita*. Aunque simplemente es una pelota de tenis, gracias a ella un grupo de personas, más tarde conocidas como *The Buffer Crowd*, se conocieron, y jugando con ella se hicieron más llevaderos los momentos complicados. Estos jugadores son Tomás, *DaGo*, Javi, Fran, Edu, Adam, Luis, Lucas, Raúl y Bea. Por último, también quiero nombrar aquí a dos profesores que conocí durante este periodo: Joaquín y Antonio.

Al empezar el doctorado, vi el mundo desde el lado oscuro: el lado de los profesores. En primer lugar quiero agradecer a mis directores, Alberto y Narciso, sus consejos, paciencia, libros y DVD. También me lo he pasado muy bien comiendo con ellos y con Ricardo, Miguel, Fernando y Clara. Además, me han presentado a muchas otras personas como Santi, Salva, Paco, Gustavo y José.

Un año después de empezar el tercer ciclo me asignaron un despacho. Allí (aquí) conocí a muchas personas que hacen que la vida diaria sea divertida. Ellos son Nacho, Enrique, Rober, Juan, dr. Álvez, Manu, Carlos, Quique Grande y Dani. Además, trabajando aquí he asistido a varias conferencias y escuelas y he conocido a mucha gente, con la que he trabajado y salido de fiesta. Me gustaría resaltar a Mihai, Till, Christian, Sebastian, *Remy*, *Sasha*, Dominik, Ewaryst, Berthold, and *Kostas*.

Por último, mi investigación ha sido financiada por el proyecto del Ministerio de Educación y Ciencia *DESAFIOS* (TIN2006-15660-C02-01), el proyecto del Ministerio de Ciencia e Innovación *DESAFIOS10* (TIN2009-14599-C03-01), los proyectos de la Comunidad de Madrid *PROMESAS* (S0505/TIC0407) y *PROMETIDOS* (S2009/TIC-1465), y las becas del Ministerio de Educación y Ciencia AP2005-007 y AP2005-0076.

Abstract

Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error. We can distinguish two phases in this scheme: in the first one the computation is represented as a debugging tree, while in the second one the debugger traverses this tree until the error is found by asking questions related to the adequacy of the judgments stored in the nodes to an external oracle, usually the user, following a navigation strategy. This debugging technique can be used to debug two different kinds of errors: *wrong answers*, that is applied when an *erroneous* result is obtained from an initial value, and *missing answers*, applied when a result is *incomplete*.

The first part of this thesis applies these ideas to build a declarative debugger for rewriting logic specifications in Maude. It shows how we have adapted the standard calculus of rewriting logic to debug wrong answers, while we debug missing answers with a new calculus that extends the one for wrong answers with judgments to infer, given a term, its normal form, its least sort, and the set of reachable terms given a pattern, a condition, and a bound in the number of steps. With this calculus we can detect errors due to wrong and missing statements (equation, membership axioms, and rules) and to wrong search conditions. We have proved that this technique is sound (the error attributed by the debugger is in fact an error) and complete (we can always find the error) if some requirements are fulfilled, namely the specification being debugged is an executable Maude module and the information introduced by the user is correct. Finally, the use of the debugger has been illustrated with several examples.

Formal systems like Maude provide several mechanisms to perform different analyses, but it is unlikely that a single system will ever supply tools for every possible analysis a programmer could require. Moreover, it is natural, when facing large systems, to specify different parts in different ways. For these reason heterogeneous specifications, which allow the programmer to use different formalisms, are becoming more and more important, especially in safety-critical areas where one cannot take the risk of malfunction. One of such systems is the Heterogeneous Tool Set (HETS), a formal integration tool that provides parsing, static analysis, and proof management for heterogeneous specifications by combining several individual specifications languages.

The second part of this thesis describes how to integrate Maude into HETS, in such a way that the tools already integrated in the system can be used to analyze Maude specifications. To achieve it we had to (i) define an institution for Maude, so the Maude logic can be integrated into HETS, (ii) define a comorphism from this institution to the one of CASL, the central logic of HETS, which combines higher-order logic and induction and that has defined comorphisms to the rest of logic integrated in the system, which allows to translate Maude to these logics, and (iii) translate Maude's structuring mechanisms to development graphs, the method used by HETS to represent structured specifications and ease proof management. In our case, Maude modules and theories are represented as nodes, importations as edges, and views as edges with associated proof obligations.

Resumen

La depuración declarativa es un método semi-automático de depuración que, empezando por un cómputo erróneo, localiza la causa del error. Esta técnica consta de dos fases, en la primera el cómputo se representa como un árbol de depuración, mientras que en la segunda el depurador guía al usuario a través del árbol, haciéndole preguntas sobre la adecuación de los resultados almacenados en los nodos respecto a su interpretación pretendida, hasta que se encuentra el error. Este estilo de depuración se puede utilizar para depurar los dos tipos de errores que pueden encontrarse al programar: *respuestas erróneas*, que se dan cuando se obtiene un resultado equivocado, y *respuestas perdidas*, que se dan cuando se obtiene un resultado incompleto.

En la primera parte de esta tesis se aplican estas ideas para construir un depurador declarativo para Maude, que representa especificaciones en lógica de reescritura. En ellos se muestra cómo hemos adaptado el cálculo habitual para lógica de reescritura para depurar respuestas erróneas, mientras que para depurar respuestas perdidas hemos extendido dicho cálculo con juicios que permiten calcular, dado un término, su forma normal, su tipo mínimo y el conjunto de términos alcanzables desde él dadas una cota en el número de pasos y una condición que debe ser satisfecha por dichos términos. Con este cálculo somos capaces de detectar errores debidos a ecuaciones, axiomas de pertenencia o reglas erróneas y omitidas y a errores en las condiciones impuestas sobre los términos alcanzables. Además, hemos demostrado que este método es correcto y completo, es decir, siempre es capaz de detectar la causa del error y dicha causa es una de las descritas anteriormente, siempre y cuando se satisfagan ciertas premisas: las especificaciones cumplen los requisitos de ejecutabilidad de Maude y la información introducida por el usuario es correcta. Finalmente, también presentamos una demostración del sistema con sus principales características.

Los sistemas formales como Maude facilitan al usuario mecanismos para realizar ciertos análisis, pero es poco probable que un único sistema pueda proporcionar alguna vez todas las posibles herramientas de análisis que un programador pueda necesitar. Para solucionar este problema se utilizan especificaciones heterogéneas, que permiten al programador especificar usando diferentes formalismos. Uno de estos sistemas es el conjunto heterogéneo de herramientas (HETS por sus siglas en inglés), una herramienta de integración formal que proporciona parsing, análisis estático y herramientas de demostración para especificaciones heterogéneas al combinar varios lenguajes de especificación individuales.

En la segunda parte de esta tesis se presenta cómo integrar Maude en HETS, de manera que las herramientas ya integradas en el sistema puedan utilizarse para analizar especificaciones de Maude. Para ello ha sido preciso: (i) definir una institución para Maude, de tal manera que Maude puede ser añadido como lógica en HETS, (ii) definir un comorfismo entre la institución previamente descrita para Maude y la institución de CASL, la lógica central de HETS, que combina lógica de primer orden e inducción y que cuenta con comorfismos al resto de lógicas integradas en el sistema, lo que permite traducir Maude a dichas lógicas, y (iii) representar los mecanismos de estructuración de Maude como grafos de desarrollo, el modo de representación usado por HETS, en el que los módulos y teorías se representan como nodos, las importaciones como aristas y las vistas como aristas que conllevan demostraciones. Tras explicar como hemos llevado a cabo estos pasos, ilustramos el uso de la integración con ejemplos.

Contents

List of Figures

Part I

Summary of the Research

Chapter 1

Introduction

This thesis began in Illinois around four years ago, although we did not know it at the time, when, while writing some chapters of the Maude book, Narciso Martí, Francisco Durán, and Santiago Escobar inadvertently introduced an error in the specification of the well-known Fibonacci function:

$$\begin{array}{rcl} \mathit{fib}(0) & = & 0 \\ \mathit{fib}(1) & = & 1 \\ \mathit{fib}(n) & = & \mathit{fib}(n-1) + \mathit{fib}(n-2) \quad \text{if } n > 1 \end{array}$$

Instead of defining this function as shown above, the addition in the third case of the definition was unwittingly written as a multiplication, making the whole specification wrong. Although insignificant, this error took a long time until it was fixed, thus resulting in the project of developing a declarative debugger for Maude specifications.

In fact, programming is a process that usually leads to errors that need to be fixed in order to obtain the behavior intended by the user. For this reason, several mechanisms have been developed to find the errors introduced by the programmer, giving rise to a process called *debugging*.[1] Assume we implement the *Fibonacci* function above in Maude, reproducing as well the error indicated before:[2]

```
op fib : Nat -> Nat .
eq fib(0) = 0 .
eq fib(1) = 1 .
ceq fib(n) = fib(n - 1) * fib(n - 2) if n > 1 .
```

The best known debugging technique is the declaration of *breakpoints*. A breakpoint simply indicates a pause in the execution of a program when a point, previously indicated by the programmer, is reached. In our program above, we could set a breakpoint in the last equation and then continue the execution of the program step-by-step to check the values of `n - 1`, `n - 2`, the application of `fib` to the corresponding results, and the result returned by the function. In this way, we would obtain the values of the recursive calls and the result of the function and could compare them with the ones we expected. Following this step-by-step strategy, several programming languages also provide traces as a mean to debug programs. This approach shows each step to the user (where the *step* depends on the programming language and can usually be customized) and its result.

Note that, since the most used programming paradigm thus far has been the *imperative* one, these approaches are specially well suited for languages of this kind, because they

[1]It seems that the etymology of the word comes from real *bugs* ruining mechanical systems.

[2]Although the Maude syntax has not been introduced yet, I consider this definition is self-explanatory.

$$\dfrac{\dfrac{}{\texttt{2 > 1}} \quad \dfrac{\dfrac{}{\texttt{2 - 1} \to \texttt{1}} \quad \dfrac{}{\texttt{fib(1)} \to \texttt{1}}}{\texttt{fib(2 - 1)} \to \texttt{1}} \quad \dfrac{\dfrac{}{\texttt{2 - 2} \to \texttt{0}} \quad \dfrac{}{\texttt{fib(0)} \to \texttt{0}}}{\texttt{fib(2 - 2)} \to \texttt{0}} \quad \dfrac{}{\texttt{1 * 0} \to \texttt{0}}}{\texttt{fib(2)} \to \texttt{0}}$$

Figure 1.1: Execution tree for the *Fibonacci* example

execute each statement in an order previously defined by the programmer (control commands such as conditionals or loops can easily be followed once the values of the variables are known), but these premises do not hold for declarative languages. The traditional separation between the problem logic (defining *what* is expected to be computed) and control (*how* computations are carried out actually) is a major advantage of these languages, but it also becomes a severe complication when considering the task of debugging erroneous computations. Indeed, the involved execution mechanisms associated to the control make it difficult to apply the typical techniques based on step-by-step trace debuggers employed in imperative languages.

Declarative debugging (also known as *abstract diagnosis* or *algorithmic debugging*) is a debugging technique that abstracts the execution details and focuses on the results, thus being well suited to debug declarative languages. Note that the word *declarative* in declarative debugging stands for this abstraction of the computation details (as said above, *what* is computed) and not for its application to declarative languages, that is, declarative debugging can be (and has been, as we will see in Chapter 3) applied to imperative languages. This technique consists of two different phases: in the first one, a data structure representing the computation—the *debugging tree*—is built, where the result in each node must follow from the ones in its children nodes. This tree is usually built following a formal calculus which allows the designer to prove the soundness and completeness of the technique. In the second phase this structure is traversed following a *navigation strategy* and by asking questions to an external oracle (usually the user) until the error is found.

Recalling our Fibonacci example, assume that we evaluate `fib(2)`. The result returned by our function would be `0`, because the condition holds (`2` is greater than `1`), `2 - 1` is `1`, `fib(1)` is evaluated to `1`, `2 - 2` is `0`, `fib(0)` is evaluated to `0` (these results are expected) and the product of these results is `0`. That is, the debugging tree should have the form shown in Figure 1.1, where the main idea is that the results in each node must follow from the results in its children. Declarative debugging would proceed now by asking to the oracle (the programmer in this case, although sometimes other oracles, such as a correct specification, are used) about the correction of the nodes with respect to the behavior the programmer had in mind while implementing the system, the *intended semantics* of the system. The aim of this navigation is to find an incorrect node (w.r.t. this intended semantics) with all its children correct (w.r.t. this intended semantics): the *buggy node*, that in this case is the root of the tree. In general, the nodes in the tree will be labeled to allow the user to identify the error; this labeling depends on the programming language and on the calculus used to generate the debugging tree; in Maude we can distinguish each equation by means of labels, and thus in this case the debugger would point to the last equation as buggy (in Figure 1.1 the inferences in all nodes except for the root are correct, which intuitively is associated to this last equation).

As we have seen, the debugging process is started by an *initial symptom* that reveals that the program is incorrect, such as the evaluation of `fib(2)` returning `0` above. We distinguish two different kinds of answers (or results) obtained by the system giving rise to these initial symptoms: *wrong answers*, which are erroneous results obtained from a valid input (e.g. the example above); and *missing answers*, which are incomplete results obtained from a valid input (e.g. `fib(2)` being evaluated to `1 + fib(0)`, which is correct but it is not the expected final result). Although both kinds of errors can be debugged by using declarative debugging, debugging of missing answers has been less studied because the calculus involved is much more complex in general than the one used for wrong answers, and it has been related in general to nondeterministic systems, a feature usually associated to declarative languages.

1.1 Declarative debugging for Maude

In this thesis we present a declarative debugger for Maude specifications. *Maude* [20] is a declarative language based on both equational and rewriting logic for the specification and implementation of a whole range of models and systems. Maude functional modules are specifications in membership equational logic, and allow the user to define data types and operations on them by means of *membership equational logic* theories that support multiple sorts, subsort relations, equations, and assertions of membership in a sort. Maude system modules are specifications in rewriting logic that, in addition to the elements present in functional modules, allow the definition of rewrite rules, representing transitions between states. For the time being we are only interested in the fact that equations and memberships are expected to be terminating and confluent, while rules can be both nonterminating and nonconfluent. In the following, we will call *reduction* to the evaluation of a term by using equations, and *rewrite* to the evaluation of a term by using rules and equations (possibly none).

That is, roughly speaking, Maude specifications are composed of equations $t_1 = t_2$, that are understood as t_1 and t_2 are equal, membership axioms $t : s$, stating that t has sort s, and rewrite rules $t_1 \Rightarrow t_2$, indicating that t_1 is rewritten to t_2. With these premises, we can easily identify what kinds of wrong answers can arise in Maude: t_2 is obtained from t_1 by using equations or rules but t_2 should not be obtained from t_1, and the sort s is inferred for t but the term does not have this sort. It is slightly more complex to establish missing answers: given the fact that equations are expected to be terminating and confluent, the user expects to obtain, from an initial term, a single term where equations cannot be further applied (i.e., a *normal form*); thus, if he obtains a term which is right but not in normal form (such as $1 + \mathit{fib}(0)$), then it is a missing answer. Moreover, sorts in Maude can be ordered (by a subset relation), and thus the user expects terms to have a unique least sort; a missing answer is that which infers as the least sort of a term a sort which is correct but not the least one. Finally, missing answers in system modules are easier to characterize: since rules are not expected to be confluent, a term can in general be rewritten to a set of terms; if we get a set smaller than the one expected by the user, then we have a missing answer.

To debug these errors we have developed a formal calculus which allows to infer all the symptoms described above, namely reductions, membership inferences, rewrites, normal forms, least sorts, and sets of reachable terms given certain conditions, which correspond to the ones required by Maude. Using this calculus we are able to build proof trees, that can be used as debugging trees for the declarative debugging process. With these trees we are able to detect several causes that may generate the errors: wrong equations, memberships, rules, and conditions (used to compute the set of reachable terms), and missing equations, membership axioms, and rules (that is, statements that should be part of the specification but are not). Although these proof trees would allow the user to debug his specifications, we have developed a "pruning" technique that shortens and eases the questions asked to the oracle, dealing in this way with one of the main problems of declarative debugging: the number and the complexity of the questions. Moreover, this technique allows the user to build different debugging trees depending on the complexity of the specification or his knowledge about it, which is an original feature of our approach. This technique, that we call *APT* (from *Abbreviated Proof Tree*), discards nodes whose correctness can be inferred from the correctness of their children, and modifies some other nodes in order to ask questions that simulate the behavior of Maude and should be more easily answered by the user. In addition to this technique to improve the proof tree, we also allow the user to use trusting mechanisms to prevent the debugger from asking about certain statements, terms, or modules. Finally, we also provide a graphical user interface to ease the interaction between the users and the tool.

This declarative debugger has been developed in several steps, starting with the minimal functionality and progressively adding more features. This development is reflected by our publications on this subject:

- We started dealing with wrong answers (which, as said above, have an associated calculus simpler than missing answers) in functional modules (which are a "subset" of system modules). The *APT* function applied to the proof trees obtained by the calculus in this initial stage generated a single debugging tree, where some nodes were removed while some others were modified in order to ease their associated questions. This version also allowed the user to trust statements, to trust complete modules, and even to use a correct module as oracle. This work was presented in [13, 14].

- The natural extension to this system is the capability to debug wrong answers in system modules. In addition to defining a new calculus and extending the existing features such as trusting to this kind of debugging, the *APT* transformation for this version of the debugger allowed the user to generate two different debugging trees, depending on the complexity of the application. This work was presented in [86].

- A description of the complete system for debugging wrong answers in Maude specifications was presented in [90].

- With the debugging of wrong answers completed, the next "natural" step was to debug missing answers in functional modules. The calculus involved in this kind of debugging extended the previous one but was much more complex: while the previous one only pointed out what was happening, this new calculus also indicated what was *not* happening. We called these kinds of information *positive* and *negative* respectively. In addition to this new calculus, we added a new trusting mechanism, allowing the user to point out *normal forms*. This work was published in [88].

- Improving the tool, we developed a debugger for both wrong and missing answers for Maude specifications. The calculus for this extension of the tool followed the ideas already stated for debugging missing answers in functional modules: we took into account both positive and negative information. The *APT* transformation also allowed the user to build two new kinds of tree (which are orthogonal to the other types of tree, being the number of possible combinations four) and a new trusting mechanism was implemented: stating of *final* sorts and operators. This work was presented in [87].

- A description of the tool, focusing on the features not described in [90], was presented in [89].

- A complete and integrated description of the system, including all the theorems and proofs, was published in [91].

1.2 Integrating Maude into Hets for proving our specifications correct

We have now fixed all the errors we found in our programs, does it mean they are correct? The best answer we can usually give is *they are correct until the next problem is found.* However, this is not a very satisfactory answer; we would like to *prove* that our programs fulfill some properties such as liveness or satisfaction of some first-order formulas. Formal systems like Maude provide some features that allow the user to perform certain analyses, such as the linear temporal logic model checker or checking of invariants through search, but it is unlikely that a single system will ever supply tools for every possible analysis a programmer could require. Moreover, it is natural, when facing large systems, to specify different parts in different ways. An interesting question arising in such systems is: *how do these parts interact with each other?*

To solve all these problems we use heterogeneous specifications, that allow the programmer to use different formalisms and that are becoming more and more important, especially in safety-critical areas where one cannot take the risk of malfunction. Some of the current heterogeneous approaches deliberately stay informal, like UML. Current approaches have the drawbacks that either they are not formal or they are uni-lateral in the sense that typically there is one logic (and one theorem prover) which serves as the central integration device, even if this central logic may not be needed or desired in particular applications.

The *Heterogeneous Tool Set* (Hets) [61, 64, 65] is a flexible, multi-lateral, and formal (i.e. based on a mathematical semantics) integration tool, providing parsing, static analysis, and proof management for heterogeneous multi-logic specifications by combining various tools for individual specification languages.

Hets is based on a graph of logics and languages, their tools, and their translations. This provides a clean semantics of heterogeneous specifications, as well as a corresponding proof calculus. For proof management it uses the calculus of development graphs [62]. This calculus, known from other large-scale proof management systems like MAYA [5], represents the specifications by using nodes for each programming unit (e.g. modules) and links for the relations between them (e.g. importation relations and proof obligations), providing an overview of the (heterogeneous) specification module hierarchy and the current proof state, and thus may be used for monitoring the overall correctness of a heterogeneous development. To ease heterogeneous specifications Hets provides the heterogeneous specifications language HetCasl. This language is based on Casl [66], the Common Algebraic Specification Language, a language based on first-order logic that works as the central logic in Hets.

In this thesis we describe how we have integrated Maude into Hets, which allows us to use the tools already integrated in Hets (and specially its provers) with Maude specifications. To achieve it we had to (i) define an institution for Maude, and a comorphism from this institution to the Casl one (we will explain institutions and comorphisms in Chapter 2; for the time being, consider an institution as a way of formalizing a logic and a comorphism as a translation between institutions), (ii) define how Maude specifications are translated into development graphs, and (iii) implement mechanisms (at the Casl level, that is, available to all the logics connected to Casl) to deal with freeness constraints, the restrictions imposed by Maude when dealing with a specific importation mechanism that will be explained in the next chapter. This work has been published in [24], while more details can be found in [23].

1.3 Summary

This thesis is divided into three parts: this part summarizes the results presented in this thesis. More concretely:

- Chapter 2 introduces the basic notions required to understand the rest of the thesis. We first introduce rewriting logic and Maude in Section 2.1 and then we introduce institutions and comorphisms in Section 2.2.
- Chapter 3 presents the declarative debugger. We show the unified calculus that allows us to debug both wrong and missing answers, the APT function, and the trusting mechanisms. Finally, we briefly show how to use the graphical user interface.
- Chapter 4 illustrates how to achieve heterogeneous verification. We present HETS, its mechanisms to represent structured specifications (the development graphs), and its techniques to work with different logics, allowing the user to prove properties about Maude specifications with other tools.
- Chapter 5 concludes and outlines some future work.

Part II presents the summary of the research in Spanish, following the same structure as this part. Finally, Part III presents the publications relevant to this thesis as they were originally published.

Chapter 2

Preliminaries

2.1 Maude

As said in the introduction, Maude is a declarative language based on both membership equational logic and rewriting logic. In this section we explain these logics and the Maude modules used to represent them. Much more information can be found in the Maude book [20].

2.1.1 Membership equational logic

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \dots k_n, k}\}_{(k_1 \dots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \dots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of membership equational logic are *equations* $t = t'$, where t and t' are Σ-terms of the same kind, and *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ.

Models of membership equational logic specifications are Σ-*algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \dots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. Given a Σ-algebra $\mathcal{A}$ and a valuation $\sigma : X \longrightarrow \mathcal{A}$ mapping variables to values in the algebra, the meaning $[\![t]\!]^\sigma_{\mathcal{A}}$ of a term t is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies, under a valuation σ,

- an equation $t = t'$, denoted $\mathcal{A}, \sigma \models t = t'$, if and only if both terms have the same meaning: $[\![t]\!]^\sigma_{\mathcal{A}} = [\![t']\!]^\sigma_{\mathcal{A}}$; we also say that the equation holds in the algebra under the valuation.

- a membership $t : s$, denoted $\mathcal{A}, \sigma \models t : s$, if and only if $[\![t]\!]^\sigma_{\mathcal{A}} \in A_s$.

Satisfaction of Horn clauses is defined in the standard way. When a formula ϕ is satisfied by all valuations, we write $\mathcal{A} \models \phi$ and say that $\mathcal{A}$ is a model of ϕ. Finally, when terms are ground, valuations play no role and thus can be omitted. A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of ground terms $[t]_E$, and where an equation or membership is satisfied if and only if it can be deduced from E by means of a sound and complete set of deduction rules [8, 57].

2.1.2 Maude functional modules

Maude functional modules [20, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational logic specifications and their semantics is given by the corresponding initial algebra in the class of algebras satisfying the specification.

In a functional module we can declare sorts (by means of the keyword `sort`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Conditions, in addition to memberships and equations, can also be *matching equations* $t := t'$, whose mathematical meaning is the same as that of an ordinary equation $t = t'$ but that operationally are solved by matching the righthand side t' against the pattern t in the lefthand side, thus instantiating possibly new variables in t.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat]` = `[Nat]`. An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

Functional modules are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness [20]. In this context, their equations $t = t'$ can be oriented from left to right, $t \to t'$ and equational conditions $u = v$ can be checked by finding a common term t such that $u \to t$ and $v \to t$; the notation we will use in the inference rules and debugging trees studied in Section 3.2 for this situation is $u \downarrow v$.

2.1.3 Rewriting logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible.

A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$,[1] where (Σ, E) is an equational specification and R is a set of labeled *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses membership equational logic, as described in the previous sections. A rule q in R has the general conditional form[2]

$$q: \ (\forall X)\ t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$$

where q is the rule label, the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [56] (see also [10]).

Models of rewrite theories are called *$\mathcal{R}$-systems* in [56]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. The rewriting logic deduction rules introduced in [56] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial models. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the initial model $\mathcal{T}_{\Sigma/E,R}$ for $\mathcal{R}$ has an underlying (Σ, E)-algebra $\mathcal{T}_{\Sigma/E}$ whose elements are equivalence classes $[t]_E$ of ground Σ-terms modulo E, and there is a transition from $[t]_E$ to $[t']_E$ when there exist terms t_1 and t_2 such that $t =_E t_1 \Rightarrow^*_R t_2 =_E t'$, where $t_1 \Rightarrow^*_R t_2$ means that the term t_1 can be rewritten into t_2 in zero or more rewrite steps applying rules in R, also denoted $[t]_E \Rightarrow^*_{R/E} [t']_E$ when rewriting is considered on equivalence classes [56, 27].

2.1.4 Maude system modules

Maude system modules [20, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`), whose conditions can be equations, matching equations, memberships, and rewrites.

The executability requirements for equations and memberships in a system module are the same as those of functional modules. With respect to rules, the satisfaction of all

[1]We do not discuss here the more complex formulation of rewriting logic containing frozen information; for more information see [20].

[2]There is no need for the condition listing first equations, then memberships, and then rewrites; this is just a notational abbreviation, since they can be listed in any order. Also note that we use $t \Rightarrow t'$ instead of the more standard notation $t \to t'$ because we are using the latter for reductions with oriented equations.

the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [20] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [101, 20]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rules in R to a term u' such that $u' =_E t'$.

2.1.5 Advanced features

In addition to the modules considered thus far, we present here some other Maude features that will be used throughout this work.

Importation modes

Maude modules can import other modules in three different modes:

- The `protecting` mode (abbreviated as `pr`) indicates that *no junk and no confusion* can be added to the imported module, where junk refers to new terms in canonical form while confusion implies that different canonical terms in the initial module are made equal by equations in the imported module.
- The `extending` mode (abbreviated as `ex`) indicates that junk is allowed but confusion is forbidden.
- The `including` mode (abbreviated as `inc`) allows both junk and confusion.

Theories

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: functional theories and system theories, with the same structure of their module counterparts, but with loose semantics (while modules have initial semantics). Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can also have rules. Unlike modules, theories need not satisfy any of the executability requirements.

Views

We use views to specify how a particular target module or theory satisfies a source theory. In general, there may be several ways in which such requirements might be satisfied by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort and operator in the source theory. The source and target of a view can be any module expression, with the source module expression evaluating to a theory and the target module expression evaluating to a module or a theory. Each view declaration has an associated set of proof obligations, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques. Such proof obligations are not discharged or checked by the system. An important fact about views is that they cannot map labels, and thus we cannot identify the statements in the theory with those in the target module.

Parameterized modules

Maude modules can be parameterized. A parameterized system module has syntax

$$\texttt{mod M\{}X_1 :: T_1, \ldots, X_n :: T_n\texttt{\} is ... endm}$$

with $n \geq 1$. Parameterized functional modules have completely analogous syntax.

The $\texttt{\{}X_1 :: T_1, \ldots, X_n :: T_n\texttt{\}}$ part is called the interface, where each pair $X_i :: T_i$ is a parameter, each X_i is an identifier—the parameter name or parameter label—, and each T_i is an expression that yields a theory—the parameter theory. Each parameter name in an interface must be unique, although there is no uniqueness restriction on the parameter theories of a module. The parameter theories of a functional module must be functional theories.

In a parameterized module M, all the sorts and statement labels coming from theories in its interface must be qualified by their names. Thus, given a parameter $X_i :: T_i$, each sort S in T_i must be qualified as $X_i\$S$, and each label l of a statement occurring in T_i must be qualified as $X_i\$l$. In fact, the parameterized module M is flattened as follows. For each parameter $X_i :: T_i$, a renamed copy of the theory T_i, called $X_i :: T_i$, is included. The renaming maps each sort S to $X_i\$S$, and each label l of a statement occurring in T_i to $X_i\$l$. The renaming percolates down through nested inclusions of theories, but has no effect on importations of modules [20]. Thus, if T_i includes a theory T', when the renamed theory $X_i :: T_i$ is created and included into M, the renamed theory $X_i :: T'$ will also be created and included into $X_i :: T_i$. However, the renaming will have no effect on modules imported by either the T_i or T'; for example, if `BOOL` is imported by one of these theories, it is not renamed, but imported in the same way into M. Moreover, sorts declared in parameterized modules can also be parameterized, and these may duplicate, omit, or reorder parameters.

The parameters in parameterized modules are bound to the formal parameters by *instantiation*. The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc., in the formal parameters to the corresponding sorts, operators (or expressions), etc., in the actual target. The instantiation of a parameterized module must be made with views defined previously.

2.1.6 Conditions and substitutions

Throughout this thesis, and specially in the calculus in Section 3.2, we only consider a special kind of conditions and substitutions that operate over them, called *admissible*, and which correspond to the ones used by Maude. They are defined as follows:

Definition 1 *A condition* $\mathcal{C} \equiv C_1 \wedge \cdots \wedge C_n$ *is* admissible *if, for* $1 \leq i \leq n$*,*

- *C_i is an equation $u_i = u'_i$ or a membership $u_i : s$ and*

$$vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$

- *C_i is a matching condition $u_i := u'_i$, u_i is a pattern and*

$$vars(u'_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$

- *C_i is a rewrite condition $u_i \Rightarrow u'_i$, u'_i is a pattern and*

$$vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j).$$

Definition 2 *A condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, where $\circledast$ denotes a special variable not occurring in the rest of the condition, is* admissible *if $P := t \wedge C_1 \wedge \cdots \wedge C_n$ is admissible for t any ground term.*

Definition 3 *A* kind-substitution*, denoted by κ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \le i \le n} . kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the associated term.*

Definition 4 *A* substitution*, denoted by θ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \le i \le n} . sort(v_i) \ge ls(t_i)$, that is, the sort of each variable is greater than or equal to the least sort of the associated term. Note that a substitution is a special type of kind-substitution where each term has the sort appropriate to its variable.*[3]

Definition 5 *Given an atomic condition C, we say that a substitution θ is* admissible for *C if*

- *C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or*
- *C is a matching condition $u := u'$ and $vars(u') \subseteq dom(\theta)$, or*
- *C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.*

2.1.7 A Maude example: A sale

We illustrate in this section the features explained before by means of an example.[4] Note that the modules, theories, and views shown below are enclosed in parentheses; this notation is used because they are introduced in Full Maude, an extension of Maude used by the debugger that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. First, we specify ordered lists[5] in a parameterized module. This module uses the theory `ORD`, that requires a sort `Elt` and an operator `_<_` over its elements fulfilling the properties of a strict total order:

[3]Kind-substitutions and substitutions are different from other substitutions defined in the literature, like the ones in [8]. There, only kinds are taken into account for substitutions, while here every variable has an associated sort, although some properties are checked at the kind level.

[4]**Caution!** Although the modules below are syntactically correct, we have intentionally committed one error. We will see how to fix this error in the next chapter.

[5]We prefer "ordered lists" over "sorted lists" because "sort" is already used to refer to types in this context.

```
(fth ORD is
  pr BOOL .

  sort Elt .
  op _<_ : Elt Elt -> Bool .

  vars X Y Z : Elt .
  eq [irreflexive] : X < X = false [nonexec] .
  ceq [transitive] : X < Z = true if X < Y = true /\ Y < Z = true [nonexec] .
  ceq [antisymmetric] : X = Y if X < Y = true /\ Y < X = true [nonexec] .
  ceq [total] : X = Y if X < Y = false /\ Y < X = false [nonexec] .
endfth)
```

where the `nonexec` attribute indicates that these equation will not be used to reduce terms. Once this theory is specified, we use it in the `OLIST` module to create generic ordered lists. The sort `List{X}` stands for arbitrary lists, whereas `OList{X}` refers to ordered lists. Since ordered lists are a particular case of lists, we use a subsort declaration to indicate it:

```
(fmod OLIST{X :: ORD} is
  sorts List{X} OList{X} .
  subsort OList{X} < List{X} .
```

We declare now the constructors of these sorts: the empty list, which is also an ordered list, is represented with the operator `nil`, while longer lists are created by means of the juxtaposition operator by adding an element in front of a list:

```
  op nil : -> OList{X} [ctor] .
  op __ : X$Elt List{X} -> List{X} [ctor] .
```

We define when a non-empty list is ordered with a membership axiom indicating that the first element of the list must be smaller than or equal to the one coming after it, and the rest of the list must also be ordered. Note how the operator `_<_` from the theory is used to compare the elements:

```
  vars E E' : X$Elt .
  var  OL : OList{X} .
  vars L L' : List{X} .

  cmb [ol1] : E (E' OL) : OList{X}
   if E < E' or E == E' /\
      E' OL : OList{X} .
```

We need another axiom to state that a singleton list is ordered:

```
  mb [ol2] : E nil : OList{X} .
```

The function `ordIns` orders a list by inserting its elements in an ordered fashion in a new list, with the auxiliary function `insertOrd`:

```
  op ordIns : List{X} -> OList{X} .
  eq [oi1] : ordIns(nil) = nil .
  eq [oi2] : ordIns(E OL) = insertOrd(ordIns(OL), E) .
```

The `insertOrd` function inserts the element in the appropriate position. When it is smaller than the first one it is placed in that position; otherwise, we continue the traversal of the list:

```
  op insertOrd : OList{X} X$Elt -> OList{X} .
  eq [io1] : insertOrd(nil, E) = E nil .
  ceq [io2] : insertOrd(E L, E') = E' (E L)
   if E' < E .
  eq [io3] : insertOrd(E L, E') = E insertOrd(L, E) [owise] .
 endfm)
```

where the `owise` attribute stands for "otherwise," that is, this equation is only applied when no other can.

In our implementation we are interested in using lists of persons, that are defined in the `PERSON` module below. A person, of sort `Person`, is composed of a string standing for the name, a natural number indicating the amount of money, and another string indicating the items he is carrying:

```
(fmod PERSON is
  pr STRING .

  sort Person .

  op [_,_,_] : String Nat String -> Person [ctor] .
```

Moreover, since we want to instantiate the `ORD` theory with persons, we have to specify a binary Boolean function over them to fulfill the `_<_` requirement. We implement a function in such a way that, given two persons, the smallest one is the one with more money. We specify the function in this way to consider the richest persons in the list in the first place:

```
  vars NAME1 NAME2 B1 B2 : String .
  vars M1 M2 : Nat .

  op _<_ : Person Person -> Bool .
  eq [NAME1, M1, B1] < [NAME2, M2, B2] = M1 > M2 .
endfm)
```

The theory and the module are related by means of the view `PersonOrd`, that maps the sort `Elt` in the theory to the sort `Person` in the module. Note that it is not necessary to specify the mapping for the `_<_` operation because it has the same name in both the theory and the module:

```
(view PersonOrd from ORD to PERSON is
  sort Elt to Person .
endv)
```

The module `SALE` specifies a shop and a list of persons waiting to enter and buy items. This module instantiates the module `OLIST` with the view above, and renames the lists from `OList{PersonOrd}` to `OList` to ease its use:

```
(mod SALE is
  pr OLIST{PersonOrd} * (sort OList{PersonOrd} to OList) .
```

Items to be sold are identified by a string with the name and a natural number with their price:

```
  sort Item .
  op <_`,_> : String Nat -> Item [ctor] .
```

A Shop is just a commutative and associative "soup" where persons and items are put together. Thus, both Person and Item are declared as subsorts of Shop. The operator empty stands for the empty shop, while bigger shops are built with the juxtaposition operator, which is associative and commutative, and has empty as identity:

```
  sort Shop .
  subsorts Item Person < Shop .
  op empty : -> Shop [ctor] .
  op __ : Shop Shop -> Shop [ctor assoc comm id: empty] .
```

A Sale is built with a shop enclosed in square brackets and an ordered list of persons "outside" this shop:

```
  sort Sale .
  op [_]_ : Shop OList -> Sale [ctor] .
```

We specify with rules the behavior of people and items. Rule in is in charge of moving people inside the shop:

```
  var  SH : Shop .
  var  P : Person .
  var  OL : OList .
  vars TN PN B : String .
  vars C M : Nat .
  var  S : Sale .

  rl [in] : [ SH ] P OL
  =>        [ SH P ] OL .
```

Rule buy removes an item from the soup and adds it to a person's purchases, decreasing his money (assuming he has enough):

```
  crl [buy] : [ SH < TN, C > [PN, M, B] ] OL
   =>         [ SH [PN, sd(M, C), B + " " + TN] ] OL
   if M >= C .
```

where sd is the symmetrical difference. We also define a function that checks whether a given person has bought the specified object in a sale:

```
  op _buys_in_ : String String Sale -> Bool .
  eq [buy1] : PN buys TN in [ [PN, M, B] SH ] OL = find(B, TN, 0) =/= notFound .
  eq [buy2] : PN buys TN in S = false [owise] .
```

where find is a predefined function that looks for the substring TN in B, starting in the position 0.

We specify now some constants that will be used to test the specification. We use the persons adri and his evil twin, et, and a lettuce l and a videogame v as items:

```
  ops adri et : -> Person .
  eq adri = ["adri", 15, ""] .
  eq et = ["et", 16, ""] .

  ops l v : -> Item .
  eq l = < "lettuce", 6 > .
  eq v = < "videogame", 10 > .
endm)
```

Now, we can use the `search` command, which performs a breadth-first search, to find the different paths where `adri` buys the videogame:

```
Maude> (search [l v] ordIns(et (adri nil)) =>* S:Sale
        s.t. "adri" buys "videogame" in S:Sale .)
search in SALE :[l v]ordIns(et adri nil) =>* S:Sale .

No solution.
```

We find out that it is impossible for `adri` to buy it! Since we know that he should be able to buy the videogame, we conclude that the specification contains an error. We will show in Section 3.4 how to debug it.

2.2 Institutions and comorphisms

We describe in this section some notions that will be used in Chapter 4. Before describing institutions we recall the notions of *category*, *dual category*, *small category*, *functor*, and *natural transformation* [80].

Definition 6 *A* category $\mathbf{C}$ *consists of:*

- *a class* $|\mathbf{C}|$ *of* objects*;*
- *a class* $hom(\mathbf{C})$ *of* morphisms *(also known as* arrows*), between the objects;*
- *operations assigning to each morphism* f *an object* $dom(f)$*, its* domain*, and an object* $cod(f)$*, its* codomain *(we write* $f : A \to B$ *to indicate that* $dom(f) = A$ *and* $cod(f) = B$*);*
- *a composition operator assigning to each pair of morphisms* f *and* g*, with* $cod(f) = dom(g)$*, a* composite *morphism* $g \circ f : dom(f) \to cod(g)$*, satisfying the associative law: for any morphisms* $f : A \to B$*,* $g : B \to C$*, and* $h : C \to D$*,* $h\circ(g\circ f) = (h\circ g)\circ f$*; and*
- *for each object* A*, an identity morphism* $id_A : A \to A$ *satisfying the identity law: for any morphism* $f : A \to B$*,* $id_B \circ f = f$ *and* $f \circ id_A = f$*.*

Definition 7 *For each category* $\mathbf{C}$*, its* dual category $\mathbf{C}^{op}$ *is the category that has the same objects as* $\mathbf{C}$ *and whose arrows are the opposites of the arrows in* $\mathbf{C}$*, that is, if* $f : A \to B$ *in* $\mathbf{C}$*, then* $f : B \to A$ *in* $\mathbf{C}^{op}$*. Composite and identity arrows are defined in the obvious way.*

Definition 8 *A category* $\mathbf{C}$ *is called* small *if both* $|\mathbf{C}|$ *and* $hom(\mathbf{C})$ *are sets and not proper classes.*

Definition 9 *Let* $\mathbf{C}$ *and* $\mathbf{D}$ *be categories. A* functor $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ *is a map taking each* $\mathbf{C}$*-object* A *to a* $\mathbf{D}$*-object* $\mathbf{F}(A)$ *and each* $\mathbf{C}$*-morphism* $f : A \to B$ *to a* $\mathbf{D}$*-morphism* $\mathbf{F}(f) : \mathbf{F}(A) \to \mathbf{F}(B)$*, such that for all* $\mathbf{C}$*-objects* A *and composable* $\mathbf{C}$*-morphisms* f *and* g*:*

- $\mathbf{F}(id_A) = id_{\mathbf{F}(A)}$*,*
- $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ \mathbf{F}(f)$*.*

Definition 10 *A* natural transformation $\eta : \mathbf{F} \to \mathbf{G}$ *between functors* $\mathbf{F}, \mathbf{G} : \mathbf{A} \to \mathbf{B}$ *associates to each* $X \in |\mathbf{A}|$ *a morphism* $\eta_X : \mathbf{F}(X) \to \mathbf{G}(X)$ *in* $\mathbf{D}$ *called the* component *of* η *at* X*, such that for every morphism* $f : X \to Y \in \mathbf{A}$ *we have* $\eta_Y \circ \mathbf{F}(f) = \mathbf{G}(f) \circ \eta_X$*.*

It is worth mentioning two interesting categories: **Set**, the category whose objects are sets and whose morphisms between sets A and B are all functions from A to B; and **Cat**, the category whose objects are all small categories and whose morphisms are functors between them.

Now we are ready to introduce *institutions* and *comorphisms* [39, 38]:

Definition 11 *An* institution *consists of:*

- *a category* **Sign** *of* signatures*;*
- *a functor* $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ *assigning a set* $\mathbf{Sen}(\Sigma)$ *of* Σ*-sentences to each signature* $\Sigma \in |\mathbf{Sign}|$*;*
- *a functor* $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$*, assigning a category* $\mathbf{Mod}(\Sigma)$ *of* Σ-models *to each signature* $\Sigma \in |\mathbf{Sign}|$*; and*
- *for each signature* $\Sigma \in |\mathbf{Sign}|$*, a* satisfaction relation $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ *between models and sentences such that for any signature morphism* $\sigma : \Sigma \to \Sigma'$*,* Σ*-sentence* $\varphi \in \mathbf{Sen}(\Sigma)$ *and* Σ'*-model* $M' \in |\mathbf{Mod}(\Sigma')|$*:*

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \iff \mathbf{Mod}(\sigma)(M') \models_\Sigma \varphi,$$

  which is called the satisfaction condition*.*

Definition 12 *Given two institutions* $\mathcal{I} = (\mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models)$ *and* $\mathcal{I}' = (\mathbf{Sign}', \mathbf{Mod}', \mathbf{Sen}', \models')$*, an* institution comorphism *from* $\mathcal{I}$ *to* $\mathcal{I}'$ *consists of a functor* $\Phi : \mathbf{Sign} \to \mathbf{Sign}'$*, a natural transformation* $\alpha : \mathbf{Sen} \Rightarrow \Phi; \mathbf{Sen}'$*, and a natural transformation* $\beta : \Phi; \mathbf{Mod}' \Rightarrow \mathbf{Mod}$ *(where* $\mathbf{F}; \mathbf{G}$ *stands for functor composition in the diagrammatic order) such that the following satisfaction condition holds for each* $\Sigma \in |\mathbf{Sign}|$*,* $\varphi \in |\mathbf{Sen}(\Sigma')|$*, and* $M' \in |\mathbf{Mod}'(\Phi(\Sigma))|$*:*

$$\beta_\Sigma(M') \models_\Sigma \varphi \iff M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi).$$

Note that several other morphisms between institutions have been defined in the literature [38], but we are only interested in comorphisms for our purposes in this work.

Chapter 3

Declarative Debugging

3.1 State of the art

As said in the introduction, declarative debugging [68] is a debugging technique that, in contrast to traditional debugging techniques such as breakpoints, abstracts away the execution details, which may be hard to follow in general in declarative languages, to focus on results. We distinguish between two different kinds of declarative debugging: debugging of *wrong answers*, that is applied when a *wrong* result is obtained from an initial value and has been widely employed in the logic [98, 52], functional [71, 82, 72], multi-paradigm [11, 54, 15], and object-oriented [12, 44] programming languages; and debugging of *missing answers* [67, 98, 52, 16, 3], applied when a result is *incomplete*, which has been less studied because the calculus involved is more complex than in the case of wrong answers.

Declarative debugging is a two-phase process: it first computes a tree, the so called *debugging tree*, where each node represents a computation step and each result (that is, the effect of each of these computation steps) must follow from the results in the children nodes of the node it is in; and a second phase where this tree is traversed following a navigation strategy and asking each time to an external oracle about the correction of the results in the nodes. This correction relies on the existence of an *intended semantics* of the program, that corresponds to the behavior the user has in mind and it is used to traverse the tree until an incorrect node with all its children correct, the buggy node, is found. This buggy node is assumed to be labeled in order to identify the error in the original program.

We present in this chapter Maude DDebugger, a declarative debugger for Maude specifications. The Maude system provides different mechanisms to debug specifications:

- It can color terms, which consists in printing with different colors the operators used to build a term that does not fully reduce. It eases the task of distinguishing the different functions that should be reduced in a term but it does not help the user to find the reasons they were not reduced.

- It has a tracer, that allows the user to follow the execution of a specification, that is, the sequence of applications of statements that take place. This tracer is highly customizable, providing options to select how the statements, conditions, substitutions, and terms are shown. The same ideas have been applied to the functional paradigm by the tracer *Hat* [18], where a graph constructed by graph rewriting is proposed as a suitable trace structure.

- It can invoke an internal debugger, that allows the user to define breakpoints by selecting some operators or statements. When a breakpoint is found the debugger is entered. There, we can see the current term and execute the next step with tracing turned on.

- Maude also provides a printing attribute that allows to print the values of the variables when a statement with such an attribute is executed. Although this attribute has not been specifically developed for debugging purposes, it can be used for this aim in a similar way to the trace: suspicious statements can be printed showing the values of its variables each time they are executed.

The tracer, the internal debugger, and the print facilities (when used for debugging) present the same drawbacks: they follow the execution of the program, which is not known *a priori* by the user, thus forgetting one of the main points of declarative programs: the abstraction from the execution details; moreover, since they are based on the trace they require the user to traverse it all checking every step, because they cannot guide the process. Our declarative debugger overcomes these problems, allowing the user to debug all kinds of Maude specifications in an easy and natural way.

One of the strong points of our approach is that, unlike other proposals like [16], it combines the treatment of wrong and missing answers and thus it is able to detect missing answers due to both wrong and missing statements. The state of the art can be found in [94], which contains a comparison among the algorithmic debuggers B.i.O. [9] (Believe in Oracles), a debugger integrated in the Curry compiler KICS; Buddha [81, 82], a debugger for Haskell 98; DDT [15], a debugger for TOY; Freja [71], a debugger for Haskell; Hat-Delta [25], part of a set of tools to debug Haskell programs; Mercury's Algorithmic Debugger [54], a debugger integrated into the Mercury compiler; Münster Curry Debugger [53], a debugger integrated into the Münster Curry compiler; and Nude [69], the NU-Prolog Debugging Environment. We extend this comparison by taking into account the features in the latest updates of the debuggers and adding two new ones: DDJ [44], a debugger for Java programs, and our own debugger, Maude DDebugger. This comparison is summarized in Tables 3.1 and 3.2, where each column shows a declarative debugger and each row a feature. More specifically:

- The implementation language indicates the language used to implement the debugger. In some cases front- and back-ends are shown: they refer, respectively, to the language used to obtain the information needed to compute the debugging tree and the language used to interact with the user.

- The target language states the language debugged by the tool.

- The strategies row indicates the different navigation strategies implemented by the debuggers. TD stands for *top-down*, that starts from the root and selects a wrong child to continue with the navigation until all the children are correct; DQ for *divide and query*, that selects in each case a node rooting a subtree with half the size of the whole tree; SS for *single stepping*, that performs a post-order traversal of the execution tree; HF for *heaviest first*, a modification of top-down that selects the child with the biggest subtree; MRF for *more rules first*, another variant of top-down that selects the child with the biggest number of different statements in its subtree; DRQ for *divide by rules and query*, an improvement of divide and query that selects the node whose subtree has half the number of associated statements of the whole tree; MD for the divide and query strategy implemented by the Mercury Debugger; SD for *subterm dependency*, a strategy that allows to track specific subterms that the user has pointed out as erroneous; and HD for the Hat-Delta heuristics.

- Database indicates whether the tool keeps a database of answers to be used in future debugging sessions, while memoization indicates whether this database is available for the current session.
- The front-end indicates whether it is integrated into the compiler or it is standalone.
- Interface shows the interface between the front-end and the back-end. Here, APT stands for the Abbreviated Proof Tree generated by Maude; ART for Augmented Redex Trail, the tree generated by Hat-Delta; ET is an abbreviation of Execution Tree; and step count refers to a specific method of the B.i.O. debugger that keeps the information used thus far into a text file.
- Debugging tree presents how the debugging trees are managed.
- The missing answers row indicates whether the tool can debug missing answers.
- Accepted answers: the different answers that can be introduced into the debugger. `yes`; `no`; `dk` (*don't know*); `tr` (*trust*); `in` (*inadmissible*), used to indicate that some arguments should not have been computed; and `my` and `mn` (*maybe yes* and *maybe no*), that behave as `yes` and `no` although the questions can be repeated if needed. More details about these debugging techniques can be found in [94, 95].
- Tracing subexpressions means that the user is able to point out a subterm as erroneous.
- ET exploration indicates whether the debugging tree can be freely traversed.
- Whether the debugging tree can be built following different strategies depending on the specific situation is shown in the Different trees? row.
- Tree compression indicates whether the tool implements tree compression [25], a technique to remove redundant nodes from the execution tree.
- Undo states whether the tool provides an undo command.
- Trusting lists the trusting options provided by each debugger. MO stands for trusting modules; FU for functions (statements); AR for arguments; and FN for final forms.
- GUI shows whether the tool provides a graphical user interface.
- Version displays the version of the tool used for the comparison.

The results shown in these tables can be interpreted as follows:

Navigation strategies. Several navigation strategies have been proposed for declarative debugging [94]. However, most of the debuggers (including Maude DDebugger) only implement the basic top-down and divide and query techniques. On the other hand, DDJ implements most of the known navigation techniques (some of them also developed by the same researchers), including an adaptation of the navigation techniques developed for Hat-Delta. Among the basic techniques, only DDJ, DDT, and Maude DDebugger provide the most efficient divide and query strategy, Hirunkitti's divide and query [94].

	Maude DDebugger	**B.i.O.**	**Buddha**	**DDJ**	**DDT**	**Freja**
Implementation language	Maude	Curry	Haskell	Java	Toy (front-end) Java (back-end)	Haskell
Target language	Maude	Curry	Haskell	Java	Toy	Haskell subset
Strategies	TD DQ	TD	TD	TD DQ DRQ SS HF MRF HD	TD DQ	TD
Database / Memoization?	NO/YES	NO/NO	NO/YES	YES/YES	NO/YES	NO/NO
Front-end	Independent prog. trans.	Independent prog. trans.	Independent prog. trans.	Independent prog. trans.	Integrated prog. trans.	Integrated compiler
Interface	APT on demand	Step count	ET	ET on demand	ET (XML/TXT)	ET
Debugging tree	Main memory on demand	Main memory on demand	Main memory on demand	Database	Main memory	Main memory
Missing answers?	YES	NO	NO	NO	YES	NO
Accepted answers	`yes no dk tr`	`yes no dk`	`yes no dk in tr`	`yes no dk tr`	`yes no dk tr`	`yes no my mn`
Tracing subexpressions?	NO	NO	NO	NO	NO	NO
ET exploration?	YES	YES	YES	YES	YES	YES
Different trees?	YES	NO	NO	YES	NO	NO
Tree compression?	NO	NO	NO	NO	NO	NO
Undo?	YES	YES	NO	YES	NO	YES
Trusting	MO/FU/FN	MO/FU/AR	MO/FU	FU	FU	MO/FU
GUI?	YES	NO	NO	YES	YES	NO
Version	2.0 (24/5/2010)	Kics 0.81893 (15/4/2009)	1.2.1 (1/12/2006)	2.4 (23/10/2010)	1.2 (29/9/2005)	(2000)

Table 3.1: A comparative of declarative debuggers I

	Maude DDebugger	**Hat-Delta**	**Mercury Debugger**	**Münster Curry Debugger**	**Nude**
Implementation language	Maude	Haskell	Mercury	Haskell (front-end) Curry (back-end)	Prolog
Target language	Maude	Haskell	Mercury	Curry	Prolog
Strategies	TD DQ	HD	TD DQ SD MD	TD	TD
Database / Memoization?	NO/YES	NO/YES	NO/YES	NO/NO	YES/YES
Front-end	Independent prog. trans.	Independent prog. trans.	Independent compiler	Integrated compiler	Independent compiler
Interface	APT	ART (native)	ET on demand	ET	ET on demand
Debugging tree	Main memory on demand	File system	Main memory on demand	Main memory	Main memory on demand
Missing answers?	YES	NO	NO	NO	NO
Accepted answers	`yes no dk tr`	`yes no`	`yes no dk in tr`	`yes no`	`yes no`
Tracing subexpressions?	NO	NO	YES	NO	NO
ET exploration?	YES	YES	YES	YES	NO
Different trees?	YES	NO	NO	NO	NO
Tree compression?	NO	YES	NO	NO	NO
Undo?	YES	NO	YES	NO	NO
Trusting	MO/FU/FN	MO	MO/FU	MO/FU	FU
GUI?	YES	NO	NO	YES	NO
Version	2.0 (24/5/2010)	2.05 (22/10/2006)	Mercury 0.13.1 (1/12/2006)	AquaCurry 1.0 (13/5/2006)	NU-Prolog 1.7.2 (13/7/2004)

Table 3.2: A comparative of declarative debuggers II

Available answers. The declarative debugging scheme relies on an external oracle answering the questions asked by the tool, and thus the bigger the set of available answers the easier the interaction. The minimum set of answers accepted by all the debuggers is composed of the answers *yes* and *no*; Hat-Delta, the Münster Curry Debugger, and Nude do not accept any more answers, but the remaining debuggers allow some others. Other well-known answers are *don't know* and *trust*; the former, that can introduce incompleteness, allows the user to skip the current question and is implemented by B.i.O., DDJ, DDT, Buddha, Mercury, and Maude DDebugger, while the latter prevents the debugger from asking questions related to the current statement and is accepted by DDJ, DDT, Buddha, the Mercury debugger, and Maude DDebugger. Buddha and the Mercury debugger have developed an answer *inadmissible* to indicate that some arguments should not have been computed, redirecting the debugging process in this direction; our debugger accepts a similar mechanism when debugging missing answers in system modules with the answer *the term n is not a solution/reachable*, which indicates that a term in a set is not a solution/reachable, leading the process in this direction. Finally, Freja accepts the answers *maybe yes* and *maybe not*, that the debugger uses as *yes* and *not*, although it will return to these questions if the bug is not found.

Database. A common feature in declarative debugging is the use of a database to prevent the tool from asking the same question twice, which is implemented by DDJ, DDT, Hat-Delta, Buddha, the Mercury debugger, Nude, and Maude DDebugger. Nude has improved this technique by allowing this database to be used during the next sessions, which has also been adopted by DDJ.

Memory. The debuggers allocate the debugging tree in different ways. The Hat-Delta tree is stored in the file system, DDJ uses a database, and the rest of the debuggers (including ours) keep it in main memory. Most debuggers improve memory management by building the tree on demand, as B.i.O., Buddha, DDJ, the Mercury debugger, Nude, and Maude DDebugger.

Tracing subexpressions. The Mercury debugger is the only one able to indicate that a specific subexpression, and not the whole term, is wrong, improving both the answers *no* and *inadmissible* with precise information about the subexpression. With this technique the navigation strategy can focus on some nodes of the tree, enhancing the debugging process.

Construction strategies. A novelty of our approach is the possibility of building different trees depending on the complexity of the specification and the experience of the user: the trees for both wrong and missing answers can be built following either a one-step or a many-step strategy (giving rise to four combinations). While with the one-step strategy the tool asks more questions in general, these questions are easier to answer than the ones presented with the many-steps strategy. An improvement of this technique has been applied in DDJ in [45], allowing the system to balance the debugging trees by combining so called *chains*, that is, sequences of statements where the final data of each step is the initial data of the following one.

Tree compression. The Hat-Delta debugger has developed a new technique to remove redundant nodes from the execution tree, called tree compression [25]. Roughly speaking, it consists in removing (in some cases) from the debugging tree the children of nodes that are related to the same error as the father, in such a way that the father will provide debugging information for both itself and these children. This technique is very similar to the balancing technique implemented for DDJ in [45].

Tree exploration. Most of the debuggers allow the user to freely navigate the debugging tree, including ours when using the graphical user interface. Only the Münster Curry Debugger and Nude do not implement this feature.

Trusting. Although all the debuggers provide some trusting mechanisms, they differ on the target: all the debuggers except Hat-Delta have mechanisms to trust specific statements, and all the debuggers except DDJ, DDT, and Nude can trust complete modules. An original approach is to allow the user to trust some arguments, which currently is only supported by B.i.O. In our case, and since we are able to debug missing answers, a novel trusting mechanism has been developed: the user can identify some sorts and some operators as *final*, that is, they cannot be further rewritten; with this method all nodes referring to "finalness" of these terms are removed from the debugging tree. Finally, a method similar to trusting consists in using a correct specification as an oracle to answer the questions; this approach is followed by B.i.O. and Maude DDebugger.

Undo command. In a technique that relies on the user as oracle, it is usual to commit an error and thus an undo command can be very useful. However, not all the debuggers have this command, with B.i.O., DDJ, Freja, the Mercury debugger, and Maude DDebugger being the only ones implementing this feature.

Graphical interface. A graphical user interface eases the interaction between the user and the tool, allowing him to freely navigate the debugging tree and showing all the features in a friendly way. In [94], only one declarative debugger—DDT—implemented such an interface, while nowadays four tools—DDT, DDJ, Münster Curry Debugger,[1] and Maude DDebugger—have this feature.

Errors detected. It is worth noticing that only DDT and Maude DDebugger can debug missing answers, while all the other debuggers are devoted exclusively to wrong answers. However, DDT only debugs missing answers due to nondeterminism, while our approach uses this technique to debug erroneous normal forms and least sorts.

Other remarks. An important subject in declarative debugging is scalability. The development of DDJ has taken special care of this subject by using a complex architecture that manages the available memory and uses a database to store the parts of the tree that do not fit in main memory. Moreover, the navigation strategies have been modified to work with incomplete trees. Regarding reusability, the latest version of B.i.O. provides a generic interface that allows other tools implementing it to use its debugging features. Finally, the DDT debugger has been improved to deal with constraints.

It is important to point out that there are other approaches for debugging wrong and missing answers. An interesting approach is the abstract diagnosis introduced in [3]. There, the authors present a framework to debug this same problem where, given a specification of the system, the tool is able to identify both wrong and missing statements by using abstract interpretation and without requiring an initial symptom. In a more recent work [2], this system has been improved, being now able to correct wrong implementations.

[1]Only available for Mac OS X.

With respect to other approaches, such as the Maude sufficient completeness checker [20, Chap. 21] or the sets of descendants [37], our tool provides a wider approach since we handle conditional statements and our equations are not required to be left-linear.

3.2 A calculus for debugging

We describe in this section how the debugging trees used by our tool are obtained from a formal calculus, which allows us to prove the soundness and completeness of the technique.

The calculus to debug wrong answers allows us to infer reductions $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$. Its inference rules, shown in Figure 3.1, are an adaptation of the rules presented in [8, 57] for membership equational logic and in [56, 10] for rewriting logic.[2]

This calculus is now extended to infer, from an initial term, its normal form, its least sort, and the set of reachable terms given a bound in the number of steps and a condition to be fulfilled. An interesting point of this calculus is that it provides two different kinds of information: why things happen (that is, why the normal form is reached, why the least sort is inferred, and why the terms are included in the set) but also why other things *do not* happen (that is, why the term is not further reduced, why a lesser sort cannot be inferred, and why no more terms are included into the set). We call the first kind of information, that was also computed by the inference rules in Figure 3.1, *positive information*, while the second kind of information, novel with respect to the previous rules, is called *negative information*. We present in Figures 3.2 and 3.3 the main inference rules of the extended calculus, while more details can be found in [87, 88]. First, we introduce the intuitive ideas of the judgments used in the inference rules. See [91] for their formal definition and the correctness proof of the calculus.

- Given an admissible substitution θ for an atomic condition C, $[C, \theta] \rightsquigarrow \Theta$ indicates that Θ is the set of substitutions that fulfill the atomic condition C and extend θ by binding the new variables appearing in C.
- Given a set of admissible substitutions Θ for an atomic condition C, $\langle C, \Theta \rangle \rightsquigarrow \Theta'$ indicates that Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.
- $\mathit{disabled}(a, t)$ means that the equation or membership a cannot be applied to t at the top.
- $t \to_{\mathit{red}} t'$ declares that the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.
- $t \to_{\mathit{norm}} t'$ is used to show that t' is in normal form with respect to the equations.
- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathit{fulfilled}(\mathcal{C}, t)$ indicates that $\mathcal{C}$ holds when $\circledast$ is substituted by t.
- Given an admissible condition $\mathcal{C}$ as before, $\mathit{fails}(\mathcal{C}, t)$ denotes that $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.
- $t :_{ls} s$ is used to show that $t : s$ and moreover s is the least sort with this property.

[2]This adaptation consists basically in orienting equations from left to right instead of considering them equalities.

(**Reflexivity**)

$$\frac{}{t \Rightarrow t}\,\mathsf{Rf}_{\Rightarrow} \qquad \frac{}{t \rightarrow t}\,\mathsf{Rf}_{\rightarrow}$$

(**Transitivity**)

$$\frac{t_1 \Rightarrow t' \qquad t' \Rightarrow t_2}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow} \qquad \frac{t_1 \rightarrow t' \qquad t' \rightarrow t_2}{t_1 \rightarrow t_2}\,\mathsf{Tr}_{\rightarrow}$$

(**Congruence**)

$$\frac{t_1 \Rightarrow t'_1 \quad \ldots \quad t_n \Rightarrow t'_n}{f(t_1, \ldots, t_n) \Rightarrow f(t'_1, \ldots, t'_n)}\,\mathsf{Cong}_{\Rightarrow} \qquad \frac{t_1 \rightarrow t'_1 \quad \ldots \quad t_n \rightarrow t'_n}{f(t_1, \ldots, t_n) \rightarrow f(t'_1, \ldots, t'_n)}\,\mathsf{Cong}_{\rightarrow}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \ \{\theta(v_j) : s_j\}_{j=1}^{m} \ \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(t) \Rightarrow \theta(t')}\,\mathsf{Rep}_{\Rightarrow}$$
if $t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) \rightarrow \theta(t')}\,\mathsf{Rep}_{\rightarrow}$$
if $t \rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$

(**Equivalence Class**) (**Subject Reduction**)

$$\frac{t \rightarrow t' \quad t' \Rightarrow t'' \quad t'' \rightarrow t'''}{t \Rightarrow t'''}\,\mathsf{EC} \qquad \frac{t \rightarrow t' \quad t' : s}{t : s}\,\mathsf{SRed}$$

(**Membership**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) : s}\,\mathsf{Mb}$$
if $t : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$

Figure 3.1: Semantic calculus for Maude modules

- $t \Rightarrow^{top} S$ means that the set S is formed by all the reachable terms (modulo equations) from t by exactly one rewrite *at the top*.
- $t \Rightarrow^{q} S$ shows that the set S is the set of reachable terms (modulo equations) obtained from t with one application of the rule q at the top.
- $t \Rightarrow_1 S$ denotes that the set S is constituted by all the reachable terms (modulo equations) from t in exactly one step, where the rewrite step can take place anywhere in t (that is, not necessarily at the top).
- $t \rightsquigarrow_n^{\mathcal{C}} S$ indicates that S is the set of all the terms (modulo equations) that satisfy the admissible condition $\mathcal{C}$ and are reachable from t in at most n steps. Similarly, $t \rightsquigarrow\!+_n^{\mathcal{C}} S$ is used when at least one step must be used and $t \rightsquigarrow\!!_n^{\mathcal{C}} S$ when only final terms are wanted.

The main rules of the calculus to infer normal forms and least sorts are shown in Figure 3.2, with the following meaning:

- Rule Norm shows that a term is in normal form when no equations can be applied to it at the top, which is indicated by the *disabled* judgments, and its subterms are also in normal form. Regarding the negative information explained before, this rule uses negative information by means of the *disabled* judgments, that witness that the term cannot be further reduced. Note that we only check the equations whose lefthand side, when considered with the variables at the kind level, match the current expression, which is indicated by the expression $e \ll_K^{top} f(t_1, \ldots, t_n)$; in this way we prevent the calculus from generating trivial subtrees that will not be useful during the debugging process.
- Rule Rdc_1 reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions.
- Rule Rdc_2 reduces a term by reducing one of its subterm to normal form, checking that the subterm was not already in normal form.
- Rule NTr describes the transitivity for $\rightarrow_{norm}$, which shows that to reach the normal form of a term it is necessary to apply at least one equation to it (possibly in its subterms) and then compute the normal form from the term thus obtained. In this case, the inference rule uses positive information provided by the $t \rightarrow_{red} t_1$ judgment.
- Rule Ls is used to infer the least sort of a term. It first computes the normal form of the term, and then it infers a sort for this term such that no lesser sorts can be obtained. In order to prevent the debugger from checking membership axioms that can never infer a sort for the term we restrict the statements to those whose lefthand side matches the term at the kind level. Similarly to the rule Norm above, the *disabled* judgments provide negative information while the membership inference provides the positive one.

We present in Figure 3.3 the main inference rules used to deduce sets of reachable terms:

- Rule Rf_1 indicates that the singleton set with the term is inferred if no more steps can be applied and the term fulfills the condition; similarly, the empty set is returned

$$\frac{disabled(e_1, f(t_1,\dots,t_n)) \quad \dots \quad disabled(e_l, f(t_1,\dots,t_n)) \quad t_1 \rightarrow_{norm} t_1 \quad \dots \quad t_n \rightarrow_{norm} t_n}{f(t_1,\dots,t_n) \rightarrow_{norm} f(t_1,\dots,t_n)} \; \mathsf{Norm}$$
$$\text{if } \{e_1,\dots,e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1,\dots,t_n)\}$$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(l) \rightarrow_{red} \theta(r)} \; \mathsf{Rdc_1} \;\; \text{if } l \to r \Leftarrow \textstyle\bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \in E$$

$$\frac{t \rightarrow_{norm} t'}{f(t_1,\dots,t,\dots,t_n) \rightarrow_{red} f(t_1,\dots,t',\dots,t_n)} \; \mathsf{Rdc_2} \;\; \text{if } t \not\equiv_A t'$$

$$\frac{t \rightarrow_{red} t_1 \quad t_1 \rightarrow_{norm} t'}{t \rightarrow_{norm} t'} \; \mathsf{NTr}$$

$$\frac{t \rightarrow_{norm} t' \quad t' : s \quad disabled(m_1, t') \quad \dots \quad disabled(m_l, t')}{t :_{ls} s} \; \mathsf{Ls}$$
$$\text{if } \{m_1,\dots,m_l\} = \{m \in E \mid m \ll_K^{top} t' \;\wedge\; sort(m) < s\}$$

Figure 3.2: Calculus for normal forms and least sorts

when the condition fails, which is shown by rule $\mathsf{Rf_2}$. The first rule provides positive information indicating why the term is included into the set, while the second one provides negative information proving that the term must not be added.

- Rule $\mathsf{Tr_1}$ applies when at least one more step can be taken. It first checks that the term fulfills the condition. Then, the set of reachable terms in exactly one step is computed, and the set of reachable terms from each of these terms is computed by using the same condition and the bound decreased one step. The result is the union of the sets thus obtained and the initial term. Rule $\mathsf{Tr_2}$ covers the case when the condition does not hold for the initial term; in that case, the computed set does not include it.

- Rule Stp returns the set of reachable terms in exactly one step. It computes, on the one hand, the set of reachable terms by applying one rule at the top and, on the other hand, the set of reachable terms for each of the subterms of the original term. The final set is composed of the reachable terms by rewriting at the top and the substitutions of the terms obtained for the subterms in the appropriate position (applying only one substitution each time).

- Rule Top is in charge of computing the set of reachable terms by applying one rule at the top. Since each rule can produce different terms due to different matchings, each application generates a set of terms. These terms are gathered to obtain the final set. Note that, in order to avoid trivial information of the form $t \Rightarrow^q \emptyset$, we only use rules whose lefthand side matches the term at the kind level. This inference rule provides both positive and negative information, depending on the result of the judgments in the premises: the empty set provides negative information, while in other case the information is positive. This information is later propagated by the rules by means of the judgment $t \Rightarrow_1 S$.

- Rule Rl returns the set of terms obtained by applying a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed,

$$\frac{\mathit{fulfilled}(\mathcal{C},t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}}\ \mathsf{Rf}_1 \qquad \frac{\mathit{fails}(\mathcal{C},t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf}_2$$

$$\frac{\mathit{fulfilled}(\mathcal{C},t)\ \ t \Rrightarrow_1 \{t_1,\ldots,t_k\}\ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1\ \ldots\ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i\ \cup\ \{t\}}\ \mathsf{Tr}_1$$

$$\frac{\mathit{fails}(\mathcal{C},t)\ \ t \Rrightarrow_1 \{t_1,\ldots,t_k\}\ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1\ \ldots\ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr}_2$$

$$\frac{f(t_1,\ldots,t_m) \Rrightarrow^{top} S_t \quad t_1 \Rrightarrow_1 S_1 \quad \cdots \quad t_m \Rrightarrow_1 S_m}{f(t_1,\ldots,t_m) \Rrightarrow_1 S_t\ \cup\ \bigcup_{i=1}^{m}\{f(t_1,\ldots,u_i,\ldots,t_m) \mid u_i \in S_i\}}\ \mathsf{Stp}$$

$$\frac{t \Rrightarrow^{q_1} S_{q_1} \quad \cdots \quad t \Rrightarrow^{q_l} S_{q_l}}{t \Rrightarrow^{top} \bigcup_{i=1}^{l} S_{q_i}}\ \mathsf{Top}\ \text{ if } \{q_1,\ldots,q_l\} = \{q \in R \mid q \ll_K^{top} t\}$$

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1\ \cdots\ \langle C_k, \Theta_{k-1}\rangle \rightsquigarrow \Theta_k}{t \Rrightarrow^{q} \bigcup_{\theta \in \Theta_k} \{\theta(r)\}}\ \mathsf{Rl}\ \text{ if } q : l \Rightarrow r \Leftarrow C_1\ \wedge\ \ldots\ \wedge\ C_k\ \in R$$

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S}\ \mathsf{Red}_1$$

Figure 3.3: Calculus for missing answers

and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms. The kind of information provided by this rule corresponds to the information provided by the substitutions; if the empty set of substitutions is obtained (negative information) then the rule computes the empty set of terms, which also corresponds with negative information proving that no terms can be obtained with this rewrite rule; analogously when the set of substitutions is nonempty (positive information). This information is propagated through the rest of inference rules justifying why some terms are reachable while others are not.

- Rule Red_1 mimics the operational behavior of Maude. It allows the user, to obtain the set of reachable terms from t, to reduce t to its normal form, search the set of reachable terms from the term thus obtained and then reduce these terms to their normal form.

Once this calculus has been introduced, we can build a proof tree for the search shown in Section 2.1.7. We recall that, starting from an initial configuration with two persons waiting on a queue, `adri` and `et`, we tried to find a configuration where `adri` had bought a `videogame`, but such configuration did not exist:

```
Maude> (search [l v] ordIns(adri et nil) =>* S:Sale
        s.t. "adri" buys "videogame" in S:Sale .)
search in SALE :[l v] ordIns(adri et nil) =>* S:Sale .
```

```
No solution.
```

The proof tree for this computation is depicted in Figure 3.4. The inference starts in Figure 3.4(a); to build this tree we need a bound in the number of steps given by the search,[3] which is 5 in this case. In this tree, `oi` stands for `ordIns`, $\mathcal{C}$ for the condition (extended with matching) (`S :=` $\circledast$) `/\ "adri" buys "videogame" in S`, and the subtrees $\bigtriangledown$'s for straightforward proofs that will not be explained in depth. Note that we have added to the inference rules $\mathsf{Rdc_1}$ and Rl the name of the statement being applied (or $\perp$ when it does not have a label) and the operator at the top in the rule Top, and that we have omitted `nil` in some terms (and the proofs related to it) for the sake of conciseness. The leftmost child of the root, T_1, is in charge of computing the reduction of the initial term to its normal form S_n, which stands for `[`l v`]` et et, where l is `< "lettuce", 6 >`, v is `< "videogame", 10 >`, and et is `["et", 16, ""]`, and the rightmost child (†) continues with the search from this term. The leftmost child of this node, T_2, is shown in Figure 3.4(b). It proves that the initial term does not satisfy the condition by matching the term with the pattern, applying the substitution thus obtained, and then reducing the obtained term with the equation `buy2`. The tree to the right of T_2 shows how we obtain the singleton set containing S_1, which stands for `[`l v et`]` et, by evolving one step S_n. The tree shows that, although the subterms cannot be rewritten, the term can be rewritten at the top with the rule `in`, thus introducing the first et into the shop. Finally, the rightmost child continues the search with the bound decreased by one step; this tree is very similar to (†) and will not be further explained.

The tree T_1 is shown in Figure 3.5 and, since it contains the buggy node, it deserves a closer examination. Figure 3.5(a) starts the computation of the normal form of the initial term; its left child reduces the term `oi(adri et)` to its normal form et et, while the right one reduces the terms `l` and `v` and checks that the terms thus obtained are in normal form. We will focus on the left child, since it contains the buggy node and, moreover, the nodes in the right child are very similar to the nodes in the left child. As mentioned before, the left child of the tree reduces `oi(adri et)` to its normal form; to do so, we first obtain the normal form of `adri`—*adri*, which stands for `["adri", 15, ""]`—by applying the corresponding (unlabeled) equation and then checking it is already in normal form by checking that its arguments are in normal form.[4] The reduction continues with T_3, which is shown in Figure 3.5(b), where the left child reduces `et` to its normal form et, while the right child starts the reduction of the function `ordIns` by applying the equation `oi2`, which returns `io(oi(`et`),` *adri*`)`. Figure 3.5(c) presents the tree T_4, which reduces `oi(`et`)` to its normal form et by applying the equations `oi2`, `oi1`, and `io1`. Once this result is obtained, the reduction is finished by the tree T_5 in Figure 3.5(d). There, the equation `io3` is applied in node ($\star$), where, from the term `io(`et`,` *adri*`)` the term et `io(nil,` et`)` is reached, which is clearly wrong because *adri* has disappeared. Since this node has no children and it is erroneous, it is a buggy node and it is associated to a erroneous piece of code: the equation `io3`. The rest of the tree propagates this error until the normal form is obtained. We will show in Section 3.4 how the tool traverses the tree to find the error in the specification.

[3]This value is automatically computed by Maude DDebugger.

[4]The subtree $\bigtriangledown$ indicates that more steps are required: since the term `15` is an abbreviation of `s(s(...s(0)))`, it requires fifteen steps until the base case is reached.

(a)
$$\dfrac{T_1 \qquad \dfrac{T_2 \qquad \dfrac{\dfrac{\dfrac{\triangledown\cdots\triangledown}{S_n \Rightarrow^{\mathtt{in}} \{S_1\}}\,\mathsf{Rl_{in}}}{S_n \Rightarrow^{top} \{S_1\}}\,\mathsf{Top_{[\_]\_}} \qquad \dfrac{\dfrac{}{l\ v \Rightarrow^{top} \emptyset}\,\mathsf{Top_{\_\_}} \quad \dfrac{\dfrac{}{l \Rightarrow^{top} \emptyset}\,\mathsf{Top_{<\_,\_>}}}{l \Rightarrow_1 \emptyset}\,\mathsf{Stp} \quad \dfrac{\dfrac{}{v \Rightarrow^{top} \emptyset}\,\mathsf{Top_{<\_,\_>}}}{v \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{l\ v \Rightarrow_1 \emptyset}\,\mathsf{Stp} \qquad \dfrac{\dfrac{}{et\ et \Rightarrow^{top} \emptyset}\,\mathsf{Top_{\_\_}} \quad \dfrac{\dfrac{}{et \Rightarrow^{top} \emptyset}\,\mathsf{Top_{[\_,\_,\_]}}}{et \Rightarrow_1 \emptyset}\,\mathsf{Stp} \quad \dfrac{\dfrac{}{et \Rightarrow^{top} \emptyset}\,\mathsf{Top_{[\_,\_,\_]}}}{et \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{et\ et \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{S_n \Rightarrow_1 \{S_1\}}\,\mathsf{Stp} \qquad \dfrac{\triangledown\cdots\triangledown}{S_1 \rightsquigarrow^{4}_{\mathcal{C}} \emptyset}\,\mathsf{Tr_1}}{(\dagger)\ S_n \rightsquigarrow^{5}_{\mathcal{C}} \emptyset}\,\mathsf{Tr_1}}{\mathtt{[l\ v]\ oi(adri\ et\ nil)} \rightsquigarrow^{5}_{\mathcal{C}} \emptyset}\,\mathsf{Red_1}$$

(b)
$$\dfrac{\dfrac{\triangledown\cdots\triangledown}{[\mathtt{S} := S_n, \emptyset] \rightsquigarrow \{\mathtt{S} \mapsto S_n\}}\,\mathsf{PatC} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{"adri"\ buys\ "videogame"\ in\ S} \to_{red} \mathtt{false}}\,\mathsf{Rdc_{1_{buy2}}} \quad \dfrac{}{\mathtt{false} \to_{norm} \mathtt{false}}\,\mathsf{Norm}}{\mathtt{"adri"\ buys\ "videogame"\ in\ S} \to_{norm} \mathtt{false}}\,\mathsf{NTr} \qquad \dfrac{}{\mathtt{true} \to_{norm} \mathtt{true}}\,\mathsf{Norm}}{[\mathtt{"adri"\ buys\ "videogame"\ in\ S} = \mathtt{true}, \{\mathtt{S} \mapsto S_n\}] \rightsquigarrow \emptyset}\,\mathsf{EqC_2}}{\langle \mathtt{"adri"\ buys\ "videogame"\ in\ S} = \mathtt{true}, \{\mathtt{S} \mapsto S_n\}\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{\mathit{fails}(\mathcal{C}, S_n)}\,\mathsf{Fail}$$

Figure 3.4: Proof tree for the search in Section 2.1.7: Main tree (a) and T_2 (b)

(a)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{adri} \to_{red} \mathit{adri}}\,\mathsf{Rdc}_{1_\perp} \quad \dfrac{\dfrac{}{\texttt{"adri"} \to_{norm} \texttt{"adri"}}\,\mathsf{Norm} \quad \dfrac{\triangledown}{\texttt{15} \to_{norm} \texttt{15}}\,\mathsf{Norm} \quad \dfrac{}{\texttt{""} \to_{norm} \texttt{""}}\,\mathsf{Norm}}{\mathit{adri} \to_{norm} \mathit{adri}}\,\mathsf{Norm}}{\texttt{adri} \to_{norm} \mathit{adri}}\,\mathsf{NTr}}{\texttt{oi(adri et)} \to_{red} \texttt{oi(}\mathit{adri}\,\texttt{et)}}\,\mathsf{Rdc}_2 \quad T_3}{\dfrac{\texttt{oi(adri et)} \to_{norm} \mathit{et}\ \mathit{et}}{\texttt{[l v] oi(adri et)} \to_{red} \texttt{[l v]}\ \mathit{et}\ \mathit{et}}\,\mathsf{Rdc}_2}\,\mathsf{NTr} \quad \dfrac{\triangledown \quad \dfrac{\triangledown \quad \dfrac{\dfrac{\triangledown \quad \triangledown}{\mathit{l\ v} \to_{norm} \mathit{l\ v}}\,\mathsf{Norm} \quad \dfrac{\triangledown \quad \triangledown}{\mathit{et}\ \mathit{et} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{Norm}}{[\mathit{l\ v}]\ \mathit{et}\ \mathit{et} \to_{norm} [\mathit{l\ v}]\ \mathit{et}\ \mathit{et}}\,\mathsf{Norm}}{\texttt{[l v]}\ \mathit{et}\ \mathit{et} \to_{norm} [\mathit{l\ v}]\ \mathit{et}\ \mathit{et}}\,\mathsf{NTr}}{\texttt{[l v]}\ \mathit{et}\ \mathit{et} \to_{norm} [\mathit{l\ v}]\ \mathit{et}\ \mathit{et}}\,\mathsf{NTr}}{\texttt{[l v] oi(adri et)} \to_{norm} [\mathit{l\ v}]\ \mathit{et}\ \mathit{et}}\,\mathsf{NTr}$$

(b)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{et} \to_{norm} \mathit{et}}\,\mathsf{Rdc}_{1_\perp} \quad \triangledown}{\texttt{et} \to_{norm} \mathit{et}}\,\mathsf{NTr}}{\texttt{oi(}\mathit{adri}\,\texttt{et)} \to_{red} \texttt{oi(}\mathit{adri}\ \mathit{et}\texttt{)}}\,\mathsf{Rdc}_2 \quad \dfrac{\dfrac{}{\texttt{oi(}\mathit{adri}\ \mathit{et}\texttt{)} \to_{red} \texttt{io(oi(}\mathit{et}\texttt{)}, \mathit{adri}\texttt{)}}\,\mathsf{Rdc}_{1_{\mathtt{oi2}}} \quad \dfrac{\dfrac{T_4}{\texttt{io(oi(}\mathit{et}\texttt{)}, \mathit{adri}\texttt{)} \to_{red} \texttt{io(}\mathit{et}, \mathit{adri}\texttt{)}}\,\mathsf{Rdc}_2 \quad T_5}{\texttt{io(oi(}\mathit{et}\texttt{)}, \mathit{adri}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{NTr}}{\texttt{oi(}\mathit{adri}\ \mathit{et}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{NTr}}{\texttt{oi(}\mathit{adri}\,\texttt{et)} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{NTr}$$

(c)

$$\dfrac{\dfrac{}{\texttt{oi(}\mathit{et}\texttt{)} \to_{red} \texttt{io(oi(nil)}, \mathit{et}\texttt{)}}\,\mathsf{Rdc}_{1_{\mathtt{oi2}}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\mathsf{Rdc}_{1_{\mathtt{oi1}}} \quad \dfrac{}{\texttt{nil} \to_{norm} \texttt{nil}}\,\mathsf{Norm}}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\mathsf{NTr}}{\texttt{io(oi(nil)}, \mathit{et}\texttt{)} \to_{red} \texttt{io(nil}, \mathit{et}\texttt{)}}\,\mathsf{Rdc}_2 \quad \dfrac{\dfrac{}{\texttt{io(nil}, \mathit{et}\texttt{)} \to_{norm} \mathit{et}}\,\mathsf{Rdc}_{1_{\mathtt{io1}}} \quad \dfrac{\triangledown \cdots \triangledown}{\mathit{et} \to_{norm} \mathit{et}}\,\mathsf{Norm}}{\texttt{io(nil}, \mathit{et}\texttt{)} \to_{norm} \mathit{et}}\,\mathsf{NTr}}{\texttt{io(oi(nil)}, \mathit{et}\texttt{)} \to_{norm} \mathit{et}}\,\mathsf{NTr}}{\texttt{oi(}\mathit{et}\texttt{)} \to_{norm} \mathit{et}}\,\mathsf{NTr}$$

(d)

$$\dfrac{(\star)\ \dfrac{}{\texttt{io(}\mathit{et}\texttt{,}\ \mathit{adri}\texttt{)} \to_{red} \mathit{et}\ \texttt{io(nil,}\ \mathit{et}\texttt{)}}\,\mathsf{Rdc}_{1_{\mathtt{io3}}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{io(nil,}\ \mathit{et}\texttt{)} \to_{red} \mathit{et}}\,\mathsf{Rdc}_{1_{\mathtt{io1}}} \quad \dfrac{\triangledown \cdots \triangledown}{\mathit{et} \to_{norm} \mathit{et}}\,\mathsf{Norm}}{\texttt{io(nil,}\ \mathit{et}\texttt{)} \to_{norm} \mathit{et}}\,\mathsf{NTr}}{\mathit{et}\ \texttt{io(nil,}\ \mathit{et}\texttt{)} \to_{red} \mathit{et}\ \mathit{et}}\,\mathsf{Rdc}_2 \quad \dfrac{\triangledown \quad \triangledown}{\mathit{et}\ \mathit{et} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{Norm}}{\mathit{et}\ \texttt{io(nil,}\ \mathit{et}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{NTr}}{\texttt{io(}\mathit{et}\texttt{,}\ \mathit{adri}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\,\mathsf{NTr}$$

Figure 3.5: Proof tree T_1 for the reduction to normal form of the initial term: T_1 (a), T_3 (b), T_4 (c), and T_5 (d)

3.3 Debugging trees

By using the proof trees computed with the calculus of the previous section as debugging trees we are able to locate wrong statements, missing statements, and wrong search conditions, that are intuitively defined as follows:[5]

- A specification has a *wrong statement* (where a statement is an equation, a membership axiom, or a rule) if there exists a term such that the user expects the statement to be applied to this term (that is, the user expects the conditions to be fulfilled) but the result of this application is erroneous.
- For a rule $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$, a specification has a wrong rule if the user expects the judgments $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, …, $[C_n, \Theta_{n-1}] \rightsquigarrow \Theta_n$ to happen but the application of Θ_n to r does not provide the set of results expected for this rule.
- A specification has a *wrong condition* $\mathcal{C} \equiv l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ if there exists a term t such that, when the $\circledast$ is substituted by t, either the condition holds but it is not expected to hold or it does not hold but it is expected to.
- A specification has a *missing equation* if there exists a term t such that it is not expected to be in normal form and none of the equations in the specification are expected to be applied to it.
- A specification has a *missing membership* if there exists a term t in normal form such that the computed least sort of t is not the expected one and none of the membership axioms in the specification are expected to be applied to it.
- A specification has a *missing rule* if there exists a term t such that all the rules applied to t at the top lead to judgments $t \Rightarrow^{q_i} S_{q_i}$ expected by the user but their union $\bigcup S_{q_i}$ does not contain all the expected reachable terms from t by using rewrites at the top.

The trees obtained with this calculus could be used as debugging trees, although they present the problems of (i) having several nodes whose correctness does not depend on the specification and (ii) containing some nodes with judgments that, once they are translated to questions to the user, are very difficult to answer. For these reasons we have developed a technique that eases and shortens the debugging process while keeping its soundness and completeness. For each proof tree T, we will use a function $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT when the tree T is clear from the context, to debug Maude specifications. These APT rules, described in [91], provide the following advantages:

- They remove from the tree the nodes whose correctness can be inferred from the correctness of its children, that is, the nodes that do not contain debugging information. For example, the information related to reflexivity is removed from the tree.
- Their use allows the tool to choose between different types of tree, depending on the complexity of the specification and the expertise of the user in charge of the debugging process: the one-step tree, which only contains rewrites in one step, and the many-steps tree, which can contain rewrites in one or more steps; the latter nodes are placed in such a way that the tree becomes more balanced, and thus the navigation strategies require less questions in general to find the buggy node,

[5]See [91] for the proof and the formal definitions.

although these questions can be more complex than the ones found in one-step trees.[6] For example, rule ($\mathbf{APT}_8^o$) builds the one-step tree for wrong answers by removing the transitivity inferences, while rule ($\mathbf{APT}_8^m$) builds the many-steps tree by keeping them. We present here the more general function APT', that returns sets of trees instead of one tree, and that is used as auxiliary function to compute the APT:

$$(\mathbf{APT}_8^o) \quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}_8^m) \quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow}\right\}$$

- They simplify the questions asked to the user because they reproduce more closely Maude's expected behavior (e.g. questions related to reductions skip the intermediate states and always ask about normal forms, while rewrites are applied once the normal form of the term has been reached). For example, rule ($\mathbf{APT}_3$) associates the debugging information in Rdc_1, an inference rule in charge of applying an equation, to the transitivity below it:

$$(\mathbf{APT}_3) \quad APT'\left(\frac{\dfrac{T_1\ \dots\ T_n}{t \to t''}\,\mathsf{Rdc}_1 \quad T'}{t \to t'}\,\mathsf{NTr}\right) =$$

$$\left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T')}{t \to t'}\,\mathsf{Rdc}_1\right\}$$

- They remove the questions difficult to ask and associate the debugging information to nodes related to easier questions. For example, questions related to rewrites at the top are replaced by rewrites in one step by rule ($\mathbf{APT}_4$):

$$(\mathbf{APT}_4) \quad APT'\left(\frac{\dfrac{T_1\ \dots\ T_n}{t \Rightarrow^{top} S'}\,\mathsf{Top} \quad T'_1 \dots T'_m}{t \Rightarrow_1 S}\,\mathsf{Stp}\right) =$$

$$\left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T'_1)\ \dots\ APT'(T'_m)}{t \Rightarrow_1 S}\,\mathsf{Top}\right\}$$

- They are applied without computing the associated proof tree, so they reduce the time and space needed to build the tree.

Given a proof tree T representing an erroneous inference in the calculus presented in Section 3.2, we proved in [91] that:

- $APT(T)$ contains at least one buggy node (completeness).
- Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition (correctness).

[6] Of course, both trees are sound and complete.

We show in Figure 3.6 the many-steps abbreviated proof tree obtained by applying the function APT to the tree in Figure 3.4. The trees $\bigtriangledown'$'s stand for the abbreviation of the $\bigtriangledown$'s in the previous figures. Figure 3.6(a) shows the start of the computation, while Figure 3.6(b) stands for T_1', Figure 3.6(c) for T_2', and Figure 3.6(d) for T_3'. We can quickly appreciate two advantages of the abbreviation: (i) the number of nodes has been reduced from 67 to 31, so the obtained tree has approximately half the size of the original tree; and (ii) the buggy node, marked with ($\star$) in Figure 3.6(b) has changed (by using the rule ($\mathbf{APT}_3$) described above) and gives rise to an easier question.

Although the tree has been greatly reduced, it can be further reduced by using the trusting techniques provided by the debugger:

- Only labeled statements are taken into account when computing the debugging tree. Moreover, modules and statements can be trusted once the module is introduced in the debugger, and statements can even be trusted on the fly. Thus, reductions performed by statements without label in Figure 3.6, that we have annotated with $\bot$, will be removed from the tree. These nodes have been marked with ($\diamondsuit$).

- A correct module can be introduced as oracle. In this way, correct nodes can be removed from the tree without asking the user.

- We can indicate that some constructed terms (terms built only with operators defined with the `ctor` attribute) are *final*, that is, they cannot be further rewritten, by pointing out some sorts or operators. In this way, if a judgment indicates that the set of reachable terms in one step is empty, it will be considered correct and removed from the tree. In our example we can consider as final the sorts `Nat`, `Bool`, `Shop`, and `OList`, and thus terms of theses sorts (and their corresponding subsorts, such as `Person` and `Item` for `Shop`) will be removed from the tree. We have marked these nodes in Figure 3.6 with ($\heartsuit$).

- Moreover, we consider that these constructed terms are in normal form and hence nodes stating this are automatically removed from the tree. The nodes marked with ($\spadesuit$) in Figure 3.6 fulfill this condition and will be removed from the tree.

3.4 Using the debugger

We present in this section how to use the debugger by means of the sale example from Section 2.1.7. Note that the following requirements are necessary for the debugger to work properly: the specification must be executable (see Section 2.1) and the information introduced by the user must be accurate, which includes his answers and the trusting information. Besides all the information given here, several other commands and options are described in [85].

As explained in the previous section, the trusting mechanisms can dramatically reduce the size of the debugging tree. Thus, we state before starting the debugging process that some of the sorts are final with the commands:

```
Maude> (set final select on .)

Final select is on.

Maude> (final select Nat Bool Shop OList .)

Sorts Bool Nat OList Shop are now final.
```

(a)
$$\dfrac{\dfrac{\dfrac{\triangledown'}{(\spadesuit)\ adri \to_{norm} adri}\,\text{Norm}}{(\diamondsuit)\ \texttt{adri} \to_{norm} adri}\,\text{Rdc}_{1\bot} \quad \dfrac{\dfrac{\triangledown'}{(\spadesuit)\ et \to_{norm} et}\,\text{Norm}}{(\diamondsuit)\ \texttt{et} \to_{norm} et}\,\text{Rdc}_{1\bot} \quad T'_1 \quad \triangledown' \quad \triangledown' \quad T'_2 \quad T'_3 \quad \dfrac{\dfrac{\triangledown' \quad \triangledown'}{(\spadesuit)\ l\ v \to_{norm} l\ v}\,\text{Norm} \quad \dfrac{\triangledown' \quad \triangledown'}{(\spadesuit)\ et\ et \to_{norm} et\ et}\,\text{Norm}}{(\spadesuit)\ \texttt{[}l\ v\texttt{]}\ \ et\ et \to_{norm} \texttt{[}l\ v\texttt{]}\ \ et\ et}\,\text{Norm} \quad \dfrac{\triangledown' \dots \triangledown'}{S_1 \rightsquigarrow^4_{\mathcal{C}} \emptyset}\,\text{Tr}_1}{\texttt{[l v] oi(adri et nil)} \rightsquigarrow^5_{\mathcal{C}} \emptyset}\,\text{Red}_1$$

(b)
$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \texttt{nil} \to_{norm} \texttt{nil}}\,\text{Norm}}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\text{Rdc}_{1\text{oi}1} \quad \dfrac{\dfrac{\triangledown' \dots \triangledown'}{(\spadesuit)\ et \to_{norm} et}\,\text{Norm}}{\texttt{io(nil,}et\texttt{)} \to_{norm} et}\,\text{Rdc}_{1\text{io}1}}{\texttt{oi(}et\texttt{)} \to_{norm} et}\,\text{Rdc}_{1\text{oi}2} \quad \dfrac{\dfrac{\dfrac{\triangledown' \dots \triangledown'}{(\spadesuit)\ et \to_{norm} et}\,\text{Norm}}{\texttt{io(nil, }et\texttt{)} \to_{norm} et}\,\text{Rdc}_{1\text{io}1} \quad \dfrac{\triangledown' \quad \triangledown'}{(\spadesuit)\ et\ et \to_{norm} et\ et}\,\text{Norm}}{(\star)\ \texttt{io(}et\texttt{, }adri\texttt{)} \to_{norm} et\ et}\,\text{Rdc}_{1\text{io}3}}{\texttt{oi(}adri\ et\texttt{)} \to_{norm} et\ et}\,\text{Rdc}_{1\text{oi}2}$$

(c)
$$\dfrac{\triangledown' \dots \triangledown' \quad \dfrac{\dfrac{}{(\spadesuit)\ \texttt{false} \to_{norm} \texttt{false}}\,\text{Norm}}{\texttt{"adri" buys "videogame" in S} \to_{norm} \texttt{false}}\,\text{Rdc}_{1\text{buy}2} \quad \dfrac{}{(\spadesuit)\ \texttt{true} \to_{norm} \texttt{true}}\,\text{Norm}}{\mathit{fails}(\mathcal{C}, S_n)}\,\text{Fail}$$

(d)
$$\dfrac{\dfrac{\triangledown' \dots \triangledown'}{S_n \Rightarrow^{\texttt{in}} \{S_1\}}\,\text{Rl}_{\texttt{in}} \quad \dfrac{\dfrac{}{(\heartsuit)\ l \Rightarrow_1 \emptyset}\,\text{Top}_{<\_,\_>} \quad \dfrac{}{(\heartsuit)\ v \Rightarrow_1 \emptyset}\,\text{Top}_{<\_,\_>}}{(\heartsuit)\ l\ v \Rightarrow_1 \emptyset}\,\text{Top}_{\_\_} \quad \dfrac{\dfrac{}{(\heartsuit)\ et \Rightarrow_1 \emptyset}\,\text{Top}_{[\_,\_,\_]} \quad \dfrac{}{(\heartsuit)\ et \Rightarrow_1 \emptyset}\,\text{Top}_{[\_,\_,\_]}}{(\heartsuit)\ et\ et \Rightarrow_1 \emptyset}\,\text{Top}_{\_\_}}{S_n \Rightarrow_1 \{S_1\}}\,\text{Top}_{[\_]\_}$$

Figure 3.6: Abbreviated proof tree for the search in Section 2.1.7: Main tree (a), T'_1 (b), T'_2 (c), and T'_3 (d)

Since we have closely inspected the specification and can be considered to be experts in the subject, we select the many-steps debugging tree:

```
Maude> (many-steps missing tree .)

Many-steps tree selected when debugging missing answers.
```

Finally, we select the top-down navigation strategy, that presents the questions associated to all the children of the root node and asks the user to select one of them as incorrect or to indicate that all of them are correct:

```
Maude> (top-down strategy .)

Top-down strategy selected.
```

We can now start the debugging process by introducing the command:

```
Maude> (missing [l v] ordIns(et adri nil) =>* S:Sale
        s.t. "adri" buys "videogame" in S:Sale .)
```

The debugger builds the tree shown in Figure 3.8, that corresponds to the Figure 3.6 with all the nodes related to trusting information removed. The first series of questions is related to the children of the root:

```
Question 1 :
Is this reduction (associated with the equation oi2) correct?

ordIns(["adri",15,""]["et",16,""]nil) -> ["et",16,""]["et",16,""]nil

Question 2 :
Did you expect [< "lettuce",6 > < "videogame",10 >]
                     ["et",16,""]["et",16,""]nil not to be a solution?

Question 3 :
Are the following terms all the reachable terms from
 [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil in one step?

1 [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil

Question 4 :
Did you expect that no solutions can be obtained from
 [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ?

Maude> (1 : no .)
```

We notice that the first reduction (($\wp$) in Figure 3.7) is incorrect, while the rest of the questions are correct. Thus, we indicate it by typing `(1 : no .)` and this subtree, which is shown in Figure 3.8 (and that corresponds to the leftmost premise of the root of the tree in Figure 3.7), is selected as the current debugging tree.[7] Thus, the next series of questions will be related to the children of this node:

[7]We could also answer `yes` to any of the other questions, which would be removed but the debugging process would not advance. For this reason, this answer is only recommended, when using the top-down navigation strategy, to simplify the presentation of the questions when several options are shown.

(a)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\mathsf{Rdc}_{1_{\mathsf{oi1}}} \quad \dfrac{}{\texttt{io(nil},et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\mathsf{io1}}}}{\texttt{oi(}et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\mathsf{oi2}}} \quad \dfrac{\dfrac{}{\texttt{io(nil,}\ et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\mathsf{io1}}}}{\texttt{io(}et\texttt{,}\ adri\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\mathsf{io3}}}}{(\wp)\ \texttt{oi(}adri\ et\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\mathsf{oi2}}} \quad T''_2 \quad \dfrac{\dfrac{}{S_n \Rrightarrow^{\texttt{in}} \{S_1\}}\,\mathsf{Rl}_{\mathsf{in}}}{S_n \Rrightarrow_1 \{S_1\}}\,\mathsf{Top}_{\mathsf{[\_]\_}} \quad \dfrac{\bigtriangledown' \ldots \bigtriangledown'}{S_1 \rightsquigarrow^4_{\mathcal{C}} \emptyset}\,\mathsf{Tr}_1}{\texttt{[l v] oi(adri et nil)} \rightsquigarrow^5_{\mathcal{C}} \emptyset}\,\mathsf{Red}_1$$

(b)

$$\dfrac{\dfrac{\dfrac{}{\texttt{["adri",15,""]nil} : \texttt{OList}}\,\mathsf{Mb}_{\mathsf{ol2}}}{\texttt{["et",16,""]["adri",15,""]nil} : \texttt{OList}}\,\mathsf{Mb}_{\mathsf{ol1}} \quad \dfrac{}{\texttt{"adri" buys "videogame" in S} \to_{norm} \texttt{false}}\,\mathsf{Rdc}_{1_{\mathsf{buy2}}}}{fails(\mathcal{C}, S_n)}\,\mathsf{Fail}$$

Figure 3.7: Final debugging tree for the search in Section 2.1.7: Main tree (a) and T''_2 (b)

$$\dfrac{\dfrac{}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\ \mathsf{Rdc}_{1_{\texttt{oi1}}} \qquad \dfrac{}{\texttt{io(nil,} \mathit{et}\texttt{)} \to_{norm} \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io1}}}}{(\S)\ \texttt{oi(}\mathit{et}\texttt{)} \to_{norm} \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{oi2}}} \qquad \dfrac{\dfrac{}{\texttt{io(nil, }\mathit{et}\texttt{)} \to_{norm} \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io1}}}}{\texttt{io(}\mathit{et}\texttt{, }\mathit{adri}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io3}}}$$
$$\overline{\qquad\texttt{oi(}\mathit{adri}\ \mathit{et}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}\qquad}\ \mathsf{Rdc}_{1_{\texttt{oi2}}}$$

Figure 3.8: Debugging tree after one answer

$$\dfrac{\dfrac{\dfrac{}{\texttt{io(nil, }\mathit{et}\texttt{)} \to_{norm} \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io1}}}}{(\P)\ \texttt{io(}\mathit{et}\texttt{, }\mathit{adri}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io3}}}}{\texttt{oi(}\mathit{adri}\ \mathit{et}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{oi2}}} \qquad\qquad \dfrac{\dfrac{}{\texttt{io(nil, }\mathit{et}\texttt{)} \to_{norm} \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io1}}}}{\texttt{io(}\mathit{et}\texttt{, }\mathit{adri}\texttt{)} \to_{norm} \mathit{et}\ \mathit{et}}\ \mathsf{Rdc}_{1_{\texttt{io3}}}$$

Figure 3.9: Debugging tree after two (a) and three answers (b)

```
Question 1 :
Is this reduction (associated with the equation oi2) correct?

ordIns(["et",16,""]nil) -> ["et",16,""]nil

Question 2 :
Is this reduction (associated with the equation io3) correct?

insertOrd(["et",16,""]nil,["adri",15,""]) -> ["et",16,""]["et",16,""]nil
```

Now, instead of answering any of these questions, we can switch to the other navigation strategy—divide and query—with the command:

```
Maude> (divide-query strategy .)

Divide & Query strategy selected.

Is this reduction (associated with the equation oi2) correct?

ordIns(["et",16,""]nil) -> ["et",16,""]nil

Maude> (yes .)
```

In this case the debugger has selected the node whose size is the closest one to half the size of the tree, which is the node marked with (§) in Figure 3.8. This inference is correct, and thus the subtree rooted at this node is removed and the tree is updated to the one shown in Figure 3.9(a). The following question, associated to the node (¶), is shown:

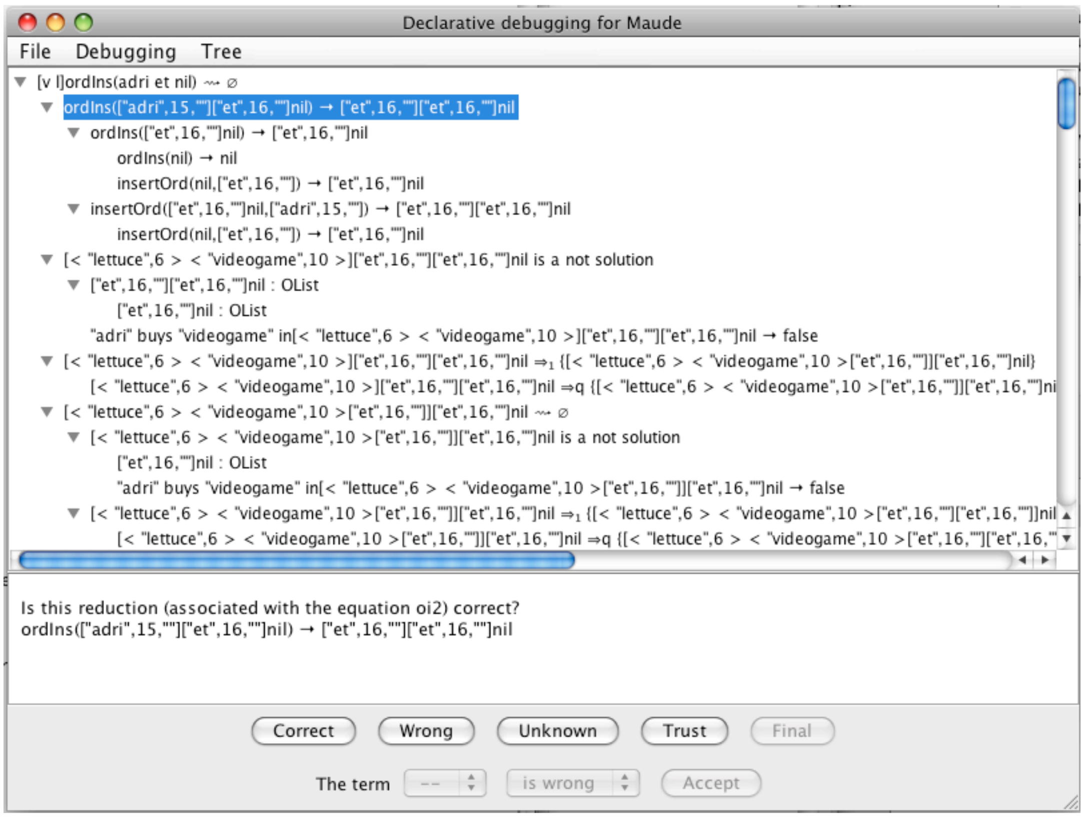
```
Is this reduction (associated with the equation io3) correct?

insertOrd(["et",16,""]nil,["adri",15,""]) -> ["et",16,""]["et",16,""]nil

Maude> (no .)
```

The answer is `(no .)`, and thus the tree is reduced to the tree in Figure 3.9(b). Since the debugger knows the root is erroneous (because it was pointed out by the user in the previous answer) the current question is related to the child of the root:

```
Is this reduction (associated with the equation io1) correct?
```



```
▼ [v l]ordIns(adri et nil) ⇝ ∅
  ▼ ordIns(["adri",15,""]["et",16,""]nil) → ["et",16,""]["et",16,""]nil
    ▼ ordIns(["et",16,""]nil) → ["et",16,""]nil
        ordIns(nil) → nil
        insertOrd(nil,["et",16,""]) → ["et",16,""]nil
    ▼ insertOrd(["et",16,""]nil,["adri",15,""]) → ["et",16,""]["et",16,""]nil
        insertOrd(nil,["et",16,""]) → ["et",16,""]nil
  ▼ [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil is a not solution
    ▼ ["et",16,""]["et",16,""]nil : OList
        ["et",16,""]nil : OList
      "adri" buys "videogame" in[< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil → false
  ▼ [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil ⇒1 {[< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil}
      [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil ⇒q {[< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]ni
  ▼ [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ⇝ ∅
    ▼ [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil is a not solution
        ["et",16,""]nil : OList
        "adri" buys "videogame" in[< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil → false
    ▼ [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ⇒1 {[< "lettuce",6 > < "videogame",10 >["et",16,""]["et",16,""]]nil
        [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ⇒q {[< "lettuce",6 > < "videogame",10 >["et",16,""]["et",16,"
```

```
Is this reduction (associated with the equation oi2) correct?
ordIns(["adri",15,""]["et",16,""]nil) → ["et",16,""]["et",16,""]nil
```

Figure 3.10: Debugging with the graphical user interface

```
insertOrd(nil,["et",16,""]) -> ["et",16,""]nil

Maude> (yes .)
```

With this information the root of the tree in Figure 3.9(b) contains incorrect information but all its children are correct, and thus it is the buggy node.[8] The debugger prints the following information:

```
The buggy node is:
insertOrd(["et",16,""]nil,["adri",15,""]) -> ["et",16,""]["et",16,""]nil
with the associated equation: io3
```

Thus, the buggy equation is `io3`, that was specified in Section 2.1.7 as follows:

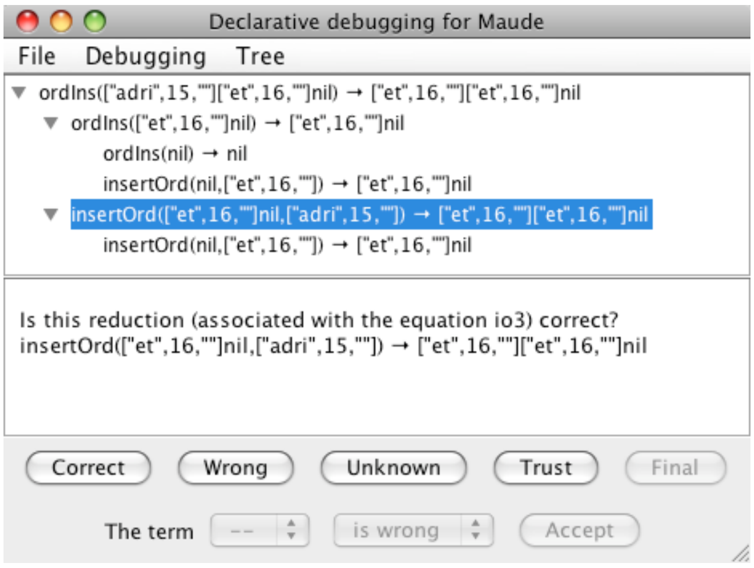
```
 eq [io3] : insertOrd(E L, E') = E insertOrd(L, E) [owise] .
```

that is, we forgot about `E'` in the recursive call. The correct code for the function is:

```
 eq [io3] : insertOrd(E L, E') = E insertOrd(L, E') [owise] .
```

We show in Figure 3.10 how this example can also be debugged with the graphical user interface. In addition to the navigation strategies implemented by the tool, when

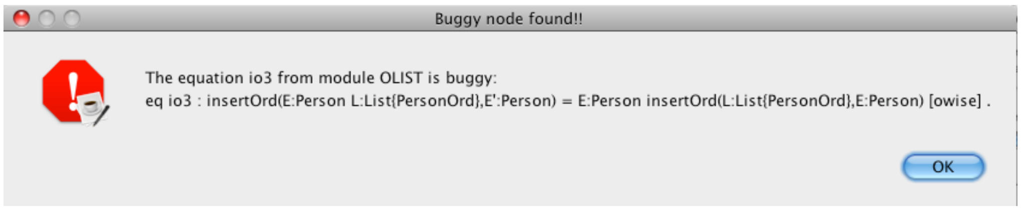
[8]This node is the one marked with $(\star)$ in Figure 3.6, indicating that it was the buggy node.



```
▼ ordIns(["adri",15,""]["et",16,""]nil) → ["et",16,""]["et",16,""]nil
  ▼ ordIns(["et",16,""]nil) → ["et",16,""]nil
      ordIns(nil) → nil
      insertOrd(nil,["et",16,""]) → ["et",16,""]nil
  ▼ insertOrd(["et",16,""]nil,["adri",15,""]) → ["et",16,""]["et",16,""]nil
      insertOrd(nil,["et",16,""]) → ["et",16,""]nil
```

Is this reduction (associated with the equation io3) correct?
insertOrd(["et",16,""]nil,["adri",15,""]) → ["et",16,""]["et",16,""]nil

Figure 3.11: Graphical user interface after one answer



The equation io3 from module OLIST is buggy:
eq io3 : insertOrd(E:Person L:List{PersonOrd},E':Person) = E:Person insertOrd(L:List{PersonOrd},E:Person) [owise] .

Figure 3.12: Information provided by the graphical user interface

using the interface it is also possible to freely navigate the debugging tree and select the different nodes to see the associated question. In this case we have selected a node that corresponds to the root of the tree in Figure 3.8 and whose associated reduction is not correct. By using the `Wrong` button we obtain the tree shown in Figure 3.11, where only the information relevant to the debugging process is shown[9] and where we have selected another wrong node; once we point it as wrong and its child as correct the interface detects it is a buggy node, showing the information in Figure 3.12.

3.5 Contributions

In this chapter we have presented a declarative debugger for Maude specifications. The main contributions of this work are the following:

Debugging of wrong answers. The tool is able to debug wrong answers in both functional (wrong reductions and membership inferences) and system (wrong rewrites) modules, where all kinds of Maude features, such as equational attributes, `frozen`, and `otherwise` can be used. It points out the wrong statement responsible for the error in the specification.

[9]Another behavior is also available when debugging with the GUI: It keeps all the nodes, associating to each of them a color indicating its status: correct, wrong, don't know, and not answered.

Debugging of missing answers. The tool can also debug missing answers in both functional (not completely reduced normal forms and least sorts bigger than the expected ones) and system (incomplete sets of reachable terms given a condition and a bound in the number of steps) modules. The detected causes in this kind of debugging are wrong statements and search conditions, and missing statements; when a missing statement is the cause of the error, the debugger is able to identify the operator at the top in the lefthand side of the missing statement.

Formal calculus. Since the trees used in the debugging process are obtained using a formal calculus, we have been able to prove the completeness and correctness of the technique.

Features. The debugger incorporates most of the features already present in other declarative debuggers:

- Techniques to shorten and improve the debugging tree, that we have called APT.
- Different navigation strategies: top-down and divide and query.
- Several different answers: in addition to the usual answers *yes* and *no* we provide an *undo* command to return to the previous state, *don't know* to skip the current question, and *the term n is not a solution/reachable* to point out that a term in a set is not a solution or reachable, instructing the tool to lead the process in this direction.
- Trusting mechanisms for statements (before starting the debugging process and on the fly), modules, normal forms, and final sorts (also before starting the process and on the fly).
- A graphical user interface, that eases the debugging process by showing the debugging tree and allowing the user to navigate it freely.

Novel features. We have developed an original feature: different debugging trees can be built depending on different factors. This idea has been reused to balance debugging trees in different ways in [45]—with the participation of the author—, and is currently used in the DDJ debugger.

Chapter 4

Heterogeneous Verification

4.1 State of the art

As pointed out in the introduction, complex systems usually require different formalisms to define its parts. For example, it is usual to apply different approaches to deal with the specification of the system, the databases, or the proofs about its expected behavior. Since a single system providing all the necessary features for all the possible requirements is unlikely, heterogeneous systems, that provide several formalisms and combinations between them, are of growing importance nowadays, because they allow to use the appropriate formalism for each part of the system and to combine them to build up the complete structure. Several tools have been proposed to deal with heterogeneous specification and verification:

UML Probably, the best known is the Unified Modeling Language (UML) [6], a language designed to ease the software development process. However, UML intentionally lacks a formal semantics and thus it is not a formal tool.

OMDoc A more formal approach is the one of OMDOC [48], an ontology language for mathematics. This language allows to represent objects such as formulas in a language extending XML called OpenMath, statements such as definitions or proofs, and theories and morphisms between them.

IMPS The Interactive Mathematical Proof System (IMPS) [34] is oriented towards mathematical reasoning. It provides a database of mathematics (represented as a network of axiomatic theories linked by theory morphisms) and a set of tools to relate and modify them.

SpecWare Specware [47] is a tool to facilitate the development of software that allows the user to formally specify the requirements and generate provably correct code for them. It can be used as a design tool, to describe complex systems; as a logic, to formally describe the requirements; and as a programming language, to implement the programs.

Prosper The PROSPER toolkit [26], based on the theorem prover HOL98 [74], provides several decision procedures and model checkers, available in a language-independet way by means of libraries. Currently, libraries for Java, C, and ML are available.

Twelf Twelf [79] is a research project concerned with the design, implementation, and application of logical frameworks. It provides the LF logical framework [41], used to describe logics, the Elf constraint logic programming language, and an Emacs interface. An inductive meta-theorem prover for LF is currently under development.

Delphin Delphin [83] is a functional language organized in two levels: a data level, where deduction systems can be specified; and the computation level, where these systems are manipulated.

Hets The Heterogeneous tool set, HETS [64, 63, 66, 7], incorporates in a common framework several provers and specification languages, and provides mechanisms to relate all of them. In HETS each logic is represented as an institution and translations are represented as institution comorphisms (see Section 2.2).

For our purposes in this thesis we are interested in HETS for a number of reasons:

- It is formal, so it allows the user to reason about the mathematical properties of his specifications, unlike other approaches such as UML or OMDOC, that only formalize some features (if any) of their approaches.
- While several approaches are unilateral, in the sense that only one logic (and one theorem prover) can be used to encode the problem (this is the case of the PROSPER toolkit and OMDOC), specifications can be introduced in HETS in any of the supported logics.
- It focuses on codings between logics, unlike other approaches such as OMDOC, SpecWare, or IMPS, that focus on codings between theories. The former allows to reason about different elements in different logics, while the latter only about different elements in the same logic.
- A current limitation of HETS is that it allows the users to use the implemented logics, but not to add new logics to the system, nor reason directly about them or about their translations, as is the case of Delphin, a logical framework where the syntax and the semantics of different logics can be represented, allowing the user to reason about them. For this reason, the ongoing project LATIN (Logic Atlas and Integrator) [49] intends to integrate these two different approaches, being the first step of the integration of LF with HETS, thus allowing the user to work with the logics integrated into HETS as usual data.

The general schema for HETS is presented in Figure 4.1. It requires parsing and static analysis for each specific tool to work with them in a common framework, where each tool can be related to the others. The relations between them are specified in the logic graph, that has as the central logic the Common Algebraic Specification Language (CASL), a language whose development was proposed by the Common Framework Initiative for algebraic specification and development, CoFI [40], that wanted to unify the different algebraic languages available, incorporating the main features of all these languages and fixing its syntax and semantics. The aim of the initiative was to create a language for specification of functional requirements; formal development of software; relation of specifications to informal requirements and implemented code; prototyping, theorem-proving, and formal testing; and tool interoperability. Following these ideas CASL was designed as a language based on first-order logic and induction by means of constructor constraints.

However, CASL was not thought as a standalone language, but as the heart of a family of languages, some of them obtained by restricting CASL and some other obtained by extending it. Hence, HETS is the result of broadening this idea by relating other

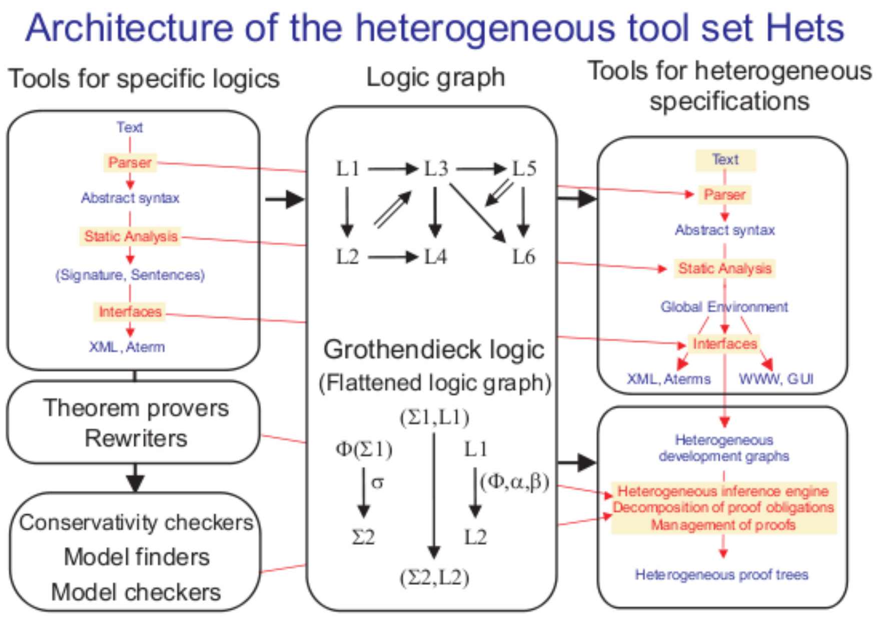


Figure 4.1: Architecture of Hets

languages independent from Casl with it; in this way it is be possible to use other tools (and hence other logics) with Casl specifications and vice versa. The logics that are currently connected with Casl, and thus supported by Hets, are:

General purpose logics: propositional logic, Casl and HetCasl (first-order logic), QBF (Quantified Boolean Formulas), and TPLP/SoftFOL (softly typed first-order logic). See [7] for more details on these languages.

Logical frameworks: LF [41], a framework to define logics; and DFOL [96], a sublogic of LF for First-Order Logic with Dependent Types.

Ontologies and constraint languages: OWL [55], the web ontology language; Common Logic [46], a framework to ease the transmission of information in computer systems; RelScheme [51], a language for relational database schemes; and ConstraintCasl, an extension of Casl with support for constraints [63].

Reactive systems: CspCasl [92], an extension of Casl that supports the process algebra CSP; CoCasl [93], a coalgebraic extension of Casl; and ModalCasl [60], that extends Casl with modal operators. See [63] for details.

Programming languages: Haskell [78], a lazy functional language.

Logics of specific tools: Reduce [42], an interactive system for general algebraic computations, and DMU [43], a dummy logic to read output of "Computer Aided Three-dimensional Interactive Application" (CATIA).

Provers: the interactive higher-order prover Isabelle [73]; the interactive prover for dynamic logic VSE [4]; an Isabelle-based prover for CspCasl [75]; the SAT solvers zChaff [58] and MiniSat [33]; the first-order automated provers SPASS [102], Vampire [100], Darwin [35], KRHyper [103], and MathServe [104]; and the description logic tableau provers Pellet [19] and FaCT++ [99].

From the point of view of HETS, the integration of Maude as a new logic provides several advantages: it will be the first dedicated rewriting engine (currently, only Isabelle's engine is available, which is specialized towards higher-order proofs), allowing to execute specifications implemented in other languages, and supporting model checking of linear temporal logic formulas.

From the Maude point of view, an integration into HETS is very interesting because it allows the user to check properties of the specifications by using the provers listed above. The Maude system has been involved in different approaches to theorem proving:

- Different theorem provers and checkers have been implemented for Maude in Maude itself, taking advantage of its metalevel capabilities, including: the Inductive Theorem Prover, ITP [22], whose last version supports several combinations of equational axioms and induction over constructors, although it only allows proofs in Church-Rosser equational theories; the termination checker [30]; the coherence checker [32]; and the Church-Rosser checker [31].

- A mapping between the logic of HOL [74], a proof system based on higher-order logic, and the logic of Nuprl [1], used to develop software systems and formal theories of computational mathematics, was implemented in Maude [70] following a formal approach.

- Currently, we have been informed that a new project is devoted to translate Maude specifications to the PVS theorem prover [76]. The transformation, written in Maude itself, has been driven by examples from security systems specifications, and therefore only the elements appearing in them (a subset of functional modules) are transformed. However, the system is still in development and more features are expected in the future.

The main drawback of all these approaches is that, as for some other tools listed above, they focus on the Maude system. Our integration would "upgrade" them by allowing specifications written in other logics to use them.

4.2 Integrating Maude into CASL

The work that needs to be done for such an integration is to prepare both Maude and the logic underlying it so that it can act as an expansion card for HETS. On the side of the semantics, this means that the logic needs to be organized as an *institution* [39, 97]. An institution for rewriting logic was already studied in [77], but it presented the drawback of using a discrete category of signatures, that is, the only morphisms were the identity morphisms. However, we need these morphisms to work with renamings, views, and parameterized modules, and thus this institution is not adequate for our purposes. On the side of the tool, we must provide a parser and static analysis mechanisms for it, in such a way that its specifications can be translated into a common framework, which in our case consists of development graphs, a graphical representation of structured specifications.

Before trying to describe another institution for the logic underlying Maude, we realized that two logics have been studied in the literature for the transition system used in Maude system modules: rewriting logic [56] and preordered algebra [36, 56]. They essentially differ in the treatment of rewrites: whereas in rewriting logic rewrites are named and different rewrites between two given states (terms) can be distinguished (which corresponds to equipping each carrier set with a category of rewrites), in preordered algebra only the existence of a rewrite matters (which corresponds to equipping each carrier set with a preorder of rewritability).

After a careful look at Maude and rewriting logic, we found out that the current Maude implementation differs from rewriting logic as defined in [56]. The reasons are:

1. In Maude, labels of rewrites cannot (and need not) be translated along signature morphisms. This means that Maude views do not lead to theory morphisms in rewriting logic!

2. Although labels of rewrites are used in traces of counterexamples, they play a subsidiary role; for example, they cannot be used in the linear temporal logic of the Maude model checker.

3. Kinds cannot be explicitly declared in Maude, and thus it is impossible to have a kind without sorts.

For these reasons it is not necessary, for the time being, to define an institution for rewriting logic to integrate Maude into Hets, and thus we will use preordered algebra [36, 56]. In this logic, rewrites are neither labeled nor distinguished, only their existence is important, which implies that Maude views lead to theory morphisms in the institution of preordered algebras. We present in the following sections an institution for Maude specifications based on preordered algebras and a comorphism from this institution to the one of Casl, the central logic of Hets. More details are presented in [24, 23].

4.2.1 An institution for Maude

The institution we are going to define for Maude, that we will denote Maude^{pre}, is very similar to the one defined in the context of CafeOBJ [36, 28] for preordered algebra (the differences are mainly limited to the discussion about operation profiles below, but this is only a matter of representation). Signatures of Maude^{pre} are tuples $(K, F, \mathit{kind} : (S, \leq) \to K)$, where K is a set of *kinds*, *kind* is a function assigning a kind to each *sort* in the poset $(S, \leq)$, and F is a set of function symbols of the form $F = \{F_{k_1 \ldots k_n \to k} \mid k_i, k \in K\} \cup \{F_{s_1 \ldots s_n \to s} \mid s_i, s \in S\}$ such that if $f \in F_{s_1 \ldots s_n \to s}$, there is a symbol $f \in F_{\mathit{kind}(s_1) \ldots \mathit{kind}(s_n) \to \mathit{kind}(s)}$. Notice that there is no essential difference between our putting operation profiles on sorts into the signatures and Meseguer's original formulation putting them into the sentences.

Given two signatures $\Sigma_i = (K_i, F_i, \mathit{kind}_i)$, $i \in \{1, 2\}$, a signature morphism $\phi : \Sigma_1 \to \Sigma_2$ consists of a function $\phi^{kind} : K_1 \to K_2$, a function between the sorts $\phi^{sort} : S_1 \to S_2$ such that $\phi^{sort}; \mathit{kind}_2 = \mathit{kind}_1; \phi^{kind}$ and the subsorts are preserved, and a function $\phi^{op} : F_1 \to F_2$ which maps operation symbols compatibly with the types. Moreover, the overloading of symbol names must be preserved, i.e. the name of $\phi^{op}(\sigma)$ must be the same both when mapping the operation symbol σ on sorts and on kinds. With composition defined component-wise, we get the category of signatures.

For a signature Σ, a model M interprets each kind k as a preorder $(M_k, \leq)$, each sort s as a subset M_s of $M_{\mathit{kind}(s)}$ that is equipped with the induced preorder, with M_s a subset of $M_{s'}$ if $s < s'$, each operation symbol $f \in F_{k_1 \ldots k_n, k}$ as a function $M_f : M_{k_1} \times \ldots \times M_{k_n} \to M_k$ which has to be monotonic and such that for each function symbol f on sorts, its interpretation must be a restriction of the interpretation of the corresponding function on kinds. For two Σ-models A and B, a homomorphism of models is a family $\{h_k : A_k \to B_k\}_{k \in K}$ of preorder-preserving functions which is also an algebra homomorphism and such that $h_{kind(s)}(A_s) \subseteq B_s$ for each sort s.

The sentences of a signature Σ are Horn clauses of the form $\forall X \,.\, A \Leftarrow A_1 \wedge \cdots \wedge A_n$, where the set of variables X used for quantification is K-sorted, built with three types of atoms: equational atoms $t = t'$, membership atoms $t : s$, and rewrite atoms $t \Rightarrow t'$, where t, t' are Σ-terms and s is a sort in S.[1]

Given a Σ-model M and a valuation $\eta = \{\eta_k\}_{k \in K}$, i.e., a K-sorted family of functions assigning elements in M to variables, M_t^η is inductively defined as usual. An equational atom $t = t'$ holds in M if $M_t^\eta = M_{t'}^\eta$, a membership atom $t : s$ holds when M_t^η is an element of M_s, and a rewrite atom $t \Rightarrow t'$ holds when $M_t^\eta \leq M_{t'}^\eta$. Satisfaction of atoms is extended to satisfaction of sentences in the obvious way. Finally, we use $M, \eta \models A$ to indicate that the model M satisfies the sentence A under the valuation η.

We prove that the satisfaction condition holds for atoms, and then the extension to Horn clauses is straightforward. To do so, we will use the following lemma:

Lemma 1 *Given a signature morphism $\sigma : \Sigma \to \Sigma'$, inducing the function $\sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ and the functor $\_|_\sigma : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$, a Σ'-model M, the sets of variables $X = \{x_1 : k_1, \ldots, x_l : k_l\}$ and $X' = \{x_1 : \sigma(k_1), \ldots, x_l : \sigma(k_l)\}$, with $k_i \in K$, $1 \leq i \leq l$, a valuation $\eta : X \to M|_\sigma$, which induces a valuation $\eta' : X' \to M$ with $\eta'(x) = \eta(x)$, and a Σ-term t with variables in X, we have $M_{\sigma(t)}^{\eta'} = (M|_\sigma)_t^\eta$.*

Proof. By structural induction on t. For $t = x$ a variable on kinds it is trivial because $\sigma(x) = x$ and $\eta(x) = \eta'(x)$. Similarly, for $t = c$ a constant it is trivial by applying the definition of morphism to operators. If $t = f(t_1, \ldots, t_n)$, then we have $M_{\sigma(t_i)}^{\eta'} = (M|_\sigma)_{t_i}^\eta$, $1 \leq i \leq n$, by induction hypothesis, and

$$\begin{array}{rcll}
M_{\sigma(f(t_1,\ldots,t_n))}^{\eta'} & = & M_{\sigma(f)(\sigma(t_1),\ldots,\sigma(t_n))}^{\eta'} & \text{(by definition of } \sigma \text{ on terms)} \\
& = & M_{\sigma(f)}(M_{\sigma(t_1)}^{\eta'}, \ldots, M_{\sigma(t_n)}^{\eta'}) & \text{(meaning of the term in the model)} \\
& = & M_{\sigma(f)}((M|_\sigma)_{t_1}^\eta, \ldots, (M|_\sigma)_{t_n}^\eta) & \text{(by induction hypothesis)} \\
& = & (M|_\sigma)_f((M|_\sigma)_{t_1}^\eta, \ldots, (M|_\sigma)_{t_n}^\eta) & \text{(by definition of } \sigma \text{ on models)} \\
& = & (M|_\sigma)_{f(t_1,\ldots,t_n)}^\eta. & \text{(meaning of the term in the model)}
\end{array}$$

□

We can use this result, combined with the bijective correspondence between η and η', to check the satisfaction condition for a Σ-equation $t = t'$:

$$\begin{array}{rcl}
M \models_{\Sigma'} \sigma(t = t') & \iff & M \models_{\Sigma'} \sigma(t) = \sigma(t') \\
& \iff & M, \eta' \models_{\Sigma'} \sigma(t) = \sigma(t') \text{ for all } \eta' \\
& \iff & M_{\sigma(t)}^{\eta'} = M_{\sigma(t')}^{\eta'} \text{ for all } \eta' \\
& \iff & (M|_\sigma)_t^\eta = (M|_\sigma)_{t'}^\eta \text{ for all } \eta \\
& \iff & M|_\sigma, \eta \models_\Sigma t = t' \text{ for all } \eta \\
& \iff & M|_\sigma \models_\Sigma t = t'
\end{array}$$

and similarly for memberships and rules.

[1]Note that this is slightly more general than Maude's version, because rewrite conditions are allowed in equations and membership axioms.

4.2.2 A comorphism from Maude to Casl

We now present an encoding of Maude into Casl, which will be formalized as an institution comorphism. The idea of the encoding of *Maude*^{pre} into Casl is that we represent rewriting as a binary predicate and we axiomatize it as a preorder compatible with operations.

First of all, we need to know the Casl institution in order to define the comorphism. We show here the main features of this institution, while more details can be found in [59, 66]. It is introduced in two steps: first, many-sorted partial first-order logic with sort generation constraints and equality ($PCFOL^=$) is introduced, and then, subsorted partial first-order logic with sort generation constraints and equality ($SubPCFOL^=$) is described in terms of $PCFOL^=$ [59]. Basically this institution is composed of:

- A *subsorted signature* $\Sigma = (S, TF, PF, P, \leq_S)$, where S is a set of sorts, TF and PF are two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of total function symbols and partial function symbols, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each $(w, s) \in S^* \times S$, $P = (P_w)_{w \in S^*}$ a family of predicate symbols, and $\leq_S$ is a reflexive and transitive subsort relation on the set S. Given two signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, a signature morphism $\sigma : \Sigma \to \Sigma'$ consists of:

  - a map $\sigma^S : S \to S'$ preserving the subsort relation,
  - a map $\sigma^F_{w,s} : TF_{w,s} \cup PF_{w,s} \to TF_{\sigma^{S^*}(w), \sigma^S(s)} \cup PF'_{\sigma^{S^*}(w), \sigma^S(s)}$ preserving totality (i.e., total function symbols must be mapped into total function symbols and partial function symbols must be mapped into partial function symbols), for each $w \in S^*, s \in S$, and
  - a map $\sigma^P_w : P_w \to P'_{\sigma^{S^*}(w)}$ for each $w \in S^*$.

  Identities and composition are defined in the obvious way.

  With each subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$ we associate a many-sorted signature $\hat{\Sigma}$, which is the extension of (S, TF, PF, P), the underlying many-sorted signature, with:

  - an overloaded total *injection* function symbol $inj : s \to s'$, for each pair of sorts $s \leq_S s'$,
  - an overloaded partial *projection* function symbol $pr : s' \to ?\, s$, for each pair of sorts $s \leq_S s'$, and
  - an overloaded unary *membership* predicate symbol $\in^s : s'$, for each pair of sorts $s \leq_S s'$.

  Signature morphisms $\sigma : \Sigma \to \Sigma'$ are extended to signature morphisms $\hat{\sigma} : \hat{\Sigma} \to \hat{\Sigma}'$ by just mapping the injections, projections, and memberships in $\hat{\Sigma}$ to the corresponding injections, projections, and memberships in $\hat{\Sigma}'$.

  For a subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$, we define *overloading relations* (also called *monotonicity orderings*), $\sim_F$ and $\sim_P$, for function and predicate symbols, respectively:

  Let $f : w_1 \to s_1, f : w_2 \to s_2 \in TF \cup PF$, then $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ iff there exist $w \in S^*$ with $w \leq_{S^*} w_1$ and $w \leq_{S^*} w_2$ and $s \in S$ with $s_1 \leq_S s$ and $s_2 \leq_S s$.

  Let $p : w_1, p : w_2 \in P$, then $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ with $w \leq_{S^*} w_1$ and $w \leq_{S^*} w_2$.

- A set of *subsorted* Σ*-sentences*, that correspond to ordinary $\hat{\Sigma}$-many-sorted sentences, that is, closed many-sorted first-order $\hat{\Sigma}$-formulas or sort generation constraints (a special formula stating which operators are used as constructors) over Σ. Sentence translation along a subsorted signature morphism σ is just sentence translation along the many-sorted signature morphism $\hat{\sigma}$.

- *Subsorted* Σ*-models* M are ordinary many-sorted $\hat{\Sigma}$-models, which are composed of:
  - a non-empty carrier set M_s for each sort $s \in S$,
  - a partial function f_M from M_w to M_s for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, $w \in S^*$, $s \in S$, the function being total if $f \in TF_{w,s}$, and
  - a predicate $p_M \subseteq M_w$ for each predicate symbol $p \in P_w$, $w \in S^*$,

  satisfying the following set of axioms $\hat{J}(\Sigma)$:
  - $\mathit{inj}_{(s,s)}(x) \overset{e}{=} x$ (identity), where $\overset{e}{=}$ stands for existential equation.
  - $\mathit{inj}_{(s,s')}(x) \overset{e}{=} \mathit{inj}_{(s,s')}(x) \implies x \overset{e}{=} y$ for $s \leq_S s'$ (embedding-injectivity),
  - $\mathit{inj}_{(s',s'')}(\mathit{inj}_{s,s'}(x)) \overset{e}{=} \mathit{inj}_{(s,s'')}(x)$ for $s \leq_S s' \leq_S s''$ (transitivity),
  - $\mathit{pr}_{(s',s)}(\mathit{inj}_{(s,s')}(x)) \overset{e}{=} x$ for $s \leq_S s'$ (projection),
  - $\mathit{pr}_{(s',s)}(x) \overset{e}{=} \mathit{pr}_{(s',s)}(y) \implies x \overset{e}{=} y$ for $s \leq_S s'$ (projection-injectivity),
  - $\in^s_{s'}(x) \iff \mathit{pr}_{(s',s)}(x)$ for $s \leq_S s'$ (membership),
  - $\mathit{inj}_{(s',s)}(f_{w',s'}(\mathit{inj}_{s_1,s'_1}(x_1), \ldots, \mathit{inj}_{s_n,s'_n}(x_n))) =$ $\mathit{inj}_{(s'',s)}(f_{w'',s''}(\mathit{inj}_{(s_1,s''_1)}(x_1), \ldots, \mathit{inj}_{(s_n,s''_n)}(x_n)))$ for $f_{w',s'} \sim_F f_{w'',s''}$, where $w \leq w', w''$, $s', s'' \leq s$, $w = s_1, \ldots, s_n$, $w' = s'_1, \ldots, s'_n$, and $w'' = s''_1, \ldots, s''_n$ (function-monotonicity), and
  - $p_{w'}(\mathit{inj}_{(s_1,s'_1)}(x_1), \ldots, \mathit{inj}_{(s_n,s'_n)}(x_n)) \iff p_{w''}(\mathit{inj}_{(s_1,s''_1)}(x_1), \ldots, \mathit{inj}_{(s_n,s''_n)}(x_n))$ for $p_{w'} \sim_P p_{w''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, and $w'' = s''_1 \ldots s''_n$ (predicate-monotonicity).

- *Satisfaction* and the *satisfaction condition* are inherited from the many-sorted institution. Roughly speaking, a formula φ is satisfied in a model M iff it is satisfied w.r.t. all variable valuations into M.

Now, we can define the comorphism. Every Maude signature $(K, F, \mathit{kind} : (S, \leq) \to K)$ is translated to the CASL theory $((S', \leq', F, P), E)$, where S' is the disjoint union of K and S, $\leq'$ extends the relation $\leq$ on sorts with pairs $(s, \mathit{kind}(s))$ for each $s \in S$, $\mathit{rew} \in P_{s,s}$ for any $s \in S'$ is a binary predicate and E contains axioms stating that for any kind k, $\mathit{rew} \in P_{k,k}$ is a preorder compatible with the operations. The latter means that for any $f \in F_{s_1 \ldots s_n, s}$ and any x_i, y_i of sort $s_i \in S'$, $i = 1, \ldots, n$, if $\mathit{rew}(x_i, y_i)$ holds, then $\mathit{rew}(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n))$ also holds.

Let Σ_i, $i \in \{1, 2\}$ be two Maude signatures and let $\varphi : \Sigma_1 \to \Sigma_2$ be a Maude signature morphism. Then its translation $\Phi(\varphi) : \Phi(\Sigma_1) \to \Phi(\Sigma_2)$, denoted ϕ, is defined as follows:

- for each $s \in S$, $\phi(s) = \varphi^{sort}(s)$ and for each $k \in K$, $\phi(k) = \varphi^{kind}(k)$.

- the subsort preservation condition of ϕ follows from the similar condition for φ.

- for each operation symbol σ, $\phi(\sigma) = \varphi^{op}(\sigma)$.

- *rew* is mapped to itself.

The sentence translation map for each signature is obtained in two steps. While the equational atoms are translated as themselves, membership atoms $t : s$ are translated to Casl memberships $t\ in\ s$ and rewrite atoms of the form $t \Rightarrow t'$ are translated as $rew(t, t')$. Then, any sentence in Maude of the form $(\forall x_i : k_i)H \implies C$,[2] where H is a conjunction of Maude atoms and C is an atom, is translated as $(\forall x_i : k_i)H' \implies C'$, where H' and C' are obtained by mapping all the Maude atoms as described before.

Given a Maude signature Σ, a model M' of its translated theory (Σ', E) is mapped to a Σ-model denoted M where:

- for each kind k, define $M_k = M'_k$ and the preorder relation on M_k is rew;
- for each sort s, define M_s to be the image of M'_s under the injection $inj_{s,kind(s)}$ generated by the subsort relation;
- for each f on kinds, let $M_f(x_1, \ldots, x_n) = M'_f(x_1, \ldots, x_n)$ and for each f on sorts of result sort s, let $M_f(x_1, \ldots, x_n) = inj_{s,kind(s)}(M'_f(x_1, \ldots, x_n))$. M_f is monotone because axioms ensure that M'_f is compatible with rew.

The reduct of model homomorphisms is the expected one.

Let Σ be a Maude signature, M', N' be two $\Phi(\Sigma)$-models (in Casl) and let $h' : M' \to N'$ be a model homomorphism. Let us denote $M = \beta_\Sigma(M')$, $N = \beta_\Sigma(N')$ and let us define $h : M \to N$ as follows: for any kind k of Σ, $h_k = h'_k$ (this is correct because the domain and the codomain match, by definition of M and N). We need to show that h is indeed a Maude model homomorphism. For this, we need to show three things:

1. h_k is preorder preserving for any kind k.

   Assume $x \leq^M_k y$. By definition, the preorder on M_k is the one given by rew, so this means $M_{rew}(x, y)$ holds. By the homomorphism condition for h' we have $N_{rew}(h'(x), h'(y))$ holds, which means by definition of the preorder on N that $h'(x) \leq^N_k h'(y)$.

2. h is an algebra homomorphism.

   This follows directly from the definition of M_f where f is an operation symbol and from the homomorphism condition for operation symbols for h'.

3. for any sort s, $h_{kind(s)}(M_s) \subseteq N_s$.

   By definition, $M_s = inj_{s,kind(s)}(M'_s)$. By the homomorphism condition for $inj_{s,kind(s)}$, which is an explicit operation symbol in Casl, we have that

   $$h_{kind(s)}(M_s) = h_{kind(s)}(inj_{s,kind(s)}(M'_s)) = inj_{s,kind(s)}(h_s(M'_s)).$$

   Since $h_s(M'_s) \subseteq N'_s$ by definition, we have that $inj_{s,kind(s)}(h_s(M'_s)) \subseteq inj_{s,kind(s)}(N'_s)$, which by definition is N_s.

4.3 Development graphs

For proof management, Hets uses *development graphs* [62]. They can be defined over an arbitrary institution and are used to encode structured specifications in various phases

[2]The declaration of any variable x of sort s is substituted by a variable x of kind $kind(s)$ and a membership axiom $x : s$.

of the development. Roughly speaking, each node of the graph represents a theory, while links define how theories can make use of other theories. In this way, we represent complex specifications by representing each component (e.g. each module) as a node in the development graph and the relation between them (e.g. importations and proof obligations) as links. We give in this section the intuitions behind development graphs, while the formal definitions can be found in [24].

Definition 13 *A* development graph *is an acyclic, directed graph* $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$*.*

$\mathcal{N}$ *is a set of nodes. Each node* $N \in \mathcal{N}$ *is a pair* (Σ^N, Φ^N) *such that* Σ^N *is a signature and* $\Phi^N \subseteq \mathbf{Sen}(\Sigma^N)$ *is the set of* **local axioms** *of* N*, (e.g. in the Maude case these sentences are the equations, membership axioms, and rules declared—not imported—in the module represented in the node).*

$\mathcal{L}$ *is a set of directed links, so-called* **definition links***, between elements of* $\mathcal{N}$*. We are interested in two kinds of links from a node* M *to a node* N*:*

- **global** *(denoted* $M \xRightarrow{\sigma} N$*), that indicate that the sentences in* M *are included, renamed by* σ*, into the theory in* N *(since all the nodes contain a signature, the renaming* σ *must make the signature in* M *equal to the corresponding (sub)signature in* N*).*
- **free** *(denoted* $M \xRightarrow[free]{\sigma} N$*), that indicate that the signature in* M *is freely included, under the renaming* σ*, into the theory in* N*.*

In addition to these links we add a new one, denoted $M \xRightarrow[n.p.free]{\sigma} N$, that stands for non-persistent free links and will be used when dealing with `protecting` importations in Maude modules. However, these links are only used for Maude specifications and thus are nonstandard; we will show in the next section how these links and the associated nodes are transformed into a new graph that only uses standard constructions. Intuitively, these links indicate that no new elements can be added to the sorts, although they can be added to the kind.

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. The idea is that a theorem link between the nodes M and N, denoted $M =\overset{\sigma}{=}\Rightarrow N$, includes all the sentences in M, under a renaming σ, as *proof obligations* in N.

4.3.1 Freeness constraints

Maude uses initial and free semantics intensively. However, its semantics of freeness is different from the one used in CASL in that the free extensions of models are required to be persistent only on sorts and new error elements can be added on the interpretation of kinds. Attempts to design the translation to CASL in such a way that Maude free links would be translated to usual free definition links in CASL have been unsuccessful, and thus we decided to introduce the non-persistent free definitions links mentioned in the previous section. In order not to break the development graph calculus, we normalize them by replacing them with a semantically equivalent development graph in CASL. The main idea is to make a free extension persistent by duplicating parameter sorts appropriately, such that the parameter is always explicitly included in the free extension. The transformation from Maude non-persistent free links to CASL free links is illustrated in Figure 4.2:

$$\begin{tikzcd}
M \arrow[rr, Rightarrow, "\sigma", "n.p.free"'] \arrow[d, Rightarrow] & & N \arrow[dd, Rightarrow] \\
M' \arrow[d, Rightarrow, "\iota_N"] & & \\
M'' \arrow[r, Rightarrow, "\sigma^{\#}", "free"'] & K \arrow[r, Rightarrow, "\iota_N", "hide"'] & N'
\end{tikzcd}$$

Figure 4.2: Normalization of Maude free links

- M' and N' stand for the translation from Maude to Casl of M and N along the comorphism in Section 4.2.
- M'' is an extension (the morphism ι is a renaming to make the signature distinct from M) of M' where the signature has been extended with sorts $[s]$ for each sort $s \in \Sigma_M$, such that $s \leq [s]$ and $[s] \leq [s']$ if $s \leq s'$; function symbols have been extended with $f : [w] \to [s]$ for each $f : w \to s \in \Sigma_M$; and new *rew* predicates have been added for these sorts.
- The node K has a signature Σ^K that consists of the signature Σ^M disjointly united with the copy of Σ^M generated in the previous step by ι, denoted $\iota(\Sigma_M)$ (let us denote with $\iota(x)$ the corresponding symbol in this copy for each symbol x from the signature Σ^M) and augmented with new operations $h : \iota(s) \to ?\, s$, for any sort s of Σ^M ($\to ?$ indicating it is a partial function) and $make_s : s \to \iota(s)$, for any sort s of the source signature Σ of the morphism σ labelling the free definition link. The axioms for these new function symbols, generated following the guidelines given in [61], are beyond the scope of this chapter and are presented in depth in [24].
- A (Casl) free link joins M'' with K, labeled with a morphism $\sigma^{\#}$ that extends σ with $[s] \mapsto [\sigma(s)]$ for each $s \in \Sigma_M$.
- Finally, a hiding link (basically, a link where the morphism is applied in the opposite direction of the link—from N' to K in this case—and where some symbols, considered hidden, are not included in the target) working as an inclusion connects K and N'.

The generic treatment of freeness constraints in Casl specifications is contained in the axioms of the node K. The generation of these axioms has been implemented while integrating Maude and Hets, so the integration has as side effect the possibility of proving freeness constraints in Casl and all the formalisms connected with it.

4.3.2 Development graph: An example

We show in this section how Maude specifications are translated to development graphs by means of an example, while more details can be found in [23]. To show that development graphs can deal with large specifications, we present the development graph for the Maude prelude in Figure 4.3. Since the graph is too big to distinguish any details, it is worth to zoom in some areas and study them carefully. We focus on lists of natural numbers, as shown in Figure 4.4. The ingredients here are: (i) the modules `BOOL` and `NAT` for Boolean and natural numbers; (ii) the theory `TRIV` requiring the existence of a sort `Elt`; (iii) the parameterized module `LIST` defining generic lists over the elements defined in

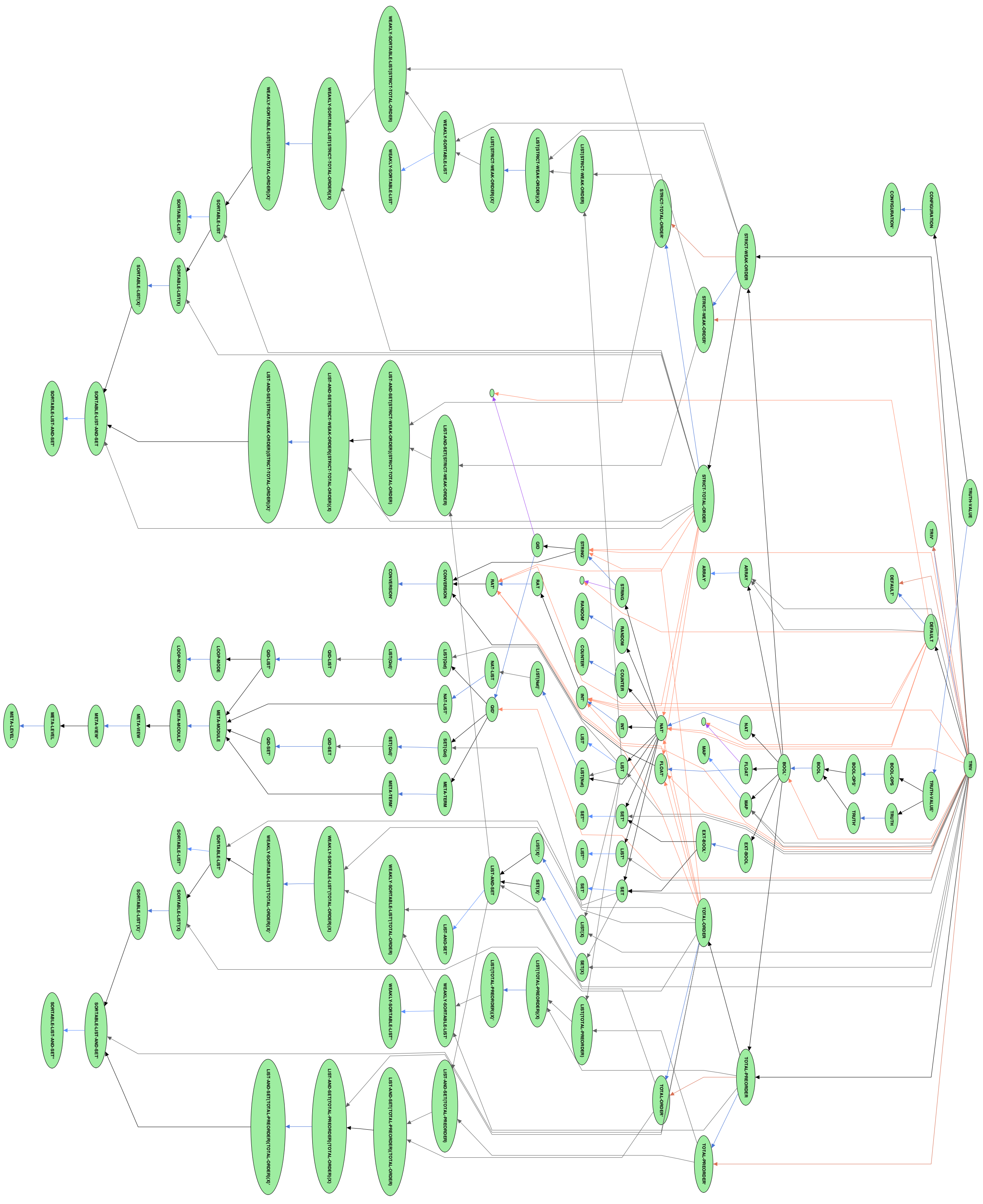


Figure 4.3: Development graph for the Maude prelude

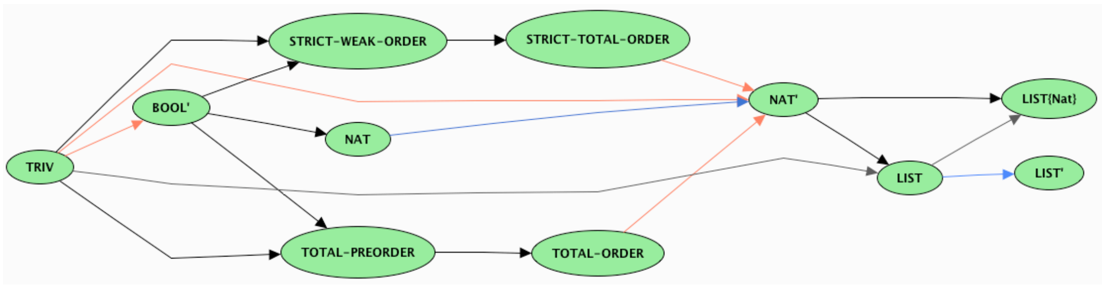


Figure 4.4: Development graph for the predefined lists of natural numbers

TRIV; (iv) two more theories STRICT-TOTAL-ORDER and TOTAL-ORDER, that import some other theories, requiring that elements of sort Elt are a strict total order and a total order, respectively; (v) three views Nat, Nat<, and Nat<= stating that we satisfy the theories above by mapping the sort Elt to the sort Nat of the natural numbers; and (vi) an instantiation of the module LIST with the view Nat. The development graph has:

- One node for each Maude theory (like TOTAL-ORDER or STRICT-TOTAL-ORDER) containing its corresponding theory (signature and sentences).

- Two nodes for each module. One contains the complete theory and stands for the usual loose semantics, and another one, linked to the first one with a free definition link, contains the same signature but no local axioms and stands for the free model of the theory. We identify the former with the name of the module (like nodes NAT or LIST), while the latter is named with this identifier primed (NAT' and LIST'). We have omitted the subgraph for BOOL to simplify the graph.

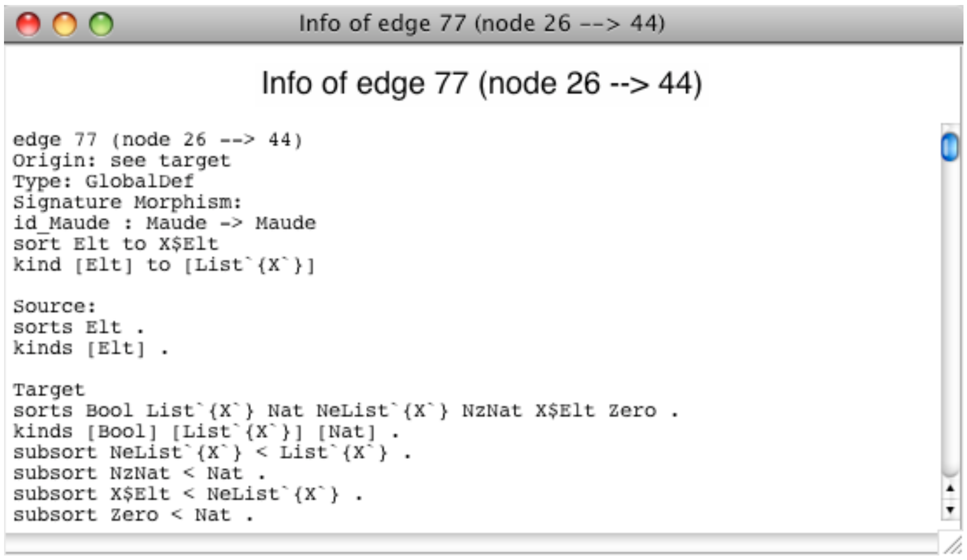
- A global definition link for **including** importations, like TRIV being imported by STRICT-WEAK-ORDER, which is itself imported by STRICT-TOTAL-ORDER.

- A global definition link when a theory is used in a formal parameter, as LIST, that is parameterized by a parameter of the form X :: TRIV. This link is labeled, as shown in Figure 4.5, with the morphism Elt $\mapsto$ X\$Elt (moreover, since the sort Elt is declared a subsort of List{X}, the kind of Elt is mapped to the kind of List{X}), which qualifies the sort with the parameter name.

- A theorem link for each view. These theorem links have as source the node standing for the theory used as source of the view and as target the node standing for the free model of the target module. For example, the view Nat, that has TRIV as source and NAT as target, generates the theorem link between TRIV and NAT'.

- A new node for each instantiation. This node is the target of several global definition links, one from the parameterized module and another one from the node standing for the free model of the target of each view used as parameter. For example, LIST{Nat} instantiates the module LIST with the view Nat. It is interesting to see that the link from LIST to LIST{Nat} is labeled, as shown in Figure 4.6, with the morphism X\$Elt $\mapsto$ Nat, List{X} $\mapsto$ List{Nat}, NeList{X} $\mapsto$ NeList{Nat}, which indicates that the sort X\$Elt has been mapped to Nat and the parameter has been instantiated with the view Nat.

The main point of using development graphs, in addition to graphically represent Maude specifications, is to ease heterogeneous verification. A calculus of development



Info of edge 77 (node 26 --> 44)

```
edge 77 (node 26 --> 44)
Origin: see target
Type: GlobalDef
Signature Morphism:
id_Maude : Maude -> Maude
sort Elt to X$Elt
kind [Elt] to [List`{X`}]

Source:
sorts Elt .
kinds [Elt] .

Target
sorts Bool List`{X`} Nat NeList`{X`} NzNat X$Elt Zero .
kinds [Bool] [List`{X`}] [Nat] .
subsort NeList`{X`} < List`{X`} .
subsort NzNat < Nat .
subsort X$Elt < NeList`{X`} .
subsort Zero < Nat .
```

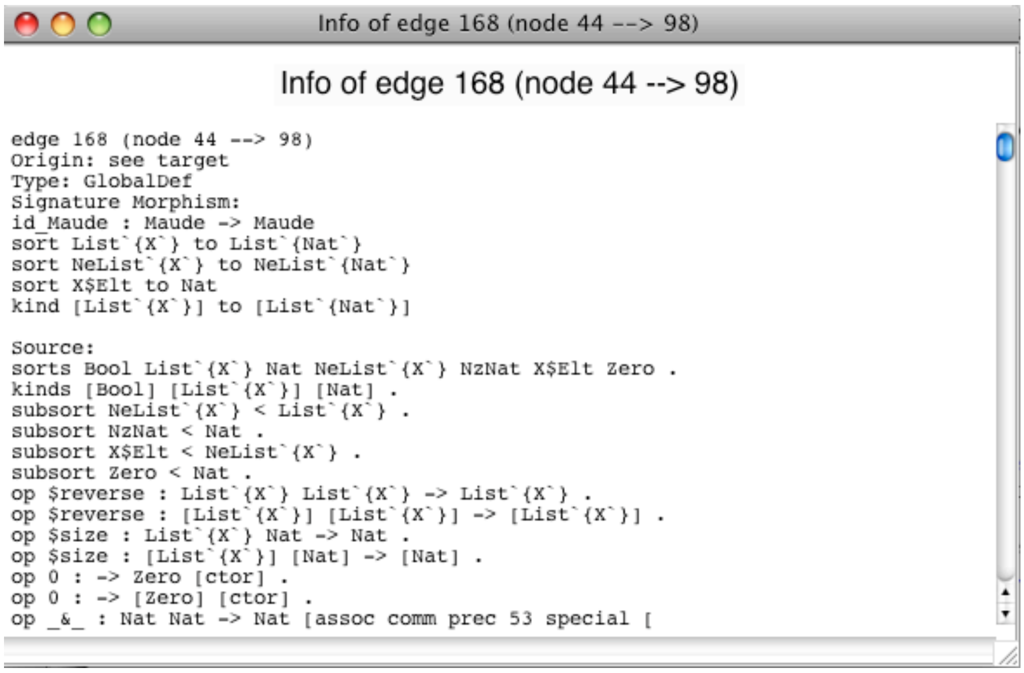
Figure 4.5: Morphism from `TRIV` to `LIST`

graphs is integrated into HETS in such a way that development graphs are transformed to deal with proof obligations. In the current case, we are interested in the `Automatic` transformation available in the `Edit/Proofs` menu: among other transformations, it "pushes" the proof obligations in theorem links to the target nodes, in order to prove them there. The result of applying this command to the previous development graph is shown in Figure 4.7, where the node `NAT'`, displayed in red by HETS, indicates that it has pending proof obligations. We can use the `Prove` option available in the node menu, that opens the window in Figure 4.8, where the different axioms (either in the theory or to be proved) can be displayed, and the theorem provers can be selected. In our case, the axioms are related to those equations stating that natural numbers conform a strict total order (transitive, irreflexive, incomparability-transitive, and total properties) and a total order (reflexive, transitive, total, and antisymmetric properties). These properties can be automatically proved with SPASS, which allows HETS to turn the `NAT'` node from red to green, shown in Figure 4.9, which indicates that all proof obligations have been discarded. Actually, note that these properties cannot be directly proved over the Maude natural numbers, because all the operations are implemented in C++ and thus the module `NAT` lacks equations. The proofs have been performed by using a CASL library that defines some of the Maude predefined modules similarly to the expected Maude specifications,[3] which is automatically loaded when these modules are used. For example, the definition of the operator `_<=_` in this library is:

```
forall m,n : Nat
     . 0 <= n = maudeTrue                          %(leq_def1_Nat)%
     . suc(n) <= 0 = maudeFalse                    %(leq_def2_Nat)%
     . suc(m) <= suc(n) = m <= n                   %(leq_def3_Nat)%
```

where the `maudeTrue`, `maudeFalse` values stand for a renaming of the Maude values `true` and `false` in order to prevent clashes with the CASL constants with the same name, and the strings on the right are just labels for the equations. Of course, this proof works

[3]An example of heterogeneous system! It combines the implementation of HETS in Haskell, a parser for Maude specifications written in Maude itself, and CASL libraries.



Info of edge 168 (node 44 --> 98)

```
edge 168 (node 44 --> 98)
Origin: see target
Type: GlobalDef
Signature Morphism:
id_Maude : Maude -> Maude
sort List`{X`} to List`{Nat`}
sort NeList`{X`} to NeList`{Nat`}
sort X$Elt to Nat
kind [List`{X`}] to [List`{Nat`}]

Source:
sorts Bool List`{X`} Nat NeList`{X`} NzNat X$Elt Zero .
kinds [Bool] [List`{X`}] [Nat] .
subsort NeList`{X`} < List`{X`} .
subsort NzNat < Nat .
subsort X$Elt < NeList`{X`} .
subsort Zero < Nat .
op $reverse : List`{X`} List`{X`} -> List`{X`} .
op $reverse : [List`{X`}] [List`{X`}] -> [List`{X`}] .
op $size : List`{X`} Nat -> Nat .
op $size : [List`{X`}] [Nat] -> [Nat] .
op 0 : -> Zero [ctor] .
op 0 : -> [Zero] [ctor] .
op _&_ : Nat Nat -> Nat [assoc comm prec 53 special [
```

Figure 4.6: Morphism from `LIST` to `LIST{Nat}`

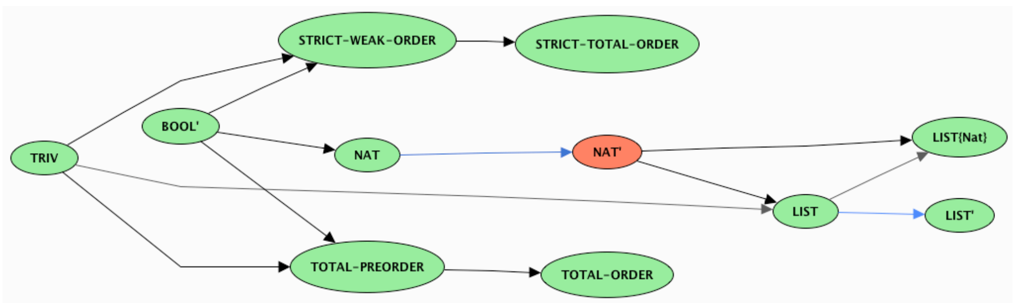


Figure 4.7: Development graph after using `Automatic`

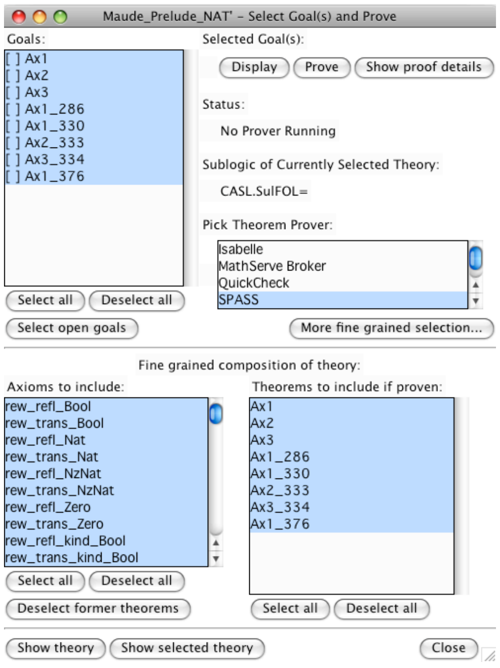
in the same way if we define the corresponding equations in Maude and prove the proof obligations in the node generated by that module.

Unfortunately, not all the proof obligations required by Maude specifications can be automatically discharged. In [24] we proved that reversing a list twice returns the original list. To do it we used the freeness transformations sketched in Section 4.3.1 to normalize the development graph and then we applied the standard transformations over it to discharge the proof obligations.

4.4 Implementation

We briefly describe here the implementation steps required to integrate Maude into HETS:

Abstract syntax. First, the abstract syntax for Maude specifications must be defined in Haskell. This abstract syntax is based on the Maude grammar presented in [20,



Maude_Prelude_NAT' – Select Goal(s) and Prove

Goals:
[ ] Ax1
[ ] Ax2
[ ] Ax3
[ ] Ax1_286
[ ] Ax1_330
[ ] Ax2_333
[ ] Ax3_334
[ ] Ax1_376

Selected Goal(s):

Status:
No Prover Running

Sublogic of Currently Selected Theory:
CASL.SulFOL=

Pick Theorem Prover:
Isabelle
MathServe Broker
QuickCheck
SPASS

Fine grained composition of theory:

Axioms to include:
rew_refl_Bool
rew_trans_Bool
rew_refl_Nat
rew_trans_Nat
rew_refl_NzNat
rew_trans_NzNat
rew_refl_Zero
rew_trans_Zero
rew_refl_kind_Bool
rew_trans_kind_Bool

Theorems to include if proven:
Ax1
Ax2
Ax3
Ax1_286
Ax1_330
Ax2_333
Ax3_334
Ax1_376

Figure 4.8: Window for proving proof obligations

Chapter 24].

Maude parsing. We must also describe how the Maude specifications introduced in HETS are parsed in order to obtain a term following the abstract syntax above. We are able to implement this parsing in Maude itself thanks to Maude's metalevel [20, Chapter 14], a module that allows the programmer to use Maude entities such as modules, equations, or rules as usual data by efficiently implementing the *reflective* capabilities of rewriting logic [21].

By using this feature we have developed a function that receives a module and returns a list of quoted identifiers standing for an object following the abstract syntax, that can be read by Haskell because the data types derive the class `Read`.

Logic. Once the Maude modules have been translated into their abstract syntax, we must implement the Haskell type classes `Language` and `Logic` provided by HETS, that define the types needed to represent each logic as an institution and the comorphisms between them. To do so, we relate each element in these classes (signature, sentences, morphisms, and so on) with the corresponding element in the abstract syntax, following the theory described in the previous sections. More details about this step can also be found in [50], whose main focus is the implementation of this step.

Development graph. Given the options received from the command line and the path of the Maude file to be parsed we must compute in Haskell the associated development graph. In fact, we build two different development graphs, the first one containing the modules used in the Maude prelude and another one with the user specification, following in both cases the ideas described in the previous sections.

Comorphism. Given a Maude signature and a list of Maude sentences, we must implement in Haskell the translation to the corresponding CASL signature and CASL

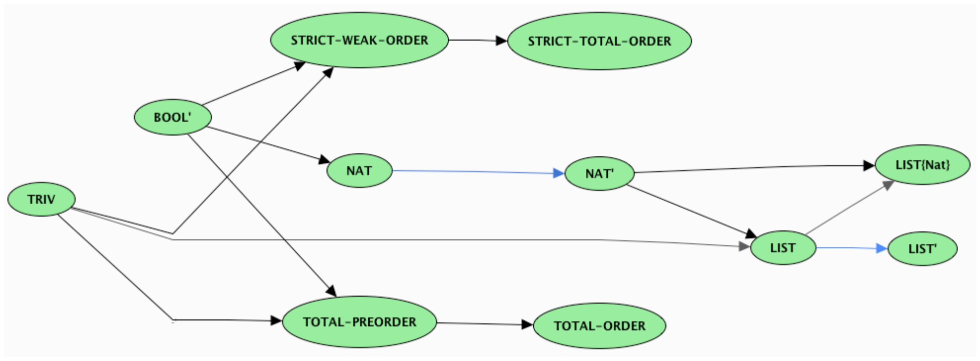


Figure 4.9: Development graph after discarding the proof obligations

sentences.

Freeness constraints. Finally, we have implemented in Haskell how, given a CASL signature and a set of CASL sentences, the freeness constraints described in Section 4.3.1 are generated.

4.5 Contributions

In this chapter we have presented an integration of Maude into HETS. The main contributions of this work are:

Institution and comorphism. An institution for Maude has been defined. This institution is based on preordered algebras, since an institution for rewriting logic is not necessary for Maude for the time being, and uses as sentences equations, memberships, and rules, differently from the proposal in [17] that only used (unconditional) rules, and allows nontrivial signature morphisms, unlike the institution in [77], which used a discrete category of signatures. We have also defined a comorphism from this institution to the institution of CASL, the central logic of HETS, which combines first-order logic and induction.

Development graph. The modules, theories, views, and structuring mechanisms of Maude have been incorporated to development graphs. In this way, we can now represent Maude specifications as development graphs, where modules and theories are represented as nodes and importations and views as links between these nodes.

Freeness constraints. We have developed, at the CASL level, a transformation that allows us to prove freeness constraints for those specifications that use the standard freeness links available in HETS and have a translation to CASL. However, this is not the case of Maude, which requires a specific treatment of freeness. For this reason, a new kind of link has been introduced in the calculus of development graphs, that is later normalized into the usual freeness link in order to use the transformation explained before.

Proofs. As a "corollary" of the contributions above, Maude has been integrated into HETS, which allows to prove properties over Maude specifications with the provers already integrated within HETS.

Chapter 5

Concluding Remarks and Future Work

The contributions of this thesis are:

- We have presented a state-of-the-art declarative debugger for Maude specifications. This debugger allows the user to debug both wrong and missing answers caused by wrong and missing statements and wrong search conditions. A great effort has been devoted to improving the usability of this tool, providing techniques to shorten and improve the debugging tree, different navigation strategies, several different answers, trusting mechanisms, a graphical user interface, and an original feature: different debugging trees can be built depending on the complexity of the specification and the knowledge of the user. Finally, since the debugging trees are built following a formal calculus, we have proved the soundness and completeness of the technique.

- We have integrated Maude into HETS, which allows to use HETS tools, and particularly its theorem provers, with Maude specifications. This integration has been achieved by (i) defining an institution for Maude and (ii) translating the structuring mechanisms of Maude into development graphs. Moreover, we have also implemented a transformation that allows the user to prove freeness constraints in general theories (not necessarily in Maude specifications).

The work presented in this thesis offers a good basis for potential extensions and enrichment that can improve its usability and generality. As future work for our debugger, we plan to add new navigation strategies like the ones shown in [94] that take into account the number of different potential errors in the subtrees, instead of their size. Moreover, the current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session. We intend to add a new command to introduce *complete* modules (that is, we would let the system know that all the correct inferences that can be done in the specification being debugged can also be done in this complete module), which would greatly reduce the number of questions asked to the user.

We are currently working on a test generator for Maude specifications that, combined with the debugger, will allow the user to test and debug specifications with the same tool. A first step in this project has been the development of a test-case generator for Maude functional modules [84], which is able to generate test cases for these modules and check their correctness with respect to a given specification or select a subset of these test cases to be checked by the user by using different strategies; since these processes are very expensive we also present different trusting techniques to ease them. We are currently working to improve the performance of this test-case generator, applying techniques as narrowing and using distributed architectures, and to extend it to test Maude system modules; since these modules are not required to be either terminating or confluent, the test cases generated for this kind of modules are completely different from the ones for functional modules.

On the Hets side, since interactive proofs are not easy to conduct, future work will move in the direction of making proving more efficient by adopting automated induction strategies like rippling [29]. Moreover, we intend to use the automatic first-order prover SPASS for induction proofs by integrating special induction strategies directly into Hets.

We are also studying the possible comorphisms from Casl to Maude. We distinguish whether the formulas in the source theory are confluent and terminating or not. In the first case, that we plan to check with the Maude termination [30] and confluence [31] checkers, we map formulas to equations, whose execution in Maude is more efficient, while in the second case we map formulas to rules.

Finally, we also plan to relate Hets' Modal Logic and Maude models in order to use the Maude model checker [20, Chapter 13] for linear temporal logic.

Part II

Resumen de la Investigación

Capítulo 6

Introducción

Esta tesis empezó en Illinois hace alrededor de cuatro años, aunque nosotros aún no lo supiésemos, cuando Narciso Martí, Francisco Durán y Santiago Escobar introdujeron inadvertidamente un error en la especificación de la función de Fibonacci mientras escribían un capítulo del libro de Maude:

$$\begin{array}{rcl} \mathit{fib}(0) & = & 0 \\ \mathit{fib}(1) & = & 1 \\ \mathit{fib}(n) & = & \mathit{fib}(n-1) + \mathit{fib}(n-2) \quad \text{si } n > 1 \end{array}$$

En vez de definir la función como se muestra arriba, la suma del tercer caso fue escrita como una multiplicación, haciendo que la especificación fuese incorrecta. Por insignificante que parezca, llevó mucho tiempo solucionar este error, lo que resultó en un proyecto para desarrollar un depurador declarativo para especificaciones en Maude.

En efecto, programar es un proceso en el que suelen aparecer errores que necesitan ser corregidos para obtener el comportamiento pretendido por el usuario. Por esta razón se han desarrollado diversos mecanismos para encontrar los errores introducidos por el programador, dando lugar a una técnica llamada *depuración*. Supongamos que implementamos la función de Fibonacci en Maude, incluyendo el error antes mencionado:[1]

```
op fib : Nat -> Nat .
eq fib(0) = 0 .
eq fib(1) = 1 .
ceq fib(n) = fib(n - 1) * fib(n - 2) if n > 1 .
```

La técnica de depuración mejor conocida es el uso de *puntos de ruptura*. Un punto de ruptura simplemente indica una pausa en la ejecución de un programa cuando se alcanza cierto punto previamente indicado por el programador. En el programa anterior podríamos colocar un punto de ruptura en la última ecuación y, una vez alcanzado este, continuar con la ejecución del programa paso a paso para comprobar el resultado de `n - 1`, de `n - 2`, de la aplicación de `fib` a los correspondientes resultados, y el resultado final de la función. De esta manera obtendríamos los valores de las llamadas recursivas y el resultado de la función y podríamos compararlos con los esperados. Siguiendo esta estrategia, muchos lenguajes de programación también proporcionan trazas como medio para depurar programas. Este método muestra cada paso al usuario (donde la definición de paso depende del lenguaje de programación y normalmente puede ser personalizado por el usuario) y su resultado.

[1]Aunque no hayamos explicado aún la sintaxis de Maude, consideramos que esta definición puede entenderse fácilmente.

$$\dfrac{\dfrac{}{\texttt{2 > 1}} \quad \dfrac{\dfrac{}{\texttt{2 - 1} \to \texttt{1}} \quad \dfrac{}{\texttt{fib(1)} \to \texttt{1}}}{\texttt{fib(2 - 1)} \to \texttt{1}} \quad \dfrac{\dfrac{}{\texttt{2 - 2} \to \texttt{0}} \quad \dfrac{}{\texttt{fib(0)} \to \texttt{0}}}{\texttt{fib(2 - 2)} \to \texttt{0}} \quad \dfrac{}{\texttt{1 * 0} \to \texttt{0}}}{\texttt{fib(2)} \to \texttt{0}}$$

Figura 6.1: Árbol de ejecución para el ejemplo de *Fibonacci*

No es casualidad que, dado que el paradigma de programación más usado es el *imperativo*, estas técnicas estén especialmente diseñadas para programas de este tipo, ya que en ellas cada instrucción se ejecuta en un orden previamente fijado por el usuario (las instrucciones de control como los condicionales o los bucles se pueden seguir fácilmente una vez se saben los valores de las variables), pero estas premisas no se dan en los lenguajes declarativos. La separación entre la lógica (*qué* se espera que el programa calcule) y el control (*cómo* se espera que los cálculos se lleven a cabo) es una de las principales ventajas de estos lenguajes, pero también supone una complicación cuando se trata de depurar resultados erróneos. De hecho, la complejidad de los mecanismos de control dificulta la aplicación de las técnicas de depuración paso a paso utilizadas en lenguajes imperativos.

La *depuración declarativa* (también conocida como *depuración algorítmica* o *diagnosis abstracta*) es una técnica de depuración que abstrae los detalles de ejecución y se centra en los resultados, lo que resulta muy adecuado para depurar lenguajes declarativos. Es importante notar que la palabra *declarativa* en "depuración declarativa" se refiere a esta abstracción de los detalles de ejecución (como decíamos antes, *qué* se computa) y no a su aplicación a lenguajes declarativos, es decir, la depuración declarativa puede ser aplicada (y lo ha sido, como veremos en el capítulo 8) a lenguajes imperativos. Este tipo de depuración consta de dos fases: en la primera, se construye una estructura de datos representando el cómputo (el *árbol de depuración*), donde el resultado concluido en cada nodo debe ser consecuencia de los resultados en los nodos hijos. Este árbol se suele construir usando un cálculo formal que permite al diseñador probar la corrección y la completitud de la técnica. En la segunda fase se recorre esta estructura siguiendo una *estrategia de navegación* y haciendo preguntas a un oráculo (normalmente el usuario) hasta que se encuentra el error.

Volviendo al ejemplo de Fibonacci, supongamos que evaluamos `fib(2)`. El resultado devuelto por nuestra función debería ser `0`, porque la condición se cumple (`2` es mayor que `1`), `2 - 1` es `1`, `fib(1)` se evalúa a `1`, `2 - 2` es `0`, `fib(0)` se evalúa a `0` (y estos resultados son los esperados) y el producto de estos resultados es `0`. Es decir, el árbol de depuración debería tener la forma mostrada en la figura 6.1, donde la idea básica es que los resultados en cada nodo son consecuencia de los resultados en sus hijos. La depuración declarativa procedería ahora preguntando a un oráculo (el programador en este caso, aunque algunas veces se pueden usar otros, como una especificación correcta) sobre la corrección de los nodos con respecto al comportamiento que el programador tenía en mente mientras implementaba el sistema, la *semántica pretendida* del sistema. El objetivo de la navegación es encontrar un nodo incorrecto (con respecto a esta semántica pretendida) con todos sus hijos correctos (con respecto a esta semántica pretendida): el *nodo defectuoso* (*buggy node* en inglés), que en este caso es la raíz del árbol. En general, los nodos del árbol están etiquetados para permitir al usuario identificar el error; este etiquetado depende del lenguaje de programación y del cálculo utilizado para crear el árbol de depuración. En el caso de Maude distinguimos cada ecuación por medio de etiquetas, y por tanto en este caso el depurador señalaría a la última ecuación como defectuosa (en la figura 6.1 las inferencias en todos los nodos excepto en la raíz, que intuitivamente está asociada a la última ecuación, son correctas).

Como hemos visto, el proceso de depuración empieza con un *síntoma inicial* que revela que el programa es incorrecto, como la evaluación de `fib(2)` a `0` en el ejemplo anterior. En nuestro esquema distinguimos dos tipos de respuestas (o resultados) obtenidos por el sistema dando lugar a estos síntomas iniciales: *respuestas erróneas*, que son resultados incorrectos obtenidos a partir de un valor de entrada válida (como en el ejemplo anterior); y *respuestas perdidas*, que son resultados incompletos obtenidos a partir de un valor de entrada válida (por ejemplo, obtener `1 + fib(0)` al evaluar `fib(2)`, lo cual es correcto pero no es el resultado final esperado). Aunque ambos tipos de errores pueden ser depurados usando depuración declarativa, la depuración de respuestas perdidas ha sido mucho menos estudiada porque el cálculo utilizado es mucho más complejo en general que el usado para respuestas erróneas y, además, suele relacionarse en general con sistemas no deterministas, una característica asociada a los lenguajes declarativos.

6.1 Depuración declarativa para Maude

En esta tesis presentamos un depurador declarativo para especificaciones en Maude. *Maude* [20] es un lenguaje declarativo basado en lógica ecuacional y lógica de reescritura para la especificación e implementación de una amplia gama de modelos y sistemas. Los módulos funcionales de Maude son especificaciones en *lógica ecuacional de pertenencia*, y permiten al usuario definir tipos de datos y operaciones sobre ellos mediante teorías en esta lógica que proporcionan múltiples tipos (*sorts* en inglés), relaciones de subtipado, ecuaciones y aserciones de pertenencia a un tipo. Los módulos de sistema de Maude son especificaciones en lógica de reescritura que, además de los elementos presentes en los módulos funcionales, permiten definir reglas de reescritura que representan transiciones entre estados. Por el momento solo estamos interesados en el hecho de que las ecuaciones y los axiomas de pertenencia deben ser terminantes y confluentes, mientras que las reglas pueden ser no terminantes y no confluentes. De ahora en adelante, llamaremos *reducción* a la evaluación de un término usando ecuaciones y *reescritura* a la evaluación de un término usando reglas y ecuaciones (quizás ninguna).

Es decir, las especificaciones en Maude se componen de ecuaciones $t_1 = t_2$, que se deben entender como t_1 y t_2 son iguales, axiomas de pertenencia $t : s$, estableciendo que t tiene tipo s, y reglas de reescritura $t_1 \Rightarrow t_2$, que indican que t_1 se reescribe a t_2. Con estas premisas, podemos identificar fácilmente qué tipo de respuestas erróneas pueden aparecer en Maude: t_2 se obtiene a partir de t_1 usando ecuaciones o reglas pero t_2 no debería ser alcanzable desde t_1, y el tipo s ha sido inferido para t pero el término no tiene ese tipo. Es un poco más complicado definir las respuestas perdidas: dado que las ecuaciones deben ser terminantes y confluentes, el usuario espera obtener, dado un término inicial, un único término al cual no se le pueden aplicar más ecuaciones (es decir, una *forma normal*); por tanto, si obtiene un término alcanzable pero que no está en forma normal (como $1+\mathit{fib}(0)$), entonces es una respuesta perdida. Además, los tipos en Maude están ordenados (por una relación de subconjunto) y por tanto el usuario espera que los términos tengan un tipo mínimo único; una respuesta perdida es aquella que infiere como tipo mínimo de un término un tipo correcto pero que no es el menor. Por último, las respuestas perdidas en módulos de sistema son más fáciles de caracterizar: dado que las reglas no se espera que sean confluentes, un término puede reescribirse en general a un conjunto de términos; si este conjunto es menor que el esperado por el usuario, entonces hemos encontrado una respuesta perdida.

Para depurar estos errores hemos desarrollado un cálculo que nos permite inferir todos los síntomas descritos, es decir, reducciones, inferencias de tipos, reescrituras, formas normales, tipos mínimos y conjuntos de términos alcanzables dadas ciertas condiciones, las cuales se corresponden con las impuestas por Maude en su comando `search`. Usando este cálculo podemos construir árboles de prueba que pueden ser usados como árboles de depuración para el proceso de depuración declarativa. Usando estos árboles somos capaces de detectar diversas causas que pueden provocar los errores: ecuaciones, axiomas de pertenencia, reglas y condiciones (usadas para calcular el conjunto de términos alcanzables) erróneas, y ecuaciones, axiomas de pertenencia y reglas perdidas (es decir, sentencias—*statement* en inglés—que deberían ser parte de la especificación y que sin embargo no lo son). Aunque estos árboles permitirían al usuario depurar sus especificaciones, hemos desarrollado una técnica de poda que reduce y simplifica las preguntas hechas al oráculo, tratando de esta manera los principales problemas de la depuración declarativa: el número y la complejidad de las preguntas. Además, esta técnica permite al usuario construir diferentes árboles de depuración dependiendo de la complejidad de la especificación y de su conocimiento sobre ella, lo que constituye una aportación original de este trabajo. Esta técnica, que hemos llamado *Árbol de Prueba Abreviado* (*APT* por sus siglas en inglés), descarta aquellos nodos cuya corrección se pueda inferir de la corrección de sus hijos, y modifica otros para realizar preguntas que simulan el comportamiento de Maude y que, por tanto, deberían ser más fáciles de contestar. Además de esta técnica para mejorar los árboles de prueba, también permitimos al usuario confiar en algunas sentencias, términos y módulos. Por último, proporcionamos una interfaz gráfica de usuario para facilitar la interacción entre los usuarios y la herramienta.

Este depurador declarativo se ha desarrollado en varios pasos, empezando con la funcionalidad mínima y añadiendo progresivamente más funciones. Este desarrollo se ve reflejado en nuestras publicaciones en esta materia:

- Empezamos tratando respuestas erróneas (que, como hemos comentado anteriormente, tienen un cálculo asociado más sencillo que las respuestas perdidas) en módulos funcionales (los cuales son un subconjunto de los módulos de sistema). La función *APT* aplicada a los árboles calculados en esta fase inicial generaba un único árbol de depuración, en el cual se habían podado ciertos nodos mientras otros se habían modificado para simplificar las preguntas asociadas. Esta versión ya permitía confiar en sentencias y en módulos completos, e incluso usar un módulo correcto como oráculo. Este trabajo se presentó en [13, 14].

- La extensión natural de esta herramienta consistía en añadir la capacidad de depurar respuestas erróneas en módulos de sistema. Además de definir un nuevo cálculo y adaptar las funcionalidades ya existentes a este tipo de depuración, la transformación *APT* para esta versión del depurador permitía al usuario generar dos árboles de depuración diferentes, dependiendo de la complejidad de la aplicación. Este trabajo fue presentado en [86].

- Una descripción del sistema completo para depurar respuestas erróneas en cualquier especificación en Maude se presentó en [90].

- Una vez completada la depuración de respuestas erróneas, el siguiente paso "natural" era depurar respuestas perdidas en módulos funcionales. El cálculo usado en este tipo de depuración extendía el cálculo anterior, aunque era mucho más complejo: mientras el anterior solo indicaba qué estaba ocurriendo, este nuevo cálculo también indicaba qué *no* estaba ocurriendo. Llamamos a estos tipos de información, respectivamente, información *positiva* y *negativa*. Además de este nuevo cálculo, añadimos un nuevo mecanismo de confianza, permitiendo al usuario indicar las *formas normales*. Este trabajo se publicó en [88].

- Mejorando la herramienta, desarrollamos un depurador de respuestas tanto erróneas como perdidas para cualquier especificación en Maude. El cálculo en esta fase usaba las ideas ya presentadas para depurar respuestas perdidas en módulos funcionales: teníamos en cuenta información positiva y negativa. La función APT permitía al usuario construir dos nuevos tipos de árboles (los cuales son independientes de los otros tipos de árboles, siendo cuatro el número de posibles combinaciones) y se añadió un nuevo mecanismo de confianza: selección de tipos y operadores *finales*. Este trabajo se presentó en [87].
- Una descripción de la herramienta, centrándose en las funciones que no se describieron en [90], se presentó en [89].
- Una descripción completa e integrada del sistema, incluyendo todos los teoremas y sus demostraciones, fue publicada en [91].

6.2 Integrando Maude en Hets para demostrar la corrección de nuestras especificaciones

Una vez hemos arreglado todos los errores encontrados en nuestros programas, ¿podemos afirmar que son correctos? La mejor respuesta que podemos dar es *son correctos mientras no encontremos nuevos problemas*. No obstante, esta no es una respuesta muy satisfactoria; lo que querríamos es *demostrar* que nuestros programas cumplen ciertas propiedades como vivacidad (*liveness* en inglés) o que satisfacen ciertas fórmulas de primer orden. Los sistemas formales como Maude proporcionan ciertos métodos que permiten al usuario realizar algunos análisis, como el comprobador de modelos para lógica lineal temporal o la comprobación de invariantes mediante búsqueda, pero es improbable que un solo sistema puede proporcionar alguna vez todas las herramientas necesarias para cada posible análisis que puede precisar un programador. Además es natural, cuando se diseñan sistemas grandes, especificar cada una de sus partes de diferentes formas. Una pregunta interesante que surge en estos sistemas es: *¿cómo interactúan estas distintas partes entre si?*

Para solucionar todos estos problemas se emplean especificaciones heterogéneas, que permiten al programador usar diferentes formalismos y que están ganando importancia en la actualidad, especialmente en áreas donde la seguridad es crítica y no se puede correr el riesgo de sufrir un error. Algunas de las aproximaciones heterogéneas actuales se mantienen deliberadamente informales, como UML. En general, muchos de estos sistemas tienen la desventaja de no ser formales o de ser unilaterales, en el sentido de proporcionar solo una lógica (y un único demostrador de teoremas) el cual sirve como dispositivo central de integración, incluso aunque esta lógica central pueda no necesitarse o desearse para aplicaciones particulares.

El *Conjunto Heterogéneo de Herramientas* (Hets por sus siglas en inglés) [61, 64, 65] es una herramienta de integración flexible, formal (es decir, basada en una semántica matemática) y multilateral que proporciona análisis sintáctico y estático y herramientas de demostración para especificaciones heterogéneas al combinar varios lenguajes de especificación individuales.

Hets está basado en un grafo de lógicas y lenguajes, sus herramientas y sus traducciones. Esto proporciona una semántica clara para las especificaciones heterogéneas y el correspondiente cálculo de demostraciones. Para gestionar las demostraciones utiliza el cálculo de los grafos de desarrollo [62]. Este cálculo, conocido por otros sistemas de gestión de demostraciones a gran escala como MAYA [5], representa las especificaciones usando nodos para cada unidad del programa (por ejemplo módulos) y aristas para las relaciones entre ellos (por ejemplo relaciones de importación y obligaciones de prueba), proporcionando una visión de conjunto de la jerarquía de módulos de la especificación heterogénea y del estado actual de las demostraciones, y puede por tanto ser usado para monitorizar la corrección del sistema. Para facilitar la especificación heterogénea HETS proporciona el lenguaje de especificación heterogéneo HetCASL. Este lenguaje está basado en el Lenguaje Común de Especificación Algebraica (CASL por sus siglas en inglés) [66], un lenguaje basado en lógica de primer orden que funciona como lógica central de HETS.

En esta tesis se describe cómo hemos integrado Maude en HETS, lo que permite usar las herramientas ya integradas en HETS (especialmente sus demostradores de teoremas) con especificaciones en Maude. Para lograr esta integración hemos tenido que (i) definir una institución para Maude, y un comorfismo desde esta institución a la de CASL (explicaremos qué son las instituciones y los comorfismos en el capítulo 7; por ahora, consideremos simplemente que una institución es una manera de formalizar una lógica y un comorfismo una traducción entre instituciones), (ii) definir cómo las especificaciones en Maude se traducen a grafos de desarrollo y (iii) implementar mecanismos (al nivel de CASL, es decir, accesibles a todas las lógicas conectadas con CASL) para manejar restricciones de extensiones libres (*freeness constraints* en inglés), que aparecen al usar un tipo concreto de importación en Maude que se explicará en el próximo capítulo. Este trabajo se publicó en [24]; información más detallada se puede encontrar en [23].

6.3 Resumen

Esta tesis se divide en tres partes. Esta parte presenta el resumen de la investigación en español, siguiendo el siguiente esquema:

- El capítulo 7 introduce las nociones básicas necesarias para comprender el resto de la tesis: la sección 7.1 presenta la lógica de reescritura y Maude, mientras que la sección 7.2 introduce las instituciones y los comorfismos.
- El capítulo 8 presenta el depurador declarativo. Mostramos el cálculo unificado que permite depurar tanto respuestas erróneas como perdidas, la función APT y los mecanismos de confianza. Por último, presentamos brevemente cómo usar la interfaz gráfica.
- El capítulo 9 muestra cómo verificar especificaciones heterogéneas. Se presenta HETS, sus mecanismos de representación de especificaciones estructuradas (los grafos de desarrollo) y sus técnicas para trabajar con diferentes lógicas, permitiendo al usuario demostrar propiedades de especificaciones en Maude con otras herramientas.
- El capítulo 10 concluye y presenta futuras líneas de investigación.

La parte I presenta el resumen de la investigación en inglés, siguiendo la misma estructura que esta parte. Por último, la parte III presenta las publicaciones relacionadas con la tesis como fueron originalmente publicadas.

Capítulo 7

Preliminares

7.1 Maude

Como se dijo en la introducción, Maude es un lenguaje declarativo basado tanto en lógica ecuacional de pertenecia como en lógica de reescritura. En esta sección explicamos estas lógicas y los módulos de Maude usados para representarlas. Mucha más información se puede encontrar en el libro de Maude [20].

7.1.1 Lógica ecuacional de pertenencia

Una *signatura* en lógica ecuacional de pertenencia es una terna (K, Σ, S) (Σ de ahora en adelante), con K un conjunto de *familias* (*kind* en inglés), $\Sigma = \{\Sigma_{k_1 \dots k_n, k}\}_{(k_1 \dots k_n, k) \in K^* \times K}$ una signatura heterogénea (*many-kinded* en inglés) y $S = \{S_k\}_{k \in K}$ una colección de conjuntos de tipos, disjuntos dos a dos, indexados por K. La familia de un tipo s se denota como $[s]$. Escribimos $T_{\Sigma,k}$ y $T_{\Sigma,k}(X)$ para referirnos, respectivamente, al conjunto de Σ-términos cerrados (*ground* en inglés) con familia k y de Σ-términos con familia k con variables en X, donde $X = \{x_1 : k_1, \dots, x_n : k_n\}$ es un conjunto de variables sobre K. Intuitivamente, los términos con una familia pero sin tipo representan elementos indefinidos o de error.

Las fórmulas atómicas de la lógica ecuacional de pertenencia son *ecuaciones* $t = t'$, donde t y t' son Σ-términos con la misma familia, y *sentencias de pertenencia* de la forma $t : s$, donde el término t tiene familia k y $s \in S_k$. Las *sentencias* (*sentence* en inglés) son cláusulas de Horn universalmente cuantificadas de la forma $(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$, donde cada A_i puede ser tanto una ecuación como una sentencia de pertenencia, y X es un conjunto de variables sobre la familia K que contiene todas las variables en los correspondientes A_i. Una *especificación* es un par (Σ, E), donde E es un conjunto de sentencias en lógica ecuacional de pertenencia sobre la signatura Σ.

Los modelos de la lógica ecuacional de pertenencia son Σ-*álgebras* $\mathcal{A}$ compuestas por un conjunto soporte A_k para cada familia $k \in K$, una función $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ para cada operador $f \in \Sigma_{k_1 \dots k_n, k}$ y un subconjunto $A_s \subseteq A_k$ para cada tipo $s \in S_k$. Dada un Σ-álgebra $\mathcal{A}$ y una valuación $\sigma : X \longrightarrow \mathcal{A}$ asignando valores en el álgebra a las variables, el significado $[\![t]\!]^{\sigma}_{\mathcal{A}}$ de un término t se define de forma inductiva de la manera habitual. De esta manera, un álgebra $\mathcal{A}$ satisface, dada una valuación σ,

- una ecuación $t = t'$, escrito $\mathcal{A}, \sigma \models t = t'$, si y solo si ambos términos tienen el mismo significado: $[\![t]\!]^{\sigma}_{\mathcal{A}} = [\![t']\!]^{\sigma}_{\mathcal{A}}$; también se dice que la ecuación se cumple en el álgebra bajo la valuación.

- una sentencia de pertenencia $t : s$, escrito $\mathcal{A}, \sigma \models t : s$, si y solo si $[\![t]\!]^{\sigma}_{\mathcal{A}} \in A_s$.

La satisfacción de cláusulas de Horn se define de la manera estándar. Cuando una fórmula ϕ es satisfecha por todas las valuaciones, escribimos $\mathcal{A} \models \phi$ y decimos que $\mathcal{A}$ es un modelo de ϕ. Por último, cuando los términos son cerrados, las valuaciones no juegan ningún papel y se pueden omitir. Una especificación en lógica ecuacional de pertenencia (Σ, E) tiene un modelo inicial $\mathcal{T}_{\Sigma/E}$ cuyos elementos son clases de equivalencia sobre E de términos cerrados $[t]_E$, y donde una ecuación o una sentencia de pertenencia se satisfacen si y solo si se pueden deducir en E mediante un conjunto correcto y completo de reglas de deducción [8, 57].

7.1.2 Módulos funcionales de Maude

Los módulos funcionales de Maude [20, capítulo 4], con sintaxis `fmod ... endfm`, son especificaciones ejecutables en lógica ecuacional de pertenencia y su semántica viene dada por la correspondiente álgebra inicial en la clase de álgebras que satisfacen la especificación.

En un módulo funcional podemos declarar tipos (usando la palabra clave `sort`); relaciones de subtipado entre tipos (`subsort`); operadores (`op`) para construir valores de estos tipos, dados los tipos de sus argumentos y del resultado, y que pueden tener atributos como, por ejemplo, ser asociativo (`assoc`) o conmutativo (`comm`); sentencias de pertenencia (`mb`) declarando que un término tiene un tipo; y ecuaciones (`eq`) identificando términos. Tanto las sentencias de pertenencia como las ecuaciones pueden ser condicionales (`cmb` y `ceq`). Las condiciones, además de condiciones de pertenencia y ecuaciones, pueden ser *ecuaciones de ajuste de patrones* $t := t'$, cuyo significado matemático es el mismo que el de una ecuación ordinaria $t = t'$ pero que operacionalmente se resuelven ajustando el lado derecho t' con el patrón en el lado izquierdo t, instanciando de esta manera las variables nuevas en t.

Maude realiza inferencia de tipos de manera automática para los tipos declarados por el usuario y sus correspondientes relaciones de subtipado. Por tanto, las familias *no* se declaran explícitamente, sino que se corresponden con las componentes conexas de la relación de subtipado. La familia asociada a un tipo `s` se denota con `[s]`. Por ejemplo, si tenemos los tipos `Nat` para los números naturales y `NzNat` para los naturales distintos de cero con la relación `NzNat < Nat`, entonces `[NzNat]` = `[Nat]`. Una declaración de operadores como

```
op _div_ : Nat NzNat -> Nat .
```

se entiende como una declaración al nivel de la familia

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

junto a la correspondiente sentencia de pertenencia

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

Una declaración de subtipado `NzNat < Nat` se entiende como una sentencia de pertenencia condicional

```
cmb N : Nat if N : NzNat .
```

Los módulos funcionales deben satisfacer los requisitos de ejecutabilidad de ser confluentes, terminantes y con tipos decrecientes [20]. En este contexto, las ecuaciones $t = t'$ se pueden orientar de izquierda a derecha, $t \to t'$, y las condiciones ecuacionales $u = v$ pueden comprobarse encontrando un término común t tal que $u \to t$ y $v \to t$; la notación que usaremos en las reglas de inferencia y en los árboles de depuración de la sección 8.2 para esta situación es $u \downarrow v$.

7.1.3 Lógica de reescritura

La lógica de reescritura extiende la lógica ecuacional introduciendo la noción de *reescrituras*, que se corresponden con transiciones entre estados; es decir, mientras que las ecuaciones se interpretan como igualdades y por tanto son simétricas, las reescrituras denotan cambios que pueden ser irreversibles.

Una especificación en lógica de reescritura, o *teoría de reescritura*, tiene la forma $\mathcal{R} = (\Sigma, E, R)$,[1] donde (Σ, E) es una especificación ecuacional y R es un conjunto de reglas etiquetadas de la manera que describimos a continuación. Con esta definición se muestra que la lógica de reescritura se construye sobre la lógica ecuacional, y por tanto la lógica de reescritura está parametrizada con respecto a la version de esta lógica ecuacional subyacente; en nuestro caso, Maude usa lógica ecuacional de pertenencia, descrita en las secciones anteriores. Una regla condicional q en R tiene la forma[2]

$$q : \; (\forall X)\; t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$$

donde q es la etiqueta de la regla, la cabecera es una reescritura y las condiciones pueden ser ecuaciones, condiciones de pertenencia y reescrituras; ambos lados de la reescritura deben tener la misma familia. Con estas reglas de reescritura se pueden deducir reescrituras de la forma $t \Rightarrow t'$ usando las reglas de deducción presentadas en [56] (para más información remitimos a [10]).

Los modelos de las teorías de reescritura son llamados $\mathcal{R}$-*sistemas* en [56]. Estos sistemas se definen como categorías con una estructura de (Σ, E)-álgebra, junto a una transformación natural para cada regla del conjunto R. De manera intuitiva, la idea es que tenemos una (Σ, E)-álgebra, como las descritas en la sección 7.1.1, con transiciones entre los elementos de cada conjunto A_k; además, estas transiciones deben satisfacer ciertos requisitos: deben existir transiciones identidad para cada elemento, las transiciones se deben poder componer secuencialmente, las operaciones en la signatura Σ están definidas apropiadamente para las transiciones y tenemos suficientes transiciones correspondientes a con las reglas en R. Las reglas de deducción de la lógica de reescritura propuestas en [56] son correctas y completas con respecto a esta noción de modelo. Además, se pueden usar para construir modelos iniciales. Dada una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, el modelo inicial $\mathcal{T}_{\Sigma/E,R}$ para $\mathcal{R}$ tiene una (Σ, E)-álgebra subyacente $\mathcal{T}_{\Sigma/E}$ cuyos elementos son clases de equivalencia $[t]_E$ de Σ-términos cerrados módulo E, y hay una transición de $[t]_E$ a $[t']_E$ si existen términos t_1 y t_2 tales que $t =_E t_1 \Rightarrow_R^* t_2 =_E t'$, donde $t_1 \Rightarrow_R^* t_2$ significa que el término t_1 puede reescribirse a t_2 en cero o más pasos aplicándole reglas en R, también escrito como $[t]_E \Rightarrow_{R/E}^* [t']_E$ cuando la reescritura se considera en clases de equivalencia [56, 27].

7.1.4 Módulos de sistema en Maude

Los módulos de sistema en Maude [20, capítulo 6], con sintaxis `mod ... endm`, son teorías de reescritura ejecutables y su semántica viene dada por el correspondiente sistema inicial en la clase de sistemas correspondiente a la teoría de reescritura. Un módulo de

[1]No tratamos aquí la formulación más compleja de lógica de reescritura con argumentos congelados; para más información nos remitimos a [20].

[2]No es necesario que la condición use en primer lugar ecuaciones, después sentencias de pertenencia y por último reescrituras; aunque usemos esta notación para abreviar la definición, las condiciones pueden aparecer en cualquier orden.

sistema puede tener todas las declaraciones de un módulo funcional y, además, declarar reglas (`rl`) y reglas condicionales (`crl`), cuyas condiciones pueden ser ecuaciones, ecuaciones de ajuste de patrones, condiciones de pertenencia y reescrituras.

Los requisitos de ejecutabilidad de las ecuaciones y las sentencias de pertenencia en un módulo de sistema son los mismos que en los módulos funcionales. Con respecto a las reglas, la satisfacción de las condiciones en una regla de reescritura se intenta secuencialmente de izquierda a derecha, resolviendo las condiciones de reescritura por medio de búsquedas; por esta razón, podemos tener nuevas variables en estas condiciones, que se instanciarán a lo largo del proceso de resolución (más detalles están disponibles en [20]). Además, la estrategia seguida por Maude al reescribir mediante reglas es calcular la forma normal del término con respecto a las ecuaciones antes de aplicar una regla. Está garantizado que esta estrategia no omite ninguna posible reescritura cuando las reglas son *coherentes* con respecto a las ecuaciones [101, 20]. De una manera similar a la confluencia, la coherencia requiere que, dado un término t, para cada reescritura de t usando una regla en R a algún término t', si u es la forma normal de t con respecto a la parte ecuacional E, entonces hay una reescritura de u con ciertas reglas en R a un término u' tal que $u' =_E t'$.

7.1.5 Características avanzadas

Además de los módulos considerados hasta ahora, presentamos algunas otras características de Maude que se usarán a lo largo de este trabajo.

Modos de importación

Los módulos de Maude pueden importar otros módulos en tres modos diferentes:

- El modo `protecting` (`pr` de manera abreviada) indica que no se puede añadir basura ni confusión a los tipos importados, es decir, que no se introduzcan nuevos elementos ni se identifiquen elementos que en el módulo importado eran diferentes.

- El modo `extending` (`ex` de manera abreviada) indica que se pueden añadir nuevos elementos pero no identificar elementos diferentes.

- El modo `including` (`inc` de manera abreviada) permite tanto añadir nuevos elementos como identificar elementos diferentes.

Teorías

Las teorías se usan para declarar interfaces, es decir, las propiedades tanto sintácticas como semánticas que deben ser satisfechas por los módulos usados para instanciar los parámetros. Análogamente a los módulos, Maude proporciona dos tipos de teorías: teorías funcionales y teorías de sistema, con la misma estructura que los módulos homólogos, pero con semántica laxa (a diferencia de los módulos, que tienen semántica inicial). Las teorías funcionales se declaran con las palabras clave `fth ... endfth`, mientras que las teorías de sistema usan `th ... endth`. Ambas pueden tener tipos, relaciones de subtipado, operadores, variables, sentencias de pertenencia y ecuaciones, y pueden importar otras teorías y módulos. Las teorías de sistema también pueden tener reglas. A diferencia de los módulos, las teorías no necesitan satisfacer ningún requisito de ejecutabilidad.

Vistas

Usamos vistas para especificar cómo un módulo o teoría destino satisface una cierta teoría fuente. En general, puede haber muchas maneras en las que los requisitos se ven satisfechos por el módulo o teoría destino; es decir, puede haber muchas vistas, cada una especificando una interpretación particular de la teoría fuente en el destino. En la definición de una vista debemos indicar su nombre, la teoría fuente, el módulo o teoría destino, y cómo asociar cada tipo y cada operador en la fuente con los correspondientes en el destino. Tanto la fuente como el destino de una vista puede ser cualquier expresión de módulos, con la expresión de módulo fuente evaluándose a una teoría y la destino tanto a una teoría como a un módulo. Cada declaración de vista tiene asociada un conjunto de obligaciones de prueba, esto es, por cada axioma en la teoría fuente se debe cumplir que la traducción del axioma por la vista sigue siendo cierto en el destino. Dado que el destino puede ser un módulo interpretado con semántica inicial, verificar estas obligaciones de prueba puede necesitar en general pruebas por inducción. Estas obligaciones no son descartadas por el sistema. Un hecho importante sobre las vistas es que no pueden asociar etiquetas, y por tanto no podemos identificar sentencias en la fuente con las correspondientes sentencias en el destino.

Módulos parametrizados

Los módulos de Maude pueden estar parametrizados. Un módulo parametrizado tiene la siguiente sintaxis:

$$\texttt{mod M\{}X_1 :: T_1, \ldots, X_n :: T_n\texttt{\} is ... endm}$$

donde $n \geq 1$. Los módulos funcionales parametrizados tienen una sintaxis análoga.

Llamamos interfaz a la parte $\{X_1 :: T_1, \ldots, X_n :: T_n\}$, donde cada par $X_i :: T_i$ es un parámetro, cada X_i es un identificador (el nombre o etiqueta del parámetro) y cada T_i es una expresión que identifica a una teoría (la teoría parámetro). El nombre de cada parámetro en un interfaz debe ser único, aunque no hay restricciones de unicidad en las teorías parámetro de un módulo. Las teorías parámetro de un módulo funcional deben ser teorías funcionales.

En un módulo parametrizado M, todos los tipos y las etiquetas que vengan de teorías en el interfaz se cualifican con sus nombres. Por tanto, dado un parámetro $X_i :: T_i$, cada tipo S en T_i debe cualificarse como $X_i\$S$, y cada etiqueta l en T_i se debe cualificar como $X_i\$l$. De hecho, el módulo parametrizado M se aplana de la siguiente manera. Para cada parámetro $X_i :: T_i$ se incluye una copia renombrada de la teoría T_i, llamada $X_i :: T_i$. El renombramiento lleva cada tipo S a $X_i\$S$ y cada etiqueta l a $X_i\$l$. El renombramiento se propaga por inclusiones anidadas de teorías, pero no afecta a las importaciones de módulos [20]. Así, si T_i incluye una teoría T', cuando la teoría renombrada $X_i :: T_i$ se crea y se incluye en M, la teoría renombrada $X_i :: T'$ también se creará y se añadirá en $X_i :: T_i$. Sin embargo, el renombramiento no tendrá efecto en modulos importados por T_i o T'; por ejemplo, si `BOOL` es importado por alguna de estas teorías no será renombrado, sino importado sin cambios en M. Además, los tipos declarados en los módulos parametrizados pueden estar también parametrizados, duplicando, omitiendo o reordenando parámetros.

Los argumentos de los módulos parametrizados se ligan a los parámetros formales mediante una *instanciación*. La instanciación requiere una vista por cada parámetro formal al correspondiente argumento. Cada una de estas vistas se usa para ligar los nombres de tipos, operadores, etc., en los parámetros formales a los correspondientes tipos, operadores (o expresiones), etc., en el argumento destino. La instanciación de un módulo parametrizado debe hacerse con vistas definidas previamente.

7.1.6 Condiciones y sustituciones

A lo largo de esta tesis, y especialmente al usar el cálculo de la Sección 8.2, solo vamos a considerar un tipo especial de condiciones y sustituciones que operan sobre ellas, llamadas *admisibles*, y que se corresponden con las usadas por Maude. Se definen de la siguiente manera:

Definición 1 *Una condición* $\mathcal{C} \equiv C_1 \wedge \cdots \wedge C_n$ *es* admisible *si, para* $1 \leq i \leq n$,

- C_i *es una ecuación* $u_i = u'_i$ *o una condición de pertenencia* $u_i : s$ *y*
$$vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \ o$$
- C_i *es una condición de ajuste de patrones* $u_i := u'_i$*,* u_i *es un patrón y*
$$vars(u'_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \ o$$
- C_i *es una condición de reescritura* $u_i \Rightarrow u'_i$*,* u'_i *es un patrón y*
$$vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j).$$

Definición 2 *Una condición* $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$*, donde* $\circledast$ *representa una variable especial que no aparece en el resto de la condición, es* admisible *si* $P := t \wedge C_1 \wedge \cdots \wedge C_n$ *es admisible para cualquier término cerrado* t*.*

Definición 3 *Una* sustitución familiar*, denotada por* κ*, es una función de la forma* $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ *que asigna términos a las variables de tal manera que* $\forall_{1 \leq i \leq n} . kind(v_i) = kind(t_i)$*, es decir, cada variable tiene la misma familia que el término asociado.*

Definición 4 *Una* sustitución*, denotada por* θ*, es una función de variables a términos de la forma* $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ *tal que* $\forall_{1 \leq i \leq n} . sort(v_i) \geq ls(t_i)$*, es decir, el tipo de cada variable es mayor o igual que el tipo mínimo del término asociado. Nótese que una sustitución es un tipo especial de sustitución familiar en la que cada término tiene el tipo apropiado a su variable.*[3]

Definición 5 *Dada una condición atómica* C*, decimos que una sustitución* θ *es* admisible para C *si*

- C *es una ecuación* $u = u'$ *o una condición de pertenencia* $u : s$ *y* $vars(C) \subseteq dom(\theta)$*, o*
- C *es una condición de ajuste de patrones* $u := u'$ *y* $vars(u') \subseteq dom(\theta)$*, o*
- C *es una condición de reescritura* $u \Rightarrow u'$ *y* $vars(u) \subseteq dom(\theta)$*.*

[3]Las sustituciones familiares y las sustituciones son diferentes de las sustituciones definidas en otros artículos como [8]. En ellos, solo las familias se tienen en cuenta en las sustituciones, mientras que aquí cada variable tiene un tipo asociado, aunque algunas propiedades solo se comprueben a nivel de familias.

7.1.7 Un ejemplo de Maude: Unas rebajas

Ilustramos en esta sección la funcionalidad explicada anteriormente con un ejemplo.[4] Nótese que los módulos, teorías y vistas a continuación están escritos entre paréntesis; usamos esta notación porque los estamos introduciendo en Full Maude, una extensión de Maude usada por el depurador que incluye funcionalidad para analizar sintácticamente, evaluar e imprimir de manera amigable términos y para mejorar la entrada/salida. Primero, vamos a especificar listas ordenadas con un módulo parametrizado. Este módulo usa la teoría `ORD`, que precisa un tipo `Elt` y un operador `_<_` definido sobre los elementos de este tipo que cumpla las propiedades de los órdenes estrictos totales:

```
(fth ORD is
  pr BOOL .

  sort Elt .
  op _<_ : Elt Elt -> Bool .

  vars X Y Z : Elt .
  eq [irreflexive] : X < X = false [nonexec] .
  ceq [transitive] : X < Z = true if X < Y = true /\ Y < Z = true [nonexec] .
  ceq [antisymmetric] : X = Y if X < Y = true /\ Y < X = true [nonexec] .
  ceq [total] : X = Y if X < Y = false /\ Y < X = false [nonexec] .
endfth)
```

donde el atributo `nonexec` indica que estas ecuaciones no puede ser usadas para reducir términos. Una vez esta teoría está especificada, podemos usarla en el módulo `OLIST` para crear listas ordenadas genéricas. El tipo `List{X}` representa listas cualesquiera, mientras `OList{X}` se refiere a listas ordenadas. Dado que las listas ordenadas son un caso particular de lista, usamos una declaración de subtipo para indicarlo:

```
(fmod OLIST{X :: ORD} is
  sorts List{X} OList{X} .
  subsort OList{X} < List{X} .
```

La lista vacía, que también es una lista ordenada, se representa con el operador `nil`, mientras que listas mayores se construyen con el operador de yuxtaposición añadiendo un elemento al principio de la lista:

```
  op nil : -> OList{X} [ctor] .
  op __ : X$Elt List{X} -> List{X} [ctor] .
```

Usamos sentencias de pertenencia para definir cuándo una lista no vacía está ordenada. Primero, indicamos que el primer elemento de la lista debe ser menor o igual que el que va después que él y que el resto de la lista debe estar ordenada. Es interesante ver cómo se usa el operador `_<_` de la teoría para comparar elementos:

```
  vars E E' : X$Elt .
  var  OL : OList{X} .
  vars L L' : List{X} .

  cmb [ol1] : E (E' OL) : OList{X}
   if E < E' or E == E' /\
      E' OL : OList{X} .
```

[4] **¡Atención!** Aunque los módulos a continuación son sintácticamente correctos, hemos cometido un error intencionadamente. En el siguiente capítulo veremos cómo encontrarlo.

También debemos indicar que las listas unitarias están ordenadas:

```
  mb [ol2] : E nil : OList{X} .
```

La función `ordIns` ordena una lista a base de insertar sus elementos de manera ordenada en una nueva lista, usando para ello la función auxiliar `insertOrd`:

```
  op ordIns : List{X} -> OList{X} .
  eq [oi1] : ordIns(nil) = nil .
  eq [oi2] : ordIns(E OL) = insertOrd(ordIns(OL), E) .
```

Esta función inserta el elemento en la posición apropiada. Cuando es menor que el primer elemento de la lista lo coloca en esta posición; en otro caso, continúa el recorrido de la lista:

```
  op insertOrd : OList{X} X$Elt -> OList{X} .
  eq [io1] : insertOrd(nil, E) = E nil .
  ceq [io2] : insertOrd(E L, E') = E' (E L)
   if E' < E .
  eq [io3] : insertOrd(E L, E') = E insertOrd(L, E) [owise] .
 endfm)
```

donde el atributo `owise` indica que la ecuación se aplica *en otro caso* (*otherwise* en inglés), es decir, esta ecuación solo se aplica si ninguna otra puede aplicarse.

En nuestra implementación estamos interesados en usar listas de personas, que se definen en el módulo `PERSON` a continuación. Una persona, de tipo `Person`, se construye con una cadena para su nombre, un número natural indicando cuánto dinero tiene y otra cadena que representa los artículos que lleva:

```
(fmod PERSON is
  pr STRING .

  sort Person .

  op [_,_,_] : String Nat String -> Person [ctor] .
```

Además, dado que queremos instanciar la teoría `ORD` con personas, tenemos que especificar una función booleana sobre personas que satisfaga los requisitos para `_<_`. Hemos implementado esta función de tal manera que, dadas dos personas, la menor de ellas es la que tiene más dinero. De esta manera, conseguiremos que la lista ordenada tenga a las personas con más dinero en primer lugar:

```
  vars NAME1 NAME2 B1 B2 : String .
  vars M1 M2 : Nat .

  op _<_ : Person Person -> Bool .
  eq [NAME1, M1, B1] < [NAME2, M2, B2] = M1 > M2 .
endfm)
```

La teoría y el módulo se relacionan por medio de la vista `PersonOrd`, que asigna al tipo `Elt` de la teoría el tipo `Person` en el módulo. Dado que la operación `_<_` tiene el mismo nombre en la teoría y en el módulo, no es necesario hacer explícita esa correspondencia:

```
(view PersonOrd from ORD to PERSON is
  sort Elt to Person .
endv)
```

El módulo `SALE` especifica una tienda y una lista de personas esperando para entrar y comprar artículos. Este módulo instancia el módulo `OLIST` con la vista anterior y renombra las listas de `OList{PersonOrd}` a `OList` para facilitar su uso:

```
(mod SALE is
  pr OLIST{PersonOrd} * (sort OList{PersonOrd} to OList) .
```

Los artículos a la venta se identifican mediante una cadena con el nombre y un número natural con su precio:

```
  sort Item .
  op <_`,_> : String Nat -> Item [ctor] .
```

Un elemento de tipo `Shop` es simplemente una "sopa" asociativa y conmutativa de personas y artículos. Por ello, tanto el tipo `Person` como `Item` se declaran subtipos de `Shop`. El operador `empty` representa la tienda vacía, mientras que tiendas mayores se construyen con el operador de yuxtaposición, el cual es asociativo y conmutativo y tiene `empty` como elemento identidad:

```
  sort Shop .
  subsorts Item Person < Shop .
  op empty : -> Shop [ctor] .
  op __ : Shop Shop -> Shop [ctor assoc comm id: empty] .
```

Los elementos de tipo `Sale` se construyen con una tienda limitada por corchetes y una lista ordenada de personas fuera de la tienda:

```
  sort Sale .
  op [_]_ : Shop OList -> Sale [ctor] .
```

Ahora debemos especificar las reglas que describen el comportamiento de las personas y de los artículos. La regla `in` se encarga de hacer entrar a la gente en la tienda:

```
  var  SH : Shop .
  var  P : Person .
  var  OL : OList .
  vars TN PN B : String .
  vars C M : Nat .
  var  S : Sale .

  rl [in] : [ SH ] P OL
  =>        [ SH P ] OL .
```

La regla `buy` elimina un artículo de la sopa y lo añade a las pertenencias de una persona, decrementando su dinero (suponiendo que tenga bastante):

```
  crl [buy] : [ SH < TN, C > [PN, M, B] ] OL
   =>         [ SH [PN, sd(M, C), B + " " + TN] ] OL
   if M >= C .
```

donde `sd` es la diferencia simétrica. También definimos una función que comprueba si una cierta persona ha comprado un artículo concreto:

```
  op _buys_in_ : String String Sale -> Bool .
  eq [buy1] : PN buys TN in [ [PN, M, B] SH ] OL = find(B, TN, 0) =/= notFound .
  eq [buy2] : PN buys TN in S = false [owise] .
```

donde `find` es una función predefinida que busca la subcadena `TN` en `B`, empezando en la posición `0`.

Además, definimos algunas constantes que nos servirán para probar el funcionamiento de la especificación. Usamos dos personas, `adri` y su gemelo maligno, `et`, y una lechuga `l` y un videojuego `v` como artículos:

```
  ops adri et : -> Person .
  eq adri = ["adri", 15, ""] .
  eq et = ["et", 16, ""] .

  ops l v : -> Item .
  eq l = < "lettuce", 6 > .
  eq v = < "videogame", 10 > .
endm)
```

Ahora podemos usar el comando `search`, que realiza una búsqueda en anchura, para encontrar las distintas maneras en las que `adri` puede comprar el videojuego:

```
Maude> (search [l v] ordIns(et (adri nil)) =>* S:Sale
        s.t. "adri" buys "videogame" in S:Sale .)
search in SALE :[l v]ordIns(et adri nil) =>* S:Sale .

No solution.
```

Al ejecutar la búsqueda descubrimos que es imposible para `adri` comprarlo. Dado que sabemos que debería ser capaz de comprar el videojuego de alguna manera, llegamos a la conclusión de que la especificación contiene un error. En la sección 8.4 explicaremos cómo encontrarlo.

7.2 Instituciones y comorfismos

En este sección presentamos varias nociones que serán necesarias en el capítulo 9. Antes de describir las instituciones, recordamos las nociones de *categoría*, *categoría dual*, *categoría pequeña*, *funtor* y *transformación natural* [80].

Definición 6 *Una* categoría $\mathbf{C}$ *se compone de:*

- *una clase* $|\mathbf{C}|$ *de* objetos*;*
- *una clase* $hom(\mathbf{C})$ *de* morfismos *(también llamados* flechas*), entre los objetos;*
- *operaciones asignando a cada morfismo f un objeto $dom(f)$, su dominio, y un objeto $cod(f)$, su codominio (escribimos $f : A \to B$ para indicar que $dom(f) = A$ y $cod(f) = B$);*

- *un operador de composición asignando a cada par de morfismos f y g, con $cod(f) = dom(g)$, un morfismo* compuesto $g \circ f : dom(f) \to cod(g)$, *que sea asociativo: para cualesquiera morfismos $f : A \to B$, $g : B \to C$, y $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$; y*
- *para cada objeto A, un morfismo identidad $id_A : A \to A$ que satisfaga la identidad: para cualquier morfismo $f : A \to B$, $id_B \circ f = f$ y $f \circ id_A = f$.*

Definición 7 *Para cada categoría* $\mathbf{C}$, *su* categoría dual $\mathbf{C}^{op}$ *es la categoría que tiene los mismos objetos que* $\mathbf{C}$ *y cuyos morfismos son los opuestos a los morfismos en* $\mathbf{C}$, *es decir, si $f : A \to B$ en* $\mathbf{C}$, *entonces $f : B \to A$ en* $\mathbf{C}^{op}$. *Los morfismos identidad y compuestos se definen de manera obvia.*

Definición 8 *Una categoría* $\mathbf{C}$ *se llama* pequeña *si tanto* $|\mathbf{C}|$ *como* $hom(\mathbf{C})$ *son conjuntos y no clases propias.*

Definición 9 *Sean* $\mathbf{C}$ *y* $\mathbf{D}$ *categorías. Un* funtor $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ *es una función que asigna a cada objecto A en* $\mathbf{C}$ *un objecto* $\mathbf{F}(A)$ *en* $\mathbf{D}$ *y a cada morfismo $f : A \to B$ en* $\mathbf{C}$ *un morfismo* $\mathbf{F}(f) : \mathbf{F}(A) \to \mathbf{F}(B)$ *en* $\mathbf{D}$ *tal que, para todo objeto A en* $\mathbf{C}$ *y cualesquiera morfismos componibles f y g en* $\mathbf{C}$*:*

- $\mathbf{F}(id_A) = id_{\mathbf{F}(A)}$,
- $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ \mathbf{F}(f)$.

Definición 10 *Una* transformación natural $\eta : \mathbf{F} \to \mathbf{G}$ *entre los funtores* $\mathbf{F}, \mathbf{G} : \mathbf{A} \to \mathbf{B}$ *asocia a cada $X \in |\mathbf{A}|$ un morfismo $\eta_X : \mathbf{F}(X) \to \mathbf{G}(X)$ en* $\mathbf{D}$ *llamado la* componente *de η en X, tal que para cada morfismo $f : X \to Y \in \mathbf{A}$ tenemos $\eta_Y \circ \mathbf{F}(f) = \mathbf{G}(f) \circ \eta_X$.*

Merece la pena mencionar dos categorías especialmente interesantes: **Set**, la categoría cuyos objetos son conjuntos y cuyos morfismos entre los conjuntos A y B son todas las funciones entre A y B; y **Cat**, la categoría cuyos objetos son todas las categorías pequeñas y cuyos morfismos son los funtores entre ellas.

Ahora estamos preparados para introducir las instituciones y los comorfismos [39, 38]:

Definición 11 *Una* institución *está compuesta por:*

- *una categoría* **Sign** *de* signaturas*;*
- *un funtor* $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ *que asigna un conjunto* $\mathbf{Sen}(\Sigma)$ *de sentencias en Σ a cada signatura $\Sigma \in |\mathbf{Sign}|$;*
- *un funtor* $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$, *asignando una categoría* $\mathbf{Mod}(\Sigma)$ *de* modelos *en Σ a cada signatura $\Sigma \in |\mathbf{Sign}|$; y*
- *para cada signatura $\Sigma \in |\mathbf{Sign}|$, una* relación de satisfacción $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ *entre modelos y sentencias tal que para cada morfismo de signaturas $\sigma : \Sigma \to \Sigma'$, Σ-sentencia $\varphi \in \mathbf{Sen}(\Sigma)$ y Σ'-modelo $M' \in |\mathbf{Mod}(\Sigma)|$:*

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \iff \mathbf{Mod}(\sigma)(M') \models_\Sigma \varphi,$$

conocida como condición de satisfacción.

Definición 12 *Dadas dos instituciones* $\mathcal{I} = (\mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models)$ *e* $\mathcal{I}' = (\mathbf{Sign}', \mathbf{Mod}', \mathbf{Sen}', \models')$*, un* comorfismo entre instituciones *de* $\mathcal{I}$ *a* $\mathcal{I}'$ *consiste en un funtor* $\Phi : \mathbf{Sign} \to \mathbf{Sign}'$*, una transformación natural* $\alpha : \mathbf{Sen} \Rightarrow \Phi; \mathbf{Sen}'$ *y una transformación natural* $\beta : \Phi; \mathbf{Mod}' \Rightarrow \mathbf{Mod}$ *(donde* $\mathbf{F}; \mathbf{G}$ *denota la composición de funtores en el orden diagramático) tal que la siguiente condición de satisfacción se cumple para cada* $\Sigma \in |\mathbf{Sign}|$*,* $\varphi \in |\mathbf{Sen}(\Sigma')|$ *y* $M' \in |\mathbf{Mod}'(\Phi(\Sigma))|$*:*

$$\beta_\Sigma(M') \models_\Sigma \varphi \iff M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi).$$

Pese a que existen muchos otros posibles morfismos entre instituciones [38], solo estamos interesados en los comorfismos para nuestros propósitos en esta tesis.

Capítulo 8

Depuración Declarativa

8.1 Estado del arte

Como se adelantó en la introducción, la depuración declarativa [68] es una técnica de depuración que, a diferencia de las técnicas de depuración tradicionales como los puntos de ruptura, abstrae los detalles de ejecución, que pueden ser difíciles de seguir en general en los lenguajes declarativos, para centrarse en los resultados. Podemos distinguir dos tipos diferentes de depuración declarativa: depuración de *respuestas erróneas*, que se aplica cuando un resultado incorrecto se obtiene a partir de un valor inicial y que ha sido ampliamente utilizada en programación lógica [98, 52], funcional [71, 82, 72], multiparadigma [11, 54, 15] y en la orientada a objetos [12, 44]; y depuración de *respuestas perdidas* [67, 98, 52, 16, 3], que se aplica cuando un resultado es *incompleto*, lo que ha sido menos estudiado porque el cálculo necesario es mucho más complejo que en el caso de respuestas erróneas.

La depuración declarativa es un proceso en dos fases: en la primera se calcula un árbol, el llamado *árbol de depuración*, en el que cada nodo representa un paso de cómputo y cada resultado (es decir, el efecto de cada uno de estos pasos de cómputo) debe ser consecuencia de los resultados en los nodos hijos del nodo en el que se encuentra; en la segunda fase este árbol se recorre siguiendo una estrategia de navegación y preguntando en cada momento a un oráculo externo sobre la corrección de los resultados en los nodos. Esta corrección depende de la existencia de una *semántica pretendida* del programa, que se corresponde con el comportamiento que el usuario tenía en mente y que se usa para recorrer el árbol hasta que se encuentra un nodo incorrecto con todos los hijos correctos, el nodo defectuoso. Este nodo defectuoso debe estar etiquetado para poder identificar el error en el programa original.

En este capítulo presentamos Maude DDebugger, un depurador declarativo para especificaciones en Maude. Maude proporciona diversos mecanismos para depurar especificaciones:

- Permite colorear términos, lo que consiste en imprimir con diferentes colores los operadores utilizados para construir un término que no se ha reducido completamente. Esto facilita reconocer las diferentes funciones que deberían haberse reducido en un término pero no ayuda al usuario a encontrar las razones por las que no se redujeron.

- Permite visualizar la traza, lo que posibilita al usuario seguir la ejecución de la especificación, es decir, la secuencia de aplicaciones de sentencias que ha tenido lugar. La traza es muy personalizable, teniendo opciones para seleccionar qué sentencias, condiciones, sustituciones y términos se muestran. Estas ideas se han aplicado en el paradigma funcional en el depurador mediante trazas *Hat* [18], donde la traza inicial del sistema se modifica mediante reescritura de grafos para adecuarla a las necesidades del usuario.

- Permite invocar un depurador interno. A él se accede colocando puntos de ruptura en ciertos operadores y sentencias; cuando se alcanza alguno de estos puntos de ruptura se entra en el depurador, en el cual el usuario puede ver el término actual y ejecutar el siguiente paso con la traza activada.

- Maude también proporciona un atributo para imprimir que muestra los valores de las variables cuando se aplica una sentencia que tenga este atributo. Aunque no esté especificamente diseñado para depurar, puede ser utilizado con esta intención de una manera similar a la traza: las sentencias sospechosas se pueden mostrar con todas sus variables cada vez que se ejecuten.

La traza, el depurador interno y el atributo de impresión (cuando se usa para depurar) tienen el mismo problema: siguen la ejecución del programa, que no es conocida *a priori* por el usuario, y por tanto deja de lado uno de los puntos fuertes de los programas declarativos: la abstracción de los detalles de ejecución; además, dado que están basados en la traza, se necesita recorrerla entera y comprobar cada paso, ya que estos procesos no proporcionan ningún tipo de guía. Nuestro depurador declarativo soluciona estos problemas permitiendo al usuario depurar cualquier tipo de especificación en Maude de una manera sencilla y natural.

Uno de los puntos fuertes de nuestra aproximación es que, a diferencia de otras propuestas como [16], combina el tratamiento de respuestas erróneas y perdidas y por tanto es capaz de detectar respuestas perdidas debidas a sentencias tanto erróneas como perdidas. El estado del arte de la depuración declarativa puede encontrarse en [94], que contiene una comparación entre los depuradores declarativos B.i.O. (Believe in Oracles) [9], un depurador integrado en el compilador de Curry KICS; Buddha [81, 82], un depurador para Haskell 98; DDT [15], un depurador para TOY; Freja [71], un depurador para Haskell; Hat-Delta [25], que forma parte de un conjunto de herramientas para depurar programas en Haskell; el Depurador Algorítmico de Mercury [54], un depurador integrado en el compilador de Mercury; el depurador de Curry de Münster [53], un depurador integrado en la distribución del compilador de Curry desarrollado en Münster; y Nude [69], el entorno de depuración de NU-Prolog. Extendemos ahora esta comparación teniendo en cuenta la nueva funcionalidad en las últimas versiones de estos depuradores y añadiendo dos nuevos: DDJ [44], un depurador para programas en Java, y nuestro propio depurador, Maude DDebugger. Esta comparación se resume en las tablas 8.1 y 8.2, donde cada columna muestra un depurador declarativo y cada fila una característica. Más concretamente:

- El lenguaje de implementación indica el lenguaje utilizado para implementar el depurador. En ciertos casos se muestra el frontal y la parte posterior (*front-end* y *back-end* en inglés, respectivamente): se refieren, respectivamente, al lenguaje usado para obtener la información necesaria para calcular el árbol de depuración y al lenguaje usado para interactuar con el usuario.

- El lenguaje objetivo indica el lenguaje depurado por la herramienta.

- La fila de estrategias indica las diferentes estrategias de navegación implementadas por los depuradores. Usamos AA para *arriba-abajo* (*top-down* en inglés), que empieza desde la raíz y selecciona un hijo incorrecto para continuar con la navegación hasta que todos los hijos son correctos; DP para *divide y pregunta* (*divide and query* en inglés), que elige en cada caso un nodo que sea raíz de un subárbol cuyo tamaño sea el más próximo a la mitad del tamaño del árbol completo; PP para *paso a paso* (*single stepping* en inglés), que recorre el árbol en postorden; MPP para *el más pesado primero* (*heaviest first* en inglés), una modificación de arriba-abajo que recorre los hijos en orden decreciente de tamaño; MRP para *más reglas primero* (*more rules first* en inglés), otra variante de arriba-abajo que recorre los hijos en orden decreciente de número de reglas en el subárbol; DRP para *divide por reglas y pregunta* (*divide by rules and query* en inglés), una optimización de divide y pregunta que selecciona el nodo cuyo subárbol tiene una cantidad de sentencias más cercana a la mitad del número de sentencias distintas en el árbol; DPM para la estrategia divide y pregunta implementada por el depurador de Mercury; DS para *dependencia de subtérminos* (*subterm dependency* en inglés), una estrategia que permite rastrear subtérminos específicos previamente señalados como erróneos por el usuario; y HD para las estrategias de Hat-Delta.

- La base de datos indica si la herramienta almacena una base de datos con las respuestas dadas por el usario para futuras sesiones de depuración, mientras que memorización indica si esta base de datos está disponible para la sesión actual.
- El frontal indica si el depurador está integrado en el compilador o es independiente.
- La interfaz muestra la interfaz entre el frontal y la parte posterior. Aquí, APT se refiere a los árboles de prueba abreviados generados por Maude; RAR a Ruta Aumentada del Redex (*Augmented Redex Trail* en inglés), el árbol generado por Hat-Delta; AE abrevia Árbol de Ejecución; y Cuenta de Pasos (*Step Count* en inglés) es el nombre del método usado por el depurador B.i.O., que guarda la información usada hasta el momento en un fichero de texto.
- Árbol de depuración muestra cómo se gestiona el árbol de depuración.
- La fila de respuestas perdidas indica si la herramienta puede depurar respuestas perdidas.
- Las respuestas permitidas son: `sí`; `no`; `ns` (*no sabe*); `co` (*confía*); `in` (*inadmisible*), usado para indicar que algunos de los argumentos no deberían haberse calculado; y `qs` y `qn` (*quizás sí* y *quizás no*), que se comportan como `sí` y `no` aunque las preguntas asociadas se pueden repetir más tarde si es necesario. Más detalles sobre estas técnicas de depuración se puede encontrar en [94, 95].
- Traza subexpresiones indica si el usario puede señalar a un subtérmino como erróneo.
- Exploración AD indica si el árbol de depuración se puede navegar libremente.
- En la fila ¿Árboles diferentes? se muestra si es posible construir diferentes árboles de depuración dependiendo de la situación.
- Compresión de árboles indica si la herramienta implementa compresión de árboles [25], una técnica que elimina los nodos redundantes de un árbol de prueba.
- Deshacer informa si la herramienta tiene un comando deshacer.
- Confianza enumera las opciones para confiar disponibles en cada depurador: MO indica que se puede confiar en módulos; FU en funciones; AR en argumentos; y FF en formas finales.

- IGU muestra si la herramienta cuenta con una interfaz gráfica de usuario.
- Versión indica la versión de la herramienta usada en la comparación.

Los resultados en estas tablas se pueden interpretar como sigue:

Estrategias de navegación. Muchas estrategias de navegación han sido propuestas para depuración declarativa [94]. Sin embargo, muchos de los depuradores (incluyendo a Maude DDebugger) solo implementan las técnicas básicas arriba-abajo y divide y pregunta. Por otro lado, DDJ implementa la mayoría de los métodos de navegación conocidos (algunos de ellos desarrollados por los mismos investigadores), incluyendo una adaptación de las técnicas de navegación desarrolladas para Hat-Delta. Entre las técnicas básicas, solo DDJ, DDT y Maude DDebugger implementan el algoritmo más eficiente de la estrategia divide y pregunta, desarrollada por Hirunkitti [94].

Respuestas disponibles. La depuración declarativa depende de un oráculo externo que responda a las preguntas hechas por la herramienta, y por tanto cuanto mayor sea la cantidad de respuestas disponibles más fácil será la interacción. El conjunto mínimo de respuestas aceptadas por todos los depurados se compone de las respuestas *sí* y *no*; Hat-Delta, el depurador de Curry de Münster y Nude no admiten más respuestas, mientras todos los demás aceptan algunas otras. Otras respuestas bien conocidas son *no sabe* y *confía*; la primera, que puede introducir incompletitud, permite al usuario saltarse la pregunta actual y es implementada por B.i.O., DDJ, DDT, Buddha, Mercury y Maude DDebugger, mientras que la segunda evita que el depurador haga preguntas relacionadas con la sentencia actual y es implementada por DDJ, DDT, Buddha, el depurador de Mercury y Maude DDebugger. Buddha y el depurador de Mercury implementan la respuesta *inadmisible*, que indica que algunos argumentos no deberían haberse calculado, redirigiendo el proceso de búsqueda en esta dirección; nuestro depurador facilita un mecanismo similar al depurar respuestas perdidas en módulos de sistema con la respuesta *el término n no es una solución/alcanzable*, que indica que un término en un conjunto no es una solución o no es alcanzable, dirigiendo el proceso en esa dirección. Por último, Freja permite las respuestas *quizás sí* y *quizás no*, que el depurador usa como *sí* y *no*, aunque puede repetir estas preguntas si el error no se encuentra.

Base de datos. Una característica habitual en depuración declarativa es el uso de una base de datos para evitar que la herramienta haga dos veces la misma pregunta, lo cual es implementado por DDJ, DDT, Hat-Delta, Buddha, el depurador de Mercury, Nude y Maude DDebugger. Nude ha mejorado esta técnica permitiendo que la base de datos se use en las sesiones siguientes, lo que ha sido también adoptado por DDJ.

Memoria. Los depuradores guardan los árboles de depuración en memoria de diferentes maneras. El árbol de Hat-Delta se guarda en un sistema de ficheros, DDJ usa una base de datos y el resto de depuradores (incluyendo el nuestro) lo almacenan en memoria principal. Además, los depuradores B.i.O., Buddha, DDJ, el depurador de Mercury, Nude y Maude DDebugger optimizan el consumo de memoria construyendo el árbol bajo demanda.

Trazado de subexpresiones. El depurador de Mercury es el único capaz de indicar que una subexpresión en concreto, y no el término completo, es erróneo, mejorando de esta manera las respuestas *no* e *inadmisible* con información precisa sobre la subexpresión. Con esta técnica la estrategia de navegación se puede concentrar en ciertos nodos del árbol, optimizando el proceso de depuración.

	Maude DDebugger	**B.i.O.**	**Buddha**	**DDJ**	**DDT**	**Freja**
Lenguaje de implementación	Maude	Curry	Haskell	Java	Toy (frontal) Java (parte posterior)	Haskell
Lenguaje Objetivo	Maude	Curry	Haskell	Java	Toy	Subconjunto de Haskell
Estrategias	AA DP	AA	AA	AA DP DRP PP MPP MRP HD	AA DP	AA
¿Base de datos / Memorización?	NO/SÍ	NO/NO	NO/SÍ	SÍ/SÍ	NO/SÍ	NO/NO
Frontal	Trans. prog. independiente	Trans. prog. independiente	Trans. prog. independiente	Trans. prog. independiente	Trans. prog. integrada	Integrado en el compilador
Interfaz	APT bajo demanda	Paso a Paso	AE	AE bajo demanda	AE (XML/TXT)	AE
Árbol de depuración	Memoria princ. bajo demanda	Memoria princ. bajo demanda	Memoria princ. bajo demanda	Base de datos	Memoria princ.	Memoria princ.
¿Respuestas perdidas?	SÍ	NO	NO	NO	SÍ	NO
Respuestas aceptadas	`sí no ns co`	`sí no ns`	`sí no ns in co`	`sí no ns co`	`sí no ns co`	`sí no qs qn`
¿Traza subexpresiones?	NO	NO	NO	NO	NO	NO
¿Exploración AD?	SÍ	SÍ	SÍ	SÍ	SÍ	SÍ
¿Árboles diferentes?	SÍ	NO	NO	SÍ	NO	NO
¿Compresión de árboles?	NO	NO	NO	NO	NO	NO
¿Deshacer?	SÍ	SÍ	NO	SÍ	NO	SÍ
Confianza	MO/FU/FF	MO/FU/AR	MO/FU	FU	FU	MO/FU
¿IGU?	SÍ	NO	NO	SÍ	SÍ	NO
Versión	2.0 (24/5/2010)	Kics 0.81893 (15/4/2009)	1.2.1 (1/12/2006)	2.4 (23/10/2010)	1.2 (29/9/2005)	(2000)

Cuadro 8.1: Comparativa de depuradores declarativos I

	Maude DDebugger	**Hat-Delta**	**Mercury Debugger**	**Münster Curry Debugger**	**Nude**
Lenguaje de implementación	Maude	Haskell	Mercury	Haskell (frontal) Curry (parte posterior)	Prolog
Lenguaje objetivo	Maude	Haskell	Mercury	Curry	Prolog
Estrategias	AA DP	HD	AA DP DS DPM	AA	AA
¿Base de datos / Memorización?	NO/SÍ	NO/SÍ	NO/SÍ	NO/NO	SÍ/SÍ
Frontal	Trans. prog. independiente	Trans. prog. independiente	Compilador independiente	Integrado en el compilador	Compilador independiente
Interfaz	APT	RAR (nativo)	AE bajo demanda	AE	AE bajo demanda
Árbol de depuración	Memoria princ. bajo demanda	Sistema de ficheros	Memoria princ. bajo demanda	Memoria princ.	Memoria princ. bajo demanda
¿Respuestas perdidas?	SÍ	NO	NO	NO	NO
Respuestas aceptadas	`sí no ns co`	`sí no`	`sí no ns in co`	`sí no`	`sí no`
¿Traza subexpresiones?	NO	NO	SÍ	NO	NO
¿Exploración AD?	SÍ	SÍ	SÍ	SÍ	NO
¿Árboles diferentes?	SÍ	NO	NO	NO	NO
¿Compresión de árboles?	NO	SÍ	NO	NO	NO
¿Deshacer?	SÍ	NO	SÍ	NO	NO
Confianza	MO/FU/FF	MO	MO/FU	MO/FU	FU
¿IGU?	SÍ	NO	NO	SÍ	NO
Versión	2.0 (24/5/2010)	2.05 (22/10/2006)	Mercury 0.13.1 (1/12/2006)	AquaCurry 1.0 (13/5/2006)	NU-Prolog 1.7.2 (13/7/2004)

Cuadro 8.2: Comparativa de depuradores declarativos II

Estrategias de construcción. Una novedad de nuestro método es la posibilidad de construir diferentes árboles dependiendo de la complejidad de la especificación y de la experiencia del usuario: los árboles para respuestas tanto erróneas como perdidas se pueden construir siguiendo bien una estrategia de un paso, bien una de varios pasos (dando lugar a cuatro combinaciones). Mientras que la estrategia en un paso hace en general más preguntas, estas son más fáciles de responder que las hechas por la estrategia de varios pasos. Una mejora de esta técnica se ha aplicado a DDJ en [45], permitiendo al sistema equilibrar los árboles de depuración combinando una estructuras llamadas *cadenas*, esto es, secuencias de sentencias donde el resultado final de cada paso son los datos iniciales del siguiente.

Compresión de árboles. El depurador Hat-Delta ha desarrollado una nueva técnica para eliminar nodos redundantes del árbol de ejecución llamada compresión de árboles [25]. Básicamente, esta técnica consiste en eliminar del árbol de depuración, en ciertos casos, los nodos relacionados con el mismo error que su padre, de tal manera que el padre proporciona la información de depuración tanto para él como para sus hijos. Esta técnica es muy similar al equilibrado implementado para DDJ en [45].

Exploración del árbol. La mayoría de los depuradores permiten al usuario navegar libremente el árbol de depuración, incluyendo el nuestro cuando se usa la interfaz gráfica de usuario. Solo el depurador de Curry de Münster y Nude no implementan esta función.

Confianza. Aunque todos los depuradores usan algún mecanismo de confianza, difieren en el objetivo: todos los depuradores excepto Hat-Delta tienen mecanismos para confiar en sentencias concretas, y todos los depuradores excepto DDJ, DDT y Nude pueden confiar en módulos completos. Una funcionalidad novedosa es permitir al usuario confiar en algunos argumentos, lo cual solo está permitido en la actualidad por B.i.O. En nuestro caso, y dado que podemos depurar respuestas perdidas, hemos desarrollado un mecanismo original de confianza: el usuario puede indicar que ciertos tipos y operadores son *finales*, es decir, no se pueden reescribir más; con este método todos los nodos que se refieran a términos finales desde el punto de vista de las reglas son eliminados del árbol de depuración. Por último, un método similar a la confianza consiste en usar una especificación correcta como oráculo para contestar a las preguntas; esta técnica se usa en B.i.O. y en Maude DDebugger.

Comando deshacer. En un método que depende del usuario como oráculo, es normal que se cometan errores y por tanto un comando para deshacer puede ser muy útil. Sin embargo, no todos los depuradores tienen este comando, siendo B.i.O., DDJ, Freja, el depurador de Mercury y Maude DDebugger los únicos que lo implementan.

Interfaz gráfica. Una interfaz gráfica facilita la interacción entre el usuario y la herramienta, permitiendo al usuario navegar libremente por el árbol de depuración y mostrando todas las funciones de manera más amigable. En [94], solo un depurador declarativo (DDT) tenía una interfaz de este tipo, mientras que en la actualidad cuatro herramientas (DDT, DDJ, el depurador de Münster[1] y Maude DDebugger) ofrecen esta posibilidad.

Errores detectados. Merece la pena señalar que solo DDT y Maude DDebugger pueden depurar respuestas perdidas, mientras todos los demás depuradores se dedican exclusivamente a respuestas erróneas. Sin embargo, DDT solo depura respuestas perdidas

[1]Solo disponible para Mac OS X.

debidas a no determinismo, mientras nuestra técnica también es capaz de depurar formas normales y tipos mínimos erróneos.

Apuntes finales. Una característica importante de la depuración declarativa es la escalabilidad. El desarrollo de DDJ ha prestado especial atención a esta característica, y por ello usa una arquitectura compleja que gestiona la memoria disponible y hace uso de una base de datos para almacenar las partes del árbol que no se pueden guardar en memoria principal. Además, las estrategias de navegación han sido modificadas para funcionar con árboles incompletos. Respecto a la reusabilidad, la última versión de B.i.O. implementa una interfaz genérica que permite a otras herramientas que también la implementen usar sus funciones de depuración. Por último, el depurador DDT ha sido actualizado para trabajar con restricciones.

Es importante señalar que hay algunos otros enfoques para depurar respuestas erróneas y perdidas. Un planteamiento interesante es la diagnosis abstracta presentada en [3]. En este artículo los autores describen un marco de depuración donde, dada una especificación del sistema, la herramienta es capaz de identificar respuestas tanto erróneas como perdidas usando interpretación abstracta y sin necesidad de un síntoma inicial. En un trabajo más reciente [2] el sistema ha sido mejorado, siendo ahora capaz de corregir ciertas especificaciones erróneas. Con respecto a otras técnicas, como el comprobador de la propiedad de ser suficientemente completo de Maude [20, Chap. 21] o los conjuntos de descendientes (*descendants* en inglés) [37], nuestra herramienta proporciona una técnica más general dado que permite usar sentencias condicionales y nuestras ecuaciones no es necesario que sean lineales por la izquierda.

8.2 Un cálculo de depuración

En esta sección describimos cómo se obtienen los árboles de depuración utilizados por nuestra herramienta gracias a un cálculo formal, el cual nos permite probar la corrección y completitud de nuestra técnica.

El cálculo para depurar respuestas erróneas nos permite inferir reducciones $t \to t'$, inferencias de tipo $t : s$ y reescrituras $t \Rightarrow t'$. Sus reglas de inferencia, que se muestran en la figura 8.1, son una adaptación de las reglas presentadas en [8, 57] para lógica ecuacional de pertenencia y en [56, 10] para lógica de reescritura.[2]

Este cálculo ha sido extendido para inferir, dado un término inicial, su forma normal, su tipo mínimo y el conjunto de términos alcanzables desde él dada una cota en el número de pasos y una condición a ser satisfecha. Un aspecto importante del cálculo es que proporciona dos tipos diferentes de información: por qué ciertas cosas ocurren (es decir, por qué se alcanza la forma normal, por qué se infiere el tipo mínimo y por qué los términos se incluyen en el conjunto) pero también por qué otras cosas *no* ocurren (esto es, por qué el término no se reduce más, por qué no se puede obtener un tipo menor y por qué no se incluyen más términos en el conjunto). Llamamos al primer tipo de información, que también se calculaba con las reglas de inferencia en 8.1, *información positiva*, mientras que al segundo tipo de información, novedosa con respecto a las reglas anteriores, lo llamamos *información negativa*. En las figuras 8.2 y 8.3 mostramos las principales reglas de inferencia de este cálculo extendido, y remitimos para más detalles a [87, 88]. En primer lugar, vamos a explicar la intuición tras las juicios usados en las reglas de inferencia. En [91] se presenta su definición formal y se demuestra la corrección del cálculo.

[2]Esta adaptación consiste básicamente en orientar de izquierda a derecha las ecuaciones, en lugar de considerarlas igualdades.

(**Reflexividad**)

$$\frac{}{t \Rightarrow t}\,\mathsf{Rf}_{\Rightarrow} \qquad \frac{}{t \rightarrow t}\,\mathsf{Rf}_{\rightarrow}$$

(**Transitividad**)

$$\frac{t_1 \Rightarrow t' \qquad t' \Rightarrow t_2}{t_1 \Rightarrow t_2}\,\mathsf{Tr}_{\Rightarrow} \qquad \frac{t_1 \rightarrow t' \qquad t' \rightarrow t_2}{t_1 \rightarrow t_2}\,\mathsf{Tr}_{\rightarrow}$$

(**Congruencia**)

$$\frac{t_1 \Rightarrow t'_1 \quad \ldots \quad t_n \Rightarrow t'_n}{f(t_1, \ldots, t_n) \Rightarrow f(t'_1, \ldots, t'_n)}\,\mathsf{Cong}_{\Rightarrow} \qquad \frac{t_1 \rightarrow t'_1 \quad \ldots \quad t_n \rightarrow t'_n}{f(t_1, \ldots, t_n) \rightarrow f(t'_1, \ldots, t'_n)}\,\mathsf{Cong}_{\rightarrow}$$

(**Remplazamiento**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \; \{\theta(v_j) : s_j\}_{j=1}^{m} \; \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(t) \Rightarrow \theta(t')}\,\mathsf{Rep}_{\Rightarrow}$$

if $t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) \rightarrow \theta(t')}\,\mathsf{Rep}_{\rightarrow}$$

if $t \rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$

(**Clase de Equivalencia**) (**Reducción de Sujeto**)

$$\frac{t \rightarrow t' \quad t' \Rightarrow t'' \quad t'' \rightarrow t'''}{t \Rightarrow t'''}\,\mathsf{EC} \qquad \frac{t \rightarrow t' \quad t' : s}{t : s}\,\mathsf{SRed}$$

(**Pertenencia**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) : s}\,\mathsf{Mb}$$

if $t : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$

Figura 8.1: Cálculo semántico para módulos de Maude

- Dada una sustitución admisible θ para una condición atómica C, $[C,\theta] \rightsquigarrow \Theta$ indica que Θ es el conjunto de sustituciones que satisfacen la condición atómica C y extienden θ ligando las variables que aparecen en C.
- Dado un conjunto de sustituciones admisibles Θ para una condición atómica C, $\langle C,\Theta\rangle \rightsquigarrow \Theta'$ indica que Θ' es el conjunto de sustituciones que satisfacen la condición C y extienden alguna de las sustituciones admisibles en Θ.
- $\mathit{disabled}(a,t)$ significa que la ecuación o axioma de pertenencia a no puede ser aplicado en la raíz del término t (*at the top* en inglés).
- $t \rightarrow_{\mathit{red}} t'$ indica que o bien el término t es reducido en la raíz en un paso o bien un subtérmino de t es sustituido por su forma normal.
- $t \rightarrow_{\mathit{norm}} t'$ denota que t' es la forma normal de t con respecto a las ecuaciones.
- Dada una condición admisible $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathit{fulfilled}(\mathcal{C},t)$ indica que $\mathcal{C}$ se cumple cuando $\circledast$ se sustituye por t.
- Dada una condición admisible $\mathcal{C}$ como en el caso anterior, $\mathit{fails}(\mathcal{C},t)$ denota que $\mathcal{C}$ no se cumple cuando $\circledast$ se sustituye por t.
- $t :_{ls} s$ indica que $t : s$ y además s es el tipo mínimo con esta propiedad.
- $t \Rightarrow^{top} S$ significa que el conjunto S está formado por todos los términos alcanzables (módulo ecuaciones) desde t en exactamente un paso de reescritura *en la raíz de t.*
- $t \Rightarrow^{q} S$ denota que el conjunto S es el conjunto de términos alcanzables (módulo ecuaciones) desde t con exactamente una aplicación de la regla q en la raíz del término.
- $t \Rightarrow_1 S$ muestra que el conjunto S está formado por todos los términos alcanzables (módulo ecuaciones) desde t en exactamente un paso, donde el paso de reescritura puede aplicarse en cualquier parte en t (esto es, no necesariamente en la raíz del término).
- $t \rightsquigarrow_n^{\mathcal{C}} S$ indica que S es el conjunto de todos los términos (módulo ecuaciones) que satisfacen la condición admisible $\mathcal{C}$ y son alcanzables desde t en como mucho n pasos. De manera similar, $t \rightsquigarrow +_n^{\mathcal{C}} S$ se usa cuando al menos se debe dar un paso y $t \rightsquigarrow !_n^{\mathcal{C}} S$ cuando solo se buscan términos finales.

Las principales reglas del cálculo para inferir formas normales y tipos mínimos se muestran en la figura 8.2, con el siguiente significado.

- La regla Norm muestra que un término está en forma normal cuando no se le pueden aplicar más ecuaciones en la raíz del término, lo cual se indica con los juicios *disabled*, y además sus subtérminos están en forma normal. Respecto a la información negativa a la que nos referimos antes, esta regla usa información negativa por medio de los juicios *disabled*, que prueban que el término no se puede reducir más. Nótese que solo comprobamos las ecuaciones cuyo lado izquierdo, en el cual consideramos las variables a nivel de familias, se ajusta al término actual, lo cual se expresa con $e \ll_K^{top} f(t_1,\ldots,t_n)$; de esta manera evitamos que el cálculo genere subárboles triviales que no serían útiles durante el proceso de depuración.

$$\frac{\mathit{disabled}(e_1, f(t_1,\ldots,t_n)) \quad \ldots \quad \mathit{disabled}(e_l, f(t_1,\ldots,t_n)) \quad t_1 \rightarrow_{norm} t_1 \quad \ldots \quad t_n \rightarrow_{norm} t_n}{f(t_1,\ldots,t_n) \rightarrow_{norm} f(t_1,\ldots,t_n)}\ \mathsf{Norm}$$
$$\text{if } \{e_1,\ldots,e_l\} = \{e \in E \mid e \ll^{top}_{K} f(t_1,\ldots,t_n)\}$$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(l) \rightarrow_{red} \theta(r)}\ \mathsf{Rdc_1}\ \text{ if } l{\rightarrow}r{\Leftarrow}\bigwedge_{i=1}^{n} u_i{=}u'_i \wedge \bigwedge_{j=1}^{m} v_j{:}s_j {\in} E$$

$$\frac{t \rightarrow_{norm} t'}{f(t_1,\ldots,t,\ldots,t_n) \rightarrow_{red} f(t_1,\ldots,t',\ldots,t_n)}\ \mathsf{Rdc_2}\ \text{ if } t{\not\equiv_A}t'$$

$$\frac{t \rightarrow_{red} t_1 \quad t_1 \rightarrow_{norm} t'}{t \rightarrow_{norm} t'}\ \mathsf{NTr}$$

$$\frac{t \rightarrow_{norm} t' \quad t' : s \quad \mathit{disabled}(m_1, t') \quad \ldots \quad \mathit{disabled}(m_l, t')}{t :_{ls} s}\ \mathsf{Ls}$$
$$\text{if } \{m_1,\ldots,m_l\} = \{m \in E \mid m \ll^{top}_{K} t' \ \wedge\ \mathit{sort}(m) < s\}$$

Figura 8.2: Cálculo para formas normales y tipos mínimos

- La regla $\mathsf{Rdc_1}$ reduce un término aplicándole una ecuación cuando comprueba que las condiciones se satisfacen, donde las condiciones de ajuste de patrones están incluidas en las condiciones ecuacionales.

- La regla $\mathsf{Rdc_2}$ reduce un término a base de reducir uno de sus subtérminos a forma normal, comprobando que dicho subtérmino no estaba ya en forma normal.

- La regla NTr describe la transitividad para $\rightarrow_{norm}$, que indica que para alcanzar la forma normal de un término es necesario aplicarle al menos una ecuación (quizás en sus subtérminos) y entonces calcular la forma normal del término obtenido de esta manera. En este caso, la regla de inferencia usa información positiva facilitada por el juicio $t \rightarrow_{red} t_1$.

- La regla Ls se usa para inferir el tipo mínimo de un término. En primer lugar calcula la forma normal del término, y entonces infiere un tipo para este término de tal manera que no se puedan obtener tipos menores. Para evitar que el depurador compruebe axiomas de pertenencia que nunca podrían inferir un tipo para el término, nos limitamos a las sentencias cuyo lado izquierdo se ajusta al término al nivel de familias. De manera similar a la regla Norm, los juicios *disabled* proporcionan la información negativa, mientras $t' : s$ proporciona la positiva.

En la figura 8.3 se muestran las principales reglas de inferencia usadas para deducir conjuntos de términos alcanzables:

- La regla $\mathsf{Rf_1}$ indica que el conjunto que únicamente contiene al término inicial se infiere si no se pueden usar más pasos y el término satisface la condición; de manera análoga, se devuelve el conjunto vacío cuando la condición falla, como muestra la regla $\mathsf{Rf_2}$. La primera regla proporciona información positiva indicando por qué se incluye el término en el conjunto, mientras que la segunda proporciona información negativa probando que el término no debe ser añadido.

- La regla $\mathsf{Tr_1}$ se aplica cuando al menos un paso más puede ser aplicado. En primer lugar se comprueba que el término satisface la condición. Entonces, se calcula el conjunto de términos alcanzables en exactamente un paso, y el conjunto de términos alcanzables desde cada uno de dichos términos dada la misma condición y la cota decrementada en uno. El resultado es la unión de los conjuntos obtenidos de esta manera y del término inicial. La regla Tr_2 se encarga del caso en el que la condición no se cumple para el término inicial; en tal caso, el conjunto resultado no lo incluye.

- La regla Stp devuelve el conjunto de términos alcanzables en exactamente un paso. Para ello calcula, por un lado, el conjunto de términos alcanzables al aplicar exactamente una regla en la raíz del término y, por otro, el conjunto de términos alcanzables para cada uno de los subtérminos del término original. El conjunto final se compone de los términos alcanzables reescribiendo en la raíz del término y la sustitución de los términos obtenidos para los subtérminos en la posición adecuada (aplicando una única sustitución cada vez).
- La regla Top se encarga de calcular el conjunto de términos alcanzables al aplicar una única regla en la raíz. Dado que cada regla puede producir diferentes términos debido a los distintos ajustes, cada aplicación genera un conjunto de términos. Para evitar información trivial de la forma $t \Rightarrow^q \emptyset$ solo usamos reglas cuyos lados izquierdos se ajusten al término al nivel de familias. Esta regla de inferencia proporciona información tanto positiva como negativa, dependiendo del resultado de los juicios en las premisas: el conjunto vacío proporciona información negativa, mientras que en otro caso la información es positiva. Esta información es después propagada por el resto de reglas con el juicio $t \Rightarrow_1 S$.
- La regla Rl devuelve el conjunto de términos obtenidos al aplicar una única regla. Primero se obtiene el conjunto de sustituciones debido al ajuste con el lado izquierdo de la regla, y entonces se usa para encontrar el conjunto de sustituciones que satisfacen la condición. Este conjunto final se usa para instanciar el lado derecho de la regla y obtener el conjunto de términos alcanzables. El tipo de información proporcionada por esta regla se corresponde con la información proporcionada por las sustituciones: si se obtiene el conjunto vacío (información negativa) entonces la regla calcula el conjunto vacío de términos, lo que se corresponde con la información negativa que demuestra que no se pueden obtener términos con esa regla; razonamos de manera análoga cuando el conjunto de sustituciones no es vacío (información positiva). Esta información se propaga por el resto de las reglas de inferencia justificando por qué algunos términos son alcanzables mientras otros no lo son.
- La regla Red_1 nos permite simular el comportamiento de Maude. Esta regla permite al usuario, a la hora de calcular el conjunto de términos alcanzables desde un término t, reducir t a su forma normal, calcular el conjunto de términos alcanzables desde dicha forma normal y, finalmente, calcular la forma normal de los términos en el conjunto.

Una vez hemos introducido el cálculo, podemos construir un árbol de prueba para la búsqueda mostrada en la sección 7.1.7. Recordemos que, partiendo de una configuración inicial con dos personas esperando en una cola, `adri` y `et`, intentamos encontrar una configuración en la que `adri` ha comprado `videogame`, pero dicha configuración no existe:

```
Maude> (search [1 v] ordIns(adri et nil) =>* S:Sale
        s.t. "adri" buys "videogame" in S:Sale .)
search in SALE :[1 v] ordIns(adri et nil) =>* S:Sale .

No solution.
```

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}}\ \mathsf{Rf_1} \qquad \frac{\mathit{fails}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf_2}$$

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)\ \ t \Rrightarrow_1 \{t_1, \ldots, t_k\}\ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1\ \ldots\ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i\ \cup\ \{t\}}\ \mathsf{Tr_1}$$

$$\frac{\mathit{fails}(\mathcal{C}, t)\ \ t \Rrightarrow_1 \{t_1, \ldots, t_k\}\ \ t_1 \rightsquigarrow_n^{\mathcal{C}} S_1\ \ldots\ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr_2}$$

$$\frac{f(t_1, \ldots, t_m) \Rrightarrow^{top} S_t \quad t_1 \Rrightarrow_1 S_1 \quad \cdots \quad t_m \Rrightarrow_1 S_m}{f(t_1, \ldots, t_m) \Rrightarrow_1 S_t\ \cup\ \bigcup_{i=1}^{m} \{f(t_1, \ldots, u_i, \ldots, t_m) \mid u_i \in S_i\}}\ \mathsf{Stp}$$

$$\frac{t \Rrightarrow^{q_1} S_{q_1} \quad \cdots \quad t \Rrightarrow^{q_l} S_{q_l}}{t \Rrightarrow^{top} \bigcup_{i=1}^{l} S_{q_i}}\ \mathsf{Top}\ \ \text{if } \{q_1, \ldots, q_l\} = \{q \in R \mid q \ll_K^{top} t\}$$

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0 \rangle \rightsquigarrow \Theta_1\ \cdots\ \langle C_k, \Theta_{k-1} \rangle \rightsquigarrow \Theta_k}{t \Rrightarrow^{q} \bigcup_{\theta \in \Theta_k} \{\theta(r)\}}\ \mathsf{Rl}\ \ \text{if } q : l \Rightarrow r \Leftarrow C_1\ \wedge\ \ldots\ \wedge\ C_k\ \in R$$

$$\frac{t \to_{norm} t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \to_{norm} t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S}\ \mathsf{Red_1}$$

Figura 8.3: Cálculo para respuestas perdidas

El árbol de prueba para este cómputo se muestra en la figura 8.4. La inferencia comienza en la figura 8.4(a); para construir este árbol necesitamos una cota del número de pasos dados en la búsqueda,[3] que en este caso es 5. En este árbol `oi` abrevia `ordIns`, $\mathcal{C}$ la condición (extendida con el ajuste de patrones) `(S :=` $\circledast$`) /\ "adri" buys "videogame" in S` y los subárboles $\bigtriangledown$ demostraciones más sencillas que no explicaremos en profundidad. Es importante notar que hemos añadido a las reglas de inferencia Rdc_1 y Rl el nombre de la sentencia que se usa (o $\perp$ cuando no está etiquetado) y el operador en la raíz en la regla Top, y que hemos omitido `nil` en algunos términos (y las demostraciones relacionadas con él) para mejorar la legibilidad. El hijo de la raíz situado más a la izquierda, T_1, está a cargo de calcular la reducción del término inicial a su forma normal S_n, que abrevia `[`l v`]`et et, donde l es `< "lettuce", 6 >`, v es `< "videogame", 10 >` y et es `["et", 16, ""]`, mientras el hijo más a la derecha (†) continúa con la búsqueda desde S_n. El hijo más a la derecha de este último nodo, T_2, se muestra en la figura 8.4(b). Este nodo demuestra que el término inicial no satisface la condición ajustando el término al patrón, aplicando la sustitución obtenida de esta manera, y entonces reduciendo el término obtenido con la ecuación `buy2`. El árbol a la derecha de T_2 describe cómo obtener el conjunto unitario que contiene a S_1, lo cual abrevia `[`l v et`]` et, mediante la aplicación de un paso a S_n. El árbol muestra que, aunque los subtérminos no se pueden reescribir, se puede reescribir el término en la raíz con la regla `in`, introduciendo el primer et en la tienda. Por último, el hijo más a la derecha continúa con la búsqueda con la cota decrementada en un paso; este árbol es muy similar a (†) y por tanto no entraremos en detalles sobre él.

El árbol T_1 se muestra en la figura 8.5 y, dado que contiene el nodo defectuoso, requiere un examen en profundidad. La figura 8.5(a) comienza el cómputo de la forma normal del término inicial; su hijo izquierdo reduce el término `oi(adri et)` a su forma normal et et, mientras que el derecho reduce los términos `l` y `v` y comprueba que los términos obtenidos de esta manera están en forma normal. Nos centraremos en el hijo izquierdo, dado que contiene el nodo defectuoso y, además, porque los nodos en el hijo derecho son muy parecidos a los que aparecen en el hijo izquierdo. Como mencionamos antes, el hijo izquierdo reduce `oi(adri et)` a su forma normal; para ello, primero obtiene la forma normal de `adri`, $adri$, que abrevia `["adri", 15, ""]`, aplicando la correspondiente ecuación (sin etiqueta) y comprobando después que está en forma normal porque sus subtérminos lo están.[4] La reducción continúa con T_3, mostrado en la figura 8.5(b), en la que el hijo izquierdo reduce `et` a su forma normal et, mientras el hijo derecho comienza la reducción de la función `ordIns` aplicando la ecuación `oi2`, que devuelve `io(oi(`et`),` $adri$`)`. La figura 8.5(c) presenta el árbol T_4, que reduce `oi(`et`)` a su forma normal et aplicando las ecuaciones `oi2`, `oi1` y `io1`. Una vez obtenido el resultado, la reducción la finaliza el árbol T_5 en la figura 8.5(d). En ella, la ecuación `io3` se aplica en el nodo ($\star$), donde se alcanza el término et `io(nil,` et`)` a partir de `io(`et`,` $adri$`)`, lo que es evidentemente erróneo porque $adri$ ha desaparecido. Puesto que este nodo no tiene hijos y es erróneo, es el nodo defectuoso y tiene asociado una fragmento incorrecto de código: la ecuación `io3`. El resto del árbol propaga este error hasta que se obtiene la forma normal. Mostraremos en la sección 8.4 cómo usar la herramienta para recorrer el árbol y encontrar el error en la especificación.

[3]Este valor lo calcula automáticamente Maude DDebugger.

[4]El subárbol $\bigtriangledown$ indica que se necesitan más pasos: dado que el término `15` es de hecho una abreviatura de `s(s(...s(0)))`, necesita quince pasos para alcanzar el caso base.

(a)

$$T_2 = \dfrac{\dfrac{\dfrac{\triangledown\ldots\triangledown}{S_n \Rightarrow^{\texttt{in}} \{S_1\}}\ \mathsf{Rl_{in}}}{S_n \Rightarrow^{top} \{S_1\}}\ \mathsf{Top_{[\_]\_}} \quad \dfrac{\dfrac{}{l\ v \Rightarrow^{top} \emptyset}\ \mathsf{Top_{\_\_}} \quad \dfrac{\dfrac{}{l \Rightarrow^{top} \emptyset}\ \mathsf{Top_{<\_,\_>}}}{l \Rightarrow_1 \emptyset}\ \mathsf{Stp} \quad \dfrac{\dfrac{}{v \Rightarrow^{top} \emptyset}\ \mathsf{Top_{<\_,\_>}}}{v \Rightarrow_1 \emptyset}\ \mathsf{Stp}}{l\ v \Rightarrow_1 \emptyset}\ \mathsf{Stp} \quad \dfrac{\dfrac{}{et\ et \Rightarrow^{top} \emptyset}\ \mathsf{Top_{\_\_}} \quad \dfrac{\dfrac{}{et \Rightarrow^{top} \emptyset}\ \mathsf{Top_{[\_,\_,\_]}}}{et \Rightarrow_1 \emptyset}\ \mathsf{Stp} \quad \dfrac{\dfrac{}{et \Rightarrow^{top} \emptyset}\ \mathsf{Top_{[\_,\_,\_]}}}{et \Rightarrow_1 \emptyset}\ \mathsf{Stp}}{et\ et \Rightarrow_1 \emptyset}\ \mathsf{Stp}}{S_n \Rightarrow_1 \{S_1\}}\ \mathsf{Stp}$$

$$T_1 = \dfrac{\dfrac{T_2 \qquad \dfrac{\triangledown\ldots\triangledown}{S_1 \rightsquigarrow^4_{\mathcal{C}} \emptyset}\ \mathsf{Tr_1}}{(\dagger)\ S_n \rightsquigarrow^5_{\mathcal{C}} \emptyset}\ \mathsf{Tr_1}}{\texttt{[l v] oi(adri et nil)} \rightsquigarrow^5_{\mathcal{C}} \emptyset}\ \mathsf{Red_1}$$

(b)

$$\dfrac{\dfrac{\triangledown\ldots\triangledown}{[\mathtt{S} := S_n, \emptyset] \rightsquigarrow \{\mathtt{S} \mapsto S_n\}}\ \mathsf{PatC} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{"adri" buys "videogame" in S} \rightarrow_{red} \texttt{false}}\ \mathsf{Rdc_{1_{buy2}}} \quad \dfrac{}{\texttt{false} \rightarrow_{norm} \texttt{false}}\ \mathsf{Norm}}{\texttt{"adri" buys "videogame" in S} \rightarrow_{norm} \texttt{false}}\ \mathsf{NTr} \quad \dfrac{}{\texttt{true} \rightarrow_{norm} \texttt{true}}\ \mathsf{Norm}}{[\texttt{"adri" buys "videogame" in S} = \texttt{true}, \{\mathtt{S} \mapsto S_n\}] \rightsquigarrow \emptyset}\ \mathsf{EqC_2}}{\langle\texttt{"adri" buys "videogame" in S} = \texttt{true}, \{\mathtt{S} \mapsto S_n\}\rangle \rightsquigarrow \emptyset}\ \mathsf{SubsCond}}{\mathit{fails}(\mathcal{C}, S_n)}\ \mathsf{Fail}$$

Figure 3.4: Proof tree for the search in Section 2.1.7

Figura 8.4: Árbol de prueba para la búsqueda en la sección 7.1.7: Árbol principal (a) y T_2 (b)

(a)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{adri} \to_{red} adri}\,\mathsf{Rdc}_{1\bot} \quad \dfrac{\dfrac{}{\texttt{"adri"} \to_{norm} \texttt{"adri"}}\,\mathsf{Norm} \quad \dfrac{\triangledown}{\texttt{15} \to_{norm} \texttt{15}}\,\mathsf{Norm} \quad \dfrac{}{\texttt{""} \to_{norm} \texttt{""}}\,\mathsf{Norm}}{adri \to_{norm} adri}\,\mathsf{Norm}}{\texttt{adri} \to_{norm} adri}\,\mathsf{NTr}}{\texttt{oi(adri et)} \to_{red} \texttt{oi(}adri\,\texttt{et)}}\,\mathsf{Rdc}_2 \quad T_3}{\texttt{oi(adri et)} \to_{norm} et\ et}\,\mathsf{NTr}}{\texttt{[l v] oi(adri et)} \to_{red} \texttt{[l v]}\ et\ et}\,\mathsf{Rdc}_2 \quad \dfrac{\triangledown \quad \dfrac{\triangledown \quad \dfrac{\dfrac{\triangledown\ \ \triangledown}{l\ v \to_{norm} l\ v}\,\mathsf{Norm} \quad \dfrac{\triangledown\ \ \triangledown}{et\ et \to_{norm} et\ et}\,\mathsf{Norm}}{[l\ v]\ \ et\ et \to_{norm} [l\ v]\ \ et\ et}\,\mathsf{Norm}}{\texttt{[l}\ v\texttt{]}\ \ et\ et \to_{norm} [l\ v]\ \ et\ et}\,\mathsf{NTr}}{\texttt{[l v]}\ \ et\ et \to_{norm} [l\ v]\ \ et\ et}\,\mathsf{NTr}}{\texttt{[l v] oi(adri et)} \to_{norm} [l\ v]\ \ et\ et}\,\mathsf{NTr}$$

(b)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{et} \to_{norm} et}\,\mathsf{Rdc}_{1\bot} \quad \triangledown}{\texttt{et} \to_{norm} et}\,\mathsf{NTr}}{\texttt{oi(}adri\,\texttt{et)} \to_{red} \texttt{oi(}adri\ et\texttt{)}}\,\mathsf{Rdc}_2 \quad \dfrac{\dfrac{}{\texttt{oi(}adri\ et\texttt{)} \to_{red} \texttt{io(oi(}et\texttt{)},adri\texttt{)}}\,\mathsf{Rdc}_{1_{\mathsf{oi2}}} \quad \dfrac{\dfrac{T_4}{\texttt{io(oi(}et\texttt{)},adri\texttt{)} \to_{red} \texttt{io(}et,adri\texttt{)}}\,\mathsf{Rdc}_2 \quad T_5}{\texttt{io(oi(}et\texttt{)},adri\texttt{)} \to_{norm} et\ et}\,\mathsf{NTr}}{\texttt{oi(}adri\ et\texttt{)} \to_{norm} et\ et}\,\mathsf{NTr}}{\texttt{oi(}adri\,\texttt{et)} \to_{norm} et\ et}\,\mathsf{NTr}$$

(c)

$$\dfrac{\dfrac{}{\texttt{oi(}et\texttt{)} \to_{red} \texttt{io(oi(nil)},et\texttt{)}}\,\mathsf{Rdc}_{1_{\mathsf{oi2}}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\mathsf{Rdc}_{1_{\mathsf{oi1}}} \quad \dfrac{}{\texttt{nil} \to_{norm} \texttt{nil}}\,\mathsf{Norm}}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\mathsf{NTr}}{\texttt{io(oi(nil)},et\texttt{)} \to_{red} \texttt{io(nil},et\texttt{)}}\,\mathsf{Rdc}_2 \quad \dfrac{\dfrac{}{\texttt{io(nil},et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\mathsf{io1}}} \quad \dfrac{\triangledown\cdots\triangledown}{et \to_{norm} et}\,\mathsf{Norm}}{\texttt{io(nil},et\texttt{)} \to_{norm} et}\,\mathsf{NTr}}{\texttt{io(oi(nil)},et\texttt{)} \to_{norm} et}\,\mathsf{NTr}}{\texttt{oi(}et\texttt{)} \to_{norm} et}\,\mathsf{NTr}$$

(d)

$$\dfrac{(\star)\ \dfrac{}{\texttt{io(}et\texttt{,}\ adri\texttt{)} \to_{red} et\ \texttt{io(nil,}\ et\texttt{)}}\,\mathsf{Rdc}_{1_{\mathsf{io3}}} \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{io(nil,}\ et\texttt{)} \to_{red} et}\,\mathsf{Rdc}_{1_{\mathsf{io1}}} \quad \dfrac{\triangledown\cdots\triangledown}{et \to_{norm} et}\,\mathsf{Norm}}{\texttt{io(nil,}\ et\texttt{)} \to_{norm} et}\,\mathsf{NTr}}{et\ \texttt{io(nil,}\ et\texttt{)} \to_{red} et\ et}\,\mathsf{Rdc}_2 \quad \dfrac{\triangledown\ \ \triangledown}{et\ et \to_{norm} et\ et}\,\mathsf{Norm}}{et\ \texttt{io(nil,}\ et\texttt{)} \to_{norm} et\ et}\,\mathsf{NTr}}{\texttt{io(}et\texttt{,}\ adri\texttt{)} \to_{norm} et\ et}\,\mathsf{NTr}$$

Figure 3.5: Proof tree T_1 for the reduction to normal form of the initial term
Figura 8.5: Árbol de prueba T_1 para la reducción a forma normal del término inicial: T_1 (a), T_3 (b), T_4 (c) y T_5 (d)

8.3 Árboles de depuración

Usando los árboles de prueba obtenidos con el cálculo de la sección anterior como árboles de depuración somos capaces de localizar sentencias erróneas y perdidas y condiciones de búsqueda erróneas, que intuitivamente se definen como sigue:[5]

- Una especificación tiene una *sentencia errónea* (donde una sentencia es tanto una ecuación como un axioma de pertenencia o una regla) si existe un término tal que el usuario espera que la sentencia se aplique al término (esto es, el usuario espera que el término se ajuste al lado izquierdo de la sentencia y la condición se satisfaga) pero el resultado de dicha aplicación es erróneo.

- Dada una regla $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$, una especificación tiene una regla errónea si el usuario espera que los juicios $[l := t, \emptyset] \leadsto \Theta_0$, $[C_1, \Theta_0] \leadsto \Theta_1$, ..., $[C_n, \Theta_{n-1}] \leadsto \Theta_n$ se den pero la aplicación de Θ_n a r no proporciona el conjunto de términos esperado para la regla.

- Una especificación tiene una *condición errónea* $\mathcal{C} \equiv l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ si existe un término t tal que, cuando $\circledast$ se sustituye por t, o bien la condición se cumple pero el usuario esperaba que no se cumpliese, o la condición no se cumple pero el usuario esperaba que lo hiciese.

- Una especificación tiene una *ecuación perdida* si existe un término t tal que no se espera que esté en forma normal pero tampoco se espera que ninguna de las ecuaciones en la especificación se le aplique.

- Una especificación tiene un *axioma de pertenencia perdido* si existe un término t en forma normal tal que el tipo mínimo calculado para t no es el esperado pero no se espera que ninguno de los axiomas de pertenencia de la especificación se le aplique.

- Una especificación tiene una *regla perdida* si existe un término t tal que todas las reglas aplicadas en la raíz del término llevan a juicios $t \Rightarrow^{q_i} S_{q_i}$ esperados por el usuario pero su unión $\bigcup S_{q_i}$ no contiene todos los términos que el usuario esperaba alcanzar mediante reescrituras en la raíz del término.

Los árboles obtenidos con este cálculo se pueden usar como árboles de depuración, pero presentan los problemas de (i) tener muchos nodos cuya corrección no depende de la especificación y (ii) contener algunos nodos cuyos juicios, una vez traducidos a preguntas al usuario, son muy difíciles de contestar. Por estas razones hemos desarrollado una técnica que facilita y acorta el proceso de depuración manteniendo su completitud y corrección. Para cada árbol de prueba T, aplicamos una función $APT(T)$ (de *Abbreviated Proof Tree*, Árbol de Prueba Abreviado), o simplemente APT cuando el árbol T puede deducirse del contexto, para depurar especificaciones en Maude. Las reglas APT, descritas en [91], proporcionan las siguientes ventajas:

- Eliminan del árbol aquellos nodos cuya corrección puede inferirse de la corrección de sus hijos, es decir, los nodos que no contienen información de depuración. Por ejemplo, la información asociada a usos de la reflexividad es eliminada del árbol.

- Su uso permite a la herramienta elegir entre diferentes tipos de árbol, dependiendo de la complejidad de la especificación y de la experiencia del usuario a cargo del

[5]Remitimos a [91] para la demostración y las definiciones formales.

proceso de depuración: el árbol de un paso, que solo contiene reescrituras en un paso, y el árbol de varios pasos, que puede contener reescrituras en uno o más pasos; en el segundo los nodos se colocan de tal manera que el árbol se equilibra, y por tanto las estrategias de navegación precisan menos preguntas en general para encontrar el nodo defectuoso, aunque estas preguntas pueden ser más complejas que las hechas en árboles de un paso.[6] Por ejemplo, la regla ($\mathbf{APT}_8^o$) construye el árbol de un paso para respuestas erróneas eliminando las inferencias por transitividad, mientras la regla ($\mathbf{APT}_8^m$) construye el árbol de varios pasos conservándolas. En esta sección presentamos una función más general APT', que devuelve conjuntos de árboles en lugar de árboles, y que es usada como función auxiliar para calcular APT:

$$(\mathbf{APT}_8^o) \quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}\mathsf{Tr}_{\Rightarrow}\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}_8^m) \quad APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}\mathsf{Tr}_{\Rightarrow}\right) = \left\{\frac{APT'(T_1) \; APT'(T_2)}{t_1 \Rightarrow t_2}\mathsf{Tr}_{\Rightarrow}\right\}$$

- Simplifican las preguntas hechas al usuario, reproduciendo el comportamiento esperado de Maude (es decir, preguntas relacionadas con reducciones evitan los estados intermedios y siempre preguntan sobre formas normales, mientras que las reescrituras se aplican una vez la forma normal del término se ha alcanzado). Por ejemplo, la regla ($\mathbf{APT}_3$) asocia la información de depuración en Rdc_1, una regla de inferencia a cargo de aplicar una ecuación, con la transitividad bajo ella:

$$(\mathbf{APT}_3) \quad APT'\left(\frac{\dfrac{T_1 \; \ldots \; T_n}{t \to t''}\mathsf{Rdc}_1 \quad T'}{t \to t'}\mathsf{NTr}\right) =$$

$$\left\{\frac{APT'\,(T_1) \; \ldots \; APT'\,(T_n) \; APT'\,(T')}{t \to t'}\mathsf{Rdc}_1\right\}$$

- Eliminan las preguntas difíciles de contestar y asocian la información de depuración a nodos con preguntas más fáciles. Por ejemplo, las preguntas relacionadas con reescrituras en la raíz se reemplazan con preguntas sobre reescrituras en un paso con la regla ($\mathbf{APT}_4$):

$$(\mathbf{APT}_4) \quad APT'\left(\frac{\dfrac{T_1 \; \ldots \; T_n}{t \Rightarrow^{top} S'}\mathsf{Top} \quad T_1' \ldots T_m'}{t \Rightarrow_1 S}\mathsf{Stp}\right) =$$

$$\left\{\frac{APT'\,(T_1) \; \ldots \; APT'\,(T_n) \; APT'\,(T_1') \; \ldots \; APT'\,(T_m')}{t \Rightarrow_1 S}\mathsf{Top}\right\}$$

- Se aplican sin necesidad de calcular el árbol de prueba asociado, con lo que reducen el tiempo y el espacio necesarios para construir el árbol.

Dado un árbol de prueba T representando una inferencia errónea en el cálculo presentado en la sección 8.2, demostramos en [91] que:

[6] Por supuesto, ambos árboles son completos y correctos.

- $APT(T)$ contiene al menos un nodo defectuoso (completitud).
- Cualquier nodo defectuoso en $APT(T)$ tiene asociada una sentencia errónea o perdida o una condición errónea (corrección).

En la figura 8.6 presentamos el árbol de prueba abreviado de varios pasos obtenido al aplicar la función APT al árbol en la figura 8.4. Los árboles $\bigtriangledown'$ representan la abreviación de los $\bigtriangledown$ en las figuras anteriores. La figura 8.6(a) muestra el inicio del cómputo, mientras que la figura 8.6(b) muestra T_1', la figura 8.6(c) T_2' y la figura 8.6(d) T_3'. A primera vista observamos dos ventajas de la abreviación: (i) el número de nodos se ha reducido de 67 a 31, esto es, el nuevo árbol tiene aproximadamente la mitad de tamaño que el árbol original, y (ii) el nodo erróneo, marcado con ($\star$) en la figura 8.6(b), ha cambiado (mediante la aplicación de la regla ($\mathbf{APT}_3$)) para dar lugar a una pregunta más sencilla.

A pesar de haberse conseguido ya una gran reducción del árbol, aún podemos usar las técnicas de confianza proporcionadas por el depurador:

- Solo las sentencias etiquetadas son tenidas en cuenta al calcular el árbol de depuración. Además, se puede confiar tanto en módulos como en sentencias antes de empezar el proceso de depuración, y en estos últimos se puede confiar también sobre la marcha. Por esta razón las reducciones llevadas a cabo por sentencias sin etiquetar en la figura 8.6, que hemos denotado con $\bot$, se eliminarán del árbol. Estos nodos se han marcado con ($\diamondsuit$).
- Se puede seleccionar un módulo correcto como oráculo. De esta manera, los nodos correctos se eliminan del árbol sin necesidad de preguntar al usuario.
- Podemos indicar que ciertos términos construidos (términos compuestos solamente por operadores definidos con el atributo `ctor`) son *finales*, es decir, no se les pueden aplicar reglas. Para ello, basta señalar qué tipos y qué operadores son finales. De esta manera, si un juicio indica que el conjunto de términos alcanzables en un paso es vacío se considerará correcto y se eliminará del árbol. En nuestro ejemplo podemos considerar como finales los tipos `Nat`, `Bool`, `Shop` y `OList`, y por tanto los términos que tengan estos tipos (y los correspondientes subtipos, como `Item` y `Person` para `Shop`) se eliminarán del árbol. Hemos marcado estos nodos en la figura 8.6 con ($\heartsuit$).
- Además, consideramos que dichos términos construidos están en forma normal y por tanto los nodos que declaran esto se eliminan automáticamente del árbol. Los nodos señalados con ($\spadesuit$) en la figura 8.6 cumplen esta condición y se eliminan del árbol.

8.4 Usando el depurador

En esta sección mostramos cómo usar el depurador mediante el ejemplo de las rebajas de la sección 7.1.7. Nótese que es necesario que se satisfagan los siguientes requisitos para que el depurador funcione adecuadamente: la especificación debe ser ejecutable (véase la sección 7.1) y la información introducida por el usuario debe ser exacta, lo que incluye tanto sus respuestas como la información de confianza. Además de la información de esta sección, muchos otros comandos y opciones se describen en [85].

Como se explicó en la sección anterior, los mecanismos de confianza pueden reducir radicalmente el tamaño del árbol de depuración. Por ello, antes de empezar el proceso de depuración indicamos que ciertos tipos son finales con los comandos:

(a)

$$\dfrac{\dfrac{\dfrac{\triangledown'}{(\spadesuit)\ adri \to_{norm} adri}\,\mathsf{Norm}}{(\diamondsuit)\ \mathtt{adri} \to_{norm} adri}\,\mathsf{Rdc}_{1\bot} \quad \dfrac{\dfrac{\triangledown'}{(\spadesuit)\ et \to_{norm} et}\,\mathsf{Norm}}{(\diamondsuit)\ \mathtt{et} \to_{norm} et}\,\mathsf{Rdc}_{1\bot} \quad T'_1 \quad \triangledown' \quad \triangledown' \quad T'_2 \quad T'_3 \quad \dfrac{\dfrac{\triangledown' \quad \triangledown'}{(\spadesuit)\ l\ v \to_{norm} l\ v}\,\mathsf{Norm} \quad \dfrac{\triangledown' \quad \triangledown'}{(\spadesuit)\ et\ et \to_{norm} et\ et}\,\mathsf{Norm}}{(\spadesuit)\ \mathtt{[}l\ v\mathtt{]}\ \ et\ et \to_{norm} \mathtt{[}l\ v\mathtt{]}\ \ et\ et}\,\mathsf{Norm} \quad \dfrac{\triangledown' \dots \triangledown'}{S_1 \rightsquigarrow^{4}_{\mathcal{C}} \emptyset}\,\mathsf{Tr}_1}{\mathtt{[l\ v]\ oi(adri\ et\ nil)} \rightsquigarrow^{5}_{\mathcal{C}} \emptyset}\,\mathsf{Red}_1$$

(b)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \mathtt{nil} \to_{norm} \mathtt{nil}}\,\mathsf{Norm}}{\mathtt{oi(nil)} \to_{norm} \mathtt{nil}}\,\mathsf{Rdc}_{1_{oi1}} \quad \dfrac{\dfrac{\triangledown' \dots \triangledown'}{(\spadesuit)\ et \to_{norm} et}\,\mathsf{Norm}}{\mathtt{io(nil,}\,et\mathtt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{io1}}}{\mathtt{oi(}et\mathtt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{oi2}} \quad \dfrac{\dfrac{\dfrac{\triangledown' \dots \triangledown'}{(\spadesuit)\ et \to_{norm} et}\,\mathsf{Norm}}{\mathtt{io(nil,}\ et\mathtt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{io1}} \quad \dfrac{\triangledown' \quad \triangledown'}{(\spadesuit)\ et\ et \to_{norm} et\ et}\,\mathsf{Norm}}{(\star)\ \mathtt{io(}et\mathtt{,}\ adri\mathtt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{io3}}}{\mathtt{oi(}adri\ et\mathtt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{oi2}}$$

(c)

$$\dfrac{\triangledown' \dots \triangledown' \quad \dfrac{\dfrac{}{(\spadesuit)\ \mathtt{false} \to_{norm} \mathtt{false}}\,\mathsf{Norm}}{\mathtt{"adri"\ buys\ "videogame"\ in\ S} \to_{norm} \mathtt{false}}\,\mathsf{Rdc}_{1_{buy2}} \quad \dfrac{}{(\spadesuit)\ \mathtt{true} \to_{norm} \mathtt{true}}\,\mathsf{Norm}}{fails(\mathcal{C}, S_n)}\,\mathsf{Fail}$$

(d)

$$\dfrac{\dfrac{\triangledown' \dots \triangledown'}{S_n \Rightarrow^{\mathtt{in}} \{S_1\}}\,\mathsf{Rl}_{\mathtt{in}} \quad \dfrac{\dfrac{}{(\heartsuit)\ l \Rightarrow_1 \emptyset}\,\mathsf{Top}_{<\_,\_>} \quad \dfrac{}{(\heartsuit)\ v \Rightarrow_1 \emptyset}\,\mathsf{Top}_{<\_,\_>}}{(\heartsuit)\ l\ v \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\_\_} \quad \dfrac{\dfrac{}{(\heartsuit)\ et \Rightarrow_1 \emptyset}\,\mathsf{Top}_{[\_,\_,\_]} \quad \dfrac{}{(\heartsuit)\ et \Rightarrow_1 \emptyset}\,\mathsf{Top}_{[\_,\_,\_]}}{(\heartsuit)\ et\ et \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\_\_}}{S_n \Rightarrow_1 \{S_1\}}\,\mathsf{Top}_{[\_]\_}$$

Figure 3.6: Abbreviated proof tree for the search in Section 2.1.7

Figura 8.6: Árbol de prueba abreviado para la búsqueda en la sección 7.1.7: Árbol principal (a), T'_1 (b), T'_2 (c) y T'_3 (d)

```
Maude> (set final select on .)

Final select is on.

Maude> (final select Nat Bool Shop OList .)

Sorts Bool Nat OList Shop are now final.
```

Dado que hemos inspeccionado exhaustivamente la especificación y podemos considerarnos expertos en la materia, seleccionamos el árbol de depuración de varios pasos:

```
Maude> (many-steps missing tree .)

Many-steps tree selected when debugging missing answers.
```

Por último, seleccionamos la estrategia de navegación arriba-abajo, que muestra las preguntas asociadas a todos los hijos de la raíz y solicita al usuario seleccionar uno de ellos como incorrecto o indicar que todos ellos son correctos:

```
Maude> (top-down strategy .)

Top-down strategy selected.
```

El proceso de depuración comienza con el comando:

```
Maude> (missing [l v] ordIns(et adri nil) =>* S:Sale
        s.t. "adri" buys "videogame" in S:Sale .)
```

El depurador construye el árbol mostrado en la figura 8.7, que se corresponde con el árbol en la figura figura 8.6 tras eliminar todos los nodos relacionados con información de confianza. La primera serie de preguntas se refiere a los hijos de la raíz:

```
Question 1 :
Is this reduction (associated with the equation oi2) correct?

ordIns(["adri",15,""]["et",16,""]nil) -> ["et",16,""]["et",16,""]nil

Question 2 :
Did you expect [< "lettuce",6 > < "videogame",10 >]
                      ["et",16,""]["et",16,""]nil not to be a solution?

Question 3 :
Are the following terms all the reachable terms from
 [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil in one step?

1 [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil

Question 4 :
Did you expect that no solutions can be obtained from
 [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ?

Maude> (1 : no .)
```

Al estudiar las preguntas, nos damos cuenta de que la primera (que se corresponde con el nodo ($\wp$) en la figura 8.7) es incorrecta, mientras que todas las demás son correctas. Por

(a)

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{oi(nil)} \rightarrow_{norm} \texttt{nil}}\,\mathsf{Rdc1_{oi1}} \quad \dfrac{}{\texttt{io(nil,}\,et\texttt{)} \rightarrow_{norm} et}\,\mathsf{Rdc1_{io1}}}{\texttt{oi(}et\texttt{)} \rightarrow_{norm} et}\,\mathsf{Rdc1_{oi2}} \quad \dfrac{\dfrac{}{\texttt{io(nil,}\ et\texttt{)} \rightarrow_{norm} et}\,\mathsf{Rdc1_{io1}}}{\texttt{io(}et\texttt{,}\ adri\texttt{)} \rightarrow_{norm} et\ et}\,\mathsf{Rdc1_{io3}}}{(\wp)\ \texttt{oi(}adri\ et\texttt{)} \rightarrow_{norm} et\ et}\,\mathsf{Rdc1_{oi2}} \quad T_2'' \quad \dfrac{\dfrac{}{S_n \Rightarrow^{\texttt{in}} \{S_1\}}\,\mathsf{Rl_{in}}}{S_n \Rightarrow_1 \{S_1\}}\,\mathsf{Top_{[\_]\_}} \quad \dfrac{\bigtriangledown' \ldots \bigtriangledown'}{S_1 \leadsto^4_{\mathcal{C}} \emptyset}\,\mathsf{Tr_1}}{\texttt{[l v] oi(adri et nil)} \leadsto^5_{\mathcal{C}} \emptyset}\,\mathsf{Red_1}$$

(b)

$$\dfrac{\dfrac{\dfrac{}{\texttt{["adri",15,""]nil : OList}}\,\mathsf{Mb_{ol2}}}{\texttt{["et",16,""]["adri",15,""]nil : OList}}\,\mathsf{Mb_{ol1}} \quad \dfrac{}{\texttt{"adri" buys "videogame" in S} \rightarrow_{norm} \texttt{false}}\,\mathsf{Rdc1_{buy2}}}{fails(\mathcal{C}, S_n)}\,\mathsf{Fail}$$

Figura 8.7: Árbol de depuración final para la búsqueda en la sección 7.1.7: Árbol principal (a) y T_2'' (b)

Figure 3.7: Final debugging tree for the search in Section 2.1.7

$$\dfrac{\dfrac{\dfrac{}{\texttt{oi(nil)} \to_{norm} \texttt{nil}}\,\mathsf{Rdc}_{1_{\texttt{oi1}}} \quad \dfrac{}{\texttt{io(nil,}\,et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\texttt{io1}}}}{(\S)\ \texttt{oi(}et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\texttt{oi2}}} \quad \dfrac{\dfrac{}{\texttt{io(nil,}\ et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\texttt{io1}}}}{\texttt{io(}et\texttt{,}\ adri\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\texttt{io3}}}}{\texttt{oi(}adri\ et\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\texttt{oi2}}}$$

Figura 8.8: Árbol de depuración tras una respuesta

$$\dfrac{\dfrac{\dfrac{}{\texttt{io(nil,}\ et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\texttt{io1}}}}{(\P)\ \texttt{io(}et\texttt{,}\ adri\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\texttt{io3}}}}{\texttt{oi(}adri\ et\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\texttt{oi2}}} \qquad \dfrac{\dfrac{}{\texttt{io(nil,}\ et\texttt{)} \to_{norm} et}\,\mathsf{Rdc}_{1_{\texttt{io1}}}}{\texttt{io(}et\texttt{,}\ adri\texttt{)} \to_{norm} et\ et}\,\mathsf{Rdc}_{1_{\texttt{io3}}}$$

Figura 8.9: Árbol de depuración tras dos (a) y tres respuestas (b)

tanto, lo indicamos escribiendo `(1 : no .)` y su subárbol, que se muestra en la figura 8.8 (y que se corresponde con la premisa izquierda de la raíz del árbol en la figura 8.7) es seleccionado como árbol actual de depuración.[7] Tras esta respuesta, la siguiente serie de preguntas está relacionada con los hijos de este nodo:

```
Question 1 :
Is this reduction (associated with the equation oi2) correct?

ordIns(["et",16,""]nil) -> ["et",16,""]nil

Question 2 :
Is this reduction (associated with the equation io3) correct?

insertOrd(["et",16,""]nil,["adri",15,""]) -> ["et",16,""]["et",16,""]nil
```

En vez de contestar cualquiera de estas preguntas, podemos cambiar a la otra estrategia de navegación, divide y pregunta, con el comando:

```
Maude> (divide-query strategy .)

Divide & Query strategy selected.

Is this reduction (associated with the equation oi2) correct?

ordIns(["et",16,""]nil) -> ["et",16,""]nil

Maude> (yes .)
```

En este caso el depurador ha seleccionado el nodo cuyo tamaño es el más cercano a la mitad del tamaño del árbol completo, que en este caso es el nodo marcado con (§) en la figura 8.8. Esta inferencia es correcta, y por tanto el subárbol que tiene como raíz este nodo se puede eliminar, actualizando el árbol mostrado en la figura 8.9(a). La siguiente pregunta, asociada al nodo(¶), es:

```
Is this reduction (associated with the equation io3) correct?
```

[7]También podríamos haber contestado `yes` a cualquier otra pregunta, lo que habría borrado dicho subárbol pero no haría avanzar el proceso de depuración. Por esta razón esta respuesta solo se recomienda, usando la estrategia arriba-abajo, para simplificar la presentación de las preguntas si se listan demasiadas.

```
insertOrd(["et",16,""]nil,["adri",15,""]) -> ["et",16,""]["et",16,""]nil

Maude> (no .)
```

La respuesta es `(no .)`, y por tanto el árbol se reduce al árbol mostrado en la figura 8.9(b). Dado que el depurador sabe que la raíz es errónea (porque el usuario lo señaló en la respuesta anterior) la pregunta está relacionada con el hijo de la raíz:

```
Is this reduction (associated with the equation io1) correct?

insertOrd(nil,["et",16,""]) -> ["et",16,""]nil

Maude> (yes .)
```

Con esta información sabemos que la raíz del árbol en la figura 8.9(b) contiene información errónea pero que todos sus hijos son correctos, y por tanto es el nodo defectuoso.[8] El depurador muestra la siguiente información:

```
The buggy node is:
insertOrd(["et",16,""]nil,["adri",15,""]) -> ["et",16,""]["et",16,""]nil
with the associated equation: io3
```

Por tanto, la ecuación errónea es `io3`, que se especificó en la sección 7.1.7 como:

```
 eq [io3] : insertOrd(E L, E') = E insertOrd(L, E) [owise] .
```

es decir, nos olvidamos de `E'` en la llamada recursiva. El código correcto para esta función es:

```
 eq [io3] : insertOrd(E L, E') = E insertOrd(L, E') [owise] .
```

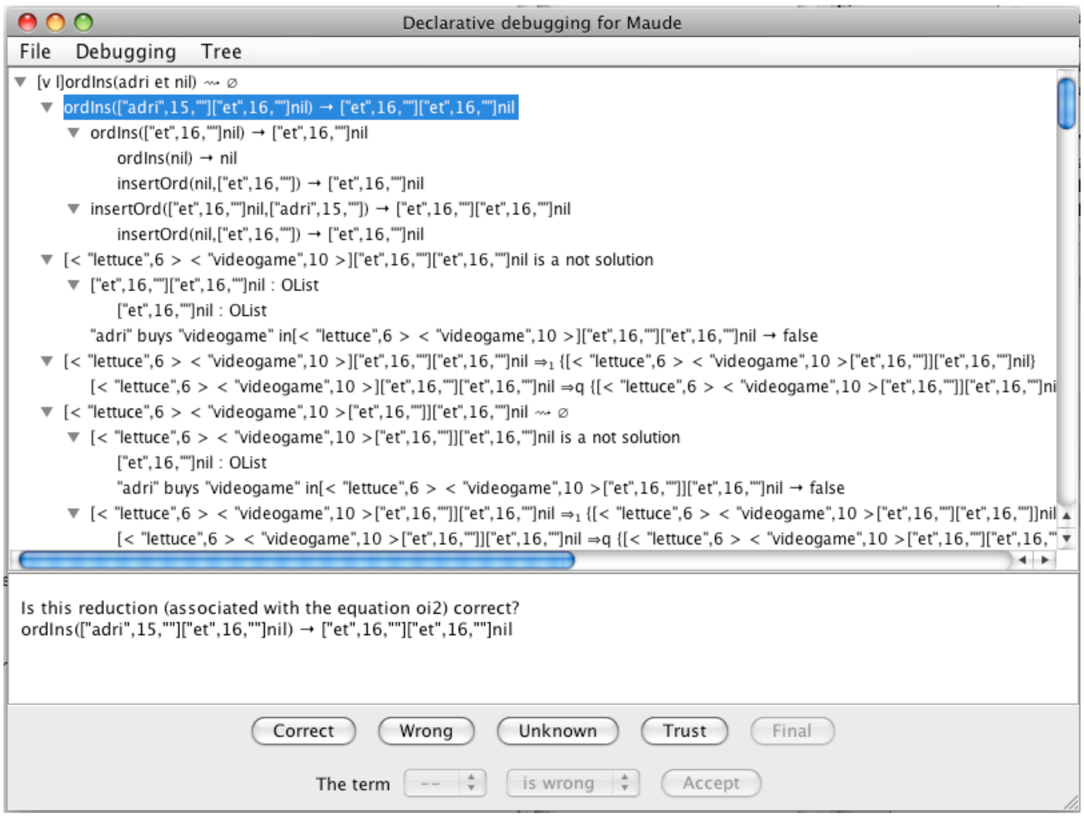
En la figura 8.10 mostramos cómo depurar este mismo ejemplo usando la interfaz gráfica de usuario. Además de las estrategias de navegación implementadas por la herramienta, la interfaz permite navegar libremente por el árbol y elegir diferentes nodos para ver las preguntas asociadas. En este caso hemos elegido un nodo que se corresponde con la raíz del árbol en la figura 8.8 y cuya reducción asociada es incorrecta. Usando el botón `Wrong` obtenemos el árbol mostrado en la figura 8.11, donde solo se muestra la información relevante para el proceso de depuración[9] y donde hemos seleccionado otro nodo incorrecto; una vez indicamos que es incorrecto y que sus hijos son incorrectos la interfaz detecta que es un nodo defectuoso, mostrando la información en la figura 8.12.

8.5 Contribuciones

En este capítulo hemos presentado un depurador declarativo para especificaciones en Maude. Las principales contribuciones de este trabajo son:

[8]Este nodo es el marcado con ($\star$) en la figura 8.6, indicando que era el nodo defectuoso.

[9]Hay otro comportamiento disponible al depurar con la interfaz: conservar todos los nodos, asociando a cada uno de ellos un color que indique su estado: correcto, erróneo, no sabe y no contestado.



```
▼ [v l]ordIns(adri et nil) ⇝ ∅
  ▼ ordIns(["adri",15,""]["et",16,""]nil) → ["et",16,""]["et",16,""]nil
    ▼ ordIns(["et",16,""]nil) → ["et",16,""]nil
        ordIns(nil) → nil
        insertOrd(nil,["et",16,""]) → ["et",16,""]nil
    ▼ insertOrd(["et",16,""]nil,["adri",15,""]) → ["et",16,""]["et",16,""]nil
        insertOrd(nil,["et",16,""]) → ["et",16,""]nil
  ▼ [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil is a not solution
    ▼ ["et",16,""]["et",16,""]nil : OList
        ["et",16,""]nil : OList
      "adri" buys "videogame" in[< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil → false
  ▼ [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil ⇒1 {[< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil}
      [< "lettuce",6 > < "videogame",10 >]["et",16,""]["et",16,""]nil ⇒q {[< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]ni
  ▼ [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ⇝ ∅
    ▼ [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil is a not solution
        ["et",16,""]nil : OList
        "adri" buys "videogame" in[< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil → false
    ▼ [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ⇒1 {[< "lettuce",6 > < "videogame",10 >["et",16,""]["et",16,""]]nil
        [< "lettuce",6 > < "videogame",10 >["et",16,""]]["et",16,""]nil ⇒q {[< "lettuce",6 > < "videogame",10 >["et",16,""]["et",16,""

Is this reduction (associated with the equation oi2) correct?
ordIns(["adri",15,""]["et",16,""]nil) → ["et",16,""]["et",16,""]nil
```

Figura 8.10: Depuración con la interfaz gráfica de usuario

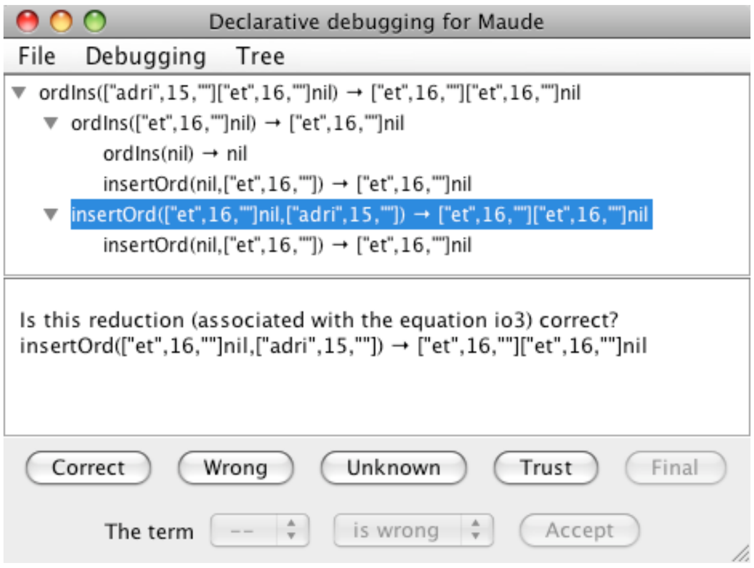
Depuración de respuestas erróneas. La herramienta es capaz de depurar respuestas erróneas en módulos tanto funcionales (reducciones e inferencias de tipo erróneas) como de sistema (reescrituras erróneas), donde todas las características de Maude, incluyendo atributos ecuacionales, `frozen` y `otherwise` se pueden utilizar en dichas especificaciones. El depurador indica la sentencia responsable del error en la especificación.

Depuración de respuestas perdidas. La herramienta puede también depurar respuestas perdidas en módulos funcionales (formas normales no totalmente reducidas y tipos mínimos mayores de lo esperado) y de sistema (conjuntos incompletos de términos alcanzables dada una condición y una cota en el número de pasos). Las causas detectadas en este tipo de depuración son sentencias erróneas y perdidas y condiciones de búsqueda erróneas; cuando una sentencia perdida es la causa del error el depurador es capaz de identificar el operador más externo que debe estar en el lado izquierdo de dicha sentencia.

Cálculo formal. Dado que los árboles usados en el proceso de depuración se obtienen usando un cálculo formal, somos capaces de probar la completitud y la corrección de la técnica.

Funcionalidad. El depurador incorpora muchas de las características presentes en otros depuradores declarativos:

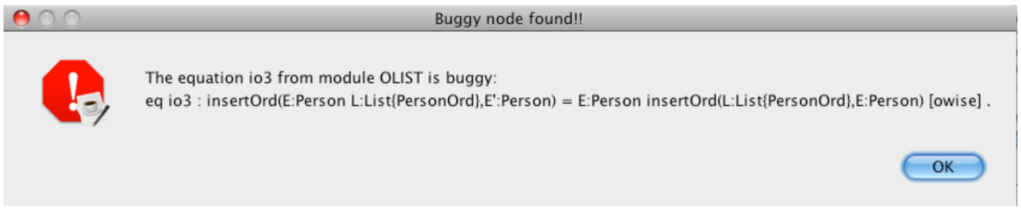
- Técnicas que acortan y mejoran el árbol de depuración, que nosotros hemos llamado *APT*.



```
▼ ordIns(["adri",15,""]["et",16,""]nil) → ["et",16,""]["et",16,""]nil
    ▼ ordIns(["et",16,""]nil) → ["et",16,""]nil
        ordIns(nil) → nil
        insertOrd(nil,["et",16,""]) → ["et",16,""]nil
    ▼ insertOrd(["et",16,""]nil,["adri",15,""]) → ["et",16,""]["et",16,""]nil
        insertOrd(nil,["et",16,""]) → ["et",16,""]nil
```

```
Is this reduction (associated with the equation io3) correct?
insertOrd(["et",16,""]nil,["adri",15,""]) → ["et",16,""]["et",16,""]nil
```

Figura 8.11: Interfaz gráfica de usuario tras una respuesta



```
The equation io3 from module OLIST is buggy:
eq io3 : insertOrd(E:Person L:List{PersonOrd},E':Person) = E:Person insertOrd(L:List{PersonOrd},E:Person) [owise] .
```

Figura 8.12: Información proporcionada por la interfaz gráfica de usuario

- Diferentes estrategias de navegación: arriba-abajo y divide y pregunta.
- Múltiples respuestas: además de las respuestas estándar *sí* y *no* permitimos un comando *deshacer* para volver al estado anterior, *no sabe* para saltarse la pregunta actual, y *el término n no es una solución/alcanzable* para indicar que un término en un conjunto no es una solución o no es alcanzable, haciendo que la herramienta dirija el proceso en esa dirección.
- Mecanismos de confianza para sentencias (antes de empezar la depuración y sobre la marcha), módulos, formas normales y tipos finales (esto último antes de empezar la depuración y sobre la marcha).
- Una interfaz gráfica de usuario, que facilita el proceso de depuración, mostrando el árbol de depuración y permitiendo al usuario navegarlo libremente.

Características originales. Hemos desarrollado una nueva característica: distintos tipos de árboles se pueden construir dependiendo de diferentes factores. Esta idea se ha usado para equilibrar árboles de depuración de distintas maneras en [45] (con la participación del autor de esta tesis), y se usa actualmente en el depurador DDJ.

Capítulo 9

Verificación Heterogénea

9.1 Estado del arte

Como se mencionó en la introducción, un sistema complejo normalmente requiere diferentes formalismos para definir sus partes. Por ejemplo, es habitual aplicar diferentes técnicas para especificar un sistema, sus bases de datos y las demostraciones sobre su comportamiento esperado. Dado que suponer que un único sistema proporcione todas las características necesarias para todos los posibles requisitos que puede necesitar un usuario es poco realista, los sistemas heterogéneos, que proporcionan diversos formalismos y combinaciones entre ellos, están ganando importancia en la actualidad, dado que permiten usar la herramienta adecuada para cada parte del sistema y combinarlas para construir la estructura completa. Muchas herramientas se han propuesto para tratar la especificación y verificación heterogénea:

UML Probablemente, el sistema mejor conocido es el Lenguaje Unificado de Modelado (UML por sus siglas en inglés) [6], un lenguaje diseñado para facilitar el proceso de desarrollo de software. Sin embargo, UML carece intencionadamente de una semántica formal y por tanto no es una herramienta formal.

OMDoc Una herramienta más formal es OMDoc [48], un lenguaje ontológico para matemáticas. Este lenguaje permite representar objetos como fórmulas en un lenguaje que extiende XML llamado OpenMath, sentencias como definiciones o demostraciones, y teorías y morfismos entre ellas.

IMPS El Sistema Interactivo de Demostración Matemática (IMPS por sus siglas en inglés) [34] está orientado a razonamiento matemático. Proporciona una base de datos de matemáticas (representada como una red de teorías axiomáticas unidas por morfismos de teorías) y un conjunto de herramientas para relacionarlas y modificarlas.

SpecWare Specware [47] es una herramienta para facilitar el desarrollo de software que permite al usuario especificar formalmente los requisitos y generar código para ellos, de tal manera que se demuestra que dicho código es correcto. Se puede usar como herramienta de diseño, para describir sistemas complejos; como una lógica, para describir formalmente los requisitos; y como un lenguaje de programación, para implementar los programas.

Prosper Prosper [26], que está basado en el demostrador de teoremas HOL98 [74], es una herramienta que proporciona distintos procedimientos de decisión y comprobadores de modelos, disponibles de una manera independiente del lenguaje por medio de bibliotecas. En la actualidad, están disponibles las bibliotecas para Java, C y ML.

Twelf Twelf [79] es un proyecto de investigación para el diseño, implementación y aplicación de marcos lógicos. Twelf proporciona el marco lógico LF [41], usado para describir lógicas, el lenguaje lógico con restricciones Elf y una interfaz en Emacs. Un demostrador inductivo de metateoremas está actualmente en desarrollo.

Delphin Delphin [83] es un lenguaje funcional organizado en dos niveles: el nivel de datos, en el que se pueden especificar sistemas de deducción; y el nivel computacional, donde dichos sistemas se manipulan.

Hets El Conjunto Heterogéneo de Herramientas (Hets por sus siglas en inglés) [64, 63, 66, 7], incorpora en un marco común varios demostradores y lenguajes de especificación y proporciona mecanismos para establecer relaciones entre ellos. En Hets cada lógica se representa como una institución y las traducciones se representan como comorfismos entre instituciones (véase la sección 7.2).

Para los objetivos de esta tesis estamos interesados en Hets por las siguientes razones:

- Es formal, por lo que permite al usuario razonar sobre las propiedades matemáticas de sus especificaciones, a diferencia de otras herramientas como UML u OMDoc, que solo formalizan algunas funciones (o ninguna).
- Mientras muchos enfoques son unilaterales, en el sentido de usar solo una lógica (y un solo demostrador de teoremas) para codificar el problema (como en el caso de Prosper y OMDoc), las especificaciones se pueden introducir en Hets en cualquiera de las lógicas integradas.
- Se centra en codificaciones entre lógicas, a diferencia de los enfoques de OMDoc, SpecWare o IMPS, que se centran en codificaciones entre teorías. Mientras lo primero permite razonar sobre distintos elementos en distintas lógicas, lo segundo solo permite razonar sobre elementos en la misma lógica.
- Una importante desventaja de Hets es que permite a los usuarios utilizar las lógicas implementadas, pero no añadir nuevas lógicas al sistema, ni razonar directamente sobre ellas o sus traducciones, como es el caso de Delphin, un marco lógico donde la sintaxis y la semántica de diferentes lógicas se puede representar, permitiendo al usuario razonar sobre ellas. Por esta razón, el proyecto Atlas Lógico e Integrador (LATIN por sus siglas en inglés) [49] pretende integrar estos dos enfoques, siendo el primer paso de este proyecto integrar LF y Hets, lo que permitirá al usuario trabajar con las lógicas integradas en Hets como datos normales.

El esquema general de Hets se presenta en la figura 9.1. Hets necesita, para cada herramienta que se quiera integrar, métodos de análisis sintáctico y estático que permitan traducir las especificaciones en cada herramienta a un marco común donde puedan interactuar entre ellas. Las relaciones entre herramientas se especifican en el grafo de lógicas, que tienen como lógica central el Lenguaje Común de Especificación Algebraica (Casl por sus siglas en inglés), un lenguaje cuyo desarrollo fue propuesto por la Iniciativa para un Marco Común para especificación y desarrollo algebraico (CoFI por sus siglas en inglés) [40], que perseguía unificar los diferentes lenguajes algebraicos disponibles, incorporando las principales características de cada uno de ellos y fijando la sintaxis y la semántica. El objetivo de esta iniciativa era crear un lenguaje para la especificación de requisitos funcionales;

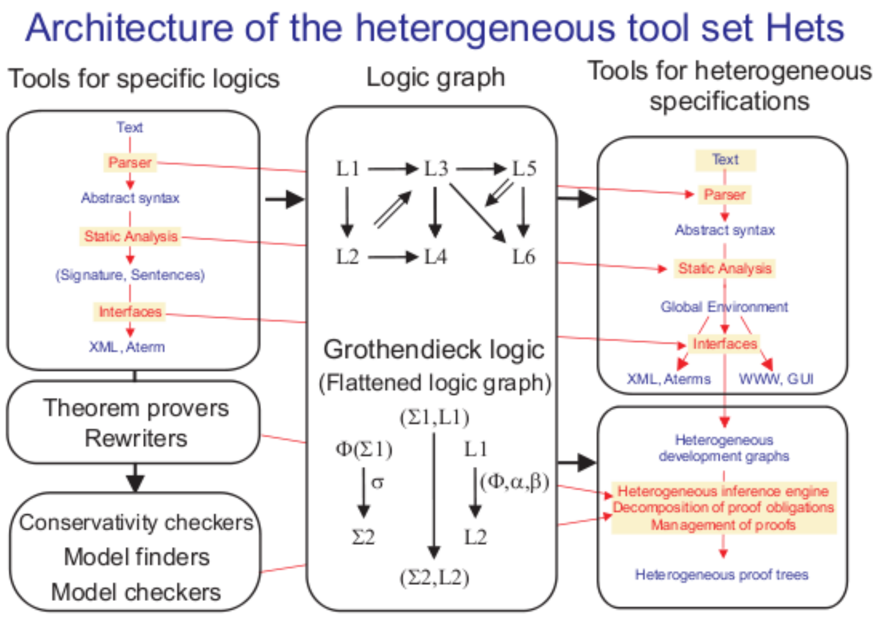


Figura 9.1: Arquitectura de Hets

el desarrollo formal de software; la relación entre especificaciones y requisitos informales y código implementado; el prototipado, la demostración de teoremas y la generación de casos de prueba de manera formal; y la interoperabilidad de herramientas. Siguiendo estas ideas Casl se diseñó como un lenguaje basado en lógica de primer orden e inducción por medio de restricciones sobre las constructoras.

Sin embargo, Casl no se pensó como un lenguaje autónomo, sino como el núcleo de una familia de lenguajes, algunos de los cuales se obtienen imponiendo ciertas restricciones a Casl mientras que otros se obtienen extendiéndolo. Por tanto, Hets es el resultado de ampliar esta idea para relacionar lenguajes independientes con Casl; de este manera es posible usar otras herramientas (y por tanto otras lógicas) con especificaciones en Casl y viceversa. Las lógicas actualmente conectadas con Casl, y por tanto soportadas por Hets, son:

Lógicas de propósito general: lógica proposicional, Casl y HetCasl (lógica de primer orden), QBF (Fórmulas Booleanas Cuantificadas) y TPLP/SoftFOL (lógica de primer orden con tipos blandos—*softly typed first-order logic* en inglés—). Más detalles sobre estos lenguajes están disponibles en [7].

Marcos lógicos: LF [41], un marco lógico para definir lógicas, y DFOL [96], una sublógica de LF para lógica de primer orden con tipos dependientes.

Ontologías y lenguajes con restricciones: OWL [55], el lenguaje de ontologías web; Common Logic [46], un marco para facilitar el intercambio de información; RelScheme [51], un lenguaje para bases de datos relacionales; y ConstraintCasl [63], una extensión de Casl con restricciones.

Sistemas reactivos: CspCasl [92], una extensión de Casl para el álgebra de procesos CSP; CoCasl [93], una extensión coalgebraica de Casl; y ModalCasl [60], que extiende Casl con operadores modales. Más detalles se pueden encontrar en [63].

Lenguajes de programación: Haskell [78], un lenguaje funcional perezoso.

Lógicas de herramientas específicas: Reduce [42], un sistema interactivo para cómputos algebraicos genéricos y DMU [43], una lógica "de fogueo" (*dummy logic* en inglés) para leer la salida de aplicaciones gráficas.

Demostradores: el demostrador interactivo de lógica de orden superior Isabelle [73]; el demostrador interactivo de lógica dinámica VSE [4]; un demostrador basado en Isabelle para CspCasl [75]; los resolutores de SAT zChaff [58] y MiniSat [33]; los demostradores automáticos de primer orden SPASS [102], Vampire [100], Darwin [35], KRHyper [103] y MathServe [104]; y los demostradores por tableau de lógicas descriptivas Pellet [19] y FaCT++ [99].

Desde el punto de vista de Hets, la integración de Maude como una nueva lógica presenta varias ventajas: sería el primer motor de reescritura integrado (en la actualidad, solo el motor de Isabelle está disponible, y está muy especializado en demostraciones en orden superior), permitiendo la ejecución de especificaciones implementadas en otros lenguajes, y permitiendo comprobación de modelos en lógica lineal temporal.

Desde el punto de vista de Maude, una integración en Hets es interesante porque permitiría al usuario demostrar propiedades de las especifiaciones usando los demostradores enumerados anteriormente. Varios enfoques dedicados a la demostración de teoremas han estado relacionados con Maude:

- Varios demostradores de teoremas y comprobadores se han implementado para Maude usando el propio Maude, usando sus capacidades de metaprogramación, entre ellas el Demostrador Inductivo de Teoremas (ITP por sus siglas en inglés) [22], cuya última versión permite diversas combinaciones de axiomas ecuacionales e inducción sobre constructoras, aunque solo permite pruebas en teorías Church-Rosser; el comprobador de terminación [30]; el comprobador de coherencia [32]; y el comprobador de la propiedad de Church-Rosser [31].

- Una traducción entre la lógica de HOL [74], un sistema de demostración basado en lógica de orden superior, y la lógica de Nuprl [1], un sistema para desarrollar sistemas software y teorías formales para matemáticas, se implementó en Maude [70] siguiendo un enfoque formal.

- En la actualidad, nos han informado de que un nuevo proyecto está dedicado a traducir especificaciones en Maude al demostrador PVS [76]. La transformación, implementada en Maude, está dirigida por ejemplos sobre especificaciones en sistemas de seguridad, y por tanto solo los elementos que aparecen en dichos ejemplos (un subconjunto de los módulos funcionales) son traducidos. Sin embargo, el sistema está aún en desarrollo y es de prever que nueva funcionalidad se añada en el futuro.

El principal inconveniente de estos enfoques es que, igual que en algunas herramientas enumeradas anteriormente, se concentran en Maude. Nuestra integración pretende mejorar estas propuestas permitiendo que especificaciones escritas en otras lógicas la usen.

9.2 Integrando Maude en Casl

El trabajo que es necesario llevar a cabo para conseguir esta integración es preparar Maude y su lógica subyacente para que pueda actuar como una expansión de Hets. Por parte de la semántica, esto significa que la lógica se tiene que formalizar como una *institución* [39, 97]. Una institución para lógica de reescritura ya fue estudiada en [77], pero presentaba el problema de usar la categoría discreta de signaturas, esto es, los únicos morfismos admitidos eran los morfismos identidad. Sin embargo, necesitamos los morfismos para trabajar con renombramientos, vistas y módulos parametrizados, y por tanto esta institución no es adecuada para nuestros propósitos. Por parte de la herramienta, debemos proporcionar mecanismos de análisis sintáctico y estático, de tal manera que las especificaciones se puedan traducir a un marco común, que en nuestro caso son los grafos de desarrollo, una representación gráfica de especificaciones estructuradas.

Antes de tratar de describir otra institución para la lógica subyacente a Maude, estudiamos las dos lógicas mejor conocidas para el sistema de transiciones que aparece en los módulos de sistema de Maude: la lógica de reescritura [56] y las álgebras preordenadas [36, 56]. Estos sistemas se distinguen principalmente en el tratamiento que hacen de las reescrituras: mientras que en la lógica de reescritura las reescrituras están etiquetadas y diferentes reescrituras entre dos estados (términos) se pueden distinguir (lo que corresponde a equipar cada conjunto soporte con una categoría de reescrituras), en las álgebras preordenadas lo único que importa es la existencia de una reescritura (lo que se corresponde con equipar cada conjunto soporte con un preorden de reescrituras).

Tras estudiar detenidamente Maude y la lógica de reescritura, decidimos que la implementación actual de Maude difiere de la lógica de reescritura definida en [56]. Las razones son:

1. En Maude, las etiquetas de las reglas no pueden ser (y no es necesario que sean) traducidas en los morfismos de signaturas. Esto significa que las vistas de Maude no conllevan morfismos de teorías en lógica de reescritura.

2. Aunque las etiquetas de las reglas se usan en las trazas de los contraejemplos, juegan un papel subsidiario; por ejemplo, no se pueden usar en la lógica lineal temporal del comprobador de modelos de Maude.

3. Las familias no se pueden declarar explícitamente en Maude, y por tanto es imposible tener una familia sin tipos.

Por estas razones no es necesario, en este momento, definir una institución para lógica de reescritura para integrar Maude en HETS, y por tanto usaremos álgebras preordenadas [36, 56]. En esta lógica, las reescrituras no se etiquetan ni se distinguen, sino que solo importa su existencia, lo que implica que las vistas de Maude no generan morfismos de teorías en la institución de las álgebras preordenadas. En las próximas secciones presentamos una institución para especificaciones en Maude basada en álgebras preordenadas y un comorfismo desde dicha institución a la institución de CASL, la lógica central de HETS. Más detalles se pueden encontrar en [24, 23].

9.2.1 Una institución para Maude

La institución que vamos a definir para Maude, que denotaremos como Maude^{pre}, es muy similar a la definida en el contexto de CafeOBJ [36, 28] para álgebras preordenadas (las diferencias están básicamente limitadas a los perfiles de los operadores que discutiremos después, lo cual es simplemente una cuestión de notación). Las signaturas de Maude^{pre} son tuplas $(K, F, \mathit{kind} : (S, \leq) \to K)$, donde K es un conjunto de *familias*, *kind* es una función que asigna una familia a cada *tipo* en el conjunto parcialmente ordenado $(S, \leq)$, y F es un conjunto de símbolos de función de la forma $F = \{F_{k_1 \ldots k_n \to k} \mid k_i, k \in K\} \cup \{F_{s_1 \ldots s_n \to s} \mid s_i, s \in S\}$ tal que si $f \in F_{s_1 \ldots s_n \to s}$, entonces hay un símbolo $f \in F_{\mathit{kind}(s_1) \ldots \mathit{kind}(s_n) \to \mathit{kind}(s)}$.

Nótese que no hay diferencias entre poner los perfiles de las operaciones con tipos en la signatura y la formulación original de Meseguer poniéndolo en las sentencias.

Dadas dos signaturas $\Sigma_i = (K_i, F_i, kind_i)$, $i \in \{1, 2\}$, un morfismo de signaturas $\phi : \Sigma_1 \to \Sigma_2$ es una función $\phi^{kind} : K_1 \to K_2$, una función entre tipos $\phi^{sort} : S_1 \to S_2$ tal que $\phi^{sort}; kind_2 = kind_1; \phi^{kind}$ y se preservan los subtipos, y una función $\phi^{op} : F_1 \to F_2$ que asocia símbolos de función de manera compatible con los tipos. Además la sobrecarga de símbolos de función se debe preservar, es decir, el identificador $\phi^{op}(\sigma)$ debe ser el mismo cuando se renombra el símbolo σ para tipos y para familias. Con la composición definida componente a componente, obtenemos la categoría de las signaturas.

Dada una signatura Σ, un modelo M interpreta cada familia k como un preorden $(M_k, \leq)$, cada tipo s como un subconjunto M_s de $M_{kind(s)}$ que está equipado con el preorden inducido, con M_s un subconjunto de $M_{s'}$ si $s < s'$, cada símbolo de función $f \in F_{k_1 \ldots k_n, k}$ como una función $M_f : M_{k_1} \times \ldots \times M_{k_n} \to M_k$ que tiene que ser monótona y tal que para cada símbolo de función f definido para tipos, su interpretación debe ser una restricción de la interpretación de la correspondiente función para familias. Dados dos Σ-modelos A y B, un homomorfismo de modelos es una familia de funciones $\{h_k : A_k \to B_k\}_{k \in K}$ que preserva el preorden, que es también un homomorfismo de álgebras y tal que $h_{kind(s)}(A_s) \subseteq B_s$ para cada tipo s.

Las sentencias de una signatura Σ son cláusulas de Horn construidas con tres tipos de átomos: átomos ecuacionales $t = t'$, átomos de pertenencia $t : s$ y átomos de reescritura $t \Rightarrow t'$, donde t, t' son Σ-términos y s es un tipo en S.[1]

Dado un Σ-modelo M y una valuación $\eta = \{\eta_k\}_{k \in K}$, es decir, una familia de funciones con tipos en K que asignan elementos en M a variables, M_t^η se define inductivamente de la manera habitual. Un átomo ecuacional $t = t'$ se satisface en M si $M_t^\eta = M_{t'}^\eta$, un átomo de pertenencia se satisface cuando M_t^η es un elemento de M_s y un átomo de reescritura $t \Rightarrow t'$ se satisface cuando $M_t^\eta \leq M_{t'}^\eta$. La satisfacción de átomos se extiende a satisfacción de sentencias de la manera obvia. Por último, usamos $M, \eta \models A$ para indicar que el modelo M satisface la sentencia A bajo la valuación η.

A continuación probamos que la condición de satisfacción se cumple para átomos, siendo la extensión a cláusulas de Horn inmediata. Para hacerlo, usaremos el siguiente lema:

Lema 1 *Dado un morfismo de signaturas $\sigma : \Sigma \to \Sigma'$, el cual induce la función $\sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ y el funtor $\_|_\sigma : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$, un Σ'-modelo M, los conjuntos de variables $X = \{x_1 : k_1, \ldots, x_l : k_l\}$ y $X' = \{x_1 : \sigma(k_1), \ldots, x_l : \sigma(k_l)\}$, con $k_i \in K$, $1 \leq i \leq l$, una valuación $\eta : X \to M|_\sigma$, la cual induce una valuación $\eta' : X' \to M$, con $\eta'(x) = \eta(x)$, y un Σ-término con variables en X, tenemos $M_{\sigma(t)}^{\eta'} = (M|_\sigma)_t^\eta$.*

Demostración. Por inducción estructural sobre t. Para $t = x$ una variable con familia es trivial porque $\sigma(x) = x$ y $\eta(x) = \eta'(x)$. De manera análoga, para $t = c$ una constante es trivial aplicando la definición de morfismo a operadores. Si $t = f(t_1, \ldots, t_n)$, entonces tenemos, por hipótesis de inducción, $M_{\sigma(t_i)}^{\eta'} = (M|_\sigma)_{t_i}^\eta$, para $1 \leq i \leq n$, y

$$\begin{array}{rcll}
M_{\sigma(f(t_1,\ldots,t_n))}^{\eta'} & = & M_{\sigma(f)(\sigma(t_1),\ldots,\sigma(t_n))}^{\eta'} & \text{(por definición de } \sigma \text{ sobre términos)} \\
 & = & M_{\sigma(f)}(M_{\sigma(t_1)}^{\eta'}, \ldots, M_{\sigma(t_n)}^{\eta'}) & \text{(significado del término en el modelo)} \\
 & = & M_{\sigma(f)}((M|_\sigma)_{t_1}^\eta, \ldots, (M|_\sigma)_{t_n}^\eta) & \text{(por hipótesis de inducción)} \\
 & = & (M|_\sigma)_f((M|_\sigma)_{t_1}^\eta, \ldots, (M|_\sigma)_{t_n}^\eta) & \text{(por definición de } \sigma \text{ sobre modelos)} \\
 & = & (M|_\sigma)_{f(t_1,\ldots,t_n)}^\eta. & \text{(significado del término en el modelo)}
\end{array}$$

[1]Nótese que esta definición es ligeramente más general que la de Maude, dado que las condiciones de reescritura están permitidas para ecuaciones y axiomas de pertenencia.

□

Podemos usar este resultado, combinado con la correspondencia biyectiva entre η y η', para comprobar la condición de satisfacción para una Σ-ecuación $t = t'$:

$$\begin{aligned}
M \models_{\Sigma'} \sigma(t = t') &\iff M \models_{\Sigma'} \sigma(t) = \sigma(t') \\
&\iff M, \eta' \models_{\Sigma'} \sigma(t) = \sigma(t') \text{ para todo } \eta' \\
&\iff M^{\eta'}_{\sigma(t)} = M^{\eta'}_{\sigma(t')} \text{ para todo } \eta' \\
&\iff (M|_\sigma)^\eta_t = (M|_\sigma)^\eta_{t'} \text{ para todo } \eta \\
&\iff M|_\sigma, \eta \models_\Sigma t = t' \text{ para todo } \eta \\
&\iff M|_\sigma \models_\Sigma t = t'
\end{aligned}$$

y de manera similar para asertos de pertenencia y reglas.

9.2.2 Un comorfismo de Maude a CASL

Presentamos ahora cómo codificar Maude en CASL, lo que se formalizará como un comorfismo entre instituciones. La idea principal en esta codificación es representar las reglas con un predicado binario y axiomatizarlo como un preorden compatible con las operaciones.

Para empezar, para definir el comorfismo necesitamos conocer la institución de CASL. A continuación describimos las principales características de dicha institución; más detalles están disponibles en [59, 66]. La institución se define en dos pasos: en primer lugar se introduce la lógica de primer orden heterogénea con funciones parciales, restricciones de generación de tipos e igualdad ($PCFOL^=$), y entonces la lógica de primer orden con subtipado, funciones parciales y restricciones de generación de tipos e igualdad ($SubPCFOL^=$) se describe en términos de $PCFOL^=$ [59]. Esta institución se compone de:

- Una *signatura con subtipado* $\Sigma = (S, TF, PF, P, \leq_S)$, donde S es un conjunto de tipos, TF y PF son dos familias, con tipos en $S^* \times S$, $TF = (TF_{w,s})_{w \in S^*, s \in S}$ y $PF = (PF_{w,s})_{w \in S^*, s \in S}$ de símbolos de función totales y parciales, respectivamente, tal que $TF_{w,s} \cap PF_{w,s} = \emptyset$, para cada $(w, s) \in S^* \times S$, $P = (P_w)_{w \in S^*}$ es una familia de símbolos de predicado, y $\leq_S$ una relación reflexiva y transitiva de subtipado en el conjunto S. Dadas dos signaturas $\Sigma = (S, TF, PF, P)$ y $\Sigma' = (S', TF', PF', P')$, un morfismo de signaturas $\sigma : \Sigma \to \Sigma'$ está compuesto por:

  – una función $\sigma^S : S \to S'$ que preserva la relación de subtipado,

  – una función $\sigma^F_{w,s} : TF_{w,s} \cup PF_{w,s} \to TF_{\sigma^{S^*}(w), \sigma^S(s)} \cup PF'_{\sigma^{S^*}(w), \sigma^S(s)}$ que asocia símbolos de función total con símbolos de función total y símbolos de función parcial con símbolos de función parcial, para cada $w \in S^*, s \in S$, y

  – una función $\sigma^P_w : P_w \to P'_{\sigma^{S^*}(w)}$ para cada $w \in S^*$.

  Las identidades y la composición se definen de manera directa.

  A cada signatura con subtipado $\Sigma = (S, TF, PF, P, \leq_S)$ le asociamos una signatura heterogénea $\hat{\Sigma}$, extensión de (S, TF, PF, P), la signatura heterogénea subyacente, con:

  – un símbolo sobrecargado *injection* de función total $inj : s \to s'$, para cada par de tipos $s \leq_S s'$,

  – un símbolo sobrecargado *projection* de función parcial $pr : s' \to ?\, s$, para cada par de tipos $s \leq_S s'$, y

- un símbolo sobrecargado de aridad 1 *membership* de predicado $\in^s\colon s'$, para cada par de tipos $s \leq_S s'$.

Los morfismos de signaturas $\sigma : \Sigma \to \Sigma'$ se extienden a morfismos de signaturas $\hat{\sigma} : \hat{\Sigma} \to \hat{\Sigma}'$ simplemente asociando las funciones y el predicado anteriores a sus homólogos en $\hat{\Sigma}'$.

Dada una signatura con subtipado $\Sigma = (S, TF, PF, P, \leq_S)$, definimos *relaciones de sobrecarga* (también llamados *órdenes monótonos*), $\sim_F$ y $\sim_P$, para símbolos de función y de predicado, respectivamente:

Sean $f : w_1 \to s_1, f : w_2 \to s_2 \in TF \cup PF$; entonces $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ si y solo si existe $w \in S^*$ con $w \leq_{S^*} w_1$ y $w \leq_{S^*} w_2$ y $s \in S$ con $s_1 \leq_S s$ y $s_2 \leq_S s$.

Sean $p : w_1, p : w_2 \in P$, entonces $p : w_1 \sim_P p : w_2$ si y solo si existe $w \in S^*$ con $w \leq_{S^*} w_1$ y $w \leq_{S^*} w_2$.

- Un conjunto de *sentencias subtipadas* en Σ, que se corresponde con sentencias heterogéneas en $\hat{\Sigma}$, esto es, fórmulas en lógica de primer orden cerradas heterogéneas en $\hat{\Sigma}$ o restricciones de generación de tipos (un tipo especial de fórmula que permite establecer cuáles son los símbolos de función usados como constructoras) en Σ. La traducción de sentencias por un morfismo de signatura subtipada es simplemente la traducción de la sentencia por un morfismo heterogéneo de signaturas en $\hat{\sigma}$.

- *Modelos M subtipados* en Σ son simplemente modelos heterogéneos en $\hat{\Sigma}$, compuestos por:

  - un conjunto soporte no vacío M_s para cada tipo $s \in S$,
  - una función parcial f_M de M_w a M_s para cada símbolo de función $f \in TF_{w,s} \cup PF_{w,s}$, $w \in S^*$, $s \in S$, siendo total la función si $f \in TF_{w,s}$, y
  - un predicado $p_M \subseteq M_w$ para cada símbolo de predicado $p \in P_w$, $w \in S^*$,

  satisfaciendo el siguiente conjunto de axiomas $\hat{J}(\Sigma)$:

  - $inj_{(s,s)}(x) \overset{e}{=} x$ (identidad), donde $\overset{e}{=}$ denota una ecuación existencial,
  - $inj_{(s,s')}(x) \overset{e}{=} inj_{(s,s')}(x) \implies x \overset{e}{=} y$ para $s \leq_S s'$ (inyectividad de la inmersión—*embedding-injectivity* en inglés—),
  - $inj_{(s',s'')}(inj_{s,s'}(x)) \overset{e}{=} inj_{(s,s'')}(x)$ para $s \leq_S s' \leq_S s''$ (transitividad),
  - $pr_{(s',s)}(inj_{(s,s')}(x)) \overset{e}{=} x$ para $s \leq_S s'$ (proyección),
  - $pr_{(s',s)}(x) \overset{e}{=} pr_{(s',s)}(y) \implies x \overset{e}{=} y$ para $s \leq_S s'$ (inyectividad de la proyección—*projection-injectivity* en inglés—),
  - $\in^s_{s'}(x) \iff pr_{(s',s)}(x)$ para $s \leq_S s'$ (pertenencia),
  - $inj_{(s',s)}(f_{w',s'}(inj_{s_1,s'_1}(x_1), \ldots, inj_{s_n,s'_n}(x_n))) =$ $inj_{(s'',s)}(f_{w'',s''}(inj_{(s_1,s''_1)}(x_1), \ldots, inj_{(s_n,s''_n)}(x_n)))$ for $f_{w',s'} \sim_F f_{w'',s''}$, donde $w \leq w', w''$, $s', s'' \leq s$, $w = s_1, \ldots, s_n$, $w' = s'_1, \ldots, s'_n$, y $w'' = s''_1, \ldots, s''_n$ (monotonía de la función—*function-monotonicity* en inglés—), y
  - $p_{w'}(inj_{(s_1,s'_1)}(x_1), \ldots, inj_{(s_n,s'_n)}(x_n)) \iff p_{w''}(inj_{(s_1,s''_1)}(x_1), \ldots, inj_{(s_n,s''_n)}(x_n))$ para $p_{w'} \sim_P p_{w''}$, donde $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, y $w'' = s''_1 \ldots s''_n$ (monotonía del predicado—*predicate-monotonicity* en inglés—).

- La *satisfacción* y la *condición de satisfacción* se heredan de la institución heterogénea. Básicamente, una fórmula φ se satisface en un modelo M si y solo si se satisface con respecto a todas las valuaciones de variables en M.

Con esta institución definida, podemos describir el comorfismo. Cada signatura de Maude $(K, F, \mathit{kind} : (S, \leq) \to K)$ se traduce a una teoría de Casl $((S', \leq', F, P), E)$, donde S' es la unión disjunta de K y S, $\leq'$ extiende la relación $\leq$ para tipos con pares $(s, \mathit{kind}(s))$ para cada $s \in S$, $\mathit{rew} \in P_{s,s}$ para todo $s \in S'$ es un predicado binario y E contiene axiomas estableciendo que para cada familia k, $\mathit{rew} \in P_{k,k}$ es un preorden compatible con las operaciones. Esto último significa que para cada $f \in F_{s_1 \ldots s_n, s}$ y cada x_i, y_i de tipo $s_i \in S'$, $i = 1, \ldots, n$, si $\mathit{rew}(x_i, y_i)$ se cumple, entonces $\mathit{rew}(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n))$ también se cumple.

Sean Σ_i, $i \in \{1, 2\}$ dos signaturas de Maude y sea $\varphi : \Sigma_1 \to \Sigma_2$ un morfismo de signaturas de Maude. Entonces su traducción $\Phi(\varphi) : \Phi(\Sigma_1) \to \Phi(\Sigma_2)$, denotada por ϕ, se define como sigue:

- para cada $s \in S$, $\phi(s) = \varphi^{sort}(s)$ y para cada $k \in K$, $\phi(k) = \varphi^{kind}(k)$.
- la condición de preservación del subtipado de ϕ se sigue de la condición análoga para φ.
- para cada símbolo de función σ, $\phi(\sigma) = \varphi^{op}(\sigma)$.
- *rew* permanece invariable.

La función de traducción de sentencias se obtiene en dos pasos. Mientras que los átomos ecuacionales no se modifican, los átomos de pertenencia $t : s$ se traducen a la notación de Casl t *in* s y los átomos de reescritura de la forma $t \Rightarrow t'$ se traducen a $\mathit{rew}(t, t')$. Por tanto, para cada sentencia en Maude de la forma $(\forall x_i : k_i) H \implies C$,[2] donde H es una conjunción de átomos de Maude y C es un átomo, se traduce como $(\forall x_i : k_i) H' \implies C'$, donde H' y C' se obtienen traduciendo todos los átomos de Maude como describimos anteriormente.

Dada una signatura de Maude Σ, un modelo M' de su teoría traducida (Σ', E) es asignado a un modelo M en Σ, donde:

- para cada familia k, se define $M_k = M'_k$ y la relación de preorden en M_k es *rew*;
- para cada tipo s, se define M_s para ser la imagen de M'_s bajo la inyección $\mathit{inj}_{s,\mathit{kind}(s)}$ generada por la relación de subtipado;
- para cada f definida sobre familias, sea $M_f(x_1, \ldots, x_n) = M'_f(x_1, \ldots, x_n)$ y para cada f definida sobre tipos con s el tipo del resultado, sea $M_f(x_1, \ldots, x_n) = \mathit{inj}_{s,\mathit{kind}(s)}(M'_f(x_1, \ldots, x_n))$. M_f es monótona porque los axiomas aseguran que M'_f es compatible con *rew*.

El reducto del homomorfismo de modelos es el esperado.

Sean Σ una signatura de Maude, M', N' dos modelos en $\Phi(\Sigma)$ (en Casl) y sea $h' : M' \to N'$ un homomorfismo de modelos. Denotamos $M = \beta_\Sigma(M')$ y $N = \beta_\Sigma(N')$ y definimos $h : M \to N$ como sigue: para toda familia k de Σ, $h_k = h'_k$ (lo cual es correcto porque el dominio y el codominio encajan, por definición de M y N). Necesitamos mostrar que h es en efecto un homomorfismo de modelos de Maude. Para ello, debemos probar tres cosas:

[2]La declaración de cualquier variable x con tipo s se sustituye por una variable x con familia $\mathit{kind}(s)$ y un axioma de pertenencia $x : s$.

1. h_k preserva el preorden para toda familia k.

   Asumamos $x \leq_k^M y$. Por definición, el preorden en M_k es el dado por *rew*, lo que significa que $M_{rew}(x, y)$ se cumple. Por la condición de homomorfismo para h' tenemos que $N_{rew}(h'(x), h'(y))$ se cumple, lo que significa por definición del preorden sobre N que $h'(x) \leq_k^N h'(y)$.

2. h es un homomorfismo de álgebras.

   Esto se obtiene directamente de la definición de M_f donde f es un símbolo de función y de la condición de homomorfismo para símbolos de función para h'.

3. para cualquier tipo s, $h_{kind(s)}(M_s) \subseteq N_s$.

   Por definición, $M_s = inj_{s,kind(s)}(M'_s)$. Por la condición de homomorfismo para $inj_{s,kind(s)}$, que es un símbolo explícito de función en CASL, tenemos que

   $$h_{kind(s)}(M_s) = h_{kind(s)}(inj_{s,kind(s)}(M'_s)) = inj_{s,kind(s)}(h_s(M'_s)).$$

   Dado que $h_s(M'_s) \subseteq N'_s$ por definición, tenemos que $inj_{s,kind(s)}(h_s(M'_s)) \subseteq inj_{s,kind(s)}(N'_s)$, lo que por definición es N_s.

9.3 Grafos de desarrollo

Para gestionar demostraciones, HETS usa grafos de desarrollo [62]. Estos grafos se pueden definir para cualquier institución y se usan para codificar especificaciones estructuradas en varias fases de su desarrollo. Básicamente, cada nodo del grafo representa una teoría, mientras que las aristas definen cómo hace uso cada teoría de las restantes. De esta manera, podemos representar especificaciones complejas representando cada componente (es decir, cada módulo) como un nodo del grafo de desarrollo y las relaciones entre ellos (es decir, importaciones y obligaciones de prueba) como aristas. En esta sección vamos a presentar las intuiciones tras los grafos de desarrollo, mientras que las definiciones formales se pueden encontrar en [24].

Definición 13 *Un* grafo de desarrollo *es un grafo dirigido y acíclico $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$.*

$\mathcal{N}$ es un conjunto de nodos. Cada nodo $N \in \mathcal{N}$ es un par (Σ^N, Φ^N) tal que Σ^N es una signatura y $\Phi^N \subseteq$ **Sen**(Σ^N) *es un conjunto de* **axiomas locales** *de N, (es decir, en el caso de Maude estos axiomas son las ecuaciones, axiomas de pertenencia y reglas declaradas—no importadas—en el módulo representado por el nodo).*

$\mathcal{L}$ es un conjunto de enlaces dirigidos, los llamados **enlaces de definición***, entre elementos de $\mathcal{N}$. Estamos interesados en dos tipos de enlaces desde un nodo M a un nodo N:*

- **global** *(denotado $M \overset{\sigma}{\Longrightarrow} N$), que indica que las sentencias en M son incluidas, renombrando con σ, en la teoría en N (dado que todos los nodos tienen una signatura, el renombramiento σ debe hacer la signatura en M igual a la correspondiente (sub)signatura en N).*

- **libres** *(denotado $M \underset{free}{\overset{\sigma}{\Longrightarrow}} N$), que indica que la signatura en M es libremente incluida, con el renombramiento σ, en la teoría en N.*

Además de estos enlaces hemos definido uno nuevo, denotado $M \xRightarrow[n.p.free]{\sigma} N$, que representa enlaces libres no persistentes y que será usado al tratar el modo de importación `protecting` de módulos de Maude. Sin embargo, estos enlaces solo se usan para especificaciones en Maude y por tanto no son estándar; en la próxima sección mostraremos cómo transformar estos enlaces y los nodos asociados en un nuevo grafo que solo usa construcciones estándar. Intuitivamente, estos enlaces indican que no se pueden añadir nuevos elementos a los tipos, aunque sí que se pueden añadir a las familias.

De manera complementaria a los enlaces de definición, los cuales *definen* las teorías de los nodos relacionados, introducimos la noción de *enlaces de teoremas*, de los que nos servimos para *postular* las relaciones entre las diferentes teorías. Los enlaces de teoremas son la estructura de datos principal para representar las obligaciones de prueba que aparecen en desarrollos formales. La idea es que un enlace de teorema entre los nodos M y N, denotado $M \overset{\sigma}{=\!=\!\Rightarrow} N$, incluye todas las sentencias en M, con un renombramiento σ, como *obligaciones de prueba* en N.

9.3.1 Restricciones de extensiones libres

Maude usa semánticas iniciales y libres intensivamente. Sin embargo, su semántica libre es diferente de la usada en Casl en que las extensiones libres de modelos tienen que ser persistentes solo en tipos y por tanto nuevos elementos de error se pueden añadir en la interpretación de familias. Intentos de diseñar una traducción a Casl de tal manera que los enlaces libres de Maude se tradujesen a los enlaces libres estándar han sido infructuosos, y por tanto decidimos introducir los enlaces libres no persistentes mencionados en la sección anterior. Para no romper el cálculo de los grafos de desarrollo, hemos normalizado estos enlaces reemplazándolos por un grafo de desarrollo semánticamente equivalente en Casl. La idea principal para hacer una extensión libre persistente es duplicar los tipos parámetros apropiadamente, de tal manera que el parámetro es siempre explícitamente incluido en la extensión libre. La transformación de enlaces libres no persistentes en Maude a enlaces libres persistentes en Casl se ilustra en la figura 9.2:

- M' and N' son las traducciones de Maude a Casl de M y N usando el comorfismo de la sección 9.2.

- M'' es una extensión de M' (donde el morfismo ι es un renombramiento para hacer la signatura distinta de M) donde la signatura se ha extendido con tipos $[s]$ para cada tipo $s \in \Sigma_M$, de tal manera que $s \leq [s]$ y $[s] \leq [s']$ si $s \leq s'$; los símbolos de función se han extendido con $f : [w] \to [s]$ para cada $f : w \to s \in \Sigma_M$; y, por último, se han añadido nuevos predicados *rew* para estos tipos.

- El nodo K tiene una signatura Σ^K que consiste en la signatura Σ^M unida de manera disjunta con una copia de Σ^M generada en el paso anterior por ι, denotada $\iota(\Sigma_M)$ (llamaremos $\iota(x)$ al correspondiente símbolo en esta copia para cada símbolo x de la signatura Σ^M) y aumentada con nuevas funciones $h : \iota(s) \to ?\, s$, para cada tipo s de Σ^M ($\to ?$ indica que es una función parcial) y $make_s : s \to \iota(s)$, para cada tipo s de la signatura fuente Σ del morfismo σ que etiqueta el enlace libre. Los axiomas para estos nuevos símbolos de función, generados siguiendo las ideas en [61], están fuera de los objetivos de este capítulo y son presentados en profundidad en [24].

- Un enlace libre de Casl une M'' con K. Este enlace está etiquetado por un morfismo $\sigma^\#$ que extiende σ con $[s] \mapsto [\sigma(s)]$ para cada $s \in \Sigma_M$.

$$\begin{tikzcd}
M \arrow[rr, Rightarrow, "\sigma", "n.p.free"'] \arrow[d, Rightarrow] & & N \arrow[dd, Rightarrow] \\
M' \arrow[d, Rightarrow, "\iota_N"] & & \\
M'' \arrow[r, Rightarrow, "\sigma^{\#}", "free"'] & K \arrow[r, Rightarrow, "\iota_N", "hide"'] & N'
\end{tikzcd}$$

Figura 9.2: Normalización de los enlaces libres de Maude

- Por último, un enlace de ocultación (básicamente, un enlace donde el morfismo es aplicado en la dirección opuesta al enlace—de N' a K en este caso—y donde algunos símbolos, considerados ocultos, no se incluyen en el destino) funciona como una inclusión que conecta K y N'.

El tratamiento genérico de restricciones de extensiones libres en CASL se encuentra en los axiomas del nodo K. La generación de estos axiomas ha sido implementada durante la integración de Maude en HETS, y por tanto la integración ofrece como efecto lateral la posibilidad de probar restricciones de extensiones libres en CASL y en todos los formalismos conectados.

9.3.2 Grafo de desarrollo: Un ejemplo

En esta sección presentamos cómo se traducen las especificaciones en Maude a grafos de desarrollo por medio de un ejemplo; más detalles están disponibles en [23]. Para mostrar que los grafos de desarrollo pueden representar grandes especificaciones, presentamos el grafo de desarrollo para el preludio de Maude en la figura 9.3. Dado que el grafo es demasiado grande para distinguir ningún detalle, merece la pena estudiar algunas zonas concretas más detenidamente. Nos vamos a centrar en las listas de números naturales, como se ve en la figura 9.4. Los ingredientes usados son: (i) los módulos `BOOL` y `NAT` para booleanos y números naturales; (ii) la teoría `TRIV` exigiendo la existencia de un tipo `Elt`; (iii) el módulo parametrizado `LIST`, que define listas genéricas de los elementos en `TRIV`; (iv) dos teorías `STRICT-TOTAL-ORDER` y `TOTAL-ORDER`, que importan algunas otras teorías y exigen que los elementos de tipo `Elt` tengan un orden total estricto y un orden total, respectivamente; (v) tres vistas `Nat`, `Nat<` y `Nat<=` que establecen que las teorías anteriores se satisfacen asignando el tipo `Nat` de los números naturales al tipo `Elt`; y (vi) una instanciación del módulo `LIST` con la vista `Nat`. El grafo de desarrollo se compone de:

- Un nodo para cada teoría de Maude (como `TOTAL-ORDER` o `STRICT-TOTAL-ORDER`) que contienen la correspondiente teoría (la signatura y las sentencias).

- Dos nodos por cada módulo. Uno contiene la teoría completa y representa la semántica laxa mientras que el otro, unido al primero con un enlace libre, contiene la misma signatura pero no tiene axiomas locales, por lo que representa el modelo libre de la teoría. Identificamos el primero con el nombre del módulo (como los nodos `NAT` o `LIST`), y el segundo con el identificador y un apóstrofo (`NAT'` y `LIST'`). En el ejemplo hemos omitido el subgrafo de `BOOL` para simplificar el grafo.

- Un enlace de definición para las importaciones en modo `including`, como `TRIV` siendo importado por `STRICT-WEAK-ORDER`, el cual es importado por `STRICT-TOTAL-ORDER`.

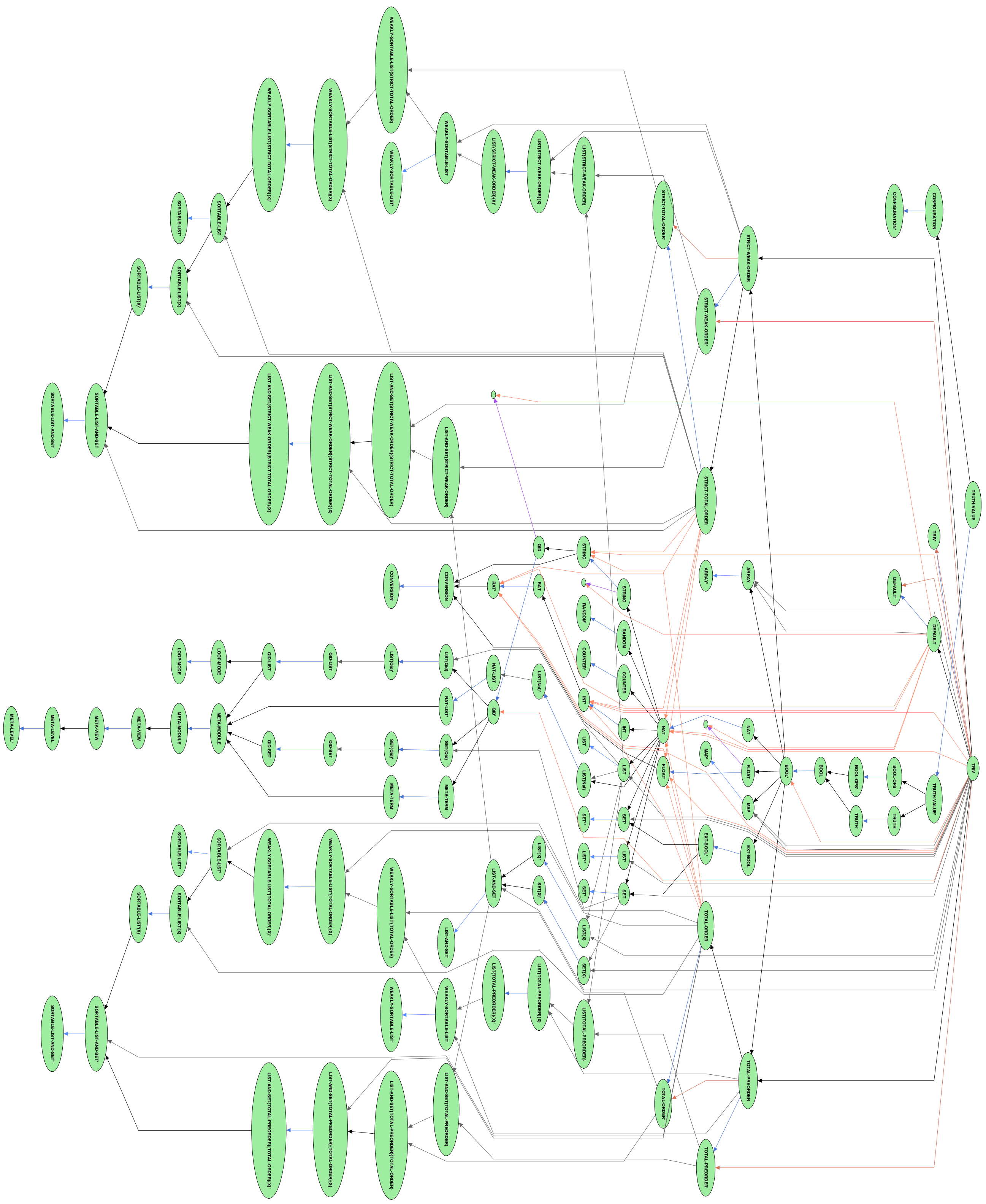


Figura 9.3: Grafo de desarrollo para el preludio de Maude

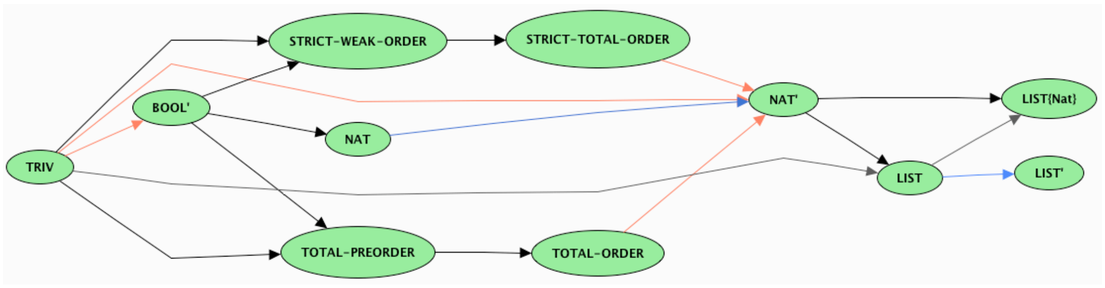


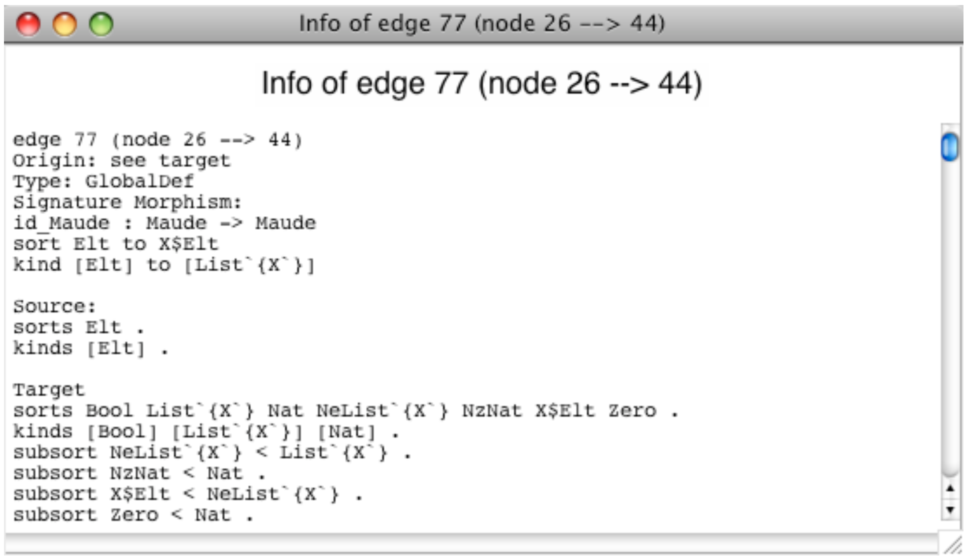
Figura 9.4: Grafo de desarrollo para las listas predefinidas de números naturales

- Un enlace de definición cuando una teoría se usa como parámetro formal, como `LIST`, que está parametrizado por un parámetro de la forma `X :: TRIV`. Este enlace está etiquetado, como se ve en la figura 9.5, con el morfismo `Elt` $\mapsto$ `X$Elt` (además, dado que el tipo `Elt` se ha declarado como un subtipo de `List{X}`, la familia de `Elt` se corresponde con la familia de `List{X}`), que cualifica al tipo con el nombre del parámetro.

- Un enlace de teorema para cada vista. Estos enlaces de teoremas tienen como fuente el nodo que representa la teoría usada como fuente de la vista y como destino el nodo que representa el modelo libre del módulo destino. Por ejemplo, la vista `Nat`, que tiene `TRIV` como fuente y `NAT` como destino, genera un enlace de teorema entre `TRIV` y `NAT'`.

- Un nuevo nodo por cada instanciación. Este nodo es el destino de varios enlaces de definición, uno desde el módulo parametrizado y otro desde el nodo que representa el modelo libre del destino de cada vista usada como parámetro. Por ejemplo, `LIST{Nat}` instancia el módulo `LIST` con la vista `Nat`. Es interesante ver que el enlace de `LIST` a `LIST{Nat}` está etiquetado, como se muestra en la figura 9.6, con el morfismo `X$Elt` $\mapsto$ `Nat`, `List{X}` $\mapsto$ `List{Nat}`, `NeList{X}` $\mapsto$ `NeList{Nat}`, que indica que el tipo `X$Elt` se corresponde con `Nat` y que el parámetro se ha instanciado con la vista `Nat`.

El principal interés de usar grafos de desarrollo, además de representar gráficamente especificaciones en Maude, es facilitar la verificación heterogénea. Un cálculo de grafos de desarrollo está integrado en Hets de tal manera que los grafos de desarrollo se transforman para tratar las obligaciones de prueba. En nuestro caso concreto, estamos interesados en la transformación `Automatic` disponible en el menú `Edit/Proofs`: entre otras transformaciones, "empuja" las obligaciones de prueba contenidas en los enlaces de teoremas a los nodos destino para que sean demostradas ahí. El resultado de aplicar este comando al grafo de desarrollo descrito anteriormente se muestra en la figura 9.7, donde el nodo `NAT'`, que Hets muestra en rojo, indica que tiene obligaciones de prueba pendientes. Ahora podemos usar la opción `Prove` en el menú del nodo, que abre la ventana mostrada en la figura 9.8, donde los diferentes axiomas (los de la teoría y los que tienen que ser probados) se pueden examinar, y donde se pueden seleccionar distintos demostradores de teoremas. En nuestro caso, estos axiomas se refieren a las ecuaciones que indican que los números naturales conforman un orden estricto total (cumplen las propiedades de transitividad, irreflexividad, transitividad de la incomparabilidad y totalidad) y un orden total (propiedades reflexiva, transitiva, total y antisimétrica). Estas propiedades se pueden probar automáticamente con SPASS, lo que permite a Hets transformar el nodo `Nat'` de rojo



Info of edge 77 (node 26 --> 44)

```
edge 77 (node 26 --> 44)
Origin: see target
Type: GlobalDef
Signature Morphism:
id_Maude : Maude -> Maude
sort Elt to X$Elt
kind [Elt] to [List`{X`}]

Source:
sorts Elt .
kinds [Elt] .

Target
sorts Bool List`{X`} Nat NeList`{X`} NzNat X$Elt Zero .
kinds [Bool] [List`{X`}] [Nat] .
subsort NeList`{X`} < List`{X`} .
subsort NzNat < Nat .
subsort X$Elt < NeList`{X`} .
subsort Zero < Nat .
```

Figura 9.5: Morfismo de `TRIV` a `LIST`

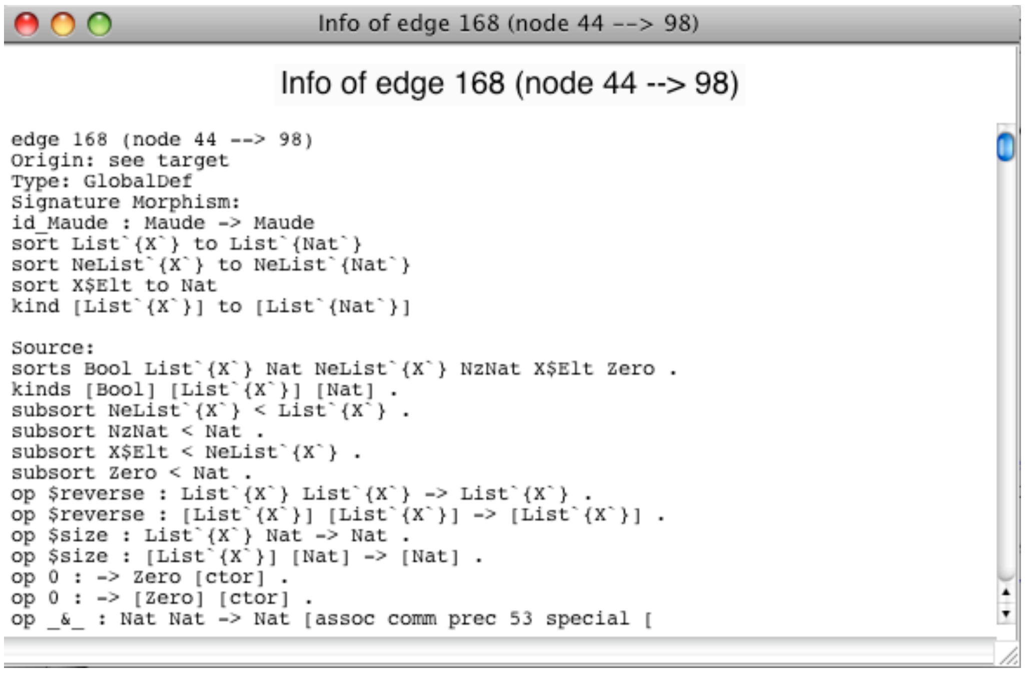
a verde, como se muestra en la figura 9.9, lo que indica que todas las obligaciones de prueba se han demostrado. Es interesante ver que estas propiedades no se han podido demostrar directamente sobre los números naturales definidos en Maude, porque todas las operaciones están implementadas en C++ y por tanto el módulo `NAT` no tiene ecuaciones. Las demostraciones se han llevado a cabo usando una biblioteca de Casl que define algunos de los módulos predefinidos de Maude de manera análoga a las especificaciones que esperaríamos en Maude,[3] que es cargada automáticamente cuando se detectan dichos módulos. Por ejemplo, la definición del operador `_<=_` en la biblioteca es:

```
forall m,n : Nat
     . 0 <= n = maudeTrue                          %(leq_def1_Nat)%
     . suc(n) <= 0 = maudeFalse                    %(leq_def2_Nat)%
     . suc(m) <= suc(n) = m <= n                   %(leq_def3_Nat)%
```

donde los valores `maudeTrue`, `maudeFalse` son un renombramiento de los valores de Maude `true` and `false` para evitar colisiones con las constantes de Casl del mismo nombre, y las cadenas en la derecha son las etiquetas de las ecuaciones. Por supuesto, esta prueba funcionaría igual si definimos las correspondientes ecuaciones en Maude y las usamos para demostrar las obligaciones de prueba en el nodo generado por el módulo correspondiente.

Desafortunadamente, no todas las obligaciones de prueba que pueden aparecen en una especificación en Maude se pueden demostrar automáticamente. En [24] probamos que invertir dos veces una lista da como resultado la lista original. Para ello, necesitamos las transformaciones para extensiones libres esbozadas en la sección 9.3.1 para normalizar el grafo de desarrollo y después aplicar las transformaciones estándar en el grafo resultante para demostrar las obligaciones de prueba.

[3]¡Esta implementación es un sistema heterogéneo! Combina la implementación de Hets en Haskell, un analizador sintáctico para especificaciones en Maude escrito en el propio Maude y bibliotecas en Casl.


Info of edge 168 (node 44 --> 98)
Info of edge 168 (node 44 --> 98)

```
edge 168 (node 44 --> 98)
Origin: see target
Type: GlobalDef
Signature Morphism:
id_Maude : Maude -> Maude
sort List`{X`} to List`{Nat`}
sort NeList`{X`} to NeList`{Nat`}
sort X$Elt to Nat
kind [List`{X`}] to [List`{Nat`}]

Source:
sorts Bool List`{X`} Nat NeList`{X`} NzNat X$Elt Zero .
kinds [Bool] [List`{X`}] [Nat] .
subsort NeList`{X`} < List`{X`} .
subsort NzNat < Nat .
subsort X$Elt < NeList`{X`} .
subsort Zero < Nat .
op $reverse : List`{X`} List`{X`} -> List`{X`} .
op $reverse : [List`{X`}] [List`{X`}] -> [List`{X`}] .
op $size : List`{X`} Nat -> Nat .
op $size : [List`{X`}] [Nat] -> [Nat] .
op 0 : -> Zero [ctor] .
op 0 : -> [Zero] [ctor] .
op _&_ : Nat Nat -> Nat [assoc comm prec 53 special [
```

Figura 9.6: Morfismo de `LIST` a `LIST{Nat}`

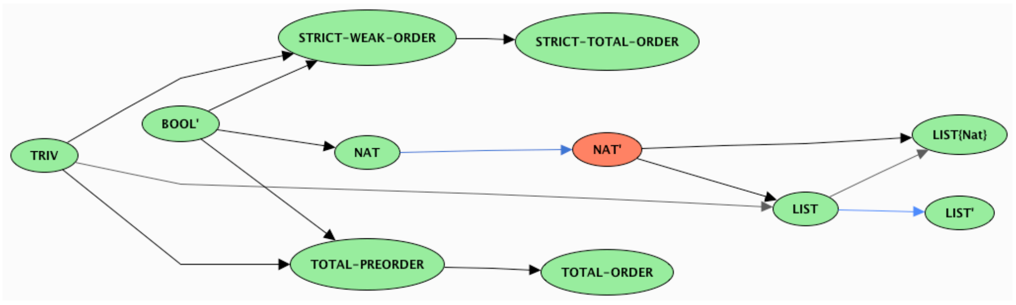


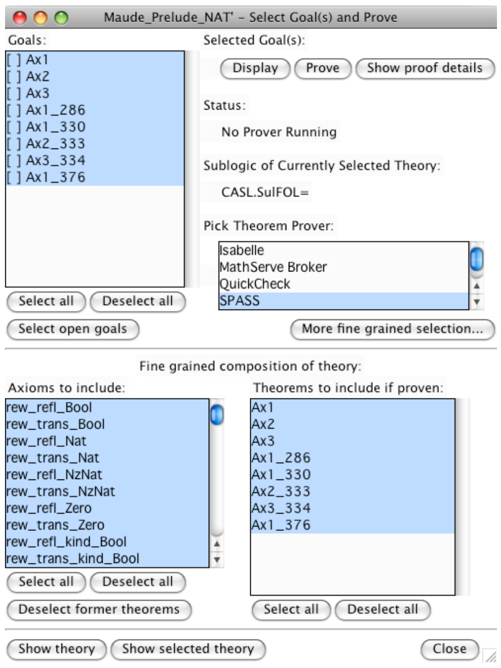
Figura 9.7: Grafo de desarrollo tras usar `Automatic`

9.4 Implementación

En esta sección vamos a describir brevemente los pasos de implementación necesarios para integrar Maude en HETS:

Sintaxis abstracta. En primer lugar, la sintaxis abstracta de las especificaciones en Maude se debe definir en Haskell. Esta sintaxis abstracta se basa en la gramática de Maude presentada en [20, capítulo 24].

Análisis sintáctico de Maude. También es necesario describir cómo analizar sintácticamente las especificaciones en Maude que se introducen en HETS, de forma que obtengamos un término construido con la sintaxis abstracta definida en el paso anterior. Este analizador sintáctico se puede implementar en el propio Maude usando su metanivel [20, capítulo 14], un módulo que permite al usuario usar entidades de Maude, como módulos, ecuaciones o reglas, como datos gracias a la eficiente implementación de las capacidades *reflexivas* de la lógica de reescritura [21].



Status:
No Prover Running

Sublogic of Currently Selected Theory:
CASL.SulFOL=

Figura 9.8: Ventana para las obligaciones de prueba

Usando esta característica hemos desarrollado una función que recibe un módulo y devuelve una lista de identificadores que representan un término en la sintaxis abstracta, que puede ser leído por Haskell gracias a que los tipos de datos derivan la clase `Read`.

Lógica. Una vez hemos traducido los módulos de Maude a la sintaxis abstracta, tenemos que implementar las clases de tipos de Haskell `Language` y `Logic` definidas en Hets, que definen los tipos necesarios para representar cada lógica como una institución y comorfismos entre ellas. Para hacerlo, debemos relacionar cada elemento en dichas clases (signaturas, morfismos, sentencias, etc.) con los elementos correspondientes en la sintaxis abstracta, siguiendo la teoría desarrollada en las secciones anteriores. Más detalles sobre este paso están disponibles en [50], que está dedicado a este paso.

Grafo de desarrollo. Dadas las opciones recibidas desde la línea de comandos y la ruta del fichero de Maude que tiene que ser analizado, debemos calcular en Haskell el grafo de desarrollo asociado. De hecho, construimos dos grafos de desarrollo diferentes, el primero con la información en el preludio de Maude y otro con la especificación del usuario, siguiendo en ambos casos las ideas descritas en las secciones anteriores.

Comorfismo. Dada una signatura de Maude y una lista de sentencias de Maude, debemos implementar en Haskell la traducción a la signatura de Casl y las sentencias de Casl.

Restricciones de extensiones libres. Por último, hemos implementado en Haskell cómo se generan, dada una signatura en Casl y un conjunto de sentencias en Casl, las restricciones de extensiones libres descritas en la sección 9.3.1.

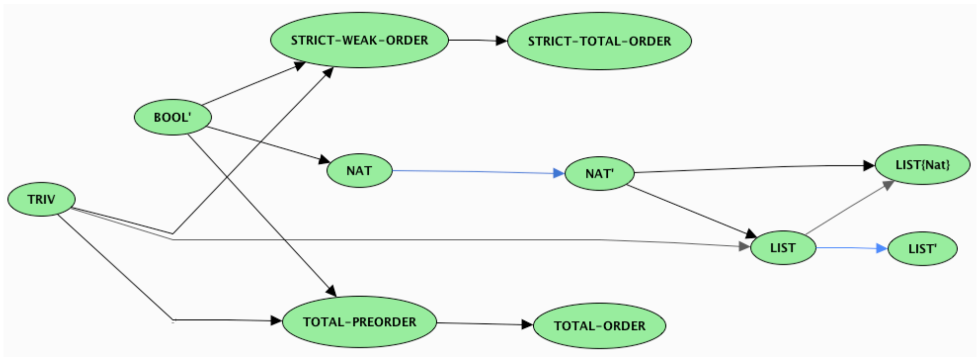


Figura 9.9: Grafo de desarrollo tras descartar las obligaciones de prueba

9.5 Contribuciones

En este capítulo hemos presentado una integración de Maude en Hets. Las principales aportaciones de este trabajo son:

Institución y comorfismo. Hemos definido una institución para Maude. Esta institución está basada en álgebras preordenadas, dado que una institución para la lógica de reescritura no era necesaria por el momento, y usa como sentencias ecuaciones, axiomas de pertenencia y reglas, a diferencia de la propuesta en [17], que solo usaba reglas incondicionales, y permite morfismos de signaturas no triviales, mejorando la propuesta en [77], que utilizaba una categoría discreta para las signaturas. Hemos definido también un comorfismo desde esta institución a la institución de Casl, la lógica central de Hets, que combina lógica de primer orden con inducción.

Grafo de desarrollo. Los módulos, teorías, vistas y mecanismos de estructuración de Maude se han incorporado a los grafos de desarrollo de Hets. De esta manera, podemos representar especificaciones en Maude como grafos de desarrollo, donde los módulos y las teorías se representan como nodos y las importaciones y las vistas como enlaces entre estos nodos.

Restricciones de extensiones libres. Hemos desarrollado, al nivel de Casl, una transformación que nos permite demostrar restricciones de extensiones libres para aquellas especificaciones que usan los enlaces libres estándar disponibles en Hets y cuenta con una traducción a Casl. Sin embargo, este no es el caso de Maude, que requiere un trato especial en el caso de estas restricciones. Por esta razón, hemos introducido un nuevo tipo de enlace en el cálculo de los grafos de desarrollo, que es después normalizado en los enlaces libres habituales para permitir usar la transformación anteriormente mencionada.

Demostraciones. Como "corolario" de las contribuciones anteriores, hemos integrado Maude en Hets, lo que permite demostrar propiedades de especificaciones en Maude con los demostradores ya integrados en Hets.

Capítulo 10

Conclusiones y trabajo futuro

Las contribuciones de esta tesis son:

- Hemos presentado un depurador declarativo para especificaciones en Maude. Este depurador permite al usuario depurar respuestas erróneas y perdidas causadas tanto por sentencias erróneas y perdidas como por condiciones de búsqueda erróneas. En esta herramienta se ha dedicado un gran esfuerzo a mejorar la usabilidad, proporcionando técnicas para mejorar y acortar el árbol de depuración, diferentes estrategias de navegación, múltiples respuestas, mecanismos de confianza, una interfaz gráfica de usuario, y una aportación original: la posibilidad de construir diferentes árboles de depuración dependiendo de la complejidad de la especificación y el conocimiento de la misma por parte del usuario. Por último, dado que los árboles de depuración se construyen siguiendo un cálculo formal, hemos demostrado la corrección y completitud de la técnica.

- Hemos integrado Maude en HETS, lo que nos permite usar las herramientas de HETS, y especialmente sus demostradores de teoremas, con especificaciones en Maude. Esta integración se ha llevado a cabo (i) definiendo una institución para Maude y (ii) traduciendo los mecanismos de estructuración de Maude a grafos de desarrollo. Además, también hemos implementado una transformación que permite demostrar restricciones de extensiones libres en cualequier teoría (no necesariamente en especificaciones en Maude).

El trabajo presentado en esta tesis nos ofrece una buena base para futuras extensiones que nos permitan mejorar su usabilidad y generalidad. Como trabajo futuro para nuestro depurador, planeamos añadir nuevas estrategias de depuración como las presentadas en [94], las cuales tienen en cuenta el número de posibles errores distintos en cada subárbol, en lugar de su tamaño. Además, la versión actual de la herramienta permite al usuario introducir un módulo correcto pero posiblemente incompleto para acortar la sesión de depuración. Ahora pretendemos añadir un nuevo comando para introducir módulos *completos* (es decir, le haríamos saber al sistema que todas las inferencias correctas que se pueden hacer en la especificación que está siendo depurada se pueden obtener en este módulo completo), lo que reduciría en gran medida el número de preguntas hechas al usuario.

Actualmente estamos trabajando en un generador de casos de prueba para especificaciones en Maude que, combinado con el depurador, permitiría al usuario probar y depurar sus especificaciones con una sola herramienta. El primer paso en el desarrollo de este proyecto ha sido el desarrollo de un generador de casos de prueba para módulos funcionales de Maude [84], que es capaz de generar casos de prueba para dichas especificaciones y comprobar su corrección con respecto a un módulo correcto o de elegir un subconjunto representativo de dichos casos de prueba, usando distintas estrategias, para que sean comprobados por el usuario; dado que estos procesos son muy costosos también presentamos varias técnicas de confianza para mejorarlos. En este momento estamos trabajando para mejorar el rendimiento de este generador de casos de prueba, aplicando técnicas de estrechamiento (*narrowing* en inglés) y usando arquitecturas distribuidas, y extendiendo la herramienta para probar módulos de sistema de Maude; dado que estos módulos no tienen por qué ser terminantes ni confluentes, la generación de casos de prueba para este tipo de especificaciones es totalmente diferente de la usada para módulos funcionales.

Como trabajo futuro de la integración en HETS, dado que las demostraciones interactivas no son fáciles de realizar, pretendemos mejorarlas adaptando estrategias automáticas de inducción como la ondulación (*rippling* en inglés) [29]. Además, pretendemos usar el demostrador automático de primer orden SPASS para pruebas con inducción integrándole estrategias directamente en HETS.

También estamos estudiando los posibles comorfismos de CASL a Maude. Para definirlos, debemos distinguir si las fórmulas en la teoría fuente son confluentes y terminantes o no. En el primer caso, pretendemos demostrar las propiedades con los comprobadores de terminación [30] y confluencia [31] de Maude y traducir las fórmulas como ecuaciones, cuya ejecución en Maude es más eficiente, mientras que en el segundo caso traduciríamos las fórmulas como reglas.

Por último, planeamos relacionar la lógica modal de HETS y los modelos de Maude para usar el comprobador de modelos para lógica lineal temporal de Maude [20, capítulo 13].

Bibliography

[1] S. F. Allen, M. Bickford, R. L. Constable, R. Eaton, C. Kreitz, L. Lorigo, and E. Moran. Innovations in computational type theory using Nuprl. *Journal of Applied Logic*, 4(4):428–469, 2006.

[2] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. An integrated framework for the diagnosis and correction of rule-based programs. *Theoretical Computer Science*, 411(47):4055–4101, 2010.

[3] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[4] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer*, 3(1):66–77, 2009.

[5] S. Autexier, D. Hutter, T. Mossakowski, and A. Schairer. The development graph manager MAYA. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology, AMAST 2002*, Lecture Notes in Computer Science, pages 495–501. Springer, 2002.

[6] T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, editors. *Proceedings of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference*, volume 3273 of *Lecture Notes in Computer Science*. Springer, 2004.

[7] M. Bidoit and P. D. Mosses. CASL *User Manual*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004.

[8] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[9] B. Braßel. The debugger B.I.O. `http://www-ps.informatik.uni-kiel.de/currywiki/tools/oracle_debugger`.

[10] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1):386–414, 2006.

[11] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In S. Antoy and M. Hanus, editors, *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia*, pages 8–13. ACM Press, 2005.

[12] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In F. J. López-Fraguas, editor, *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.

[13] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of membership equational logic specifications. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 174–193. Springer, 2008.

[14] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. In G. Roşu, editor, *Proceedings of the 7th International Workshop on Rewriting Logic and its Applications, WRLA 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 63–81. Elsevier, 2009.

[15] R. Caballero and M. Rodríguez-Artalejo. DDT: A declarative debugging tool for functional-logic languages. In Y. Kameyama and P. J. Stuckey, editors, *Proceedings 7th International Symposium on Functional and Logic Programming, FLOPS 2004, Nara, Japan*, volume 2998 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[16] R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative diagnosis of missing answers in constraint functional-logic programming. In J. Garrigue and M. V. Hermenegildo, editors, *Proceedings of 9th International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2008.

[17] M. V. Cengarle. The rewriting logic institution. Technical Report 9801, Institut für Informatik, Ludwig-Maximilians-Universität München, May 1998.

[18] O. Chitil and Y. Luo. Structure and properties of traces for functional programs. In I. Mackie, editor, *Proceedings of the Third International Workshop on Term Graph Rewriting, TERMGRAPH 2006*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 39–63. Elsevier, 2007.

[19] Clark & Parsia. Pellet: OWL 2 reasoner for Java. `http://clarkparsia.com/pellet/`.

[20] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[21] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[22] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from the 5th Spanish Conference on Programming and Languages, PROLE 2005.

[23] M. Codescu, T. Mossakowski, A. Riesco, and C. Maeder. Integrating Maude into Hets. Technical Report SIC-07/10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2010.

[24] M. Codescu, T. Mossakowski, A. Riesco, and C. Maeder. Integrating Maude into Hets. In M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2011.

[25] T. Davie and O. Chitil. Hat-Delta: One right does make a wrong. In H. Nillsson, editor, *7th Symposium on Trends in Functional Programming, TFP 2006*. Intellect, 2006.

[26] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):189–210, 2002.

[27] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990.

[28] R. Diaconescu. *Institution-independent Model Theory*. Birkhäuser Basel, 2008.

[29] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2004.

[30] F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, IJCAR 2008, Sydney, Australia, August 12-15*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.

[31] F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In P. C. Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2010.

[32] F. Durán and J. Meseguer. A Maude coherence checker tool for conditional order-sorted rewrite theories. In P. C. Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 86–103. Springer, 2010.

[33] N. Een and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[34] W. M. Farmer, J. D. Guttman, and F. J. Thayer. IMPS: An interactive mathematical proof system. *Computer Science Journal of Automated Reasoning*, 11(2):213–248, 1993.

[35] A. Fuchs. Darwin: A theorem prover for the model evolution calculus. Master's thesis, University of Koblenz-Landau, 2004.

[36] K. Futatsugi and R. Diaconescu. *CafeOBJ Report.* World Scientific, AMAST Series, 1998.

[37] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications, RTA 1998*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[38] J. Goguen and G. Roşu. Institution morphisms. *Formal Aspects of Computing*, 13:274–307, 2002.

[39] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.

[40] C. T. T. Group. CoFI, the common framework initiative for algebraic specification and development. `http://www.cofi.info`.

[41] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, 1993.

[42] A. C. Hearn. REDUCE User's manual – version 3.8, February 2004.

[43] IBM. CATIA V5 system requirements, 2007.

[44] D. Insa and J. Silva. An algorithmic debugger for Java. In M. Lanza and A. Marcus, editors, *Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010*, pages 1–6. IEEE Computer Society, 2010.

[45] D. Insa, J. Silva, and A. Riesco. Balancing execution trees. In V. M. Gulías, J. Silva, and A. Villanueva, editors, *Proceedings of the 10th Spanish Workshop on Programming Languages, PROLE 2010*, pages 129–142. Ibergarceta Publicaciones, 2010.

[46] ISO/IEC. Information technology — common logic (CL): a framework for a family of logic-based languages. International Standard ISO/IEC 24707:2007, 2007.

[47] Kestrel Development Corporation. Specware 4.2 user manual, 2009. `http://www.specware.org/`.

[48] M. Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics (SIGSAM)*, 34(2):43–48, 2000.

[49] M. Kohlhase, T. Mossakowski, and F. Rabe. The LATIN project. `https://trac.omdoc.org/LATIN`.

[50] M. Kühl. Integrating Maude into Hets. Master's thesis, Universität Bremen, 2010.

[51] O. Kutz, D. Lucke, and T. Mossakowski. Heterogeneously structured ontologies - integration, connection, and refinement. In T. Meyer and M. A. Orgun, editors, *Knowledge Representation Ontology Workshop, KROW 2008*, volume 90 of *Conferences in Research and Practice in Information Technology*, pages 41–50. Australian Computer Society, 2008.

[52] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[53] W. Lux. *Münster Curry User's Guide, 0.9.10 edition, 2006.*

[54] I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.

[55] D. L. McGuinness and F. van Harmelen. OWL 2 web ontology language document overview, 2009. `http://www.w3.org/TR/owl-features/`.

[56] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[57] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT 1997, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[58] M. W. Moskewicz and C. F. Madigan. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.

[59] T. Mossakowski. Relating Casl with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[60] T. Mossakowski. ModalCasl - specification with multi-modal logics. Language summary, 2004.

[61] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Technical report, Universität Bremen, 2005. Habilitation thesis.

[62] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - proof management for structured specifications. *Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques*, 67(1-2):114–145, 2006.

[63] T. Mossakowski, C. Maeder, M. Codescu, and D. Lücke. Hets user guide –version 0.97–. Technical report, DFKI GmbH, Formal Methods for Software Development, February 2011.

[64] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.

[65] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In B. Beckert, editor, *VERIFY 2007, 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 119–135, 2007.

[66] P. Mosses, editor. Casl *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

[67] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.

[68] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[69] L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog debugging environment. Technical Report 88/31, Department of Computer Science, University of Melbourne, Australia, June 1989. `http://www.cs.mu.oz.au/~lee/papers/nude/`.

[70] P. Naumov, M.-O. Stehr, and J. Meseguer. The HOL/NuPRL proof translator (a practical approach to formal interoperability). In R. J. Boulton and P. B. Jackson, editors, *Proceedings of the 14th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2001*, volume 2152 of *Lecture Notes in Computer Science*, pages 329–345. Springer, 2001.

[71] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[72] H. Nilsson and P. Fritzson. Algorithmic debugging of lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[73] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[74] M. Norrish and K. Slind. The HOL system tutorial, September 2010.

[75] L. O'Reilly, M. Roggenbach, and Y. Isobe. CSP-CASL-Prover: A generic tool for process and data refinement. In *Proceedings of the Eighth International Workshop on Automated Verification of Critical Systems, AVoCS 2008*, volume 250(2) of *Electronic Notes in Theoretical Computer Science*, pages 69 – 84, 2009.

[76] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction, CADE*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer, 1992.

[77] M. Palomino. A comparison between two logical formalisms for rewriting. *Theory and Practice of Logic Programming*, 7(1-2):183–213, 2007.

[78] S. Peyton Jones. *Haskell 98 language and libraries: the Revised Report*. Cambridge University Press, 2003.

[79] F. Pfenning and C. Schuermann. Twelf user's guide 1.4, 2002.

[80] B. C. Pierce. *Basic Category Theory for Computer Scientists*. The MIT Press, 1991.

[81] B. Pope. Declarative debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming - 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer, 2005.

[82] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

[83] A. Poswolsky and C. Schürmann. System description: Delphin – a functional programming language for deductive systems. In A. Abel and C. Urban, editors, *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice, LFMTP 2008*, volume 228 of *Electronic Notes in Computer Science*, pages 113–120. Elsevier, 2009.

[84] A. Riesco. Test-case generation for Maude functional modules. In *Proceedings of the 20th International Workshop on Algebraic Development Techniques, WADT 2010*, Lecture Notes in Computer Science. Springer, 2011. To appear.

[85] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009. `http://maude.sip.ucm.es/debugging`.

[86] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of rewriting logic specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques, WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2009.

[87] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative debugging of missing answers for Maude specifications. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 277–294. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[88] A. Riesco, A. Verdejo, and N. Martí-Oliet. Enhancing the debugging of Maude specifications. In P. C. Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2010.

[89] A. Riesco, A. Verdejo, and N. Martí-Oliet. A complete declarative debugger for Maude. In M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2011.

[90] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. A declarative debugger for Maude. In J. Meseguer and G. Roşu, editors, *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, AMAST 2008*, volume 5140 of *Lecture Notes in Computer Science*, pages 116–121. Springer, 2008.

[91] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. To appear.

[92] M. Roggenbach. CSP-CASL: a new integration of process algebra and algebraic specification. *Theoretical Computer Science*, 354(1):42–71, 2006.

[93] L. Schröder and T. Mossakowski. Coalgebraic modal logic in CoCasl. In J. L. Fiadeiro, editor, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006*, volume 4409 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2007.

[94] J. Silva. A comparative study of algorithmic debugging strategies. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[95] J. Silva. *Debugging Techniques for Declarative Languages: Profiling, Program Slicing, and Algorithmic Debugging.* PhD thesis, Technical University of Valencia, June, 2007.

[96] K. Sojakova and F. Rabe. Translating a dependently-typed logic to first-order logic. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques, WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 326–341. Springer, 2008.

[97] A. Tarlecki. Institutions: An abstract framework for formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, pages 105–130. Springer, 1999.

[98] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.

[99] D. Tsarkov and I. Horrocks. FaCT++. `http://owl.man.ac.uk/factplusplus/`.

[100] D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks. Using Vampire to reason with OWL. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference, ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 2004.

[101] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[102] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction, CADE 2009*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145. Springer, 2009.

[103] C. Wernhard. System description: Krhyper. Fachberichte Informatik 14–2003, Universität Koblenz-Landau, 2003.

[104] J. Zimmer and S. Autexier. The MathServe system for semantic web reasoning services. In U. Furbach and N. Shankar, editors, *Proceedings of the 3rd International Joint Conference on Automated Reasoning, IJCAR 2006*, volume 4130 of *Lecture Notes in Computer Science*, pages 140–144. Springer, 2006.

Part III

Papers Related to This Thesis

List of Publications

- R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. In G. Roşu, editor, *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 63–81. Elsevier, 2009.
- R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of membership equational logic specifications. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 174–193. Springer, 2008.
- A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of rewriting logic specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques, WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2009.
- A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. A declarative debugger for Maude. In J. Meseguer and G. Roşu, editors, *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, AMAST 2008*, volume 5140 of *Lecture Notes in Computer Science*, pages 116–121. Springer, 2008.
- A. Riesco, A. Verdejo, and N. Martí-Oliet. Enhancing the debugging of Maude specifications. In P. C. Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2010.
- A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative debugging of missing answers for Maude specifications. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 277–294. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.
- A. Riesco, A. Verdejo, and N. Martí-Oliet. A complete declarative debugger for Maude. In M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2011.
- A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 2011. To appear.

- M. Codescu, T. Mossakowski, A. Riesco, and C. Maeder. Integrating Maude into Hets. In M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 60–75. Springer, 2011.
- M. Codescu, T. Mossakowski, A. Riesco, and C. Maeder. Integrating Maude into Hets. Technical Report SIC-07/10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2010.

Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 238 (2009) 63–81

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Declarative Debugger for Maude Functional Modules

R. Caballero, N. Martí-Oliet, A. Riesco and A. Verdejo

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract

A declarative debugger for Maude functional modules, which correspond to executable specifications in membership equational logic, is presented. Starting from an incorrect computation, declarative debugging builds a debugging tree as a logical representation of the computation, that then is traversed by asking questions to an external oracle until the error is found. We summarize the construction of appropriate debugging trees for oriented equational and membership inferences, where all the nodes whose correctness does not need any justification have been collapsed. The reflective features of Maude allow us to generate and navigate the debugging tree of a Maude computation using operations in Maude itself; even the user interface of the declarative debugger can be specified in this way. We present the debugger's main features, such as two different strategies to traverse the debugging tree, use of a correct module to reduce the number of questions asked to the user, selection of trusted vs. suspicious statements by means of labels, and trusting of statements "on the fly."

Keywords: declarative debugging, membership equational logic, Maude, functional modules, metalevel implementation

1 Introduction

Maude is a high-level language and high-performance system supporting both equational and rewriting logic [12] computation for a wide range of applications. In particular, Maude functional modules correspond to specifications in *membership equational logic* (*MEL*) [2,13], which, in addition to equations, allows the statement of *membership assertions* characterizing the elements of a sort. In this way, Maude makes possible the faithful specification of data types (like sorted lists or search trees) whose data are defined not only by means of constructors, but also by the satisfaction of additional properties.

The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [7, Chap. 22]. The tracing

⋆ Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

doi:10.1016/j.entcs.2009.05.013

facilities allow us to follow the execution on a specification, that is, the sequence of rewrites that take place. Term coloring consists in printing with different colors the operators used to build a term that does not fully reduce. The Maude debugger allows the user to define break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered. There, we can see the current term and execute the next rewrite with tracing turned on.

The Maude debugger has as a disadvantage that, since it is based on the trace, it shows to the user every small step obtained by using a single statement. Thus the user can loose the general view of the *proof* of the incorrect inference that produced the wrong result. That is, when the user detects an unexpected statement application it is difficult to know where the incorrect inference started.

Different debugging approaches based on the language's semantics have been introduced in the field of declarative languages, such as *abstract diagnosis*, which formulates a debugging methodology based on abstract interpretation [9,1], or *declarative debugging*, also known as *algorithmic debugging*, which was first introduced by E. Y. Shapiro [19] and that constitutes the framework of this work. Declarative debugging has been widely employed in the logic [10,14,22], functional [21,17,16,18], and multiparadigm programming [5,3,11] languages. Declarative debugging is a semi-automatic technique that starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. The declarative debugging scheme [15] uses a *debugging tree* as a logical representation of the computation. Each node in the tree represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. A buggy node is a node containing an erroneous result, but whose children all have correct results. Hence, a buggy node has produced an erroneous output from correct inputs and corresponds to an erroneous fragment of code, which is pointed out as an error.

During the debugging process, the user does not need to understand the computation operationally. Any buggy node represents an erroneous computation step, and the debugger can display the program fragment responsible for it. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely the debugging tree *generation* and its *navigation* following some suitable strategy [20].

Here we present a declarative debugger for *Maude functional modules* [7, Chap. 4]. The debugging process starts with an incorrect transition from the initial term to a fully reduced unexpected one. Our debugger, after building a proof tree for that inference, will present to the user questions of the following form: "Is it correct that T fully reduces to T'?", which in general are easier to answer. Moreover, since the questions are located in the proof tree, the answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process.

The current version of the tool has the following characteristics:

- It supports all kinds of functional modules: operators can be declared with any combination of axiom attributes (except for the attribute `strat`, that allows to specify an evaluation strategy); equations can be defined with the `otherwise` attribute; and modules can be parameterized.

- It provides two strategies to traverse the debugging tree: *top-down*, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and *divide and query*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.
- Before starting the debugging process, the user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number of questions asked to the user.
- It allows the user to debug Maude functional modules where some equations and memberships are suspicious and have been labeled (each one with a different label). Only these labeled statements generate nodes in the proof tree, while the unlabeled ones are considered correct. The user is in charge of this labeling. Moreover, the user can answer that he trusts the statement associated with the currently questioned inference; that is, statements can be trusted "on the fly."

Exploiting the fact that rewriting logic is reflective [6,8], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [7, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [7, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our declarative debugger for Maude functional modules, including its user interactions, is implemented in Maude itself.

Complete explanations about the fundamentals of our declarative debugging approach, additional examples, and more information about the implementation can be found in the technical report [4], which, together with the Maude source files for the debugger, is available from the webpage `http://maude.sip.ucm.es/debugging`.

2 Declarative debugging of Maude functional modules

As mentioned in the introduction, Maude uses membership equational logic [2,13], a very expressive version of equational logic which, in addition to equations, allows the statement of membership assertions characterizing the elements of a sort. We present below how its specifications are represented as Maude functional modules and a brief description of the theoretical basics of our debugger.

2.1 Membership equational logic

A *signature* in *MEL* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n,k}\}_{(k_1 \ldots k_n,k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. Intuitively, terms with a kind but without a sort represent undefined or error elements. *MEL* atomic formulas are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership assertion. Order-sorted notation $s_1 < s_2$ (with $s_1, s_2 \in S_k$ for some kind k) can be used to abbreviate the conditional membership $(\forall x : k)\, x : s_2 \Leftarrow x : s_1$. A *specification* is a pair (Σ, E), where E is a set of sentences over the signature Σ.

Models of *MEL* specifications are called algebras. A Σ-*algebra* $\mathcal{A}$ consists of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$. A specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [2,13] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, initial and free algebras, and specification morphisms.

Since the *MEL* specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \to t'$, exactly when $\mathcal{A} \models t = t'$, i.e., when $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$, that is, $u \downarrow v$. This is the notation we will use in the inference rules and debugging trees studied in Sect. 2.3.

2.2 Representation in Maude

Maude functional modules, introduced with syntax `fmod...endfm`, are executable *MEL* specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification.

In a functional module we can declare sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly, and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat]` = `[Nat]`.

An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

2.2.1 *An example: binary search trees*

As an example of Maude functional modules, we show how to specify binary search trees without repeated elements, whose nodes contain elements that satisfy the theory `STOSET` (defining a strict total order on them) [7, Sect. 8.3].

```
fmod SEARCH-TREE{X :: STOSET} is
 sorts NeSearchTree{X} SearchTree{X} Tree{X} .
 subsorts NeSearchTree{X} < SearchTree{X} < Tree{X} .
 op empty : -> SearchTree{X} [ctor] .
 op ___ : Tree{X} X$Elt Tree{X} -> Tree{X} [ctor] .
```

where the operation for building non-empty search trees uses juxtaposition and `X$Elt` denotes the sort `Elt` from the theory `STOSET`.

A tree is a search tree when its root is bigger than all the elements in the left subtree and smaller than all the elements in the right subtree; this requirement is specified by means of memberships. Assuming that the subtrees are search trees, instead of comparing with all their elements, it is enough to compare with the minimum or maximum of the appropriate subtree.

```
 vars E E' : X$Elt .
 vars L R : SearchTree{X} .
 vars L' R' : NeSearchTree{X} .
 mb [leaf]     : empty E empty : NeSearchTree{X} .
 cmb [1ch1] : L' E empty    : NeSearchTree{X} if max(L') < E .
 cmb [1ch2] : empty E R'    : NeSearchTree{X} if E < min(R') .
 cmb [2ch]  : L' E R'       : NeSearchTree{X}
  if max(L') < E /\ E < max(R') .
 ops min max : NeSearchTree{X} -> X$Elt .
 ceq [mn1] : min(empty E R) = E if empty E R : NeSearchTree{X} .
 ceq [mn2] : min(L' E R)    = min(L') if L' E R : NeSearchTree{X} .
 ceq [mx1] : max(L E empty) = E if L E empty : NeSearchTree{X} .
 ceq [mx2] : max(L E R')    = max(R') if L E R' : NeSearchTree{X} .
```

The `delete` operation is specified as usual by structural induction, and in the non-empty case by comparing the element to be deleted with the root of the tree and distinguishing the three cases according to whether the former is smaller than, equal to, or bigger than the latter.

```
  op delete : SearchTree{X} X$Elt -> SearchTree{X} .
  eq  [dl1] : delete(empty, E)     = empty .
  ceq [dl2] : delete(L E R, E')    = delete(L, E') E R
              if E' < E /\ L E R : NeSearchTree{X} .
  ceq [dl3] : delete(L E R, E')    = L E delete(R, E')
              if E < E' /\ L E R : NeSearchTree{X} .
  ceq [dl4] : delete(empty E R, E) = R if empty E R : NeSearchTree{X} .
  ceq [dl5] : delete(L E empty, E) = L if L E empty : NeSearchTree{X} .
  ceq [dl6] : delete(L' E R', E)   = L' E' delete(R', E)
              if E' := min(R') /\ L' E R' : NeSearchTree{X} .
 endfm
```

This specification could be completed with other operations for insertion and look up.

Now we can instantiate this module with the predefined module INT of integer numbers, and reduce the following term

```
Maude> red delete((empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))), 5) .
result NeSearchTree{Int}:
   (empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty)))
```

We obtain a *tree with repetitions*. Moreover, Maude infers that *it is a search tree!* Did you notice the bugs? We will show in Sect. 3.3 how to use the debugger to detect them.

2.3 Declarative debugging

The inference rules of the calculus defining the operational semantics can be found in Fig. 1. They are an adaptation to the case of Maude functional modules of the deduction rules for *MEL* presented in [2]. We assume the existence of an *intended interpretation* $\mathcal{I}$ of the specification, which is a Σ-algebra corresponding to the model that the user had in mind while writing the statements E. The user expects that $\mathcal{I} \models e \to e'$, $\mathcal{I} \models e : s$ for each reduction $e \to e'$ and membership $e : s$ computed w.r.t. the specification (Σ, E).

We will say that $e \to e'$ (respectively $e : s$) is *valid* when it holds in $\mathcal{I}$, and *invalid* otherwise. Declarative debuggers rely on some external oracle, normally the user, in order to obtain information about the validity of some nodes in the debugging tree. The concept of validity can be extended to distinguish *wrong equations* and *wrong membership axioms*, which are those specification pieces that can deduce something invalid from valid information.

It will be convenient to represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. In declarative debugging we are specially interested in *buggy nodes* which are invalid nodes with all its children valid. Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy:

Input: A tree T with an invalid root.

(**Reflexivity**) $$\frac{}{e \to e}\ (Rf)$$

(**Transitivity**) $$\frac{e_1 \to e' \qquad e' \to e_2}{e_1 \to e_2}\ (Tr)$$

(**Congruence**) $$\frac{e_1 \to e_1' \quad \dots \quad e_n \to e_n'}{f(e_1, \dots, e_n) \to f(e_1', \dots, e_n')}\ (Cong)$$

(**Subject Reduction**) $$\frac{e \to e' \quad e' : s}{e : s}\ (SRed)$$

(**Membership**) $$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{1 \le i \le n} \quad \{\theta(v_j) : s_j\}_{1 \le j \le m}}{\theta(e) : s}\ (Mb)$$
$$\text{if } e : s \Leftarrow u_1 = u_1' \wedge \cdots \wedge u_n = u_n' \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$$

(**Replacement**) $$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{1 \le i \le n} \quad \{\theta(v_j) : s_j\}_{1 \le j \le m}}{\theta(e) \to \theta(e')}\ (Rep)$$
$$\text{if } e \to e' \Leftarrow u_1 = u_1' \wedge \cdots \wedge u_n = u_n' \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$$

Fig. 1. Semantic calculus for Maude functional modules

Output: A buggy node in T.

Description: Consider the root N of T. There are two possibilities:

- If all the children of N are valid, then finish identifying N as buggy.
- Otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T. As a consequence, if T is a proof tree with an invalid root, then there exists a buggy node $N \in T$ such that all the ancestors of N are invalid.

However, we will not use the proof tree T as debugging tree, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*). In order to simplify the proof trees we take advantage of a property that every Σ-algebra $\mathcal{A}$ satisfies: if $e \to e'$ (respectively $e : s$) can be deduced by any of the first four inference rules of the calculus using premises that hold in $\mathcal{A}$, then $\mathcal{A} \models e \to e'$ (respectively $\mathcal{A} \models e : s$). This property cannot be extended to the *membership* and *replacement* inference rules, where the correctness of the conclusion depends not only on the calculus but also on the associated specification statement, which could be wrong. Therefore the only inferences that can obtain an invalid conclusion from valid premises, i.e., the only possible buggy nodes, correspond to the *replacement* and *membership* inferences. The $APT(T)$ keeps only the nodes corresponding to these inferences. Fig. 2 shows the definition of $APT(T)$, where the T_i represent proof trees corresponding to the premises in some inferences.

The rule APT_1 keeps the root unaltered and employs the auxiliary function APT' to produce the child subtrees. APT' is defined in rules $APT_2 \dots APT_8$. It takes a proof tree as input parameter and returns a forest $\{T_1, \dots, T_n\}$ of APTs as result. The rules for APT' are assumed to be tried top-down, in particular APT_4 must not be applied if APT_3 is also applicable. It is easy to check that every node $N \in T$ conclusion of a *replacement* or *membership* inference has its corresponding node $N' \in APT(T)$ labeled with the same abbreviation, and conversely, that for

$$(\mathbf{APT}_1)\ \ APT\left(\frac{T_1\dots T_n}{af}{\scriptstyle(R)}\right) = \frac{APT'\left(\dfrac{T_1\dots T_n}{af}{\scriptstyle(R)}\right)}{af}\ \ \text{(with } (R) \text{ any inference rule)}$$

$$(\mathbf{APT}_2)\ \ APT'\left(\frac{}{e\to e}{\scriptstyle(Rf)}\right) = \emptyset$$

$$(\mathbf{APT}_3)\ \ APT'\left(\frac{\dfrac{T_1\dots T_n}{e_1\to e'}{\scriptstyle(Rep)}\quad T_{n+1}}{e_1\to e_2}{\scriptstyle(Tr)}\right) = \left\{\frac{APT'(T_1)\dots APT'(T_n)\ APT'(T_{n+1})}{e_1\to e_2}{\scriptstyle(Rep)}\right\}$$

$$(\mathbf{APT}_4)\ \ APT'\left(\frac{T_1\quad T_2}{e_1\to e_2}{\scriptstyle(Tr)}\right) = \{APT'(T_1),\ APT'(T_2)\}$$

$$(\mathbf{APT}_5)\ \ APT'\left(\frac{T_1\dots T_n}{e_1\to e_2}{\scriptstyle(Cong)}\right) = \{APT'(T_1),\dots,APT'(T_n)\}$$

$$(\mathbf{APT}_6)\ \ APT'\left(\frac{T_1\quad T_2}{e:s}{\scriptstyle(SRed)}\right) = \{APT'(T_1),\ APT'(T_2)\}$$

$$(\mathbf{APT}_7)\ \ APT'\left(\frac{T_1\dots T_n}{e:s}{\scriptstyle(Mb)}\right) = \left\{\frac{APT'(T_1)\dots APT'(T_n)}{e:s}{\scriptstyle(Mb)}\right\}$$

$$(\mathbf{APT}_8)\ \ APT'\left(\frac{T_1\dots T_n}{e_1\to e_2}{\scriptstyle(Rep)}\right) = \left\{\frac{APT'(T_1)\dots APT'(T_n)}{e_1\to e_2}{\scriptstyle(Rep)}\right\}$$

Fig. 2. Transformation rules for obtaining abbreviated proof trees

each $N' \in APT(T)$ different from the root, there is a node $N \in T$, which is the conclusion of a *replacement* or *membership* inference. In particular the node associated to $e_1 \to e_2$ in the righthand side of APT_3 is the node $e_1 \to e'$ of the proof tree T, which is not included in the $APT(T)$. We have chosen to introduce $e_1 \to e_2$ instead of simply $e_1 \to e'$ in the $APT(T)$ as a pragmatic way of simplifying the structure of the APTs, since e_2 is obtained from e' and hence likely simpler (the root of the tree T_{n+1} in APT_3 must be necessarily of the form $e' \to e_2$ by the structure of the inference rule for transitivity in Fig. 1).

Although $APT(T)$ is no longer a proof tree we keep the inference labels (*Rep*) and (*Mb*), assuming implicitly that they contain a reference to the equation or membership axiom used at the corresponding step in the original proof tree. This information is used by the debugger in order to single out the incorrect fragment of specification code. The abbreviation of the tree reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique, as the following theorem (proved in [4]) guarantees:

Theorem 2.1 *Let S be a specification, $\mathcal{I}$ its intended interpretation, and T a finite proof tree with invalid root. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*

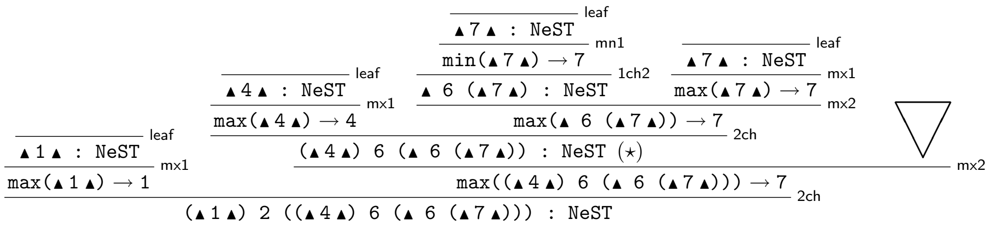


Fig. 3. Abbreviated proof tree

- *Any buggy node in APT(T) has an associated wrong statement in S (soundness).*

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user answers correctly all the questions about the validity of the *APT* nodes asked by the debugger (see Sect. 3.1).

The *APT* for the wrong membership inference of Sect. 2.2.1 is shown in Fig. 3, where ▲ denotes the `empty` search tree and ▽ represents the already depicted proof subtree with root `max(▲ 6 (▲7▲))` → `7`.

3 Using the debugger

Before describing the basics of the user interaction with the debugger, we make explicit what is assumed about the modules introduced by the user; then we present the available commands and how to use them to debug the buggy example introduced in Sect. 2.2.1.

3.1 Assumptions

Since we are debugging Maude functional modules, they are expected to satisfy the appropriate executability requirements, namely, the specifications have to be terminating, confluent, and sort decreasing.

One interesting feature of our tool is that the user is allowed to trust some statements, by means of labels applied to the suspicious statements. This means that the unlabeled statements are assumed to be correct. A trusted statement is treated in the implementation as the first four rules in Fig. 1 are treated in the *APT* transformation; more specifically, an instance of the *membership* or *replacement* inference rules corresponding to a trusted statement is collapsed in the abbreviated proof tree. In order to obtain a nonempty abbreviated proof tree, the user must have labeled some statements (all with different labels); otherwise, everything is assumed to be correct. In particular, the buggy statement must be labeled in order to be found. When not all the statements are labeled, the correctness and completeness results shown in Sect. 2.3 are conditioned by the goodness of the labeling for which the user is responsible.

Although the user can introduce a module importing other modules, the debugging process takes place in the *flattened* module. However, the debugger allows the user to trust a whole imported module.

As already mentioned, navigation of the debugging tree takes place by asking questions to an external oracle, which in our case is either the user or another module introduced by the user. In both cases the answers are assumed to be correct. If either the module is not really correct or the user provides an incorrect answer, the result is unpredictable. Notice that the information provided by the correct module need not be complete, in the sense that some functions can be only partially defined.

3.2 Commands

The debugger is initiated in Maude by loading the file `fdd.maude`, which starts an input/output loop that allows the user to interact with the tool.

As we said in the introduction, the generated proof tree can be navigated by using two different strategies, namely, *top-down* and *divide and query*, being the latter the default one. The user can switch between them by using the commands `(top-down strategy .)` and `(divide-query strategy .)`. If a module with correct definitions is used to reduce the number of questions, it must be indicated before starting the debugging with the command `(correct module MODULE-NAME .)`.

The user can choose between using all the labeled statements in the debugging process (by default) or selecting some of them by means of the command

```
(set debug select on .)
```

Once this mode is activated, the user can select and deselect statements by using

```
(debug select LABELS .)
(debug deselect LABELS .)
```

where `LABELS` is a list of labels separated by spaces.

Moreover, all the labels in statements of a flattened module can be selected or deselected with the following commands:

```
(debug include MODULES .)
(debug exclude MODULES .)
```

where `MODULES` is a list of module names separated by spaces.

Once we have selected the strategy and, optionally, the correct module and suspicious labels, we start the debugging process with the command [1]

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM .)
```

In the same way, we can debug a membership inference with the command

```
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
```

How the process continues depends on the selected strategy. In case the top-down strategy is selected, several nodes will be displayed in each question. If there is an invalid node, we must select one of them with the command `(node N .)`, where `N` is the identifier of that wrong node. If all the nodes are correct, we type `(all valid .)`. In the divide and query strategy, each question refers to one inference that can be either correct or wrong. The different answers are transmitted to the

[1] If no module name is given, the current module is used by default.

$$\dfrac{\dfrac{\dfrac{}{\blacktriangle 4 \blacktriangle\ :\ \texttt{NeST}}\,\text{leaf}}{\texttt{max}(\blacktriangle 4 \blacktriangle) \longrightarrow 4}\,\text{mx1} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\blacktriangle 7 \blacktriangle\ :\ \texttt{NeST}}\,\text{leaf}}{\texttt{min}(\blacktriangle 7 \blacktriangle) \longrightarrow 7}\,\text{mn1}}{\blacktriangle\ 6\ (\blacktriangle 7 \blacktriangle)\ :\ \texttt{NeST}}\,\text{1ch2} \quad \dfrac{\dfrac{}{\blacktriangle 7 \blacktriangle\ :\ \texttt{NeST}}\,\text{leaf}}{\texttt{max}(\blacktriangle 7 \blacktriangle) \longrightarrow 7}\,\text{mx1}}{\texttt{max}(\blacktriangle\ 6\ (\blacktriangle 7 \blacktriangle)) \longrightarrow 7\ (\dagger)}\,\text{mx2}}{(\blacktriangle 4 \blacktriangle)\ 6\ (\blacktriangle\ 6\ (\blacktriangle 7 \blacktriangle))\ :\ \texttt{NeST}}\,\text{2ch}$$

Fig. 4. Abbreviated proof tree after the first answer

$$\dfrac{\dfrac{\dfrac{}{\blacktriangle 4 \blacktriangle\ :\ \texttt{NeST}}\,\text{leaf}}{\texttt{max}(\blacktriangle 4 \blacktriangle) \longrightarrow 4}\,\text{mx1}}{(\blacktriangle 4 \blacktriangle)\ 6\ (\blacktriangle\ 6\ (\blacktriangle 7 \blacktriangle))\ :\ \texttt{NeST}}\,\text{2ch}$$

Fig. 5. Abbreviated proof tree after the second answer

debugger with the commands (yes .) and (no .). Instead of just answering yes, we can also *trust* some statements on the fly if, once the process has started, we decide the bug is not there. To trust the current statement we type the command (trust .).

Finally, we can return to the previous state in both strategies by using the command (undo .).

3.3 *Binary search trees revisited*

We recall from Sect. 2.2.1 that the deletion in our binary search trees specification is incorrect. In particular Maude assigns the sort NeSearchTree{Int} to a tree with repetitions. We can debug the inference of this membership with the command

```
Maude> (debug in SEARCH-TREE-TEST :
(empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty))) : NeSearchTree{Int} .)
```

that generates the debugging tree shown in Fig. 3 by considering all the labeled statements as suspicious. Since the default navigation strategy is divide and query, the debugger selects the node marked with $\star$ in the figure, and then asks the following question:

```
Is this membership (associated with the membership 2ch) correct?
(empty 4 empty) 6 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
Maude> (no .)
```

Since the answer is no, the debugger discards the rest of the tree and focuses in the subtree with this node as root (see Fig. 4). The next question corresponds to the node marked with † in the figure.

```
Is this transition (associated with the equation mx2) correct?
max(empty 6 (empty 7 empty)) -> 7
Maude> (yes .)
```

When the answer is yes the corresponding subtree is deleted, obtaining in this case the tree in Fig. 5. The next question is

```
Is this transition (associated with the equation mx1) correct?
max(empty 4 empty) -> 4
Maude> (trust .)
```

In the last question, we realized that the equation applied is so simple that we

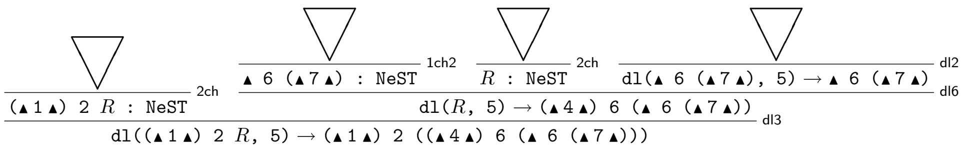

(▲ 1 ▲) 2 R : NeST
2ch
▲ 6 (▲ 7 ▲) : NeST
1ch2
R : NeST
2ch
dl(▲ 6 (▲ 7 ▲), 5) → ▲ 6 (▲ 7 ▲)
dl2
dl(R, 5) → (▲ 4 ▲) 6 (▲ 6 (▲ 7 ▲))
dl6
dl3
dl((▲ 1 ▲) 2 R, 5) → (▲ 1 ▲) 2 ((▲ 4 ▲) 6 (▲ 6 (▲ 7 ▲)))

Fig. 6. Abbreviated proof tree for the top down strategy

can *trust* it. This answer has a behavior similar to `yes`: it deletes all the subtrees whose root is labeled as the current statement. With these answers, we obtain a tree with only one node and the debugger is able to conclude which is the buggy membership.

```
The buggy node is:
(empty 4 empty) 6 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
With the associated membership: 2ch
```

In fact, if we check now this membership we notice that it compares the root with the biggest value of the right subtree, when it should be compared with the smallest one. After fixing this error, the `delete` function is still incorrect, so we debug this function (using the top-down strategy for illustration's sake) as follows:

```
Maude> (top-down strategy .)
Top-down strategy selected.
Maude> (set debug select on .)
Debug select is on.
Maude> (debug select leaf 1ch1 1ch2 2ch dl2 dl3 dl4 dl5 dl6 .)
Labels leaf 1ch1 1ch2 2ch dl2 dl3 dl4 dl5 dl6 are now suspicious.
Maude> (debug in SEARCH-TREE-TEST :
  delete((empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))), 5)
        -> (empty 1 empty) 2 ((empty 4 empty) 6 (empty 6 (empty 7 empty))) .)
```

where we have decided to mark as suspicious the memberships and the non-trivial equations of `delete`. In this case, the debugger builds the proof tree (partially) shown in Fig. 6 (where R denotes the search tree `(▲ 4 ▲) 5 (▲ 6 (▲ 7 ▲))`), so it asks the following questions:

```
Please, choose a wrong node:
Node 0 : (empty 1 empty) 2 ((empty 4 empty) 5 (empty 6 (empty 7 empty))) :
       NeSearchTree{Int}
Node 1 : delete((empty 4 empty) 5 (empty 6 (empty 7 empty)), 5) ->
       (empty 4 empty) 6 (empty 6 (empty 7 empty))
Maude> (node 1 .)

Please, choose a wrong node:
Node 0 : empty 6 (empty 7 empty) : NeSearchTree{Int}
Node 1 :(empty 4 empty) 5 (empty 6 (empty 7 empty)) : NeSearchTree{Int}
Node 2 : delete(empty 6 (empty 7 empty), 5) -> empty 6 (empty 7 empty)
Maude> (all valid .)

The buggy node is:
delete((empty 4 empty) 5 (empty 6 (empty 7 empty)), 5) ->
    (empty 4 empty) 6 (empty 6 (empty 7 empty))
With the associated equation: dl6
```

The debugger concludes that the problem is within the equation `dl6`. We leave to the interested reader the task of fixing it.

4 Implementation

As we said in the introduction, the reflective power of Maude allows us to generate and navigate the debugging tree of a computation in Maude itself. Since navigation is done by asking questions to the user, this stage has to handle the navigation strategy together with the input/output interaction with the user. Indeed, this interaction can also be implemented in Maude by using the predefined module `LOOP-MODE` [7, Chap. 17], that handles the input/output and maintains the persistent state of the tool.

Below we show the main functions involved in the implementation; the technical report [4] provides a full explanation of the complete implementation, including the user interaction.

4.1 Debugging tree construction

To build the debugging tree we use the facts that the equations defined in Maude functional modules are both *terminating* and *confluent*. Instead of creating the complete proof tree and then abbreviating it, we build the abbreviated proof tree directly. The information kept in each node corresponds to an inference, represented by a statement's label, its lefthand side (a term), and its righthand side (either a term or a sort).

The function `createTree` controls the construction of this tree (it implements the function APT from Fig. 2). It receives the module where a suspicious inference took place, a correct module (or the constant `maybe` when no such module is provided) to prune the tree, the term initially reduced, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial reduction as the root of the tree and uses an auxiliary function `createForest` (implementing the function APT' from Fig. 2) that, in addition to the arguments received by `createTree`, receives the module "cleaned" of suspicious statements (by using `transform`), and generates the forest of abbreviated trees corresponding to the reduction between the two terms given as arguments. This transformed module is used to improve the efficiency of the tree construction, because we can use it to check if a term reaches its final form by using only trusted statements, thus avoiding to build a tree that will be finally empty.

```
op createTree : Module Maybe{Module} Term Term QidSet -> Tree .
ceq createTree(M, CM, T, T', QS) = contract(
             tree(node('root : T -> T', getOffspring*(F) + 1), F))
 if M' := transform(M, QS) /\
    F := createForest(M, M', CM, normal(M, T), normal(M, T'), QS) .
```

where `contract` prunes the root of the tree if it is duplicated after the computation of the tree.

We use the function `createForest` to create a forest of abbreviated trees. This function checks if the terms can be reduced by using only trusted statements or if the correct module can compute this reduction; in such cases, there is no need to compute the forest. Otherwise, it works with the same innermost strategy as the Maude interpreter: It first tries to fully reduce the subterms (by means of the function `reduceSubterms`), and once all the subterms have been reduced, if the result is not the final one, it tries to reduce at the top (by using the function `applyEq`), to reach the final result by *transitivity*.

```
op createForest : Module Module Maybe{Module} Term Term QidSet -> Forest .
ceq createForest(OM, TM, CM, T, T', QS) = mtForest
 if reduce(TM, T) == T' or-else reduce(M, T) == reduce(M, T') .
ceq createForest(OM, TM, CM, T, T', QS) =
                          if T'' == T' then F
                          else F applyEq(OM, TM, CM, T'', T', QS)
                          fi
 if < T'', F > := reduceSubterms(OM, TM, CM, T, QS) [owise] .
```

The `reduceSubterms` function returns a pair consisting of the term with its subterms fully reduced (that is, this function mimics a specific behavior of the *congruence* rule in Fig. 1) and the forest of abbreviated trees generated by these reductions.

The function `applyEq` tries to apply (at the top) one equation,[2] by using the *replacement* rule from Fig. 1, with the constraint that we cannot apply equations with the `otherwise` attribute while other equations can be applied. To apply an equation we check if the term we are trying to reduce matches the lefthand side of the equation and its conditions are fulfilled. If this happens, we obtain a substitution (from both the matching with the lefthand side and the matching conditions) that we can apply to the righthand side of the equation. Note that if we can obtain the transition in the correct module, the forest is not calculated.

```
op applyEq : Module Module Maybe{Module} Term Term QidSet -> Forest .
op applyEq : Module Module Maybe{Module} Term Term QidSet EquationSet -> Forest .
ceq applyEq(OM, TM, CM, T, T', QS) = mtForest
 if reduce(TM, T) == T' or-else reduce(CM, T) == T' .
eq applyEq(OM, TM, CM, T, T', QS) =
          applyEq(OM, TM, CM, T, T', QS, getEqs(OM)) [owise] .
```

First, we try to apply the equations without the `otherwise` attribute. Otherwise, we check the other equations.

```
ceq applyEq(OM, TM, CM, T, T', QS, Eq EqS) =
        if in?(AtS, QS)
        then tree(node(label(AtS) : T -> T', getOffspring*(F) + 1), F)
        else F
        fi
 if ceq L = R if C [AtS] . := generalEq(Eq) /\
    not owise?(AtS) /\
    SB := metaMatch(OM, L, T, C, 0) /\
    R' := normal(OM, substitute(R, SB)) /\
    F := conditionForest(substitute(C, SB), OM, TM, CM, QS)
         createForest(OM, TM, CM, R', T', QS) .
```

where the function `in?` checks if the equation `Eq` is suspicious. If this is the case, a new node corresponding to the applied equation is generated. The forest for the conditions is generated by the function `conditionForest`; since it is used after

[2] Since the module is assumed to be confluent, we can choose any equation and the final result should be the same.

having checked that the condition is fulfilled (by the function `metaMatch` above), it does not check it again. It distinguishes between the different types of conditions. If the condition is an equation, trees of the reduction of the terms to their normal forms are generated.

```
op conditionForest : Condition Module Module Maybe{Module} QidSet -> Forest .
eq conditionForest(T = T' /\ COND, OM, TM, CM, QS) =
              createForest(OM, TM, CM, T, reduce(OM, T), QS)
              createForest(OM, TM, CM, T', reduce(OM, T), QS)
              conditionForest(COND, OM, TM, CM, QS) .
```

The case of the matching conditions is very similar. In the membership case, we use the version of `createForest` that builds a forest for a membership inference where the sort is the least one assignable to the term in the condition.

```
eq conditionForest(T : S /\ COND, OM, TM, CM, QS) =
              createForest(OM, TM, CM, T, type(OM, T), QS)
              conditionForest(COND, OM, TM, CM, QS) .
```

To generate the forest for memberships we use another version of the function `createForest`, that mimics the *subject reduction* rule from Fig. 1 by first computing the tree for the full reduction of the term (by means of `createForest`) and then computing the tree for the membership inference by using an auxiliary version of `createForest` that uses the operator declarations and the membership axioms. Note that if we can infer the type from the correct module, there is no need to calculate the forest.

```
op createForest : Module Module Maybe{Module} Term Sort QidSet -> Forest .
op createForest : Module Module Maybe{Module} Term Sort QidSet OpDeclSet
                  MembAxSet -> Forest .
ceq createForest(OM, TM, CM, T, S, QS) = mtForest
 if Ty := type(CM, T) /\ sortLeq(CM, Ty, S) .
ceq createForest(OM, TM, CM, T, S, QS) =
      createForest(OM, TM, CM, T, T', QS)
      createForest(OM, TM, CM, T', S, QS, getOps(OM), getMbs(OM))
 if T' := reduce(OM, T) [owise] .
```

The auxiliary `createForest` computes a forest for a membership inference of the least sort of a term previously fully reduced; this corresponds to a concrete application of the *membership* inference rule from Fig. 1. It first checks if the membership has been inferred by using the operator declarations. If the membership has not been computed by using these declarations, it checks the membership axioms.

To check the operators we examine that both the arity and co-arity of the term and the declaration fit (with function `checkTypes`) and recursively calculate the forest generated by the subterms (by using `createForest*`). Notice that we never generate a new node for the application of an operator, because we always trust the signature.

```
op applyOp : Module Module Maybe{Module} Term Sort QidSet OpDeclSet -> Maybe{Forest} .
ceq applyOp(OM, TM, CM, Q[TL], Ty, QS, op Q : TyL -> Ty [AtS] . ODS) =
             createForest*(OM, TM, CM, TL, QS)
 if checkTypes(TL, TyL, OM) .
ceq applyOp(OM, TM, CM, CONST, S, QS, op Q : nil -> Ty [AtS] . ODS) = mtForest
 if getName(CONST) = Q /\ getType(CONST) = Ty .
eq applyOp(OM, TM, CM, T, S, QS, ODS) = noProof [owise] .
```

We check the membership axioms in a similar fashion to the equation application, that is, we only generate a new root below the forest for the conditions if the membership is suspicious. The unconditional axioms generate leaves of the tree, while the conditional ones generate nodes with (possibly) non-empty forests.

```
op applyMb : Module Module Maybe{Module} Term Sort QidSet MembAxSet -> Forest .
ceq applyMb(OM, TM, CM, T', S, QS, MA MAS) =
        if in?(AtS, QS)
        then tree(node(label(AtS) : T' : S, getOffspring*(F) + 1), F)
        else F
        fi
 if cmb T : S if C [AtS] . := generalMb(MA) /\
    SB := metaMatch(OM, T, T', C, 0) /\
    F := conditionForest(substitute(C, SB), OM, TM, CM, QS) .
eq applyMb(OM, TM, CM, T, S, QS, MA) = mtForest [owise] .
```

4.2 Debugging tree navigation

Regarding the navigation of the debugging tree, we have implemented two strategies. In the top-down strategy the selection of the next node of the debugging tree is done by the user, thus we do not need any function to compute it. The divide and query strategy used to traverse the debugging tree selects each time the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.

The function `searchBestNode` calculates the best node by searching for a subtree that minimizes the function `getDiff`, where the first argument is the size of the whole tree and the second one the size of the subtree.

```
op getDiff : Nat Nat -> Nat .
eq getDiff(N, N') = sd(N, 2 * N') .
```

Since we use the symmetric difference function, the difference between the size of the whole tree and the double of the size of the current subtree will initially decrease (while the double of the size of the subtree is bigger than the size of the tree) and finally it will increase (when the size of the tree is bigger than the double of the size of the subtree). Thus, the function `searchBestNode` keeps the information about the last difference in order to stop searching in the subtree when the current difference is bigger than the last one. It uses an auxiliary function that receives the tree, the total number of nodes in the whole tree, the last and the best difference so far, the identifier of the best node, and the identifier of the root of the subtree it is currently traversing. The last and best difference are initialized with a value big enough (ten times the number of nodes), in order to avoid the selection of the initial root as the best node.

```
op searchBestNode : Tree -> NatList .
op searchBestNode : Tree Nat Nat Nat NatList NatList -> NPair .

eq searchBestNode(tree(node(I, NODES), F)) =
   first(searchBestNode(tree(node(I, NODES), F), NODES,
                        10 * NODES, 10 * NODES, nil, nil)) .

ceq searchBestNode(T, NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL) =
                                                     < BEST_NODE, BEST_DIFF >
   if LAST_DIFF <= getDiff(NODES, getOffspring(T)) .
```

If the new difference is better than the last one, the function recursively traverses the forest of the current node with the function `searchBestNode*`.

```
  ceq searchBestNode(tree(ND, F), NODES, LAST_DIFF, BEST_DIFF, BEST_NODE, NL) =
       if NEW_DIFF < BEST_DIFF then
         searchBestNode*(F, NODES, NEW_DIFF, NEW_DIFF, NL, NL, 0)
       else
         searchBestNode*(F, NODES, NEW_DIFF, BEST_DIFF, BEST_NODE, NL, 0)
       fi
   if NEW_DIFF := getDiff(NODES, offspring(ND)) /\
      LAST_DIFF > NEW_DIFF .
```

5 Conclusions and future work

In this paper we have presented how to use the Maude reflective capabilities to implement a debugger for Maude functional modules. It complements other debugging techniques for Maude, such as tracing and term coloring, by allowing to debug a large range of modules (only the `strat` attribute is forbidden, although we expect to allow it in a near future). An important advantage of this kind of debuggers is the help provided in locating the buggy statements, assuming the user answers correctly the corresponding questions.

From the theoretical point of view, the main novelty of our approach w.r.t. other proposals for declarative debugging of functional languages such as [21,17,18] is that our debugging tree (the *APT*) is obtained from a proof tree in a suitable semantic calculus, which allows us to prove the correctness and completeness of the debugging technique. Furthermore, our debugging of *MEL* specifications has required an appropriate treatment of memberships which do not appear in previous works.

The complexity of the debugging process increases with the size of the proof tree. In the case of the top-down strategy the number of questions for a tree T is proportional to $\mathit{depth}(T) * \mathit{degree}(T)$. In the case of the divide and query strategy the number of questions is, on average, proportional to $\log \mathit{size}(T)$. Note that the size of the tree does not depend on the total number of statements but on the number of applications of suspicious statements involved in the wrong inference. Moreover, bugs found when reducing complex initial terms can be, in general, reproduced with simpler terms which give rise to smaller proof trees.

We can minimize the number of questions by trusting statements or keeping track of the questions already answered, in order to avoid asking the same question twice.

Since one of the requirements of this kind of debuggers is the interaction with an oracle, that typically is the user, one of the principal aspects that must be improved is the user interface. We plan to provide a complementary graphical interface that allows the user to navigate the tree with more freedom.

We plan to extend our framework by studying how to debug *system modules*, which correspond to rewriting logic specifications and have rules in addition to memberships and equations. These rules can be non-terminating and non-confluent, and thus behave very differently from the statements in the specifications we handle here. Indeed, if the rules of a system module are confluent and terminating, we can use the current version of our debugger by first translating the rewrite rules into equations.

In the context of general system modules, we also plan to study how to debug *missing answers* [14] in addition to the wrong answers we have treated thus far. That is, the non-determinism inherent to a system module implies that a term can be rewritten in several different ways. If the specification does not fulfill the intended model, it may be the case that not all the possible solutions are reached.

Acknowledgement

We thank Santiago Escobar for motivating this research and the anonymous referees for their helpful suggestions to improve this paper.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *WCFLP '05: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming*, pages 8–13, New York, NY, USA, 2005. ACM Press.

[4] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of Maude functional modules. Technical Report SIC-4/07, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2007. http://maude.sip.ucm.es/debugging.

[5] R. Caballero and M. Rodríguez-Artalejo. DDT: A declarative debugging tool for functional-logic languages. In *Proc. 7th International Symposium on Functional and Logic Programming (FLOPS'04)*, volume 2998 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[6] M. Clavel. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications.* CSLI Publications, Stanford University, 2000.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[9] M. Comini, G. Levi, M. C. Meo, and G. Vitiello. Abstract diagnosis. *Journal of Logic Programming*, 39(1-3):43–93, 1999.

[10] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[11] I. MacLarty. *Practical Declarative Debugging of Mercury Programs.* PhD thesis, University of Melbourne, 2005.

[12] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[13] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[14] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.

[15] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[16] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[17] H. Nilsson and P. Fritzson. Algorithmic debugging of lazy functional languages. *Journal of Functional Programming*, 4(3):337–370, 1994.

[18] B. Pope. Declarative debugging with Buddha. In *Advanced Functional Programming - 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer, 2005.

[19] E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1983.

[20] J. Silva. A comparative study of algorithmic debugging strategies. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[21] N. Takahashi and S. Ono. DDS: A declarative debugging system for functional programs. *Systems and Computers in Japan*, 21(11):21–32, 1990.

[22] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, pages 151–174. Springer, 2000.

Solution to the challenge in page 74: To fix the equation d16 we must delete the element E' instead of E in the subtree.

```
ceq [d16] : delete(L' E R', E) = L' E' delete(R', E')
 if E' := min(R') /\ L' E R' : NeSearchTree{X} .
```

Declarative Debugging of Membership Equational Logic Specifications*

Rafael Caballero, Narciso Martí-Oliet, Adrián Riesco, and Alberto Verdejo

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Algorithmic debugging has been applied to many declarative programming paradigms; in this paper, it is applied to the rewriting paradigm embodied in Maude. We introduce a declarative debugger for executable specifications in membership equational logic which correspond to Maude functional modules. Declarative debugging is based on the construction and navigation of a debugging tree which logically represents the computation steps. We describe the construction of appropriate debugging trees for oriented equational and membership inferences. These trees are obtained as the result of collapsing in proof trees all those nodes whose correctness does not need any justification. We use an extended example to illustrate the use of the declarative debugger and its main features, such as two different strategies to traverse the debugging tree, use of a correct module to reduce the number of questions asked to the user, and selection of trusted vs. suspicious statements by means of labels. The reflective features of Maude have been extensively used to develop a prototype implementation of the declarative debugger for Maude functional modules using Maude itself.

Keywords: declarative debugging, membership equational logic, Maude, functional modules.

1 Introduction

As argued in [23], the application of declarative languages out of the academic world is inhibited by the lack of convenient auxiliary tools such as *debuggers*. The traditional separation between the problem logic (defining *what* is expected to be computed) and control (*how* computations are carried out actually) is a major advantage of these languages, but it also becomes a severe complication when considering the task of debugging erroneous computations. Indeed, the involved execution mechanisms associated to the control make it difficult to apply the typical techniques employed in imperative languages based on step-by-step trace debuggers.

Consequently, new debugging approaches based on the language's semantics have been introduced in the field of declarative languages, such as *abstract diagnosis*, which formulates a debugging methodology based on abstract interpretation [9,1], or *declarative debugging*, also known as *algorithmic debugging*, which was first introduced by E. Y. Shapiro [19] and that constitutes the framework of this work. Declarative debugging

* Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

has been widely employed in the logic [10,14,22], functional [21,17,16,18], and multi-paradigm [5,3,11] programming languages. Declarative debugging is a semi-automatic technique that starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. The declarative debugging scheme [15] uses a *debugging tree* as a logical representation of the computation. Each node in the tree represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. A buggy node is a node containing an erroneous result, but whose children all have correct results. Hence, a buggy node has produced an erroneous output from correct inputs and corresponds to an erroneous fragment of code, which is pointed out as an error.

During the debugging process, the user does not need to understand the computation operationally. Any buggy node represents an erroneous computation step, and the debugger can display the program fragment responsible for it. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely the debugging tree *generation* and its *navigation* following some suitable strategy [20].

In this paper we present a declarative debugger for *Maude functional modules* [7, Chap. 4]. Maude is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. It is a declarative language because Maude modules correspond in general to specifications in rewriting logic [12], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude functional modules correspond to specifications in *membership equational logic* [2,13], which, in addition to equations, allows the statement of *membership assertions* characterizing the elements of a sort. In this way, Maude makes possible the faithful specification of data types (like sorted lists or search trees) whose data are defined not only by means of constructors, but also by the satisfaction of additional properties.

For a specification in rewriting or membership equational logic to be executable in Maude, it must satisfy some executability requirements. In particular, Maude functional modules are assumed to be confluent, terminating, and sort-decreasing[1] [7], so that, by orienting the equations from left to right, each term can be reduced to a unique canonical form, and semantic equality of two terms can be checked by reducing both of them to their respective canonical forms and checking that they coincide. Since we intend to debug functional modules, we will assume throughout the paper that our membership equational logic specifications satisfy these executability requirements.

The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [7, Chap. 22]. The tracing facilities allow us to follow the execution on a specification, that is, the sequence of rewrites that take place. Term coloring consists in printing with different colors the operators used to build a term that does not fully reduce. The Maude debugger allows the user to define break points in the execution by selecting some operators or statements. When

[1] All these requirements must be understood *modulo* some axioms such as associativiy and commutativity that are associated to some binary operations.

a break point is found the debugger is entered. There, we can see the current term and execute the next rewrite with tracing turned on.

The Maude debugger has as a disadvantage that, since it is based on the trace, it shows to the user every small step obtained by using a single statement. Thus the user can loose the general view of the *proof* of the incorrect inference that produced the wrong result. That is, when the user detects an unexpected statement application it is difficult to know where the incorrect inference started.

Here we present a different approach based on declarative debugging that solves this problem for functional modules. The debugging process starts with an incorrect transition from the initial term to a fully reduced unexpected one. Our debugger, after building a proof tree for that inference, will present to the user questions of the following form: "Is it correct that T fully reduces to T'?", which in general are easy to answer. Moreover, since the questions are located in the proof tree, the answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process.

The current version of the tool has the following characteristics:

- It supports all kinds of functional modules: operators can be declared with any combination of axiom attributes (except for the attribute `strat`, that allows to specify an evaluation strategy); equations can be defined with the `otherwise` attribute; and modules can be parameterized.[2]
- It provides two strategies to traverse the debugging tree: *top-down*, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and *divide and query*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise.
- Before starting the debugging process, the user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number of questions asked to the user.
- It allows the user to debug Maude functional modules where some equations and memberships are suspicious and have been labeled (each one with a different label). Only these labeled statements generate nodes in the proof tree, while the unlabeled ones are considered correct. The user is in charge of this labeling. Moreover, the user can answer that he trusts the statement associated with the currently questioned inference; that is, statements can be trusted "on the fly."

Detailed proofs of the results, additional examples, and much more information about the implementation can be found in the technical report [4], which, together with the Maude source files for the debugger, is available from the webpage `http://maude.sip.ucm.es/debugging`.

[2] For the sake of simplicity, our running example will be unparameterized, but it can easily be parameterized, as shown in [4].

2 Maude Functional Modules

Maude uses a very expressive version of equational logic, namely membership equational logic (*MEL*) [2,13], which, in addition to equations, allows the statement of membership assertions characterizing the elements of a sort. Below we present the logic and how its specifications are represented as Maude functional modules.

2.1 Membership Equational Logic

A *signature* in *MEL* is a triple (K,Σ,S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1\dots k_n,k}\}_{(k_1\dots k_n,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1,\dots,x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements. *MEL* atomic formulas are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership assertions* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)A_0 \Leftarrow A_1 \wedge \dots \wedge A_n$, where each A_i is either an equation or a membership assertion, and X is a set of K-kinded variables containing all the variables in the A_i. Order-sorted notation $s_1 < s_2$ (with $s_1,s_2 \in S_k$ for some kind k) can be used to abbreviate the conditional membership $(\forall x : k)x : s_2 \Leftarrow x : s_1$. A *specification* is a pair (Σ,E), where E is a set of sentences in *MEL* over the signature Σ.

Models of *MEL* specifications are called algebras. A Σ-*algebra* $\mathcal{A}$ consists of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times\cdots\times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1\dots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$, with the meaning that the elements in sorts are well-defined, whereas elements in a kind not having a sort are undefined or error elements. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$.

A specification (Σ,E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [2,13] for a detailed presentation of (Σ,E)-algebras, sound and complete deduction rules, initial algebras, and specification morphisms.

Since the *MEL* specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \to t'$, exactly when $\mathcal{A} \models t = t'$, i.e., when $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$. This is the notation we will use in the inference rules and debugging trees studied in Sect. 3.

2.2 Representation in Maude

Maude functional modules, introduced with syntax `fmod ... endfm`, are executable *MEL* specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification.

In a functional module we can declare sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly, and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat] = [Nat]`.

An operator declaration like[3]

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

2.3 A Buggy Example: Non-empty Sorted Lists

Let us see a simple example showing how to specify sorted lists of natural numbers in Maude. The following module includes the predefined module `NAT` defining the natural numbers.

```
(fmod SORTED-NAT-LIST is
  pr NAT .
```

We introduce sorts for non-empty lists and sorted lists. We identify a natural number with a sorted list with a single element by means of a subsort declaration.

```
  sorts NatList SortedNatList .
  subsorts Nat < SortedNatList < NatList .
```

The lists that have more than one element are built by means of the associative juxtaposition operator `__`.

```
  op __ : NatList NatList -> NatList [ctor assoc] .
```

We define now when a list (with more than one element) is sorted by means of a membership assertion. It states that the first number must be smaller than the first of the rest of the list, and that the rest of the list must also be sorted.

[3] The underscores indicate the places where the arguments appear in mixfix syntax.

```
  vars E E' : Nat .   var L : NatList .   var OL : SortedNatList .
  cmb [olist] : E L : SortedNatList if E <= head(L) /\ L : SortedNatList .
```

The definition of the `head` function distinguishes between lists with a single element and longer ones.

```
  op head : NatList -> Nat .
  eq [hd1] : head(E) = E .
  eq [hd2] : head(L E) = E .
```

We also define a sort function which sorts a list by successively inserting each natural number in the appropriate position in the sorted sublist formed by the numbers previously considered.

```
  op insertion-sort : NatList -> SortedNatList .
  op insert-list : SortedNatList Nat -> SortedNatList .
  eq [is1] : insertion-sort(E) = E .
  eq [is2] : insertion-sort(E L) = insert-list(insertion-sort(L), E) .
```

The function `insert-list` distinguishes several cases. If the list has only one number, the function checks if it is bigger than the number being inserted, and returns the sorted list. If the list has more than one element, the function checks that the list is previously sorted; if the number being inserted is smaller than the first of the list, it is located as the (new) first element, while if it is bigger we keep the first element and recursively insert the element in the rest of the list.

```
  ceq [il1] : insert-list(E, E') = E' E if E' < E .
  eq [il2] : insert-list(E, E') = E E' [owise] .
  ceq [il3] : insert-list(E OL, E') = E E' OL
   if E' <= E /\ E OL : SortedNatList .
  ceq [il4] : insert-list(E OL, E') = E insert-list(OL, E')
   if E OL : SortedNatList [owise] .
 endfm)
```

Now, we can reduce a term in this module. For example, we can try to sort the list `3 4 7 6` with

```
 Maude> (red insertion-sort(3 4 7 6) .)
 result SortedNatList : 6 3 4 7
```

But... the list obtained *is not sorted!* Moreover, Maude infers that *it is sorted*. Did you notice the bugs? We will show how to use the debugger in Sect. 4.3 to detect them.

3 Declarative Debugging of Maude Functional Modules

We describe how to build the debugging trees for *MEL* specifications. Detailed proofs can be found in [4].

(**Reflexivity**) $$\frac{}{e \to e}\ (Rf)$$

(**Transitivity**) $$\frac{e_1 \to e' \qquad e' \to e_2}{e_1 \to e_2}\ (Tr)$$

(**Congruence**) $$\frac{e_1 \to e_1' \quad \ldots \quad e_n \to e_n'}{f(e_1, \ldots, e_n) \to f(e_1', \ldots, e_n')}\ (Cong)$$

(**Subject Reduction**) $$\frac{e \to e' \quad e' : s}{e : s}\ (SRed)$$

(**Membership**) $$\frac{\{\theta(u_i) \to t_i \leftarrow \theta(u_i')\}_{1 \le i \le n} \quad \{\theta(v_j) : s_j\}_{1 \le j \le m}}{\theta(e) : s}\ (Mb)$$
$$\text{if } e : s \Leftarrow u_1 = u_1' \wedge \cdots \wedge u_n = u_n' \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$$

(**Replacement**) $$\frac{\{\theta(u_i) \to t_i \leftarrow \theta(u_i')\}_{1 \le i \le n} \quad \{\theta(v_j) : s_j\}_{1 \le j \le m}}{\theta(e) \to \theta(e')}\ (Rep)$$
$$\text{if } e \to e' \Leftarrow u_1 = u_1' \wedge \cdots \wedge u_n = u_n' \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m$$

Fig. 1. Semantic calculus for Maude functional modules

3.1 Proof Trees

Before defining the debugging trees employed in our declarative debugging framework we need to introduce the semantic rules defining the specification semantics. The inference rules of the calculus can be found in Fig. 1, where θ denotes a substitution.

They are an adaptation to the case of Maude functional modules of the deduction rules for *MEL* presented in [13]. The notation $\theta(u_i) \to t_i \leftarrow \theta(u_i')$ must be understood as a shortcut for $\theta(u_i) \to t_i$, $\theta(u_i') \to t_i$. We assume the existence of an *intended interpretation* I of the specification, which is a Σ-algebra corresponding to the model that the user had in mind while writing the statements E, i.e., the user expects that $I \models e \to e'$, $I \models e : s$ for each reduction $e \to e'$ and membership $e : s$ computed w.r.t. the specification (Σ, E). As a Σ-algebra, I must satisfy the following proposition:

Proposition 1. *Let $S = (\Sigma, E)$ be a MEL specification and let $\mathcal{A}$ be any Σ-algebra. If $e \to e'$ (respectively $e : s$) can be deduced using the semantic calculus rules* reflexivity, transitivity, congruence, *or* subject reduction *using premises that hold in $\mathcal{A}$, then $\mathcal{A} \models e \to e'$ (respectively $\mathcal{A} \models e : s$).*

Observe that this proposition cannot be extended to the *membership* and *replacement* inference rules, where the correctness of the conclusion depends not only on the calculus but also on the associated specification statement, which could be wrong.

We will say that $e \to e'$ (respectively $e : s$) is *valid* when it holds in I, and *invalid* otherwise. Declarative debuggers rely on some external oracle, normally the user, in order to obtain information about the validity of some nodes in the debugging tree. The concept of validity can be extended to distinguish *wrong equations* and *wrong* *membership axioms*, which are those specification pieces that can deduce something invalid from valid information.

Definition 1. *Let $R \equiv (af \Leftarrow u_1 = u'_1 \wedge \cdots \wedge u_n = u'_n \wedge v_1 : s_1 \wedge \cdots \wedge v_m : s_m)$, where af denotes an atomic formula, that is, either an oriented equation or a membership axiom in a specification S. Then:*

- $\theta(R)$ *is a* wrong equation instance *(respectively, a* wrong membership axiom instance*) w.r.t. an intended interpretation I when*
  - *There exist $t_1, \ldots, t_n$ such that $I \models \theta(u_i) \to t_i$, $I \models \theta(u'_i) \to t_i$ for $i = 1 \ldots n$.*
  - *$I \models \theta(v_j) : s_j$ for $j = 1 \ldots m$.*
  - *$\theta(af)$ does not hold in I.*
- *R is a* wrong equation *(respectively, a* wrong membership axiom*) if it admits some wrong instance.*

It will be convenient to represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. For example, the proof tree depicted in Fig. 2 corresponds to the result of the reduction in the specification for sorted lists described at the end of Sect. 2.3. For obvious reasons, the operation names have been abbreviated in a self-explanatory way; furthermore, each node corresponding to an instance of the *replacement* or *membership* inference rules has been labelled with the label of the equation or membership statement which is being applied.

In declarative debugging we are specially interested in *buggy nodes* which are invalid nodes with all its children valid. The following proposition characterizes buggy nodes in our setting.

Proposition 2. *Let N by a buggy node in some proof tree in the calculus of Fig. 1 w.r.t. an intended interpretation I. Then:*

1. *N is the consequence of either a* membership *or a* replacement *inference step.*
2. *The equation associated to N is a wrong equation or a wrong membership axiom.*

3.2 Abbreviated Proof Trees

Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy:

Input: A tree T with an invalid root.
Output: A buggy node in T.
Description: Consider the root N of T. There are two possibilities:
- If all the children of N are valid, then finish identifying N as buggy.
- Otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T. As an easy consequence, the following result holds:

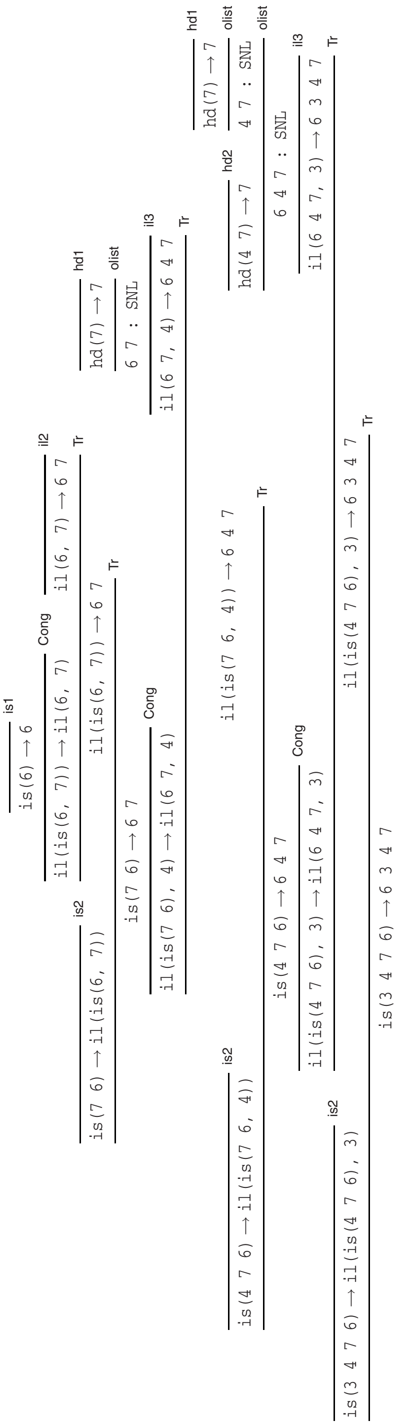


Fig. 2. Proof tree for the sorted lists example

Proposition 3. *Let T be a proof tree with an invalid root. Then there exists a buggy node $N \in T$ such that all the ancestors of N are invalid.*

However, we will not use the proof tree T as debugging tree, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT if the proof tree T is clear from the context. The reason for preferring the APT to the original proof tree is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. In particular the APT contains only nodes related to the *replacement* and *membership* inferences using statements included in the specification, which are the only possible buggy nodes as Proposition 2 indicates. Fig. 3 shows the definition of $APT(T)$. The T_i represent proof trees corresponding to the premises in some inferences.

The rule APT_1 keeps the root unaltered and employs the auxiliary function APT' to produce the children subtrees. APT' is defined in rules $APT_2 \ldots APT_8$. It takes a proof tree as input parameter and returns a forest $\{T_1, \ldots, T_n\}$ of APTs as result. The rules for APT' are assumed to be tried top-down, in particular APT_4 must not be applied if APT_3 is also applicable. It is easy to check that every node $N \in T$ that is the conclusion of a *replacement* or *membership* inference has its corresponding node $N' \in APT(T)$ labeled with the same abbreviation, and conversely, that for each $N' \in APT(T)$ different from the root, there is a node $N \in T$, which is the conclusion of a *replacement* or *membership* inference. In particular the node associated to $e_1 \to e_2$ in the righthand side of APT_3 is the node $e_1 \to e'$ of the proof tree T, which is not included in the $APT(T)$. We have chosen to introduce $e_1 \to e_2$ instead of simply $e_1 \to e'$ in the $APT(T)$ as a pragmatic way of simplifying the structure of the APTs, since e_2 is obtained from e' and hence likely simpler (the root of the tree T' in APT_3 must be necessarily of the form $e' \to e_2$ by the structure of the inference rule for transitivity in Fig. 1). We will formally state below (Theorem 1) that skipping $e_1 \to e'$ and introducing instead $e_1 \to e_2$ is safe from the point of view of the debugger.

Although $APT(T)$ is no longer a proof tree we keep the inference labels (Rep) and (Mb), assuming implicitly that they contain a reference to the equation or membership axiom used at the corresponding step in the original proof trees. It will be used by the debugger in order to single out the incorrect fragment of specification code.

Before proving the correctness and completeness of the debugging technique we need some auxiliary results. The first one indicates that APT' transforms a tree with invalid root into a set of trees such that at least one has an invalid root. We denote the root of a tree T as $root(T)$.

Lemma 1. *Let T be a proof tree such that $root(T)$ is invalid w.r.t. an intended interpretation I. Then there is some $T' \in APT'(T)$ such that $root(T')$ is invalid w.r.t. I.*

An immediate consequence of this result is the following:

Lemma 2. *Let T be a proof tree and $APT(T)$ its abbreviated proof tree. Then the root of $APT(T)$ cannot be buggy.*

The next theorem guarantees the correctness and completeness of the debugging technique based on $APTs$:

$$(\mathbf{APT}_1)\ APT\left(\frac{T_1 \ldots T_n}{af}(R)\right) = \frac{APT'\left(\frac{T_1 \ldots T_n}{af}(R)\right)}{af} \text{ (with } (R) \text{ any inference rule)}$$

$$(\mathbf{APT}_2)\ APT'\left(\frac{}{e \to e}(Rf)\right) = \emptyset$$

$$(\mathbf{APT}_3)\ APT'\left(\frac{\frac{T_1 \ldots T_n}{e_1 \to e'}(Rep) \quad T'}{e_1 \to e_2}(Tr)\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)\ APT'(T')}{e_1 \to e_2}(Rep)\right\}$$

$$(\mathbf{APT}_4)\ APT'\left(\frac{T_1 \quad T_2}{e_1 \to e_2}(Tr)\right) = \{APT'(T_1),\ APT'(T_2)\}$$

$$(\mathbf{APT}_5)\ APT'\left(\frac{T_1 \ldots T_n}{e_1 \to e_2}(Cong)\right) = \{APT'(T_1), \ldots, APT'(T_n)\}$$

$$(\mathbf{APT}_6)\ APT'\left(\frac{T_1 \quad T_2}{e : s}(SRed)\right) = \{APT'(T_1),\ APT'(T_2)\}$$

$$(\mathbf{APT}_7)\ APT'\left(\frac{T_1 \ldots T_n}{e : s}(Mb)\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)}{e : s}(Mb)\right\}$$

$$(\mathbf{APT}_8)\ APT'\left(\frac{T_1 \ldots T_n}{e_1 \to e_2}(Rep)\right) = \left\{\frac{APT'(T_1) \ldots APT'(T_n)}{e_1 \to e_2}(Rep)\right\}$$

Fig. 3. Transforming rules for obtaining abbreviated proof trees

Theorem 1. *Let S be a specification, I its intended interpretation, and T a finite proof tree with invalid root. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong equation in S.*

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user answers correctly all the questions about the validity of the *APT* nodes asked by the debugger (see Sect. 4.1).

The tree depicted in Fig. 4 is the abbreviated proof tree corresponding to the proof tree in Fig. 2, using the same conventions w.r.t. abbreviating the operation names. The debugging example described in Sect. 4.3 will be based on this abbreviated proof tree.

4 Using the Debugger

Before describing the basics of the user interaction with the debugger, we make explicit what is assumed about the modules introduced by the user; then we present the available commands and how to use them to debug the buggy example introduced in Sect. 2.3.

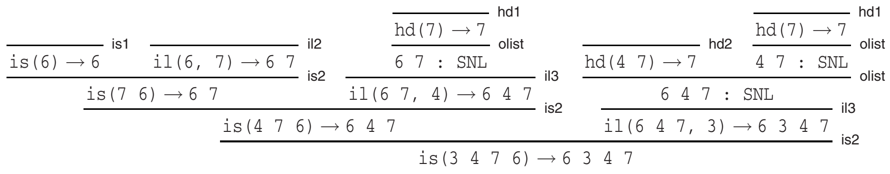


Fig. 4. Abbreviated proof tree for the sorted lists example

4.1 Assumptions

Since we are debugging Maude functional modules, they are expected to satisfy the appropriate executability requirements, namely, the specifications have to be terminating, confluent, and sort-decreasing.

One interesting feature of our tool is that the user is allowed to trust some statements, by means of labels applied to the suspicious statements. This means that the unlabeled statements are assumed to be correct. A trusted statement is treated in the implementation as the *reflexivity*, *transitivity*, and *congruence* rules are treated in the *APT* transformation described in Fig. 3; more specifically, an instance of the *membership* or *replacement* inference rules corresponding to a trusted statement is collapsed in the abbreviated proof tree.

In order to obtain a nonempty abbreviated proof tree, the user must have labeled some statements (all with different labels); otherwise, everything is assumed to be correct. In particular, the buggy statement must be labeled in order to be found. When not all the statements are labeled, the correctness and completeness results shown in Sect. 3 are conditioned by the goodness of the labeling for which the user is responsible.

Although the user can introduce a module importing other modules, the debugging process takes place in the *flattened* module. However, the debugger allows the user to trust a whole imported module.

As mentioned in the introduction, navigation of the debugging tree takes place by asking questions to an external oracle, which in our case is either the user or another module introduced by the user. In both cases the answers are assumed to be correct. If either the module is not really correct or the user provides an incorrect answer, the result is unpredictable. Notice that the information provided by the correct module need not be complete, in the sense that some functions can be only partially defined. In the same way, the signature of the correct module need not coincide with the signature of the module being debugged. If the correct module cannot help in answering a question, the user may have to answer it.

4.2 Commands

The debugger is initiated in Maude by loading the file `dd.maude` (available from `http://maude.sip.ucm.es/debugging`), which starts an input/output loop that allows the user to interact with the tool.

As we said in the introduction, the generated proof tree can be navigated by using two different strategies, namely, *top-down* and *divide and query*, the latter being the default one. The user can switch between them by using the commands `(top-down strategy .)` and `(divide-query strategy .)`. If a module with correct definitions is used to reduce the number of questions, it must be indicated before starting the debugging process with the command `(correct module MODULE-NAME .)`. Moreover, the user can trust all the statements in several modules with the command `(trust[*] MODULE-NAMES-LIST .)` where `*` means that modules are considered flattened.

Once we have selected the strategy and, optionally, the module above, we start the debugging process with the command[4]

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM .)
```

If we want to debug only with a subset of the labeled statements, we use the command

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM with LABELS .)
```

where `LABELS` is the set of suspicious equation and membership axiom labels that must be taken into account when generating the debugging tree.

In the same way, we can debug a membership inference with the commands

```
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT with LABELS .)
```

How the process continues depends on the selected strategy. In case the top-down strategy is selected, several nodes will be displayed in each question. If there is an invalid node, we must select one of them with the command `(node N .)`, where `N` is the identifier of this wrong node. If all the nodes are correct, we type `(all valid .)`.

In the divide and query strategy, each question refers to one inference that can be either correct or wrong. The different answers are transmitted to the debugger with the commands `(yes .)` and `(no .)`. Instead of just answering `yes`, we can also *trust* some statements on the fly if, once the process has started, we decide the bug is not there. To trust the current statement we type the command `(trust .)`.

Finally, we can return to the previous state in both strategies by using the command `(undo .)`.

4.3 Sorted Lists Revisited

We recall from Sect. 2.3 that if we try to sort the list `3 4 7 6`, we obtain the strange result

```
Maude> (red insertion-sort(3 4 7 6) .)
result SortedNatList : 6 3 4 7
```

That is, the function returns an unsorted list, but Maude infers it is sorted. We can debug the buggy specification by using the command

```
Maude> (debug in SORTED-NAT-LIST : insertion-sort(3 4 7 6) -> 6 3 4 7 .)
```

[4] If no module name is given, the current module is used by default.

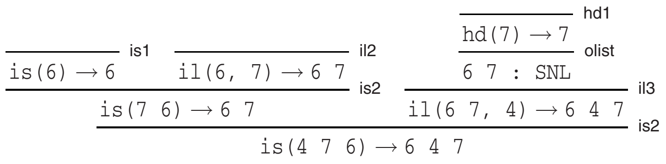


Fig. 5. Abbreviated proof tree after the first question

With this command the debugger computes the tree shown in Fig. 4. Since the default navigation strategy is divide and query, the first question is

```
Is this transition (associated with the equation is2) correct?
insertion-sort(4 7 6) -> 6 4 7
Maude> (no .)
```

We expect `insertion-sort` to order the list, so we answer negatively and the subtree in Fig. 5 is selected to continue the debugging. The next question is

```
Is this transition (associated with the equation is2) correct?
insertion-sort(7 6) -> 6 7
Maude> (yes .)
```

Since the list is sorted, we answer `yes`, so this subtree is deleted (Fig. 6 left). The debugger asks now the question

```
Is this membership (associated with the membership olist) correct?
6 7 : SortedNatList
Maude> (yes .)
```

This sort is correct, so this subtree is also deleted (Fig. 6 right) and the next question is prompted.

```
Is this transition (associated with the equation il3) correct?
insert-list(6 7, 4) -> 6 4 7
Maude> (no .)
```

With this information the debugger selects the subtree and, since it is a leaf, it concludes that the node is associated with the buggy equation.

```
The buggy node is:
insert-list(6 7, 4) -> 6 4 7
With the associated equation: il3
```

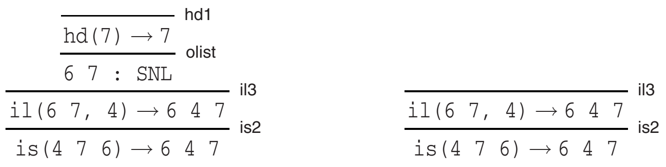


Fig. 6. Abbreviated proof trees after the second and third questions

That is, the debugger points to the equation `il3` as buggy. If we examine it

```
  ceq [il3] : insert-list(E OL, E') = E E' OL
   if E' <= E /\ E OL : SortedNatList .
```

we can see that the order of `E` and `E'` in the righthand side is wrong and we can proceed to fix it appropriately.

We can check the fixed function by sorting again the list `3 4 7 6`. We obtain now the sorted list `3 4 6 7`. Then, we have solved one problem, but if we reduce the unsorted list `6 3 4 7`

```
Maude> (red 6 3 4 7 .)
result SortedNatList : 6 3 4 7
```

we can see that Maude continues assigning to it an incorrect sort.

We can check this inference by using the command

```
Maude> (debug 6 3 4 7 : SortedNatList .)
```

The first question the debugger prompts is

```
Is this membership (associated with the membership olist) correct?
3 4 7 : SortedNatList
Maude> (yes .)
```

Of course, this list is sorted. The following question is

```
Is this transition (associated with the equation hd2) correct?
head(3 4 7) -> 7
Maude> (no .)
```

But the head of a list should be the first element, not the last one, so we answer `no`. With only these two questions the debugger prints

```
The buggy node is:
head(3 4 7) -> 7
With the associated equation: hd2
```

If we check the equation `hd2`, we can see that we take the element from the wrong side.

```
  eq [hd2] : head(L E) = E .
```

To debug this module we have used the default divide and query strategy. Let us check it now with the top-down strategy. We debug again the inference

```
insertion-sort(3 4 7 6) -> 6 3 4 7
```

in the initial module with the two errors. The first question asked in this case is

```
Is any of these nodes wrong?
Node 0 : insertion-sort(4 7 6) -> 6 4 7
Node 1 : insert-list(6 4 7, 3) -> 6 3 4 7
Maude> (node 0 .)
```

Both nodes are wrong, so we select, for example, the first one. The next question is

```
Is any of these nodes wrong?
Node 0 : insertion-sort(7 6) -> 6 7
Node 1 : insert-list(6 7, 4) -> 6 4 7
Maude> (node 1 .)
```

This time, only the second node is wrong, so we select it. The debugger prints now

```
Is any of these nodes wrong?
Node 0 : 6 7 : SortedNatList
Maude> (all valid .)
```

There is only one node, and it is correct, so we give this information to the debugger, and it detects the wrong equation.

```
The buggy node is:
insert-list(6 7, 4) -> 6 4 7
With the associated equation: il3
```

But remember that we chose a node randomly when the debugger showed two wrong nodes. What happens if we select the other one? The following question is printed.

```
Is any of these nodes wrong?
Node 0 : 6 4 7 : SortedNatList
Maude> (node 0 .)
```

Since this single node is wrong, we choose it and the debugger asks

```
Is any of these nodes wrong?
Node 0 : head(4 7) -> 7
Node 1 : 4 7 : SortedNatList
Maude> (node 0 .)
```

The first node is the only one erroneous, so we select it. With this information, the debugger prints

```
The buggy node is:
head(4 7) -> 7
With the associated equation: hd2
```

That is, the second path finds the other bug. In general, this strategy finds different bugs if the user selects different wrong nodes.

In order to prune the debugging tree, we can define a module specifying the sorting function `sort` in a correct, but inefficient, way. This module will define the functions `insertion-sort` and `insert-list` by means of `sort`.

```
(fmod CORRECT-SORTING is
  pr NAT .
  sorts NatList SortedNatList .
  subsorts Nat < SortedNatList < NatList .
  vars E E' : Nat .  vars L L' : NatList .  var OL : SortedNatList .
  op __ : NatList NatList -> NatList [ctor assoc] .
  cmb E E' : SortedNatList if E <= E' .
  cmb E E' L : SortedNatList if E <= E' /\ E' L : SortedNatList .
```

The sort function is defined by switching unsorted adjacent elements in all the possible cases for lists.

```
  op sort : NatList -> SortedNatList .
  ceq sort(L E E' L') = sort(L E' E L') if E' < E .
  ceq sort(L E E') = sort(L E' E) if E' < E .
  ceq sort(E E' L) = sort(E' E L) if E' < E .
  ceq sort(E E') = E' E if E' < E .
  eq sort(L) = L [owise] .
```

We now use sort to implement insertion-sort and insert-list.

```
  op insertion-sort : NatList -> SortedNatList .
  op insert-list : SortedNatList Nat -> SortedNatList .
  eq insertion-sort(L) = sort(L) .
  eq insert-list(OL, E) = sort(E OL) .
 endfm)
```

We can use this module to prune the debugging trees built by the debug commands if we previously introduce the command

```
 Maude> (correct module CORRECT-SORTING .)
```

Now, we try to debug the initial module (with two errors) again. In this example, all the questions about correct inferences have been pruned, so all the answers are negative. In general, the correct module does not have to be complete, so some correct inferences could be presented to the user.

```
 Maude> (debug in SORTED-NAT-LIST : insertion-sort(3 4 7 6) -> 6 3 4 7 .)

 Is this transition (associated with the equation il3) correct?
 insert-list(6 4 7, 3) -> 6 3 4 7
 Maude> (no .)

 Is this membership (associated with the membership olist) correct?
 6 4 7  : SortedNatList
 Maude> (no .)

 Is this transition (associated with the equation hd2) correct?
 head(4 7) -> 7
 Maude> (no .)
```

```
The buggy node is:
head(4 7) -> 7
With the associated equation: hd2
```

The correct module also improves the debugging of the membership. With only one question we obtain the buggy equation.

```
Maude> (debug in SORTED-NAT-LIST : 6 3 4 7 : SortedNatList .)

Is this transition (associated with the equation hd2) correct?
head(3 4 7) -> 7
Maude> (no .)

The buggy node is:
head(3 4 7) -> 7
With the associated equation: hd2
```

4.4 Implementation

Exploiting the fact that rewriting logic is reflective [6,8], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [7, Chap. 14], a feature that makes Maude remarkably extensible and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [7, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our declarative debugger for Maude functional modules, including its user interactions, is implemented in Maude itself, as an extension of Full Maude [7, Chap. 18]. As far as we know, this is the first declarative debugger implemented using such reflective techniques.

The implementation takes care of the two stages of generating and navigating the debugging tree. Since navigation is done by asking questions to the user, this stage has to handle the navigation strategy together with the input/output interaction with the user.

To build the debugging tree we use the facts that the equations defined in Maude functional modules are both *terminating* and *confluent*. Instead of creating the complete proof tree and then abbreviating it, we build the abbreviated proof tree directly.

The main function in the implementation of the debugging tree generation is called `createTree`. It receives the module where a wrong inference took place, a correct module (or the constant `noModule` when no such module is provided) to prune the tree, the term initially reduced, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial inference as the root of the tree and uses an auxiliary function `createForest` that, in addition to the arguments received by `createTree`, receives the module "cleaned" of suspicious statements, and generates the abbreviated forest corresponding to the reduction between the two terms passed as arguments. This transformed module is used to improve the efficiency of the tree construction, because we can use it to check if a term reaches its final form only using trusted equations, thus avoiding to build a tree that will be finally empty.

Regarding the navigation of the debugging tree, we have implemented two strategies. In the top-down strategy the selection of the next node of the debugging tree is done by the user, thus we do not need any function to compute it. The divide and query strategy used to traverse the debugging tree selects each time the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise. The function `searchBestNode` calculates this best node by searching for a subtree minimizing an appropriate function.

The technical report [4] provides a full explanation of this implementation, including the user interaction.

5 Conclusions and Future Work

In this paper we have developed the foundations of declarative debugging of executable *MEL* specifications, and we have applied them to implement a debugger for Maude functional modules. As far as we know, this is the first debugger implemented in the same language it debugs. This has been made possible by the reflective features of Maude. In our opinion, this debugger provides a complement to existing debugging techniques for Maude, such as tracing and term coloring. An important contribution of our debugger is the help provided by the tool in locating the buggy statements, assuming the user answers correctly the corresponding questions. The debugger keeps track of the questions already answered, in order to avoid asking the same question twice.

We want to improve the interaction with the user by providing a complementary graphical interface that allows the user to navigate the tree with more freedom. We are also studying how to handle the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness.

We plan to extend our framework by studying how to debug *system modules*, which correspond to rewriting logic specifications and have rules in addition to memberships and equations. These rules can be non-terminating and non-confluent, and thus behave very differently from the statements in the specifications we handle here. In this context, we also plan to study how to debug *missing answers* [14] in addition to the wrong answers we have treated thus far.

References

1. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract diagnosis of functional programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 1–16. Springer, Heidelberg (2003)
2. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
3. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: WCFLP 2005: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, pp. 8–13. ACM Press, New York (2005)

4. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: Declarative debugging of Maude functional modules. Technical Report 4/07, Dpto.Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2007), `http://maude.sip.ucm.es/debugging`
5. Caballero, R., Rodríguez-Artalejo, M.: DDT: A declarative debugging tool for functional-logic languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
6. Clavel, M.: Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications. CSLI Publications, Stanford University (2000)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
8. Clavel, M., Meseguer, J.: Reflection in conditional rewriting logic. Theoretical Computer Science 285(2), 245–288 (2002)
9. Comini, M., Levi, G., Meo, M.C., Vitiello, G.: Abstract diagnosis. Journal of Logic Programming 39(1-3), 43–93 (1999)
10. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing 5(2), 133–154 (1987)
11. MacLarty, I.: Practical Declarative Debugging of Mercury Programs. PhD thesis, University of Melbourne (2005)
12. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
13. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
14. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
15. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming 1997(3), 1–27 (1997)
16. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
17. Nilsson, H., Fritzson, P.: Algorithmic debugging of lazy functional languages. Journal of Functional Programming 4(3), 337–370 (1994)
18. Pope, B.: Declarative debugging with Buddha. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 273–308. Springer, Heidelberg (2005)
19. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
20. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
21. Takahashi, N., Ono, S.: DDS: A declarative debugging system for functional programs. Systems and Computers in Japan 21(11), 21–32 (1990)
22. Tessier, A., Ferrand, G.: Declarative diagnosis in the CLP scheme. In: Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project), pp. 151–174. Springer, Heidelberg (2000)
23. Wadler, P.: Why no one uses functional languages. SIGPLAN Not. 33(8), 23–27 (1998)

Declarative Debugging of Rewriting Logic Specifications[⋆]

Adrian Riesco, Alberto Verdejo, Rafael Caballero, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es, {alberto,rafa,narciso}@sip.ucm.es

Abstract. Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error by building a tree representing this computation and guiding the user through it to find the wrong statement. This paper presents the fundamentals for the declarative debugging of rewriting logic specifications, realized in the Maude language, where a wrong computation can be a reduction, a type inference, or a rewrite. We define appropriate debugging trees obtained as the result of collapsing in proof trees all those nodes whose correctness does not need any justification. Since these trees are obtained from a suitable semantic calculus, the correctness and completeness of the debugging technique can be formally proved. We illustrate how to use the debugger by means of an example and succinctly describe its implementation in Maude itself thanks to its reflective and metalanguage features.

1 Introduction

In this paper we present a declarative debugger for *Maude specifications*, including equational functional specifications and concurrent systems specifications. Maude [10] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in *rewriting logic* [14], a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. This logic is an extension of equational logic; in particular, Maude *functional modules* correspond to specifications in *membership equational logic* [1, 15], which, in addition to equations, allows the statement of *membership axioms* characterizing the elements of a sort. In this way, Maude makes possible the faithful specification of data types (like sorted lists or search trees) whose data are not only defined by means of constructors, but also by the satisfaction of additional properties. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. Maude *system modules* are used to define specifications in this logic.

[⋆] Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [10, Chap. 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of applications of statements that take place. The same ideas have been applied to the functional paradigm by the tracer *Hat* [9], where a graph constructed by graph rewriting is proposed as suitable trace structure. Term coloring consists in printing with different colors the operators used to build a term that does not fully reduce. The Maude debugger allows to define break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered. There, we can see the current term and execute the next rewrite with tracing turned on. The Maude debugger has as a disadvantage that, since it is based on the trace, it shows to the user every small step obtained by using a single statement. Thus the user can lose the general view of the *proof* of the incorrect inference that produced the wrong result. That is, when the user detects an unexpected statement application it is difficult to know where the incorrect inference started. Here we present a different approach based on declarative debugging that solves this problem for Maude specifications.

Declarative debugging, also known as algorithmic debugging, was first introduced by E. Y. Shapiro [23]. It has been widely employed in the logic [12, 16, 25], functional [18, 19, 20], multi-paradigm [3, 7, 13], and object-oriented [4] programming languages. Declarative debugging starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. The declarative debugging scheme [17] uses a *debugging tree* as logical representation of the computation. Each node in the tree represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. A buggy node is a node containing an erroneous result, but whose children have all correct results. Hence, a buggy node has produced an erroneous output from correct inputs and corresponds to an erroneous fragment of code, which is pointed out as an error. From an explanatory point of view, declarative debugging can be described as consisting of two stages, namely the debugging tree *generation* and its *navigation* following some suitable strategy [24].

The application of declarative debugging to Maude functional modules was already studied in our previous papers [5, 6]. The executability requirements of Maude functional modules mean that they are assumed to be confluent, terminating, and sort-decreasing[1] [10]. These requirements are assumed in the form of the questions appearing in the debugging tree. In this paper, we considerably extend that work by also considering system modules. Now, since the specifications described in this kind of modules can be non-terminating and non-confluent, their handling must be quite different.

The debugging process starts with an incorrect computation from the initial term to an unexpected one. The debugger then builds an appropriate debugging

[1] All these requirements must be understood *modulo* some axioms such as associativity and commutativity that are associated to some binary operations.

tree which is an abbreviation of the corresponding proof tree obtained by applying the inference rules of membership equational logic and rewriting logic. The abbreviation consists in collapsing those nodes whose correctness does not need any justification, such as those related with transitivity or congruence. Since the questions are located in the debugging tree, the answers allow the debugger to discard a subset of the questions, leading and shortening the debugging process. In the case of functional modules, the questions have the form “Is it correct that T fully reduces to T'?”, which in general are easier to answer. However, in the absence of confluence and termination, these questions do not make sense; thus, in the case of system modules, we have decided to develop two different trees whose nodes produce questions of the form “Is it correct that T is rewritten to T'?” where the difference consists in the number of steps involved in the rewrite. While one of the trees refers only to one-step rewrites, which are often easier to answer, the other one can also refer to many-steps rewrites that, although may be harder to answer, in general discard a bigger subset of nodes. The user, depending on the debugged specification or his “ability” to answer questions involving several rewrite steps, can choose between these two kinds of trees.

Moreover, exploiting the fact that rewriting logic is *reflective* [11], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [10, Chap. 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [10, Chap. 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [10, Chap. 18], that includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems, that can also be debugged. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

The rest of the paper is structured as follows. Sect. 2 provides a summary of the main concepts of both membership equational logic and rewriting logic, and how their specifications are realized in Maude functional and system modules, respectively. Sect. 3 describes the theoretical foundations of the debugging trees for inferences in both logics. Sect. 4 shows how to use the debugger by means of an example, while Sect. 5 comments some aspects of the Maude implementation. Finally, Sect. 6 concludes and mentions some future work.

Detailed proofs of the results, additional examples, and much more information about the implementation can be found in the technical report [21], which, together with the Maude source files for the debugger, is available from the webpage `http://maude.sip.ucm.es/debugging`

2 Rewriting Logic and Maude

As mentioned in the introduction, Maude modules are executable rewriting logic specifications. Rewriting logic [14] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [1, 15], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort.

2.1 Membership Equational Logic

A *signature* in *MEL* is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of *MEL* are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in *MEL* over the signature Σ.

Models of *MEL* specifications are Σ*-algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$.

A *MEL* specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [1, 15] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules (that we adapt to our purposes in Fig. 1 in Sect. 3.1), as well as the construction of initial and free algebras. Since the *MEL* specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness, their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A} \models t \to t'$, exactly when $\mathcal{A} \models t = t'$, i.e., when $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$. The notation we will use in the inference rules studied in Sect. 3 for this situation is $u \downarrow v$.

2.2 Rewriting Logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible. A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses *MEL*, as described in the previous section. A rule in R has the general conditional form[2]

$$(\forall X)\ t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$$

where the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [2, 14], that we have adapted to our purposes.

Models of rewrite theories are called $\mathcal{R}$-*systems* in [14]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Sect. 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. Then, if we keep in this context the notation $\mathcal{A}$ to denote an $\mathcal{R}$-system, a rewrite $t \Rightarrow t'$ is satisfied by $\mathcal{A}$, denoted $\mathcal{A} \models t \Rightarrow t'$, when there is a transition $[\![t]\!]_{\mathcal{A}} \Rightarrow_{\mathcal{A}} [\![t']\!]_{\mathcal{A}}$ in the system between the corresponding meanings of both sides of the rewrite, where $\Rightarrow_{\mathcal{A}}$ will be our notation for such transitions. The rewriting logic deduction rules introduced in [14] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial and free models; see [14] for details.

2.3 Maude Modules

Maude functional modules [10, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification. In a functional module we can declare sorts

[2] Note that we use the notation $\Rightarrow$ for rewrites (as in Maude) and $\rightarrow$ for *oriented* equations and reductions using such equations. Other papers on rewriting logic use instead the notation $\rightarrow$ for rewrites.

(by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude system modules [10, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

The executability requirements for equations and memberships are confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [10] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [10, 26].

The following section describes an example of a Maude system module with both equations and rules.

2.4 An Example: Knight's Tour Problem

A knight's tour is a journey around the chessboard in such a way that the knight lands on each square exactly once. The legal move for a knight is two spaces in one direction, then one in a perpendicular direction. We want to solve the problem for a 3×4 chessboard with the knight initially located in one corner.

We represent positions in the chessboard as pairs of integers and journeys as lists of positions.

```
(mod KNIGHT is
  protecting INT .
  sorts Position Movement Journey Problem .
  subsort Position < Movement .
  subsorts Position < Journey < Problem .
  op [_,_] : Int Int -> Position .
  op nil : -> Journey .
  op __ : Journey Journey -> Journey [assoc id: nil] .
  vars N X Y : Int .  vars P Q : Position .  var  J : Journey .
```

The term `move P` represents a position reachable from position `P`. Since the reachable positions are not unique, this operation is defined by means of rewrite rules, instead of equations. The reachable positions can be outside the chessboard, so we define the operation `legal`, that checks if a position is inside the 3×4 chessboard.

```
  op move_ : Position -> Movement .
  rl [mv1] : move [X, Y] => [X + 2, Y + 1] .
  ...
  rl [mv8] : move [X, Y] => [X - 1, Y - 2] .
  op legal : Position -> Bool .
  eq [leg] : legal([X, Y]) = X >= 1 and Y >= 1 and X <= 3 and Y <= 4 .
```

The function `contains(J, P)` checks if position `P` occurs in the journey `J`.

```
  op contains : Journey Position -> Bool .
  eq [con1] : contains(P J, P) = true .
  eq [con2] : contains(J, P) = false [otherwise] .
```

`knight(N)` represents a journey where the knight has performed `N` hops. When no hops are taken, the knight remains at the first position `[1, 1]`. When `N > 0` the problem is recursively solved (using backtracking in an implicit way) as follows: first a legal journey of `N - 1` steps is found, then a new hop from the last position of that journey is performed, and finally it is checked that this last hop is legal and compatible with the other ones.

```
  op knight : Nat -> Problem .
  rl [k1] : knight(0) => [1, 1] .
  crl [k2] : knight(N) => J P Q
   if N > 0
   /\ knight(N - 1) => J P
   /\ move P => Q
   /\ legal(Q)
   /\ not(contains(J P, Q)) .
endm)
```

The solution to the 3×4 chessboard can be found by looking for a journey with 11 hops, but we obtain the following unexpected, wrong result, where the journey contains repeated positions. We will show how to debug it in Sect. 4.

```
Maude> (rew knight(11) .)
result Journey :
   [1,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3][3,1][2,3]
```

3 Debugging Trees for Maude Specifications

Now we will describe debugging trees for both *MEL* specifications and rewriting logic specifications. Since a *MEL* specification coincides with a rewrite theory with an empty set of rules, our treatment will simply be at the level of rewrite theories. Our proof and debugging trees will include statements for reductions $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$, and in the following sections we will describe how to build the debugging trees from the proof trees taking into account each kind of statement.

3.1 Proof Trees

Before defining the debugging trees employed in our declarative debugging framework we introduce the semantic rules defining the semantics of a rewrite theory $\mathcal{R}$. The inference rules of the calculus can be found in Fig. 1, where θ denotes a substitution. The rules allow to deduce statements of the three kinds and are an adaptation of the rules presented in [1, 15] for *MEL* and in [2, 14] for rewriting logic. With respect to *MEL*, because of the executability assumptions, we have a more operational interpretation of the equations, which are oriented from left to right. With respect to rewriting logic, we work with terms (as in [2]) instead of equivalence classes of terms (as in [14]); moreover, unlike [2], replacement is not nested. Both changes make the logical representation closer to the way the Maude system operates. As usual, we represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. We assume that the inference labels ($Rep_{\Rightarrow}$), ($Rep_{\rightarrow}$), and (Mb) decorating the inference steps contain information about the particular rewrite rule, equation, and membership axiom, respectively, applied during the inference. This information will be used by the debugger in order to present to the user the incorrect fragment of code causing the error.

In our debugging framework we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. The intended interpretation must be an $\mathcal{R}$-system corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user expects that $\mathcal{I} \models t \Rightarrow t'$, $\mathcal{I} \models t \rightarrow t'$, and $\mathcal{I} \models t : s$ for each rewrite $t \Rightarrow t'$, reduction $t \rightarrow t'$, and membership $t : s$ computed w.r.t. the specification $\mathcal{R}$. We will say that a statement $t \Rightarrow t'$ (respectively $t \rightarrow t'$, $t : s$) is *valid* when it holds in $\mathcal{I}$, and *invalid* otherwise. Declarative debuggers rely on some external oracle, normally the user, in order to obtain information about the validity of some nodes in the debugging tree. The concept of validity can be extended to distinguish *wrong rules*, *wrong equations*, and *wrong membership axioms*, which are those specification pieces that can deduce something invalid from valid information.

Definition 1. *Let* $r \equiv (af \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k)$ *where* af *denotes an atomic formula, that is,* r *is either a rewrite rule, an oriented equation, or a membership axiom (in the last two cases* $l = 0$*) in some rewrite theory* $\mathcal{R}$*. Then:*

– $\theta(r)$ *is a* wrong rewrite rule instance *(respectively* wrong equation instance *and* wrong membership axiom instance*) w.r.t. an intended interpretation* $\mathcal{I}$ *when*
  1. *There exist terms* $t_1, \ldots, t_n$ *such that* $\mathcal{I} \models \theta(u_i) \rightarrow t_i$, $\mathcal{I} \models \theta(u'_i) \rightarrow t_i$ *for* $i = 1 \ldots n$.
  2. $\mathcal{I} \models \theta(v_j) : s_j$ *for* $j = 1 \ldots m$.
  3. $\mathcal{I} \models \theta(w_k) \Rightarrow \theta(w'_k)$ *for* $k = 1 \ldots l$.
  4. $\theta(af)$ *does not hold in* $\mathcal{I}$.

– r *is a* wrong rewrite rule *(respectively,* wrong equation *and* wrong membership axiom*) if it admits some wrong instance.*

(**Reflexivity**)

$$\frac{}{t \Rightarrow t}\,{\scriptstyle (Rf_{\Rightarrow})} \qquad\qquad \frac{}{t \rightarrow t}\,{\scriptstyle (Rf_{\rightarrow})}$$

(**Transitivity**)

$$\frac{t_1 \Rightarrow t' \qquad t' \Rightarrow t_2}{t_1 \Rightarrow t_2}\,{\scriptstyle (Tr_{\Rightarrow})} \qquad\qquad \frac{t_1 \rightarrow t' \qquad t' \rightarrow t_2}{t_1 \rightarrow t_2}\,{\scriptstyle (Tr_{\rightarrow})}$$

(**Congruence**)

$$\frac{t_1 \Rightarrow t'_1 \quad \ldots \quad t_n \Rightarrow t'_n}{f(t_1, \ldots, t_n) \Rightarrow f(t'_1, \ldots, t'_n)}\,{\scriptstyle (Cong_{\Rightarrow})} \qquad \frac{t_1 \rightarrow t'_1 \quad \ldots \quad t_n \rightarrow t'_n}{f(t_1, \ldots, t_n) \rightarrow f(t'_1, \ldots, t'_n)}\,{\scriptstyle (Cong_{\rightarrow})}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \;\{\theta(v_j) : s_j\}_{j=1}^{m} \;\{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(t) \Rightarrow \theta(t')}\,{\scriptstyle (Rep_{\Rightarrow})}$$

if $t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) \rightarrow \theta(t')}\,{\scriptstyle (Rep_{\rightarrow})} \text{ if } \; t \rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$$

(**Equivalence Class**) (**Subject Reduction**)

$$\frac{t \rightarrow t' \quad t' \Rightarrow t'' \quad t'' \rightarrow t'''}{t \Rightarrow t'''}\,{\scriptstyle (EC)} \qquad\qquad \frac{t \rightarrow t' \quad t' : s}{t : s}\,{\scriptstyle (SRed)}$$

(**Membership**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(t) : s}\,{\scriptstyle (Mb)} \qquad \text{if } \; t : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$$

Fig. 1. Semantic calculus for Maude modules

The general schema of [17] presents declarative debugging as the search of *buggy nodes* (invalid nodes with all children valid) in a debugging tree representing an erroneous computation. In our scheme instance, the proof trees constructed by the inferences of Fig. 1 seem natural candidates for debugging trees. Although this is a possible option, we will use instead a suitable abbreviation of these trees. This is motivated by the following result:

Proposition 1. *Let N be a buggy node in some proof tree in the calculus of Fig. 1 w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N is the result of either a* membership *or a* replacement *inference step.*
2. *The statement associated to N is either a wrong rewrite rule, a wrong equation, or a wrong membership axiom.*

Both points are a consequence of the definition of the semantic calculus. The first result states that all the inference steps different from *membership* and *replacement* are logically sound w.r.t. the definition of $\mathcal{R}$-system, i.e., they always produce valid results from valid premises. The second result can be checked by observing that any *membership* or *replacement* buggy node satisfies the requirements of Def. 1: the valid premises correspond to the points 1-3 of the definition, while the invalid conclusion fulfills the last point.

3.2 Abbreviated Proof Trees

Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy:

Input: A tree T with an invalid root.
Output: A buggy node in T.
Description: Consider the root N of T. There are two possibilities: if all the children of N are valid, then finish pointing out at N as buggy; otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T.

However, we will not use the proof tree T as debugging tree, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT if the proof tree T is clear from the context. The reason for preferring the APT to the original proof tree is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. In particular the APT essentially contains only nodes related to the *replacement* and *membership* inferences using statements included in the specification, which are the only possible buggy nodes as Prop. 1 indicates. Thus, in order to minimize the number of questions asked to the user the debugger should consider the validity of $(Rep_{\Rightarrow})$, $(Rep_{\rightarrow})$, or (Mb). The APT rules can be seen in Fig. 2.

The rules are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule of least number. As a matter of fact, the figure includes rules for two different possible $APTs$, which we call *one-step* abbreviated proof tree (in short $APT^o(T)$), defined by all the rules in the figure excluding $(\mathbf{APT}_4^m)$, and *many-steps* abbreviated proof tree (in short $APT^m(T)$), defined by all the rules in the figure excluding $(\mathbf{APT}_4^o)$. Analogously, we will use the notation $APT'^o(T)$ (resp. $APT'^m(T)$) for the subset of rules of APT' excluding $(\mathbf{APT}_4^m)$ (resp. $(\mathbf{APT}_4^o)$).

The one-step debugging tree follows strictly the idea of keeping only nodes corresponding to the *replacement* and *membership* inference rules. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rule for rewrites. The user will choose which debugging tree (one-step or many-steps) will be used for the declarative debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping

$$(\mathbf{APT}_1)\ \ APT\left(\frac{T_1 \dots T_n}{af}(R)\right) = \frac{APT'\left(\frac{T_1 \dots T_n}{af}(R)\right)}{af}$$

$$(\mathbf{APT}_2)\ \ APT'\left(\frac{}{t \Rrightarrow t}(Rf_{\Rrightarrow})\right) = \emptyset$$

$$(\mathbf{APT}_3)\ \ APT'\left(\frac{\frac{T_1 \dots T_n}{t_1 \to t'}(Rep_{\to}) \quad T'}{t_1 \to t_2}(Tr_{\to})\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)\ APT'(T')}{t_1 \to t_2}(Rep_{\to})\right\}$$

$$(\mathbf{APT}_4^o)\ \ APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}(Tr_{\Rightarrow})\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}_4^m)\ \ APT'\left(\frac{T_1 \quad T_2}{t_1 \Rightarrow t_2}(Tr_{\Rightarrow})\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t_1 \Rightarrow t_2}(Tr_{\Rightarrow})\right\}$$

$$(\mathbf{APT}_5)\ \ APT'\left(\frac{T_1 \dots T_n}{t_1 \Rrightarrow t_2}(Cong_{\Rrightarrow})\right) = APT'(T_1) \bigcup \dots \bigcup APT'(T_n)$$

$$(\mathbf{APT}_6)\ \ APT'\left(\frac{T_1 \quad T_2}{t : s}(SRed)\right) = APT'(T_1) \bigcup APT'(T_2)$$

$$(\mathbf{APT}_7)\ \ APT'\left(\frac{T_1 \dots T_n}{t : s}(Mb)\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)}{t : s}(Mb)\right\}$$

$$(\mathbf{APT}_8)\ \ APT'\left(\frac{T_1 \dots T_n}{t_1 \Rrightarrow t_2}(Rep_{\Rrightarrow})\right) = \left\{\frac{APT'(T_1) \dots APT'(T_n)}{t_1 \Rrightarrow t_2}(Rep_{\Rrightarrow})\right\}$$

$$(\mathbf{APT}_9)\ \ APT'\left(\frac{T' \quad \frac{T_1 \dots T_n}{t \Rightarrow t'}(Rep_{\Rightarrow}) \quad T''}{t_1 \Rightarrow t_2}(EC)\right) = \left\{\frac{APT'(T')\ APT'(T_1) \dots APT'(T_n)\ APT'(T'')}{t_1 \Rightarrow t_2}(Rep_{\Rightarrow})\right\}$$

$$(\mathbf{APT}_{10})\ \ APT'\left(\frac{T_1 \dots T_n}{t_1 \Rightarrow t_2}(EC)\right) = APT'(T_1) \bigcup \dots \bigcup APT'(T_n)$$

(R) any inference rule $\Rrightarrow$ either $\to$ or $\Rightarrow$
af either $t_1 \to t_2$, $t : s$ or $t_1 \Rightarrow t_2$

Fig. 2. Transforming rules for obtaining abbreviated proof trees

the transitivity nodes for rewrites shapes some parts of the debugging tree as a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to use very efficient navigation strategies [23, 24]. On the contrary, removing the transitivity inferences for rewrites (as rule ($\mathbf{APT}_4^o$) does) produces flattened trees where this strategy is no longer efficient. On the other hand, in rewrites $t \Rightarrow t'$ appearing as conclusion of the transitivity inference rule, the term t' can contain the result of rewriting several subterms of t, and determining the validity of such nodes can be complicated, while in the one-step debugging tree each rewrite node $t \Rightarrow t'$ corresponds to a single rewrite applied at t and checking its validity is usually easier. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

The rules ($\mathbf{APT}_3$) and ($\mathbf{APT}_9$) deserve a more detailed explanation. They keep the corresponding label ($Rep_{\Rrightarrow}$) but changing the conclusion of the replacement inference in the lefthand side. For instance, ($\mathbf{APT}_3$) replaces $t_1 \to t'$ by the conclusion of the next transitivity inference $t_1 \to t_2$. We do this as a pragmatic way of simplifying the structure of the APTs, since t_2 is obtained from t' and hence likely simpler (the root of the tree T' in ($\mathbf{APT}_3$) must be necessarily of the form $t' \to t_2$ by the structure of the inference rule for transitivity in Fig. 1). A similar reasoning explains the form of ($\mathbf{APT}_9$). We will formally state now that these changes are safe from the point of view of the debugger.

Theorem 1. *Let T be a finite proof tree representing an inference in the calculus of Fig. 1 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *Both $APT^o(T)$ and $APT^m(T)$ contain at least one buggy node (completeness).*
- *Any buggy node in $APT^o(T)$, $APT^m(T)$ has an associated wrong statement in $\mathcal{R}$ (correctness).*

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user answers correctly all the questions about the validity of the APT nodes asked by the debugger.

4 A Debugging Session

The debugger is initiated in Maude by loading the file `dd.maude` (available from `http://maude.sip.ucm.es/debugging`). This starts an input/output loop that allows the user to interact with the tool. Then, the user can enter Full Maude modules and commands, as well as commands for the debugger. The current version supports all kinds of modules. When debugging a rewrite computation, two different debugging trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The latter tree is partially built so that any node corresponding to a one-step rewrite is expanded only when the navigation process reaches it.

The debugger provides two strategies to traverse the debugging tree: *top-down*, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and *divide and query*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise. Note that, although the navigation strategy can be changed during the debugging session, the construction strategy is selected before the tree is built and cannot be changed.

The user can select a module containing only correct statements. By checking the correctness of the inferences with respect to this module (i.e., using this module as oracle) the debugger can reduce the number of questions. The debugger allows us to debug specifications where some statements are suspicious and have been labeled. Only these labeled statements generate nodes in the proof tree, being the user in charge of this labeling. The user can decide to use all the labeled statements as suspicious or can use only a subset by trusting labels and modules. Moreover, the user can answer that he trusts the statement associated with the currently questioned inference; that is, statements can be trusted "on the fly." The user can also give the answer "don't know," that postpones the answer to that question by asking alternative questions. An `undo` command, allowing the user to return to the previous state, is also provided. We refer the reader to [21, 22] for further information.

In Sect. 2.4 we described a system module that simulates a knight's tour. However, this system module contains a bug and the knight repeats some positions in its tour. This error is also obtained when looking for a 3 steps journey:

```
Maude> (rew knight(3) .)
result List : [1,1][2,3][3,1][2,3]
```

Thus, we debug this smaller computation. Moreover, after inspecting the rewrite rules describing the eight possible moves, we are sure that they are not responsible for the error; therefore, we trust them by using commands that allow us to select the suspicious statements.

```
Maude> (set debug select on .)
Maude> (debug select con1 con2 leg k1 k2 .)
Maude> (debug knight(3) =>* [1,1][2,3][3,1][2,3] .)
```

The default one-step tree construction strategy is used and the tree shown below is built, where every operation has been abbreviated with its first letter.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{k(0)} \Rightarrow_1 \texttt{[1,1]}}\,\mathsf{k1} \quad \dfrac{}{\texttt{l([2,3])} \rightarrow \texttt{t}}\,\mathsf{leg} \quad \dfrac{}{\texttt{c([1,1],[2,3])} \rightarrow \texttt{f}}\,\mathsf{con2}}{\texttt{k(1)} \Rightarrow_1 \texttt{J2}}\,\mathsf{k2} \quad \dfrac{}{\texttt{l([3,1])} \rightarrow \texttt{t}}\,\mathsf{leg} \quad \dfrac{}{\texttt{c(J2,[3,1])} \rightarrow \texttt{f}}\,\mathsf{con2}}{\texttt{k(2)} \Rightarrow_1 \texttt{J1}}\,\mathsf{k2} \quad \dfrac{}{\texttt{l([2,3])} \rightarrow \texttt{t}}\,\mathsf{leg} \quad \dfrac{}{\texttt{c(J1,[2,3])} \rightarrow \texttt{f}}\,\mathsf{con2}}{\texttt{k(3)} \Rightarrow_1 \texttt{[1,1][2,3][3,1][2,3]}}\,\mathsf{k2}$$

where `J1` denotes the journey `[1,1][2,3][3,1]` and `J2` denotes `[1,1][2,3]`.

Since the tree is navigated by using the default divide and query strategy, the first two questions asked by the debugger are

```
Is this rewrite (associated with the rule k2) correct?
knight(1) =>1 [1,1][2,3]
Maude> (yes .)
Is this rewrite (associated with the rule k2) correct?
knight(2) =>1 [1,1][2,3][3,1]
Maude> (yes .)
```

Notice the form `=>1` of the arrow in the rewrites appearing in the questions, to emphasize that they are one-step rewrites.

In both cases the answer is `yes` because these paths are possible, legal behaviors of the knight when it can do one or two hops. These two subtrees are removed and the current tree looks as follows:

$$\dfrac{\dfrac{}{\texttt{l([2,3])} \rightarrow \texttt{t}}\,\mathsf{leg} \quad \dfrac{}{\texttt{c(J1,[2,3])} \rightarrow \texttt{f}}\,\mathsf{con2}}{\texttt{k(3)} \Rightarrow_1 \texttt{[1,1][2,3][3,1][2,3]}}\,\mathsf{k2}$$

The next question is

```
Is this reduction (associated with the equation con2) correct?
contains([1,1][2,3][3,1],[2,3]) -> false
Maude> (no .)
```

Clearly, this is not a correct reduction, since position `[2,3]` is already in the path `[1,1][2,3][3,1]`. With this answer this subtree is selected and, since it is a single node, the bug is located:

```
The buggy node is:
contains([1,1][2,3][3,1],[2,3]) -> false
with the associated equation: con2
```

Looking at the definition of the `contains` operation, we realize that it defines the membership operation for *sets*, not for lists. A correct definition of the `contains` operation is as follows:

```
  eq [con1] : contains(nil, P) = false .
  eq [con2] : contains(Q J, P) = P == Q or contains(J, P) .
```

5 The Implementation

As mentioned in the introduction, a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [10, Chap. 14]. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [10, Chap. 17], which can be used to specify input/output interactions with the user. Thus, our declarative debugger, including its user interface, is implemented in Maude itself, as an extension of Full Maude [10, Chap. 18]. Instead of creating the complete proof tree and then abbreviating it, we build the abbreviated proof tree directly. Since navigation is done by asking questions to the user, this stage has to handle the navigation strategy together with the input/output interaction with the user. The technical report [21] provides a full explanation of this implementation, including the user interaction.

The way in which the debugging trees for reductions and memberships or rewrites are built is completely different. In the first case, we use the facts that equations and membership axioms are both terminating and confluent, which allow us to build the debugging tree in a "greedy" way, selecting at each moment the first equation applicable to the current term. However, we have to use a different methodology in the construction of the debugging tree for incorrect rewrites. We use breadth-first search to find from the initial term the wrong term introduced by the user, and then we use the found path to build the debugging tree in the two possible ways described in previous sections.

The functions in charge of building the debugging trees, that correspond to the APT function from Fig. 2, have a common initial behavior. They receive the module where the wrong inference took place, a correct module (or a special constant when no such module is provided) to prune the tree, the initial term, the (erroneous) result obtained, and the set of suspicious statements labels. They keep the initial inference as the root of the tree and generate the forest of abbreviated trees corresponding to the inference with functions that, in addition to the arguments above, receive the initial module "cleaned" of suspicious statements and correspond to the APT' function from Fig. 2. This transformed module is used to improve the efficiency of the tree construction, because we can use it to check if an inference can be obtained by using only trusted statements, thus avoiding to build a tree that will be finally empty.

The function that builds debugging trees for wrong reductions works with the same innermost strategy as the Maude interpreter: it first fully reduces the subterms recursively building their debugging trees (it mimics a specific behavior of the *congruence* rule in Fig. 1), and once all the subterms have been reduced, if the result is not the final one, it tries to reduce at the top to reach the final result by *transitivity*. Reduction at the top tries to apply one equation,[3] by

[3] Since the module is assumed to be confluent, we can choose any equation and the final result should be the same.

using the *replacement* rule from Fig. 1. Debugging trees for the conditions of the equation are also built and placed as children of the replacement rule. The construction of debugging trees for wrong memberships mimics the *subject reduction* rule from Fig. 1 by computing the tree for the full reduction of the term and then computing the tree for the membership inference of its least sort by using the operator declarations and the membership axioms, which corresponds to a concrete application of the *membership* inference rule.

The one-step tree for wrong rewrites computes the tree for the reduction from the initial term to normal form and then computes the rest of the tree, that corresponds to a rewrite from a fully reduced term (this corresponds to a concrete application of the *equivalence class* inference rule from Fig. 1). The debugging tree for this rewrite is computed from the trace, that is obtained with the predefined function `metaSearchPath`. Each step of the trace corresponds to the application of one rule, that generates a tree, with the trees corresponding to the conditions of the rule as its children (reproducing the *replacement* rule). Note that although the information in the trace is related to the whole rewritten term, the application of a rule can be in a subterm, which corresponds with the *congruence* inference rule, so only the rewritten subterms appear in the debugging tree. Other children are generated for the reduction to normal form due to the *equivalence class* inference rule. Finally, all the steps are put together as children of the same root by using the *transitivity* inference rule. The many-steps debugging tree is built *by demand*, so that the debugging subtrees corresponding to one-step rewrites are only generated when they are pointed out as wrong. These one-step nodes are used to create a balanced binary tree, by dividing them into two forests of approximately the same size, recursively creating their trees, and then using them as children of a new binary tree that has as root the combination by *transitivity* of the rewrites in their roots.

6 Concluding Remarks

In this paper we have developed the foundations of declarative debugging of executable rewriting logic specifications, and we have applied them to implement a debugger for Maude modules. The work encompasses and extends previous presentations [5, 6] on the declarative debugging of Maude functional modules, which constitute now a particular case of a more general setting.

We have formally described how debugging trees can be obtained from Maude proof trees, proving the correctness and completeness of the debugging technique. The tool based on these ideas allows the user to concentrate on the logic of the program disregarding the operational details. In order to deal with the possibly complex questions associated to rewrite statements, the tool offers the possibility of choosing between two different debugging trees: the one-step trees, with simpler questions and likely longer debugging sessions, and the many-steps trees, which in general require fewer but more complex questions before finding the bug. The experience will show the user which one must be chosen in each case depending on the complexity of the specification.

In our opinion, this debugger provides a complement to existing debugging techniques for Maude, such as tracing and term coloring. An important advantage of our debugger is the help provided by the tool in locating the buggy statements, assuming the user answers correctly the corresponding questions.

As future work we want to provide a graphical interface, that allows the user to navigate the tree with more freedom. We are also investigating how to improve the questions done in the presence of the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness. Finally, we plan to study how to debug *missing answers* [8, 16] in addition to the wrong answers we have treated thus far.

References

1. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
2. Bruni, R., Meseguer, J.: Semantic foundations for generalized rewrite theories. Theoretical Computer Science 360(1), 386–414 (2006)
3. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP 2005), pp. 8–13. ACM Press, Tallinn (2005)
4. Caballero, R., Hermanns, C., Kuchen, H.: Algorithmic debugging of Java programs. In: López-Fraguas, F.J. (ed.) 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid. Electronic Notes in Theoretical Computer Science, vol. 177, pp. 75–89. Elsevier, Amsterdam (2007)
5. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: Declarative debugging of membership equational logic specifications. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 174–193. Springer, Heidelberg (2008)
6. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: A declarative debugger for Maude functional modules. In: Roşu, G. (ed.) Proceedings Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008, Budapest. Electronic Notes in Computer Science, vol. 238, pp. 63–81. Elsevier, Amsterdam (2009)
7. Caballero, R., Rodríguez-Artalejo, M.: DDT: A declarative debugging tool for functional-logic languages. In: Kameyama, Y., Stuckey, P.J. (eds.) FLOPS 2004. LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
8. Caballero, R., Rodríguez-Artalejo, M., del Vado Vírseda, R.: Declarative diagnosis of missing answers in constraint functional-logic programming. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 305–321. Springer, Heidelberg (2008)
9. Chitil, O., Luo, Y.: Structure and properties of traces for functional programs. In: Mackie, I. (ed.) Proceedings of the Third International Workshop on Term Graph Rewriting (TERMGRAPH 2006). Electronic Notes in Theoretical Computer Science, vol. 176, pp. 39–63. Elsevier, Amsterdam (2007)
10. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)

11. Clavel, M., Meseguer, J., Palomino, M.: Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. Theoretical Computer Science 373(1-2), 70–91 (2007)
12. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing 5(2), 133–154 (1987)
13. MacLarty, I.: Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne (2005)
14. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
15. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
16. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
17. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming 1997(3) (1997)
18. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
19. Nilsson, H., Fritzson, P.: Algorithmic debugging of lazy functional languages. Journal of Functional Programming 4(3), 337–370 (1994)
20. Pope, B.: A Declarative Debugger for Haskell. PhD thesis, The University of Melbourne, Australia (2006)
21. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of Maude modules. Technical Report SIC-6/2008, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2008), `http://maude.sip.ucm.es/debugging`
22. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: A declarative debugger for Maude. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 116–121. Springer, Heidelberg (2008)
23. Shapiro, E.Y.: Algorithmic Program Debugging. In: ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
24. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
25. Tessier, A., Ferrand, G.: Declarative diagnosis in the CLP scheme. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)
26. Viry, P.: Equational rules for rewriting logic. Theoretical Computer Science 285(2), 487–517 (2002)

A Declarative Debugger for Maude*

Adrian Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero

Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Declarative debugging has been applied to many declarative programming paradigms; in this paper, a declarative debugger for rewriting logic specifications, embodied in the Maude language, is presented. Starting from an incorrect computation (a reduction, a type inference, or a rewrite), the debugger builds a tree representing this computation and guides the user through it to find a wrong statement. We present the debugger's main features, such as support for functional and system modules, two possible constructions of the debugging tree, two different strategies to traverse it, use of a correct module to reduce the number of questions asked to the user, selection of trusted vs. suspicious statements, and trusting of statements "on the fly".

1 Introduction

Declarative debugging, introduced by E. Y. Shapiro [8], is a semi-automatic technique that starts from a computation considered incorrect by the user (error symptom) and locates a program fragment responsible for the error. It has been widely employed in the logic [6], functional [7], and multiparadigm [3] programming languages. The declarative debugging scheme uses a *debugging tree* as a logical representation of the computation. Each node in the tree represents the result of a computation step, which must follow from the results of its child nodes by some logical inference. Diagnosis proceeds by traversing the debugging tree, asking questions to an external oracle (generally the user) until a so-called *buggy node* is found. Any buggy node represents a wrong computation step, and the debugger can display the program fragment responsible for it.

Maude [4] is a declarative language based on both equational and rewriting logic for the specification and implementation of a whole range of models and systems. Here we present a declarative debugger for *Maude functional and system modules*. Functional modules define data types and operations on them by means of *membership equational logic* theories that support multiple sorts, subsort relations, equations, and assertions of membership in a sort. Declarative debugging of functional modules has been presented in [2,1]. System modules specify rewrite theories that also support *rules*, defining local concurrent transitions that can take place in a system.

The debugging process starts with an incorrect computation from an initial term. Our debugger, after building a proof tree for that inference, will present to the user questions about the computation. Moreover, since the questions are located in the proof tree, the

* Research supported by MEC Spanish projects *DESAFIOS* (TIN2006-15660-C02-01) and *MERIT-FORMS* (TIN2005-09027-C03-03), and Comunidad de Madrid program *PROMESAS* (S-0505/TIC/0407).

answer allows the debugger to discard a subset of the questions, leading and shortening the debugging process. The current version of the tool supports all kinds of modules (except for the attribute `strat`), different ways of trusting statements, two possible constructions of the debugging tree for rewritings, and two strategies for traversing it. The debugger is implemented on top of Full Maude [4, Chap. 18]—allowing to debug the different modules provided by it, such as object-oriented and parameterized ones—and exploiting the reflective capabilities of Maude. Complete explanations about the fundamentals and novelties of our debugging approach can be found in the technical report [10], which, together with the source files for the debugger, examples, and related papers, is available from the webpage `http://maude.sip.ucm.es/debugging`.

2 Using the Debugger

We make explicit first what is assumed about the modules introduced by the user; then we present the available commands.

Assumptions. A rewrite theory has an underlying equational theory, containing equations and memberships, which is expected to satisfy the appropriate executability requirements, namely, it has to be terminating, confluent, and sort decreasing. Rules are assumed to be coherent with respect to the equations; for details, see [4].

In our debugger, unlabeled statements are assumed to be correct. Moreover, the user can trust more statements or introduce a correct module to check the inferences. In order to obtain a nonempty abbreviated proof tree, at least the buggy statement must be suspicious; the user is responsible for the correctness of these decisions.

Commands. The debugger is initiated in Maude by loading the file `dd.maude`, which starts an input/output loop that allows the user to interact with the tool. Since the debugger is implemented on top of Full Maude, all modules must be introduced enclosed in parentheses. If a module with correct definitions is used to reduce the number of questions, it must be indicated before starting the debugging with the command `(correct module MODULE-NAME .)`. Since rewriting with rules is not assumed to terminate, a bound, which is 42 by default although can be `unbounded`, is used when searching in the correct module and can be set with the command `(set bound BOUND .)`. The user can debug with only a subset of the labeled statements by using the command `(set debug select on .)`. Once this mode is activated, the user can select and deselect statements by using `(debug [de]select LABELS .)`. Moreover, all the labeled statements of a flattened module can be selected or deselected with the commands `(debug include/exclude MODULES .)`. When debugging rewrites, two different trees can be built: one whose questions are related to one-step rewrites and another one whose questions are related to several steps. The user can switch between these trees with the commands `(one-step tree .)`, which is the default one, and `(many-steps tree .)`, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The proof tree can be navigated by using two different strategies: the more intuitive *top-down* strategy, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and the more efficient *divide and query* strategy, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, the latter being the default one. The user can switch between them with the commands `(top-down strategy .)` and `(divide-query strategy .)`. Debugging is started with the following commands for wrong reductions, memberships, and rewrites.[1]

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM .)
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
(debug [in MODULE-NAME :] INITIAL-TERM =>* WRONG-TERM .)
```

How the process continues depends on the selected strategy. In case the top-down strategy is selected, several nodes will be displayed in each question. If there is an invalid node, we must select one of them with the command `(node N .)`. If all the nodes are correct, we answer `(all valid .)`. In the divide and query strategy, each question refers to one inference that can be either correct or wrong. The different answers are transmitted with the commands `(yes .)` and `(no .)`. Instead of just answering `yes`, we can also *trust* some statements on the fly if, once the process has started, we decide the bug is not there. To trust the current statement we type the command `(trust .)`. Finally, we can return to the previous state by using the command `(undo .)`.

We show in the next sections how to use these commands to debug several examples.

3 Functional Module Example: Multisets

We use sets and multisets to illustrate how to debug functional modules. We describe sets by means of a membership that asserts that a set is a multiset without repetitions. However, the equation `mt2` is wrong, because it should add 1 to `mult(N, S)`:

```
   cmb [set] : N S : Set if S : Set /\ mult(N, S) = 0 .
   eq [mt2] : mult(N, N S) = mult(N, S) .
```

If we check now the type of `1 1 2 3` we obtain it is `Set`! We debug this wrong behavior with the command

```
Maude> (debug 1 1 2 3 : Set .)
```

that builds the associated debugging tree, and selects a node using divide and query:

```
Is this membership (associated with the membership set) correct?
1 2 3  : Set
Maude> (yes .)
```

The debugger continues asking the questions below, now associated to equations:

```
Is this reduction (associated with the equation mt3) correct?
mult(1, 2 3) -> 0
Maude> (yes .)
Is this reduction (associated with the equation mt2) correct?
mult(1, 1 2 3) -> 0
Maude> (no .)
```

[1] If no module name is given, the current module is used by default.

With this information, the debugger finds the wrong statement:

```
The buggy node is: mult(1, 1 2 3) -> 0
With the associated equation: mt2
```

4 System Module Example: Operational Semantics

We illustrate in this section how to debug system modules by means of the semantics of the WhileL language, a simple imperative language described in [5] and represented in Maude in [9]. The syntax of the language includes skip, assignment, composition, conditional statement, and while loop. The state of the execution is kept in the *store*, a set of pairs of variables and values.

Evaluation semantics. The evaluation semantics takes a pair consisting of a command and a store and returns a store.[2] However, we have committed an error in the while loop:

```
 crl [WhileR2] : < While be Do C, st > => < skip, st' >
             if < be, st > => T /\ < C, st > => < skip, st' > .
```

That is, if the condition is true, the body is evaluated only once. Thus, if we execute the program below to multiply `x` and `y` and keep the result in `z`

```
Maude> (rew < z := 0 ; (While Not Equal(x, 0) Do
                        z := z +. y ; x := x -. 1), x = 2 y = 3 z = 1 > .)
result Statement : < skip, y = 3 z = 3 x = 1 >
```

we obtain `z = 3`, while we expected to obtain `z = 6`. We debug this behavior with the top-down strategy and the default one-step tree by typing the commands

```
Maude> (top-down strategy .)
Maude> (debug < z := 0 ; (While Not Equal(x, 0) Do
                           z := z +. y ; x := x -. 1), x = 2 y = 3 z = 1 >
        =>* < skip, y = 3 z = 3 x = 1 > .)
```

The debugger computes the tree and asks about the validity of the root's children:

```
Please, choose a wrong node:
Node 0 : < z := 0, x = 2 y = 3 z = 1 > =>1 < skip, x = 2 y = 3 z = 0 >
Node 1 : < While Not Equal(x,0) Do z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
         =>1 < skip, y = 3 z = 3 x = 1 >
Maude> (node 1 .)
```

The second node is erroneous, because `x` has not reached `0`, so the user selects this node to continue the debugging, and the following question is related to its children:

```
Please, choose a wrong node:
Node 0 : < Not Equal(x,0), x = 2 y = 3 z = 0 > =>1 T
Node 1 : < z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
         =>1 < skip, y = 3 z = 3 x = 1 >
Maude> (all valid .)
```

[2] In order to reuse this module later, the returned result is a pair `< skip, st >`.

Since both nodes are right, the debugger determines that the current node is buggy:

```
The buggy node is:
< While Not Equal(x,0) Do z := z +. y ; x := x -. 1, x = 2 y = 3 z = 0 >
=>1 < skip, y = 3 z = 3 x = 1 >
With the associated rule: WhileR2
```

Computation semantics. In contrast to the evaluation semantics, the computation semantics describes the behavior of programs in terms of small steps. In order to illustrate this, we make a mistake in the rule describing the semantics of the composition, keeping the initial state instead of the new one computed in the condition:

```
 crl [ComRc1] : < C ; C', st > => < C'' ; C', st >
             if < C, st > => < C'', st' > /\ C =/= C'' .
```

If we rewrite now a program to swap the values of two variables, their values are not exchanged. We use the many-steps tree to debug this wrong behavior:

```
Maude> (many-steps tree .)
Maude> (debug < x := x -. y ; y := x +. y ; x := y -. x, x = 5 y = 2 >
           =>* < skip, y = 2 x = 0 > .)
Is this rewrite correct?
< y := x +. y ; x := y -. x, x = 5 y = 2 > =>+ < skip, y = 2 x = 0 >
Maude> (no .)
```

The transition is wrong because the variables have not been properly updated.

```
Is this rewrite (associated with the rule ComRc1) correct?
< y := x +. y ; x := y -. x,x = 5 y = 2 > =>1 < x := y -. x,x = 5 y = 2 >
Maude> (no .)
Is this rewrite (associated with the rule OpR) correct?
< x +. y, x = 5 y = 2 > =>1 7
Maude> (trust .)
```

We consider that the application of a primitive operation is simple enough to be trusted. The next question is related to the application of an equation to update the store

```
Is this reduction (associated with the equation st1) correct?
x = 5 y = 2[7 / y] -> x = 5 y = 7
Maude> (yes .)
```

Finally, a question about assignment is posed:

```
Is this rewrite (associated with the rule AsRc) correct?
< y := x +. y, x = 5 y = 2 > =>1 < skip, x = 5 y = 7 >
Maude> (yes .)
```

With this information, the debugger is able to find the bug. However, since we have the evaluation semantics of the language already specified and debugged, we can use that module as correct module to reduce the number of questions to only one.

```
Maude> (correct module EVALUATION-WHILE .)
Maude> (debug ... .)
Is this rewrite correct?
< y := x +. y ; x := y -. x, x = 5 y = 2 > =>+ < skip, y = 2 x = 0 >
Maude> (no .)
```

5 Conclusions

We have implemented a declarative debugger for Maude modules that allows to debug wrong reductions, type inferences, and rewrites. Although the complexity of the debugging process increases with the size of the proof tree, it does not depend on the total number of statements but on the number of applications of suspicious statements involved in the wrong inference. Moreover, bugs found when reducing complex initial terms can, in general, be reproduced with simpler terms which give rise to smaller proof trees. We plan to improve the interaction with the user by providing a complementary graphical interface that allows the user to navigate the tree with more freedom. This interaction could also be improved by allowing the user to give the answer "don't know," that would postpone the answer to the question by asking alternative questions. We are also studying how to handle the `strat` operator attribute, that allows the specifier to define an evaluation strategy. This can be used to represent some kind of laziness.

References

1. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: A declarative debugger for Maude functional modules. In: Proceedings Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008. Elsevier, Amsterdam (to appear, 2008)
2. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: Declarative debugging of membership equational logic specifications. In: Degano, P., Nicola, R.D., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 174–193. Springer, Heidelberg (2008)
3. Caballero, R., Rodríguez-Artalejo, M.: DDT: A declarative debugging tool for functional-logic languages. In: Proc. 7th International Symposium on Functional and Logic Programming (FLOPS 2004). LNCS, vol. 2998, pp. 70–84. Springer, Heidelberg (2004)
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude: A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
5. Hennessy, M.: The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics. John Wiley & Sons, Chichester (1990)
6. Lloyd, J.W.: Declarative error diagnosis. New Generation Computing 5(2), 133–154 (1987)
7. Nilsson, H., Fritzson, P.: Algorithmic debugging of lazy functional languages. Journal of Functional Programming 4(3), 337–370 (1994)
8. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
9. Verdejo, A., Martí-Oliet, N.: Executable structural operational semantics in Maude. Journal of Logic and Algebraic Programming 67, 226–293 (2006)
10. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of Maude modules. Technical Report SIC-6/08, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2008), `http://maude.sip.ucm.es/debugging`

Enhancing the Debugging of Maude Specifications[⋆]

Adrian Riesco, Alberto Verdejo, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
`ariesco@fdi.ucm.es`, `{alberto,narciso}@sip.ucm.es`

Abstract. Declarative debugging is a semi-automatic technique that locates a program fragment responsible for the error by building a tree representing the computation and guiding the user through it to find the error. Two different kinds of errors are considered for debugging: *wrong answers*—a wrong result obtained from an initial value—and *missing answers*—a term that should be reachable but cannot be obtained from an initial value—, where the latter has only been considered in nondeterministic systems. However, we consider that missing answers can also appear in deterministic systems, when we obtain correct results that do not provide all the expected information, which corresponds, in the context of Maude modules, to terms whose normal form is not reached and to terms whose computed least sort is, although correct, bigger than the expected one. We present in this paper a calculus to deduce normal forms and least sorts, and a proper abbreviation of the trees obtained with it. These trees increase both the causes (missing equations and memberships) and the errors (erroneous normal forms and least sorts) detected in our debugging framework.

Keywords: declarative debugging, Maude, rewriting logic, membership equational logic, wrong answers, missing answers.

1 Introduction

Declarative debugging (also known as declarative diagnosis or algorithmic debugging) [17] is a debugging technique that abstracts the computation details to focus on results. It starts from an incorrect computation, the error symptom, and locates a program fragment responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [10], where each node stands for a computation step and must follow from the results of its child nodes by some logical inference. This tree is traversed by asking questions to an external oracle (generally the user) until a *buggy node*—a node containing an erroneous result, but whose children are all correct—is found. Traditional debugging techniques are devoted to fixing errors in specifications when an erroneous result, called a wrong answer, is found. Declarative debugging of this

⋆ Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

kind of errors has been widely studied in the logic [9,19], functional [11,12], and multi-paradigm [3,7] programming languages. Another kind of errors, called *missing answers* [4,1], appears in nondeterministic systems when a term that should be reachable cannot be obtained from an initial one. This kind of errors has been less studied because it can only be applied to nondeterministic systems and because the associated calculus may be much more complicated than the one associated to wrong answers, making the debugging process unbearable.

Maude [5] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [8], a logic that allows the representation of many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [2], an equational logic that, in addition to equations, allows to state *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, that represent transitions in a concurrent system. The Maude system supports several approaches for debugging: tracing, term coloring, and using an internal debugger [5, Chap. 22]. As part of an ongoing project to develop a declarative debugger for Maude specifications, we have already studied wrong answers in both functional and system modules [14] and missing answers in rewrites [15]. We now extend our framework by developing a calculus to deduce normal forms and least sorts seeing that the errors associated to these deductions correspond to missing answers in a deterministic framework. With this calculus we can detect errors due not only to wrong statements in a given specification but also to statements that the user *forgot* to specify,[1] indicating in this last case the operator at the top that the statement needs. These features improve our debugger in two ways: allowing to debug missing answers in the equational part of Maude modules and increasing the range of errors detected by the tool. For example, we can now debug missing answers when a rule cannot be applied because the term does not reach its normal form due to a missing equation or because the lefthand side does not match the term because it has a wrong least sort. We illustrate this improvement in Section 3 with a system module that, if debugged with the previous version of our tool, would print `Error: With the given information (labeling, correct module, and answers) it is impossible to debug.`, while in the current version the error is located.

The rest of the paper is organized as follows: after briefly introducing Maude modules with an example, Section 2 presents the calculus for missing answers and how the proof trees built with it are pruned in order to obtain appropriate debugging trees. Section 3 presents our tool by debugging some examples, while Section 4 concludes and outlines some future work.

The Maude source of the debugger, a user guide [13], additional examples, and other papers on this subject, including detailed proofs of the results [16], are all available from the webpage `http://maude.sip.ucm.es/debugging`.

[1] Note that the treatment of these missing statements is more powerful than the one currently applied in the Maude sufficient completeness checker [6], because it can be used with conditional and non left-linear statements.

1.1 An Example: Heaps

We show in this section how to specify in Maude binary heaps, that is, binary trees fulfilling that (1) all levels of the tree, except possibly the last one, are complete and, if the last level of the tree is not complete, the nodes of that level are filled from left to right; and (2) the value in each node is greater than the value in each of its children. The module `HEAP` defines binary trees (`BTree`) and `Heap`s and its nonempty variants (`NeBTree` and `NeHeap`), using a theory `TH` (not shown here) that defines the functions `min`, `max`, and a total order `_<_` over the elements of the sort `Elt`:

```
(fmod HEAP{X :: TH} is
  pr NAT .

  sorts BTree Heap NeBTree NeHeap .
  subsort NeHeap < NeBTree Heap < BTree .

  op mt : -> Heap [ctor] .
  op ___ : BTree X$Elt BTree -> NeBTree [ctor] .
```

We state by means of memberships when a binary tree is a heap:

```
  vars E E' : X$Elt .                    vars BT BT' : BTree .
  vars L L' R R' : Heap .                vars NL NR : NeHeap .

  cmb [h1] : NL E mt : NeHeap
   if max(NL) < E /\ depth(NL) == 1 .
  cmb [h2] : NL E NR : NeHeap
   if max(NL) < E /\ max(NR) < E /\
      (depth(NL) == depth(NR) and complete(NL)) or
      (depth(NL) == s(depth(NR)) and complete(NR)) .
```

where the auxiliary function `depth` computes the depth of a binary tree; `max` returns the value at the root of a nonempty heap (i.e., its maximum); and `complete` checks whether a binary tree is complete:

```
  op depth : BTree -> Nat .
  eq [dp1] : depth(mt) = 0 .
  eq [dp2] : depth(BT N BT') = max(depth(BT), depth(BT')) + 1 .

  op max : NeHeap -> X$Elt .
  ceq [max] : max(L E R) = E if L E R : NeHeap .

  op complete : BTree -> Bool .
  eq [cmp1] : complete(mt) = true .
  eq [cmp2] : complete(BT E BT') = complete(BT) and complete(BT') and
                                   depth(BT) == depth(BT') .
```

The function `insert` introduces a new element in a heap by sinking it to the appropriate position:

```
  op insert : X$Elt Heap ~> NeHeap .
  eq [ins1] : insert(E, mt) = mt E mt .
  ceq [ins2] : insert(E, L E' R) = L' max(E, E') R
   if L E' R : NeHeap /\
      not complete(L) or ((depth(L) > depth(R)) and complete(R)) /\
      L' := insert(min(E, E'), L) .
  ceq [ins3] : insert(E, L E' R) = L max(E, E') R'
   if L E' R : NeHeap /\
      not complete(R) or (depth(L) > depth(R)) and complete(L) /\
      R' := insert(min(E, E'), R) .
endfm)
```

We use a view HN (not shown here) to instantiate the values of the heap as natural numbers and we define a constant heap for testing:

```
(fmod NAT-HEAP is
  pr HEAP{HN} .
  op heap : -> NeHeap .
  eq heap = (mt 4 mt) 5 (mt 3 mt) .
endfm)
```

If we check in our specification the type of the constant heap:

```
Maude> (red heap .)
result NeBTree : (mt 4 mt) 5 (mt 3 mt)
```

we realize that although it has a correct sort (it is a NeBTree) its expected least sort, NeHeap, has not been obtained. We will show in Section 3 how to debug it.

2 Debugging Trees for Normal Forms and Least Sorts

We present in this section a calculus to compute the normal form and the least sort of a given term. The proof trees computed with this calculus contain the information proving why the term has been reduced to this normal form or this sort has been inferred (positive information) and also why the term has not been further reduced or a lesser sort has not been computed (negative information). The calculus is introduced as an extension of the calculus in [14] that allowed to deduce judgments corresponding to oriented equations $t \to t'$ and memberships $t : s$, and improves the calculus of missing answers of [15] by adding new causes to the errors debugged thus far. Once this extended calculus is presented, we show how to use it to define appropriate debugging trees.

2.1 A Calculus for Normal Forms and Least Sorts

From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the Maude executability requirements, i.e., E is confluent and terminating, maybe modulo some equational attributes such as associativity and commutativity, while R is coherent with respect to E. Equations corresponding to the equational attributes form the set A and the equations in $E - A$ can be oriented from left to right.

Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*. They correspond to the ones used in Maude modules and are defined as follows:

Definition 1. *A condition $C_1 \wedge \cdots \wedge C_n$ is* admissible *if, for $1 \leq i \leq n$, C_i is*

- *an equation $u_i = u'_i$ or a membership $u_i : s$ and $vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or*
- *a matching condition $u_i := u'_i$, u_i is a pattern and $vars(u'_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or*
- *a rewrite condition $u_i \Rightarrow u'_i$, u'_i is a pattern and $vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$.*

Note that the lefthand side of matching conditions and the righthand side of rewrite conditions can contain extra variables that will be instantiated once the condition is solved.

Definition 2. *A* kind-substitution*, denoted by κ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} \,.\, kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the term it binds.*

Definition 3. *A* substitution*, denoted by θ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} \,.\, sort(v_i) \geq ls(t_i)$, that is, the sort of each variable is greater than or equal to the least sort of the term it binds. Note that a substitution is a special type of kind-substitution where each term has the sort appropriate to its variable.*

Definition 4. *Given an atomic condition C, we say that a substitution θ is* admissible for C *if*

- *C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or*
- *C is a matching condition $u := u'$ and $vars(u') \subseteq dom(\theta)$, or*
- *C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.*

The calculus presented in this section (Figures 1 and 2) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model [8,16] $\mathcal{T}' = \mathcal{T}_{\Sigma/E',R'}$ defined by equations and memberships E' and by rules R':

- Given a term t and a kind-substitution κ, $\mathcal{T}' \models adequateSorts(\kappa) \rightsquigarrow \Theta$ when either $\Theta = \{\kappa\} \wedge \forall v \in dom(\kappa). \mathcal{T}' \models \kappa[v] : sort(v)$ or $\Theta = \emptyset \wedge \exists v \in dom(\kappa). \mathcal{T}' \not\models \kappa[v] : sort(v)$, where $\kappa[v]$ denotes the term bound by v in κ. That is, when all the terms bound in the kind-substitution κ have the appropriate sort, then κ is a substitution and it is returned; otherwise (at least one of the terms has an incorrect sort), the kind-substitution is not a substitution and the empty set is returned.
- Given an admissible substitution θ for an atomic condition C, $\mathcal{T}' \models [C, \theta] \rightsquigarrow \Theta$ when $\Theta = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \restriction_{dom(\theta)} = \theta\}$, that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ.

$$\frac{\theta(t_2) \rightarrow_{norm} t' \quad adequateSorts(\kappa_1) \rightsquigarrow \Theta_1 \quad \ldots \quad adequateSorts(\kappa_n) \rightsquigarrow \Theta_n}{[t_1 := t_2, \theta] \rightsquigarrow \bigcup_{i=1}^{n} \Theta_i} \ \mathsf{PatC}$$
$$\text{if } \{\kappa_1, \ldots, \kappa_n\} = \{\kappa\theta \mid \kappa(\theta(t_1)) \equiv_A t'\}$$

$$\frac{t_1 : sort(v_1) \quad \ldots \quad t_n : sort(v_n)}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \{v_1 \mapsto t_1; \ldots; v_n \mapsto t_n\}} \ \mathsf{AS_1}$$

$$\frac{t_i :_{ls} s_i}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \emptyset} \ \mathsf{AS_2} \ \text{ if } s_i \not\leq sort(v_i)$$

$$\frac{\theta(t) : s}{[t : s, \theta] \rightsquigarrow \{\theta\}} \ \mathsf{MbC_1} \qquad \frac{\theta(t) :_{ls} s'}{[t : s, \theta] \rightsquigarrow \emptyset} \ \mathsf{MbC_2} \ \text{ if } s' \not\leq s$$

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \rightsquigarrow \{\theta\}} \ \mathsf{EqC_1} \qquad \frac{\theta(t_1) \rightarrow_{norm} t'_1 \quad \theta(t_2) \rightarrow_{norm} t'_2}{[t_1 = t_2, \theta] \rightsquigarrow \emptyset} \ \mathsf{EqC_2} \ \text{ if } t'_1 \not\equiv_A t'_2$$

$$\frac{\theta(t_1) \rightsquigarrow_{n+1}^{t_2 := \circledast} S}{[t_1 \Rightarrow t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}} \ \mathsf{RIC} \qquad \frac{[C, \theta_1] \rightsquigarrow \Theta_1 \quad \cdots \quad [C, \theta_m] \rightsquigarrow \Theta_m}{\langle C, \{\theta_1, \ldots, \theta_m\}\rangle \rightsquigarrow \bigcup_{i=1}^{m} \Theta_i} \ \mathsf{SubsCond}$$
$$\text{if } n = min(x \in \mathbb{N} : \forall i \geq 0 \ (\theta(t_1) \rightsquigarrow_{x+i}^{t_2 := \circledast} S))$$

Fig. 1. Calculus for substitutions

– Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$ when $\Theta' = \{\theta' \mid \mathcal{T}', \theta' \models C$ and $\theta' \restriction_{dom(\theta)} = \theta$ for some $\theta \in \Theta\}$, that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.
– Given an equation or membership a and a term t, $\mathcal{T}' \models disabled(a, t)$ when a cannot be applied to t at the top.
– Given two terms t and t', $\mathcal{T}' \models t \rightarrow_{red} t'$ when $\mathcal{T}' \models t \rightarrow^{1}_{E'} t'$ or $\mathcal{T}' \models t_i \rightarrow^{!}_{E'} t'_i$, with $t_i \neq t'_i$, for some subterm t_i of t such that $t' = t[t_i \mapsto t'_i]$, that is, the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.
– Given two terms t and t', $\mathcal{T}' \models t \rightarrow_{norm} t'$ when $\mathcal{T}' \models t \rightarrow^{!}_{E'} t'$, that is, t' is in normal form with respect to the equations E'.
– Given a term t and a sort s, $\mathcal{T}' \models t :_{ls} s$ when $\mathcal{T}' \models t : s$ and moreover s is the least sort with this property (with respect to the ordering on sorts obtained from the signature Σ and the equations and memberships E' defining the Σ-term model $\mathcal{T}'$).

We introduce in Figure 1 the inference rules defining the relations $[C, \theta] \rightsquigarrow \Theta$, $\langle C, \Theta\rangle \rightsquigarrow \Theta'$, and $adequateSorts(\kappa) \rightsquigarrow \Theta$. Intuitively, these judgments will provide positive information when they lead to nonempty sets (indicating that the condition holds in the first two judgments or that the kind-substitution is a substitution in the third one) and negative information when they lead to the empty set (indicating respectively that the condition fails or the kind-substitution is not a substitution):

- Rule PatC computes all the possible substitutions that extend θ and satisfy the matching of the term t_2 with the pattern t_1 by first computing the normal form t' of t_2, obtaining then all the possible kind-substitutions κ that make t' and $\theta(t_1)$ equal modulo axioms (indicated by $\equiv_A$), and finally checking that the terms assigned to each variable in the kind-substitutions have the appropriate sort with $adequateSorts(\kappa)$. The union of the set of substitutions thus obtained constitutes the set of substitutions that satisfy the matching.
- Rule AS_1 checks whether the terms of the kind-substitution have the appropriate sort to match the variables. In this case the kind-substitution is a substitution and it is returned.
- Rule AS_2 indicates that, if the least sort of any of the terms in the kind-substitution is bigger than the required one, then it is not a substitution and thus the empty set of substitutions is returned.
- Rule MbC_1 returns the current substitution if a membership condition holds.
- Rule MbC_2 is used when the membership condition is not satisfied. It checks that the least sort of the term is not less than or equal to the required one, and thus the substitution does not satisfy the condition and the empty set is returned.
- Rule EqC_1 returns the current substitution when an equality condition holds, that is, when the two terms can be joined with equations, abbreviated as $t_1 \downarrow t_2$.
- Rule EqC_2 checks that an equality condition fails by obtaining the normal forms of both terms and then examining that they are different.
- Rewrite conditions are handled by rule RlC. This rule extends the set of substitutions by computing all the reachable terms that satisfy the pattern (using the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ explained in [16]) and then using these terms to obtain the new substitutions.
- Finally, rule SubsCond computes the extensions of a set of admissible substitutions $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

We use these judgments to define the inference rules of Figure 2, that describe how the normal form and the least sort of a term are computed:

- Rule Dsb indicates when an equation or membership a cannot be applied to a term t. It checks that there are no substitutions that satisfy the matching of the term with the lefthand side of the statement and that fulfill its condition. Note that we check the conditions from left to right, following the same order as Maude and making all the substitutions admissible.
- Rule Rdc_1 reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions. While in the previous rule we made explicit the evaluation from left to right of the condition to show that finally the set of substitutions fulfilling it was empty, in this case we only need one substitution to fulfill the condition and the order is unimportant.
- Rule Rdc_2 reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \quad \ldots \quad \langle C_n, \Theta_{n-1}\rangle \rightsquigarrow \emptyset}{disabled(a, t)}\ \mathsf{Dsb}$$
$$\text{if } a \equiv l \to r \Leftarrow C_1 \wedge \ldots \wedge C_n \in E \text{ or}$$
$$a \equiv l : s \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$$

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^n \quad \{\theta(v_j) : s_j\}_{j=1}^m}{\theta(l) \to_{red} \theta(r)}\ \mathsf{Rdc_1}\ \text{ if } l \to r \Leftarrow \bigwedge_{i=1}^n u_i = u_i' \wedge \bigwedge_{j=1}^m v_j : s_j \in E$$

$$\frac{t \to_{norm} t'}{f(t_1, \ldots, t, \ldots, t_n) \to_{red} f(t_1, \ldots, t', \ldots, t_n)}\ \mathsf{Rdc_2}\ \text{ if } t \not\equiv_A t'$$

$$\frac{disabled(e_1, f(t_1, \ldots, t_n)) \quad \ldots \quad disabled(e_l, f(t_1, \ldots, t_n)) \quad t_1 \to_{norm} t_1 \quad \ldots \quad t_n \to_{norm} t_n}{f(t_1, \ldots, t_n) \to_{norm} f(t_1, \ldots, t_n)}\ \mathsf{Norm}$$
$$\text{if } \{e_1, \ldots, e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1, \ldots, t_n)\}$$

$$\frac{t \to_{red} t_1 \quad t_1 \to_{norm} t'}{t \to_{norm} t'}\ \mathsf{NTr}$$

$$\frac{t \to_{norm} t' \quad t' : s \quad disabled(m_1, t') \quad \ldots \quad disabled(m_l, t')}{t :_{ls} s}\ \mathsf{Ls}$$
$$\text{if } \{m_1, \ldots, m_l\} = \{m \in E \mid m \ll_K^{top} t' \ \wedge\ sort(m) < s\}$$

Fig. 2. Calculus for normal forms and least sorts

- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top considering the variables at the kind level (which is indicated by $\ll_K^{top}$) and that all its subterms are already in normal form.
- Rule NTr describes the transitivity for the reduction to normal form. It reduces the term with the relation $\to_{red}$ and the term thus obtained then is reduced to normal form by using again $\to_{norm}$.
- Rule Ls computes the least sort of the term t. It computes a sort for its normal form (that has the least sort of the terms in the equivalence class) and then checks that memberships deducing lesser sorts, applicable at the top with the variables considered at the kind level, cannot be applied.

In these rules Dsb provides the negative information, proving why the statements (either equations or membership axioms) cannot be applied, while the remaining rules provide the positive information indicating why the normal form and the least sort are obtained.

Theorem 1. *The calculus of Figures 1 and 2 is correct w.r.t.* $\mathcal{R} = (\Sigma, E, R)$ *in the sense that for any judgment* φ*,* φ *is derivable in the calculus if and only if* $\mathcal{T}_{\Sigma/E,R} \models \varphi$*, with* $\mathcal{T}_{\Sigma/E,R}$ *being the corresponding initial model.*

Once these rules have been presented, we can compute the proof tree associated to the erroneous computation shown in Section 1.1 for the heaps example. Remember that the least sort of the term `heap`, that should be `NeHeap`, was instead `NeBTree`. Figures 3 and 4 show the associated proof tree, where h stands for the term `(mt 4 mt) 5 (mt 3 mt)`, l for the lefthand side of the membership

$$\dfrac{\dfrac{\dfrac{}{\texttt{heap} \to_{red} h}\,\mathsf{Rdc_1} \quad \dfrac{}{h \to_{norm} h}\,\mathsf{Norm}}{\texttt{heap} \to_{norm} h}\,\mathsf{NTr} \quad \dfrac{}{h : \texttt{NeBTree}}\,\mathsf{Mb} \quad T_1}{\texttt{heap} :_{ls} \texttt{NeBTree}}\,\mathsf{Ls}$$

Fig. 3. Proof tree for the heap example

$$\dfrac{\dfrac{\dfrac{}{h \to_{norm} h}\,\mathsf{Norm} \quad \dfrac{\dfrac{\nabla}{\texttt{mt 4 mt} :_{ls} \texttt{NeBTree}}\,\mathsf{Ls}}{adequateSorts(l,\theta)}\,\mathsf{AS_2}}{[l := h] \rightsquigarrow \emptyset}\,\mathsf{PatC} \quad \dfrac{}{\langle C_1,\emptyset\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond} \quad \dots \quad \dfrac{}{\langle C_n,\emptyset\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{disabled(\texttt{h2}, h)}\,\mathsf{Dsb}$$

Fig. 4. Proof tree T_1, proving the matching with `h2`

`h2`, namely `NL E NR` with `NL` and `NR` variables of sort `NeHeap` and `E` a natural number, C_1 and C_n are respectively the first condition and last condition of `h2`, θ is $\texttt{NL} \mapsto \texttt{mt 4 mt}; \texttt{E} \mapsto 5; \texttt{NR} \mapsto \texttt{mt 3 mt}$, and ∇ represents a tree similar to the one depicted in Figure 3.

The tree shown in Figure 3 illustrates that to compute the least sort of `heap` first it obtains its normal form and then it checks that no memberships can be applied to this term (and thus the sort is inferred by using the operator declarations). To check that no memberships are applied it only checks whether `h2` is used, because the other membership does not match the term with the variables at the kind level. The tree T_1, depicted in Figure 4, is in charge of this proof, that is, it provides the negative information proving that the membership cannot be applied. First, it checks that the lefthand side of the membership does not match the term because `mt 4 mt` has as least sort `NeBTree` and hence it does not match the variable `NL`, that has sort `NeHeap`. Since the empty set of substitutions is computed for this matching, the rest of conditions of the membership cannot be fulfilled, which is proved by the nodes associated with the rule SubsCond.

Following the approach shown in [14], we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. We say that a judgment is *valid* when it holds in $\mathcal{I}$, and *invalid* otherwise. The basis of declarative debugging consists in searching *buggy nodes* (invalid nodes with all its children valid) [10] in a debugging tree standing for a problematic computation. In our debugging framework, we are able to locate wrong equations, wrong memberships, missing equations, and missing memberships,[2] which are defined as follows:

[2] It is important not to confuse wrong and missing answers with wrong and missing statements. The former are the initial symptoms that indicate the specifications fails, while the latter are the errors that generated this misbehavior.

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$ or a membership $l : s$) and a substitution θ, the *statement instance* $\theta(A) \Leftarrow \theta(C_1) \wedge \cdots \wedge \theta(C_n)$ is *wrong* when all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$ but $\theta(A)$ is not.
- Given a term t, there is a *missing equation for* t if the computed normal form of t does not correspond with the one expected in $\mathcal{I}$.
- A specification has a *missing equation* if there exists a term t such that there is a missing equation for t.
- Given a term t, there is a *missing membership for* t if the computed least sort for t does not correspond with the one expected in $\mathcal{I}$.
- A specification has a *missing membership* if there exists a term t such that there is a missing membership for t.

Regarding missing statements, what the debugger reports is that a statement is missing *or* the conditions in the remaining statements are not the intended ones (thus they are not applied when expected and another one would be needed), but the error *is not located* in the statements used in the conditions, since they are also checked during the debugging process.

Proposition 1. *Let N be a buggy node in some proof tree in the calculus of Figures 1 and 2 w.r.t. an intended interpretation $\mathcal{I}$. Then the error associated to N is a wrong equation, a missing equation, or a missing membership.*

Although these are the errors detected by the calculus presented in this paper, since it is integrated with both the calculus of wrong answers [14] and the calculus for missing answers [15], the debugger as a whole can also detect wrong memberships and wrong and missing rules.

2.2 Abbreviated Proof Trees

We describe in this section how the proof trees shown in the previous section can be abbreviated in order to ease the questions posed to the user while keeping the completeness and correctness of the technique. To achieve this aim we extend the notion of $APT(T)$ introduced in [14]; $APT(T)$ (from *Abbreviated Proof Tree*) is obtained by a transformation based on deleting nodes whose correctness only depends on the correctness of their children. For example, nodes related to judgments about sets of substitutions, that can be complicated due to matching modulo, are removed.

The rules to compute the abbreviated proof tree, which are assumed to be applied in order (i.e., a rule cannot be applied if there is another one with a lower index that can be used), are described in Figure 5:

- Rule ($\mathbf{APT}_1$) keeps the root of the tree and applies the general function APT', that returns a set of trees, to the tree.
- Rule ($\mathbf{APT}_2$) improves the questions presented to the user when the inference rule NTr is used. This abbreviation associates the equation applied in the left branch (in the inference rule Rdc_1) to the judgment rooting the tree. In this way we ask about reductions to normal form instead of reductions in one step.

$$(\mathbf{APT}_1)\ APT\left(\frac{T_1 \ldots T_n}{aj} R_1\right) = \frac{APT'\left(\frac{T_1 \ldots T_n}{aj} R_1\right)}{aj} R_1$$

$$(\mathbf{APT}_2)\ APT'\left(\frac{\frac{T_1 \ \ldots \ T_n}{t \to t''} \mathsf{Rdc}_1 \quad T'}{t \to t'} \mathsf{NTr}\right) = \left\{\frac{APT'(T_1) \ \ldots \ APT'(T_n) \ APT'(T')}{t \to t'} \mathsf{Rdc}_1\right\}$$

$$(\mathbf{APT}_3)\ APT'\left(\frac{T_{t \to_{norm} t'} \ T_1 \ldots T_n}{t :_{ls} s} \mathsf{Ls}\right) = \left\{\frac{APT'(T_{t \to_{norm} t'}) \ APT'(T_1) \ \ldots \ APT'(T_n)}{t' :_{ls} s} \mathsf{Ls}\right\}$$

$$(\mathbf{APT}_4)\ APT'\left(\frac{T_1 \ldots T_n}{aj} R_2\right) = \left\{\frac{APT'(T_1) \ \ldots \ APT'(T_n)}{aj} R_2\right\}$$

$$(\mathbf{APT}_5)\ APT'\left(\frac{T_1 \ldots T_n}{aj} R_1\right) = APT'(T_1) \bigcup \ldots \bigcup APT'(T_n)$$

R_1 any inference rule $\qquad R_2$ Rdc_1, or Norm $\qquad aj$ any judgment

Fig. 5. APT rules

– Rule $(\mathbf{APT}_3)$ improves the questions about least sorts by asking about the normal form of the term and thus the user is not in charge of computing it.
– Rule $(\mathbf{APT}_4)$ keeps the conclusion of the inference rules that contain debugging information.
– Rule $(\mathbf{APT}_5)$ discards the conclusion of the rules which do not contain debugging information.

Theorem 2. *Let T be a finite proof tree representing an inference in the calculus of Figures 1 and 2 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

– *$APT(T)$ contains at least one buggy node (completeness).*
– *Any buggy node in $APT(T)$ has an associated wrong equation, missing equation, or missing membership axiom in $\mathcal{R}$ (correctness).*

The abbreviated proof tree obtained by applying these rules to the proof tree depicted in Figures 3 and 4 is shown in Figure 6. This proof tree has been obtained by combining different features available in our tool:

– Judgments of the form $t \to_{norm} t$, that indicate that t is in normal form, are dropped from the proof tree if they are built only with constructors. In our example, the nodes corresponding to $h \to_{norm} h$ have been removed.
– Only labeled statements generate nodes in the abbreviated proof tree. For example, the equation to reduce the constant `heap` is not labeled and thus the node `heap` $\to_{red} h$ (or its corresponding abbreviation) does not appear in the abbreviated tree. Moreover, the debugger provides some other trusting mechanisms: statements and imported modules can be trusted before starting the debugging process; statements can also be trusted on the fly; and a correct module, introduced before starting the debugging process, can be used as oracle before asking the user.

$$\dfrac{\dfrac{\dfrac{}{(\ddagger)\ \texttt{mt} :_{ls} \texttt{Heap}}\ \textsf{Ls}}{(\dagger)\ h :_{ls} \texttt{NeBTree}}\ \textsf{Ls}}{\texttt{heap} :_{ls} \texttt{NeBTree}}\ \textsf{Ls}$$

Fig. 6. Abbreviated proof tree for the heap example

- The signature is always considered correct, and hence judgments inferred by using it do not appear in the abbreviated tree. For example, the membership inference h : `BTree` only uses operator declarations and thus it does not appear in the final tree.
- The rest of nodes have been pruned by the APT rules. For example, they prevent all the judgments using substitutions from being asked.

Furthermore, the user can also follow some strategies to reduce the size of the debugging tree:

- If an error is found using a complex initial term, this error can probably be reproduced with a simpler one. Using this simpler term leads to easier debugging sessions.
- When facing a problem with both wrong and missing answers, it is usually better to debug first the wrong answers, because questions related to them are easier to answer and fixing them can also solve the missing answers problem.
- The Maude profiler [5, Chap. 22] indicates the most frequently used statements for a given computation. Trusting these statements will greatly reduce the size of the tree, although it requires the user to make sure that these statements are indeed correct.

Once the tree has been abbreviated we only have a subset of the original nodes and hence only the correctness of the judgments in these nodes concerns the debugging process. We present here the questions derived only from the calculus presented here, while the rest of the questions asked by the debugger can be found in [13]:

- When a term cannot be further reduced and it is not built only by constructors the debugger asks "Is t in normal form?," which is correct if the user expected t to be a normal form.
- When a term t has been reduced by using equations to another term t', the debugger asks questions of the form "Is this reduction correct? $t \rightarrow t'$." These judgments are correct if the user expected t to be reduced to t'.
- When a sort s is inferred for a term t, the debugger prompts questions of the form "Is this membership correct? $t : s$." This judgment is correct if t has sort s.
- When the judgment refers to the least sort ls of a term t, the tool makes questions of the form "Did you expect t to have least sort ls?." In this case, the judgment is correct if the intended least sort of t is exactly ls.

3 A Debugging Session

We describe in this section how to debug the specification shown in Section 1.1. To debug the error discovered in this specification (the least sort of the term `heap` is `NeBTree`) we use the command:

```
Maude> (missing heap : NeBTree .)
```

This command builds the tree depicted in Figure 6 and asks the following question, associated with the node marked with (†) in the figure:[3]

```
Is NeBTree the least sort of mt 4 mt ?
Maude> (no .)
```

Since we expected the term to have sort `NeHeap` the judgment is erroneous and the next question, that is associated to the node (‡) in Figure 6, is:

```
Is Heap the least sort of mt ?
Maude> (yes .)
```

With this answer the node (‡) disappears from the tree and the node (†) becomes buggy, because it is associated to an incorrect judgment and it has no children. The debugger presents the following message:

```
The buggy node is:
The least sort of mt 4 mt is NeBTree
Either the operator ___ needs more membership axioms or the conditions of
the current axioms are not written in the intended way.
```

Actually, if we check the specification we notice that the membership corresponding to the case when both heaps are empty was not stated. We should add to the specification the membership axiom:

```
  mb [h3] : mt E mt : NeHeap .
```

We can use now these heaps to implement another application. We present here a very simple specification of an auction. The module `AUCTION` defines the sort `People` as a multiset of `Person` (a pair of names and bids) and an `Auction` as some people and a heap, defined in `NS-HEAP`, containing elements of the form `[N,S]`, where `N` is a natural number standing for the bid and `S` a `String` with the name of the bidder. The winner of the auction will be the person on the top of the heap:

```
(mod AUCTION is
  pr NS-HEAP .

  sorts Person People Auction .
```

[3] Although the debugger provides two different navigation strategies, in this simple tree both of them choose the same node.

```
  subsort Person < People .

  op <_`,_> : String Nat -> Person [ctor] .
  op nobody : -> People [ctor] .
  op __ : People People -> People [ctor comm assoc id: nobody] .
  op _`[_`] : People Heap -> Auction [ctor] .
```

The rule `bid` inserts a bid into the heap:

```
  var N : Nat .                     var H : Heap .
  var P : People .                  var S : String .

  rl [bid] : (P < S, N >) [H] => P [insert([N,S], H)] .
endm)
```

If we search now for the possible winners of an auction, where `initial` stands for `< "aida", 5 > < "nacho", 4 > < "charlie", 3 > [mt]`:

```
Maude> (search in AUCTION : initial =>!
                            nobody [L:Heap [N:Nat, S:String] R:Heap] .)
No solution.
```

no solutions are found. Since one solution is expected, we debug the specification with the command:

```
Maude> (missing initial =>! nobody [ L:Heap [N:Nat, S:String] R:Heap ] .)
```

This command builds the corresponding debugging tree and traverses it with the default divide and query strategy, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, keeping only this subtree if its root is incorrect, and deleting the whole subtree otherwise. The first question is:

```
Are the following terms all the reachable terms from
(< "aida", 5 > < "charlie", 3 > < "nacho", 4 >)[mt] in one step?
1 (< "aida", 5 > < "nacho", 4 >)[mt [3, "charlie"] mt]
2 (< "aida", 5 > < "charlie", 3 >)[mt [4, "nacho"] mt]
3 (< "charlie", 3 > < "nacho", 4 >)[mt [5, "aida"] mt]
Maude> (yes .)
```

The rule has inserted each person into the heap and thus the transition is correct. After some other questions related to rewrites in the style of [15], the debugger asks:

```
Is insert([4,"nacho"],mt[3,"charlie"]mt) in normal form?
Maude> (no .)
```

This term is not in normal form because we expected `insert` to be reduced. The next questions are also related to normal forms:[4]

[4] Note that, in these cases, the `String` values are not built with constructors and thus this question is not automatically removed by the debugger. If we defined our own constants for the names with the `ctor` attribute, these questions would not appear.

```
Is mt [3, "charlie"] mt in normal form?
Maude> (yes .)

Is [4,"nacho"] in normal form?
Maude> (yes .)
```

In these cases the judgment is correct because no equations should be applied to them. The next questions refer to reductions:

```
Is this reduction (associated with the equation dp1) correct?
depth(mt) -> 0
Maude> (trust .)

Is this reduction (associated with the equation cmp1) correct?
complete(mt) -> true
Maude> (trust .)
```

Since these reductions were associated to simple equations we have used the command `trust` to prevent the debugger from asking questions related to these equations again. The next question deals with memberships:

```
Is this membership (associated with the membership h3) correct?
mt [3, "charlie"] mt : NeHeap
Maude> (yes .)
```

The membership is correct because it only contains the value at the root. With this information the debugger finds the following bug:

```
The buggy node is:
insert([4,"nacho"], mt [3, "charlie"] mt) is in normal form.
Either the operator insert needs more equations or the conditions of
the current equations are not written in the intended way.
```

If we carefully inspect the equations for `insert` we notice that we have not treated the case where the tree is complete and a new level has to be started. We can add the appropriate equation or fix the equation `ins2`, that distinguishes a case that cannot occur in heaps. If we choose the latter, it should be fixed as follows:

```
  ceq [ins2] : insert(E, L E' R) = L' max(E, E') R
   if L E' R : NeHeap /\
      not complete(L) or ((depth(L) == depth(R)) and complete(R)) /\
      L' := insert(min(E, E'), L) .
```

4 Future Work

In this paper we have presented a calculus to debug erroneous normal forms and least sorts by abbreviating the proof trees obtained with it. This calculus, besides allowing to debug these new errors, improves the former versions of our debugger by allowing the debugging of new causes of missing answers in rewrites: missing equations and memberships. These debugging features have also been integrated with the graphical user interface [13].

Although the current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session [14], we also want to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. We also intend to add new navigation strategies like the ones shown in [18] that take into account the number of different potential errors in the subtrees, instead of their size.

Finally, we plan to use the new narrowing features of Maude to implement a test generator for Maude specifications. This generator would allow to check Maude specifications and then to invoke the debugger when one of the test cases fails.

Acknowledgements. We cordially thank Martin Wirsing for encouraging us to investigate the causes for missing answers in a deterministic context, and the referees for their useful comments.

References

1. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract diagnosis of functional programs. In: Leuschel, M. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 1–16. Springer, Heidelberg (2003)
2. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236, 35–132 (2000)
3. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP 2005), Tallinn, Estonia, pp. 8–13. ACM Press, New York (2005)
4. Caballero, R., Rodríguez-Artalejo, M., del Vado Vírseda, R.: Declarative diagnosis of missing answers in constraint functional-logic programming. In: Garrigue, J., Hermenegildo, M.V. (eds.) FLOPS 2008. LNCS, vol. 4989, pp. 305–321. Springer, Heidelberg (2008)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
6. Hendrix, J., Meseguer, J., Ohsaki, H.: A sufficient completeness checker for linear order-sorted specifications modulo axioms. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 151–155. Springer, Heidelberg (2006)
7. MacLarty, I.: Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne (2005)
8. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
9. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
10. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming (3) (1997)

11. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
12. Pope, B.: A Declarative Debugger for Haskell. PhD thesis, The University of Melbourne, Australia (2006)
13. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2009), `http://maude.sip.ucm.es/debugging`
14. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of rewriting logic specifications. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 308–325. Springer, Heidelberg (2009)
15. Riesco, A., Verdejo, A., Martí-Oliet, N.: Declarative debugging of missing answers in rewriting logic. Technical Report SIC-6-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2009), `http://maude.sip.ucm.es/debugging`
16. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Technical Report SIC-02-10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2010), `http://maude.sip.ucm.es/debugging`
17. Shapiro, E.Y.: Algorithmic Program Debugging. In: ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
18. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
19. Tessier, A., Ferrand, G.: Declarative diagnosis in the CLP scheme. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)

International Conference on Rewriting Techniques and Applications 2010 (Edinburgh), pp. 277-294
http://rewriting.loria.fr/rta/

DECLARATIVE DEBUGGING OF MISSING ANSWERS FOR MAUDE SPECIFICATIONS

ADRIÁN RIESCO [1] AND ALBERTO VERDEJO [1] AND NARCISO MARTÍ-OLIET [1]

[1] Facultad de Informática, Universidad Complutense de Madrid, Spain
E-mail address: `ariesco@fdi.ucm.es,{alberto,narciso}@sip.ucm.es`

Abstract. Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error by building a tree representing this computation and guiding the user through it to find the error. Membership equational logic (*MEL*) is an equational logic that in addition to equations allows to state of membership axioms characterizing the elements of a sort. Rewriting logic is a logic of change that extends *MEL* by adding rewrite rules, that correspond to transitions between states and can be nondeterministic. In this paper we propose a calculus to infer normal forms and least sorts with the equational part, and sets of reachable terms through rules. We use an abbreviation of the proof trees computed with this calculus to build appropriate debugging trees for missing answers (results that are erroneous because they are incomplete), whose adequacy for debugging is proved. Using these trees we have implemented a declarative debugger for Maude, a high-performance system based on rewriting logic, whose use is illustrated with an example.

1. Introduction

Declarative debugging [20], also known as declarative diagnosis or algorithmic debugging, is a debugging technique that abstracts the execution details, which may be difficult to follow in declarative languages, and focus on the results. We can distinguish between two different kinds of declarative debugging: debugging of *wrong answers*, that is applied when a *wrong* result is obtained from an initial value and has been widely employed in the logic [12, 22], functional [14, 15], multi-paradigm [3, 9], and object-oriented [4] programming languages; and debugging of *missing answers* [5, 1], applied when a result is *incomplete*, which has been less studied because the calculus involved is more complex than in the case of wrong answers. Declarative debugging starts from an incorrect computation, the error symptom, and locates the code (or the absence of code) responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [13], where each node stands for a computation step and must follow from the results of its child nodes by some logical inference. This tree is traversed by asking questions to an external oracle (generally the user)

1998 ACM Subject Classification: D.2.5 [Software Engineering]: Testing and Debugging – Debugging aids, D.3.2 [PROGRAMMING LANGUAGES]: Language Classifications – Nondeterministic languages.

Key words and phrases: Declarative debugging, Maude, Missing answers, Rewriting Logic.

Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

Proceedings of the 21st International Conference on Rewriting Techniques and Applications, Edinburgh, July, 2010
Editors: Christopher Lynch
LIPIcs - Leibniz International Proceedings in Informatics. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Germany
Digital Object Identifier: 10.4230/LIPIcs.RTA.2010.277

until a *buggy node*—a node containing an erroneous result, but whose children are all correct—is found. Hence, we distinguish two phases in this scheme: the debugging tree *generation* and its *navigation* following some suitable strategy [21].

In this paper we present a declarative debugger of missing answers for *Maude specifications*. Maude [6] is a high-level language and high-performance system supporting both equational and rewriting logic computation. The Maude system supports several approaches for debugging: tracing, term coloring, and using an internal debugger [6, Chap. 22]. However, these tools have the disadvantages that they are supposed to be used only when a wrong result *is found*; and both the trace and the Maude debugger (that is based on the trace) show the statements applied in the order in which they are executed and thus the user can lose the general view of the *proof* of the incorrect computation that produced the wrong result.

Declarative debugging of wrong answers in Maude specifications was already studied in [17], where we presented how to debug wrong results due to errors in the statements of the specification. In [18] we extended the concept of missing answers, usually attached to incomplete sets of results, to deal with erroneous normal forms and least sorts in equational theories. However, in a nondeterministic context such as that of Maude modules other problems can arise. We show in this paper how to debug missing answers in rewriting specifications, that is, expected results that the specification is not able to compute. This kind of problems appears in Maude when using its breadth-first search, that finds all the reachable terms from an initial one given a pattern, a condition, and a bound in the number of steps. To debug this kind of errors we have extended our calculus to deduce sets of reachable terms given an initial term, a bound in the number of rewrites, and a condition to be fulfilled. Unlike other proposals like [5], our debugging framework combines the treatment of wrong and missing answers and, moreover, is able to detect missing answers due to both missing rules and wrong statements. The state of the art can be found in [21], where different algorithmic debuggers are compared and that will include our debugger in its next version. Roughly speaking, our debugger has the pros of building different kinds of debugging trees (one-step and many-steps) and applying the missing answers technique to debug normal forms and least sorts[1] (the different trees are a novelty in the declarative debugging world), and only it and DDT [3] implement the Hirunkitti's divide and query navigation strategy, provide a graphical interface, and debug missing answers; as cons, we do not provide answers like "maybe yes," "maybe not," and "inadmissible," and do not perform tree compression. However, these features have recently been introduced in specific debuggers, and we expect to implement them in our debugger soon. Finally, some of the features shared by most of the debuggers are: the trees are abbreviated in order to shorten and ease the debugging process (in our case, since we obtain the trees from a formal calculus, we are able to prove the correctness and completeness of the technique), which mitigates the main problem of declarative debugging, the complexity of the questions asked to the user; trusting of statements; `undo` and `don't know` commands; and different strategies to traverse the tree. We refer to [21, 16] for the meaning of these concepts. With respect to other approaches, such as the Maude sufficient completeness checker [6, Chap. 21] or the sets of descendants [8], our tool provides a wider approach, since we handle conditional statements and our equations are not required to be left-linear.

The rest of the paper is structured as follows. Section 2 provides a summary of the main concepts of rewriting logic and Maude specifications. Section 3 describes our calculus and Section 4 shows the debugging trees obtained from it. Finally, Section 5 concludes and mentions some future work.

Detailed proofs of the results can be found in [19], while additional examples, the source code of the tool, and other papers on the subject, including the user guide [16], where a graphical user interface for the debugger is presented, are all available from the webpage `http://maude.sip.ucm.es/debugging`.

[1]Although the least sort error can be seen as a Maude-directed problem, normal forms are a common feature in several programming languages.

2. Rewriting Logic and Maude

Maude modules are executable rewriting logic specifications. Rewriting logic [10] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses membership equational logic (*MEL*) [2], which, in addition to equations, allows to state of membership axioms characterizing the elements of a sort. Rewriting logic extends *MEL* by adding rewrite rules.

For our purposes in this paper, we are interested in a subclass of rewriting logic models [10] that we call *term models*, where the syntactic structure of terms is kept and associated notions such as variables, substitutions, and term rewriting make sense. These models will be used in Section 4 to represent the *intended interpretation* that the user had in mind while writing a specification. Since we want to find the discrepancies between the intended model and the initial model of the specification as written, we need to consider the relationship between a specification defined by a set of equations E and a set of rules R, and a model defined by possibly different sets of equations E' and of rules R'; in particular, when $E' = E$ and $R' = R$, the term model coincides with the initial model built in [10].

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with Σ a signature, E a set of equations, and R a set of rules, a Σ-term model has an underlying (Σ, E')-algebra whose elements are equivalence classes $[t]_{E'}$ of ground Σ-terms modulo some set of equations and memberships E' (which may be different from E), and there is a transition from $[t]_{E'}$ to $[t']_{E'}$ when $[t]_{E'} \rightarrow^*_{R'/E'} [t']_{E'}$, where rewriting is considered on equivalence classes [10, 7]. The set of rules R' may also be different from R, that is, the term model is $\mathcal{T}_{\Sigma/E',R'}$ for some E' and R'. In such term models, the notion of valuation coincides with that of (ground) substitution. A term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies, under a substitution θ, an equation $u = v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u = v$, when $\theta(u) =_{E'} \theta(v)$, or equivalently, when $[\theta(u)]_{E'} = [\theta(v)]_{E'}$; a membership $u : s$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u : s$, when the Σ-term $\theta(u)$ has sort s according to the information in the signature Σ and the equations and memberships E'; a rewrite $u \Rightarrow v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u \Rightarrow v$, when there is a transition in $\mathcal{T}_{\Sigma/E',R'}$ from $[\theta(u)]_{E'}$ to $[\theta(v)]_{E'}$, that is, when $[\theta(u)]_{E'} \rightarrow^*_{R'/E'} [\theta(v)]_{E'}$. Satisfaction is extended to conditional sentences as usual. A Σ-term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ when $\mathcal{T}_{\Sigma/E',R'}$ satisfies the equations and memberships in E and the rewrite rules in R in this sense. For example, this is obviously the case when $E \subseteq E'$ and $R \subseteq R'$; as mentioned above, when $E' = E$ and $R' = R$ the term model coincides with the initial model for $\mathcal{R}$.

Maude functional modules [6, Chap. 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications that allow the definition of sorts (by means of keyword `sort(s)`); *subsort* relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Maude system modules [6, Chap. 6], introduced with syntax `mod ... endm`, are executable rewrite theories. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

We present how to use this syntax by means of an example. Given a maze, we want to obtain all the possible paths to the exit. First, we define the sorts `Pos`, `List`, and `State` that stand for positions in the labyrinth, lists of positions, and the path traversed so far respectively:

```
(mod MAZE is
  pr NAT .              sorts Pos List State .
```

Terms of sort `Pos` have the form `[X,Y]`, where `X` and `Y` are natural numbers, and lists are built with `nil` and the juxtaposition operator `_ _`:

```
  subsort Pos < List .       op [_,_] : Nat Nat -> Pos [ctor] .
  op nil : -> List [ctor] . op _ _ : List List -> List [ctor assoc id: nil] .
```

Terms of sort `State` are lists enclosed by curly brackets, that is, `{_}` is an "encapsulation operator" that ensures that the whole state is used:

```
  op {_} : List -> State [ctor] .
```

The predicate `isSol` checks whether a list is a solution in a 5×5 labyrinth:

```
  vars X Y : Nat .     var P Q : Pos .     var L : List .
  op isSol : List -> Bool .
  eq [is1] : isSol(L [5,5]) = true .
  eq [is2] : isSol(L) = false [owise] .
```

The next position is computed with rule `expand`, that extends the solution with a new position by rewriting `next(L)` to obtain a new position and then checking whether this list is correct with `isOk`. Note that the choice of the next position, that could be initially wrong, produces an implicit backtracking:

```
  crl [expand] : { L } => { L P } if next(L) => P /\ isOk(L P) .
```

The function `next` is defined in a nondeterministic way, where `sd` denotes the symmetric difference:

```
  op next : List -> Pos .
  rl [n1] : next(L [X,Y]) => [X, Y + 1] .
  rl [n2] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [n3] : next(L [X,Y]) => [X, sd(Y, 1)] .
```

`isOk(L P)` checks that the position `P` is within the limits of the labyrinth, not repeated in `L`, and not part of the wall by using an auxiliary function `contains`:

```
  op isOk : List -> Bool .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 5 and Y <= 5
             and not(contains(L, [X,Y])) and not(contains(wall, [X,Y])) .
  op contains : List Pos -> Bool .
  eq [c1] : contains(nil, P) = false .
  eq [c2] : contains(Q L, P) = if P == Q then true else contains(L, P) fi .
```

Finally, we define the `wall` of the labyrinth as a list of positions:

```
  op wall : -> List .
  eq wall = [2,1] [2,2] [3,2] [2,3] [4,3] [5,3] [1,5] [2,5] [3,5] [4,5] .
endm)
```

Now, we can use the module to search the labyrinth's exit from the position `[1,1]` with the Maude command `search`, but it cannot find any path to escape. We will see in Section 4.1 how to debug it.

3. A Calculus for Missing Answers

We describe in this section a calculus to infer, given a term and some constraints, the *complete* set of reachable terms from this term that fulfill the requirements. The proof trees built with this calculus have nodes that justify why the terms are included in the corresponding sets (positive information) but also nodes that justify why there are no more terms (negative information). These latter nodes are then used in the debugging trees to localize as much as possible the reasons responsible for missing answers. This calculus integrates the calculus to deduce substitutions, normal forms, and least sorts that was presented in [18], and that we reproduce here to give the reader an overall view of debugging of missing answers in Maude specifications. Moreover, these calculi extend the calculus in [17], used to deduce judgments corresponding to oriented equations $t \to t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$, and to debug wrong answers. All the results in this paper refer to the complete calculus comprising these three calculi, and thus we consider this work as the final step in the development of foundations for a complete declarative debugger for Maude.

From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the Maude executability requirements, i.e., E is confluent, terminating, maybe modulo some equational axioms such as associativity and commutativity, and sort-decreasing, while R is coherent with respect to E; see [6] for details. Equations corresponding to the axioms form the set A and the equations in $E - A$ can be oriented from left to right.

We introduce the inference rules used to obtain the set of reachable terms given an initial one, a pattern [6], a condition, and a bound in the number of rewrites. First, the pattern P and the condition $\mathcal{C}$ (that can use variables bound by the pattern) are put together by creating the condition $\mathcal{C}' \equiv P := ⊛ \wedge \mathcal{C}$, where $⊛$ is a "hole" that will be filled by the concrete terms to check if they fulfill both the pattern and the condition. Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*. They correspond to the ones used in Maude modules and are defined as follows:

Definition 3.1. A condition $C_1 \wedge \cdots \wedge C_n$ is *admissible* if, for $1 \le i \le n$, C_i is

- an equation $u_i = u'_i$ or a membership $u_i : s$ and $vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or
- a matching condition $u_i := u'_i$, u_i is a pattern and $vars(u'_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$, or
- a rewrite condition $u_i \Rightarrow u'_i$, u'_i is a pattern and $vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j)$.

Note that the lefthand side of matching conditions and the righthand side of rewrite conditions can contain extra variables that will be instantiated once the condition is solved.

Definition 3.2. A condition $\mathcal{C} \equiv P := ⊛ \wedge C_1 \wedge \cdots \wedge C_n$ is *admissible* if $P := t \wedge C_1 \wedge \cdots \wedge C_n$ is admissible for t any ground term.

Definition 3.3. A *kind-substitution*, denoted by κ, is a mapping between variables and terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \le i \le n} . kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the term it binds.

Definition 3.4. A *substitution*, denoted by θ, is a mapping between variables and terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \le i \le n} . sort(v_i) \ge ls(t_i)$, that is, the sort of each variable is greater than or equal to the least sort of the term it binds. Note that a substitution is a special type of kind-substitution where each term has the sort appropriate to its variable.

Definition 3.5. Given an atomic condition C, we say that a substitution θ is *admissible for* C if

- C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or
- C is a matching condition $u := u'$ and $vars(u') \subseteq dom(\theta)$, or
- C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.

The calculus presented in this section (in Figures 1–4) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model $\mathcal{T}' = \mathcal{T}_{\Sigma/E',R'}$ defined by equations and memberships E' and by rules R':

- Given a term t and a kind-substitution κ, $\mathcal{T}' \models adequateSorts(\kappa) \rightsquigarrow \Theta$ when either $\Theta = \{\kappa\} \wedge \forall v \in dom(\kappa). \mathcal{T}' \models \kappa[v] : sort(v)$ or $\Theta = \emptyset \wedge \exists v \in dom(\kappa). \mathcal{T}' \not\models \kappa[v] : sort(v)$, where $\kappa[v]$ denotes the term bound by v in κ. That is, when all the terms bound in the kind-substitution κ have the appropriate sort, then κ is a substitution and it is returned; otherwise (at least one of the terms has an incorrect sort), the kind-substitution is not a substitution and the empty set is returned.
- Given an admissible substitution θ for an atomic condition C, $\mathcal{T}' \models [C, \theta] \rightsquigarrow \Theta$ when $\Theta = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \restriction_{dom(\theta)} = \theta\}$, that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ by binding the new variables appearing in C.

- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta \rangle \rightsquigarrow \Theta'$ when $\Theta' = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \upharpoonright_{dom(\theta)} = \theta \text{ for some } \theta \in \Theta\}$, that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.
- $\mathcal{T}' \models \mathit{disabled}(a, t)$ when the equation or membership a cannot be applied to t at the top.
- $\mathcal{T}' \models t \rightarrow_{red} t'$ when either $\mathcal{T}' \models t \rightarrow^1_{E'} t'$ or $\mathcal{T}' \models t_i \rightarrow^!_{E'} t'_i$, with $t_i \neq t'_i$, for some subterm t_i of t such that $t' = t[t_i \mapsto t'_i]$, that is, the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.
- $\mathcal{T}' \models t \rightarrow_{norm} t'$ when $\mathcal{T}' \models t \rightarrow^!_{E'} t'$, that is, t' is in normal form with respect to the equations E'.
- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t)$ when there exists a substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ holds when $\circledast$ is substituted by t.
- Given an admissible condition $\mathcal{C}$ as before, $\mathcal{T}' \models \mathit{fails}(\mathcal{C}, t)$ when there exists *no* substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.
- $\mathcal{T}' \models t :_{ls} s$ when $\mathcal{T}' \models t : s$ and moreover s is the least sort with this property (with respect to the ordering on sorts obtained from the signature Σ and the equations and memberships E' defining the Σ-term model $\mathcal{T}'$).
- $\mathcal{T}' \models t \Rightarrow^{top} S$ when $S = \{t' \mid t \rightarrow^{top}_{R'} t'\}$, that is, the set S is formed by all the reachable terms from t by exactly one rewrite *at the top* with the rules R' defining $\mathcal{T}'$. Moreover, equality in S is modulo E', i.e., we are implicitly working with equivalence classes of ground terms modulo E'.
- $\mathcal{T}' \models t \Rightarrow^q S$ when $S = \{t' \mid t \rightarrow^{top}_{\{q\}} t'\}$, that is, the set S is the complete set of reachable terms (modulo E') obtained from t with one application of the rule $q \in R'$ at the top.
- $\mathcal{T}' \models t \Rightarrow_1 S$ when $S = \{t' \mid t \rightarrow^1_{R'} t'\}$, that is, the set S is constituted by all the reachable terms (modulo E') from t in exactly one step, where the rewrite step can take place anywhere in t.
- $\mathcal{T}' \models t \rightsquigarrow^{\mathcal{C}}_n S$ when $S = \{t' \mid t \rightarrow^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t')\}$, that is, S is the set of all the terms (modulo E') that satisfy the admissible condition $\mathcal{C}$ and are reachable from t in at most n steps.

We first introduce in Figure 1 the inference rules defining the relations $[C, \theta] \rightsquigarrow \Theta$, $\langle C, \Theta \rangle \rightsquigarrow \Theta'$, and $\mathit{adequateSorts}(\kappa) \rightsquigarrow \Theta$. Intuitively, these judgments will provide positive information when they lead to nonempty sets (indicating that the condition holds in the first two judgments or that the kind-substitution is a substitution in the third one) and negative information when they lead to the empty set (indicating respectively that the condition fails or the kind-substitution is not a substitution):

- Rule PatC computes all the possible substitutions that extend θ and satisfy the matching of the term t_2 with the pattern t_1 by first computing the normal form t' of t_2, obtaining then all the possible kind-substitutions κ that make t' and $\theta(t_1)$ equal modulo axioms (indicated by $\equiv_A$), and finally checking that the terms assigned to each variable in the kind-substitutions have the appropriate sort with $\mathit{adequateSorts}(\kappa)$. The union of the set of substitutions thus obtained constitutes the set of substitutions that satisfy the matching.
- Rule AS_1 checks whether the terms of the kind-substitution have the appropriate sort to match the variables. In this case the kind-substitution is a substitution and it is returned.
- Rule AS_2 indicates that, if any of the terms in the kind-substitution has a sort bigger than the required one, then it is not a substitution and thus the empty set of substitutions is returned.
- Rule MbC_1 returns the current substitution if a membership condition holds.

$$\frac{\theta(t_2) \to_{norm} t' \quad adequateSorts(\kappa_1) \rightsquigarrow \Theta_1 \quad \ldots \quad adequateSorts(\kappa_n) \rightsquigarrow \Theta_n}{[t_1 := t_2, \theta] \rightsquigarrow \bigcup_{i=1}^{n} \Theta_i} \ \mathsf{PatC}$$
$$\text{if } \{\kappa_1, \ldots, \kappa_n\} = \{\kappa\theta \mid \kappa(\theta(t_1)) \equiv_A t'\}$$

$$\frac{t_1 : sort(v_1) \quad \ldots \quad t_n : sort(v_n)}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \{v_1 \mapsto t_1; \ldots; v_n \mapsto t_n\}} \ \mathsf{AS_1}$$

$$\frac{t_i :_{ls} s_i}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \emptyset} \ \mathsf{AS_2} \ \text{ if } s_i \not\leq sort(v_i)$$

$$\frac{\theta(t) : s}{[t : s, \theta] \rightsquigarrow \{\theta\}} \ \mathsf{MbC_1} \qquad \frac{\theta(t) :_{ls} s'}{[t : s, \theta] \rightsquigarrow \emptyset} \ \mathsf{MbC_2} \ \text{ if } s' \not\leq s$$

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \rightsquigarrow \{\theta\}} \ \mathsf{EqC_1} \qquad \frac{\theta(t_1) \to_{norm} t'_1 \quad \theta(t_2) \to_{norm} t'_2}{[t_1 = t_2, \theta] \rightsquigarrow \emptyset} \ \mathsf{EqC_2} \ \text{ if } t'_1 \not\equiv_A t'_2$$

$$\frac{\theta(t_1) \rightsquigarrow_{n+1}^{t_2 := \circledast} S}{[t_1 \Rightarrow t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}} \ \mathsf{RlC} \qquad \frac{[C, \theta_1] \rightsquigarrow \Theta_1 \quad \cdots \quad [C, \theta_m] \rightsquigarrow \Theta_m}{\langle C, \{\theta_1, \ldots, \theta_m\}\rangle \rightsquigarrow \bigcup_{i=1}^{m} \Theta_i} \ \mathsf{SubsCond}$$
$$\text{if } n = min(x \in \mathbb{N} : \forall i \geq 0 \ (\theta(t_1) \rightsquigarrow_{x+i}^{t_2 := \circledast} S))$$

Figure 1: Calculus for substitutions

- Rule $\mathsf{MbC_2}$ is used when the membership condition is not satisfied. It checks that the least sort of the term is not less than or equal to the required one, and thus the substitution does not satisfy the condition and the empty set is returned.
- Rule $\mathsf{EqC_1}$ returns the current substitution when an equality condition holds, that is, when the two terms can be joined with equations, abbreviated as $t_1 \downarrow t_2$.
- Rule $\mathsf{EqC_2}$ checks that an equality condition fails by obtaining the normal forms of both terms and then examining that they are different.
- Rewrite conditions are handled by rule RlC. This rule extends the set of substitutions by computing all the reachable terms that satisfy the pattern (using the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ explained below) and then using these terms to obtain the new substitutions.
- Finally, rule $\mathsf{SubsCond}$ computes the extensions of a set of admissible substitutions $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

We use these judgments to define the inference rules of Figure 2, that describe how the normal form and the least sort of a term are computed:

- Rule Dsb indicates when an equation or membership a cannot be applied to a term t. It checks that there are no substitutions that satisfy the matching of the term with the lefthand side of the statement and that fulfill its condition. Note that we check the conditions from left to right, following the same order as Maude and making all the substitutions admissible.
- Rule $\mathsf{Rdc_1}$ reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions. While in the previous rule we made explicit the evaluation from left to right of the condition to show that finally the set of substitutions fulfilling it was empty, in this case we only need one substitution to fulfill the condition and the order is unimportant.
- Rule $\mathsf{Rdc_2}$ reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \quad \dots \quad \langle C_n, \Theta_{n-1}\rangle \rightsquigarrow \emptyset}{\mathit{disabled}(a, t)} \ \mathsf{Dsb}$$

$$\text{if } a \equiv l \to r \Leftarrow C_1 \wedge \ldots \wedge C_n \in E \text{ or}$$
$$a \equiv l : s \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^n \quad \{\theta(v_j) : s_j\}_{j=1}^m}{\theta(l) \to_{red} \theta(r)} \ \mathsf{Rdc}_1 \ \text{ if } l \to r \Leftarrow \bigwedge_{i=1}^n u_i = u'_i \wedge \bigwedge_{j=1}^m v_j : s_j \in E$$

$$\frac{t \to_{norm} t'}{f(t_1, \ldots, t, \ldots, t_n) \to_{red} f(t_1, \ldots, t', \ldots, t_n)} \ \mathsf{Rdc}_2 \ \text{ if } t \not\equiv_A t'$$

$$\frac{\mathit{disabled}(e_1, f(t_1, \ldots, t_n)) \quad \dots \quad \mathit{disabled}(e_l, f(t_1, \ldots, t_n)) \quad t_1 \to_{norm} t_1 \quad \dots \quad t_n \to_{norm} t_n}{f(t_1, \ldots, t_n) \to_{norm} f(t_1, \ldots, t_n)} \ \mathsf{Norm}$$
$$\text{if } \{e_1, \ldots, e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1, \ldots, t_n)\}$$

$$\frac{t \to_{red} t_1 \quad t_1 \to_{norm} t'}{t \to_{norm} t'} \ \mathsf{NTr} \qquad \frac{t \to_{norm} t' \quad t' : s \quad \mathit{disabled}(m_1, t') \quad \dots \quad \mathit{disabled}(m_l, t')}{t :_{ls} s} \ \mathsf{Ls}$$
$$\text{if } \{m_1, \ldots, m_l\} = \{m \in E \mid m \ll_K^{top} t' \ \wedge \ \mathit{sort}(m) < s\}$$

Figure 2: Calculus for normal forms and least sorts

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}} \ \mathsf{Rf}_1 \qquad \frac{\mathit{fails}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset} \ \mathsf{Rf}_2$$

$$\frac{\theta(P) \downarrow t \quad \{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^n \quad \{\theta(v_j) : s_j\}_{j=1}^m \quad \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^l}{\mathit{fulfilled}(\mathcal{C}, t)} \ \mathsf{Fulfill}$$
$$\text{if } \mathcal{C} \equiv P := \circledast \wedge \bigwedge_{i=1}^n u_i = u'_i \wedge \bigwedge_{j=1}^m v_j : s_j \wedge \bigwedge_{k=1}^l w_k \Rightarrow w'_k$$

$$\frac{[P := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1}\rangle \rightsquigarrow \emptyset}{\mathit{fails}(\mathcal{C}, t)} \ \mathsf{Fail} \ \text{ if } \mathcal{C} \equiv P := \circledast \wedge C_1 \ \wedge \ \ldots \ \wedge \ C_k$$

Figure 3: Calculus for solutions

- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top considering the variables at the kind level (which is indicated by $\ll_K^{top}$) and that all its subterms are already in normal form.
- Rule NTr describes the transitivity for the reduction to normal form. It reduces the term with the relation $\to_{red}$ and the term thus obtained then is reduced to normal form by using again $\to_{norm}$.
- Rule Ls computes the least sort of the term t. It computes a sort for its normal form (that has the least sort of the terms in the equivalence class) and then checks that memberships deducing lesser sorts, applicable at the top with the variables considered at the kind level, cannot be applied.

In these rules Dsb provides the negative information, proving why the statements (either equations or membership axioms) cannot be applied, while the remaining rules provide the positive information indicating why the normal form and the least sort are obtained.

Once these rules have been introduced, we can use them in the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$. First, we present in Figure 3 the rules related to $n = 0$ steps:

$$\frac{\mathit{fulfilled}(\mathcal{C},t) \quad t \Rightarrow_1 \{t_1,\ldots,t_k\} \quad t_1 \rightsquigarrow^{\mathcal{C}}_n S_1 \;\ldots\; t_k \rightsquigarrow^{\mathcal{C}}_n S_k}{t \rightsquigarrow^{\mathcal{C}}_{n+1} \bigcup_{i=1}^{k} S_i \;\cup\; \{t\}}\ \mathsf{Tr_1}$$

$$\frac{\mathit{fails}(\mathcal{C},t) \quad t \Rightarrow_1 \{t_1,\ldots,t_k\} \quad t_1 \rightsquigarrow^{\mathcal{C}}_n S_1 \;\ldots\; t_k \rightsquigarrow^{\mathcal{C}}_n S_k}{t \rightsquigarrow^{\mathcal{C}}_{n+1} \bigcup_{i=1}^{k} S_i}\ \mathsf{Tr_2}$$

$$\frac{f(t_1,\ldots,t_m) \Rightarrow^{top} S_t \quad t_1 \Rightarrow_1 S_1 \quad \cdots \quad t_m \Rightarrow_1 S_m}{f(t_1,\ldots,t_m) \Rightarrow_1 S_t \;\cup\; \bigcup_{i=1}^{m}\{f(t_1,\ldots,u_i,\ldots,t_m) \mid u_i \in S_i\}}\ \mathsf{Stp}$$

$$\frac{t \Rightarrow^{q_1} S_{q_1} \quad \cdots \quad t \Rightarrow^{q_l} S_{q_l}}{t \Rightarrow^{top} \bigcup_{i=1}^{l} S_{q_i}}\ \mathsf{Top} \quad \text{if } \{q_1,\ldots,q_l\} = \{q \in R \mid q \ll^{top}_{K} t\}$$

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \;\cdots\; \langle C_k, \Theta_{k-1}\rangle \rightsquigarrow \Theta_k}{t \Rightarrow^{q} \bigcup_{\forall\,\theta \in \Theta_k} \{\theta(r)\}}\ \mathsf{Rl} \quad \text{if } q : l \Rightarrow r \Leftarrow C_1 \wedge \ldots \wedge C_k \in R$$

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow^{\mathcal{C}}_n \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow^{\mathcal{C}}_n \{t'\} \cup S}\ \mathsf{Red}$$

Figure 4: Calculus for missing answers

- Rule Rf$_1$ indicates that when only zero steps are used and the current term fulfills the condition, the set of reachable terms consists only of this term.
- Rule Rf$_2$ complements Rf$_1$ by defining the empty set as result when the condition does not hold.
- Rule Fulfill checks whether a term satisfies a condition. The premises of this rule check that all the atomic conditions hold, taking into account that it starts with a matching condition with a hole that must be filled with the current term and thus proved with the premise $\theta(P) \downarrow t$. Note that when the condition is satisfied we do not need to check *all* the substitutions, but only to verify that there exists *one* substitution that makes the condition true.
- To check that a term does not satisfy a condition, it is not enough to check that there exists a substitution that makes it to fail; we must make sure that there is no substitution that makes it true. We use the rules shown in Figure 1 to prove that the set of substitutions that satisfy the condition (where the first set of substitutions is obtained from the first matching condition filling the hole with the current term) is empty. Note that, while rule Fulfill provides the positive information indicating that a condition is fulfilled, this one provides negative information, proving that the condition fails.

Now we introduce in Figure 4 the rules defining the relation $t \rightsquigarrow^{\mathcal{C}}_n S$ when the bound n is greater than 0, which can be understood as searches in *zero or more* steps:

- Rules Tr$_1$ and Tr$_2$ show the behavior of the calculus when at least one step can be used. First, we check whether the condition holds (rule Tr$_1$) or not (rule Tr$_2$) for the current term, in order to introduce it in the result set. Then, we obtain all the terms reachable in one step with the relation $\Rightarrow_1$, and finally we compute the reachable solutions from these terms constrained by the same condition and the bound decreased in one step. The union of the sets obtained in this way and the initial term, if needed, corresponds to the final result set.

- Rule Stp shows how the set for one step is computed. The result set is the union of the terms obtained by applying each rule *at the top* (calculated with $t \Rightarrow^{top} S$) and the terms obtained by rewriting one step the arguments of the term. This rule can be straightforwardly adapted to the more general case in which the operator f has some *frozen* arguments (i.e., that cannot be rewritten); the implementation of the debugger makes use of this more general rule.
- How to obtain the terms by rewriting at the top is explained by rule Top, that specifies that the result set is the union of the sets obtained with all the possible applications of each rule of the program. We restrict these rules to those whose lefthand side, with the variables considered at the kind level, matches the term.
- Rule Rl uses the rules in Figure 1 to compute the set of terms obtained with the application of a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed, and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms. The kind of information provided by this rule corresponds to the information provided by the substitutions; if the empty set of substitutions is obtained (negative information) then the rule computes the empty set of terms, which also corresponds with negative information proving that no terms can be obtained with this program rule; analogously when the set of substitutions is nonempty (positive information). This information is propagated through the rest of inference rules justifying why some terms are reachable while others are not.
- Finally, rule Red reduces the reachable terms in order to obtain their normal forms. We use this rule to reproduce Maude behavior, first the normal form is computed and then the rules are applied.

Now we state that this calculus is correct in the sense that the derived judgments with respect to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ coincide with the ones satisfied by the corresponding initial model $\mathcal{T}_{\Sigma/E,R}$, i.e., for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$. This is well known for the judgments corresponding to equations $t = t'$, memberships $t : s$, and rewrites $t \Rightarrow t'$ [11, 10].

Theorem 3.6. *The calculus of Figures 1, 2, 3, and 4 is correct.*

Once these rules are defined, we can build the tree corresponding to the search result shown in Section 2 for the maze example. We recall that we have defined a system to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it. First of all, we have to use a concrete bound to build the tree. It must suffice to compute all the reachable terms, and in this case the least of these values is 4. We have depicted the tree in Figure 5, where we have abbreviated the equational condition `{L:List}` $:= \circledast \wedge$ `isSol(L:List) = true` by $\mathcal{C}$ and `isSol(L) = true` by `isSol(L)`. The leftmost tree justifies that the search condition does not hold for the initial term (this is the reason why Tr_2 has been used instead of Tr_1) and thus it is not a solution. Note that first the substitutions from the matching with the pattern are obtained (`L` $\mapsto$ `[1,1]` in this case), and then these substitutions are used to instantiate the rest of the condition, that for this term does not hold, which is proved by $*_1$. The next tree shows the set of reachable terms in one step (the tree $*_2$, explained below, computes the terms obtained by rewrites at the top, while the tree on its right shows that the subterms cannot be further rewritten) and finally the rightmost tree, that has a similar structure to this one and will not be studied in depth, continues the search with a decreased bound.

The tree $*_1$ shows why the current list is not a solution (i.e., the tree provides the negative information proving that this fragment of the condition does not hold). The reason is that the term `isSol(L)` is reduced to `false`, when we needed it to be reduced to `true`.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \to_{norm} 1}\,\text{Norm}}{\texttt{[1,1]} \to_{norm} \texttt{[1,1]}}\,\text{Norm}}{\texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\,\text{Norm}}{[\texttt{\{L:List\}} := \texttt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \texttt{L} \mapsto \texttt{[1,1]\}}}\,\text{PatC} \quad \dfrac{*_1}{\langle \texttt{isSol(L)}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \rightsquigarrow \emptyset}\,\text{SubsCond}}{fails(\mathcal{C}, \texttt{\{[1,1]\}})}\,\text{Fail} \quad \dfrac{*_2 \quad \dfrac{\dfrac{}{\texttt{[1,1]} \Rightarrow^{top} \emptyset}\,\text{Top} \quad \dfrac{\dfrac{}{1 \Rightarrow^{top} \emptyset}\,\text{Top}}{1 \Rightarrow_1 \emptyset}\,\text{Stp}}{\texttt{[1,1]} \Rightarrow_1 \emptyset}\,\text{Stp}}{\texttt{\{[1,1]\}} \Rightarrow_1 \texttt{\{[1,1][1,2]\}}}\,\text{Stp} \quad \dfrac{*_3}{\texttt{\{[1,1][1,2]\}} \rightsquigarrow^{\mathcal{C}}_3 \emptyset}\,\text{Tr}_2}{\texttt{\{[1,1]\}} \rightsquigarrow^{\mathcal{C}}_4 \emptyset}\,\text{Tr}_2$$

Figure 5: Tree for the maze example

$$\dfrac{\dfrac{\dfrac{}{\texttt{isSol([1,1])} \to_{red} \texttt{false}}\,\text{Rdc}_1 \quad \dfrac{}{\texttt{false} \to_{norm} \texttt{false}}\,\text{Norm}}{\texttt{isSol([1,1])} \to_{norm} \texttt{false}}\,\text{NTr} \quad \dfrac{}{\texttt{true} \to_{norm} \texttt{true}}\,\text{Norm}}{[\texttt{isSol(L) = true}, \texttt{L} \mapsto \texttt{[1,1]}] \rightsquigarrow \emptyset}\,\text{EqC}_2$$

Figure 6: Tree $*_1$ for the search condition

The tree labeled with $*_2$ is sketched in Figure 7. In this tree the applications of all the rules whose lefthand side matches the current term (`{[1,1]}`) are tried. In this case only the rule `expand` (abbreviated by `e`) can be used, and it generates a list with the new position `[1,2]`; the tree $*_4$ is used to justify that the first condition of `expand` holds and extends the set of substitutions that fulfill the condition thus far to the set $\{\theta_1, \theta_2, \theta_3\}$, where $\theta_1 \equiv \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,2]}$, $\theta_2 \equiv \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,0]}$, and $\theta_3 \equiv \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[0,1]}$. The substitution θ_1 also fulfills the next condition, `isOk(L P)`, which is proved with the rule EqC_1 in ♣ (where the big triangle is a computation in the calculus of [17] proving that the conditions of the equations hold), while the substitutions θ_2 and θ_3 fail; the trees $\bigtriangledown$ proving it are analogous to the one shown in Figure 6. This substitution θ_1 is thus the only one inferred in the root of the tree, where the node ♣ provides the positive information proving why the substitution is obtained and its siblings ($\bigtriangledown$) the negative information proving why the other substitutions are not in the set.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{1 \to_{norm} 1}\,\text{Norm}}{\texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\,\text{Norm}}{[\texttt{\{L\}} := \texttt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]}\}}\,\text{PatC} \quad *_4 \quad \dfrac{\dfrac{\dfrac{\bigtriangledown}{\texttt{isOk([1,1][1,2])} \to \texttt{true}}\,\text{Rep}_{\to} \quad \dfrac{}{\texttt{true} \to \texttt{true}}\,\text{Rf}_{\to}}{(♣)\ [\texttt{isOk(L P)}, \theta_1] \rightsquigarrow \{\theta_1\}}\,\text{EqC}_1 \quad \bigtriangledown \quad \bigtriangledown}{\langle \texttt{isOk(L P)}, \{\theta_1, \theta_2, \theta_3\}\rangle \rightsquigarrow \{\theta_1\}}\,\text{SubsCond}}{\dfrac{\texttt{\{[1,1]\}} \Rightarrow^{\texttt{e}} \texttt{\{[1,1][1,2]\}}}{\texttt{\{[1,1]\}} \Rightarrow^{top} \texttt{\{[1,1][1,2]\}}}\,\text{Top}}\,\text{Rl}$$

Figure 7: Tree $*_2$ for the applications at the top

The tree $*_4$, shown in Figure 8, is in charge of inferring the set of substitutions obtained when checking the first condition of the rule `expand`, namely `next(L) => P`. The condition is instantiated with the substitution obtained from matching the term with the lefthand side of the rule (in this case $\texttt{L} \mapsto \texttt{[1,1]}$) and, since it is a rewrite condition, the set of reachable terms—computed with $*_5$, which will not be further discussed here—is used to extend this substitution, obtaining a set with three different substitutions (that we previously abbreviated as θ_1, θ_2, and θ_3).

4. Debugging Trees

We describe in this section how to obtain appropriate debugging trees from the proof trees introduced in the previous section. Following the approach shown in [17], we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user *expects* that $\mathcal{I} \models \varphi$ for any judgment φ deduced with respect to the

$$\dfrac{\dfrac{\dfrac{*_5}{\texttt{next([1,1])} \leadsto_2^{\texttt{P}:=\circledast} \{\texttt{[1,2]}, \texttt{[1,0]}, \texttt{[0,1]}\}}\ \mathsf{Tr}_2}{[\texttt{next(L)} \Rightarrow \texttt{P}, \texttt{L} \mapsto \texttt{[1,1]}] \leadsto \{\texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}\}}\ \mathsf{RIC}}{\langle \texttt{next(L)} \Rightarrow \texttt{P}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \leadsto \{\texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]}; \texttt{P} \mapsto \texttt{[0,1]}\}}\ \mathsf{SubsCond}$$

Figure 8: Tree $*_4$ for the first condition of `expand`

rewrite theory $\mathcal{R}$. We will say that a judgment is *valid* when it holds in the intended interpretation $\mathcal{I}$, and *invalid* otherwise. Our goal is to find a buggy node in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking questions to the user about the validity of the nodes in the tree according to the following *top-down* strategy: If all the children of N are valid, then finish pointing out at N as buggy; otherwise, select the subtree rooted by any invalid child and use recursively the same strategy to find the buggy node. Proving that this strategy is complete is straightforward by using induction on the height of T. By using the proof trees computed with the calculus of the previous section as debugging trees we are able to locate wrong statements, missing statements, and wrong search conditions, which are defined as follows:

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$, a membership $l : s$, or a rule $l \Rightarrow r$) and a substitution θ, the *statement instance* $\theta(A) \Leftarrow \theta(C_1) \wedge \cdots \wedge \theta(C_n)$ is *wrong* when all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$ but $\theta(A)$ is not.
- Given a rule $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$ and a term t, the rule has a *wrong instance* if the judgments $[l := t, \emptyset] \leadsto \Theta_0$, $[C_1, \Theta_0] \leadsto \Theta_1$, $\cdots$, $[C_n, \Theta_{n-1}] \leadsto \Theta_n$ are valid in $\mathcal{I}$ but the application of Θ_n to the righthand side does not provide all the results expected for this rule.
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if $[l := t, \emptyset] \leadsto \Theta_0$, $[C_1, \Theta_0] \leadsto \Theta_1$, $\cdots$, $[C_n, \Theta_{n-1}] \leadsto \emptyset$ are valid in $\mathcal{I}$ (meaning that the condition does not hold for t) but the user expected the condition to hold, then we have a *wrong search condition instance.*
- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if there exists a substitution θ such that $\theta(l) \equiv_A t$ and all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$, but the condition is not expected to hold, then we also have a *wrong search condition instance.*
- A statement or condition is *wrong* when it admits a wrong instance.
- Given a term t, there is a *missing equation for* t if the computed normal form of t does not correspond with the one expected in $\mathcal{I}$. A specification has a *missing equation* if there exists a term t such that there is a missing equation for t.
- Given a term t, there is a *missing membership for* t if the computed least sort for t does not correspond with the one expected in $\mathcal{I}$. A specification has a *missing membership* if there exists a term t such that there is a missing membership for t.
- Given a term t, there is a *missing rule for* t if all the rules applied to t at the top lead to judgments $t \Rightarrow^{q_i} S_{q_i}$ valid in $\mathcal{I}$ but the union $\bigcup S_{q_i}$ does not contain all the reachable terms from t by using rewrites at the top. A specification has a *missing rule* if there exists a term t such that there is a missing rule for t.[2]

In our debugging framework, when a wrong statement is found, this specific statement is pointed out; when a missing statement is found, the debugger indicates the operator at the top of the term in the lefthand side of the statement that is missing; and when a wrong condition is found, the specific condition is shown. We will see in the next section that some extra information will be kept in the tree to provide this information. It is important not to confuse missing answers with missing

[2]Actually, what the debugger reports is that a statement is missing *or* the conditions in the remaining statements are not the intended ones (thus they are not applied when expected and another one would be needed), but the error *is not located* in the statements used in the conditions, since they are also checked during the debugging process.

statements; the current calculus detects missing answers due to both wrong and missing statements and wrong search conditions.

4.1. Abbreviated Proof Trees

We will not use proof trees T directly as debugging trees, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT when T is clear from the context. The reason for preferring the APT is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. This transformation relies on the following proposition:

Proposition 4.1. *Let N be a buggy node in some proof tree in the calculus of Figures 1, 2, 3, and 4, w.r.t. an intended interpretation $\mathcal{I}$. Then:*

(1) *N is the consequence of a* Rep$_{\rightarrow}$, Rep$_{\Rightarrow}$, Mb, Rdc$_1$, Norm, Ls, Fulfill, Fail, Top, *or* Rl *inference rule.*

(2) *The error associated to N is a wrong statement, a missing statement, or a wrong search condition.*

To indicate the error associated to the buggy node, we assume that the nodes inferred with these inference rules are decorated with some extra information to identify the error when they are pointed out as buggy. More specifically, nodes related to wrong statements keep the label of the statement, nodes related to missing statements keep the operator at the top that requires more statements to be defined, and nodes related to wrong conditions keep the condition.

The key idea in the APT, whose rules are shown in Figure 9, is to keep the nodes related to the inference rules indicated in Proposition 4.1, making use of the rest of rules to improve the questions asked to the user. The abbreviation always starts by applying ($\mathbf{APT}_1$). This rule simply duplicates the root of the tree and applies APT', which receives a proof tree and returns a forest (i.e., a set of trees). Hence without this duplication the result of the abbreviation could be a forest instead of a single tree. The rest of the APT rules correspond to the function APT' and are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule of least number. The following advantages are obtained:

- Questions associated to nodes with reductions are improved (rules ($\mathbf{APT}_2$), ($\mathbf{APT}_3$), ($\mathbf{APT}_5$), ($\mathbf{APT}_6$), and ($\mathbf{APT}_7$)) by asking about normal forms instead of asking about intermediate states. For example, in rule ($\mathbf{APT}_2$) the error associated to $t \rightarrow t'$ is the one associated to $t \rightarrow t''$, which is not included in the APT. We have chosen to introduce $t \rightarrow t'$ instead of simply $t \rightarrow t''$ in the APT as a pragmatic way of simplifying the structure of the APTs, since t' is obtained from t'' and hence likely simpler.
- The rule ($\mathbf{APT}_4$) deletes questions about rewrites *at the top* (that can be difficult to answer due to matching modulo) and associates the information of those nodes to questions related to the set of reachable terms in one step with rewrites in any position, that are in general easier to answer.
- It creates, with the variants of the rules ($\mathbf{APT}_8$) and ($\mathbf{APT}_9$), two different kinds of tree, one that contains judgments of rewrites with several steps and another that only contains rewrites in one step. The one-step debugging tree follows strictly the idea of keeping only nodes corresponding to relevant information. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rules. The user will choose which debugging tree (one-step or many-steps) will be used for the debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites gives to some parts of the debugging tree the shape of a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to use

$$(\mathbf{APT}_1)\quad APT\left(\dfrac{T_1\dots T_n}{aj}\,{\scriptstyle R_1}\right) = \dfrac{APT'\left(\dfrac{T_1\dots T_n}{aj}\,{\scriptstyle R_1}\right)}{aj}$$

$$(\mathbf{APT}_2)\quad APT'\left(\dfrac{\dfrac{T_1\dots T_n}{t\to t''}\,\textsf{Rep}_{\to}\qquad T'}{t\to t'}\,\textsf{Tr}_{\to}\right) = \left\{\dfrac{APT'(T_1)\dots APT'(T_n)\ APT'(T')}{t\to t'}\,\textsf{Rep}_{\to}\right\}$$

$$(\mathbf{APT}_3)\quad APT'\left(\dfrac{\dfrac{T_1\ \dots\ T_n}{t\to t''}\,\textsf{Rdc}_1\ \ T'}{t\to t'}\,\textsf{NTr}\right) = \left\{\dfrac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T')}{t\to t'}\,\textsf{Rdc}_1\right\}$$

$$(\mathbf{APT}_4)\quad APT'\left(\dfrac{\dfrac{T_1\ \dots\ T_n}{t\Rightarrow^{top} S'}\,\textsf{Top}\ \ T'_1\dots T'_m}{t\Rightarrow_1 S}\,\textsf{Stp}\right) = \left\{\dfrac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T'_1)\ \dots\ APT'(T'_m)}{t\Rightarrow_1 S}\,\textsf{Top}\right\}$$

$$(\mathbf{APT}_5)\quad APT'\left(\dfrac{T'\quad \dfrac{T_1\dots T_n}{t\Rightarrow t'}\,\textsf{Rep}_{\Rightarrow}\quad T''}{t_1\Rightarrow t_2}\,\textsf{EC}\right) = \left\{\dfrac{APT'(T')\ APT'(T_1)\dots APT'(T_n)\ APT'(T'')}{t_1\Rightarrow t_2}\,\textsf{Rep}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}_6)\quad APT'\left(\dfrac{T\quad \dfrac{T_1\dots T_n}{aj'}\,{\scriptstyle R_1}\ \ T'}{aj}\,\textsf{Red}\right) = \left\{\dfrac{APT'(T)\ APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T)}{aj}\,{\scriptstyle R_1}\right\}$$

$$(\mathbf{APT}_7)\quad APT'\left(\dfrac{T_{t\to_{norm}t'}\ T_1\dots T_n}{t:_{ls} s}\,\textsf{Ls}\right) = \left\{\dfrac{APT'(T_{t\to_{norm}t'})\ APT'(T_1)\ \dots\ APT'(T_n)}{t':_{ls} s}\,\textsf{Ls}\right\}$$

$$(\mathbf{APT}^o_8)\quad APT'\left(\dfrac{T_1\quad T_2}{t\Rightarrow t'}\,\textsf{Tr}_{\Rightarrow}\right) = APT'(T_1)\bigcup APT'(T_2)$$

$$(\mathbf{APT}^m_8)\quad APT'\left(\dfrac{T_1\quad T_2}{t\Rightarrow t'}\,\textsf{Tr}_{\Rightarrow}\right) = \left\{\dfrac{APT'(T_1)\ APT'(T_2)}{t\Rightarrow t'}\,\textsf{Tr}_{\Rightarrow}\right\}$$

$$(\mathbf{APT}^o_9)\quad APT'\left(\dfrac{T_1\dots T_n}{aj}\,\textsf{Tr}\right) = APT'(T_1)\bigcup\dots\bigcup APT'(T_n)$$

$$(\mathbf{APT}^m_9)\quad APT'\left(\dfrac{T_1\dots T_n}{aj}\,\textsf{Tr}_i\right) = \left\{\dfrac{APT'(T_1)\ \dots\ APT'(T_n)}{aj}\,\textsf{Tr}_i\right\}$$

$$(\mathbf{APT}_{10})\quad APT'\left(\dfrac{T_1\dots T_n}{aj}\,{\scriptstyle R_2}\right) = \left\{\dfrac{APT'(T_1)\dots APT'(T_n)}{aj}\,{\scriptstyle R_2}\right\}$$

$$(\mathbf{APT}_{11})\quad APT'\left(\dfrac{T_1\dots T_n}{aj}\,{\scriptstyle R_1}\right) = APT'(T_1)\bigcup\dots\bigcup APT'(T_n)$$

R_1 any inference rule R_2 either Mb, Rep$_{\to}$, Rep$_{\Rightarrow}$, Rdc$_1$, Norm, Fulfill, Fail, Ls, Rl, or Top

$1 \leq i \leq 2$ aj, aj' any judgment

Figure 9: Transforming rules for obtaining abbreviated proof trees

efficiently the divide and query navigation strategy. On the contrary, removing the transitivity inferences for rewrites (as rules $(\mathbf{APT}^o_8)$ and $(\mathbf{APT}^o_9)$ do) produces flattened trees where this strategy is no longer efficient. On the other hand, in rewrites $t \Rightarrow t'$ and searches $t \leadsto^{\mathcal{C}}_n S$ appearing as conclusion of a transitivity inference rule, the judgment can be more complicated because it combines several inferences. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \texttt{1} \to_{norm} \texttt{1}}\,\mathsf{Norm}_{\texttt{s\_}}}{(\spadesuit)\ \texttt{[1,1]} \to_{norm} \texttt{[1,1]}}\,\mathsf{Norm}_{\texttt{[\_,\_]}}}{(\spadesuit)\ \texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\,\mathsf{Norm}_{\texttt{\{\_\}}} \quad \dfrac{}{\texttt{isSol}(P_1) \to \texttt{f}}\,\mathsf{Rdc}_{\texttt{is2}} \quad \star_1 \quad \bigtriangledown \quad \dots \quad \bigtriangledown \quad \star_2}{\texttt{\{[1,1]\}} \rightsquigarrow^{\mathcal{C}}_4 \emptyset}\,\mathsf{Tr}_2$$

Figure 10: Abbreviated proof tree for the maze example

- The rule ($\mathbf{APT}_{11}$) removes from the tree all the nodes not associated with relevant information, since the rule ($\mathbf{APT}_{10}$) keeps the relevant information and the rules are applied in order. We remove, for example, nodes related to judgments about sets of substitutions, disabled statements, and rewrites with a concrete rule, that can be in general difficult to answer. Moreover, it removes from the tree trivial judgments like the ones related to reflexivity or congruence.
- Since the APT is built without computing the associated proof tree, it reduces the time and space needed to build the tree.

The following theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and functional modules: the technique will find a buggy node starting from any initial symptom. We assume that the information introduced by the user during the session is correct.

Theorem 4.2. *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 2, 3, and 4 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ s.t. the root of T is invalid in $\mathcal{I}$. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ (correctness).*

The trees in Figures 10–12 depict the (one-step) abbreviated proof tree for the maze example, where $\mathcal{C}$ stands for `{L:List}:=` $\circledast$ $\wedge$ `isSol(L:List)`, P_1 for `[1,1]`, L_1 for `[1,1][1,2]`, L_2 for `[1,1][1,0]`, L_3 for `[1,1][0,1]`, `t` for `true`, `f` for false, `n` for `next`, `e` for `expand`, L for `[1,1][1,2][1,3][1,4]`, and $*_5'$ for the application of APT' to $*_5$. We have also extended the information in the labels with the operator or statement associated to the inference. More concretely, the tree in Figure 10 abbreviates the tree in Figure 5; the first two premises in the abbreviated tree abbreviate the first premise in the proof tree (which includes the tree in Figure 6), keeping only the nodes associated with relevant information according to Proposition 4.1: Norm, with the operator associated to the reduction, and Rdc_1, with the label of the associated equation. The tree $\star_1$, shown in Figure 11, abbreviates the second premise of the tree in Figure 5 as well as the trees in Figures 7 and 8; it only keeps the nodes referring to normal forms, searches in one step, that are now associated to the rule Top, each of them referring to a different operator (the operator `s_` is the successor constructor for natural numbers), and the applications of rules (Rl) and equations ($\mathsf{Rep}_{\to}$). Note that the equation describing the behavior of `isOk` has not got any label, which is indicated with the symbol $\perp$; we will show below how the debugger deals with these nodes. The tree $\star_2$, presented in Figure 12, shares these characteristics and only keeps nodes related to one-step searches and application of rules.

These APT rules are combined with trusting mechanisms that further reduce the proof tree (note that the correctness of these techniques relies on the decisions made by the user):

- Statements can be trusted in several ways: non labelled statements are always trusted (i.e., the nodes marked with ($\diamondsuit$) in Figure 11 will be discarded by the debugger), statements and modules can be trusted before starting the debugging process, and statements can also be trusted on the fly.

$$\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \mathtt{1} \to_{norm} \mathtt{1}}\,\mathsf{Norm}_{\mathtt{s\_}}}{(\spadesuit)\ \mathtt{[1,1]} \to_{norm} \mathtt{[1,1]}}\,\mathsf{Norm}_{\mathtt{[\_,\_]}} \quad \dfrac{*'_5 \quad \dfrac{}{(\Diamond)\ \mathtt{isOk}(L_1) \to \mathtt{t}}\,\mathsf{Rep}_{\perp} \quad \dfrac{}{(\Diamond)\ \mathtt{isOk}(L_2) \to \mathtt{f}}\,\mathsf{Rep}_{\perp} \quad \dfrac{}{(\Diamond)\ \mathtt{isOk}(L_3) \to \mathtt{f}}\,\mathsf{Rep}_{\perp}}{\mathtt{\{[1,1]\}} \Rightarrow^{\mathsf{e}} \mathtt{\{[1,1][1,2]\}}}\,\mathsf{Rl}_{\mathsf{e}} \quad \dfrac{\dfrac{}{(\heartsuit)\ \mathtt{1} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathtt{s\_}}}{(\heartsuit)\ \mathtt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathtt{[\_,\_]}}}{\mathtt{\{[1,1]\}} \Rightarrow_1 \mathtt{\{[1,1][1,2]\}}}\,\mathsf{Top}_{\mathtt{\{\_\}}}$$

Figure 11: Abbreviated tree $\star_1$

$$\dfrac{\dfrac{\dfrac{\dfrac{\triangledown \quad \cdots \quad \triangledown}{\mathtt{n}(L) \Rightarrow^{\mathtt{n1}} \mathtt{[1,5]}}\,\mathsf{Rl}_{\mathtt{n1}} \quad \dfrac{\triangledown \quad \cdots \quad \triangledown}{\mathtt{n}(L) \Rightarrow^{\mathtt{n2}} \mathtt{[0,4]}}\,\mathsf{Rl}_{\mathtt{n2}} \quad \dfrac{\triangledown \quad \cdots \quad \triangledown}{\mathtt{n}(L) \Rightarrow^{\mathtt{n3}} \mathtt{[1,3]}}\,\mathsf{Rl}_{\mathtt{n3}}}{(\ddagger)\ \mathtt{n}(L) \Rightarrow_1 \{\mathtt{[1,5]}, \mathtt{[0,4]}, \mathtt{[1,3]}\}}\,\mathsf{Top}_{\mathtt{n}} \quad \triangledown \quad \cdots \quad \triangledown}{(\dagger)\ \mathtt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow^{\mathsf{e}} \emptyset}\,\mathsf{Rl}_{\mathsf{e}}}{(\dagger)\ \mathtt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathtt{\{\_\}}}$$

Figure 12: Abbreviated tree $\star_2$

- A correct module can be given before starting a debugging session. By checking the correctness of the judgments against this module, correct nodes can be deleted from the tree.
- We consider that constructed terms (terms built only with constructors, pointed out with the `ctor` attribute, and also known as data terms in other contexts) are in normal form and thus inferences of the form $t \to_{norm} t$ with t constructed are removed from the tree. This would remove from the tree the nodes marked with ($\spadesuit$) in Figures 10 and 11.
- Constructed terms of certain sorts or built with some operators can be considered *final*, which indicates that they cannot be further rewritten. For example, we could consider terms of sorts `Nat` and `List` to be final and thus the nodes marked with ($\heartsuit$) in Figure 11 would be removed from the tree.

Once this tree has been built, we can use it to debug the error shown in Section 2. Using the top-down navigation strategy our tool would show all the children of the root and ask the user to select an incorrect one. The last one (the root of $\star_2$) is incorrect and can be selected, and then the user has to answer about the validity of the child of this node. Since it is also incorrect the debugger selects it as current one (the path thus far has been marked with ($\dagger$) in Figure 12) and the debugger shows its children. The first child ($\ddagger$) is erroneous, but this time its children are all correct, so the tool points it out as buggy and it is associated to an erroneous fragment of code. More concretely, the rule used to infer this judgment was Top, and it is associated with the operator `next` (that was abbreviated as `n`), i.e., another rule for this operator is needed. Indeed, if we check the module we notice that the movement to the right has not been specified. We can fix it by adding:

```
rl [n4] : next(L [X,Y]) => [X + 1, Y] .
```

A detailed session of this example is available in the webpage `maude.sip.ucm.es/debugging`.

5. Conclusions and Future Work

We have presented in this paper a debugger of missing answers for Maude specifications. The trees for this kind of debugging are obtained from an abbreviation of a proper calculus whose adequacy for debugging has been proved. This work extends our previous work on wrong and missing answers [17, 18] and provides a powerful and complete debugger for Maude specifications. Moreover, we also provide a graphical user interface that eases the interaction with the debugger and improves its traversal. The tree construction, its navigation, and the user interaction (excluding the GUI) have been all implemented in Maude itself. For more information, see `http://maude.sip.ucm.es/debugging`.

We plan to add new navigation strategies like the ones shown in [21] that take into account the number of different potential errors in the subtrees, instead of their size. Moreover, the current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session. We intend to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. Finally, we also plan to create a test generator to test Maude specifications and debug the erroneous test with the debugger.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming (WCFLP'05), Tallinn, Estonia*, pages 8–13. ACM Press, 2005.

[4] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In F. J. López-Fraguas, editor, *15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.

[5] R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative diagnosis of missing answers in constraint functional-logic programming. In J. Garrigue and M. V. Hermenegildo, editors, *Proceedings of 9th International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2008.

[6] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[7] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990.

[8] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proceedings of the 9th International Conference on Rewriting Techniques and Applications (RTA 98)*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998.

[9] I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.

[10] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[11] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[12] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.

[13] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[14] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[15] B. Pope. *A Declarative Debugger for Haskell*. PhD thesis, The University of Melbourne, Australia, 2006.

[16] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009. `http://maude.sip.ucm.es/debugging`.

[17] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of rewriting logic specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques (WADT 2008)*, volume 5486 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2009.

[18] A. Riesco, A. Verdejo, and N. Martí-Oliet. Enhancing the debugging of Maude specifications. In *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications (WRLA 2010)*, Lecture Notes in Computer Science, 2010. To appear.

[19] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. Declarative debugging of rewriting logic specifications. Technical Report SIC 02/10, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2010. `http://maude.sip.ucm.es/debugging`.

[20] E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1983.

[21] J. Silva. A comparative study of algorithmic debugging strategies. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[22] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.

A Complete Declarative Debugger for Maude[⋆]

Adrián Riesco, Alberto Verdejo, and Narciso Martí-Oliet

Facultad de Informática, Universidad Complutense de Madrid, Spain
ariesco@fdi.ucm.es, {alberto,narciso}@sip.ucm.es

Abstract. We present a declarative debugger for Maude specifications that allows to debug wrong answers (a *wrong* result is obtained) and missing answers (a correct but *incomplete* result is obtained) due to both wrong and missing statements and wrong search conditions. The debugger builds a tree representing the computation and guides the user through it to find the bug. We present the debugger's latest commands and features, illustrating its use with several examples.

Keywords: Declarative debugging, Maude, missing answers, wrong answers.

1 Introduction

Declarative debugging [8] is a semi-automatic debugging technique that focuses on results, which makes it specially suited for declarative languages, whose operational details may be hard to follow. Declarative debuggers represent the computation as a tree, called *debugging tree*, where each node must be logically inferred from the results in its children. In our case, these trees are obtained by abbreviating proof trees obtained in a formal calculus [4,5]. Debugging progresses by asking questions related to the nodes of this tree (i.e., questions related to subcomputations of the wrong result being debugged) to an external oracle (usually the user), discarding nodes in function of the answers until a *buggy node*—a node with an erroneous result and with all its children correct—is located. Since each node in the tree has associated a piece of code, when this node is found, the bug, either a wrong or a missing statement,[1] is also found.

Maude [2] is a declarative language based on both equational and rewriting logic for the specification and implementation of a whole range of models and systems. Functional modules define data types and operations on them by means of *membership equational logic* theories that support multiple sorts, subsort relations, equations, and assertions of membership in a sort, while system modules specify rewrite theories that also support *rules*, defining local concurrent transitions that can take place in a system. As a programming language, a distinguishing feature of Maude is its use of reflection, that allows many metaprogramming applications. Moreover, the debugger is implemented on top of Full Maude [2, Chap. 18], a tool completely written in Maude which

[⋆] Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

[1] It is important not to confuse wrong and missing answers with wrong and missing statements. The former are the initial symptoms that indicate the specifications fails, while the latter is the error that generated this misbehavior.

includes features for parsing and pretty-printing terms, improving the input/output interaction. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

We extend here the tool presentation in [7], based on [1,4], for the debugging of wrong answers (wrong results, which correspond in our case to erroneous reductions, sort inferences, and rewrites) with the debugging of missing answers (incomplete results, which correspond here to not completely reduced normal forms, greater than expected least sorts, and incomplete sets of reachable terms), showing how the theory introduced in [5,6] is applied. The reasons the debugger is able to attribute to these errors are wrong and missing statements and, since missing answers in system modules are usually found with the `search` command [2, Chap. 6] that performs a reachability analysis, wrong search conditions.

With this extension we are able to present a state-of-the-art debugger, with several options to build, prune, and traverse the debugging tree. Following the classification in [9], these are the main characteristics of our debugger. Although we have not implemented all the possible strategies to shorten and navigate the debugging tree, like the latest tree compression technique or the answers "maybe yes," "maybe not," and "inadmissible," our trees are abbreviated (in our case, since we obtain the trees from a formal calculus, we are able to prove the correctness and completeness of the technique), statements can be trusted in several ways, a correct module can be used as oracle, `undo` and `don't know` commands are provided, the tree can be traversed with different strategies, and a graphical interface is available. Furthermore, we have developed a new technique to build the debugging tree: before starting the debugging process, the user can choose between different debugging trees, one that leads to shorter sessions (from the point of view of the number of questions) but with more complex questions, and another one that presents longer, although easier, sessions. We have successfully applied this approach to both wrong and missing answers.

Complete explanations about our debugger, including a user guide [3] that describes the graphical user interface, the source files for the debugger, examples, and related papers, are available in the webpage `http://maude.sip.ucm.es/debugging`.

2 Using the Debugger

We make explicit first what is assumed about the modules introduced by the user; then we present the new available commands.

Assumptions. A rewrite theory has an underlying equational theory, containing equations and memberships, which is expected to satisfy the appropriate executability requirements, namely, it has to be terminating, confluent, and sort decreasing. Rules are assumed to be coherent with respect to the equations; for details, see [2].

The tool allows to shorten the debugging trees in several ways: statements and complete modules can be trusted, a correct module can be used as oracle, constructed terms (terms built only with constructors, indicated with the attribute `ctor`) are considered to be in normal form, and constructed terms of some sorts or built with some operators can be pointed out as *final* (they cannot be further rewritten). This information, as well as the answers given during the debugging process and the signature of the module, is assumed to be correct.

Commands. The debugger is started by loading the file `dd.maude`, which starts an input/output loop that allows the user to interact with the tool. Since the debugger is implemented on top of Full Maude, all modules and commands must be introduced enclosed in parentheses. Debugging of missing answers uses all the features already described for wrong answers: use of a correct module as oracle, trusting of statements and modules, and different types of debugging tree; see [3,7] for details about the corresponding commands.

When debugging missing answers we can select some sorts as final (i.e., they cannot be further rewritten) with `(final [de]select SORTS .)`, that only works once the final mode has been activated with `(set final select on/off .)`.

When debugging missing answers in system modules, two different trees can be built: one whose questions are related to one-step searches and another one whose questions are related to many-steps searches; the user can switch between these trees with the commands `(one-step missing-tree .)`, which is the default one, and `(many-steps missing-tree .)`, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions but with likely more complicated questions.

The user can prioritize questions related to the fulfillment of the search condition from questions involving the statements defining it with `(solutions prioritized on/off .)`.

The debugging tree can be navigated by using two different strategies: the more intuitive *top-down strategy*, that traverses the tree from the root asking each time for the correctness of all the children of the current node, and then continues with one of the incorrect children; and the more efficient *divide and query strategy*, that each time selects the node whose subtree's size is the closest one to half the size of the whole tree, discarding the whole subtree if the inference in this node is correct and continuing the process with this subtree in other case. The user can switch between them with the commands `(top-down strategy .)` and `(divide-query strategy .)`, being divide and query the default strategy.

The debugging process is started with:

```
(missing [in MODULE-NAME :] INIT-TERM -> NF .)
(missing [in MODULE-NAME :] INIT-TERM : LS .)
(missing [[depth]] [in MODULE-NAME :] INIT-TERM ARROW PATTERN [s.t. COND] .)
```

The first command debugs normal forms, where `INIT-TERM` is the initial term and `NF` is the obtained unexpected normal form. Similarly, the second command starts the debugging of incorrect least sorts, where `LS` is the computed least sort. The last command refers to incomplete sets found when using search, where `depth` indicates the bound in the number of steps, which is considered unbounded when omitted; `ARROW` is `=>*` for searches in zero or more steps, `=>+` for searches in one or more steps, and `=>!` for final terms; and `COND` is the optional condition to be fulfilled by the results. Finally, when no module name is specified in a command, the default one is used.

When the divide and query strategy is selected, one question, that can be either correct or wrong (w.r.t. the intended behavior the user has in mind), is presented in each step. The different answers are transmitted with the commands `(yes .)` and `(no .)`. Instead of just answering `yes`, we can also *trust* some statements on the fly if, once the process has started, we decide the bug is not there. To trust the current statement we type the command `(trust .)`. If a question refers to a set of reachable terms and one of these terms is not reachable, the user can point it out with the answer `(I is wrong .)` where `I` is the index of the wrong term in the set; in case the question is related to a set of reachable *solutions*, if one of the terms should not be a solution the user can indicate it with `(I is not a solution .)`. Information about final terms can also be given on the fly with `(its sort is final .)`, which indicates that the least sort of the term currently displayed is final. If the current question is too complicated, it can be skipped with the command `(don't know .)`, although this answer can, in general, introduce incompleteness in the debugging process. When the top-down strategy is used, several questions will be displayed in each step. The answer in this case is transmitted with `(N : ANSWER)`, where `N` is the number of the question and `ANSWER` the answer the user would give in the divide and query strategy. As a shortcut to answer `yes` to all nodes, the tool also provides the answer `(all : yes .)`. Finally, we can return to the previous state by using the command `(undo .)`.

Graphical User Interface. The graphical user interface allows the user to visualize the debugging tree and navigate it with more freedom. More advanced users can take advantage of visualizing the whole tree to select the questions that optimize the debugging process (keeping in mind that the user is looking for incorrect nodes with all its children correct, he can, for example, search for erroneous nodes with few children or even erroneous leaves), while average users can just follow the implemented strategies and answer the questions in a friendlier way. Moreover, the interface eases the trusting of statements by providing information about the statements in each module and the subsort relations between sorts; provides three different navigation strategies (top-down, divide and query, and free); and implements two different modes to modify the tree once the user introduces an answer: in the *prune* mode only the minimum amount of relevant information is depicted in each step, while in the *keep* mode the complete tree is kept through the whole debugging process, coloring the nodes depending on the information given by the user. The first mode centers on the debugging process, while the second one allows the user to see how different answers modify the tree.

3 Debugging Sessions

We illustrate how to use the debugger with two examples, the first one shows how to debug an erroneous normal form due to a missing statement and the second one how to debug an incomplete set of reachable terms due to a wrong statement. Complete details about both examples are available in the webpage.

Example 1. We want to implement a function that, given an initial city (built with the operator `city`, that receives a natural number as argument), a number of cities, and a cost graph (i.e., a partial function from pairs of cities to natural numbers indicating the cost of traveling between cities), returns a tour around all the cities that finishes in the initial one. First, we specify a priority queue where the states of this function will be kept:

```
sorts Node PNodeQueue .
subsort Node < PNodeQueue .

op node : Path Nat -> Node [ctor] .
op mtPQueue : -> PNodeQueue [ctor] .
op _ _ : PNodeQueue PNodeQueue -> PNodeQueue [ctor assoc id: mtPQueue] .
```

where a `Path` is just a list of cities. We use this priority queue to implement the function `travel` in charge of computing this tour:

```
op travel : City Nat Graph -> TravelResult .
ceq travel(C, N, G) = travel(C, N, G, R, node(C, 0))
 if R := greedyTravel(C, N, G) .
```

This function uses an auxiliary function `travel` that, in addition to the parameters above, receives a potential best result, created with the operator `result` and computed with the greedy function `greedyTravel`, and the initial priority queue, that only contains the node `node(C, 0)`. This auxiliary function is specified with the equations:

```
ceq [tr1] : travel(C, N, G, R, ND PQ) = travel(C, N, G, R, PQ')
 if not isResult(ND, N) /\ N' := getCost(R) /\ N'' := getCost(ND) /\
    N'' < N' /\ PQ' := expand(ND, N, G, PQ) .

ceq [tr2] : travel(C, N, G, R, node(P C', N') PQ) =
                                  travel(C, N, G, result(P C' C, UB), PQ)
 if isResult(node(P C', N'), N) /\ N'' := getCost(R) /\
    UB := N' + (G [road(C, C')]) /\ UB < N'' .

ceq [tr3] : travel(C, N, G, R, node(P C', N') PQ) = travel(C, N, G, R, PQ)
 if isResult(node(P C', N'), N) /\ N'' := getCost(R) /\
    UB := N' + (G [road(C, C')]) /\ UB >= N'' .

ceq [tr4] : travel(C, N, G, R, ND PQ) = R
 if N' := getCost(R) /\ N'' := getCost(ND)  /\ N'' >= N' .
```

However, we forget to specify the equation that returns the result when the queue is empty:

```
eq [tr5] : travel(C, N, G, R, emptyPQueue) = R .
```

Without this equation, Maude is not able to compute the desired normal form in an example for 4 cities. To debug it, we first consider that `GTRAVELER`, the module defining the greedy algorithm, is correct and can be trusted:

```
Maude> (set debug select on .)
Debug select is on.

Maude> (debug exclude GTRAVELER .)
Labels cheap1 cheap2 cheap3 gc1 gc2 gc3 gt1 gt2 in1 in2 in3 mi1 mi2 sz1
       sz2 are now trusted.
```

The command indicates that all the statements in the GTRAVELER module, whose labels are shown, are now trusted (unlabeled statements are trusted by default). Now, we debug a wrong reduction with the following command, where the term on the left of the arrow is the initial term we tried to reduce, the term on the right is the obtained result, and G abbreviates the cost graph:

```
(missing travel(city(0), 3, generateCostMatrix(3)) -> travel(city(0),3,G,
      result(city(0)city(3)city(1)city(2)city(0),15),emptyPQueue) .)
```

With this command the debugger builds the debugging tree, that is navigated with the default divide and query strategy. The first question is:

```
Is travel(city(0), 3, G, result(city(0) city(3) city(1) city(2) city(0), 15),
emptyPQueue) in normal form?
Maude> (no .)
```

Since we expected travel to be reduced to a result, this term is not in normal form. The next question is:

```
Is PNodeQueue the least sort of emptyPQueue ?
Maude> (yes .)
```

In fact, emptyPQueue—the empty priority queue—has as least sort PNodeQueue, the sort for priority queues. With this information the debugger locates the error, indicating that either another equation is required for the operator travel or that the conditions in the current equations are wrong:

```
The buggy node is:
travel(city(0), 3, G, result(city(0) city(3) city(1) city(2) city(0), 15),
emptyPQueue) is in normal form.
Either the operator travel needs more equations or the conditions of the
current equations are not written in the intended way.
```

When a missing statement is detected, the debugger indicates that either a new statement is needed or the conditions can be changed to allow more matchings (e.g., the user can change N > 2 by N >= 2). Note that the error may be located in the conditions but *not* in the statements defining them, since they are also checked during the debugging process.

Example 2. Assume now we have specified the district in charge of a firefighters brigade. Buildings are represented by their street, avenue (New York alike coordinates), time to collapse (all of them natural numbers), and status, that can be ok or fire. When the status is fire the time to collapse is reduced until it reaches 0, when the fire cannot be extinguished, and the fire can be propagated to the nearest buildings:

```
  sorts Building Neighborhood Status FFDistrict .
  subsort Building < Neighborhood .

  ops ok fire : -> Status [ctor] .
  op < av :_, st :_, tc :_, sts :_> : Nat Nat Nat Status -> Building [ctor] .
```

The neighborhood is built with the empty neighborhood and the juxtaposition operator:

```
  op empty : -> Neighborhood [ctor] .
  op _ _ : Neighborhood Neighborhood -> Neighborhood
                                       [ctor assoc comm id: empty] .
```

Finally, the district contains the buildings in the neighborhood and two natural numbers indicating the avenue and the street where the firefighters are currently located:

```
  op [_]_ _ : Neighborhood Nat Nat -> FFDistrict [ctor] .
```

The firefighters travel with the rule:

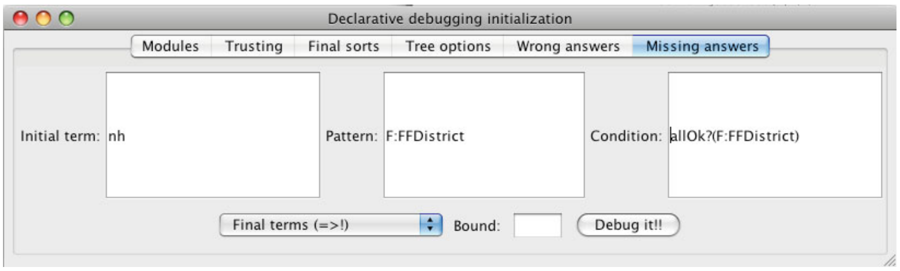
```
  crl [go] : [NH] Av1 St1
   => [NH''] Av2 St2
   if extinguished?(NH, Av1, St1) /\
      B NH' := NH /\
      fire?(B) /\
      Av2 := getAvenue(B) /\
      St2 := getStreet(B) /\
      N := distance(Av1, St1, Av2, St2) /\
      NH'' := update(NH, N) .
```

where the conditions check that the building in the current location is not in fire with `extinguished?`, search for a building in fire `B` with the matching condition and the condition `fire?`, extract the avenue `Av2` and the street `St2` of this building with `getAvenue` and `getStreet`, compute the distance between the current and the new location with `distance`, and finally update the neighborhood (the time to collapse of the buildings is reduced an amount equal to the distance previously computed) with `update`. We are interested in the equational condition `fire?(B)`, because the equation `f2` specifying `fire?`, a function that checks whether the status of a building is `fire`, is buggy and returns `false` when it should return `true`:

```
  op fire? : Building -> Bool .
  eq [f1] : fire?(< av : Av, st : St, tc : TC, sts : ok >) = false .
  eq [f2] : fire?(< av : Av, st : St, tc : TC, sts : fire >) = false .
```

If we use the command `search` to find the final states where the fire in both buildings has been extinguished, in an example with two buildings on fire with time to collapse 4 located in $(1,2)$ and $(2,1)$, and the firefighters initially in $(0,0)$, we realize that no states fulfilling the condition are found:

```
Maude> (search nh =>! F:FFDistrict s.t. allOk?(F:FFDistrict) .)
search in TEST : nh =>! F:FFDistrict .
No solution.
```



Initial term: nh

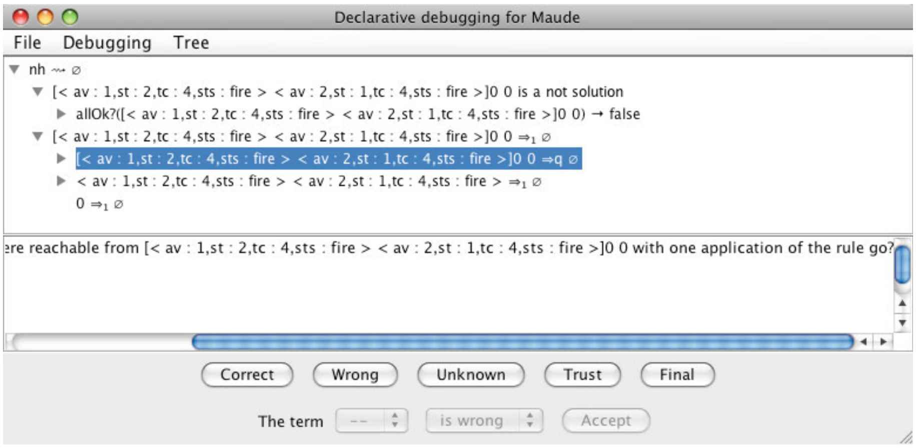
Pattern: F:FFDistrict

Condition: allOk?(F:FFDistrict)

Final terms (=>!)

Bound:

Fig. 1. Initial command for the firefighters example



```
▼ nh ⇝ ∅
  ▼ [< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >]0 0 is a not solution
    ▶ allOk?([< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >]0 0) → false
  ▼ [< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >]0 0 ⇒1 ∅
    ▶ [< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >]0 0 ⇒q ∅
    ▶ < av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire > ⇒1 ∅
      0 ⇒1 ∅
```

```
ere reachable from [< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >]0 0 with one application of the rule go?
```

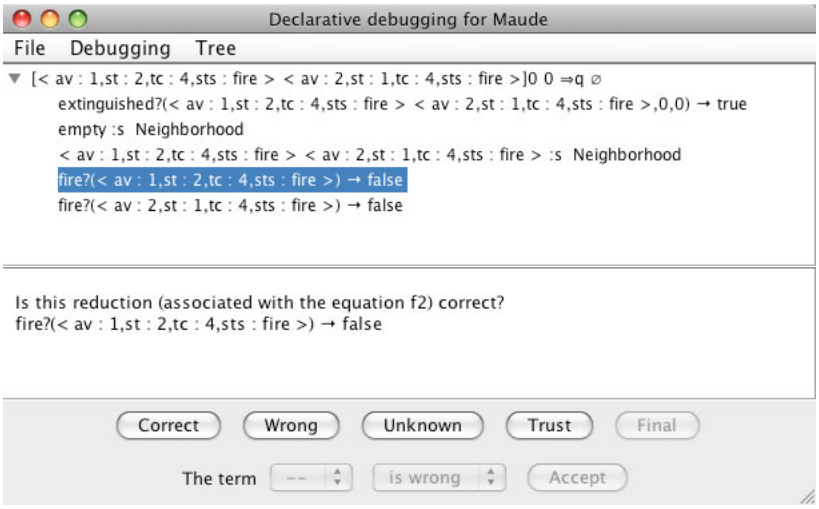
The term -- is wrong

Fig. 2. Debugging tree for the firefighters example

where `nh` is a constant with value `[< av : 1, st : 2, tc : 4, sts : fire > < av : 2, st : 1, tc : 4, sts : fire >] 0 0`. We can debug this behavior with the command shown in Figure 1, that builds the default one-step tree. We show in Figure 2 how the corresponding tree, with only two levels expanded, is depicted by the graphical user interface.

Although this graphical interface has both the divide and query and the top-down navigation strategies implemented, advanced users can find more useful the free navigation strategy, that can greatly reduce the number of questions asked to the user. To achieve this, the user must find either correct nodes (which are removed from the tree) rooting big subtrees or wrong nodes (which are selected as current root) rooting small trees.

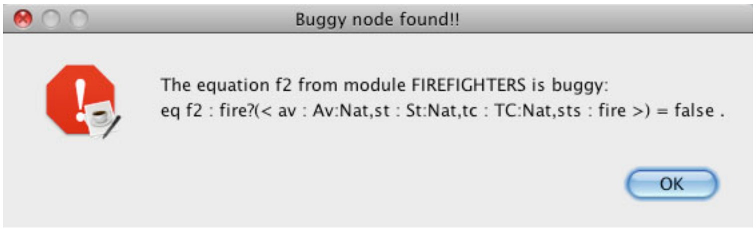
In the tree depicted in Figure 2 we notice that Maude cannot apply the rule `go` (the rule is shown once the node is selected) to a configuration with some buildings with status `fire`, which is indicated by the interface by showing that the term is rewritten to the empty set, $\emptyset$. Since this is incorrect, we can use the button `Wrong` to indicate it and the tree in Figure 3 is shown. Note that this node is associated to a concrete rule



```
▼ [< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >]0 0 ⇒q ∅
    extinguished?(< av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire >,0,0) → true
    empty :s Neighborhood
    < av : 1,st : 2,tc : 4,sts : fire > < av : 2,st : 1,tc : 4,sts : fire > :s Neighborhood
    fire?(< av : 1,st : 2,tc : 4,sts : fire >) → false
    fire?(< av : 2,st : 1,tc : 4,sts : fire >) → false

Is this reduction (associated with the equation f2) correct?
fire?(< av : 1,st : 2,tc : 4,sts : fire >) → false
```

Fig. 3. Debugging tree for the firefighters example after one answer



```
The equation f2 from module FIREFIGHTERS is buggy:
eq f2 : fire?(< av : Av:Nat,st : St:Nat,tc : TC:Nat,sts : fire >) = false .
```

Fig. 4. Bug found in the firefighters example

and then the debugger allows the user to use the command `Trust`, that would remove all the nodes associated to this same rule from the debugging tree; trusting the most frequent statements is another strategy that can easily be followed by the user when using the graphical interface. However, since the question is incorrect, we cannot use this command to answer it.

In this case, the first three nodes are correct (the notation `_:s_` appearing in the second and third nodes indicates that the term has this sort as least sort), but we can select any of the other nodes as erroneous; the interested reader will find that both of them lead to the same error. In our case, we select the fourth node as erroneous and the debugger shows the error, as shown in Figure 4.

4 Conclusions

We have implemented a declarative debugger of wrong and missing answers for Maude, that is able to detect wrong and missing statements and wrong search conditions. Since one of the main drawbacks of declarative debugging is the size of the debugging trees and the complexity of the questions, we provide several mechanisms to shorten and ease the tree, including a graphical user interface. As future work, we plan to implement a test case generator, that integrated with the debugger will allow to test Maude specifications and debug the erroneous cases.

References

1. Caballero, R., Martí-Oliet, N., Riesco, A., Verdejo, A.: A declarative debugger for Maude functional modules. In: Roşu, G. (ed.) Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008. ENTCS, vol. 238(3), pp. 63–81. Elsevier, Amsterdam (2009)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
3. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid (2009), `http://maude.sip.ucm.es/debugging`
4. Riesco, A., Verdejo, A., Caballero, R., Martí-Oliet, N.: Declarative debugging of rewriting logic specifications. In: Corradini, A., Montanari, U. (eds.) WADT 2008. LNCS, vol. 5486, pp. 308–325. Springer, Heidelberg (2009)
5. Riesco, A., Verdejo, A., Martí-Oliet, N.: Declarative debugging of missing answers for Maude specifications. In: Lynch, C. (ed.) Proceedings of the 21st International Conference on Rewriting Techniques and Applications (RTA 2010). Leibniz International Proceedings in Informatics, vol. 6, pp. 277–294. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik (2010)
6. Riesco, A., Verdejo, A., Martí-Oliet, N.: Enhancing the debugging of Maude specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 226–242. Springer, Heidelberg (2010)
7. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: A declarative debugger for Maude. In: Meseguer, J., Roşu, G. (eds.) AMAST 2008. LNCS, vol. 5140, pp. 116–121. Springer, Heidelberg (2008)
8. Shapiro, E.Y.: Algorithmic Program Debugging. In: ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
9. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)

Declarative debugging of rewriting logic specifications[☆]

Adrián Riesco[*], Alberto Verdejo, Narciso Martí-Oliet, Rafael Caballero

Facultad de Informática, Universidad Complutense de Madrid
Dpto. Sistemas Informáticos y Computación

Abstract

Declarative debugging is a semi-automatic technique that starts from an incorrect computation and locates a program fragment responsible for the error by building a tree representing this computation and guiding the user through it to find the error. Membership equational logic (*MEL*) is an equational logic that in addition to equations allows to state membership axioms characterizing the elements of a sort. Rewriting logic is a logic of change that extends *MEL* by adding rewrite rules, which correspond to transitions between states and can be nondeterministic. We propose here a calculus to infer reductions, sort inferences, normal forms, and least sorts with the equational subset of rewriting logic, and rewrites and sets of reachable terms through rules. We use an abbreviation of the proof trees computed with this calculus to build appropriate debugging trees for both wrong (an incorrect result obtained from an initial result) and missing answers (results that are erroneous because they are incomplete), whose adequacy for debugging is proved. Using these trees we have implemented Maude DDebugger, a declarative debugger for Maude, a high-performance system based on rewriting logic. We illustrate its use with an example.

Keywords: declarative debugging, rewriting logic, Maude, wrong answers, missing answers

1. Introduction

Declarative debugging [35], also known as declarative diagnosis or algorithmic debugging, is a debugging technique that abstracts the execution details, which may be difficult to follow in declarative languages, to focus on the results. We can distinguish between two different kinds of declarative debugging: debugging of *wrong answers*, applied when a *wrong* result is obtained from an initial value, which has been widely employed in the logic [38, 18], functional [26, 28], multi-paradigm [5, 20], and object-oriented [6] programming languages; and debugging of *missing answers* [23, 38, 18, 10, 1], applied when a result is *incomplete*, which has been less studied because the calculus involved is more complex than in the case of wrong answers. Declarative debugging starts from an incorrect computation, the error symptom, and locates the code (or the absence of code) responsible for the error. To find this error the debugger represents the computation as a *debugging tree* [24], where each node stands for a computation step and must follow from the results of its child nodes by some logical inference. This tree is traversed by asking questions to an external oracle (generally the user) until a *buggy node*—a node containing an erroneous result, but whose children are all correct—is found. Hence, we distinguish two phases in this scheme: the debugging tree *generation* and its *navigation* following some suitable strategy [36].

[☆]Research supported by MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

[*]Corresponding author. Address: Facultad de Informática (Universidad Complutense de Madrid). C/ Prof. José García Santesmases s/n, 28040 Madrid (Spain). email: ariesco@fdi.ucm.es. Tel: (+34)913947648. Fax: (+34)913947529.

Email addresses: `ariesco@fdi.ucm.es` (Adrián Riesco), `alberto@sip.ucm.es` (Alberto Verdejo), `narciso@sip.ucm.es` (Narciso Martí-Oliet), `rafa@sip.ucm.es` (Rafael Caballero)

We present here Maude DDebugger, a declarative debugger for *Maude specifications*. Maude [12] is a high-level language and high-performance system supporting both equational and rewriting logic computation. Maude modules correspond to specifications in *rewriting logic* [21], a logic that allows the representation of many models of concurrent and distributed systems. This logic is an extension of *membership equational logic* [2], an equational logic that, in addition to equations, allows to state *membership axioms* characterizing the elements of a sort. Rewriting logic extends membership equational logic by adding rewrite rules, which represent transitions in a concurrent system and can be nondeterministic. The Maude system supports several approaches for debugging Maude programs: tracing, term coloring, and using an internal debugger [12, Chap. 22]. The tracing facilities allow us to follow the execution of a specification, that is, the sequence of applications of statements that take place. The same ideas have been applied to the functional paradigm by the tracer *Hat* [11], where a graph constructed by graph rewriting is proposed as a suitable trace structure. Term coloring uses different colors to print the operators used to build a term that does not fully reduce. Finally, the Maude internal debugger allows the definition of break points in the execution by selecting some operators or statements. When a break point is found the debugger is entered, where we can see the current term and execute the next rewrite with tracing turned on. However, these tools have the disadvantage that, since they are based on the trace, show the statements applied in the order in which they are executed, and thus the user can lose the general view of the proof of the incorrect computation that produced the wrong result.

Declarative debugging of wrong answers of membership equational logic specifications was studied in [8, 7], and was later extended to debugging of wrong answers in rewriting logic specifications in [30], while descriptions of the implemented system can be found in [34], where we present how to debug wrong results due to errors in the statements of the specification. In [32] we investigated how to apply declarative debugging of missing answers, traditionally associated with nondeterministic frameworks [10, 23], to membership equational logic specifications. We achieve this by broadening the concept of missing answers to deal with erroneous normal forms and least sorts. Finally, we extended the calculus developed thus far in [31] to debug missing answers in rewriting logic specifications, that is, expected results that the specification is not able to compute. A description of the whole system is presented in [33].

One of the strong points of our approach is that, unlike other proposals like [10], it combines the treatment of wrong and missing answers and thus it is able to detect missing answers due to both wrong and missing statements. The state of the art can be found in [36], which contains a comparison among the algorithmic debuggers B.i.O. [3] (Believe in Oracles), a debugger integrated in the Curry compiler KICS; Buddha [27, 28], a debugger for Haskell 98; DDT [9], a debugger for TOY; Freja [26], a debugger for Haskell; Hat-Delta [14], part of a set of tools to debug Haskell programs; Mercury's Algorithmic Debugger [20], a debugger integrated into the Mercury compiler; Münster Curry Debugger [19], a debugger integrated into the Münster Curry compiler; and Nude [25], the NU-Prolog Debugging Environment. We extend this comparison by taking into account the features in the latest updates of the debuggers and adding two new ones: DDJ [16], a debugger for Java programs, and our own debugger, Maude DDebugger. This comparison is summarized in Tables 1 and 2, where each column shows a declarative debugger and each row a feature. More specifically:

- The implementation language indicates the language used to implement the debugger. In some cases front- and back-ends are shown: they refer, respectively, to the language used to obtain the information needed to compute the debugging tree and the language used to interact with the user.

- The target language states the language debugged by the tool.

- The strategies row indicates the different navigation strategies implemented by the debuggers. TD stands for *top-down*, that starts from the root and selects a wrong child to continue with the navigation until all the children are correct; DQ for *divide and query*, that selects in each case a node rooting a subtree half the size of the whole tree; SS for *single stepping*, that performs a post-order traversal of the execution tree; HF for *heaviest first*, a modification of top-down that selects the child with the biggest subtree; MRF for *more rules first*, another variant of top-down that selects the child with the biggest number of different statements in its subtree; DRQ for *divide by rules and query*, an improvement of divide and query that selects the node whose subtree has half the number of associated statements of the whole tree; MD for the divide and query strategy implemented by the Mercury Debugger; SD for *subterm dependency*, a strategy that allows to track specific subterms that the user has pointed out as erroneous; and HD for the Hat-Delta heuristics.

- Database indicates whether the tool keeps a database of answers to be used in future debugging sessions, while memoization indicates whether this database is available for the current session.
- The front-end indicates whether it is integrated into the compiler or it is standalone.
- Interface shows the interface between the front-end and the back-end. Here, APT stands for the Abbreviated Proof Tree generated by Maude; ART for Augmented Redex Trail, the tree generated by Hat-Delta; ET is an abbreviation of Execution Tree; and step count refers to a specific method of the B.i.O. debugger that keeps the information used thus far into a text file.
- Debugging tree presents how the debugging trees are managed.
- The missing answers row indicates whether the tool can debug missing answers.
- Accepted answers: the different answers that can be introduced into the debugger. `yes`; `no`; `dk` (*don't know*); `tr` (*trust*); `in` (*inadmissible*), used to indicate that some arguments should not have been computed; and `my` and `mn` (*maybe yes* and *maybe no*), that behave as yes and no although the questions can be repeated if needed. More details about these debugging techniques can be found in [36, 37].
- Tracing subexpressions means that the user is able to point out a subterm as erroneous.
- ET exploration indicates whether the debugging tree can be freely traversed.
- Whether the debugging tree can be built following different strategies depending on the specific situation is shown in the Different trees? row.
- Tree compression indicates whether the tool implements tree compression [14], a technique to remove redundant nodes from the execution tree.
- Undo states whether the tool provides an undo command.
- Trusting lists the trusting options provided by each debugger. MO stands for trusting modules; FU for functions (statements); AR for arguments; and FN for final forms.
- GUI shows whether the tool provides a graphical user interface.
- Version displays the version of the tool used for the comparison.

The results shown in these tables can be interpreted as follows:

Navigation strategies. Several navigation strategies have been proposed for declarative debugging [36]. However, most of the debuggers (including Maude DDebugger) only implement the basic top-down and divide and query techniques. On the other hand, DDJ implements most of the known navigation techniques (some of them also developed by the same researchers), including an adaptation of the navigation techniques developed for Hat-Delta. Among the basic techniques, only DDJ, DDT, and Maude DDebugger provide the most efficient divide and query strategy, Hirunkitti's divide and query [36].

Available answers. The declarative debugging scheme relies on an external oracle answering the questions asked by the tool, and thus the bigger the set of available answers the easier the interaction. The minimum set of answers accepted by all the debuggers is composed of the answers *yes* and *no*; Hat-Delta, the Münster Curry Debugger, and Nude do not accept any more answers, but the remaining debuggers allow some others. Other well-known answers are *don't know* and *trust*; the former, that can introduce incompleteness, allows the user to skip the current question and is implemented by

	Maude DDebugger	**B.i.O.**	**Buddha**	**DDJ**	**DDT**	**Freja**
Implementation language	Maude	Curry	Haskell	Java	Toy (front-end) Java (back-end)	Haskell
Target language	Maude	Curry	Haskell	Java	Toy	Haskell subset
Strategies	TD DQ	TD	TD	TD DQ DRQ SS HF MRF HD	TD DQ	TD
Database / Memoization?	NO/YES	NO/NO	NO/YES	YES/YES	NO/YES	NO/NO
Front-end	Independent prog. trans.	Independent prog. trans.	Independent prog. trans.	Independent prog. trans.	Integrated prog. trans.	Integrated compiler
Interface	APT on demand	Step count	ET	ET on demand	ET (XML/TXT)	ET
Debugging tree	Main memory on demand	Main memory on demand	Main memory on demand	Database	Main memory	Main memory
Missing answers?	YES	NO	NO	NO	YES	NO
Accepted answers	`yes no dk tr`	`yes no dk`	`yes no dk in tr`	`yes no dk tr`	`yes no dk tr`	`yes no my mn`
Tracing subexpressions?	NO	NO	NO	NO	NO	NO
ET exploration?	YES	YES	YES	YES	YES	YES
Different trees?	YES	NO	NO	YES	NO	NO
Tree compression?	NO	NO	NO	NO	NO	NO
Undo?	YES	YES	NO	YES	NO	YES
Trusting	MO/FU/FN	MO/FU/AR	MO/FU	FU	FU	MO/FU
GUI?	YES	NO	NO	YES	YES	NO
Version	2.0 (24/5/2010)	Kics 0.81893 (15/4/2009)	1.2.1 (1/12/2006)	2.4 (23/10/2010)	1.2 (29/9/2005)	(2000)

Table 1: A comparative of declarative debuggers I

	Maude DDebugger	**Hat-Delta**	**Mercury Debugger**	**Münster Curry Debugger**	**Nude**
Implementation language	Maude	Haskell	Mercury	Haskell (front-end) Curry (back-end)	Prolog
Target language	Maude	Haskell	Mercury	Curry	Prolog
Strategies	TD DQ	HD	TD DQ SD MD	TD	TD
Database / Memoization?	NO/YES	NO/YES	NO/YES	NO/NO	YES/YES
Front-end	Independent prog. trans.	Independent prog. trans.	Independent compiler	Integrated compiler	Independent compiler
Interface	APT	ART (native)	ET on demand	ET	ET on demand
Debugging tree	Main memory on demand	File system	Main memory on demand	Main memory	Main memory on demand
Missing answers?	YES	NO	NO	NO	NO
Accepted answers	`yes no dk tr`	`yes no`	`yes no dk in tr`	`yes no`	`yes no`
Tracing subexpressions?	NO	NO	YES	NO	NO
ET exploration?	YES	YES	YES	YES	NO
Different trees?	YES	NO	NO	NO	NO
Tree compression?	NO	YES	NO	NO	NO
Undo?	YES	NO	YES	NO	NO
Trusting	MO/FU/FN	MO	MO/FU	MO/FU	FU
GUI?	YES	NO	NO	YES	NO
Version	2.0 (24/5/2010)	2.05 (22/10/2006)	Mercury 0.13.1 (1/12/2006)	AquaCurry 1.0 (13/5/2006)	NU-Prolog 1.7.2 (13/7/2004)

Table 2: A comparative of declarative debuggers II

B.i.O., DDJ, DDT, Buddha, Mercury, and Maude DDebugger, while the latter prevents the debugger from asking questions related to the current statement and is accepted by DDJ, DDT, Buddha, the Mercury debugger, and Maude DDebugger. Buddha and the Mercury debugger have developed an answer *inadmissible* to indicate that some arguments should not have been computed, redirecting the debugging process in this direction; our debugger accepts a similar mechanism when debugging missing answers in system modules with the answer *the term n is not a solution/reachable*, which indicates that a term in a set is not a solution/reachable, leading the process in this direction. Finally, Freja accepts the answers *maybe yes* and *maybe not*, that the debugger uses as *yes* and *not*, although it will return to these questions if the bug is not found.

Database. A common feature in declarative debugging is the use of a database to prevent the tool from asking the same question twice, which is implemented by DDJ, DDT, Hat-Delta, Buddha, the Mercury debugger, Nude, and Maude DDebugger. Nude has improved this technique by allowing this database to be used during the next sessions, which has also been adopted by DDJ.

Memory. The debuggers allocate the debugging tree in different ways. The Hat-Delta tree is stored in the file system, DDJ uses a database, and the rest of the debuggers (including ours) keep it in main memory. Most debuggers improve memory management by building the tree on demand, as B.i.O., Buddha, DDJ, the Mercury debugger, Nude, and Maude DDebugger.

Tracing subexpressions. The Mercury debugger is the only one able to indicate that a specific subexpression, and not the whole term, is wrong, improving both the answers *no* and *inadmissible* with precise information about the subexpression. With this technique the navigation strategy can focus on some nodes of the tree, enhancing the debugging process.

Construction strategies. A novelty of our approach is the possibility of building different trees depending on the complexity of the specification and the experience of the user: the trees for both wrong and missing answers can be built following either a one-step or a many-step strategy (giving rise to four combinations). While with the one-step strategy the tool asks more questions in general, these questions are easier to answer than the ones presented with the many-steps strategy. An improvement of this technique has been applied in DDJ in [17], allowing the system to balance the debugging trees by combining so called *chains*, that is, sequences of statements where the final data of each step is the initial data of the following one.

Tree compression. The Hat-Delta debugger has developed a new technique to remove redundant nodes from the execution tree, called tree compression [14]. Roughly speaking, it consists in removing (in some cases) from the debugging tree the children of nodes that are related to the same error as the father, in such a way that the father will provide debugging information for both itself and these children. This technique is very similar to the balancing technique implemented for DDJ in [17].

Tree exploration. Most of the debuggers allow the user to freely navigate the debugging tree, including ours when using the graphical user interface. Only the Münster Curry Debugger and Nude do not implement this feature.

Trusting. Although all the debuggers provide some trusting mechanisms, they differ on the target: all the debuggers except Hat-Delta have mechanisms to trust specific statements, and all the debuggers except DDJ, DDT, and Nude can trust complete modules. An original approach is to allow the user to trust some arguments, which currently is only supported by B.i.O. In our case, and since we are able to debug missing answers, a novel trusting mechanism has been developed: the user can identify some sorts and some operators as *final*, that is, they cannot be further reduced; with this method all nodes referring to "finalness" of these terms are removed from the debugging tree. Finally, a method similar to trusting consists in using a correct specification as an oracle to answer the questions; this approach is followed by B.i.O. and Maude DDebugger.

Undo command. In a technique that relies on the user as oracle, it is usual to commit an error and thus an undo command can be very useful. However, not all the debuggers have this command, with B.i.O., DDJ, Freja, the Mercury debugger, and Maude DDebugger being the only ones implementing this feature.

Graphical interface. A graphical user interface eases the interaction between the user and the tool, allowing him to freely navigate the debugging tree and showing all the features in a friendly way. In [36], only one declarative debugger—DDT—implemented such an interface, while nowadays four tools—DDT, DDJ, Münster Curry Debugger,[1] and Maude DDebugger—have this feature.

Errors detected. It is worth noticing that only DDT and Maude DDebugger can debug missing answers, while all the other debuggers are devoted exclusively to wrong answers. However, DDT only debugs missing answers due to nondeterminism, while our approach uses this technique to debug erroneous normal forms and least sorts.

Other remarks. An important subject in declarative debugging is scalability. The development of DDJ has taken special care of this subject by using a complex architecture that manages the available memory and uses a database to store the parts of the tree that do not fit in main memory. Moreover, the navigation strategies have been modified to work with incomplete trees. Regarding reusability, the latest version of B.i.O. provides a generic interface that allows other tools implementing it to use its debugging features. Finally, the DDT debugger has been improved to deal with constraints.

Exploiting the fact that rewriting logic is *reflective* [13], a key distinguishing feature of Maude is its systematic and efficient use of reflection through its predefined `META-LEVEL` module [12, Chap. 14], a feature that makes Maude remarkably extensible and powerful, and that allows many advanced metaprogramming and metalanguage applications. This powerful feature allows access to metalevel entities such as specifications or computations as usual data. Therefore, we are able to generate and navigate the debugging tree of a Maude computation using operations in Maude itself. In addition, the Maude system provides another module, `LOOP-MODE` [12, Chap. 17], which can be used to specify input/output interactions with the user. However, instead of using this module directly, we extend Full Maude [12, Chap. 18], which includes features for parsing, evaluating, and pretty-printing terms, improving the input/output interaction. Moreover, Full Maude allows the specification of concurrent object-oriented systems, which can also be debugged. Thus, our declarative debugger, including its user interactions, is implemented in Maude itself.

The rest of the paper is structured as follows. Section 2 presents the preliminaries of our debugging approach. Section 3 describes our calculus while the next section explains how to transform the proof trees built with this calculus into appropriate debugging trees. Section 5 shows how to use the debugger, while Section 6 illustrates it with an example. Section 7 describes the implementation of our tool and Section 8 concludes and presents some future work. We present in Appendix A the detailed proofs of the results stated throughout the paper.

Additional examples, the source code of the tool, and other papers on the subject, including the user guide [29], where the graphical user interface for the debugger is presented, are all available from the webpage `http://maude.sip.ucm.es/debugging`.

2. Preliminaries

In the following sections we present both membership equational logic and rewriting logic, and how their specifications are represented as Maude modules. Then, we state the assumptions made on those specifications.

[1]Only available for Mac OS X.

2.1. Membership equational logic

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1\ldots k_n,k}\}_{(k_1\ldots k_n,k)\in K^*\times K}$ a many-kinded signature, and $S = \{S_k\}_{k\in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of membership equational logic are *equations* $t = t'$, where t and t' are Σ-terms of the same kind, and *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \cdots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ.

Models of membership equational logic specifications are Σ-*algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1\ldots k_n,k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. Given a Σ-algebra $\mathcal{A}$ and a valuation $\sigma : X \longrightarrow \mathcal{A}$ mapping variables to values in the algebra, the meaning $[\![t]\!]^\sigma_{\mathcal{A}}$ of a term t is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies, under a valuation σ,

- an equation $t = t'$, denoted $\mathcal{A}, \sigma \models t = t'$, if and only if both terms have the same meaning: $[\![t]\!]^\sigma_{\mathcal{A}} = [\![t']\!]^\sigma_{\mathcal{A}}$; we also say that the equation holds in the algebra under the valuation.
- a membership $t : s$, denoted $\mathcal{A}, \sigma \models t : s$, if and only if $[\![t]\!]^\sigma_{\mathcal{A}} \in A_s$.

Satisfaction of Horn clauses is defined in the standard way. Finally, when terms are ground, valuations play no role and thus can be omitted. A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of ground terms $[t]_E$, and where an equation or membership is satisfied if and only if it can be deduced from E by means of a sound and complete set of deduction rules [2, 22].

Since the membership equational logic specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness [12], their equations $t = t'$ can be oriented from left to right, $t \to t'$. Such a statement holds in an algebra, denoted $\mathcal{A}, \sigma \models t \to t'$, exactly when $\mathcal{A}, \sigma \models t = t'$, i.e., when $[\![t]\!]^\sigma_{\mathcal{A}} = [\![t']\!]^\sigma_{\mathcal{A}}$. Moreover, under those assumptions an equational condition $u = v$ in a conditional equation can be checked by finding a common term t such that $u \to t$ and $v \to t$; the notation we will use in the inference rules and debugging trees studied in Section 3 for this situation is $u \downarrow v$. Also, the notation $t =_E t'$ means that the equation $t = t'$ can be deduced from E, equivalently, that $[t]_E = [t']_E$.

2.2. Maude functional modules

Maude functional modules [12, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational logic specifications and their semantics is given by the corresponding initial algebra in the class of algebras satisfying the specification.

In a functional module we can declare sorts (by means of the keyword `sorts`); *subsort* relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`). Conditions, in addition to memberships and equations, can also be *matching equations* $t := t'$, whose mathematical meaning is the same as that of an ordinary equation $t = t'$ but that operationally are solved by matching the righthand side t' against the pattern t in the lefthand side, thus instantiating possibly new variables in t.

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat]` = `[Nat]`.

An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

2.3. Rewriting logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible.

A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses membership equational logic, as described in the previous sections. A rule q in R has the general conditional form[2]

$$q : \ (\forall X)\ e \Rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$$

where q is the rule label, the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [21] (see also [4]).

Models of rewrite theories are called $\mathcal{R}$*-systems* in [21]. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. The rewriting logic deduction rules introduced in [21] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial models. Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, the initial model $\mathcal{T}_{\Sigma/E,R}$ for $\mathcal{R}$ has an underlying (Σ, E)-algebra $\mathcal{T}_{\Sigma/E}$ whose elements are equivalence classes $[t]_E$ of ground Σ-terms modulo E, and there is a transition from $[t]_E$ to $[t']_E$ when there exist terms t_1 and t_2 such that $t =_E t_1 \rightarrow_R^* t_2 =_E t'$, where $t_1 \rightarrow_R^* t_2$ means that the term t_1 can be rewritten into t_2 in zero or more rewrite steps applying rules in R, also denoted $[t]_E \rightarrow_{R/E}^* [t']_E$ when rewriting is considered on equivalence classes [21, 15].

However, for our purposes in this paper, we are interested in a subclass of rewriting logic models [21] that we call *term models*, where the syntactic structure of terms is kept and associated notions such as variables, substitutions, and term rewriting make sense. These models will be used in the next section to represent

[2]There is no need for the condition to list equations first, then memberships, and then rewrites; this is just a notational abbreviation, since they can be listed in any order.

the *intended interpretation* that the user had in mind while writing a specification. Since we want to find the discrepancies between the intended model and the initial model of the specification as written, we need to consider the relationship between a specification defined by a set of equations E and a set of rules R, and a model defined by possibly different sets of equations E' and of rules R'; in particular, when $E' = E$ and $R' = R$, the term model coincides with the initial model built in [21].

Given a rewrite theory $\mathcal{R} = (\Sigma, E, R)$, with Σ a signature, E a set of equations, and R a set of rules, a Σ-term model has an underlying (Σ, E')-algebra whose elements are equivalence classes $[t]_{E'}$ of ground Σ-terms modulo some set of equations and memberships E' (which may be different from E), and there is a transition from $[t]_{E'}$ to $[t']_{E'}$ when $[t]_{E'} \rightarrow^*_{R'/E'} [t']_{E'}$, where rewriting is considered on equivalence classes [21, 15]. The set of rules R' may also be different from R, that is, the term model is $\mathcal{T}_{\Sigma/E',R'}$ for some E' and R'. In such term models, the notion of valuation coincides with that of (ground) substitution. A term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies, under a substitution θ,

- an equation $u = v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u = v$, when $\theta(u) =_{E'} \theta(v)$, or equivalently, when $[\theta(u)]_{E'} = [\theta(v)]_{E'}$;
- a membership $u : s$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u : s$, when the Σ-term $\theta(u)$ has sort s according to the information in the signature Σ and the equations and memberships E';
- a rewrite $u \Rightarrow v$, denoted $\mathcal{T}_{\Sigma/E',R'}, \theta \models u \Rightarrow v$, when there is a transition in $\mathcal{T}_{\Sigma/E',R'}$ from $[\theta(u)]_{E'}$ to $[\theta(v)]_{E'}$, that is, when $[\theta(u)]_{E'} \rightarrow^*_{R'/E'} [\theta(v)]_{E'}$.

Satisfaction is extended to conditional sentences as usual. A Σ-term model $\mathcal{T}_{\Sigma/E',R'}$ satisfies a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ when $\mathcal{T}_{\Sigma/E',R'}$ satisfies the equations and memberships in E and the rewrite rules in R in this sense. For example, this is obviously the case when $E \subseteq E'$ and $R \subseteq R'$; as mentioned above, when $E' = E$ and $R' = R$ the term model coincides with the initial model for $\mathcal{R}$.

2.4. Maude system modules

Maude system modules [12, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`), whose conditions can be equations, matching equations, memberships, and rewrites.

The executability requirements for equations and memberships in a system module are the same as those of functional modules, namely, confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [12] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [39, 12]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rule in R to a term u' such that $u' =_E t'$.

The following section describes an example of a Maude system module with both equations and rules.

2.5. An example of system module: A maze

Given a maze, we want to obtain all the possible paths to the exit. First, we define the sorts `Pos`, `Pos?`, `List`, and `State`, that stand for positions in the labyrinth, incorrect positions (that we will use later to indicate that terms with this sort must be rewritten to become a correct position) lists of positions, and the path traversed so far, respectively:

```
(mod MAZE is
  pr NAT .
  sorts Pos Pos? List State .
```

Terms of sort `Pos` have the form `[X,Y]`, where `X` and `Y` are natural numbers, and lists are built with `nil` and the juxtaposition operator `__`:

```
  subsorts Pos < Pos? List .

  op [_,_] : Nat Nat -> Pos [ctor] .
  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .
```

Terms of sort `State` are lists enclosed by curly brackets, that is, `{_}` is an "encapsulation operator" that ensures that the whole state is used:

```
  op {_} : List -> State [ctor] .
```

The predicate `isSol` checks whether a list is a solution in a 8×8 labyrinth:

```
  vars X Y : Nat .
  vars P Q : Pos .
  var L : List .
  op isSol : List -> Bool .
  eq [is1] : isSol(L [8,8]) = true .
  eq [is2] : isSol(L) = false [owise] .
```

The next position is computed with rule `expand`, that extends the solution with a new position by rewriting `next(L)` to obtain a new position and then checking whether this list is correct with `isOk`. Note that the choice of the next position, that could be initially wrong, produces an implicit backtracking:

```
  crl [expand] : { L } => { L P } if next(L) => P /\ isOk(L P) .
```

The function `next`, that builds terms of the sort `Pos?`, is defined in a nondeterministic way with the rules:

```
  op next : List -> Pos? .
  rl [n1] : next(L [X,Y]) => [X, Y + 1] .
  rl [n2] : next(L [X,Y]) => [sd(X, 1), Y] .
  rl [n3] : next(L [X,Y]) => [X, sd(Y, 1)] .
```

where `sd` denotes symmetric difference on natural numbers.

`isOk(L P)` checks that the position `P` is within the limits of the labyrinth, not repeated in `L`, and not part of the wall by using an auxiliary function `contains`:

```
  op isOk : List -> Bool .
  eq isOk(L [X,Y]) = X >= 1 and Y >= 1 and X <= 8 and Y <= 8
           and not(contains(L, [X,Y])) and not(contains(wall, [X,Y])) .
  op contains : List Pos -> Bool .
  eq [c1] : contains(nil, P) = false .
  eq [c2] : contains(Q L, P) = if P == Q then true else contains(L, P) fi .
```

Finally, we define the `wall` of the labyrinth as a list of positions:

```
  op wall : -> List .
  eq wall =       [2,1]       [4,1]
                  [2,2] [3,2]             [6,2] [7,2]
                  [2,3]       [4,3] [5,3] [6,3] [7,3]

            [1,5] [2,5] [3,5] [4,5] [5,5] [6,5]       [8,5]
                                          [6,6]       [8,6]
                                          [6,7]
                                          [6,8] [7,8]       .
endm)
```

Now, we can use the module to search the labyrinth's exit from the position `[1,1]` with the Maude command `search`, but it cannot find any path to escape. We will see in Section 5 how to debug this specification.

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
No solution.
```

2.6. Assumptions

Since we are debugging Maude modules, they are expected to satisfy the appropriate executability requirements indicated in the previous sections. Namely, the specifications in functional modules have to be terminating, confluent, sort decreasing and, given an equation $t_1 = t_2$ *if* $C_1 \wedge \cdots \wedge C_n$, all the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated by matching [12, Section 4.6]. While the equational part of system modules has to fulfill these requirements, rewrite rules must be coherent with respect to the equations and, given a rule $t_1 \Rightarrow t_2$ *if* $C_1 \wedge \cdots \wedge C_n$, the variables occurring in t_2 and $C_1 \ldots C_n$ must appear in t_1 or become instantiated in matching or rewriting conditions [12, Section 6.3].

One interesting feature of our tool is that the user can trust some statements, by means of labels applied to the suspicious statements. This means that the unlabeled statements are assumed to be correct, and only their conditions will generate questions. In order to obtain a nonempty abbreviated proof tree, the user must have labeled some statements (all with different labels); otherwise, everything is assumed to be correct. In particular, the wrong statement must be labeled in order to be found. Likewise, when debugging missing answers, constructed terms (terms built only with constructors, indicated with the attribute `ctor`, and also known as data terms in other contexts) are considered to be in normal form, and some of these constructed terms can be pointed out as "final" (they cannot be further rewritten). Thus, this information has to be accurate in order to find the buggy node.

Although the user can introduce a module importing other modules, the debugging process takes place in the *flattened* module. However, the debugger allows the user to trust a whole imported module.

Navigation of the debugging tree takes place by asking questions to an external oracle, which in our case is either the user or another module introduced by the user. In both cases the answers are assumed to be correct. If either the module is not really correct or the user provides an incorrect answer, the result is unpredictable. Notice that the information provided by the correct module need not be complete, in the sense that some functions can be only partially defined. In the same way, it is not required to use the same signature in the correct and the debugged modules. If the correct module cannot help in answering a question, the user may have to answer it.

Finally, all the information in the signature (sorts, subsorts, operators, and equational attributes such as assoc, comm, etc.) is supposed to be correct and will not be considered during the debugging process.

3. A calculus for debugging

Now we will describe debugging trees for both wrong and missing answers. First, Section 3.1 presents a calculus to deduce reductions, memberships, and rewrites. We will extend this calculus in Section 3.2 to describe a calculus to compute normal forms, least sorts, and sets of reachable terms. From now on, we assume a rewrite theory $\mathcal{R} = (\Sigma, E, R)$ satisfying the assumptions stated in the previous section.

(**Reflexivity**)

$$\frac{}{e \Rightarrow e}\,\mathsf{Rf}_{\Rightarrow} \qquad \frac{}{e \rightarrow e}\,\mathsf{Rf}_{\rightarrow}$$

(**Transitivity**)

$$\frac{e_1 \Rightarrow e' \qquad e' \Rightarrow e_2}{e_1 \Rightarrow e_2}\,\mathsf{Tr}_{\Rightarrow} \qquad \frac{e_1 \rightarrow e' \qquad e' \rightarrow e_2}{e_1 \rightarrow e_2}\,\mathsf{Tr}_{\rightarrow}$$

(**Congruence**)

$$\frac{e_1 \Rightarrow e'_1 \quad \ldots \quad e_n \Rightarrow e'_n}{f(e_1, \ldots, e_n) \Rightarrow f(e'_1, \ldots, e'_n)}\,\mathsf{Cong}_{\Rightarrow} \qquad \frac{e_1 \rightarrow e'_1 \quad \ldots \quad e_n \rightarrow e'_n}{f(e_1, \ldots, e_n) \rightarrow f(e'_1, \ldots, e'_n)}\,\mathsf{Cong}_{\rightarrow}$$

(**Replacement**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \ \{\theta(v_j) : s_j\}_{j=1}^{m} \ \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\theta(e) \Rightarrow \theta(e')}\,\mathsf{Rep}_{\Rightarrow}$$
if $e \Rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(e) \rightarrow \theta(e')}\,\mathsf{Rep}_{\rightarrow}$$
if $e \rightarrow e' \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$

(**Equivalence Class**) (**Subject Reduction**)

$$\frac{e \rightarrow e' \quad e' \Rightarrow e'' \quad e'' \rightarrow e'''}{e \Rightarrow e'''}\,\mathsf{EC} \qquad \frac{e \rightarrow e' \quad e' : s}{e : s}\,\mathsf{SRed}$$

(**Membership**)

$$\frac{\{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(e) : s}\,\mathsf{Mb}$$
if $e : s \Leftarrow \bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j$

Figure 1: Semantic calculus for Maude modules

3.1. A calculus for wrong answers

We show here a calculus to deduce judgments for reductions $e \rightarrow e'$, memberships $e : s$, and rewrites $e \Rightarrow e'$. The inference rules for this calculus, shown in Figure 1, are an adaptation of the rules presented in [2, 22] for membership equational logic and in [21, 4] for rewriting logic. Remember that the notation $\theta(u_i) \downarrow \theta(u'_i)$ is an abbreviation of $\exists t_i.\theta(u_i) \rightarrow t_i \wedge \theta(u'_i) \rightarrow t_i$. As usual, we represent deductions in the calculus as *proof trees*, where the premises are the child nodes of the conclusion at each inference step. We assume that the inference labels $\mathsf{Rep}_{\Rightarrow}$, $\mathsf{Rep}_{\rightarrow}$, and Mb decorating the inference steps contain information about the particular rewrite rule, equation, and membership axiom, respectively, applied during the inference. This information will be used by the debugger in order to present to the user the incorrect fragment of code causing the error.

For example, we can try to build the proof tree for the following reduction:

```
Maude> (red isOk([1,1][1,2]) .)
result Bool : true
```

Figures 2 and 3 depict the proof tree associated to this reduction, where `c` stands for `contains`, `t`

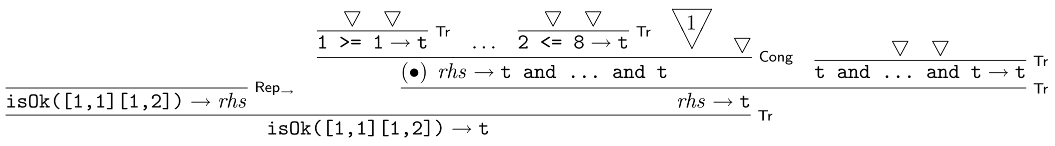


Figure 2: Tree for the reduction of `isOk([1,1][1,2])`

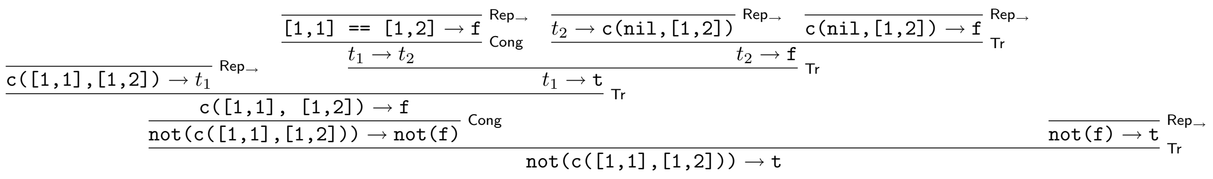


Figure 3: Tree $\overset{1}{\bigtriangledown}$ for `not(c([1,1],[1,2]))`

for `true`, `f` for `false`, *rhs* for `1 >= 1 and 2 >= 1 and 1 <= 8 and 2 <= 8 and not(c([1,1],[1,2])) and not(c(wall,[1,2]))`, t_1 for `if [1,1] == [1,2] then t else c(nil,[1,2]) fi`, t_2 for `if f then t else c(nil,[1,2]) fi`, and each $\bigtriangledown$ abbreviates a computation not shown here. In order to obtain the result we use the transitivity inference rule, whose left premise applies the replacement rule with the equation for `isOk`, obtaining the term *rhs*, that will be further reduced in the right premise to obtain `t` by means of another transitivity step. The left child of this last node reduces all the subterms in *rhs* to `t`, while the right one just applies the usual equations for conjunctions to obtain the final result. While the first reductions in the premises of the node (•) correspond to arithmetic computations and will not been shown here, the last two are more complex. Figure 3 describes the tree $\overset{1}{\bigtriangledown}$, that proves how one of the subterms using equations defined by the user is reduced to `t`, while the tree on its right is very similar and will not be studied in depth. The tree $\overset{1}{\bigtriangledown}$ reduces in its left child the inner subterm to `f` by traversing the list of positions (in this case the only element in the list is `[1,1]`), reducing the `if_then_else_fi` term in t_1 and then applying the equation for the empty list `nil`. Then, the right child of the root applies the predefined equation for `not` to obtain the final result.

In our debugging framework we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. Therefore the user expects that $\mathcal{I} \models e \Rightarrow e'$, $\mathcal{I} \models e \to e'$, and $\mathcal{I} \models e : s$ for each rewrite $e \Rightarrow e'$, reduction $e \to e'$, and membership $e : s$ computed w.r.t. the specification $\mathcal{R}$. As a term model, $\mathcal{I}$ must satisfy the following proposition:

Proposition 1. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let $\mathcal{T} = \mathcal{T}_{\Sigma/E',R'}$ be any Σ-term model. If a statement $e \Rightarrow e'$ (respectively $e \to e'$, $e : s$) can be deduced using the semantic calculus rules* reflexivity, transitivity, congruence, equivalence class, *or* subject reduction *using premises that hold in $\mathcal{T}$, then $\mathcal{T} \models e \Rightarrow e'$ (respectively $\mathcal{T} \models e \to e'$, $\mathcal{T} \models e : s$).*

Observe that this proposition cannot be extended to the *membership* and *replacement* inference rules, where the correctness of the conclusion depends not only on the calculus but also on the associated specification statement, which could be wrong.

3.2. A calculus for missing answers

The calculus in this section, that extends the one shown in the previous section, will be used to infer the normal form of a term, the least sort of a term, and, given a term and some constraints, the *complete* set of reachable terms from this term that fulfill the requirements.[3] The proof trees built with this calculus have

[3]The requirements of this last inference mimic the ones used in the Maude's breadth-first search, which is usually used to detect the existence of missing answers.

nodes that justify the positive information (why the normal form is reached, the least sort is obtained, and the terms are included in the corresponding sets) but also nodes that justify the negative information (why the normal form is no further reduced, why no smaller sort can be obtained for the term, and why there are no more terms in the sets). These latter nodes are then used in the debugging trees to localize as much as possible the reasons responsible for missing answers. Throughout this paper we only consider a special kind of conditions and substitutions that operate over them, called *admissible*, that we define as follows:

Definition 1. *A condition $\mathcal{C} \equiv C_1 \wedge \cdots \wedge C_n$ is* admissible *if, for $1 \leq i \leq n$,*

- *C_i is an equation $u_i = u_i'$ or a membership $u_i : s$ and*

$$vars(C_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$

- *C_i is a matching condition $u_i := u_i'$, u_i is a pattern and*

$$vars(u_i') \subseteq \bigcup_{j=1}^{i-1} vars(C_j), \text{ or}$$

- *C_i is a rewrite condition $u_i \Rightarrow u_i'$, u_i' is a pattern and*

$$vars(u_i) \subseteq \bigcup_{j=1}^{i-1} vars(C_j).$$

Definition 2. *A condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, where $\circledast$ denotes a special variable not occurring in the rest of the condition, is* admissible *if $P := t \wedge C_1 \wedge \cdots \wedge C_n$ is admissible for t any ground term.*

Definition 3. *A* kind-substitution*, denoted by κ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} . kind(v_i) = kind(t_i)$, that is, each variable has the same kind as the associated term.*

Definition 4. *A* substitution*, denoted by θ, is a mapping from variables to terms of the form $v_1 \mapsto t_1; \ldots; v_n \mapsto t_n$ such that $\forall_{1 \leq i \leq n} . sort(v_i) \geq ls(t_i)$, that is, the sort of each variable is greater than or equal to the least sort of the associated term. Note that a substitution is a special type of kind-substitution where each term has the sort appropriate to its variable.*

Definition 5. *Given an atomic condition C, we say that a substitution θ is* admissible for C *if*

- *C is an equation $u = u'$ or a membership $u : s$ and $vars(C) \subseteq dom(\theta)$, or*
- *C is a matching condition $u := u'$ and $vars(u') \subseteq dom(\theta)$, or*
- *C is a rewrite condition $u \Rightarrow u'$ and $vars(u) \subseteq dom(\theta)$.*

The calculus presented in this section (in Figures 4–7, and 12) will be used to deduce the following judgments, that we introduce together with their meaning for a Σ-term model $\mathcal{T}' = \mathcal{T}_{\Sigma/E',R'}$ defined by equations and memberships E' and by rules R':

- Given a term t and a kind-substitution κ, $\mathcal{T}' \models adequateSorts(\kappa) \rightsquigarrow \Theta$ when either $\Theta = \{\kappa\}$ and $\forall v \in dom(\kappa).\mathcal{T}' \models \kappa[v] : sort(v)$, or $\Theta = \emptyset$ and $\exists v \in dom(\kappa).\mathcal{T}' \not\models \kappa[v] : sort(v)$, where $\kappa[v]$ denotes the term bound by v in κ. That is, when all the terms bound in the kind-substitution κ have the appropriate sort, then κ is a substitution and it is returned; otherwise (at least one of the terms has an incorrect sort), the kind-substitution is not a substitution and the empty set is returned.[4]

[4]Do not confuse, in the judgments inferring sets of substitutions, the empty set of substitutions $\emptyset$, which indicates that no substitutions fulfill the condition, with the set containing the empty substitution $\{\emptyset\}$, which indicates that the condition is fulfilled and the condition is ground.

- Given an admissible substitution θ for an atomic condition C, $\mathcal{T}' \models [C, \theta] \rightsquigarrow \Theta$ when

$$\Theta = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \upharpoonright_{dom(\theta)} = \theta\},$$

  that is, Θ is the set of substitutions that fulfill the atomic condition C and extend θ by binding the new variables appearing in C.

- Given a set of admissible substitutions Θ for an atomic condition C, $\mathcal{T}' \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$ when

$$\Theta' = \{\theta' \mid \mathcal{T}', \theta' \models C \text{ and } \theta' \upharpoonright_{dom(\theta)} = \theta \text{ for some } \theta \in \Theta\},$$

  that is, Θ' is the set of substitutions that fulfill the condition C and extend any of the admissible substitutions in Θ.

- $\mathcal{T}' \models \mathit{disabled}(a, t)$ when the equation or membership a cannot be applied to t at the top.

- $\mathcal{T}' \models t \rightarrow_{red} t'$ when either $\mathcal{T}' \models t \rightarrow^1_{E'} t'$ or $\mathcal{T}' \models t_i \rightarrow^!_{E'} t'_i$, with $t_i \neq t'_i$, for some subterm t_i of t such that $t' = t[t_i \mapsto t'_i]$, that is, the term t is either reduced one step at the top or reduced by substituting a subterm by its normal form.

- $\mathcal{T}' \models t \rightarrow_{norm} t'$ when $\mathcal{T}' \models t \rightarrow^!_{E'} t'$, that is, t' is in normal form with respect to the equations E'.

- Given an admissible condition $\mathcal{C} \equiv P := \circledast \wedge C_1 \wedge \cdots \wedge C_n$, $\mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t)$ when there exists a substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ holds when $\circledast$ is substituted by t.

- Given an admissible condition $\mathcal{C}$ as before, $\mathcal{T}' \models \mathit{fails}(\mathcal{C}, t)$ when there exists *no* substitution θ such that $\mathcal{T}', \theta \models P := t \wedge C_1 \wedge \cdots \wedge C_n$, that is, $\mathcal{C}$ does not hold when $\circledast$ is substituted by t.

- $\mathcal{T}' \models t :_{ls} s$ when $\mathcal{T}' \models t : s$ and moreover s is the least sort with this property (with respect to the ordering on sorts obtained from the signature Σ and the equations and memberships E' defining the Σ-term model $\mathcal{T}'$).

- $\mathcal{T}' \models t \Rightarrow^{top} S$ when $S = \{t' \mid t \rightarrow^{top}_{R'} t'\}$, that is, the set S is formed by all the reachable terms from t by exactly one rewrite *at the top* with the rules R' defining $\mathcal{T}'$. Moreover, equality in S is modulo E', i.e., we are implicitly working with equivalence classes of ground terms modulo E'.

- $\mathcal{T}' \models t \Rightarrow^{q} S$ when $S = \{t' \mid t \rightarrow^{top}_{\{q\}} t'\}$, that is, the set S is the complete set of reachable terms (modulo E') obtained from t with one application of the rule $q \in R'$ at the top.

- $\mathcal{T}' \models t \Rightarrow_1 S$ when $S = \{t' \mid t \rightarrow^1_{R'} t'\}$, that is, the set S is constituted by all the reachable terms (modulo E') from t in exactly one step, where the rewrite step can take place anywhere in t.

- $\mathcal{T}' \models t \rightsquigarrow^{\mathcal{C}}_n S$ when $S = \{t' \mid t \rightarrow^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t')\}$, that is, S is the set of all the terms (modulo E') that satisfy the admissible condition $\mathcal{C}$ and are reachable from t in at most n steps.

- $\mathcal{T}' \models t \rightsquigarrow+^{\mathcal{C}}_n S$ as before, but with reachability from t in at least one step and in at most n steps.

- $\mathcal{T}' \models t \rightsquigarrow!^{\mathcal{C}}_n S$ when $S = \{t' \mid t \rightarrow^{\leq n}_{R'} t' \text{ and } \mathcal{T}' \models \mathit{fulfilled}(\mathcal{C}, t') \text{ and } t' \not\rightarrow_{R'}\}$, that is, now the terms (modulo E') in S are *final*, meaning that they cannot be further rewritten.

We first introduce in Figure 4 the inference rules defining the relations $[C, \theta] \rightsquigarrow \Theta$, $\langle C, \Theta\rangle \rightsquigarrow \Theta'$, and $\mathit{adequateSorts}(\kappa) \rightsquigarrow \Theta$. Intuitively, these judgments will provide positive information when they lead to nonempty sets (indicating that the condition holds in the first two judgments or that the kind-substitution is a substitution in the third one) and negative information when they lead to the empty set (indicating respectively that the condition fails or the kind-substitution is not a substitution):

$$\frac{\theta(t_2) \to_{norm} t' \quad adequateSorts(\kappa_1) \rightsquigarrow \Theta_1 \quad \ldots \quad adequateSorts(\kappa_n) \rightsquigarrow \Theta_n}{[t_1 := t_2, \theta] \rightsquigarrow \bigcup_{i=1}^{n} \Theta_i} \ \mathsf{PatC}$$
if $\{\kappa_1, \ldots, \kappa_n\} = \{\kappa\theta \mid \kappa(\theta(t_1)) \equiv_A t'\}$

$$\frac{t_1 : sort(v_1) \quad \ldots \quad t_n : sort(v_n)}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \{v_1 \mapsto t_1; \ldots; v_n \mapsto t_n\}} \ \mathsf{AS}_1$$

$$\frac{t_i :_{ls} s_i}{adequateSorts(v_1 \mapsto t_1; \ldots; v_n \mapsto t_n) \rightsquigarrow \emptyset} \ \mathsf{AS}_2 \ \text{if } s_i \not\leq sort(v_i)$$

$$\frac{\theta(t) : s}{[t : s, \theta] \rightsquigarrow \{\theta\}} \ \mathsf{MbC}_1 \qquad \frac{\theta(t) :_{ls} s'}{[t : s, \theta] \rightsquigarrow \emptyset} \ \mathsf{MbC}_2 \ \text{ if } s' \not\leq s$$

$$\frac{\theta(t_1) \downarrow \theta(t_2)}{[t_1 = t_2, \theta] \rightsquigarrow \{\theta\}} \ \mathsf{EqC}_1 \qquad \frac{\theta(t_1) \to_{norm} t'_1 \quad \theta(t_2) \to_{norm} t'_2}{[t_1 = t_2, \theta] \rightsquigarrow \emptyset} \ \mathsf{EqC}_2 \ \text{if } t'_1 \not\equiv_A t'_2$$

$$\frac{\theta(t_1) \rightsquigarrow_{n+1}^{t_2 := \circledast} S}{[t_1 \Rightarrow t_2, \theta] \rightsquigarrow \{\theta'\theta \mid \theta'(\theta(t_2)) \in S\}} \ \mathsf{RIC} \ \text{ if } n = min(x \in \mathbb{N} : \forall i \geq 0 \ (\theta(t_1) \rightsquigarrow_{x+i}^{t_2 := \circledast} S))$$

$$\frac{[C, \theta_1] \rightsquigarrow \Theta_1 \quad \cdots \quad [C, \theta_m] \rightsquigarrow \Theta_m}{\langle C, \{\theta_1, \ldots, \theta_m\}\rangle \rightsquigarrow \bigcup_{i=1}^{m} \Theta_i} \ \mathsf{SubsCond}$$

Figure 4: Calculus for substitutions

- Rule PatC computes all the possible substitutions that extend θ and satisfy the matching of the term t_2 with the pattern t_1 by first computing the normal form t' of t_2, obtaining then all the possible kind-substitutions κ that make t' and $\theta(t_1)$ equal modulo axioms (indicated by $\equiv_A$), and finally checking that the terms assigned to each variable in the kind-substitutions have the appropriate sort with $adequateSorts(\kappa)$. The union of the set of substitutions thus obtained constitutes the set of substitutions that satisfy the matching.

- Rule AS_1 checks whether the terms of the kind-substitution have the appropriate sort to match the variables. In this case the kind-substitution is a substitution and it is returned.

- Rule AS_2 indicates that, if any of the terms in the kind-substitution has a sort bigger than the required one, then it is not a substitution and thus the empty set of substitutions is returned.

- Rule MbC_1 returns the current substitution if a membership condition holds.

- Rule MbC_2 is used when the membership condition is not satisfied. It checks that the least sort of the term is not less than or equal to the required one, and thus the substitution does not satisfy the condition and the empty set is returned.

- Rule EqC_1 returns the current substitution when an equality condition holds, that is, when the two terms can be joined.

- Rule EqC_2 checks that an equality condition fails by obtaining the normal forms of both terms and then examining that they are different.

- Rewrite conditions are handled by rule RIC. This rule extends the set of substitutions (where we use the juxtaposition of substitutions to express composition) by computing all the reachable terms that satisfy the pattern (using the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ explained below) and then using these terms to obtain the new substitutions.

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \quad \ldots \quad \langle C_n, \Theta_{n-1}\rangle \rightsquigarrow \emptyset}{\mathit{disabled}(a, t)}\ \mathsf{Dsb}$$

$$\text{if } a \equiv l \to r \Leftarrow C_1 \wedge \ldots \wedge C_n \in E \text{ or}$$
$$a \equiv l : s \Leftarrow C_1 \wedge \ldots \wedge C_n \in E$$

$$\frac{\{\theta(u_i) \downarrow \theta(u_i')\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m}}{\theta(l) \to_{red} \theta(r)}\ \mathsf{Rdc_1}\ \text{if } l \to r \Leftarrow \textstyle\bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \in E$$

$$\frac{t \to_{norm} t'}{f(t_1, \ldots, t, \ldots, t_n) \to_{red} f(t_1, \ldots, t', \ldots, t_n)}\ \mathsf{Rdc_2}\ \text{if } t \not\equiv_A t'$$

$$\frac{\mathit{disabled}(e_1, f(t_1, \ldots, t_n)) \quad \ldots \quad \mathit{disabled}(e_l, f(t_1, \ldots, t_n)) \quad t_1 \to_{norm} t_1 \quad \ldots \quad t_n \to_{norm} t_n}{f(t_1, \ldots, t_n) \to_{norm} f(t_1, \ldots, t_n)}\ \mathsf{Norm}$$
$$\text{if } \{e_1, \ldots, e_l\} = \{e \in E \mid e \ll_K^{top} f(t_1, \ldots, t_n)\}$$

$$\frac{t \to_{red} t_1 \quad t_1 \to_{norm} t'}{t \to_{norm} t'}\ \mathsf{NTr}$$

$$\frac{t \to_{norm} t' \quad t' : s \quad \mathit{disabled}(m_1, t') \quad \ldots \quad \mathit{disabled}(m_l, t')}{t :_{ls} s}\ \mathsf{Ls}$$
$$\text{if } \{m_1, \ldots, m_l\} = \{m \in E \mid m \ll_K^{top} t' \ \wedge\ \mathit{sort}(m) < s\}$$

Figure 5: Calculus for normal forms and least sorts

- Finally, rule SubsCond computes the extensions of a set of admissible substitutions for C $\{\theta_1, \ldots, \theta_n\}$ by using the rules above with each of them.

We use these judgments to define the inference rules of Figure 5, that describe how the normal form and the least sort of a term are computed:

- Rule Dsb indicates when an equation or membership a cannot be applied to a term t. It checks that there are no substitutions that satisfy the matching of the term with the lefthand side of the statement and that fulfill its condition. Note that we check the conditions from left to right, following the same order as Maude and making all the substitutions admissible.

- Rule $\mathsf{Rdc_1}$ reduces a term by applying one equation when it checks that the conditions can be satisfied, where the matching conditions are included in the equality conditions. While in the previous rule we made explicit the evaluation from left to right of the condition to show that finally the set of substitutions fulfilling it was empty, in this case we only need one substitution to fulfill the condition and the order is unimportant.

- Rule $\mathsf{Rdc_2}$ reduces a term by reducing a subterm to normal form (checking in the side condition that it is not already in normal form).

- Rule Norm states that the term is in normal form by checking that no equations can be applied at the top considering the variables at the kind level (which is indicated by $\ll_K^{top}$) and that all its subterms are already in normal form.

- Rule NTr describes the transitivity for the reduction to normal form. It reduces the term with the relation $\to_{red}$ and the term thus obtained then is reduced to normal form by using again $\to_{norm}$.

- Rule Ls computes the least sort of the term t. It computes a sort for its normal form (that has the least sort of the terms in the equivalence class) and then checks that memberships deducing smaller sorts cannot be applied.

$$\frac{\mathit{fulfilled}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \{t\}}\ \mathsf{Rf_1} \qquad \frac{\mathit{fails}(\mathcal{C}, t)}{t \rightsquigarrow_0^{\mathcal{C}} \emptyset}\ \mathsf{Rf_2}$$

$$\frac{\theta(P) \downarrow t \quad \{\theta(u_i) \downarrow \theta(u'_i)\}_{i=1}^{n} \quad \{\theta(v_j) : s_j\}_{j=1}^{m} \quad \{\theta(w_k) \Rightarrow \theta(w'_k)\}_{k=1}^{l}}{\mathit{fulfilled}(\mathcal{C}, t)}\ \mathsf{Fulfill}$$

$$\text{if } \mathcal{C} \equiv P := \circledast \wedge \textstyle\bigwedge_{i=1}^{n} u_i = u'_i \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w'_k$$

$$\frac{[P := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0\rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1}\rangle \rightsquigarrow \emptyset}{\mathit{fails}(\mathcal{C}, t)}\ \mathsf{Fail} \ \text{ if } \mathcal{C} \equiv P := \circledast \wedge C_1 \ \wedge \ \ldots \ \wedge \ C_k$$

Figure 6: Calculus for solutions

In these rules Dsb provides the negative information, proving why the statements (either equations or membership axioms) cannot be applied, while the remaining rules provide the positive information indicating why the normal form and the least sort are obtained.

Once these rules have been introduced, we can use them in the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$. First, we present in Figure 6 the rules related to $n = 0$ steps:

- Rule $\mathsf{Rf_1}$ indicates that when only zero steps can be used and the current term fulfills the condition, the set of reachable terms consists only of this term.

- Rule $\mathsf{Rf_2}$ complements $\mathsf{Rf_1}$ by defining the empty set as result when the condition does not hold.

- Rule Fulfill checks whether a term satisfies a condition. The premises of this rule check that all the atomic conditions hold, taking into account that it starts with a matching condition with a hole that must be filled with the current term and thus proved with the premise $\theta(P) \downarrow t$ (the rest of the matching conditions are included in the equality conditions). Note that when the condition is satisfied we do not need to check *all* the substitutions, but only to verify that there exists *one* substitution that makes the condition true.

- To check that a term does not satisfy a condition, it is not enough to check that there exists a substitution that makes it fail; we must make sure that there is no substitution that makes it true. This is indicated by rule Fail, which uses the rules shown in Figure 4 to prove that the set of substitutions that satisfy the condition (where the first set of substitutions is obtained from the first matching condition filling the hole with the current term) is empty. Note that, while rule Fulfill provides the positive information indicating that a condition is fulfilled, this one provides the negative information, proving that the condition does not hold.

Now we introduce in Figure 7 the rules defining the relation $t \rightsquigarrow_n^{\mathcal{C}} S$ when the bound n is greater than 0, which can be understood as searches in *zero or more* steps:

- Rules $\mathsf{Tr_1}$ and $\mathsf{Tr_2}$ show the behavior of the calculus when at least one step can be used. First, we check whether the condition holds (rule $\mathsf{Tr_1}$) or not (rule $\mathsf{Tr_2}$) for the current term, in order to introduce it in the result set. Then, we obtain all the terms reachable in one step with the relation $\Rightarrow_1$, and finally we compute the reachable solutions from these terms constrained by the same condition and the bound decreased by one step. The union of the sets obtained in this way and the initial term, if needed, corresponds to the final result set.

- Rule Stp shows how the set for one step is computed. The result set is the union of the terms obtained by applying each rule *at the top* (calculated with $t \Rightarrow^{top} S$) and the terms obtained by rewriting the arguments of the term one step. This rule can be straightforwardly adapted to the more general case in which the operator f has some *frozen* arguments (i.e., that cannot be rewritten); the implementation of the debugger makes use of this more general rule.

$$\frac{\mathit{fulfilled}(\mathcal{C}, t) \quad t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i \ \cup \ \{t\}} \ \mathsf{Tr_1}$$

$$\frac{\mathit{fails}(\mathcal{C}, t) \quad t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow_n^{\mathcal{C}} S_1 \ \ldots \ t_k \rightsquigarrow_n^{\mathcal{C}} S_k}{t \rightsquigarrow_{n+1}^{\mathcal{C}} \bigcup_{i=1}^{k} S_i} \ \mathsf{Tr_2}$$

$$\frac{f(t_1, \ldots, t_m) \Rightarrow^{top} S_t \quad t_1 \Rightarrow_1 S_1 \quad \cdots \quad t_m \Rightarrow_1 S_m}{f(t_1, \ldots, t_m) \Rightarrow_1 S_t \ \cup \ \bigcup_{i=1}^{m} \{f(t_1, \ldots, u_i, \ldots, t_m) \mid u_i \in S_i\}} \ \mathsf{Stp}$$

$$\frac{t \Rightarrow^{q_1} S_{q_1} \quad \cdots \quad t \Rightarrow^{q_l} S_{q_l}}{t \Rightarrow^{top} \bigcup_{i=1}^{l} S_{q_i}} \ \mathsf{Top} \ \text{ if } \{q_1, \ldots, q_l\} = \{q \in R \mid q \ll_K^{top} t\}$$

$$\frac{[l := t, \emptyset] \rightsquigarrow \Theta_0 \quad \langle C_1, \Theta_0 \rangle \rightsquigarrow \Theta_1 \ \cdots \ \langle C_k, \Theta_{k-1} \rangle \rightsquigarrow \Theta_k}{t \Rightarrow^q \bigcup_{\theta \in \Theta_k} \{\theta(r)\}} \ \mathsf{Rl} \ \text{ if } q : l \Rightarrow r \Leftarrow C_1 \ \wedge \ \ldots \ \wedge \ C_k \ \in R$$

$$\frac{t \rightarrow_{norm} t_1 \quad t_1 \rightsquigarrow_n^{\mathcal{C}} \{t_2\} \cup S \quad t_2 \rightarrow_{norm} t'}{t \rightsquigarrow_n^{\mathcal{C}} \{t'\} \cup S} \ \mathsf{Red_1}$$

Figure 7: Calculus for missing answers

- How to obtain the terms by rewriting at the top is explained by rule Top, which specifies that the result set is the union of the sets obtained with all the possible applications of each rule in the program. We have restricted these rules to those whose lefthand side, with the variables considered at the kind level, matches the term, represented with notation $q \ll_K^{top} t$, where q is the label of the rule and t the current term.
- Rule Rl uses the rules in Figure 4 to compute the set of terms obtained with the application of a single rule. First, the set of substitutions obtained from matching with the lefthand side of the rule is computed, and then it is used to find the set of substitutions that satisfy the condition. This final set is used to instantiate the righthand side of the rule to obtain the set of reachable terms. The kind of information provided by this rule corresponds to the information provided by the substitutions; if the empty set of substitutions is obtained (negative information) then the rule computes the empty set of terms, which also corresponds with negative information proving that no terms can be obtained with this rewrite rule; analogously when the set of substitutions is nonempty (positive information). This information is propagated through the rest of the inference rules justifying why some terms are reachable while others are not.
- Finally, rule $\mathsf{Red_1}$ reduces the reachable terms in order to obtain their normal forms. We use this rule to reproduce Maude behavior, first the normal form of the term is computed and then the rules are applied.

Now we prove that this calculus is correct in the sense that the derived judgments with respect to the rewrite theory $\mathcal{R} = (\Sigma, E, R)$ coincide with the ones satisfied by the corresponding initial model $\mathcal{T}_{\Sigma/E,R}$, i.e., for any judgment φ, φ is derivable in the calculus if and only if $\mathcal{T}_{\Sigma/E,R} \models \varphi$.

Theorem 1. *The calculus of Figures 4, 5, 6, and 7 is correct.*

Once these rules are defined, we can build the tree corresponding to the search result shown in Section 2.5 for the maze example. We recall that we have defined a system to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it:

```
Maude> (search {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
search in MAZE :{[1,1]} =>* {L:List}.

No solution.
```

First of all, we have to use a concrete bound to build the tree. It must suffice to compute all the reachable terms, and in this case the least of these values is 4. We have depicted the tree in Figure 8, where we have abbreviated the equational condition `{L:List}` $:= \circledast \wedge$ `isSol(L:List) = true` by $\mathcal{C}$ and `isSol(L:List) = true` by `isSol(L)`. The leftmost tree justifies that the search condition does not hold for the initial term (this is the reason why Tr_2 has been used instead of Tr_1) and thus it is not a solution. Note that first the substitutions from the matching with the pattern are obtained (`L` $\mapsto$ `[1,1]` in this case), and then these substitutions are used to instantiate the rest of the condition, that for this term does not hold, which is proved by $\overset{2}{\bigtriangledown}$. The next tree shows the set of reachable terms in one step (the tree $\overset{3}{\bigtriangledown}$, explained below, computes the terms obtained by rewrites at the top, while the tree on its right shows that the subterms cannot be further rewritten) and finally the rightmost tree, that has a similar structure to this one and will not be studied in depth, continues the search with the bound decreased in one step.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{1} \rightarrow_{norm} \mathtt{1}}\,\mathsf{Norm}}{\mathtt{[1,1]} \rightarrow_{norm} \mathtt{[1,1]}}\,\mathsf{Norm}}{\mathtt{\{[1,1]\}} \rightarrow_{norm} \mathtt{\{[1,1]\}}}\,\mathsf{Norm}}{[\{\mathtt{L}\} := \mathtt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \{\mathtt{L} \mapsto \mathtt{[1,1]}\}}\,\mathsf{PatC} \quad \dfrac{\overset{2}{\bigtriangledown}}{\langle \mathtt{isSol(L)}, \{\mathtt{L} \mapsto \mathtt{[1,1]}\}\rangle \rightsquigarrow \emptyset}\,\mathsf{SubsCond}}{\mathit{fails}(\mathcal{C}, \mathtt{\{[1,1]\}})}\,\mathsf{Fail} \quad \dfrac{\overset{3}{\bigtriangledown} \quad \dfrac{\dfrac{}{\mathtt{[1,1]} \Rightarrow^{top} \emptyset}\,\mathsf{Top} \quad \dfrac{\dfrac{}{\mathtt{1} \Rightarrow^{top} \emptyset}\,\mathsf{Top}}{\mathtt{1} \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{\mathtt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Stp}}{\mathtt{\{[1,1]\}} \Rightarrow_1 \{\mathtt{\{[1,1][1,2]\}}\}}\,\mathsf{Stp} \quad \dfrac{\bigtriangledown}{\mathtt{\{[1,1][1,2]\}} \rightsquigarrow^{\mathcal{C}}_3 \emptyset}\,\mathsf{Tr}_2}{\mathtt{\{[1,1]\}} \rightsquigarrow^{\mathcal{C}}_4 \emptyset}\,\mathsf{Tr}_2$$

Figure 8: Tree for the maze example

The tree $\overset{2}{\bigtriangledown}$ shows why the current list is not a solution (i.e., the tree provides the negative information proving that this fragment of the condition does not hold). The reason is that the function `isSol` is reduced to `false`, when we needed it to be reduced to `true`.

$$\dfrac{\dfrac{\dfrac{}{\mathtt{isSol([1,1])} \rightarrow_{red} \mathtt{false}}\,\mathsf{Rdc}_1 \quad \dfrac{}{\mathtt{false} \rightarrow_{norm} \mathtt{false}}\,\mathsf{Norm}}{\mathtt{isSol([1,1])} \rightarrow_{norm} \mathtt{false}}\,\mathsf{NTr} \quad \dfrac{}{\mathtt{true} \rightarrow_{norm} \mathtt{true}}\,\mathsf{Norm}}{[\mathtt{isSol(L)} = \mathtt{true}, \mathtt{L} \mapsto \mathtt{[1,1]}] \rightsquigarrow \emptyset}\,\mathsf{EqC}_2$$

Figure 9: Tree $\overset{2}{\bigtriangledown}$ for the search condition

The tree labeled with $\overset{3}{\bigtriangledown}$ is sketched in Figure 10. In this tree the applications of all the rules whose lefthand side matches the current term (`{[1,1]}`) are tried. In this case only the rule `expand` (abbreviated by `e`) can be used, and it generates a list with the new position `[1,2]`; the tree $\overset{4}{\bigtriangledown}$ is used to justify that the first condition of `expand` holds and extends the set of substitutions that fulfill the condition thus far to the set $\{\theta_1, \theta_2, \theta_3\}$, where $\theta_1 \equiv$ `L` $\mapsto$ `[1,1];P` $\mapsto$ `[1,2]`, $\theta_2 \equiv$ `L` $\mapsto$ `[1,1];P` $\mapsto$ `[1,0]`, and $\theta_3 \equiv$ `L` $\mapsto$ `[1,1];P` $\mapsto$ `[0,1]`. The substitution θ_1 also fulfills the next condition, `isOk(L P)`, which is proved with the rule EqC_1 in ($\clubsuit$) (where $\overset{5}{\bigtriangledown}$ is the proof tree shown in Figure 2, proving that the condition holds), while the substitutions θ_2 and θ_3 fail; the trees $\bigtriangledown$ proving it are analogous to the one shown in Figure 9. This substitution θ_1 is thus the only one inferred in the root of the tree, where the node ($\clubsuit$) provides the positive information proving why the substitution is obtained and its siblings ($\bigtriangledown$) the negative information proving why the other substitutions are not in the set.

The tree $\overset{4}{\bigtriangledown}$, shown in Figure 11, is in charge of inferring the set of substitutions obtained when checking the first condition of the rule `expand`, namely `next(L) => P`. The condition is instantiated with the substitution obtained from matching the term with the lefthand side of the rule (in this case `L` $\mapsto$ `[1,1]`) and, since it is a rewrite condition, the set of reachable terms is used to extend this substitution, obtaining a set with three different substitutions (that we previously abbreviated as θ_1, θ_2, and θ_3).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{1 \to_{norm} 1}\,\textsf{Norm}}{\texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\,\textsf{Norm}}{[\texttt{\{L\}} := \texttt{\{[1,1]\}},\ \emptyset] \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]}\}}\,\textsf{PatC} \quad \overset{4}{\bigtriangledown} \quad \dfrac{\dfrac{\overset{5}{\bigtriangledown} \quad \dfrac{}{\texttt{true} \to \texttt{true}}\,\textsf{Rf}_{\to}}{(\clubsuit)\ [\texttt{isOk(L P)},\theta_1] \rightsquigarrow \{\theta_1\}}\,\textsf{EqC}_1 \quad \bigtriangledown \quad \bigtriangledown}{\langle \texttt{isOk(L P)}, \{\theta_1,\theta_2,\theta_3\}\rangle \rightsquigarrow \{\theta_1\}}\,\textsf{SubsCond}}{\texttt{\{[1,1]\}} \Rightarrow^{\mathsf{e}} \{\texttt{\{[1,1][1,2]\}}\}}\,\textsf{Rl}}{\texttt{\{[1,1]\}} \Rightarrow^{top} \{\texttt{\{[1,1][1,2]\}}\}}\,\textsf{Top}$$

Figure 10: Tree $\overset{3}{\bigtriangledown}$ for the applications at the top

$$\dfrac{\dfrac{\dfrac{\bigtriangledown}{\texttt{next([1,1])} \rightsquigarrow_2^{\texttt{P}:=\circledast} \{\texttt{[1,2]}, \texttt{[1,0]}, \texttt{[0,1]}\}}\,\textsf{Tr}_2}{[\texttt{next(L)} \Rightarrow \texttt{P}, \texttt{L} \mapsto \texttt{[1,1]}] \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[0,1]}\}}\,\textsf{RlC}}{\langle \texttt{next(L)} \Rightarrow \texttt{P}, \{\texttt{L} \mapsto \texttt{[1,1]}\}\rangle \rightsquigarrow \{\texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,2]}, \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[1,0]}, \texttt{L} \mapsto \texttt{[1,1]};\texttt{P} \mapsto \texttt{[0,1]}\}}\,\textsf{SubsCond}$$

Figure 11: Tree $\overset{4}{\bigtriangledown}$ for the first condition of `expand`

There are two additional kinds of search allowed in our framework: searches for final terms and searches in *one or more* steps. Figure 12 presents the inference rules for these cases:

- Rules $\textsf{Rf}_3$ and $\textsf{Rf}_4$ are applied when the set of reachable terms in one step is empty (that is, when the term is final). They check whether the term, in addition to being final, fulfills the condition in order to insert it in the result set when appropriate.
- Rule $\textsf{Rf}_5$ specifies that, if the term is not final but no more steps are allowed, then the set of reachable final terms is empty.
- Rule $\textsf{Tr}_3$ shows the transitivity for this kind of search. Since the term is not final, it is not necessary to check whether it fulfills the condition.
- Rule $\textsf{Red}_2$ reduces the reachable final terms in order to obtain their normal forms.
- If only zero steps are available in searches where at least one is required, the empty set is obtained, which is indicated in rule $\textsf{Rf}_6$.
- When at least one step can be used we apply rule $\textsf{Tr}_4$, that indicates that one step is used, and then the relation for zero or more steps is used with the results in order to obtain the final solutions.

The correctness of these inference rules with respect to the initial model $\mathcal{T}_{\Sigma/E,R}$ is proved in the following theorem:

Theorem 2. *The calculus of Figure 12 is correct.*

Following the approach shown in the previous section, we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. As any Σ-term model, $\mathcal{I}$ must satisfy the following soundness propositions:

Proposition 2. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, C an atomic condition, θ an admissible substitution, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If adequateSorts$(\kappa) \rightsquigarrow \Theta$, $[C,\theta] \rightsquigarrow \Theta$, or $\langle C, \Theta\rangle \rightsquigarrow \Theta'$ can be deduced using the rules from Figure 4 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models$ adequateSorts$(\kappa) \rightsquigarrow \Theta$, $\mathcal{T}_{\Sigma/E',R'} \models [C,\theta] \rightsquigarrow \Theta$, and $\mathcal{T}_{\Sigma/E',R'} \models \langle C, \Theta\rangle \rightsquigarrow \Theta'$, respectively.*

Proposition 3. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and φ a judgment deduced with the inference rules* Dsb*,* $\textsf{Rdc}_2$*, or* NTr *from Figure 5 from premises that hold in $\mathcal{T}_{\Sigma/E',R'}$. Then also $\mathcal{T}_{\Sigma/E',R'} \models \varphi$.*

$$\frac{\mathit{fulfilled}(\mathcal{C}, t) \quad t \Rightarrow_1 \emptyset}{t \rightsquigarrow !^{\mathcal{C}}_n \{t\}} \; \mathsf{Rf}_3$$

$$\frac{\mathit{fails}(\mathcal{C}, t) \quad t \Rightarrow_1 \emptyset}{t \rightsquigarrow !^{\mathcal{C}}_n \emptyset} \; \mathsf{Rf}_4$$

$$\frac{t \Rightarrow_1 S}{t \rightsquigarrow !^{\mathcal{C}}_0 \emptyset} \; \mathsf{Rf}_5 \quad S \neq \emptyset$$

$$\frac{t \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow !^{\mathcal{C}}_n S_1 \quad \ldots \quad t_k \rightsquigarrow !^{\mathcal{C}}_n S_k}{t \rightsquigarrow !^{\mathcal{C}}_{n+1} \bigcup_{i=1}^{k} S_i} \; \mathsf{Tr}_3 \quad \text{if } k > 0$$

$$\frac{t \to t_1 \quad t_1 \rightsquigarrow !^{\mathcal{C}}_n \{t_2\} \cup S \quad t_2 \to t'}{t \rightsquigarrow !^{\mathcal{C}}_n \{t'\} \cup S} \; \mathsf{Red}_2$$

$$\frac{}{t \rightsquigarrow +^{\mathcal{C}}_0 \emptyset} \; \mathsf{Rf}_6$$

$$\frac{t \to t' \quad t' \Rightarrow_1 \{t_1, \ldots, t_k\} \quad t_1 \rightsquigarrow^{\mathcal{C}}_n S_1 \quad \ldots \quad t_k \rightsquigarrow^{\mathcal{C}}_n S_k}{t \rightsquigarrow +^{\mathcal{C}}_{n+1} \bigcup_{i=1}^{k} S_i} \; \mathsf{Tr}_4$$

Figure 12: Calculus for final and one or more steps searches

Proposition 4. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow^{\mathcal{C}}_0 S$ can be deduced using rules Rf_1 or Rf_2 from Figure 6 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow^{\mathcal{C}}_0 S$.*

Proposition 5. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow^{\mathcal{C}}_n S$ or $t \Rightarrow_1 S$ can be deduced by means of the rules in Figure 7 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow^{\mathcal{C}}_n S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow_1 S$, respectively.*

Proposition 6. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If a statement $t \rightsquigarrow !^{\mathcal{C}}_n S$ or $t \rightsquigarrow +^{\mathcal{C}}_n S$ can be deduced by means of the rules in Figure 12 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow !^{\mathcal{C}}_n S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow +^{\mathcal{C}}_n S$, respectively.*

Observe that these soundness propositions cannot be extended to the Ls, Fulfill, Fail, Top, and Rl inference rules, where the soundness of the conclusion depends not only on the calculus but also on the specification, which could be wrong.

4. Debugging trees

We describe in this section how to obtain appropriate debugging trees from the proof trees introduced in the previous section. First, we describe the errors that can be found with these proof trees; then, we describe how they can be abbreviated in such a way that soundness and completeness are kept while easing the debugging sessions.

4.1. Debugging with proof trees

As explained in the previous sections, we assume the existence of an *intended interpretation* $\mathcal{I}$ of the given rewrite theory $\mathcal{R} = (\Sigma, E, R)$. This intended interpretation is a Σ-term model corresponding to the model that the user had in mind while writing the specification $\mathcal{R}$. We will say that a judgment is *valid* when it holds in the intended interpretation $\mathcal{I}$, and *invalid* otherwise. Our goal is to find a buggy node (an invalid node with all its children correct) in any proof tree T rooted by the initial error symptom detected by the user. This could be done simply by asking the user questions about the validity of the nodes in the tree according to the following *top-down* strategy:

> **Input:** A tree T with an invalid root.
>
> **Output:** A buggy node in T.
>
> **Description:** Consider the root N of T. There are two possibilities:
>
> - If all the children of N are valid, then finish pointing out N as buggy.
> - Otherwise, select the subtree rooted by any invalid child and recursively use the same strategy to find the buggy node.

Proving that this strategy is complete is straightforward by using induction on the height of T. As an easy consequence, the following result holds:

Proposition 7. *Let T be a proof tree with an invalid root. Then there exists a buggy node $N \in T$ such that all the ancestors of N are invalid.*

By using the proof trees computed with the calculus of the previous section as debugging trees we are able to locate wrong statements, missing statements, and wrong search conditions, which are defined as follows:

- Given a statement $A \Leftarrow C_1 \wedge \cdots \wedge C_n$ (where A is either an equation $l = r$, a membership $l : s$, or a rule $l \Rightarrow r$) and a substitution θ, the *statement instance* $\theta(A) \Leftarrow \theta(C_1) \wedge \cdots \wedge \theta(C_n)$ is *wrong* when all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$ but $\theta(A)$ is not.

- Given a rule $l \Rightarrow r \Leftarrow C_1 \wedge \cdots \wedge C_n$ and a term t, the rule has a *wrong instance* if the judgments $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, …, $[C_n, \Theta_{n-1}] \rightsquigarrow \Theta_n$ are valid in $\mathcal{I}$ but the application of Θ_n to the righthand side does not provide all the results expected for this rule.

- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if $[l := t, \emptyset] \rightsquigarrow \Theta_0$, $[C_1, \Theta_0] \rightsquigarrow \Theta_1$, …, $[C_n, \Theta_{n-1}] \rightsquigarrow \emptyset$ are valid in $\mathcal{I}$ (meaning that the condition does not hold for t) but the user expected the condition to hold, then we have a *wrong search condition instance.*

- Given a condition $l := \circledast \wedge C_1 \wedge \cdots \wedge C_n$ and a term t, if there exists a substitution θ such that $\theta(l) \equiv_A t$ and all the atomic conditions $\theta(C_i)$ are valid in $\mathcal{I}$, but the condition is not expected to hold, then we also have a *wrong search condition instance.*

- A statement or condition is *wrong* when it admits a wrong instance.

- Given a term t, there is a *missing equation for t* if t is not expected to be in normal form and none of the equations in the specification are expected to be applied to it.

- A specification has a *missing equation* if there exists a term t such that there is a missing equation for t.

- Given a term t, there is a *missing membership for t* if t is an expected normal form such that the computed least sort of t is not the expected one and none of the membership axioms in the specification are expected to be applied to it.

$\mathsf{Rep}_{\rightarrow}$	Wrong equation
$\mathsf{Rep}_{\Rightarrow}$	Wrong rule
Mb	Wrong membership
Rdc_1	Wrong equation
Norm	Missing equation
Ls	Missing membership
Fulfill	Wrong search condition
Fail	Wrong search condition
Top	Missing rule
Rl	Wrong rule

Table 3: Errors detected by the proof trees

- A specification has a *missing membership* if there exists a term t such that there is a missing membership for t.
- Given a term t, there is a *missing rule for* t if all the rules applied to t at the top lead to judgments $t \Rightarrow^{q_i} S_{q_i}$ valid in $\mathcal{I}$ but the union $\bigcup S_{q_i}$ does not contain all the reachable terms from t by using rewrites at the top.
- A specification has a *missing rule* if there exists a term t such that there is a missing rule for t.

We relate these definitions with our calculus in the following proposition:

Proposition 8. *Let N be a buggy node in some proof tree in the calculus of Figures 1, 4, 5, 6, 7, and 12 w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N corresponds to the consequence of an inference rule in the first column of Table 3.*
2. *The error associated to N can be obtained from the inference rule as shown in the second column of Table 3.*

We assume that the nodes inferred with these inference rules are decorated with some extra information to identify the error when they are pointed out as buggy. More specifically, nodes related to wrong statements keep the label of the statement, nodes related to missing statements keep the operator at the top that requires more statements to be defined, and nodes related to wrong conditions keep the condition. With this information available, when a wrong statement is found this specific statement is pointed out; when a missing statement is found, the debugger indicates the operator at the top of the term in the lefthand side of the statement that is missing; and when a wrong condition is found, the specific condition is shown. Actually, when a missing statement is found what the debugger reports is that a statement is missing *or* the conditions in the remaining statements are not the intended ones (thus they are not applied when expected and another one would be needed), but the error *is not located* in the statements used in the conditions, since they are also checked during the debugging process. Finally, it is important not to confuse missing answers with missing statements; the current calculus detects missing answers due to both wrong and missing statements and wrong search conditions.

4.2. Abbreviated proof trees

We will not use the proof trees T computed in the previous sections directly as debugging trees, but a suitable abbreviation which we denote by $APT(T)$ (from *Abbreviated Proof Tree*), or simply APT if the proof tree T is clear from the context. The reason for preferring the APT to the original proof tree is that it reduces and simplifies the questions that will be asked to the user while keeping the soundness and completeness of the technique. This transformation relies on Proposition 8: only potential buggy nodes are kept.

The rules for deriving an APT can be seen in Figure 13. The abbreviation always starts by applying ($\mathbf{APT}_1$). This rule simply duplicates the root of the tree and applies APT', which receives a proof tree and returns a forest (i.e., a set of trees). Hence without this duplication the result of the abbreviation could be a forest instead of a single tree. The rest of the APT rules correspond to the function APT' and are assumed to be applied top-down: if several APT rules can be applied at the root of a proof tree, we must choose the first one, that is, the rule with the lowest index. The following advantages are obtained with this transformation:

- Questions associated to nodes with reductions are improved (rules ($\mathbf{APT}_2$), ($\mathbf{APT}_3$), ($\mathbf{APT}_5$), ($\mathbf{APT}_6$), and ($\mathbf{APT}_7$)) by asking about normal forms instead of asking about intermediate states. For example, in rule ($\mathbf{APT}_2$) the error associated to $t_1 \to t_2$ is the one associated to $t_1 \to t'$, which is not included in the APT. We have chosen to introduce $t_1 \to t_2$ instead of simply $t_1 \to t'$ in the APT as a pragmatic way of simplifying the structure of the APTs, since t_2 is obtained from t' and hence likely simpler.

- The rule ($\mathbf{APT}_4$) deletes questions about rewrites *at the top* of a given term (that may be difficult to answer due to matching modulo) and associates the information of those nodes to questions related to the set of reachable terms in one step with rewrites in any position, that are in general easier to answer.

- It creates, with the variants of the rules ($\mathbf{APT}_8$) and ($\mathbf{APT}_9$), two different kinds of tree, one that contains judgments of rewrites with several steps and another that only contains rewrites in one step. The one-step debugging tree strictly follows the idea of keeping only nodes corresponding to relevant information. However, the many-steps debugging tree also keeps nodes corresponding to the *transitivity* inference rules. The user will choose which debugging tree (one-step or many-steps) will be used for the debugging session, taking into account that the many-steps debugging tree usually leads to shorter debugging sessions (in terms of the number of questions) but with likely more complicated questions. The number of questions is usually reduced because keeping the transitivity nodes for rewrites gives some parts of the debugging tree the shape of a balanced binary tree (each transitivity inference has two premises, i.e., two child subtrees), and this allows the debugger to efficiently use the divide and query navigation strategy. On the contrary, removing the transitivity inferences for rewrites (as rules ($\mathbf{APT}_8^o$) and ($\mathbf{APT}_9^o$) do) produces flattened trees where this strategy is no longer so efficient. On the other hand, in rewrites $t \Rightarrow t'$ and searches $t \leadsto_n^{\mathcal{C}} S$ appearing as the conclusion of a transitivity inference rule, the judgment can be more complicated because it combines several inferences. The user must balance the pros and cons of each option, and choose the best one for each debugging session.

- The rule ($\mathbf{APT}_{11}$) removes from the tree all the nodes which are not associated with relevant information, since the rule ($\mathbf{APT}_{10}$) keeps the relevant information and the rules are applied in order. We remove, for example, nodes related to judgments about sets of substitutions, disabled statements, and rewrites with a concrete rule. Moreover, it removes trivial judgments, like the ones related to reflexivity or congruence, from the tree.

- Since the APT is built without computing the associated proof tree, it reduces the time and space needed to build the tree.

We can state the correctness and completeness of the debugging technique based on APTs:

Theorem 3. *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 4, 5, 6, 7, and 12 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*

- *Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ according to Table 3 (correctness).*

$$(\mathbf{APT}_1)\quad APT\left(\frac{T_1\dots T_n}{\varphi}\,\mathsf{R_1}\right) = \frac{APT'\left(\frac{T_1\dots T_n}{\varphi}\,\mathsf{R_1}\right)}{\varphi}$$

$$(\mathbf{APT}_2)\quad APT'\left(\frac{\dfrac{T_1\dots T_n}{t_1\to t'}\,\mathsf{Rep_{\to}}\quad T'}{t_1\to t_2}\,\mathsf{Tr_{\to}}\right) = \left\{\frac{APT'(T_1)\dots APT'(T_n)\ APT'(T')}{t_1\to t_2}\,\mathsf{Rep_{\to}}\right\}$$

$$(\mathbf{APT}_3)\quad APT'\left(\frac{\dfrac{T_1\ \dots\ T_n}{t\to t''}\,\mathsf{Rdc1}\quad T'}{t\to t'}\,\mathsf{NTr}\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T')}{t\to t'}\,\mathsf{Rdc1}\right\}$$

$$(\mathbf{APT}_4)\quad APT'\left(\frac{\dfrac{T_1\ \dots\ T_n}{t\Rightarrow^{top} S'}\,\mathsf{Top}\quad T'_1\dots T'_m}{t\Rightarrow_1 S}\,\mathsf{Stp}\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T'_1)\ \dots\ APT'(T'_m)}{t\Rightarrow_1 S}\,\mathsf{Top}\right\}$$

$$(\mathbf{APT}_5)\quad APT'\left(\frac{T'\quad \dfrac{T_1\dots T_n}{t\Rightarrow t'}\,\mathsf{Rep_{\Rightarrow}}\quad T''}{t_1\Rightarrow t_2}\,\mathsf{EC}\right) = \left\{\frac{APT'(T')\ APT'(T_1)\dots APT'(T_n)\ APT'(T'')}{t_1\Rightarrow t_2}\,\mathsf{Rep_{\Rightarrow}}\right\}$$

$$(\mathbf{APT}_6)\quad APT'\left(\frac{T\quad \dfrac{T_1\dots T_n}{\varphi'}\,\mathsf{R_1}\quad T'}{\varphi}\,\mathsf{Red_i}\right) = \left\{\frac{APT'(T)\ APT'(T_1)\ \dots\ APT'(T_n)\ APT'(T)}{\varphi}\,\mathsf{R_1}\right\}$$

$$(\mathbf{APT}_7)\quad APT'\left(\frac{T_{t\to_{norm}t'}\ T_1\dots T_n}{t :_{ls} s}\,\mathsf{Ls}\right) = \left\{\frac{APT'\left(T_{t\to_{norm}t'}\right)\ APT'(T_1)\ \dots\ APT'(T_n)}{t' :_{ls} s}\,\mathsf{Ls}\right\}$$

$$(\mathbf{APT}_8^o)\quad APT'\left(\frac{T_1\quad T_2}{t_1\Rightarrow t_2}\,\mathsf{Tr_{\Rightarrow}}\right) = APT'(T_1)\ \bigcup\ APT'(T_2)$$

$$(\mathbf{APT}_8^m)\quad APT'\left(\frac{T_1\quad T_2}{t_1\Rightarrow t_2}\,\mathsf{Tr_{\Rightarrow}}\right) = \left\{\frac{APT'(T_1)\ APT'(T_2)}{t_1\Rightarrow t_2}\,\mathsf{Tr_{\Rightarrow}}\right\}$$

$$(\mathbf{APT}_9^o)\quad APT'\left(\frac{T_1\dots T_n}{\varphi}\,\mathsf{Tr_j}\right) = APT'(T_1)\ \bigcup\ \dots\ \bigcup\ APT'(T_n)$$

$$(\mathbf{APT}_9^m)\quad APT'\left(\frac{T_1\dots T_n}{\varphi}\,\mathsf{Tr_j}\right) = \left\{\frac{APT'(T_1)\ \dots\ APT'(T_n)}{\varphi}\,\mathsf{Tr_j}\right\}$$

$$(\mathbf{APT}_{10})\quad APT'\left(\frac{T_1\dots T_n}{\varphi}\,\mathsf{R_2}\right) = \left\{\frac{APT'(T_1)\dots APT'(T_n)}{\varphi}\,\mathsf{R_2}\right\}$$

$$(\mathbf{APT}_{11})\quad APT'\left(\frac{T_1\dots T_n}{\varphi}\,\mathsf{R_1}\right) = APT'(T_1)\ \bigcup\ \dots\ \bigcup\ APT'(T_n)$$

R_1 any inference rule R_2 either Mb, $\mathsf{Rep_{\to}}$, $\mathsf{Rep_{\Rightarrow}}$, $\mathsf{Rdc_1}$, Norm, Fulfill, Fail, Ls, Rl, or Top

$1 \le i \le 2$ $1 \le j \le 4$ φ, φ' any judgment

Figure 13: Transforming rules for obtaining abbreviated proof trees

$$\dfrac{\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \texttt{1} \to_{norm} \texttt{1}}\,\mathsf{Norm}_{\mathsf{s\_}}}{(\spadesuit)\ \texttt{[1,1]} \to_{norm} \texttt{[1,1]}}\,\mathsf{Norm}_{\mathsf{[\_,\_]}}}{(\spadesuit)\ \texttt{\{[1,1]\}} \to_{norm} \texttt{\{[1,1]\}}}\,\mathsf{Norm}_{\mathsf{\{\_\}}} \quad \dfrac{}{\texttt{isSol}(P_1) \to \texttt{f}}\,\mathsf{Rdc}_{\mathsf{is2}} \quad \overset{6}{\bigtriangledown} \quad \bigtriangledown \;\cdots\; \bigtriangledown \quad \overset{7}{\bigtriangledown}}{\texttt{\{[1,1]\}} \leadsto^{\mathcal{C}}_{4} \emptyset}\,\mathsf{Tr}_2$$

Figure 14: Abbreviated proof tree for the maze example

$$\dfrac{\dfrac{\dfrac{}{(\spadesuit)\ \texttt{1} \to_{norm} \texttt{1}}\,\mathsf{Norm}_{\mathsf{s\_}}}{(\spadesuit)\ \texttt{[1,1]} \to_{norm} \texttt{[1,1]}}\,\mathsf{Norm}_{\mathsf{[\_,\_]}} \quad \dfrac{\bigtriangledown \quad \overset{8}{\bigtriangledown} \quad \dfrac{}{(\diamondsuit)\ \texttt{isOk}(L_2) \to \texttt{f}}\,\mathsf{Rep}_{\bot} \quad \dfrac{}{(\diamondsuit)\ \texttt{isOk}(L_3) \to \texttt{f}}\,\mathsf{Rep}_{\bot}}{\texttt{\{[1,1]\}} \Rightarrow^{\mathsf{e}} \{\texttt{\{[1,1][1,2]\}}\}}\,\mathsf{Rl}_{\mathsf{e}} \quad \dfrac{\dfrac{}{(\heartsuit)\ \texttt{1} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathsf{s\_}}}{(\heartsuit)\ \texttt{[1,1]} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathsf{[\_,\_]}}}{\texttt{\{[1,1]\}} \Rightarrow_1 \{\texttt{\{[1,1][1,2]\}}\}}\,\mathsf{Top}_{\mathsf{\{\_\}}}$$

Figure 15: Abbreviated tree $\overset{6}{\bigtriangledown}$

The theorem states that we can safely employ the abbreviated proof tree as a basis for the declarative debugging of Maude system and functional modules: the technique will find a buggy node starting from any initial symptom detected by the user. Of course, these results assume that the user correctly answers all the questions about the validity of the APT nodes asked by the debugger (see Section 2.6).

The trees in Figures 14–17 depict the (one-step) abbreviated proof tree for the maze example, where $\mathcal{C}$ stands for `{L:List}:=` $\circledast\ \wedge$ `isSol(L:List)`, P_1 for `[1,1]`, L_1 for `[1,1][1,2]`, L_2 for `[1,1][1,0]`, L_3 for `[1,1][0,1]`, `t` for `true`, `f` for false, `n` for `next`, `e` for `expand`, and L for `[1,1][1,2][1,3][1,4]`. We have also extended the information in the labels with the operator or statement associated to the inference. More concretely, the tree in Figure 14 abbreviates the tree in Figure 8; the first two premises in the abbreviated tree stand for the first premise in the proof tree (which includes the tree in Figure 9), keeping only the nodes associated with relevant information according to Proposition 8: Norm, with the operator associated to the reduction, and $\mathsf{Rdc_1}$, with the label of the associated equation. The tree $\overset{6}{\bigtriangledown}$, shown in Figure 15, abbreviates the second premise of the tree in Figure 8 as well as the trees in Figures 10 and 11; it only keeps the nodes referring to normal forms, searches in one step, that are now associated to the rule Top, each of them referring to a different operator (the operator `s_` is the successor constructor for natural numbers), and the applications of rules (Rl) and equations ($\mathsf{Rep}_{\to}$). Note that the equation describing the behavior of `isOk` has not got any label, which is indicated with the symbol $\bot$; we will show below how the debugger deals with these nodes. The tree $\overset{7}{\bigtriangledown}$, presented in Figure 16, shares these characteristics and only keeps nodes related to one-step searches and application of rules. The tree $\overset{8}{\bigtriangledown}$ abbreviates the proof tree for the reduction shown in Figure 2, where the important result of the abbreviation is that all replacement inferences are related now to reductions to normal form, thus easing the questions that will be asked to the user.

These abbreviation rules are combined with trusting mechanisms that further reduce the proof tree:

- Statements can be trusted in several ways: non labelled statements, which include the predefined functions, are always trusted (i.e., the nodes marked with ($\diamondsuit$) in Figures 15 and 17 will be discarded by the debugger); statements and modules can be trusted before starting the debugging process; and statements can also be trusted on the fly.

$$\dfrac{\dfrac{\bigtriangledown \;\cdots\; \bigtriangledown \quad \dfrac{\dfrac{\bigtriangledown \;\cdots\; \bigtriangledown}{\texttt{n}(L) \Rightarrow^{\mathtt{n1}} \{\texttt{[1,5]}\}}\,\mathsf{Rl}_{\mathsf{n1}} \quad \dfrac{\bigtriangledown \;\cdots\; \bigtriangledown}{\texttt{n}(L) \Rightarrow^{\mathtt{n2}} \{\texttt{[0,4]}\}}\,\mathsf{Rl}_{\mathsf{n2}} \quad \dfrac{\bigtriangledown \;\cdots\; \bigtriangledown}{\texttt{n}(L) \Rightarrow^{\mathtt{n3}} \{\texttt{[1,3]}\}}\,\mathsf{Rl}_{\mathsf{n3}}}{(\ddagger)\ \texttt{n}(L) \Rightarrow_1 \{\texttt{[1,5]}, \texttt{[0,4]}, \texttt{[1,3]}\}}\,\mathsf{Top}_{\mathsf{n}}}{(\wr)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow^{\mathsf{e}} \emptyset}\,\mathsf{Rl}_{\mathsf{e}}}{(\dagger)\ \texttt{\{[1,1][1,2][1,3][1,4]\}} \Rightarrow_1 \emptyset}\,\mathsf{Top}_{\mathsf{\{\_\}}}$$

Figure 16: Abbreviated tree $\overset{7}{\bigtriangledown}$

$$\dfrac{\dfrac{\bigtriangledown}{(\Diamond)\ \mathtt{1\ >=\ 1} \to \mathtt{t}}\,\mathsf{Rep}_\perp \quad \dots \quad \dfrac{\bigtriangledown}{(\Diamond)\ \mathtt{2\ <=\ 8}\to\mathtt{t}}\,\mathsf{Rep}_\perp \quad \dfrac{\dfrac{\dfrac{\dfrac{}{\mathtt{c(nil,[1,2])}\to\mathtt{f}}\,\mathsf{Rep}_{c1}}{(\Diamond)\ t_2\to\mathtt{f}}\,\mathsf{Rep}_\perp}{(\Diamond)\ t_1\to\mathtt{t}}\,\mathsf{Rep}_\perp}{\mathtt{c([1,1],\ [1,2])}\to\mathtt{f}}\,\mathsf{Rep}_{c2} \quad \dfrac{}{(\Diamond)\ \mathtt{not(f)}\to\mathtt{t}}\,\mathsf{Rep}_\perp \quad \dfrac{\bigtriangledown}{(\Diamond)\ \mathtt{t\ and\ ...\ and\ t}\to\mathtt{t}}\,\mathsf{Rep}_\perp}{(\Diamond)\ \mathtt{isOk([1,1][1,2])}\to\mathtt{t}}\,\mathsf{Rep}_\perp$$

Figure 17: Abbreviated proof tree $\overset{8}{\bigtriangledown}$

- A correct module can be given before starting a debugging session. By checking the correctness of the judgments against this module, correct nodes can be deleted from the tree.
- Constructed terms (that is, terms built only with constructors, defined by means of the `ctor` attribute) of certain sorts or built with some operators can be considered *final*, which indicates that they cannot be further rewritten. For example, we could consider terms of sorts `Nat` and `List` (and hence its subsort `Pos`) to be final and thus the nodes marked with (♡) in Figure 15 would be removed from the tree.
- Moreover, we consider that constructed terms are in normal form and thus they are automatically removed from the tree. For example, the nodes marked with (♠) in Figures 14 and 15 will be removed from the debugging tree.

5. Using the debugger

We introduce in this section how to create and navigate the debugging tree.

5.1. Creating the debugging tree

We describe in this section how to start the debugging process, describing the commands that must be used before creating the debugging tree and the different commands to create it.

The debugger is initiated in Maude by loading the file `dd.maude` (available from `http://maude.sip.ucm.es/debugging`), which starts an input/output loop that allows the user to interact with the tool. Then, the user can enter Full Maude modules and commands, as well as commands for the debugger. Tables 4 and 5 present a summary of the commands explained below.

The user can choose between using all the labeled statements in the debugging process (by default) or selecting some of them by means of the command

```
(set debug select on .)
```

Once this mode is activated, the user can select and deselect statements by using[5]

```
(debug select LABELS .)
(debug deselect LABELS .)
```

where `LABELS` is a list of statement labels separated by spaces.

Moreover, all the labels in statements of a flattened module can be selected or deselected with the commands

```
(debug include MODULES .)
(debug exclude MODULES .)
```

[5]Although these labels, as well as the set of labels from a module and the final sorts below, can be selected and deselected with the corresponding modes switched off, they will have effect only when the corresponding modes are activated.

Command	**Effect**	**When**
Trusting		
`(set debug select on .)`	Activates trusting	Before starting the debugging
`(set debug select off .)`	Deactivates trusting	Before starting the debugging
`(debug select LABELS .)`	Suspects of `LABELS`	Before starting the debugging
`(debug deselect LABELS .)`	Trusts `LABELS`	Before starting the debugging
`(debug include MODULES .)`	Suspects of `MODULES`	Before starting the debugging
`(debug exclude MODULES .)`	Trusts `MODULES`	Before starting the debugging
`(debug include eqs MODULES .)`	Suspects of the equations in `MODULES`	Before starting the debugging
`(debug exclude eqs MODULES .)`	Trusts the equations in `MODULES`	Before starting the debugging
`(debug include mbs MODULES .)`	Suspects of the memberships in `MODULES`	Before starting the debugging
`(debug exclude mbs MODULES .)`	Trusts the memberships in `MODULES`	Before starting the debugging
`(debug include rls MODULES .)`	Suspects of the rules in `MODULES`	Before starting the debugging
`(debug exclude rls MODULES .)`	Trusts the rules in `MODULES`	Before starting the debugging
`(correct module MODULE-NAME .)`	Sets the correct module	Before starting the debugging
`(set bound BOUND .)`	Sets the bound for the correct module	Before starting the debugging
`(delete correct module .)`	Deletes the correct module	Before starting the debugging
`(set final select on .)`	Activates “final” trusting	Before starting the debugging
`(set final select off .)`	Deactivates “final” trusting	Before starting the debugging
`(final select SORTS .)`	Sorts `SORTS` are final	Before starting the debugging
`(final deselect SORTS .)`	Sorts `SORTS` are not final	Before starting the debugging
Tree options		
`(one-step tree .)`	Selects the one-step tree for wrong rewrites	Before starting the debugging
`(many-steps tree .)`	Selects the many-steps tree for wrong rewrites	Before starting the debugging
`(one-step missing tree .)`	Selects the one-step tree for missing rewrites	Before starting the debugging
`(many-steps missing tree .)`	Selects the many-steps tree for missing rewrites	Before starting the debugging
`(solutions prioritized on .)`	Prioritizes questions about solutions	Before starting the debugging
`(solutions prioritized off .)`	Does not prioritize questions about solutions	Before starting the debugging
Strategies		
`(top-down strategy .)`	Switches to the top-down strategy	At any time
`(divide-query strategy .)`	Switches to the divide and query strategy	At any time

Table 4: Available commands I

Command	**Effect**	**When**
Debugging		
`(debug [in MOD :] INIT-TERM -> WRONG-TERM .)`	Starts the debugging for wrong reductions	At any time
`(debug [in MOD :] INIT-TERM : WRONG-SORT .)`	Starts the debugging for wrong sort inferences	At any time
`(debug [in MOD :] INIT-TERM =>* WRONG-TERM .)`	Starts the debugging for wrong rewrites	At any time
`(missing [in MOD :] INIT-TERM -> ERR-NORMAL-FORM .)`	Starts the debugging for incomplete normal forms	At any time
`(missing [in MOD :] INIT-TERM : ERR-LEAST-SORT .)`	Starts the debugging for bigger than expected least sorts	At any time
`(missing [[dpth]] [in MOD :] TERM =>* PAT [s.t. CND] .)`	Starts the debugging for incomplete sets in zero or more steps	At any time
`(missing [[dpth]] [in MOD :] TERM =>+ PAT [s.t. CND] .)`	Starts the debugging for incomplete sets in one or more steps	At any time
`(missing [[dpth]] [in MOD :] TERM =>! PAT [s.t. CND] .)`	Starts the debugging for incomplete sets of final terms	At any time
Answers		
`(yes .)`	The judgment is correct	Divide and query strategy
`(N : yes .)`	The `N`th judgment is correct	Top-down strategy
`(all : yes .)`	All the judgments are correct	Top-down strategy
`(no .)`	The judgment is incorrect	Divide and query strategy
`(N : no .)`	The `N`th judgment is incorrect	Top-down strategy
`(trust .)`	The statement associated with the current judgment is trusted	Divide and query strategy
`(N : trust .)`	The statement associated with the `N`th judgment is trusted	Top-down strategy
`(I is wrong .)`	The `I`th element is not reachable	Divide and query strategy
`(N : I is wrong .)`	The `I`th element of the `N`th judgment is not reachable	Top-down strategy
`(I is not a solution .)`	The `I`th element is not a solution	Divide and query strategy
`(N : I is not a solution .)`	The `I`th element of the `N`th judgment is not a solution	Divide and query strategy
`(its sort is final .)`	The sort of the current term is final	Divide and query strategy
`(N : its sort is final)`	The sort of the term in the `N`th judgment is final	Top-down strategy
`(don't know .)`	Skips the current judgment	Divide and query strategy
`(undo .)`	Returns to the previous state	At any time

Table 5: Available commands II

where MODULES is a list of module names separated by spaces.

The selection mode can be switched off by using the command

```
(set debug select off .)
```

In a similar way, it is also possible to indicate that some terms are final, that is, that they cannot be further rewritten:

- By using the value `final` in the attribute `metadata` of an operator declaration, that indicates that the terms built with this operator at the top are final.
- By selecting a set of final sorts. In this case, constructed terms having one of these sorts (or having a subsort of these sorts) are considered final.
- On the fly, as will be explained below.

In the first two cases, the user must activate the final sorts mode with the command

```
(set final select on .)
```

While the attribute `metadata` must be written in the Maude file, final sorts can be selected/deselected with the commands

```
(final select SORTS .)
(final deselect SORTS .)
```

where SORTS is a list of sort identifiers separated by spaces.

This option can be switched off with the command

```
(set final select off .)
```

A module with only correct definitions can be used to reduce the number of questions. In this case, it must be indicated before starting the debugging process with the command

```
(correct module MODULE-NAME .)
```

and can be deselected with the command

```
(delete correct module .)
```

Since rewriting is not assumed to terminate, a bound, which is 42 by default, is used when searching in the correct module and can be set with the command

```
(set bound BOUND .)
```

where BOUND is either a natural number or the constant `unbounded`. Note that if it is 0 the correct module will not be used for rewrites, while if it is `unbounded` the correct module is assumed to be terminating.

When debugging wrong rewrites, two different trees can be built: one whose questions are related to one-step rewrites and another whose questions are related to several steps. The user can switch between these trees, before starting the debugging process, with the commands

```
(one-step tree .)
(many-steps tree .)
```

the first of which is the default one.

In the same way, when debugging missing answers we distinguish between trees whose nodes are related to sets of terms obtained with one (the default case) or many steps. The user can select them with the commands

```
(one-step missing tree .)
(many-steps missing tree .)
```

When debugging missing answers, the user can prioritize questions related to the fulfillment of the search condition from questions involving the statements defining it. This option, switched off by default, can be activated with the command

```
(solutions prioritized on .)
```

and can be switched off again with

```
(solutions prioritized off .)
```

The debugging process for wrong answers is started with the commands

```
(debug [in MODULE-NAME :] INITIAL-TERM -> WRONG-TERM .)
(debug [in MODULE-NAME :] INITIAL-TERM : WRONG-SORT .)
(debug [in MODULE-NAME :] INITIAL-TERM =>* WRONG-TERM .)
```

for wrong reductions, memberships, and rewrites, respectively. `MODULE-NAME` is the module where the computation took place; if no module name is given, the current module is used by default. Similarly, we start the debugging of missing answers with the commands

```
(missing [in MODULE-NAME :] INITIAL-TERM -> ERR-NORMAL-FORM .)
(missing [in MODULE-NAME :] INITIAL-TERM : ERR-LEAST-SORT .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>* PATTERN [s.t. CONDITION] .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>+ PATTERN [s.t. CONDITION] .)
(missing [[depth]] [in MODULE-NAME :] INITIAL-TERM =>! PATTERN [s.t. CONDITION] .)
```

where the first command debugs erroneous normal forms, the second one erroneous least sorts, and the remaining ones refer to incomplete sets found when using search. More specifically, the third command specifies a search in zero or more steps, the fourth command in one or more steps, and the last one only checks final terms. The `depth` argument indicates the bound in the number of steps allowed in the search, and it is considered unbounded when omitted, while `MODULE-NAME` has the same behavior as in the commands above.

5.2. Navigating the debugging tree

We describe in this section how the debugging tree created with the commands described in the previous section is traversed. The debugging tree can be navigated by using two different strategies, namely, *top-down* and *divide and query*, the latter being the default one. The user can switch between them at any moment by using the commands

```
(top-down strategy .)
(divide-query strategy .)
```

In the divide and query strategy, each question refers to one judgment that can be either correct or wrong. The different answers are transmitted to the debugger with the answers

```
(yes .)
(no .)
```

If the question asked is too difficult, the user can avoid answering with[6]

[6]Notice that in the current version of the debugger the question will not be asked again, thus this answer can lead to incompleteness.

```
(don't know .)
```

To know the appropriate answer, we briefly describe the different kinds of questions asked by the debugger, defining for each of them when they are considered correct and describing the additional answers that can be used in each specific case. The possible questions are related to:

Reductions. When a term t has been reduced by using equations to another term t', the debugger asks questions of the form “Is this reduction correct? $t \to t'$.” These judgments are correct if the user expected t to be fully reduced to t' by using the equational part (equations and memberships) of the module.

In addition to the general answers, when the question corresponds to the application of a specific statement (either a equation, like in this case, a membership, or a rule), instead of just answering `yes`, we can also *trust* the statement on the fly if we decide the bug is not there. To trust the current statement we answer `(trust .)`.

Normal forms. When a term cannot be further reduced and it is not a constructed term, the debugger asks “Is t in normal form?,” which is correct if the user expected t to be a normal form.

Memberships. When a sort s is inferred for a term t, the debugger prompts questions of the form “Is this membership correct? $t : s$.” These judgments are correct if the expected least sort of t is a subsort of s or s itself.

Least sorts. When the judgment refers to the least sort ls of a term t, the tool makes questions of the form “Did you expect t to have least sort ls?.” In this case, the judgment is correct if the intended least sort of t is exactly ls.

Rewrites in one step. When a term t is rewritten into another term t' in only one step, the debugger asks questions of the form “Is this rewrite correct? $t \Rightarrow_1 t'$,” where t' has already been fully reduced by using equations. This judgment is correct if the user expected to obtain t' from t modulo equations with only one rewrite.

Rewrites in several steps. When a term t is rewritten into another one t' after several rewrite steps, the debugger shows the question “Is this rewrite correct? $t \Rightarrow^+ t'$,” where t' is fully reduced. This question is only prompted if the user selects the many-steps tree for wrong answers. This judgment is correct if t' is expected to be reachable from t.

Final terms. When a term t cannot be further rewritten, the debugger asks “Did you expect t to be final?.” This judgment is correct if the user expected that no rules can be applied to t.

Additional information for this question can be given by answering `(its sort is final .)`, that indicates to the debugger that all the constructed terms with the same sort as this term are final.

Solutions. When a term t fulfills the search condition, the debugger shows questions of the form “Did you expect t to be a solution?.” This judgment is correct if t is one of the intended solutions. In the same way, if a term does not fulfill the search condition the debugger asks “Did you expect t not to be a solution?,” that is correct if t is not one of the expected solutions.

Reachable terms in one step. When all the possible applications of each rule in the current specification to a term t lead to a set of terms $\{t_1, \ldots, t_n\}$, with $n > 0$, the debugger prompts the question “Are the following terms all the reachable terms from t in one step? $t_1, \ldots, t_n$.” This judgment is correct if all the expected terms from t in one step constitute the set $\{t_1, \ldots, t_n\}$.

In this case, if one of the terms is not reachable, the user can point it out with the answer `(I is wrong .)` where `I` is the index of the wrong term in the set. With this answer the debugger focuses on debugging this wrong judgment. This answer can also be used for reachable terms with one rule and in several steps.

Reachable terms with one rule. Given a term t and a rule r, when all the possible applications of r to t produce a set of terms $\{t_1, \ldots, t_n\}$, the debugger presents questions of the form "Are the following terms all the reachable terms from t with one application of the rule r? $t_1, \ldots, t_n$." This judgment is correct if all the expected reachable terms from t with one application of r form the set $\{t_1, \ldots, t_n\}$. When $n = 0$ the debugger prompts questions of the form "Did you expect that no terms can be obtained from t by applying the rule r?," that is correct if the rule r is not expected to be applied to t.

Reachable terms in several steps. Given an initial term t, a condition c, and a bound in the number of steps n, when all the terms reachable in at most n steps from t that fulfill c are $t_1, \ldots, t_m$, with $m > 0$, the debugger makes the following distinction:

- If the condition c defines the initial condition of the search, the tool asks questions of the form "Are the following terms all the possible solutions from t in n steps? $t_1, \ldots, t_m$," where the bound is omitted if it is unbounded. This judgment is correct if all the solutions that the user expected to obtain from t in at most n steps constitute the set $\{t_1, \ldots, t_m\}$. If $m = 0$ the debugger asks questions of the form "Did you expect that no solutions are reachable from t in n steps?," where the bound is again omitted if it is unbounded. In this case, the judgment is correct if no solutions were expected from t in at most n steps.

  In this case, if one of the solutions is reachable but it should not fulfill the search condition, the user can indicate it with `(I is not a solution .)`, where `I` is the index of the term that should not be in the set. With this answer the user indicates that the definition of the search condition is erroneous and the debugger centers on it to continue the process.

- If the condition c has been obtained from a rewrite condition $t' \Rightarrow p$, then c is just a matching condition with the pattern p, and n is unbounded. In this case, the questions have the form "Are the following terms all the reachable terms from t that match the pattern p? $t_1, \ldots, t_m$." This judgment is correct if all the terms that should be obtained from t and match the pattern p constitute the set $\{t_1, \ldots, t_m\}$. When $m = 0$ the questions have the form "Did you expect that no terms matching the pattern p can be obtained from t?," that is correct if t is expected to be final or all the terms reachable from t are not expected to match p.

These questions are only asked if the many-steps tree for missing answers is used.

In case the top-down strategy is selected, several questions will be displayed in each step. The user can then introduce answers of the form `(N : answer .)`, where `N` is the index of the question and `answer` is the same answer that would be used in the divide and query strategy for this question. Moreover, as a shortcut to answer `(yes .)` to all the questions, the debugger provides the answer

```
(all : yes .)
```

Finally, we can return to the previous state in both strategies by using the command

```
(undo .)
```

5.3. Recommendations

We recommend following some tips to ease the questions asked during the debugging process:

- It is usually more complicated to answer questions related to many steps (both in wrong and missing answers) than questions related to one step. Thus, if a specification is complex it is better to debug it with a one-step tree.

- There are some sorts that are usually final, such as `Bool` and `Nat`, so identifying them as final can avoid several tedious questions.

- If an error is found using a complex initial term, this error can probably be reproduced with a simpler one. Using this simpler term leads to easier debugging sessions.
- When facing a problem with both wrong and missing answers, it is usually better to debug the wrong answers first, because questions related to them are usually easier to answer and fixing them can also solve the missing answers problem.
- When a question is related to a set of reachable terms that contains some wrong terms, it is recommended to point out one of these terms as erroneous instead of indicating the whole set as wrong.
- When using the top-down navigation strategy, several questions are prompted. To point out one as erroneous or all of them as valid will shorten the debugging process, while pointing out one question as correct usually only eases the current set of questions. Thus, to indicate that a question is valid is only recommended for extremely complicated or large sets of questions.

If the user follows these tips and uses the trusting mechanisms it is possible to debug very large specifications, because:

- Specifications are assumed to be structured, and usually the module being debugged imports several other auxiliary modules. These modules should have been debugged before testing the current one, and thus they can be trusted (maybe some complex functions from these auxiliary modules can be suspicious).
- Specific reductions/sort inferences/rewrites usually do not apply every statement in the specification, but a small subset of them. From this point of view, debugging a large specification should not be harder than debugging a smaller one.
- The debugger assists the user through the computation, making the debugging process easier than checking by hand thousands of statements and than traversing the trace without any guide.

6. A debugging session

We describe in this section how to debug the maze example shown in Section 2.5. We recall that we have specified a module to search a path out of a labyrinth but, given a concrete labyrinth with an exit, the program is unable to find it. We start the debugging process with the command:

```
Maude> (missing {[1,1]} =>* {L:List} s.t. isSol(L:List) .)
```

With this command the debugger builds a debugging tree for missing answers in zero or more steps with the questions about solutions not prioritized, and navigated with the default divide and query strategy. The first question is:

```
Did you expect {[1,1][1,2][1,3][1,4]} to be final?

Maude> (no .)
```

Since we expected to reach the position `[2,4]` from `[1,4]`, this state should be rewritten and thus it is not final. The next question is:

```
Is this reduction (associated with the equation c2) correct?

contains([2,1][4,1][2,2][3,2][6,2][7,2][2,3][4,3][5,3][6,3][7,3][1,5][2,5][3,5][4,5][5,5]
         [6,5][8,5][6,6][8,6][6,7][6,8][7,8],[1,3]) -> false

Maude> (yes .)
```

That is, the debugger asks whether it is correct that the position `[1,3]` is not included in the wall. We answer that it is correct and the next question is:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (no .)
```

The answer is `no` because the set of terms is incomplete: we expected to find the movement to the right too. The debugger now asks:

```
Did you expect [1,4] to be final?

Maude> (yes .)
```

The answer is `yes` because we have not defined rules for positions, thus they cannot evolve. The following series of questions are:

```
Did you expect [1,3] to be final?

Maude> (yes .)

Did you expect [1,2] to be final?

Maude> (yes .)

Did you expect [1,1][1,2][1,3][1,4] to be final?

Maude> (yes .)
```

We use the same reasoning about final terms to answer these questions. The next questions are:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule n2 ?

1 [0,4]

Maude> (yes .)

Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule n3 ?

1 [1,3]

Maude> (yes .)

Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4])
with one application of the rule n1 ?

1 [1,5]

Maude> (yes .)
```

All these questions are related to the appropriate application of certain rules; these rules move the last position of the list to the left, up, and down, and thus they are correct. With this information, the debugger is able to find the bug, prompting:

```
The buggy node is:
next([1,1][1,2][1,3][1,4]) =>1 {[1,5], [1,3], [0,4]}

Either the operator next needs more rules or the conditions of the current rules are not written
in the intended way.
```

In fact, if we check the code we realize that we forgot to define the rule that specifies movements to the right. We must add the rule:

```
  rl [next4] : next(L [X,Y]) => [X + 1, Y] .
```

However, we noticed that this session required us to answer a lot of similar questions. We can enhance the behavior of the debugger by using features such as selection of final terms on the fly. For example, when the fourth question is prompted:

```
Did you expect [1,4] to be final?

Maude> (its sort is final .)

Terms of sort Pos are final.
```

we can indicate that not only this term, but all the terms with its sort (not necessarily as least one, that is, subsorts are also checked) are final. With this answer the debugging tree is pruned, and the next question is:

```
Did you expect [1,1][1,2][1,3][1,4] to be final?

Maude> (its sort is final .)

Terms of sort List are final.
```

We use this answer again, although in this case it does not reduce the number of questions. As before, the debugger finishes with the same three questions as above.

Although the number of questions has been reduced, we still face some questions that we would like to avoid about final terms. To do this, we can activate the final selection mode before starting the debugging:

```
Maude> (set final select on .)

Final select is on.
```

Once this mode is active, we can point out the sorts of the terms that will not be rewritten. Note that terms whose least sort is a subsort of the sorts selected will also be considered as final. For example, we consider in our specification the sorts `Nat` and `List` as final, which implicitly indicates that the sort `Pos`, subsort of `List`, is also final:

```
Maude> (final select Nat List .)

Sorts List Nat are now final.
```

Moreover, since we know that the rules `next1`, `next2`, and `next3` are correct, we can avoid questions about them by pointing out that the rest of the statements are suspicious with the commands:

```
Maude> (set debug select on .)

Debug select is on.

Maude> (debug select is1 is2 c1 c2 expand .)

Labels c1 c2 expand is1 is2 are now suspicious.
```

Once these options are introduced, we can start the debugging process with the same command as before:

```
Maude> (missing {[1,1]} =>* {L:List} s.t. isSol(L:List) .)

Are the following terms all the reachable terms from {[1,1][1,2][1,3]} in one step?

1 {[1,1][1,2][1,3][1,4]}

Maude> (yes .)

Are the following terms all the reachable terms from {[1,1][1,2]} in one step?

1 {[1,1][1,2][1,3]}

Maude> (yes .)
```

Given the labyrinth's limits and wall, we must go down in both cases to find the exit. The next question selected by the debugger is:

```
Did you expect that no terms can be obtained from {[1,1][1,2][1,3][1,4]} by applying the rule
expand ?

Maude> (no .)
```

As we know, the list of positions should evolve to find the exit. The debugger asks now:

```
Is this reduction (associated with the equation c2) correct?

contains([2,1][4,1][2,2][3,2][6,2][7,2][2,3][4,3][5,3][6,3][7,3][1,5][2,5]
         [3,5][4,5][5,5][6,5][8,5][6,6][8,6][6,7][6,8][7,8],[1,3]) -> false

Maude> (trust .)
```

We realize now that the equation c2 is simple enough to be trusted, although we pointed it out as suspicious at the beginning of the session. We use the command trust and the following question is prompted:

```
Is this reduction (associated with the equation c1) correct?

contains(nil,[1,5]) -> false

Maude> (trust .)
```

We consider that this equation can also be trusted. Finally, the debugger detects the problem with the next answer:

```
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
```

```
2 [1,3]
3 [0,4]

Maude> (no .)

The buggy node is:
next([1,1][1,2][1,3]) =>1 {[1,4], [1,2], [0,3]}

Either the operator next needs more rules or the conditions of the current rules are not written
in the intended way.
```

Although in this example we have used the default divide and query navigation strategy, it is also possible to use the top-down one by using:

```
Maude> (top-down strategy .)

Top-down strategy selected.
```

In this case we reduce the number of questions by considering that the sorts `Nat` and `List` are final and that the suspicious statements are the equations defining the solution, `is1` and `is2`:

```
Maude> (set final select on .)

Final select is on.

Maude> (final select Nat List .)

Sorts List Nat are now final.

Maude> (set debug select on .)

Debug select is on.

Maude> (debug select is1 is2 .)

Labels is1 is2 are now suspicious.
```

We can follow how this strategy proceeds with the trees in Figures 14 and 16. Once we introduce the debugging command, the first series of questions, which refers to the premises of the root in Figure 14 (although without some nodes, as the second one, deleted by the trusting mechanisms), is prompted:

```
Maude> (missing { [1,1] } =>* { L:List } s.t. isSol(L:List) .)

Question 1 :
Did you expect {[1,1]} not to be a solution?

Question 2 :
Are the following terms all the reachable terms from {[1,1]} in one step?

1 {[1,1][1,2]}

Question 3 :
Did you expect {[1,1][1,2]} not to be a solution?

Question 4 :
Are the following terms all the reachable terms from {[1,1][1,2]} in one step?
```

```
1 {[1,1][1,2][1,3]}

Question 5 :
Did you expect {[1,1][1,2][1,3]} not to be a solution?

Question 6 :
Are the following terms all the reachable terms from {[1,1][1,2][1,3]} in one step?

1 {[1,1][1,2][1,3][1,4]}

Question 7 :
Did you expect {[1,1][1,2][1,3][1,4]} not to be a solution?

Question 8 :
Did you expect {[1,1][1,2][1,3][1,4]} to be final?

Maude> (8 : no .)
```

The eighth question (corresponding to the root of the tree in Figure 16, marked with (†)) is erroneous because position `[2,4]` is reachable from `[1,4]` and it is free of wall, so we do not expect this term to be final. The following questions are:[7]

```
Question 1 :
Is next([1,1][1,2][1,3][1,4]) in normal form?

Question 2 :
Is Pos? the least sort of next([1,1][1,2][1,3][1,4]) ?

Question 3 :
Are the following terms all the reachable terms from next([1,1][1,2][1,3][1,4]) in one step?

1 [1,5]
2 [1,3]
3 [0,4]

Maude> (3 : no .)
```

With this answer we have pointed out the node marked (‡) in Figure 16 as wrong. Since all its children correspond to applications of equations that were trusted (`n1`, `n2`, and `n3`, while the only suspicious statements were `is1` and `is2`), this node is now a leaf and thus it corresponds to a buggy node:

```
The buggy node is:
next([1,1][1,2][1,3][1,4]) =>1 {[1,5], [1,3], [0,4]}

Either the operator next needs more rules or the conditions of the current rules are not written
in the intended way.
```

Many more examples are available at `http://maude.sip.ucm.es/debugging/`.

7. Implementation

We show here how the ideas described in the previous sections are implemented. This implementation is done in Maude itself by means of its reflective capabilities, which allow us to use Maude terms and modules

[7]Note that the child of this node, marked with (≀), is skipped because the corresponding equation has been trusted.

as data [12, Chapter 14]. Sections 7.1 and 7.2 describe the tree construction stage, where the abbreviated proof trees are constructed. The interaction with the user is explained in Section 7.3.

The complete code of the tool is contained in the file `dd.maude`, available at `http://maude.sip.ucm.es/debugging/`.

7.1. Debugging trees definition

In this section we show how to represent the debugging trees in Maude. First, we implement parametric general trees with generic data in each node. Then, we instantiate them by defining the concrete data for building our debugging trees.

The parameterized module that describes the behavior of the tree receives the theory `TRIV` (that simply requires a sort `Elt`) as parameter. We use lists of natural numbers to identify (the position of) each node. General trees are defined by means of the constructor `tree`, composed of some contents (received from the theory), the size of the tree, and a `Forest`, which in turn is a list of trees:

```
fmod TREE{X :: TRIV} is
 pr NAT-LIST .

 sorts Tree Forest .
 subsort Tree < Forest .

 op tree(_,_,_) : X$Elt Nat Forest -> Tree [ctor format (ngi o d d d d ++i n--i d)] .

 op mtForest : -> Forest [ctor format (ni d)] .
 op __ : Forest Forest -> Forest [ctor assoc id: mtForest] .
 ...
endfm
```

We use the sort `Judgment` to define the values kept in the debugging trees. When keeping reductions and memberships, we want to know the name of the statement associated with the node and the lefthand and righthand sides of the computation, or the term and sort of a membership, respectively.

```
fmod DEBUGGING-TREE-NODE is
 pr META-LEVEL .
 sort Judgment .

 op _:_->_ : Qid Term Term -> Judgment [ctor format (b o d b o d)] .
 op _:_:_ : Qid Term Type -> Judgment [ctor format (b o d b o d)] .
```

If the inferred type is the least sort, we use the special notation below:

```
 op _:ls_ : Term Type -> Judgment [ctor format (d b o d)] .
```

In the case of rewrites, we distinguish between nodes in the one-step tree and nodes in the many-steps tree:

```
 op _:_=>1_ : Qid Term Term -> Judgment [ctor format (b o d b o d)] .
 op _=>+_ : Term Term -> Judgment [ctor format (d b o d)] .
```

Since the many-steps tree is computed on demand, its leaves corresponding to one-step rewrites are kept as "frozen," and will be evaluated only if needed:

```
 op _=>f_ : Term Term -> Judgment [ctor format (d b o d)] .
```

The nodes for debugging missing answers in system modules keep the initial term and the list of possible results. We distinguish between:

- The set of reachable terms in one step:

```
op _=>1{_} : Term TermList -> Judgment [ctor format (d b o d d d)] .
```

- The set of reachable terms by applying one rule:

```
op _=>q[_]{_} : Term Qid TermList -> Judgment [ctor format (d b o d d d d d d)] .
```

- The set of reachable terms when many rewrite steps are used. In this case we also keep the bound, the pattern, the condition and a Boolean value indicating whether this search corresponds to the initial one, and thus these terms are the reachable *solutions* from the initial one, or corresponds to a search due to a rewrite condition:

```
op _~>[_]{_}s.t._&_[_] : Term Bound TermList Term Condition Bool
                         -> Judgment [ctor format (d b o d d d d d d d d d d d d)] .
```

We use the operator `sol` to indicate (the Boolean value in the fourth argument) whether a term (the first argument) matches the pattern given as second argument and fulfills the condition given as third argument. When the questions about solutions are prioritized these nodes are frozen and are expanded on demand, so it has a Boolean value (the fifth argument) indicating whether the node has been already expanded. Finally, the last Boolean value indicates whether this term is a solution of the initial search condition or it is a solution of a rewrite condition:

```
 op sol : Term Term Condition Bool Bool Bool -> Judgment [ctor format (b o)] .
```

The operator `normal` indicates that a term is in normal form with respect to the equational theory:

```
 op normal : Term -> Judgment [ctor format (r o)] .
```

Finally, we define a constant `unknown`, that will be used when the user answers `don't know` to any question:

```
 op unknown : -> Judgment [ctor] .
endfm
```

We use this module to create a view from the `TRIV` theory and we obtain our debugging trees by instantiating the module `TREE` above with this view:

```
view DebuggingTreeNode from TRIV to DEBUGGING-TREE-NODE is
 sort Elt to Judgment .
endv

fmod PROOF-TREE is
 pr TREE{DebuggingTreeNode} .
 ...
endfm
```

7.2. Debugging trees construction

In this section we describe how the different debugging trees are built. First, we describe the construction of debugging trees for wrong reductions, memberships, and rewrites and then we use them in the construction of the trees for erroneous normal forms, least sorts, and sets of reachable terms. Instead of creating the complete proof trees and then abbreviating them, we build the abbreviated proof trees directly.

7.2.1. Debugging trees for wrong reductions and memberships

The function `createTree` builds debugging trees for wrong reductions and memberships. It exploits the fact that the equations and membership axioms are both *terminating* and *confluent*. It receives the module where a wrong inference took place, a correct module (or the constant `undefMod` when no such module is provided) to prune the tree, the initial term, the (erroneous) result obtained, and the set of suspicious statement labels. It keeps the initial reduction as the root of the tree and uses an auxiliary function `createForest` that, in addition to the arguments received by `createTree`, receives the module "cleaned" of suspicious statements (by using `deleteSuspicious`), and generates the forest of abbreviated trees corresponding to the reduction between the two terms given as arguments. The transformed module is used to improve the efficiency of the tree construction, because we can use it to check whether a term reaches its final form by using only trusted statements, preventing the debugger from building a tree that will be finally empty.

```
op createTree : Module Maybe{Module} Term Term QidSet -> Tree .
ceq createTree(M, CM, T, T', QS) =
              contract(tree('root@#$% : T -> T', getOffspring*(F) + 1, F))
 if ST? := strat?(M) /\
    M' := deleteSuspicious(M, QS) /\
    F := createForest(M, M', CM, T, T', QS) .
```

We use the function `createForest` to create a forest of abbreviated trees. It receives as parameters the module where the computation took place, the transformed module (that only contains trusted statements), a correct module (possibly `undefMod`) to check the inferences, two terms representing the inference whose proof tree we want to generate, and a set of labels of suspicious equations and memberships. First, the function checks if the terms are equal, the result can be reached by using only trusted statements, or the correct module can calculate this inference; in such cases, there is no need to calculate the tree, so the empty forest is returned. Otherwise, it applies the function `createForest2`:

```
op createForest : Module Module Maybe{Module} Term Term QidSet ~> Forest .
eq createForest(OM, TM, CM, T, T', QS) =
 if T == T' or-else reduce(TM, T) == T' or-else reduce(CM, T) == T' then mtForest
 else createForest2(OM, TM, CM, T, T', QS)
 fi .
```

The function `createForest2` checks first whether the current term is of the form `if T1 then T2 else T3 fi`. In this case, the debugger evaluates `T1` and then, depending on the result, it evaluates either `T2` or `T3` following the same evaluation strategy as Maude:[8]

```
op createForest2 : Module Module Maybe{Module} Term Term QidSet ~> Forest .
eq createForest2(OM, TM, CM, 'if_then_else_fi[T1, T2, T3], T', QS)  =
   createForest(OM, TM, CM, T1, reduce(OM, T1), QS)
   if reduce(OM, T1) == 'true.Bool then
      createForest(OM, TM, CM, T2, T', QS)
   else
      if reduce(OM, T1) == 'false.Bool then
         createForest(OM, TM, CM, T3, T', QS)
      else
         createForest(OM, TM, CM, T2, reduce(OM, T2), QS)
         createForest(OM, TM, CM, T3, reduce(OM, T3), QS)
      fi
   fi .
```

[8]Note that it is possible to obtain neither `true` nor `false` when evaluating the condition. In this case, both branches will be evaluated and the term thus obtained (which is not fully evaluated) used in the rest of the computation, possibly leading to a missing answer.

Otherwise, the debugger follows the Maude innermost strategy: it first tries to fully reduce the subterms (by means of the function `reduceSubterms`), and once all the subterms have been reduced, if the result is not the final one, it tries to reduce at the top (by using the function `applyEq`), to reach the final result by *transitivity*:

```
ceq createForest2(OM, TM, CM, T, T', QS) =
                                   if T'' == T' then F
                                   else F applyEq(OM, TM, CM, T'', T', QS)
                                   fi
 if < T'', F > := reduceSubterms(OM, TM, CM, T, QS) [owise] .
```

The function `applyEq` tries to apply (at the top) one equation,[9] by using the *replacement* rule from Figure 1, with the constraint that we cannot apply equations with the `otherwise` attribute if other equations can be applied. To apply an equation we check whether the term we are trying to reduce matches the lefthand side of the equation and its conditions are fulfilled. If this happens, we obtain a substitution (from both the matching with the lefthand side and the conditions) that we can apply to the righthand side of the equation. Note that, if we can obtain the transition in the correct module, the forest is not computed:

```
op applyEq : Module Module Maybe{Module} Term Term QidSet -> Maybe{Forest} .
op applyEq : Module Module Maybe{Module} Term Term QidSet EquationSet -> Maybe{Forest} .

eq applyEq(OM, TM, CM, T, T', QS) =
       if reduce(TM, T) == T' or-else reduce(CM, T) == T' then mtForest
       else applyEq(OM, TM, CM, T, T', QS, getEqs(OM))
       fi .
```

For example, the equations without the `otherwise` attribute as applied as follows:

```
ceq applyEq(OM, TM, CM, T, T', QS, Eq EqS) =
            if in?(AtS, QS) then
            tree(label(AtS) : T -> T', getOffspring*(F) + 1, F)
            else F
            fi
 if ceq L = R if C [AtS] . := generalEq(Eq) /\
    not owise?(AtS) /\
    sameKind(OM, type(OM, L), type(OM, T)) /\
    SB := metaMatch(OM, L, T, C, 0) /\
    R' := substitute(OM, R, SB) /\
    F := conditionForest(substitute(OM, C, SB), OM, TM, CM, QS)
         createForest(OM, TM, CM, R', T', QS) .
```

where we distinguish with the function `in?(AtS, QS)` whether the equation is trusted (the attribute set does not contain a label or the label is contained in the set `QS` of trusted labels) to generate the node.

7.2.2. Debugging trees for wrong rewrites

We use a different methodology in the construction of the debugging tree for incorrect rewrites. Since these modules are not assumed to be confluent or terminating, we use the predefined breadth-first search function `metaSearchPath` to, from the initial term, find the wrong term introduced by the user, and then we use the returned trace to build the debugging tree. The trace returned by Maude when searching from `T` to `T'` is a list of steps of the form:

```
{T1, Ty1, R1} ... {Tn, Tyn, Rn}
```

[9]Since the module is assumed to be confluent, we can choose any equation and the final result should be the same.

where `Tyi` is the type of `Ti`, `T1` is the normal form of `T`, `Ri` is the rule applied to (possibly a subterm of) `Ti` to obtain `Ti+1` (which is already in normal form), and `T'` is the result of applying `Rn` to `Tn`.

The function `createRewTree`, given the module where the rewrite took place, a module with correct statements (possibly `undefMod`), the rewritten term, the result term, the set of suspicious labels, the type of tree selected (many-steps or one-step, identified by constants `ms` and `os` in the module `TREE-TYPE`), and the bound of the search in the correct module, creates the corresponding debugging tree:

```
op createRewTree : Module Maybe{Module} Term Term QidSet TreeType Bound -> Maybe{Tree} .
eq createRewTree(OM, CM, T, T', QS, os, B) = oneStepTree(OM, CM, T, T', QS, B) .
eq createRewTree(OM, CM, T, T', QS, ms, B) = manyStepsTree(OM, CM, T, T', QS, B) .
```

The function `oneStepTree` creates a complete debugging tree with only one-step rewrites in its nodes. It puts the complete judgment as the root of the tree, computes the tree for the reduction from the initial term to normal form with the function `createForest` from Section 7.2.1, and then computes the rest of the tree with the function `oneStepForest`. This corresponds to a concrete application of the *equivalence class* inference rule from Figure 1:

```
op oneStepTree : Module Maybe{Module} Term Term QidSet Bound -> Maybe{Tree} .
ceq oneStepTree(OM, CM, T, T', QS, B) =
           contract(tree(T =>+ T', getOffspring*(F) + 1, F))
 if TM := deleteSuspicious(OM, QS) /\
    T1 := reduce(OM, T) /\
    F := createForest(OM, TM, CM, T, T1, QS, strat?(OM))
         oneStepForest(OM, TM, CM, T1, T', QS, B) .
eq oneStepTree(OM, CM, T, T', QS, B) = error [owise] .
```

`oneStepForest` computes the trace of a rewrite with the predefined function `metaSearchPath` and uses it to generate a debugging tree by using `trace2forest`, which generates a forest of one-step rewrites by extracting each step of the trace and creating its corresponding tree:

```
op oneStepForest : Module Module Maybe{Module} Term Term QidSet Bound -> Maybe{Forest} .
ceq oneStepForest(OM, TM, CM, T, T', QS, B) = F
 if TR := metaSearchPath(OM, T, T', nil, '*, unbounded, 0) /\
    F := trace2forest(OM, TM, CM, TR, T', QS, B) .

eq oneStepForest(OM, TM, CM, T, T', QS, B) = noProof [owise] .
```

The many-steps debugging tree is built with the function `manyStepsTree`. This tree is computed *on demand*, so that the debugging subtrees corresponding to one-step rewrites are only generated when they are pointed out as wrong. It uses an auxiliary function `manyStepsTree2`, which also receives as a parameter the module cleaned of suspicious statements with `deleteSuspicious`:

```
op manyStepsTree : Module Maybe{Module} Term Term QidSet Bound -> Maybe{Tree} .
ceq manyStepsTree(OM, CM, T, T', QS, B) =
                contract(tree(T =>+ T', getOffspring*(F) + 1, F))
 if F := manyStepsTree2(OM, deleteSuspicious(OM, QS), CM, T, T', QS, B) .
eq manyStepsTree(OM, CM, T, T', QS, B) = error [owise] .
```

This auxiliary function uses the function `metaSearchPath` to compute the trace. If it is not empty, the forest for the reduction of the initial term to normal form is built with the function `createForest` and the tree for the rewrites is appended to this forest. If the trace consists of only one step, it is expanded with the function `stepForest`. Otherwise, the many-steps tree from the trace is built with the function `trace2tree`, that traverses the trace and creates a balanced tree from the forest of leaves obtained from it:

```
op manyStepsTree2 : Module Module Maybe{Module} Term Term QidSet Bound ~> Maybe{Forest} .
ceq manyStepsTree2(OM, TM, CM, T, T', QS, B) = F
 if {T'', Ty, R} TR := metaSearchPath(OM, T, T', nil, '*, unbounded, 0) /\
    F := createForest(OM, TM, CM, T, T'', QS, strat?(OM))
         if TR =/= nil
         then trace2tree(OM, TM, CM, {T'', Ty, R} TR, T', QS, B, mtForest, 0)
         else stepForest(OM, TM, CM, T'', T', R, QS, B, ms)
         fi .
```

If the trace is empty, only the tree for the reduction is computed:

```
ceq manyStepsTree2(OM, TM, CM, T, T', QS, B) = createForest(OM, TM, CM, T, T', QS, strat?(OM))
 if nil == metaSearchPath(OM, T, T', nil, '*, unbounded, 0) .
```

Finally, if the final term is not reachable from the initial term, an error is returned. Note that errors due to non-termination cannot be detected:

```
eq manyStepsTree2(OM, TM, CM, T, T', QS, B) = noProof [owise] .
```

7.2.3. Debugging trees for missing answers

The debugging tree for normal forms is built with the function `createMissingTree`. It receives the module where the reduction took place, a correct module, the initial term, the reached normal form, and a set of suspicious labels:

```
op createMissingTree : Module Maybe{Module} Term Term QidSet -> Tree .
ceq createMissingTree(M, CM, T, T', QS) = tree('root : T -> T'', getOffspring*(F) + 1, F)
 if TM := deleteSuspicious(M, QS) /\
    T'' := reduce(M, T') /\
    F := cleanTree*(M, false, none, createMissingForest(M, TM, CM, T, T'', QS)) .
```

The function `createMissingForest` checks whether the result can be obtained in the trusted or correct modules. When this happens, it only generates a forest proving the term is in normal form with `proveNormal`; otherwise, it uses the auxiliary function `createMissingForest2`:

```
op createMissingForest : Module Module Maybe{Module} Term Term QidSet -> Forest .
ceq createMissingForest(OM, TM, CM, T, T', QS) = F
 if T == T' or-else reduce(TM, T) == T' or-else reduce(CM, T) == T' /\
    F := proveNormal(OM, TM, CM, T', QS) .
eq createMissingForest(OM, TM, CM, T, T', QS) =
         createMissingForest2(OM, TM, CM, T, T', QS) [owise] .
```

`createMissingForest2` generates the forest for the subterms with `reduceSubtermsMissing` and then distinguishes whether the final result has been reached, proving in that case whether the term is in normal form with `proveNormal`, or not, then applying the next equation with `applyEqMissing`:

```
ceq createMissingForest2(OM, TM, CM, T, T', QS) =
                             if T'' == T' then F proveNormal(OM, TM, CM, T', QS)
                             else F applyEqMissing(OM, TM, CM, T'', T', QS)
                             fi
 if < T'', F > := reduceSubtermsMissing(OM, TM, CM, T, QS) [owise] .
```

The debugging tree for incomplete sets of reachable terms is built with the function `createMissingTree`, that receives:

- the module where the terms should be found,
- a correct module (possibly `undefMod`),

- the initial term, the pattern,
- the condition to be fulfilled,
- the bound in the number of rewrites for *wrong* rewrites,
- the number of steps that can be given in the search,
- the search type,
- the type of tree to be built (one-step or many-steps) for both wrong and missing answers,
- the set of suspicious labels,
- the set of final sorts,
- a Boolean value indicating whether the search introduced by the user was unbounded, and
- a Boolean value pointing out whether the questions about solutions are prioritized.

The forest is generated with an auxiliary function `createMissingForest` that receives, in addition to the values above, a Boolean value indicating whether the forest currently built corresponds to the initial search or to a search due to a rewrite condition, which is `true` in the first case. Once the tree has been built, the questions associated with terms that the user has declared as final are pruned with `cleanTree*`:

```
op createMissingTree : Module Maybe{Module} Term Term Condition Bound Bound SearchType
                       TreeType TreeType QidSet Bool QidSet Bool Bool -> Tree .

ceq createMissingTree(M, CM, T, PAT, C, BW, BM, ST, TTW, TTM, QS, BFS, FS, UB?, SP) =
      contract(tree(T ~>[B'] {clean(extractTerms(F))} s.t. PAT & C [true],
                                         1 + getOffspring*(F), F))
 if TM := deleteSuspicious(M, QS) /\
    T' := getTerm(metaReduce(M, T)) /\
    F := cleanTree*(M, BFS, FS, createForest(M, TM, CM, T, T', QS, strat?(M))
                          createMissingForest(labeling(M), TM, CM, T', PAT,
                                  C, BW, BM, ST, TTW, TTM, QS, FS, UB?, SP, true)) /\
    B' := if UB? then unbounded else BM fi .
```

If the tree to be built cannot evolve (the bound is `0`) and zero or more steps can be used, then we use the function `solutionTree` to create a tree that proves whether the condition is satisfied or not:

```
op createMissingForest : Module Module Maybe{Module} Term Term Condition Bound Bound SearchType
                         TreeType TreeType QidSet QidSet Bool Bool Bool -> Forest .
eq createMissingForest(OM, TM, CM, T, PAT, C, BW, 0, zeroOrMore, TTW, TTM, QS,
                       FS, UB?, SP, FST) =
       solutionTree(OM, TM, CM, T, PAT, C, BW, zeroOrMore, TTW, TTM, QS, FS, SP, FST) .
```

When the terms can still evolve (the bound is greater than `0`), we compute all the possible reachable terms in exactly one step with the function `oneStepMissingTree` and evolve each of them with `createMissingForest*`. The solutions obtained are gathered with `extractTerms`, while we check whether the current term is a valid solution with the function `solveCondition`. Finally, if the tree selected by the user is for many-steps transitions we create a root for the generated forest specifying the number of steps, while if we want one-step transitions only the forest is returned:

```
ceq createMissingForest(OM, TM, CM, T, PAT, C, BW, s(N'), zeroOrMore, TTW, TTM,
                       QS, FS, UB?, SP, FST) =
           if TTM == os then RF
           else tree(T ~>[B'] {TL''} s.t. PAT & C [FST], 1 + getOffspring*(RF), RF)
           fi
   if tree(T =>1 {TL}, N, F) := oneStepMissingTree(OM, TM, CM, T, QS, FS, BW, zeroOrMore,
                                                   TTW, TTM, SP) /\
      F' := createMissingForest*(OM, TM, CM, TL, PAT, C, BW, N', zeroOrMore,
                                 TTW, TTM, QS, FS, UB?, SP, FST) /\
      TL' := if solveCondition(OM, T, PAT, C) then T
                                              else empty fi /\
      TL'' := clean((extractTerms(F'), TL')) /\
      CF := solutionTree(OM, TM, CM, T, PAT, C, BW, zeroOrMore, TTW, TTM, QS, FS, SP, FST) /\
      RF := CF tree(T =>1 {TL}, N, F) F' /\
      B' := if UB? then unbounded else s(N') fi .
```

7.3. The debugger environment

We implement our system on top of Full Maude, a language that extends Maude with support for object-oriented specification and advanced module operations [12, Part II]. The implementation of Full Maude includes code for parsing user input and pretty-printing; storing modules, theories, and views; and transforming object-oriented modules into system modules.

To parse some input using the built-in function `metaParse`, Full Maude needs the meta-representation of the signature in which the input has to be parsed. Thus, we define the signature of the debugger in a module that extends the Full Maude signature:

```
 fmod DD-SIGNATURE is
  including FULL-MAUDE-SIGN .
  op debug_. : @Bubble@ -> @Command@ .
  op missing_. : @Bubble@ -> @Command@ .
  ...
 endfm
```

This signature is included in the meta-module `GRAMMAR` to obtain the grammar `DD-GRAMMAR`, that allows us to parse both Full Maude modules and commands together with the debugger commands:

```
 fmod META-DD-SIGN is
  inc META-FULL-MAUDE-SIGN .
  inc UNIT .
  op DD-GRAMMAR : -> FModule [memo] .
  eq DD-GRAMMAR = addImports((including 'DD-SIGNATURE .), GRAMMAR) .
  ...
 endfm
```

The module `DD-COMMAND-PROCESSING` is in charge of processing the commands dealing with suspicious statements, final sorts, and the debugging commands:

```
fmod DD-COMMAND-PROCESSING is
  pr COMMAND-PROCESSING .
  pr META-DD-SIGN .
  pr MISSING-ANSWERS-TREE .
  pr SEARCH-TYPE .
  pr PRINT .
```

For example, the parsing of the debugging command for wrong answers returns a tuple containing the generated tree, the module where the computation took place, the set of suspicious statements, and a list of quoted identifiers indicating the errors that occurred during the parsing:

```
  sort DebugTuple .
  op <_,_,_,_> : Forest Maybe{Module} QidSet QidList -> DebugTuple .
```

The parsing of the command is done in the GRAMMAR-DEB module, where the first bubble can contain either a module or just the initial term:

```
  op GRAMMAR-DEB : -> FModule [memo] .
  eq GRAMMAR-DEB = addOps(op '_->_. : '@Bubble@ '@Bubble@ -> '@Judgment@ [none] .
                          op '_:_. : '@Bubble@ '@Bubble@ -> '@Judgment@ [none] .
                          op '_=>*_. : '@Bubble@ '@Bubble@ -> '@Judgment@ [none] .,
                          addSorts('@Judgment@, GRAMMAR-RED)) .
```

The function procDebug processes a bubble and returns either a tree for the corresponding debug command or an error message. It receives the term to be parsed, a correct module (possibly undefMod), a Boolean indicating whether debug-select is on or off, the set of suspicious labels, the selected type of tree, the bound of the search in the correct module, the default module, and Full Maude's database of modules.

After finding out the kind of the debugging command (reduction, membership, or rewrite) and if a module name has been selected by the command, the function procDebug builds the appropriate tree by using the functions createTree and createRewTree explained in Section 7.2:

```
  op procDebug : Term Maybe{Module} Bool QidSet TreeType Bound ModuleExpression
                 Database -> DebugTuple .
  ...
 endfm
```

The persistent state of Full Maude's system is given by a single object of class DatabaseClass, which maintains the database of the system. We extend the Full Maude system by defining a subclass of DatabaseClass inheriting its behavior and adding new attributes to it:

```
 mod DD-DATABASE-HANDLING is
  inc DATABASE-HANDLING .
  pr DD-COMMAND-PROCESSING .
  pr TREE-PRUNING .
  pr DIVIDE-QUERY-STRATEGY .
  pr LIST{DDState} .
  pr LIST{Answer} .
  sort DDDatabaseClass .
  subsort DDDatabaseClass < DatabaseClass .
  op DDDatabase : -> DDDatabaseClass [ctor] .
```

The new attributes include, for example:

- the debugging tree, which initially is empty, and that will be traversed during the debugging process:

```
    op tree :_ : Forest -> Attribute [ctor] .
```

- the strategy to traverse the tree. The top-down strategy is represented by the constant td, whereas divide and query is represented by dq:

```
    op strategy :_ : Strat -> Attribute [ctor] .
```

- the set of labels considered suspicious:

```
    op suspicious :_ : QidSet -> Attribute [ctor gather(&)] .
```

- the set of final sorts:

```
    op finalSorts :_ : QidSet -> Attribute  [ctor gather(&)] .
```

The behavior of the debugger commands is described by means of rewrite rules that change the state of these attributes. Below we show some of the most interesting rules.

The rule `debug` starts the debugging process for wrong answers. It receives a term that will be processed with the function `procDebug` explained above. If there is no error (that is, the returned list of quoted identifiers is `nil`), the tree, the module, and the set of suspicious labels are updated with the appropriate information, while the answers given by the user so far and the previous states are reset. However, if the command was incorrect, the error is shown and the state is set to `finished`:

```
crl [debug] :
    < O : DDDC | db : DB, input : ('debug_.[T]), output : nil,
                 default : ME, tree : F, module : MM, correction : MM',
                 previousStates : LS, answers : LA, state : TS,
                 treeType : TT, currentTTW : CTTW, bound : BND, select : B,
                 suspicious : QS, currentSuspicious : QS', AtS >
 => if QIL == nil then
    < O : DDDC | db : DB, input : nilTermList, output : nil, default : ME,
                 tree : F', module : MM'', correction : MM',
                 previousStates : nil, answers : nil, state : computing,
                 treeType : TT, currentTTW : TT, bound : BND, select : B,
                 suspicious : QS, currentSuspicious : QS'', AtS >
    else
    < O : DDDC | db : DB, input : nilTermList, output : QIL, default : ME,
                 tree : mtForest, module : MM, correction : MM',
                 previousStates : nil, answers : nil, state : finished,
                 treeType : TT, currentTTW : CTTW, bound : BND, select : B,
                 suspicious : QS, currentSuspicious : QS', AtS >
    fi
 if < F', MM'', QS'', QIL > := procDebug(T, MM', B, QS, TT, BND, ME, DB) .
```

When a correct module expression is introduced, `correct-module` keeps the associated module if it exists, and shows an error message otherwise:

```
crl [correct-module] :
    < O : DDDC | db : DB, input : ('correct`module_.[T]), output : nil, correction : MM, AtS >
 => if M? :: Module
    then < O : DDDC | db : DB, input : nilTermList, output : ('\n add-spaceR(printME(ME)) '\b
                                                 'selected 'as 'correct 'module. '\o '\n),
                      correction : M?, AtS >
    else < O : DDDC | db : DB, input : nilTermList, output : ('\n '\r 'Error: '\o getMsg(M?)),
                      correction : MM, AtS >
    fi
 if ME := parseModExp(T) /\
    M? := if compiledModule(ME, DB)
          then getFlatModule(ME, DB)
          else getFlatModule(modExp(evalModExp(ME, DB)), database(evalModExp(ME, DB)))
          fi .
```

The rule `top-down-strategy` fixes the value of the navigation strategy to `td`, and changes the state to `computing` if the debugging has not finished to show the appropriate question:

```
rl [top-down-strategy] :
   < O : DDDC | input : ('top-down`strategy`..@Command@), output : nil,
                strategy : STRAT, state : TS, AtS >
=> < O : DDDC | input : nilTermList, output : ('\n '\b 'Top-down 'strategy
                                              'selected. '\o '\n),
                strategy : td, state : if TS == finished then TS
                                                         else computing fi, AtS > .
```

In the top-down strategy, when the user introduces the identifier of a wrong question, the debugger updates the list of answers and the previous states, and changes the current tree by the appropriate child of the root:

```
crl [top-down-traversal-no] :
    < O : DDDC | input : ('_:`no`.['token[T]]), strategy : td, tree : PT,
                 previousStates : LS, answers : LA, state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : td, tree : PT',
                 previousStates : LS < nil, PT, td >,
                 answers : LA getAnswer(PT', wrong), state : computing, AtS >
 if UPT := removeUnknownChildren(PT) /\
    N := downNat*(T) /\
    N > 0 /\
    N <= size(getForest(UPT, nil)) /\
    PT' := getSubTree(UPT, sd(N, 1)) .
```

where the function `getAnswer` constructs an answer given the current node and the answer given by the user.

The rule `missing-wrong` is used when, while debugging missing answers with the divide and query strategy, the user points out that a certain term is not reachable. The rule checks that the current question is related to an inference of a set of terms with `setInference?` and that the selected question points to one of these terms, and then creates the debugging tree for wrong answers with `createRewTree`:

```
crl [missing-wrong] :
    < O : DDDC | input : ('_is`wrong`.['token[T]]), strategy : dq, tree : PT,
                 current : NL, previousStates : LS, answers : LA, state : waiting,
                 currentSuspicious : QS, bound : BND, module : M, correction : MM,
                 currentTTW : TT, AtS >
 => < O : DDDC | input : nilTermList, strategy : dq, tree : PT',
                 current : NL, previousStates : LS < NL, PT, dq >,
                 answers : LA getAnswer(getSubTree(PT, NL), wrong),
                 state : computing, currentSuspicious : QS, bound : BND,
                 module : M, correction : MM, currentTTW : TT, AtS >
 if N := downNat*(T) /\
    setInference?(getContents(PT, NL)) /\
    N > 0 /\
    N <= numTermsInRootSet(getSubTree(PT, NL)) /\
    T1 := getFirstTerm(getSubTree(PT, NL)) /\
    T2 := getWrongTerm(getSubTree(PT, NL), N) /\
    PT' := createRewTree(labeling(M), MM, T1, T2, QS, TT, BND) .
```

When the divide and query strategy is selected and the user decides to trust a statement, the current subtree is deleted and the resulting tree is pruned in order to delete the nodes associated with the trusted statement:

```
crl [divide-query-traversal] :
    < O : DDDC | input : ('trust`..@Command@), strategy : dq, tree : PT,
                 current : NL, previousStates : LS, answers : LA,
                 state : waiting, AtS >
 => < O : DDDC | input : nilTermList, strategy : dq, tree : PT', current : NL,
                 previousStates : LS < NL, PT, dq >,
                 answers : LA getAnswer(getSubTree(PT, NL), right),
                 state : computing, AtS >
if Q := getLabel(PT, NL) /\
   PT' := prune(deleteSubTree(PT, NL), Q) .
```

In the divide and query strategy, when the user indicates that the sort of a certain term is final on the fly the rule `sort-final` is applied. It checks that the question is related to final terms with the function `finalQuestion?` and then prunes all the tree with the function `pruneFinalSort`:

```
 crl [sort-final] :
     < O : DDDC | input : ('its`sort`is`final`..@Command@), output : nil,
                  tree : PT, current : NL, module : M, state : waiting, AtS >
  => < O : DDDC | input : nilTermList, output : ('\n '\b 'Terms 'of 'sort '\o Ty
                                                '\b 'are 'final. '\o '\n),
                  tree : PT', current : NL, module : M, state : computing, AtS >
  if finalQuestion?(getContents(PT, NL)) /\
     T := getFirstTerm(getSubTree(PT, NL)) /\
     Ty := getType(metaReduce(M, T)) /\
     PT' := pruneFinalSort(M, Ty, PT) .
```

When the user decides to switch the select mode on to use a subset of the labeled statements as suspicious, the `select` attribute is set to `true`:

```
 rl [select] :
    < O : DDDC | input : ('set`debug`select`on`..@Command@), select : B,
                 output : nil, AtS >
 => < O : DDDC | input : nilTermList, select : true,
                 output : ('\n '\b 'Debug 'select 'is 'on. '\o '\n), AtS > .
```

The module `DD` manages the introduction of data by the user and the output of the debugger's answers. Full Maude uses the input/output facility provided by the `LOOP-MODE` module [12, Chapter 17], which consists of an operator `[_,_,_]` with an input stream (the first argument), an output stream (the third argument), and a state (given by its second argument):

```
mod DD is
 inc DD-DATABASE-HANDLING .
 inc LOOP-MODE .
 inc META-DD-SIGN .
 op o : -> Oid .
 --- State for LOOP mode:
 subsort Object < State .
 op init-debug : -> System .

 rl [init] :
    init-debug
 => [nil, < o : DDDatabase | input : nilTermList, output : nil, init-state >, dd-banner] .
```

The rule `in` below parses the data introduced by the user, which appears in the first argument of the loop, in the module `DD-GRAMMAR` and introduces it in the `input` attribute if it is correctly built:

```
 crl [in] :
     [QIL, < O : X@Database | input : nilTermList, Atts >, QIL']
  => [nil,
      < O : X@Database | input : getTerm(metaParse(DD-GRAMMAR, QIL, '@Input@)), Atts >,
      QIL']
  if QIL =/= nil /\
     metaParse(DD-GRAMMAR, QIL, '@Input@) : ResultPair .
```

The rule `out` is in charge of printing the messages from the debugger by moving the data in the `output` attribute to the third component of the loop:

```
 rl [out] :
    [QIL, < O : X@Database | output : (QI QIL'), Atts >, QIL'']
 => [QIL, < O : X@Database | output : nil, Atts >, (QIL'' QI QIL')] .
endm
```

8. Conclusions and future work

We have presented in this paper a declarative debugger for Maude specifications. The debugging trees used in the debugging process are obtained from an abbreviation of a proper calculus whose adequacy for debugging has been proved. This work comprises our previous work on wrong [30, 8, 34] and missing answers [32, 31], and provides a powerful and complete debugger for Maude specifications. Moreover, we also provide a graphical user interface that eases the interaction with the debugger and allows one to traverse the debugging tree with more freedom [29, 33]. The tree construction, its navigation, and the user interaction (excluding the GUI) have all been implemented in Maude itself. For more information, see `http://maude.sip.ucm.es/debugging`.

We plan to add new navigation strategies like the ones shown in [36] that take into account the number of different potential errors in the subtrees, instead of their size. Moreover, the current version of the tool allows the user to introduce a correct but maybe incomplete module in order to shorten the debugging session. We intend to add a new command to introduce *complete* modules, which would greatly reduce the number of questions asked to the user. Finally, we also plan to create a test generator to test Maude specifications and debug the erroneous tests with the debugger.

References

[1] M. Alpuente, M. Comini, S. Escobar, M. Falaschi, and S. Lucas. Abstract diagnosis of functional programs. In M. Leuschel, editor, *Logic Based Program Synthesis and Transformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2002.

[2] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[3] B. Braßel. The debugger B.I.O. `http://www-ps.informatik.uni-kiel.de/currywiki/tools/oracle_debugger`.

[4] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1):386–414, 2006.

[5] R. Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In S. Antoy and M. Hanus, editors, *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia*, pages 8–13. ACM Press, 2005.

[6] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In F. J. López-Fraguas, editor, *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.

[7] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. Declarative debugging of membership equational logic specifications. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 174–193. Springer, 2008.

[8] R. Caballero, N. Martí-Oliet, A. Riesco, and A. Verdejo. A declarative debugger for Maude functional modules. In G. Roşu, editor, *Proceedings of the Seventh International Workshop on Rewriting Logic and its Applications, WRLA 2008*, volume 238(3) of *Electronic Notes in Theoretical Computer Science*, pages 63–81. Elsevier, 2009.

[9] R. Caballero and M. Rodríguez-Artalejo. DDT: A declarative debugging tool for functional-logic languages. In Y. Kameyama and P. J. Stuckey, editors, *Proceedings of the 7th International Symposium on Functional and Logic Programming, FLOPS 2004, Nara, Japan*, volume 2998 of *Lecture Notes in Computer Science*, pages 70–84. Springer, 2004.

[10] R. Caballero, M. Rodríguez-Artalejo, and R. del Vado Vírseda. Declarative diagnosis of missing answers in constraint functional-logic programming. In J. Garrigue and M. V. Hermenegildo, editors, *Proceedings of the 9th International Symposium on Functional and Logic Programming, FLOPS 2008, Ise, Japan*, volume 4989 of *Lecture Notes in Computer Science*, pages 305–321. Springer, 2008.

[11] O. Chitil and Y. Luo. Structure and properties of traces for functional programs. In I. Mackie, editor, *Proceedings of the Third International Workshop on Term Graph Rewriting, TERMGRAPH 2006*, volume 176 of *Electronic Notes in Theoretical Computer Science*, pages 39–63. Elsevier, 2007.

[12] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[13] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[14] T. Davie and O. Chitil. Hat-Delta: One right does make a wrong. In *7th Symposium on Trends in Functional Programming, TFP 06*, 2006.

[15] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320. North-Holland, 1990.

[16] D. Insa and J. Silva. An algorithmic debugger for Java. In M. Lanza and A. Marcus, editors, *Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010*, pages 1–6. IEEE Computer Society, 2010.

[17] D. Insa, J. Silva, and A. Riesco. Balancing execution trees. In V. M. Gulías, J. Silva, and A. Villanueva, editors, *Proceedings of the 10th Spanish Workshop on Programming Languages, PROLE 2010*, pages 129–142. Ibergarceta Publicaciones, 2010.

[18] J. W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[19] W. Lux. *Münster Curry User's Guide, 0.9.10 edition, 2006.*

[20] I. MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.

[21] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[22] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[23] L. Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.

[24] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[25] L. Naish, P. W. Dart, and J. Zobel. The NU-Prolog debugging environment. Technical Report 88/31, Department of Computer Science, University of Melbourne, Australia, June 1989. `http://www.cs.mu.oz.au/~lee/papers/nude/`.

[26] H. Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[27] B. Pope. Declarative debugging with Buddha. In V. Vene and T. Uustalu, editors, *Advanced Functional Programming - 5th International School, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer, 2005.

[28] B. Pope. *A Declarative Debugger for Haskell.* PhD thesis, The University of Melbourne, Australia, 2006.

[29] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. A declarative debugger for Maude specifications - User guide. Technical Report SIC-7-09, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, 2009. `http://maude.sip.ucm.es/debugging`.

[30] A. Riesco, A. Verdejo, R. Caballero, and N. Martí-Oliet. Declarative debugging of rewriting logic specifications. In A. Corradini and U. Montanari, editors, *Recent Trends in Algebraic Development Techniques, WADT 2008*, volume 5486 of *Lecture Notes in Computer Science*, pages 308–325. Springer, 2009.

[31] A. Riesco, A. Verdejo, and N. Martí-Oliet. Declarative debugging of missing answers for Maude specifications. In C. Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 277–294. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2010.

[32] A. Riesco, A. Verdejo, and N. Martí-Oliet. Enhancing the debugging of Maude specifications. In P. C. Ölveczky, editor, *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 226–242. Springer, 2010.

[33] A. Riesco, A. Verdejo, and N. Martí-Oliet. A complete declarative debugger for Maude. In M. Johnson and D. Pavlovic, editors, *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology, AMAST 2010*, volume 6486 of *Lecture Notes in Computer Science*, pages 216–225. Springer, 2011.

[34] A. Riesco, A. Verdejo, N. Martí-Oliet, and R. Caballero. A declarative debugger for Maude. In J. Meseguer and G. Roşu, editors, *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology, AMAST 2008*, volume 5140 of *Lecture Notes in Computer Science*, pages 116–121. Springer, 2008.

[35] E. Y. Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1983.

[36] J. Silva. A comparative study of algorithmic debugging strategies. In G. Puebla, editor, *Logic-Based Program Synthesis and Transformation*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[37] J. Silva. *Debugging Techniques for Declarative Languages: Profiling, Program Slicing, and Algorithmic Debugging.* PhD thesis, Technical University of Valencia, June, 2007.

[38] A. Tessier and G. Ferrand. Declarative diagnosis in the CLP scheme. In P. Deransart, M. V. Hermenegildo, and J. Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.

[39] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

Appendix A. Proofs

Proposition 1. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and let $\mathcal{T} = \mathcal{T}_{\Sigma/E',R'}$ be any Σ-term model. If a statement $e \Rightarrow e'$ (respectively $e \rightarrow e'$, $e : s$) can be deduced using the semantic calculus rules* reflexivity, transitivity, congruence, equivalence class, *or* subject reduction *using premises that hold in $\mathcal{T}$, then $\mathcal{T} \models e \Rightarrow e'$ (respectively $\mathcal{T} \models e \rightarrow e'$, $\mathcal{T} \models e : s$).*

Proof. The result is a direct consequence of the definition of satisfaction of rewrite theories. For instance we check the result for the *transitivity* rules $\mathsf{Tr}_{\Rightarrow}$ and $\mathsf{Tr}_{\rightarrow}$, and for the *subject reduction* rule SRed:

- $\mathsf{Tr}_{\Rightarrow}$. Suppose that $\mathcal{T} \models e_1 \Rightarrow e'$ and $\mathcal{T} \models e' \Rightarrow e_2$. Then $[\![e_1]\!]_{\mathcal{A}} \rightarrow^*_{R'/E'} [\![e']\!]_{\mathcal{A}}$, $[\![e']\!]_{\mathcal{A}} \rightarrow^*_{R'/E'} [\![e_2]\!]_{\mathcal{A}}$. Since $\rightarrow^*_{R'/E'}$ is compositional, $[\![e_1]\!]_{\mathcal{A}} \rightarrow^*_{R'/E'} [\![e_2]\!]_{\mathcal{A}}$, i.e., $\mathcal{T} \models e_1 \Rightarrow e_2$.
- $\mathsf{Tr}_{\rightarrow}$. If $\mathcal{T} \models e_1 \rightarrow e'$ and $\mathcal{T} \models e' \rightarrow e_2$ then $[\![e_1]\!]_{\mathcal{A}} = [\![e']\!]_{\mathcal{A}}$ and $[\![e']\!]_{\mathcal{A}} = [\![e_2]\!]_{\mathcal{A}}$. Therefore $[\![e_1]\!]_{\mathcal{A}} = [\![e_2]\!]_{\mathcal{A}}$ and $\mathcal{T} \models e_1 \rightarrow e_2$.
- SRed. If $\mathcal{T} \models e \rightarrow e'$ and $\mathcal{T} \models e' : s$, then $[\![e]\!]_{\mathcal{A}} = [\![e']\!]_{\mathcal{A}}$ and $[\![e']\!]_{\mathcal{A}} \in A_s$, and hence $[\![e]\!]_{\mathcal{A}} \in A_s$.

The *reflexivity*, *congruence*, and *equivalence class* rules are checked analogously. □

Theorem 1. *The calculus of Figures 4, 5, 6, and 7 is correct.*

Proof. By induction over proof trees; we distinguish cases over the different kinds of judgments:

- $\mathit{adequateSorts}(\kappa) \rightsquigarrow \Theta$ is correct. Given a kind-substitution κ, when it has the variables of the appropriate sorts only the rule $\mathsf{SubsCond}$ can be applied and the set containing κ is returned. If the matching fails, AS_2 has to be applied and the empty substitution set is returned, being the judgment correct.
- $[C, \theta] \rightsquigarrow \Theta$ is correct. We distinguish subcases over the different kinds of conditions:
  - $C \equiv t_1 = t_2$. Since we work with admissible conditions, we know that $\theta(t_1)$ and $\theta(t_2)$ are ground, and thus the only possible substitution that can be included in Θ is θ. If the condition is fulfilled only rule EqC_1 can be used, and $\{\theta\}$ is returned, which is correct. Otherwise, only EqC_2 can be used, returning now the empty set which is again correct.
  - $C \equiv t_1 := t_2$. We assume that $\theta(t_2) \rightarrow_{norm} t'$ so, given the complete set of kind-substitutions, we restrict them to those that are substitutions, thus returning the correct set.
  - $C \equiv t : s$. Like in equational conditions, $\theta(t)$ is ground and the resulting set can only contain θ. If the condition is fulfilled only MbC_1 can be applied and the set obtained is correct. Analogously, if the condition does not hold, only MbC_2 can be used and the correct result is the empty set.
  - $C \equiv t_1 \Rightarrow t_2$. We assume that the set of reachable terms from $\theta(t_1)$ that match $\theta(t_2)$ is correct, and thus by definition the set computed by rule RlC, the only one applicable here, is correct.
- $\langle C, \Theta \rangle \rightsquigarrow \Theta'$ is correct. The only rule that deals with this judgment is $\mathsf{SubsCond}$. Assuming the premises correct, the conclusion is also correct.
- $\mathit{disabled}(e, t)$ is correct. The only rule that deals with this judgment is Dsb. Assuming the premises correct there are no substitutions satisfying the conditions and making the lefthand side of the equation or membership match the term, so it cannot be applied and the judgment is correct.
- $t \rightarrow_{red} t'$ is correct. In this case two rules can be used: Rdc_1 and Rdc_2. The first one covers reductions at the top, while the second one covers reductions on the subterms, thus dealing with all possibilities. Assuming the premises correct, in the first case we verify that one step is used because it corresponds to the application of one equation, while in the second one we check with the side condition that at least one step is used and thus the judgment is correct.

- $t \rightarrow_{norm} t'$ is correct. The rules that deal with this case are Norm and NTr, that distinguish whether the term is already in normal form or can be further reduced. In the first case if we assume the premises correct then the term is in normal form and then the same term has to be returned. In the second case, assuming the premises correct and a confluent specification, the conclusion is correct.
- *fulfilled*$(\mathcal{C}, t)$. This judgment is correct when there exists a substitution that makes $\mathcal{C}$ with the hole $\circledast$ filled by t hold. Rule Fulfill, the only one that can be used to prove this predicate, states this fact and thus the judgment is correct.
- *fails*$(\mathcal{C}, t)$. This judgment is correct when $\mathcal{C}$ with t filling its hole $\circledast$ cannot be satisfied. Since the only rule that can be used for this predicate is Fail and the premise indicates that the set of substitutions that fulfill the condition is empty, the judgment is correct.
- $t \Rightarrow^{q} S$. This judgment is only computed with rule Rl. By hypothesis, all the substitutions that fulfill the conditions and make t match the lefthand side of the rule are in Θ_k, thus by definition the union of the application of all the substitutions in Θ_k to the lefthand side of the rule generate the set we are looking for and the judgment is correct.
- $t \Rightarrow^{top} S$. This judgment is only computed with rule Top. First, we notice that the rules in $\{q_1, \ldots, q_l\}$ are the only ones that can be applied to t (it does not match the lefthand side of the rest of the rules) and thus the correctness is not affected by this selection. We know by hypothesis that each S_i, the set of reachable terms obtained from t with the rule q_i, is correct and hence the union of all these sets is by definition the set of reachable terms by rewriting at the top and the judgment is correct.
- $t \Rightarrow_1 S$. This judgment is only computed with rule Stp. By hypothesis, we know that S_t contains the set of reachable terms obtained by rewriting t at the top, while S_i contains the reachable terms in one step from t_i. Since the set of reachable terms in one step from t is the union of the terms obtained by one rewriting at the top and the set created by substituting each subterm by all the reachable terms in one step from it, the judgment is correct.
- $t \rightsquigarrow_n^{\mathcal{C}} S$. For this judgment, rule Red$_1$ can always be applied. Since we work with a coherent theory, the set of reachable terms from both t and t_1 are the same, while t_2 and t' are in the same equivalence class and thus are equal modulo E.

  When $n = 0$, rules Rf$_1$ or Rf$_2$ are used and the result is straightforward.

  If $n > 0$ and the term fulfills the condition, rule Tr$_1$ is applied. Since the condition holds, the result set must contain t, that is added in the conclusion of the rule. Moreover, the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps, that is, the union of the S_i is the set of reachable terms in at least one step and at most n, and thus the union of this set with the singleton set $\{t\}$ creates a correct set for this judgment. Analogously, when $n > 0$ and the condition does not hold, rule Tr$_2$ is applied.

$\square$

Theorem 2. *The calculus of Figure 12 is correct.*

Proof.

- $t \rightsquigarrow !_n^{\mathcal{C}} S$. For this judgment, rule Red$_2$ can always be applied. Since we work with a coherent theory, the set of reachable terms from both t and t_1 are the same, while t_2 and t' are equal modulo E.

  When $n = 0$, rules Rf$_3$, Rf$_4$, and Rf$_5$ can be used. If t is not final only Rf$_5$ can be used and, since no more steps are allowed, the empty set of results is returned, which is correct by definition. If t is final we have to check whether the term fulfills the condition; if the condition holds only Rf$_3$ can be used and hence the singleton set consisting of the term is returned, while if the condition fails Rf$_4$ is applied and the empty set is returned. In both cases the result is correct by definition.

When $n > 0$ rules Rf_3, Rf_4, and Tr_3 can be used. If the term is final, Rf_3 and Rf_4 are applied and the result holds as in the previous case. If the term is not final, then Tr_3 is applied; the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps, that is, the union of the S_i is the set of reachable terms in at least one step and at most n and, since the current term cannot be a solution because it is not final, the judgment is correct.

- $t \rightsquigarrow +_n^{\mathcal{C}} S$. We distinguish cases over n:

  When $n = 0$, only rule Rf_6 can be applied; since the judgment requires at least one step, the set of reachable terms is empty by definition.

  When $n > 0$, rule Tr_4 is applied. Since $t \rightarrow t'$ and the specification is coherent, we know that the set of reachable terms from both t and t' is the same; the terms $t_1, \ldots, t_k$ are the reachable terms from t in exactly one step, while S_i is the set of reachable terms from t_i in zero or more steps (note that the judgments in the premises are different from the one in the conclusion), that is, the union of the S_i is the set of reachable terms in at least one step and at most n and hence the judgment is correct.

$\square$

Proposition 2. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, C an atomic condition, θ an admissible substitution, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If adequateSorts$(\kappa) \rightsquigarrow \Theta$, $[C, \theta] \rightsquigarrow \Theta$, or $\langle C, \Theta \rangle \rightsquigarrow \Theta'$ can be deduced using the rules from Figure 4 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models$ adequateSorts$(\kappa) \rightsquigarrow \Theta$, $\mathcal{T}_{\Sigma/E',R'} \models [C, \theta] \rightsquigarrow \Theta$, and $\mathcal{T}_{\Sigma/E',R'} \models \langle C, \Theta \rangle \rightsquigarrow \Theta'$, respectively.*

Proof. We apply the definition of satisfaction for each rule:

EqC_1 From the premises we deduce that $[\theta(t_1)]_{E'} = [\theta(t_2)]_{E'}$, that is, the condition is satisfied with the current substitution θ. Since θ already binds all the variables in the condition, it cannot be extended and θ itself is the result.

EqC_2 From the premises we deduce that $[\theta(t_1)]_{E'} \neq [\theta(t_2)]_{E'}$, thus the condition fails and there is no substitution that could satisfy it.

PatC We know that $[\theta(t_2)]_{E'} = [t']_{E'}$ and that matching conditions can have variables in its lefthand side that are not bound in θ. Thus, the substitution is extended with all the substitutions θ' that match t' and, since t' is equal (modulo E') to $\theta(t_2)$ by hypothesis, these are all the substitutions that satisfy the condition.

AS_1 We know that the terms in the kind-substitution have the adequate sort, so it is a substitution.

AS_2 When one term in the kind-substitution has an incorrect sort the match fails.

MbC_1 We know that the condition is fulfilled and θ binds all the variables, therefore it cannot be extended and the single substitution that verifies the condition is θ itself.

MbC_2 Similarly to EqC_2, we know by hypothesis that the condition does not hold, thus there is no substitution able to satisfy it and the empty set of substitutions is computed.

RlC In this case θ can be extended because rewrite conditions can contain new variables in their righthand side. We assume that S contains all the terms reachable from $\theta(t_1)$ that match the pattern t_2, and then use it to extend θ with all the substitutions θ' that bind the new variables in t_2 to match the terms in S, obtaining by definition all the substitutions that verify the condition.

$\mathsf{SubsCond}$ We assume that, for each θ_i, $1 \leq i \leq n$, we obtain the set of substitutions S_i that extend $[C, \theta_i]$. By definition, $\langle C, \{\theta_1, \ldots, \theta_n\} \rangle$ computes the set of substitutions that extend any $[C, \theta_i]$, i.e., the union of the S_i, thus the inference is sound.

□

Proposition 3. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory and φ a judgment deduced with the inference rules* Dsb, Rdc$_2$, *or* NTr *from Figure 5 from premises that hold in $\mathcal{T}_{\Sigma/E',R'}$. Then also $\mathcal{T}_{\Sigma/E',R'} \models \varphi$.*

Proof. We apply the definition of satisfaction for each rule:

Dsb If the matching with the lefthand side and the conditions cannot be satisfied, then it is straightforward to see that the statement cannot be applied.

Rdc$_2$ The substitution of a subterm by its normal form is correct if the normal form is correct.

NTr Since the specification is confluent, we can use any equations to evolve a term and then compute the normal form from this new term.

□

Proposition 4. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow_0^{\mathcal{C}} S$ can be deduced using rules* Rf$_1$ *or* Rf$_2$ *from Figure 6 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_0^{\mathcal{C}} S$.*

Proof. We apply the definition of satisfaction for each rule:

Rf$_1$ We know by hypothesis that the term t fulfills the condition thus, by definition, the set of reachable terms in zero steps is the singleton set with t as single element.

Rf$_2$ In a similar way to the case above, if the condition does not hold with the term t, then the set of reachable terms is empty.

□

Proposition 5. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If $t \rightsquigarrow_n^{\mathcal{C}} S$ or $t \Rightarrow_1 S$ can be deduced by means of the rules in Figure 7 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow_n^{\mathcal{C}} S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \Rightarrow_1 S$, respectively.*

Proof. We apply the definition of satisfaction for each rule:

Tr$_1$ We know that the condition is fulfilled by t, that t in exactly one step is rewritten to the set $\{t_1, \ldots, t_k\}$, and that each of these terms is rewritten in at most n steps to $S_1, \ldots, S_k$. Since $\{t_1, \ldots, t_k\}$ have been obtained in one step, the terms in $S_1, \ldots, S_k$ have been computed in at most $n+1$ steps and in at least 1 step. Since we are looking for the solutions in zero or more steps, we have to compute the union of these sets with the set of reachable terms in zero steps, that in this case is the singleton set containing the term t itself, because we are assuming it fulfills the condition. Thus, the inference is sound.

Tr$_2$ Analogous to the case above.

Stp We assume that all the possible rewrites in exactly one step at the top of $f(t_i)$, $0 \le i \le m$, lead to the set S_t and that all the reachable terms in exactly one step of each subterm t_i form the set S_i. By definition, all the reachable terms in exactly one step is the union of the set of all the terms obtained by rewrites at the top and the sets built by substituting each subterm by each reachable term from it (only one subterm is substituted at the same time), so the inference is sound.

Red$_1$ Since we know that $t \to t_1$, by coherence the same reachable terms are obtained from t and t_1. Moreover, since $t_2 =_{E'} t'$ we can substitute t_2 by t' and the set remains unchanged.

□

Proposition 6. *Let $\mathcal{R} = (\Sigma, E, R)$ be a rewrite theory, $\mathcal{C}$ an admissible condition, n a natural number, and $\mathcal{T}_{\Sigma/E',R'}$ any Σ-term model. If a statement $t \rightsquigarrow!^{\mathcal{C}}_n S$ or $t \rightsquigarrow+^{\mathcal{C}}_n S$ can be deduced by means of the rules in Figure 12 using premises that hold in $\mathcal{T}_{\Sigma/E',R'}$, then also $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow!^{\mathcal{C}}_n S$ or $\mathcal{T}_{\Sigma/E',R'} \models t \rightsquigarrow+^{\mathcal{C}}_n S$, respectively.*

Proof.

Rf_3 In this case we know that the term fulfills the condition and that it is final, so by definition the set of final reachable terms consists exactly of the term itself.

Rf_4 If the term is final but it does not satisfy the condition, then the set of reachable states is empty by definition.

Rf_5 If no more steps can be used and the term is not final, the set of reachable terms is empty by definition.

Tr_3 We know that the term is not final, so we can split the search into two different searches, one in one step that leads to $\{t_1, \ldots, t_k\}$, and another in n steps from these terms, that we know generate the sets $S_1, \ldots, S_k$. Thus, the result is the union of these sets.

Red_2 Analogous to Red_1 in Proposition 5.

Rf_6 By definition the relation requires at least one step, thus if only zero steps are available the result is the empty set.

Tr_4 First, we know that $t \to t'$, hence, by coherence, the same reachable terms are obtained from t and t'. Again, we distinguish the first step of the search, that leads to $\{t_1, \ldots, t_k\}$, and the next n steps. Since the terms in this second phase of the search have already evolved one step, the single requirement is to fulfill the condition, and thus the union of the sets obtained with the relation for zero or more steps has to be the result.

□

Proposition 8. *Let N be a buggy node in some proof tree in the calculus of Figures 1, 4, 5, 6, 7, and 12 w.r.t. an intended interpretation $\mathcal{I}$. Then:*

1. *N corresponds to the consequence of an inference rule in the first column of Table 3.*
2. *The error associated to N can be obtained from the inference rule as shown in the second column of Table 3.*

Proof. The first item is a straightforward consequence of Propositions 1, 2, 3, 4, 5, and 6: N buggy means N invalid with all its children valid, and these are the only possible inference rules at N.

For the second property we study each inference rule separately:

$\mathsf{Rep}_\to$ In this case the associated equation is wrong as a direct consequence of having a wrong statement instance: N is invalid in $\mathcal{I}$, while the previous conditions, which state the validity of the statements in the equation condition instance, correspond to the premises of the $\mathsf{Rep}_\to$ inference rule (see Figure 1), which are valid in $\mathcal{I}$ because N is buggy.

$\mathsf{Rep}_\Rightarrow$ **and** Mb Analogous to the case above.

Rdc_1 In this case it is possible to have an erroneous result when the conditions hold. The reason is that the equation can be wrong, and thus we would have a wrong equation instance.

Norm If the conclusion of this rule is erroneous but its premises hold this means that the specification does not have all the required equations, that is, an error in this node is associated with a missing equation.

Ls Similarly to the case above, if the conclusion of this rule is wrong while its premises hold this means that the specification lacks some membership, that is, an error in this node is associated with a missing membership.

Fulfill If this node is buggy then there exists a substitution that satisfies the condition but the condition should not hold, thus we have a wrong condition. In this case the condition in the buggy node is pointed out as the error in the specification.

Fail In this case the set of substitutions that fulfill the condition is empty but the condition should hold, so the node is associated with a wrong condition. As in the case above, the error in the specification is related to the condition in the buggy node.

Top When this node is buggy all the possible rules have been applied at the top and their results are correct, but the union of these terms does not lead to all the intended reachable terms by rewriting the term at the top, so this node is related to a missing rule. In this case, we will point to the operator at the top of the term in the lefthand side of the buggy node as incompletely defined.

Rl The nodes computing the set of substitutions that fulfill the condition of the rule are correct, but once the righthand side of the rule is instantiated with these substitutions there are reachable terms in the intended interpretation that are not in this set. Thus, in this case the buggy node is associated with a wrong rule and the rule applied in the node is pointed out as buggy.

□

Lemma 1. *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 4, 5, 6, 7, and 12 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root N of T is invalid in $\mathcal{I}$. Then:*

(a) If T contains only one node, then $APT'(T) = \{T\}$.
(b) There is a $T' \in APT'(T)$ such that T' has an invalid root.

Proof. If T contains only one node N then N is an invalid node without children and therefore buggy. By Proposition 8 the inference step proving this node must be $\mathsf{Rep}_{\rightarrow}$, Mb, $\mathsf{Rep}_{\Rightarrow}$, Rdc_1, Norm, Fulfill, Fail, Ls, Rl, or Top. In all these cases the rule $(\mathbf{APT}_{10})$ of Figure 13 must be applied and the result holds, since it returns a singleton set with the same root.

The second item can be proved by induction on the number of nodes of T, which we denote as $n(T)$. If $n(T) = 1$ the property is straightforward from the part (a) above because $T \in APT'(T)$. If $n(T) > 1$ we distinguish cases depending on the rule for APT' that can be applied at the root of T:

- If it is either $(\mathbf{APT}_2)$, $(\mathbf{APT}_3)$, $(\mathbf{APT}_4)$, $(\mathbf{APT}_5)$, $(\mathbf{APT}_6)$, $(\mathbf{APT}_7)$, $(\mathbf{APT}^m_8)$, $(\mathbf{APT}^m_9)$, or $(\mathbf{APT}_{10})$ the result holds directly because the result is a singleton set with the same invalid root (in the case of $(\mathbf{APT}_7)$ an equivalent root).
- If it is $(\mathbf{APT}^o_8)$, $(\mathbf{APT}^o_9)$, or $(\mathbf{APT}_{11})$ by Proposition 8 N has some invalid child, which corresponds to the root of some premise T_i. By the induction hypothesis, there is some $T' \in APT'(T_i)$ with invalid root. And by observing the rules of Figure 13 it can be checked that every subtree T_i of the root of T verifies $APT'(T_i) \subseteq APT'(T)$. Then $T' \in APT'(T)$.

□

Theorem 3. *Let T be a finite proof tree representing an inference in the calculus of Figures 1, 4, 5, 6, 7, and 12 w.r.t. some rewrite theory $\mathcal{R}$. Let $\mathcal{I}$ be an intended interpretation of $\mathcal{R}$ such that the root of T is invalid in $\mathcal{I}$. Then:*

- *$APT(T)$ contains at least one buggy node (completeness).*
- *Any buggy node in $APT(T)$ has an associated wrong statement, missing statement, or wrong condition in $\mathcal{R}$ according to Table 3 (correctness).*

Proof. We prove each item separately:

- $APT(T)$ contains at least one invalid node, since its root is the root of T, and any debugging tree containing an invalid node contains a buggy node by Proposition 7.
- First we observe that the root of $APT(T)$ cannot be buggy, because if it is invalid then it has an invalid child (Lemma 1(b)). Therefore any buggy node must be part of $APT'(T)$ (the premise in $(\mathbf{APT}_1)$).

  Let N be a buggy node occurring in $APT'(T)$. Then N is the root of some tree T_N, subtree of some $T' \in APT'(T)$. By the structure of the APT' rules this means that there is a subtree T' of T such that $T_N \in APT'(T')$. We prove that N has an associated wrong statement in S by induction on the number of nodes of T', $n(T')$.

  If $n(T') = 1$ then T' contains only one node and $APT'(T') = \{T'\}$ by Lemma 1(a). Then the only possible buggy node is N, which means that N is also buggy in T and that the associated fragment of code is wrong by Proposition 8.

  If $n(T') > 1$ we examine the APT rule applied at the root of T':

  $(\mathbf{APT}_2)$ Then T' is of the form

$$\dfrac{\dfrac{T_1 \ldots T_n}{e_1 \to e'}\,\mathsf{Rep}_{\to} \qquad T''}{e_1 \to e_2}\,\mathsf{Tr}_{\to}$$

  Hence $N \equiv (e_1 \to e_2)$ and T_N is

$$\dfrac{APT'(T_1)\ldots APT'(T_n)\ APT'(T'')}{e_1 \to e_2}\,\mathsf{Rep}_{\to}$$

  Since N is buggy in T_N it is invalid w.r.t. $\mathcal{I}$. By Proposition 8, $e_1 \to e_2$ cannot be buggy in T', i.e., either T'' has an invalid root or $e_1 \to e'$ is invalid. But T'' cannot be invalid because $APT'(T'')$ is a child subtree of N and by Lemma 1(b) it would contain a tree T''' with invalid root, which is not possible because T''' is a child of the buggy node N in T_N. Therefore $e_1 \to e'$ is invalid. Moreover, the roots of $T_1, \ldots, T_n$ are also valid by the same reason: $APT'(T_1), \ldots, APT'(T_n)$ are child subtrees of N in T_N and cannot have an invalid root. Therefore $e_1 \to e'$ is buggy in T', i.e., is buggy in T and by Proposition 8 the equation associated to label $\mathsf{Rep}_{\to}$ is wrong. And this label is the same that can be found associated to N in the APT' T_N. Therefore the buggy node N of the APT' has an associated wrong equation.

  $(\mathbf{APT}_3)$ In this case T' has the form

$$\dfrac{\dfrac{T_1 \quad \ldots \quad T_n}{t \to_{red} t''}\,\mathsf{Rdc}_1 \qquad T}{t \to_{norm} t'}\,\mathsf{NTr}$$

  Thus $t \to_{norm} t'$ and T_N is

$$\dfrac{APT'(T_1) \quad \ldots \quad APT'(T_n) \quad APT'(T)}{t \to_{norm} t'}\,\mathsf{Rdc}_1$$

  By Proposition 8 we know that N cannot be buggy in T', thus either $t \to_{red} t''$ or the root of T is invalid. However, if the root of T were invalid we know by Lemma 1 that the set obtained with APT' would contain a tree with an invalid root and then N cannot be buggy. Therefore, $t \to_{red} t''$ is invalid but, for the same reason as before, $T_1 \ldots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 8 the rule label Rdc_1 has associated a wrong equation. Since this same label has been now assigned to N, the buggy node in the abbreviated proof tree has an associated wrong equation.

($\mathbf{APT}_4$) In this case T' has the form

$$\cfrac{\cfrac{T_1 \quad \dots \quad T_n}{t \Rightarrow^{top} S'}\ \mathsf{Top} \qquad T'_1 \quad \dots \quad T'_n}{t \Rightarrow_1 S}\ \mathsf{Stp}$$

Thus $N \equiv t \Rightarrow_1 S$ and T_N is

$$\frac{APT'(T_1) \quad \dots \quad APT'(T_n) \quad APT'(T'_1) \quad \dots \quad APT'(T'_n)}{t \Rightarrow_1 S}\ \mathsf{Top}$$

By Proposition 5 we know that N cannot be buggy in T', thus either of $t \Rightarrow^{top} S'$ or the root of one of $T'_1 \dots T'_n$ is invalid. However, if the root of one of the trees $T'_1 \dots T'_n$ were invalid we know by Lemma 1 that the set obtained with APT' would contain a tree with an invalid root and then N cannot be buggy. Therefore, $t \Rightarrow^{top} S'$ is invalid but, for the same reason as before, $T_1 \dots T_n$ cannot be invalid, so it is also buggy in T' and by Proposition 8 the rule label Top has associated a missing rule. Since this same label has been now assigned to N, the buggy node in the abbreviated proof tree has an associated missing rule.

($\mathbf{APT}_5$) and ($\mathbf{APT}_6$) Analogous to the previous cases.

($\mathbf{APT}_7$) T' has the form

$$\frac{T_{t \to_{norm} t'} \ T_1 \ \dots \ T_n}{t :_{ls} s}\mathsf{Ls}$$

Then $N \equiv t :_{ls} s$ and T_N is

$$\frac{APT'(T_{t \to_{norm} t'})\ APT'(T_1)\ \dots\ APT'(T_n)}{t' :_{ls} s}\mathsf{Ls}$$

Since N is buggy in T_N all the trees in $APT'(T_{t \to_{norm} t'})$ $APT'(T_1)$ $\dots$ $APT'(T_n)$ are valid and by Lemma 1 the roots of $T_{t \to_{norm} t'}$ T_1 $\dots$ T_n are also valid and N is buggy in T'. By Proposition 8 it is associated with a missing membership in T' and, since we have the same label in T_N, the result holds.

($\mathbf{APT}^o_8$), ($\mathbf{APT}^o_9$), ($\mathbf{APT}_{11}$) Then $T_N \in APT'(T_i)$ for some child subtree T_i of the root of T' and the result holds by the induction hypothesis.

($\mathbf{APT}^m_8$) We check that actually this rule cannot be applied to produce a buggy node and therefore must not be considered here. If ($\mathbf{APT}^m_8$) is applied then T' must be of the form

$$\frac{T_1 \quad T_2}{e_1 \Rightarrow e_2}\mathsf{Tr}_{\Rightarrow}$$

N is $e_1 \Rightarrow e_2$ and T_N is

$$\frac{APT'(T_1)\ APT'(T_2)}{e_1 \Rightarrow e_2}\mathsf{Tr}_{\Rightarrow}$$

And N can be invalid but not buggy in T' (and hence in T) by Proposition 8, because it is the conclusion of a transitivity inference, and thus either T_1 or T_2 has an invalid root. Then by Lemma 1(b), either $APT'(T_1)$ or $APT'(T_2)$ have an invalid root and N is not buggy in T_N.

($\mathbf{APT}^m_9$) Analogous to the previous case.

($\mathbf{APT}_{10}$) We present the proof for the inference rule Fulfill, with the other cases being analogous. T' has the form

$$\frac{T_1 \ \dots \ T_n}{\mathit{fulfilled}(\mathcal{C}, t)}\mathsf{Fulfill}$$

Then $N \equiv \mathit{fulfilled}(\mathcal{C}, t)$ and T_N is

$$\frac{APT'(T_1)\ \ldots\ APT'(T_n)}{\mathit{fulfilled}(\mathcal{C}, t)}\,\mathsf{Fulfill}$$

Since N is buggy in T_N all the trees in $APT'(T_1)\ \ldots\ APT'(T_n)$ are valid and by Lemma 1 the roots of $T_1 \ldots T_n$ are also valid and N is buggy in T'. By Proposition 8 it is associated with a wrong statement in T' and, since we have the same label in T_N, the result holds.

$\square$

Integrating Maude into Hets

Mihai Codescu[1], Till Mossakowski[1], Adrián Riesco[2], and Christian Maeder[1]

[1] DFKI GmbH Bremen and University of Bremen, Germany
[2] Facultad de Informática, Universidad Complutense de Madrid, Spain

Abstract. Maude modules can be understood as models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. However, Maude lacks the formal tools to perform some of these analyses and thus they can only be done by hand. The Heterogeneous Tool Set HETS is an institution-based combination of different logics and corresponding rewriting, model checking and proof tools. We present in this paper an integration of Maude into HETS that allows to use the logics and tools already integrated in HETS with Maude specifications. To achieve such integration we have defined an institution for Maude based on preordered algebras and a comorphism between Maude and CASL, the central logic in HETS.

Keywords: Heterogeneous specifications, rewriting logic, institution, Maude, CASL.

1 Introduction

Maude [3] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in rewriting logic, a simple and expressive logic which allows the representation of many models of concurrent and distributed systems.

The key point is that there are three different uses of Maude modules:

1. As programs, to implement some application. We may have chosen Maude because its features make the programming task easier and simpler than other languages.
2. As formal executable specifications, that provide a rigorous mathematical model of an algorithm, a system, a language, or a formalism. Because of the agreement between operational and mathematical semantics, this mathematical model is at the same time executable.
3. As models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. For example, we may want to prove that our Maude module terminates; or that a given function, equationally defined in the module, satisfies some properties expressed as first-order formulas.

However, when we follow this last approach we find that, although Maude can automatically perform analyses like model checking of temporal formulas or verification of invariants, other formal analyses have to be done "by hand," thus disconnecting the real Maude code from its logical meaning. Although some efforts, like the Inductive Theorem Prover [4], have been dedicated to palliate this problem, they are restricted to inductive proofs in Church-Rosser equational theories, and they lack the generality to deal with all the features of Maude. With our approach, we cover arbitrary first-order properties (also written in logics different from Maude), and open the door to automated induction strategies such as those of ISAplanner [7].

The Heterogeneous Tool Set, HETS [16] is an institution-based combination of different logics and corresponding rewriting, model checking and proof tools. Tools that have been integrated into HETS include the SAT solvers zChaff and MiniSat, the automated provers SPASS, Vampire and Darwin, and the interactive provers Isabelle and VSE.

In this paper, we describe an integration of Maude into HETS from which we expect several benefits: On the one hand, Maude will be the first dedicated rewriting engine that is integrated into HETS (so far, only the rewriting engine of Isabelle is integrated, which however is quite specialized towards higher-order proofs). On the other hand, certain features of the Maude module system like views lead to proof obligations that cannot be checked with Maude—HETS will be the suitable framework to prove them, using the above mentioned proof tools.

The rest of the paper is organized as follows: after briefly introducing HETS in Section 2 and Maude in Section 3, Section 4 describes the institution we have defined for Maude and the comorphism from this institution to CASL. Section 5 shows how development graphs for Maude specifications are built, and then how they are normalized to deal with freeness constraints. Section 6 illustrates the integration of Maude into Hets with the help of an example, while Section 7 concludes and outlines the future work.

2 Hets

The central idea of HETS is to provide a general logic integration and proof management framework. One can think of HETS acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations.

The benefit of plugging in a new logic and tool such as Maude into the HETS motherboard is the gained interoperability with the other logics and tools available in HETS.

The work that needs to be done for such an integration is to prepare both the Maude logic and tool so that it can act as an expansion card for HETS. On the side of the semantics, this means that the logic needs to be organized as an *institution* [12]. Institutions capture in a very abstract and flexible way the notion of a logical system, by leaving open the details of signatures, models, sentences (axioms) and satisfaction (of sentences in models). The only condition governing the behavior of institutions is the *satisfaction condition*, stating that *truth is invariant under change of notation* (or enlargement of context), which is captured by the notion of signature morphism (which leads to translations of sentences and reductions of models), see [12] for formal details.

Indeed, HETS has interfaces for plugging in the different components of an institution: signatures, signature morphisms, sentences, and their translation along signature morphisms. Recently, even (some) models and model reducts have been covered, although this is not needed here. Note, however, that the model theory of an institution (including model reducts and the satisfaction condition) is essential when relating different logics via institution comorphisms. The logical correctness of their use in multi-logic proofs is ensured by model-theoretic means.

For proof management, HETS uses *development graphs* [15]. They can be defined over an arbitrary institution, and they are used to encode structured specifications in various phases of the development. Roughly speaking, each node of the graph represents a theory. The links of the graph define how theories can make use of other theories.

Definition 1. *A* development graph *is an acyclic, directed graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$.*

$\mathcal{N}$ is a set of nodes. Each node $N \in \mathcal{N}$ is a tuple (Σ^N, Φ^N) such that Σ^N is a signature and $\Phi^N \subseteq Sen(\Sigma^N)$ is the set of **local axioms** *of N.*

$\mathcal{L}$ is a set of directed links, so-called **definition links**, *between elements of $\mathcal{N}$. Each definition link from a node M to a node N is either*

– **global** *(denoted $M \overset{\sigma}{\Longrightarrow} N$), annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or*
– **local** *(denoted $M \xrightarrow{\sigma} N$), again annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or*
– **hiding** *(denoted $M \underset{hide}{\overset{\sigma}{\Longrightarrow}} N$), annotated with a signature morphism $\sigma : \Sigma^N \to \Sigma^M$* going against the direction of the link, *or*
– **free** *(denoted $M \underset{free}{\overset{\sigma}{\Longrightarrow}} N$), annotated with a signature morphism $\sigma : \Sigma \to \Sigma^M$* where Σ is a subsignature of Σ^M.

Definition 2. *Given a node M in a development graph $\mathcal{DG}$, its associated class* $\mathbf{Mod}_{\mathcal{DG}}(M)$ *of models (or M-models for short) is inductively defined to consist of those Σ^M-models m for which*

1. *m satisfies the local axioms Φ^M,*
2. *for each $N \overset{\sigma}{\Longrightarrow} M \in \mathcal{DG}$, $m|_\sigma$ is an N-model,*
3. *for each $N \xrightarrow{\sigma} M \in \mathcal{DG}$, $m|_\sigma$ satisfies the local axioms Φ^N,*
4. *for each $N \underset{hide}{\overset{\sigma}{\Longrightarrow}} M \in \mathcal{DG}$, m has a σ-expansion m' (i.e. $m'|_\sigma = m$) that is an N-model, and*
5. *for each $N \underset{free}{\overset{\sigma}{\Longrightarrow}} M \in \mathcal{DG}$, m is an N-model that is persistently σ-free in* $\mathbf{Mod}(N)$. *The latter means that for each N-model m' and each model morphism $h : m|_\sigma \to m'|_\sigma$, there exists a unique model morphism $h^{\#} : m \to m'$ with $h^{\#}|_\sigma = h$.*

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Again, we distinguish between local and global theorem links (denoted by $N \overset{\sigma}{=\!=\!\Rightarrow} M$ and $N \overset{\sigma}{-\!-\!\succ} M$ respectively). We also need theorem links $N \underset{hide\ \theta}{\overset{\sigma}{=\!=\!\Rightarrow}} M$ (where for some Σ, $\theta : \Sigma \rightarrow \Sigma^N$ and $\sigma : \Sigma \rightarrow \Sigma^M$) involving hiding. The semantics of global theorem links is given by the next definition; the others do not occur in our examples and we omit them.

Definition 3. *Let $\mathcal{DG}$ be a development graph and N, M nodes in $\mathcal{DG}$.*
$\mathcal{DG}$ **implies** *a global theorem link $N \overset{\sigma}{=\!=\!\Rightarrow} M$ (denoted $\mathcal{DG} \models N \overset{\sigma}{=\!=\!\Rightarrow} M$) iff for all $m \in Mod(M)$, $m|_\sigma \in Mod(N)$.*

3 Rewriting Logic and Maude

Maude is an efficient tool for equational reasoning and rewriting. Methodologically, Maude specifications are divided into a specification of the data objects and a specification of some concurrent transition system, the states of which are given by the data part. Indeed, at least in specifications with initial semantics, the states can be thought of as equivalence classes of terms. The data part is written in a variant of subsorted conditional equational logic. The transition system is expressed in terms of a binary rewriting relation, and also may be specified using conditional Horn axioms.

Two corresponding logics have been introduced and studied in the literature: rewriting logic and preordered algebra [13]. They essentially differ in the treatment of rewrites: whereas in rewriting logic, rewrites are named, and different rewrites between two given states (terms) can be distinguished (which corresponds to equipping each carrier set with a category of rewrites), in preordered algebra, only the existence of a rewrite does matter (which corresponds to equipping each carrier set with a preorder of rewritability).

Rewriting logic has been announced as the logic underlying Maude [3]. Maude modules lead to rewriting logic theories, which can be equipped with loose semantics (`fth`/`th` modules) or initial/free semantics (`fmod`/`mod` modules). Although rewriting logic is not given as an institution [6], a so-called specification frame (collapsing signatures and sentences into theories) would be sufficient for our purposes.

However, after a closer look at Maude and rewriting logic, we found out that de facto, the logic underlying Maude differs from the rewriting logic as defined in [13]. The reasons are:

1. In Maude, labels of rewrites cannot (and need not) be translated along signature morphisms. This means that *e.g. Maude views do not lead to theory morphisms in rewriting logic!*

2. Although labels of rewrites are used in traces of counterexamples, they play a subsidiary role, because they cannot be used in the linear temporal logic of the Maude model checker.

Specially the first reason completely rules out a rewriting logic-based integration of Maude into Hets: if a view between two modules is specified, Hets definitely needs a theory morphism underlying the view.[1] However, the Maude user does not need to provide the action of the signature morphism on labeled rewrites, and generally, there is more than one possibility to specify this action.

The conclusion is that the most appropriate logic to use for Maude is preordered algebra [10]. In this logic, rewrites are neither labeled nor distinguished, only their existence is important. This implies that Maude views lead to theory morphisms in the institution of preordered algebras. Moreover, this setting also is in accordance with the above observation that in Maude, rewrite labels are not first-class citizens, but are mere names of sentences that are convenient for decorating tool output (e.g. traces of the model checker). Labels of sentences play a similar role in Hets, which perfectly fits here.

Actually, the switch from rewriting logic to preordered algebras has effects on the consequence relation, contrary to what is said in [13]. Consider the following Maude theory:

```
th A is
 sorts S T .
 op a : -> S .
 eq X:S = a .
 ops h k : S -> T .
 rl [r] : a => a .
 rl [s] : h(a) => k(a) .
endfth
```

This logically implies $h(x) \Rightarrow k(x)$ in preordered algebra, but not in rewriting logic, since in the latter logic it is easy to construct models in which the naturality condition $r; k(r) = h(r); s$ fails to hold.

Before describing how to encode Maude into Hets we briefly outline the structuring mechanisms used in Maude specifications:

Module importation. In Maude, a module can be imported in three different modes, each of them stating different semantic constraints: Importing a module in `protecting` mode intuitively means that *no junk and no confusion* are added; importing a module in `extending` mode indicates that junk is allowed, but *confusion is forbidden*; finally, importing a module in `including` mode indicates that *no requirements* are assumed.

Module summation. The summation module operation creates a new module that includes all the information in its summands.

[1] If the Maude designers would let (and force) users to specify the action of signature morphisms on rewrite labels, it would not be difficult to switch the Hets integration of Maude to being based on rewriting logic.

Renaming. The renaming expression allows to rename sorts, operators (that can be distinguished by their profiles), and labels.

Theories. Theories are used to specify the requirements that the parameters used in parameterized modules must fulfill. Functional theories are membership equational specifications with *loose* semantics. Since the statements specified in theories are not expected to be executed in general, they do not need to satisfy the executability requirements.

Views. A view indicates how a particular module satisfies a theory, by mapping sorts and operations in the theory to those in the target module, in such a way that the induced translations on equations and membership axioms are provable in the module. Note that Maude does not provide a syntax for mapping rewrite rules; however, the existence of rewrites between terms must be preserved by views.

4 Relating the Maude and CASL Logics

In this section, we will relate Maude and CASL at the level of logical systems. The structuring level will be considered in the next section.

4.1 Maude

As already motivated in Section 3, we will work with preordered algebra semantics for Maude. We will define an institution, that we will denote $Maude^{pre}$, which can be, like in the case of Maude's logic, parametric over the underlying equational logic. Following the Maude implementation, we have used membership equational logic [14]. Notice that the resulting institution $Maude^{pre}$ is very similar to the one defined in the context of CafeOBJ [10,6] for preordered algebra (the differences are mainly given by the discussion about operation profiles below, but this is only a matter of representation). This allows us to make use of some results without giving detailed proofs.

Signatures of $Maude^{pre}$ are tuples $(K, F, kind : (S, \leq) \to K)$, where K is a set (of *kinds*), *kind* is a function assigning a kind to each *sort* in the poset $(S, \leq)$, and F is a set of function symbols of the form $F = \{F_{k_1 \ldots k_n \to x} \mid k_i, k \in K\} \cup \{F_{s_1 \ldots s_n \to s} \mid s_i, s \in S\}$ such that if $f \in F_{s_1 \ldots s_n \to s}$, there is a symbol $f \in F_{kind(s_1) \ldots kind(s_n) \to kind(s)}$. Notice that there is actually no essential difference between our putting operation profiles on sorts into the signatures and Meseguer's original formulation putting them into the sentences.

Given two signatures $\Sigma_i = (K_i, F_i, kind_i)$, $i \in \{1, 2\}$, a signature morphism $\phi : \Sigma_1 \to \Sigma_2$ consists of a function $\phi^{kind} : K_1 \to K_2$ which preserves $\leq_1$, a function between the sorts $\phi^{sort} : S_1 \to S_2$ such that $\phi^{sort}; kind_2 = kind_1; \phi^{kind}$ and the subsorts are preserved, and a function $\phi^{op} : F_1 \to F_2$ which maps operation symbols compatibly with the types. Moreover, the overloading of symbol names must be preserved, i.e. the name of $\phi^{op}(\sigma)$ must be the same both when mapping the operation symbol σ on sorts and on kinds. With composition defined component-wise, we get the category of signatures.

For a signature Σ, a model M interprets each kind k as a preorder $(M_k, \leq)$, each sort s as a subset M_s of $M_{kind(s)}$ that is equipped with the induced preorder, with M_s a subset of $M_{s'}$ if $s < s'$, and each operation symbol $f \in F_{k_1 \ldots k_n, k}$ as a function $M_f : M_{k_1} \times \ldots \times M_{k_n} \rightarrow M_k$ which has to be monotonic and such that for each function symbol f on sorts, its interpretation must be a restriction of the interpretation of the corresponding function on kinds. For two Σ-models A and B, a homomorphism of models is a family $\{h_k : A_k \rightarrow B_k\}_{k \in K}$ of preorder-preserving functions which is also an algebra homomorphism and such that $h_{kind(s)}(A_s) \subseteq B_s$ for each sort s.

The sentences of a signature Σ are Horn clauses built with three types of atoms: equational atoms $t = t'$, membership atoms $t : s$, and rewrite atoms $t \Rightarrow t'$, where t, t' are F-terms and s is a sort in S. Given a Σ-model M, an equational atom $t = t'$ holds in M if $M_t = M_{t'}$, a membership atom $t : s$ holds when M_t is an element of M_s, and a rewrite atom $t \Rightarrow t'$ holds when $M_t \leq M_{t'}$. Notice that the set of variables X used for quantification is K-sorted. The satisfaction of sentences extends the satisfaction of atoms in the obvious way.

4.2 CASL

Casl, the Common Algebraic Specification Language [1,5], has been designed by CoFI, the international *Common Framework Initiative for algebraic specification and development*. Its underlying logic combines first-order logic and induction (the latter is expressed using so-called sort generation constraints, which express term-generatedness of a part of a model; this is needed for the specification of the usual inductive datatypes) with subsorts and partial functions. The institution underlying Casl is introduced in two steps: first, many-sorted partial first-order logic with sort generation constraints and equality ($PCFOL^=$) is introduced, and then, subsorted partial first-order logic with sort generation constraints and equality ($SubPCFOL^=$) is described in terms of $PCFOL^=$. In contrast to Maude, Casl's subsort relations may be interpreted by arbitrary injections $inj_{s,t}$, not only by subsets. We refer to [5] for details. We will only need the Horn Clause fragment of first-order logic. For freeness (see Sect. 5.1), we will also need sort generation constraints, as well as the *second-order* extension of Casl with quantification over predicates.

4.3 Encoding Maude in CASL

We now present an encoding of Maude into Casl. It can be formalized as a so-called institution comorphism [11]. The idea of the encoding of $Maude^{pre}$ in Casl is that we represent rewriting as a binary predicate and we axiomatize it as a preorder compatible with operations.

Every Maude signature $(K, F, kind : (S, \leq) \rightarrow K)$ is translated to the Casl theory $((S', \leq', F, P), E)$, where S' is the disjoint union of K and S, $\leq'$ extends the relation $\leq$ on sorts with pairs $(s, kind(s))$, for each $s \in S$, $rew \in P_{s,s}$ for any $s \in S'$ is a binary predicate and E contains axioms stating that for any kind

k, $rew \in P_{k,k}$ is a preorder compatible with the operations. The latter means that for any $f \in F_{s_1..s_n,s}$ and any x_i, y_i of sort $s_i \in S'$, $i = 1,..,n$, if $rew(x_i, y_i)$ holds, then $rew(f(x_1, \dots, x_n), f(y_1, \dots, y_n))$ also holds.

Let $\Sigma_i,\ i = 1, 2$ be two Maude signatures and let $\varphi : \Sigma_1 \to \Sigma_2$ be a Maude signature morphism. Then its translation $\Phi(\varphi) : \Phi(\Sigma_1) \to \Phi(\Sigma_2)$ denoted ϕ, is defined as follows:

– for each $s \in S$, $\phi(s) := \varphi^{sort}(s)$ and for each $k \in K$, $\phi(k) := \varphi^{kind}(k)$.
– the subsort preservation condition of ϕ follows from the similar condition for φ.
– for each operation symbol σ, $\phi(\sigma) := \varphi^{op}(\sigma)$.
– *rew* is mapped identically.

The sentence translation map for each signature is obtained in two steps. While the equational atoms are translated as themselves, membership atoms $t : s$ are translated to CASL memberships $t\ in\ s$ and rewrite atoms of form $t \Rightarrow t'$ are translated as $rew(t, t')$. Then, any sentence of Maude of the form $(\forall x_i : k_i)H \implies C$, where H is a conjunction of Maude atoms and C is an atom is translated as $(\forall x_i : k_i)H' \implies C'$, where H' and C' are obtained by mapping all the Maude atoms as described before.

Given a Maude signature Σ, a model M' of its translated theory (Σ', E) is reduced to a Σ-model denoted M where:

– for each kind k, define $M_k = M'_k$ and the preorder relation on M_k is *rew*;
– for each sort s, define M_s to be the image of M'_s under the injection $inj_{s,kind(s)}$ generated by the subsort relation;
– for each f on kinds, let $M_f(x_1, .., x_n) = M'_f(x_1, .., x_n)$ and for each f on sorts of result sort s, let $M_f(x_1, .., x_n) = inj_{s,kind(s)}(M'_f(x_1, .., x_n))$. M_f is monotone because axioms ensure that M'_f is compatible with *rew*.

The reduct of model homomorphisms is the expected one; the only thing worth noticing is that $h_{kind(s)}(M_s) \subseteq N_s$ for each sort s follows from the CASL model homomorphism condition of h.

Notice that the model reduct is an isomorphism of categories.

5 From Maude Modules to Development Graphs

We describe in this section how Maude structuring mechanisms described in Section 3 are translated into development graphs. Then, we explain how these development graphs are normalized to deal with freeness constraints.

Signature morphisms are produced in different ways; explicitly, renaming of module expressions and views lead to signature morphisms; however, implicitly we also find other morphisms: the sorts defined in the theories are qualified with the parameter in order to distinguish sorts with the same name that will be instantiated later by different ones; moreover, sorts defined (not imported) in parameterized modules can be parameterized as well, so when the theory is instantiated with a view these sorts are also renamed (e.g. the sort `List{X}` for generic lists can become `List{Nat}`).

Each Maude module generates two nodes in the development graph. The first one contains the theory equipped with the usual loose semantics. The second one, linked to the first one with a free definition link (whose signature morphism is detailed below), contains the same signature but no local axioms and stands for the free models of the theory. Note that Maude theories only generate one node, since their initial semantics is not used by Maude specifications. When importing a module, we will select the node used depending on the chosen importation mode:

– The `protecting` mode generates a non-persistent free link between the current node and the node standing for the free semantics of the included one.
– The `extending` mode generates a global link with the annotation PCons?, that stands for proof-theoretic conservativity and that can be checked with a special conservativity checker that is integrated into Hets.
– The `including` mode generates a global definition link between the current node and the node standing for the loose semantics of the included one.

The summation module expression generates a new node that includes all the information in its summands. Note that this new node can also need a node with its free model if it is imported in protecting mode.

The model class of parameterized modules consists of free extensions of the models of their parameters, that are persistent on sorts, but not on kinds. This notion of freeness has been studied in [2] under assumptions like existence of top sorts for kinds and sorted variables in formulas; our results hold under similar hypotheses. Thus, we use the same non-persistent free links described for protecting importation to link these modules with their corresponding theories. Views do not generate nodes in the development graph but theorem links between the node corresponding to the source theory and the node with the free model of the target. However, Maude views provide a special kind of mapping between terms, that can in general map functions of different arity. When this mapping is used we generate a new inner node extending the signature of the target to include functions of the adequate arity.

We illustrate how to build the development graph with an example. Consider the following Maude specifications:

```
fmod M1 is
 sort S1 .
 op _+_ : S1 S1 -> S1 [comm] .
endfm
```

```
fmod M2 is
  sort S2 .
endfm
```

```
th T is
 sort S1 .
 op _._ : S1 S1 -> S1 .
 eq V1:S1 . V2:S1 = V2:S1 . V1:S1 [nonexec] .
endth
```

```
mod M3{X :: T} is
 sort S4 .
endm
```

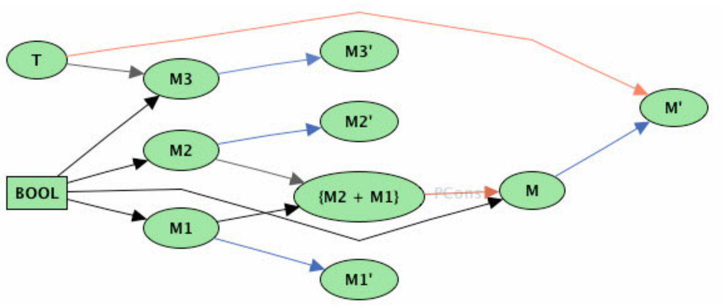


Fig. 1. Development Graph for Maude Specifications

```
mod M is
 ex M1 + M2 * (sort S2 to S) .
endm
```

```
view V from T to M is
 op _._ to _+_ .
endv
```

HETS builds the graph shown in Fig. 1, where the following steps take place:

– Each module has generated a node with its name and another primed one that contains the initial model, while both of them are linked with a non-persistent free link. Note that theory T did not generate this primed node.
– The summation expression has created a new node that includes the theories of M1 and M2, importing the latter with a renaming; this new node, since it is imported in `extending` mode, uses a link with the PCons? annotation.
– There is a theorem link between T and the free (here: initial) model of M. This link is labeled with the mapping defined in the view V.
– The parameterized module M3 includes the theory of its parameter with a renaming, that qualifies the sort. Note that these nodes are connected by means of a non-persistent freeness link.

It is straightforward to show:

Theorem 1. *The translation of Maude modules into development graphs is semantics-preserving.*

Once the development graph is built, we can apply the (logic independent) calculus rules that reduce global theorem links to local theorem links, which are in turn discharged by local theorem proving [15]. This can be used to prove Maude views, like e.g. “natural numbers are a total order.” We show in the next section how we deal with the freeness constraints imposed by free definition links.

5.1 Normalization of Free Definition Links

Maude uses initial and free semantics intensively. The semantics of freeness is, as mentioned, different from the one used in CASL in that the free extensions of models are required to be persistent only on sorts and new error elements can be added on the interpretation of kinds. Attempts to design the translation to CASL in such a way that Maude free links would be translated to usual free definition links in CASL have been unsuccessful. We decided thus to introduce a special type of links to represent Maude's freeness in CASL. In order not to break the development graph calculus, we need a way to normalize them. The idea is to replace them with a semantically equivalent development graph in CASL. The main idea is to make a free extension persistent by duplicating parameter sorts appropriately, such that the parameter is always explicitly included in the free extension.

For any Maude signature Σ, let us define an extension $\Sigma^{\#} = (S^{\#}, \leq^{\#}, F^{\#}, P^{\#})$ of the translation $\Phi(\Sigma)$ of Σ to CASL as follows:

– $S^{\#}$ unites with the sorts of $\Phi(\Sigma)$ the set $\{[s] \mid s \in Sorts(\Sigma)\}$;
– $\leq^{\#}$ extends the subsort relation $\leq$ with pairs $(s, [s])$ for each sort s and $([s], [s'])$ for any sorts $s \leq s'$;
– $F^{\#}$ adds the function symbols $\{f : [w] \to [s]\}$ for all function symbols on sorts $f : w \to s$;[2]
– $P^{\#}$ adds the predicate symbol *rew* on all new sorts.

Now, we consider a Maude non-persistent free definition link and let $\sigma : \Sigma \to \Sigma'$ be the morphism labeling it.[3] We define a CASL signature morphism $\sigma^{\#} : \Phi(\Sigma) \to \Sigma'^{\#}$: on sorts, $\sigma^{\#}(s) := \sigma^{sort}(s)$ and $\sigma^{\#}([s]) := [\sigma^{sort}(s)]$; on operation symbols, we can define $\sigma^{\#}(f) := \sigma^{op}(f)$ and this is correct because the operation symbols were introduced in $\Sigma'^{\#}$; *rew* is mapped identically.

The normalization of Maude freeness is then illustrated in Fig.2. Given a free non-persistent definition link $M \overset{\sigma}{\underset{free}{\Longrightarrow}} N$, with $\sigma : \Sigma \to \Sigma_N$, we first take the translation of the nodes to CASL (nodes M' and N') and then introduce a new node, K, labeled with $\Sigma_N^{\#}$, a global definition link from M' to M'' labeled with the inclusion ι_N of Σ_N in $\Sigma_N^{\#}$, a free definition link from M'' to K labeled with $\sigma^{\#}$ and a hiding definition link from K to N' labeled with the inclusion ι_N.[4]

Notice that the models of N are Maude reducts of CASL models of K, reduced along the inclusion ι_N.

The next step is to eliminate CASL free definition links. The idea is to use then a transformation specific to the second-order extension of CASL to normalize freeness. The intuition behind this construction is that it mimics the quotient term algebra construction, that is, the free model is specified as the homomorphic image of an absolutely free model (i.e. term model).

We are going to make use of the following known facts [18]:

$$\begin{tikzcd}
M \arrow[rr, Rightarrow, "\sigma", "n.p.free"'] \arrow[d, Rightarrow] & & N \arrow[dd, Rightarrow] \\
M' \arrow[d, Rightarrow, "\iota_N"] & & \\
M'' \arrow[r, Rightarrow, "\sigma^{\#}", "free"'] & K \arrow[r, Rightarrow, "\iota_n", "hide"'] & N'
\end{tikzcd}$$

Fig. 2. Normalization of Maude free links

[2] $[x_1 \ldots x_n]$ is defined to be $[x_1] \ldots [x_n]$.
[3] In Maude, this would usually be an injective renaming.
[4] The arrows without labels in Fig.2 correspond to heterogeneous links from Maude to CASL.

Fact 1. *Extensions of theories in Horn form admit free extensions of models.*

Fact 2. *Extensions of theories in Horn form are monomorphic.*

Given a free definition link $M \overset{\sigma}{\underset{free}{\Longrightarrow}} N$, with $\sigma : \Sigma \to \Sigma^N$ such that $Th(M)$ is in Horn form, replace it with $M \xRightarrow{incl} K \overset{incl}{\underset{hide}{\Longrightarrow}} N'$, where N' has the same signature as N, *incl* denotes inclusions and the node K is constructed as follows.

The signature Σ^K consists of the signature Σ^M disjointly united with a copy of Σ^M, denoted $\iota(\Sigma_M)$ which makes all function symbols total (let us denote $\iota(x)$ the corresponding symbol in this copy for each symbol x from the signature Σ^M) and augmented with new operations $h : \iota(s) \to ?s$, for any sort s of Σ^M and $make_s : s \to \iota(s)$, for any sort s of the source signature Σ of the morphism σ labelling the free definition link.

The axioms ψ^K of the node K consist of:

- sentences imposing the bijectivity of *make*;
- axiomatization of the sorts in $\iota(\Sigma_M)$ as free types with all operations as constructors, including *make* for the sorts in $\iota(\Sigma)$;
- homomorphism conditions for h:

$$h(\iota(f)(x_1, \ldots, x_n)) = f(h(x_1), \ldots, h(x_n))$$

  and

$$\iota(p)(t_1, \ldots, t_n) \Rightarrow p(h(t_1), \ldots, h(t_n))$$

- surjectivity of homomorphisms:

$$\forall y : s.\exists x : \iota(s).h(x) \overset{e}{=} y$$

- a second-order formula saying that the kernel of h is the least partial predicative congruence[5] satisfying $Th(M)$. This is done by quantifying over a predicate symbol for each sort for the binary relation and one predicate symbol for each relation symbol as follows:

$$\begin{array}{l}\forall \{P_s : \iota(s), \iota(s)\}_{s \in Sorts(\Sigma_M)}, \{P_{p:w} : \iota(w)\}_{p:w \in \Sigma_M} \\ .\ \mathit{symmetry} \wedge \mathit{transitivity} \wedge \mathit{congruence} \wedge \mathit{satThM} \implies \mathit{largerThenKerH}\end{array}$$

  where *symmetry* stands for

$$\bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s).P_s(x, y) \implies P_s(y, x),$$

[5] A partial predicative congruence consists of a symmetric and transitive binary relation for each sort and a relation of appropriate type for each predicate symbol.

transitivity stands for

$$\bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s), z : \iota(s). P_s(x,y) \wedge P_s(y,z) \implies P_s(x,z),$$

congruence stands for

$$\begin{array}{l} \bigwedge_{f_{w \to s} \in \Sigma^M} \forall x_1 \ldots x_n : \iota(w), y_1 \ldots y_n : \iota(w) \,. \\ D(\iota(f_{w,s})(\bar{x})) \wedge D(\iota(f_{w,s})(\bar{y})) \wedge P_w(\bar{x},\bar{y}) \implies P_s(\iota(f_{w,s})(\bar{x}), \iota(f_{w,s})(\bar{y})) \end{array}$$

and

$$\begin{array}{l} \bigwedge_{p_w \in \Sigma^M} \forall x_1 \ldots x_n : \iota(w), y_1 \ldots y_n : \iota(w) \,. \\ D(\iota(f_{w,s})(\bar{x})) \wedge D(\iota(f_{w,s})(\bar{y})) \wedge P_w(\bar{x},\bar{y}) \implies P_{p:w}(\bar{x}) \Leftrightarrow P_{p:w}(\bar{y}) \end{array}$$

where D indicates definedness. *satThM* stands for

$$Th(M)[\overset{e}{=} /P_s; p : w/P_{p:w}; D(t)/P_s(t,t); t = u/P_s(t,u) \vee (\neg P_s(t,t) \wedge \neg P_s(u,u))]$$

where, for a set of formulas Ψ, $\Psi[sy_1/sy'_1; \ldots; sy_n/sy'_n]$ denotes the simultaneous substitution of sy'_i for sy_i in all formulas of Ψ (while possibly instantiating the meta-variables t and u). Finally *largerThenKerH* stands for

$$\begin{array}{l} \bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s). h(x) \overset{e}{=} h(y) \implies P_s(x,y) \\ \bigwedge \wedge_{p_w \in \Sigma^M} \forall \bar{x} : \iota(w). \iota(p : w)(\bar{x}) \implies P_{p:w}(\bar{x}) \end{array}$$

Proposition 1. *The models of the nodes N and N' are the same.*

6 An Example: Reversing Lists

The example we are going to present is a standard specification of lists with empty lists, composition and reversal. We want to prove that by reversing a list twice we obtain the original list. Since Maude syntax does not support marking sentences of a theory as theorems, in Maude we would normally write a view (*PROVEIDEM* in Fig. 3, left side) from a theory containing the theorem (*REVIDEM*) to the module with the axioms defining *reverse* (*LISTREV*).

The first advantage the integration of Maude in HETS brings in is that we can use heterogeneous CASL structuring mechanisms and the *%implies* annotation to obtain the same development graph in a shorter way – see the right side of Fig. 3. Notice that we made the convention in HETS to have non-persistent freeness for Maude specifications, modifying thus the usual institution-independent semantics of the freeness construct.

For our example, the development calculus rules are applied as follows. First, the library is translated to CASL; during this step, Maude non-persistent free links are normalized. The next step is to normalize CASL free links, using Freeness rule. We then apply the Normal-Form rule which introduces normal forms for the nodes with incoming hiding links (introduced at the previous step) and

```
fmod MYLIST is
 sorts Elt List .
 subsort Elt < List .
 op nil : -> List [ctor] .
 op __ : List List -> List
    [ctor assoc id: nil] .
endfm
fmod MYLISTREV is
 pr MYLIST .
 op reverse : List -> List .
 var L : List .
 var E : Elt .
 eq reverse(nil) = nil .
 eq reverse(E L) = reverse(L) E .
endfm
fth REVIDEM  is
 pr MYLIST .
 op reverse : List -> List .
 var L : List .
 eq reverse(reverse(L)) = L .
endfth
view PROVEIDEM
 from REVIDEM to MYLISTREV is
sort List to List .
op reverse to reverse .
endv
```

logic MAUDE
spec PROVEIDEM =
  free
  {*sorts Elt List .*
   subsort Elt < *List .*
   op nil : −> *List* [*ctor*] .
   op __ : *List List* −> *List*
  [*ctor assoc id*: *nil*] .
  }
then {*op reverse* : *List* −> *List .*
   var L : *List .*   *var E* : *Elt .*
   eq reverse(*nil*) = *nil .*
   eq reverse(*E L*) = *reverse*(*L*) *E .*
  } **then %implies**
  {*var L* : *List .*
   eq reverse(*reverse*(*L*)) = *L .*
  }

Fig. 3. Lists with reverse, in Maude (left) and CASL (right) syntax

then Theorem-Hide-Shift rule which moves the target of any theorem link targeting a node with incoming hiding links to the normal form of the latter. Calling then Proofs/Automatic, the proof obligation is delegated to the normal form node.

In this node, we now have a proof goal for a second-order theory. It can be discharged using the interactive theorem prover Isabelle/HOL [17]. We have set up a series of lemmas easing such proofs. First of all, normalization of freeness introduces sorts for the free model which are axiomatized to be the homomorphic image of a set of the absolutely free (i.e. term) model. A transfer lemma (that exploits surjectivity of the homomorphism) enables us to transfer any proof goal from the free model to the absolutely free model. Since the absolutely free model is term generated, we can use induction proofs here. For the case of datatypes with total constructors (like lists), we prove by induction that the homomorphism is total as well. Two further lemmas on lists are proved by induction: (1) associativity of concatenation and (2) the reverse of a concatenation is the concatenation (in reverse order) of the reversed lists. This infrastructure then allows us to demonstrate (again by induction) that $reverse(reverse(L)) = L$.

While proof goals in Horn clause form often can be proved with induction, other proof goals like the inequality of certain terms or extensionality of sets cannot. Here, we need to prove inequalities or equalities with more complex premises, and this calls for use of the special axiomatization of the kernel of the homomorphism. This axiomatization is rather complex, and we are currently setting up the infrastructure for easing such proofs in Isabelle/HOL.

7 Conclusions and Future Work

We have presented in this paper how Maude has been integrated into HETS, a parsing, static analysis and proof management tool that combines various tools for different specification languages. To achieve this integration, we consider preordered algebra semantics for Maude and define an institution comorphism from Maude to CASL. This integration allows to prove properties of Maude specifications like those expressed in Maude views. We have also implemented a normalization of the development graphs that allows us to prove freeness constraints. We have used this transformation to connect Maude to Isabelle [17], a Higher Order Logic prover, and have demonstrated a small example proof about reversal of lists. Moreover, this encoding is suited for proofs of e.g. extensionality of sets, which require first-order logic, going beyond the abilities of existing Maude provers like ITP.

Since interactive proofs are often not easy to conduct, future work will make proving more efficient by adopting automated induction strategies like rippling [7]. We also have the idea to use the automatic first-order prover SPASS for induction proofs by integrating special induction strategies directly into HETS.

We have also studied the possible comorphisms from CASL to Maude. We distinguish whether the formulas in the source theory are confluent and terminating or not. In the first case, that we plan to check with the Maude termination [8] and confluence checker [9], we map formulas to equations, whose execution in Maude is more efficient, while in the second case we map formulas to rules.

Finally, we also plan to relate HETS' Modal Logic and Maude models in order to use the Maude model checker [3, Chapter 13] for linear temporal logic.

Acknowledgments. We wish to thank Francisco Durán for discussions regarding freeness in Maude, Martin Kühl for cooperation on the implementation of the theory presented here, and Maksym Bortin for help with the Isabelle proofs. This work has been supported by the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe), the German Research Council (DFG) under grant KO-2428/9-1 and the MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01).

References

1. Bidoit, M., Mosses, P.D.: CASL User Manual. LNCS (IFIP Series), vol. 2900. Springer, Heidelberg (2004)
2. Bouhoula, A., Jouannaud, J.-P., Meseguer, J.: Specification and proof in membership equational logic. Theoretical Computer Science 236(1-2), 35–132 (2000)

3. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
4. Clavel, M., Palomino, M., Riesco, A.: Introducing the ITP tool: a tutorial. Journal of Universal Computer Science 12(11), 1618–1650 (2006); Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages
5. Mosses, P.D. (ed.): CASL Reference Manual. LNCS, vol. 2960. Springer, Heidelberg (2004)
6. Diaconescu, R.: Institution-independent Model Theory. Birkhäuser, Basel (2008)
7. Dixon, L., Fleuriot, J.D.: Higher order rippling in ISAplanner. In: Slind, K., Bunker, A., Gopalakrishnan, G. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 83–98. Springer, Heidelberg (2004)
8. Durán, F., Lucas, S., Meseguer, J.: MTT: The Maude Termination Tool (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 313–319. Springer, Heidelberg (2008)
9. Durán, F., Meseguer, J.: A Church-Rosser checker tool for conditional order-sorted equational maude specifications. In: Ölveczky, P.C. (ed.) WRLA 2010. LNCS, vol. 6381, pp. 69–85. Springer, Heidelberg (2010)
10. Futatsugi, K., Diaconescu, R.: CafeOBJ Report. AMAST Series. World Scientific, Singapore (1998)
11. Goguen, J., Roşu, G.: Institution morphisms. Formal Aspects of Computing 13, 274–307 (2002)
12. Goguen, J.A., Burstall, R.M.: Institutions: Abstract model theory for specification and programming. Journal of the Association for Computing Machinery 39, 95–146 (1992)
13. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science 96(1), 73–155 (1992)
14. Meseguer, J.: Membership algebra as a logical framework for equational specification. In: Parisi-Presicce, F. (ed.) WADT 1997. LNCS, vol. 1376, pp. 18–61. Springer, Heidelberg (1998)
15. Mossakowski, T., Autexier, S., Hutter, D.: Development graphs - proof management for structured specifications. Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques 67(1-2), 114–145 (2006)
16. Mossakowski, T., Maeder, C., Lüttich, K.: The Heterogeneous Tool Set. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 519–522. Springer, Heidelberg (2007)
17. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. LNCS, vol. 2283. Springer, Heidelberg (2002)
18. Reichel, H.: Initial computability, algebraic specifications, and partial algebras. Oxford University Press, Inc, New York (1987)

Integrating Maude into Hets

Mihai Codescu, Till Mossakowski, Adrián Riesco, and Christian Maeder

Technical Report 07/10

Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid

September 2010
(Revised April 7, 2011)

Abstract

Maude modules can be understood as models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. However, Maude lacks the formal tools to perform some of these analyses and thus they can only be done by hand. The Heterogeneous Tool Set HETS is an institution-based combination of different logics and corresponding rewriting, model checking, and proof tools. We present in this paper an integration of Maude into HETS that allows to use the logics and tools already integrated in HETS with Maude specifications. To achieve such integration we have defined an institution for Maude based on preordered algebras and a comorphism between Maude and CASL, the central logic in HETS.

Keywords: rewriting logic, heterogeneous specifications, Maude, CASL

Contents

1 Introduction

Maude [7] is a high-level language and high-performance system supporting both equational and rewriting logic computation for a wide range of applications. Maude modules correspond to specifications in rewriting logic, a simple and expressive logic which allows the representation of many models of concurrent and distributed systems. The key point is that there are three different uses of Maude modules:

1. As programs, to implement some application. We may have chosen Maude because its features make the programming task easier and simpler than other languages.

2. As formal executable specifications, that provide a rigorous mathematical model of an algorithm, a system, a language, or a formalism. Because of the agreement between operational and mathematical semantics, this mathematical model is at the same time executable. Therefore, we can use it as a precise prototype of our system to simulate its behavior.

3. As models that can be formally analyzed and verified with respect to different properties expressing various formal requirements. For example, we may want to prove that our Maude module terminates; or that a given function, equationally defined in the module, satisfies some properties expressed as first-order formulas.

However, when we follow this last approach we find that, although Maude can automatically perform analyses like model checking of temporal formulas or verification of invariants, other formal analyses have to be done "by hand," thus disconnecting the real Maude code from its logical meaning. Although some efforts, like the Inductive Theorem Prover [9], have been dedicated to palliate this problem, they are restricted to inductive proofs in Church-Rosser equational theories, and they lack the generality to deal with all the features of Maude. Hence, it would be useful to connect Maude to other systems in a correct and general way in order to these systems to prove properties in Maude specifications. Moreover, when designing complex systems it is usually required to use different formalisms to define their different parts such as databases, specifications, or real-time mechanisms. For this reason, it would also be interesting to have a framework where different tools can be used and related to each other.

Since it is clear that no single language can suit all purposes, a original approach was followed to develop the Common Algebraic Specification Language, CASL [1]. Its development was proposed by the Common Framework Initiative for algebraic specification and development, CoFI [19], that wanted to unify the different algebraic languages available, incorporating the main features of all these languages and fixing its syntax and semantics. The aim of the initiative was to create a language for specification of functional requirements; formal development of software; relation of specifications to informal requirements and implemented code; prototyping, theorem-proving, and formal testing; and tool interoperability. Following these ideas CASL was designed as a language based on first-order logic and induction by means of constructor constraints.

However, CASL was not thought as a standalone language, but as the heart of a family of languages, some of them obtained by restricting CASL and some other obtained by extending it. Broadening this idea, it is also possible to relate other languages independent from CASL with it; in this way it would be possible to use other tools (and hence other logics) with CASL specifications and vice versa. This is the aim of the Heterogeneous Tool Set (HETS) [29], an integration tool that combines different logics and corresponding rewriting, model checking, and proof tools with CASL at his heart. The correctness of the integration in this tool is granted because the tools introduced are formalized as institutions and the translations (ideally to CASL or any of its sublogics) as institution comorphisms. The reasons for using HETS are:

- It is formal, that is, it allows the user to reason about the mathematical properties of his specifications. This feature makes it much more adequate for our purposes than the Unified Modeling Language UML [3], probably the best known system of this kind.

- It is multilateral, in the sense that the specifications can be introduced in any of the logics supported by HETS, while other approaches, like the PROSPER toolkit [10], which provides several decision procedures and model checkers based in the theorem prover HOL98 [33], only provide one logic, and all specifications must be translated to it before using the tool.

- It focuses on codings between logics, unlike other approaches that focus on codings between theories as OMDOC [20], an ontology language for mathematics. The former allows to reason about different elements in different logics, while the latter only permits to reason about different elements in the same logic.

- Several tools have already been integrated into HETS, including the SAT solvers zChaff [24] and MiniSat [15], the automated provers SPASS [39], Vampire [37], and Darwin [16], and the interactive provers Isabelle [32] and VSE [2].

In this paper, we describe an integration of Maude into HETS from which we expect several benefits. On the one hand, Maude will be the first dedicated rewriting engine that is integrated into HETS (so far, only the rewriting engine of Isabelle is integrated, which however is quite specialized towards higher-order proofs). On the other hand, certain features of the Maude module system like views lead to proof obligations that cannot be checked with Maude—HETS will be the suitable framework to prove them, using the above mentioned proof tools; with our approach, we cover arbitrary first-order properties (also written in logics different from Maude), and open the door to automated induction strategies such as those of ISAplanner [12].

The rest of the paper is organized as follows. After briefly introducing rewriting logic in Section 2 and HETS in Section 3, Section 4 describes the institution we have defined for Maude and the comorphism from this institution to CASL. Section 5 shows how development graphs for Maude specifications are built, and then how they are normalized to deal with freeness constraints. Section 6 illustrates the integration of Maude into Hets with the help of an example, while Section 7 outlines the implementation of the integration. Section 8 concludes and outlines some future work.

2 Rewriting logic and Maude

As mentioned in the introduction, Maude modules are executable rewriting logic specifications. Rewriting logic [22] is a logic of change very suitable for the specification of concurrent systems that is parameterized by an underlying equational logic, for which Maude uses *membership equational logic* [5, 23], which, in addition to equations, allows the statement of membership axioms characterizing the elements of a sort. In the following sections we present both logics and how their specifications are represented as Maude modules.

2.1 Membership equational logic

A *signature* in membership equational logic is a triple (K, Σ, S) (just Σ in the following), with K a set of *kinds*, $\Sigma = \{\Sigma_{k_1 \ldots k_n, k}\}_{(k_1 \ldots k_n, k) \in K^* \times K}$ a many-kinded signature, and $S = \{S_k\}_{k \in K}$ a pairwise disjoint K-kinded family of sets of *sorts*. The kind of a sort s is denoted by $[s]$. We write $T_{\Sigma,k}$ and $T_{\Sigma,k}(X)$ to denote respectively the set of ground Σ-terms with kind k and of Σ-terms with kind k over variables in X, where $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ is a set of K-kinded variables. Intuitively, terms with a kind but without a sort represent undefined or error elements.

The atomic formulas of membership equational logic are either *equations* $t = t'$, where t and t' are Σ-terms of the same kind, or *membership axioms* of the form $t : s$, where the term t has kind k and $s \in S_k$. *Sentences* are universally-quantified Horn clauses of the form $(\forall X)\, A_0 \Leftarrow A_1 \wedge \ldots \wedge A_n$, where each A_i is either an equation or a membership axiom, and X is a set of K-kinded variables containing all the variables in the A_i. A *specification* is a pair (Σ, E), where E is a set of sentences in membership equational logic over the signature Σ.

Models of membership equational logic specifications are Σ-*algebras* $\mathcal{A}$ consisting of a set A_k for each kind $k \in K$, a function $A_f : A_{k_1} \times \cdots \times A_{k_n} \longrightarrow A_k$ for each operator $f \in \Sigma_{k_1 \ldots k_n, k}$, and a subset $A_s \subseteq A_k$ for each sort $s \in S_k$. The meaning $[\![t]\!]_{\mathcal{A}}$ of a term t in an algebra $\mathcal{A}$ is inductively defined as usual. Then, an algebra $\mathcal{A}$ satisfies an equation $t = t'$ (or the equation holds in the algebra), denoted $\mathcal{A} \models t = t'$, when both terms have the same meaning: $[\![t]\!]_{\mathcal{A}} = [\![t']\!]_{\mathcal{A}}$. In the same way, satisfaction of a membership is defined as: $\mathcal{A} \models t : s$ when $[\![t]\!]_{\mathcal{A}} \in A_s$.

A membership equational logic specification (Σ, E) has an initial model $\mathcal{T}_{\Sigma/E}$ whose elements are E-equivalence classes of terms $[t]$. We refer to [5, 23] for a detailed presentation of (Σ, E)-algebras, sound and complete deduction rules, as well as the construction of initial and free algebras.

2.2 Maude functional modules

Maude functional modules [7, Chapter 4], introduced with syntax `fmod ... endfm`, are executable membership equational specifications and their semantics is given by the corresponding initial membership algebra in the class of algebras satisfying the specification. The membership equational logic specifications that we consider are assumed to satisfy the executability requirements of confluence, termination, and sort-decreasingness.

In a functional module we can declare sorts (by means of keyword `sort(s)`); subsort relations between sorts (`subsort`); operators (`op`) for building values of these sorts, giving the sorts of their arguments and result, and which may have attributes such as being associative (`assoc`) or commutative (`comm`), for example; memberships (`mb`) asserting that a term has a sort; and equations (`eq`) identifying terms. Both memberships and equations can be conditional (`cmb` and `ceq`).

Maude does automatic kind inference from the sorts declared by the user and their subsort relations. Kinds are *not* declared explicitly, and correspond to the connected components of the subsort relation. The kind corresponding to a sort `s` is denoted `[s]`. For example, if we have sorts `Nat` for natural numbers and `NzNat` for nonzero natural numbers with a subsort `NzNat < Nat`, then `[NzNat]` = `[Nat]`.

An operator declaration like

```
op _div_ : Nat NzNat -> Nat .
```

is logically understood as a declaration at the kind level

```
op _div_ : [Nat] [Nat] -> [Nat] .
```

together with the conditional membership axiom

```
cmb N div M : Nat if N : Nat and M : NzNat .
```

A subsort declaration `NzNat < Nat` is logically understood as the conditional membership axiom

```
cmb N : Nat if N : NzNat .
```

2.3 Rewriting logic

Rewriting logic extends equational logic by introducing the notion of *rewrites* corresponding to transitions between states; that is, while equations are interpreted as equalities and therefore they are symmetric, rewrites denote changes which can be irreversible.

A rewriting logic specification, or *rewrite theory*, has the form $\mathcal{R} = (\Sigma, E, R)$, where (Σ, E) is an equational specification and R is a set of *rules* as described below. From this definition, one can see that rewriting logic is built on top of equational logic, so that rewriting logic is parameterized with respect to the version of the underlying equational logic; in our case, Maude uses membership equational logic, as described in the previous sections. A rule in R has the general conditional form[1]

$$(\forall X)\ t \Rightarrow t' \Leftarrow \bigwedge_{i=1}^{n} u_i = u_i' \wedge \bigwedge_{j=1}^{m} v_j : s_j \wedge \bigwedge_{k=1}^{l} w_k \Rightarrow w_k'$$

where the head is a rewrite and the conditions can be equations, memberships, and rewrites; both sides of a rewrite must have the same kind. From these rewrite rules, one can deduce rewrites of the form $t \Rightarrow t'$ by means of general deduction rules introduced in [22] (for a generalization see also [6]).

Models of rewrite theories are called *$\mathcal{R}$-systems*. Such systems are defined as categories that possess a (Σ, E)-algebra structure, together with a natural transformation for each rule in the set R. More intuitively, the idea is that we have a (Σ, E)-algebra, as described in Section 2.1, with transitions between the elements in each set A_k; moreover, these transitions must satisfy several additional requirements, including that there are identity transitions for each element, that transitions can be sequentially composed, that the operations in the signature Σ are also appropriately defined for the transitions, and that we have enough transitions corresponding to the rules in R. Then, if we keep in this context the notation $\mathcal{A}$ to denote an $\mathcal{R}$-system, a rewrite $t \Rightarrow t'$ is satisfied by $\mathcal{A}$, denoted $\mathcal{A} \models t \Rightarrow t'$, when there is a transition $[\![t]\!]_{\mathcal{A}} \to_{\mathcal{A}} [\![t']\!]_{\mathcal{A}}$ in the system between the corresponding meanings of both sides of the rewrite, where $\to_{\mathcal{A}}$ will be our notation for such transitions.

The rewriting logic deduction rules introduced in [22] are sound and complete with respect to this notion of model. Moreover, they can be used to build initial and free models; see [22] for details.

[1]There is no need for the condition listing first equations, then memberships, and then rewrites: this is just a notational abbreviation, they can be listed in any order.

2.4 Maude system modules

Maude system modules [7, Chapter 6], introduced with syntax `mod ... endm`, are executable rewrite theories and their semantics is given by the initial system in the class of systems corresponding to the rewrite theory. A system module can contain all the declarations of a functional module and, in addition, declarations for rules (`rl`) and conditional rules (`crl`).

The executability requirements for equations and memberships in a system module are the same as those of functional modules, namely, confluence, termination, and sort-decreasingness. With respect to rules, the satisfaction of all the conditions in a conditional rewrite rule is attempted sequentially from left to right, solving rewrite conditions by means of search; for this reason, we can have new variables in such conditions but they must become instantiated along this process of solving from left to right (see [7] for details). Furthermore, the strategy followed by Maude in rewriting with rules is to compute the normal form of a term with respect to the equations before applying a rule. This strategy is guaranteed not to miss any rewrites when the rules are *coherent* with respect to the equations [38, 7]. In a way quite analogous to confluence, this coherence requirement means that, given a term t, for each rewrite of it using a rule in R to some term t', if u is the normal form of t with respect to the equations and memberships in E, then there is a rewrite of u with some rule in R to a term u' such that $u' =_E t'$ (that is, the equation $t' = u'$ can be deduced from E).

2.5 Advanced features

In addition to the modules presented thus far, we present in this section some other Maude features that will be used throughout this paper. More information on these topics can be found in [7].

2.5.1 Module operations

To ease the specification of large systems, Maude provides several mechanisms to structure its modules. We describe in this section these structuring mechanisms, that will be used later to build the development graphs in HETS.

Maude modules can import other modules in three different modes:

- The `protecting` mode (abbreviated as `pr`) indicates that *no junk and no confusion* can be added to the imported module, where junk refers to new terms in canonical form while confusion implies that different canonical terms in the initial module are made equal by equations in the imported module.
- The `extending` mode (abbreviated as `ex`) indicates that junk is allowed but confusion is forbidden.
- The `including` mode (abbreviated as `inc`) allows both junk and confusion.

More specifically, these importation modes do not import modules but *module expressions* that, in addition to a single module identifier, can be:

- A summation of two module expressions ME_1 `+` ME_2, which creates a new module that includes all the information in its summands.
- A renaming ME `* (`*Renaming*`)`, where *Renaming* is a list of renamings. They can be renaming of sorts:

  `sort` $sort_1$ `to` $sort_2$ `.`

  of operators, distinguishing whether it renames all the operators with the given identifier (when the attributes are modified, only `prec`, `gather`, and `format` are allowed, see [7])

  `op` id_1 `to` id_2 `.`
  `op` id_1 `to` id_2 `[`*atts*`] .`

  or it renames the operators of the given arity:

  `op` id_1 : *arity* `->` *coarity* `to` id_2 `.`
  `op` id_1 : *arity* `->` *coarity* `to` id_2 `[`*atts*`] .`

or of labels:

`label` *label*$_1$ `to` *label*$_2$ `.`

2.5.2 Theories

Theories are used to declare module interfaces, namely the syntactic and semantic properties to be satisfied by the actual parameter modules used in an instantiation. As for modules, Maude supports two different types of theories: functional theories and system theories, with the same structure of their module counterparts, but with a different semantics. Functional theories are declared with the keywords `fth ... endfth`, and system theories with the keywords `th ... endth`. Both of them can have sorts, subsort relationships, operators, variables, membership axioms, and equations, and can import other theories or modules. System theories can also have rules. Although there is no restriction on the operator attributes that can be used in a theory, there are some subtle restrictions and issues regarding the mapping of such operators (see Section 2.5.3). Like functional modules, functional theories are membership equational logic theories, but they do not need to be Church-Rosser and terminating.

For example, we can define a theory for some processes. First, we indicate that a sort for processes is required:

```
fth PROCESS is
 pr BOOL .

 sort Process .
```

Then, we state that two operators, one updating the processes and another one checking whether a process has finished, have to be defined:

```
 op update : Process -> Process .
 op finished? : Process -> Bool .
```

Finally, we require an operator `_<_` over processes that is required to be irreflexive and transitive:

```
 vars X Y Z : Process .

 op _<_ : Process Process -> Bool .
 eq X < X = false [nonexec label irreflexive] .
 ceq X < Z = true if X < Y /\ Y < Z [nonexec label transitive] .
endfth
```

2.5.3 Views

We use views to specify how a particular target module or theory satisfies a source theory. In general, there may be several ways in which such requirements might be satisfied by the target module or theory; that is, there can be many different views, each specifying a particular interpretation of the source theory in the target. In the definition of a view we have to indicate its name, the source theory, the target module or theory, and the mapping of each sort and operator in the source theory. The source and target of a view can be any module expression, with the source module expression evaluating to a theory and the target module expression evaluating to a module or a theory. Each view declaration has an associated set of proof obligations, namely, for each axiom in the source theory it should be the case that the axiom's translation by the view holds true in the target. Since the target can be a module interpreted initially, verifying such proof obligations may in general require inductive proof techniques. Such proof obligations are not discharged or checked by the system.

The mappings allowed in views are:

- Mappings between sorts:

  `sort` *sort*$_1$ `to` *sort*$_2$ `.`

- Mappings between operators, where the user can specify the arity and coarity of the operators to disambiguate them:

`op` id_1 `to` id_2 `.`

`op` id_1 : *arity* `->` *coarity* `to` id_2 `.`

- In addition to these mappings, the user can map a term $term_1$, that can only be a single operator applied to variables, to any term $term_2$ in the target module, where the sorts of the variables in the first term have been translated by using the sort mappings. Note that in that case the arity of the operator in the source theory and the one in the target module can be different:

`op` $term_1$ `to term` $term_2$ `.`

Notice that we cannot map labels, and thus we cannot identify the statements in the theory with those in the target module.

We can now create a view `NatProcess` from the theory `PROCESS` in the previous section to `NAT`, the predefined module for natural numbers:

```
view NatProcess from PROCESS to NAT is
```

We need a sort in `NAT` to identify processes. We use `Nat`, the sort for natural numbers:

```
  sort Process to Nat .
```

Since we identify now processes with natural numbers, we can `update` a process by applying the successor function, which is declared as `s_` in `NAT`:

```
  op update to s_ .
```

We map the operator `finished?` in a different way: we create a term with this operator with a variable as argument, and it is mapped to a term in the syntax of the target module. In that case we consider a process has finished if it reaches `100`:

```
  op finished?(P:Process) to term P:Nat < 100 .
```

Since the `NAT` module already contains an operator `_<_`, it is not necessary to explicitly indicate the corresponding mapping, i.e., identity mappings for sorts and operators can be omitted when defining views.

2.5.4 Parameterized modules

Maude modules can be parameterized. A parameterized system module has syntax

`mod M{`$X_1 :: T_1, \ldots, X_n :: T_n$`} is ... endm`

with $n \geq 1$. Parameterized functional modules have completely analogous syntax.

The `{`$X_1 :: T_1, \ldots, X_n :: T_n$`}` part is called the interface, where each pair $X_i :: T_i$ is a parameter, each X_i is an identifier—the parameter name or parameter label—, and each T_i is an expression that yields a theory—the parameter theory. Each parameter name in an interface must be unique, although there is no uniqueness restriction on the parameter theories of a module. The parameter theories of a functional module must be functional theories.

In a parameterized module M, all the sorts and statement labels coming from theories in its interface must be qualified by their names. Thus, given a parameter $X_i :: T_i$, each sort S in T_i must be qualified as $X_i\$S$, and each label l of a statement occurring in T_i must be qualified as $X_i\$l$. In fact, the parameterized module M is flattened as follows. For each parameter $X_i :: T_i$, a renamed copy of the theory T_i, called $X_i :: T_i$ is included. The renaming maps each sort S to $X_i\$S$, and each label l of a statement occurring in T_i to $X_i\$l$. The renaming has no effect on importations of modules. Thus, if T_i includes a theory T', when the renamed theory $X_i :: T_i$ is created and included into M, the renamed theory $X_i :: T'$ will also be created and included into $X_i :: T_i$. However, the renaming will have no effect on modules imported by either the T_i or T'; for example, if `BOOL` is imported by one of these theories, it is not renamed, but imported in the same way into M. Moreover, sorts declared in parameterized modules can also be parameterized, and these may duplicate, omit, or reorder parameters.

The parameters in parameterized modules are bound to the formal parameters by *instantiation.* The instantiation requires a view from each formal parameter to its corresponding actual parameter. Each such view is then used to bind the names of sorts, operators, etc. in the formal parameters to the corresponding sorts, operators (or expressions), etc. in the actual target. The instantiation of a parameterized module must be made with views explicitly defined previously.

We can define a parameterized module for multisets of the processes shown in Section 2.5.3. This module defines the sort `MSet{X}` for multisets, which is a supersort of `Process`:

```
fmod PROCESS_MSET{X :: PROCESS} is
 sort MSet{X} .
 subsort X$Process < MSet{X} .
```

The constructors of multisets are `empty` for the empty multiset and the juxtaposition operator `__` for bigger multisets:

```
 op empty : -> MSet{X} [ctor] .
 op __ : MSet{X} MSet{X} -> MSet{X} [ctor assoc comm id: empty] .
```

We can also use the operators declared in the view. For example, we can remove a process from the multiset if it is finished:

```
 var P : X$Process .
 var MS : MSet{X} .

 ceq P MS = MS if finished?(P) .
endfm
```

We can use the view `NatProcess` to instantiate this parameterized module and create multisets of processes identified as natural numbers.

```
fmod NAT_PROCSES_MSET is
 pr PROCESS_MSET{NatProcess} .
endfm
```

3 Hets

The central idea of Hets [26, 28, 29, 30] is to provide a general logic integration and proof management framework. One can think of Hets acting like a motherboard where different expansion cards can be plugged in, the expansion cards here being individual logics (with their analysis and proof tools) as well as logic translations. The benefit of plugging in a new logic and tool such as Maude into the Hets motherboard is the gained interoperability with the other logics and tools available in Hets; for example, we can now compose the translation from Maude to Casl with the corresponding translation to Isabelle to prove properties in Maude specifications.

Figure 1 shows how, by adding up different tools for single logics, we obtain a bigger tool which supports all of them by (i) providing a small set of mechanisms specific for each tool, (ii) relating the logics in such a way that all of them are connected, and (iii) using these relations to translate the initial specification to another logic where the required tool is available. The current logic graph for Hets is depicted in Figure 2 (where different colors stand for different subgraphs and relations we are not interested here), where each node corresponds to a different logic and the links are relations between them. Note that, as sketched in the introduction, the central node of this graph is Casl; new logics are expected to be translated to it in order to relate them with all the other logics in an easy and efficient way.

Following the ideas shown in Figure 1, the work that needs to be done for such an integration is to prepare both the Maude logic and tool so that it can act as an expansion card for Hets:

- On the side of the semantics, this means that the logic needs to be organized as an *institution* [18, 36].
- On the side of the tool, we must provide a parser and static analysis mechanisms for it, in such a way that its specifications can be translated into a common framework, which in our case consists of development graphs, a graphical representation of structured specifications.

Before describing how this aims have been achieved we present the Casl syntax and basic notions on institutions and comorphisms.

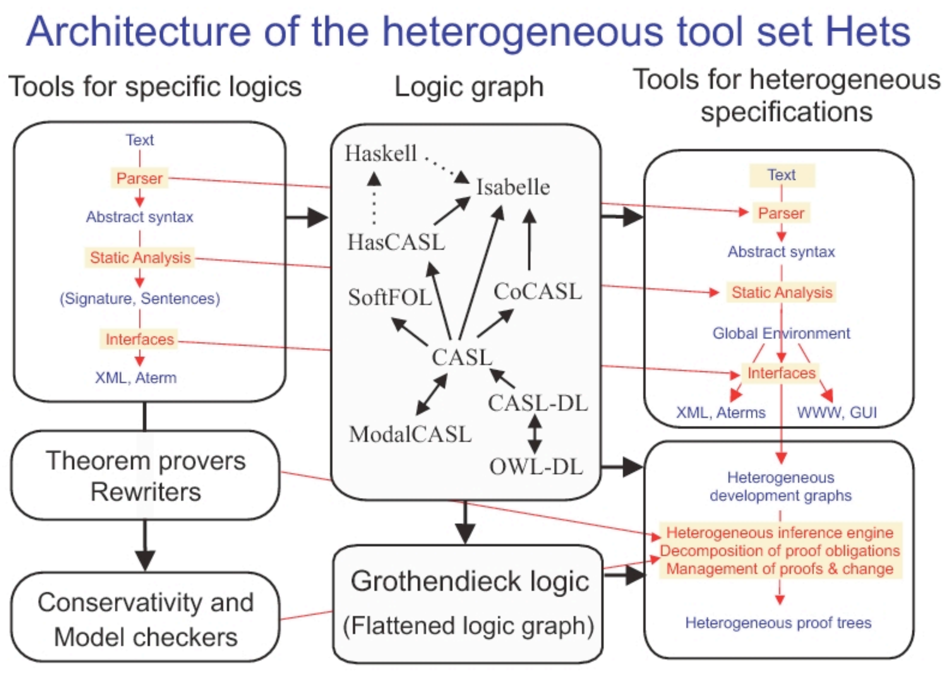

Architecture of the heterogeneous tool set Hets
Tools for specific logics
Logic graph
Tools for heterogeneous specifications
Text
Parser
Abstract syntax
Static Analysis
(Signature, Sentences)
Interfaces
XML, Aterm
Haskell
Isabelle
HasCASL
SoftFOL
CoCASL
CASL
CASL-DL
ModalCASL
OWL-DL
Theorem provers Rewriters
Conservativity and Model checkers
Grothendieck logic (Flattened logic graph)
Global Environment
XML, Aterms
WWW, GUI
Heterogeneous development graphs
Heterogeneous inference engine
Decomposition of proof obligations
Management of proofs & change
Heterogeneous proof trees

Figure 1: The Heterogeneous Tool Set

3.1 Casl

Since we intend to translate Maude specifications into Casl ones, it is worth to introduce the basic Casl notions in order to understand the translated specifications. Casl [1, 31, 4] is an expressive language for specifying requirements and design for conventional software based on both first-order logic and induction, the latter by means of constructor constraints. It was designed as a general-purpose algebraic specification language, subsuming many previous languages for formal specification, with the intention of basing its design on a critical selection of concepts and constructs from existing specification languages.

In this section we will explain the Casl syntax needed to understand a translated Maude specification; several other features, as well as shortcuts and variants of the syntax presented here, are explained in [4]. A Casl specification, with syntax `spec ... end` allows declarations of sorts, subsorts, operations, and predicates. As in the Maude case, sorts are interpreted as sets of terms, subsort declarations as embeddings. For example, we could start the specification of natural numbers in the `My_Nat` module, with sorts `Zero`, `NzNat`, and `Nat` (declared with `sorts`) and the standard subsorts (with syntax `<`) as follows:

```
spec My_Nat =
  sorts Zero < NzNat
  sorts NzNat < Nat
```

Constructors are declared with the keyword `type` followed by the type identifier and `::=`, while on the righthand side we place the constructors of the type separated by `|`. Thus, we define natural numbers as follows:

```
  type Zero ::= 0
  type NzNat ::= suc (Nat)
```

We can now define operators on these sorts with the keyword `ops` and indicating their arity and coarity:

```
  ops   __+__, min, max:  Nat * Nat ->  Nat;
```

where `__` (a double underscore) is a placeholder and allows the user to use mixfix notation in the usual way.

We can also define predicates as follows with the keyword `pred` and indicating their arity:

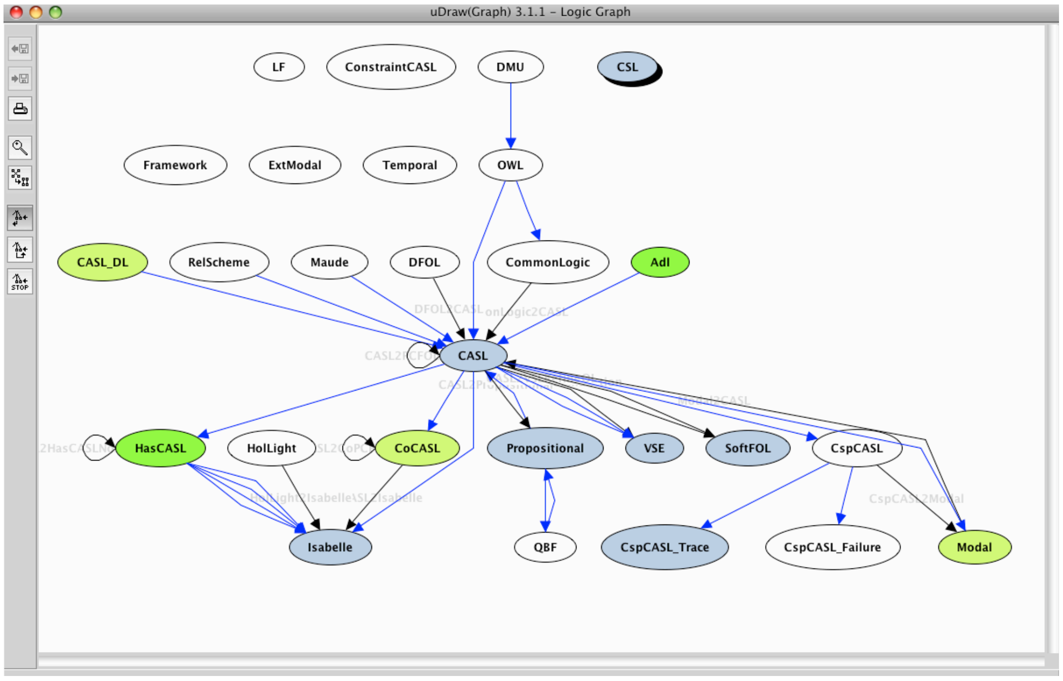


Figure 2: Current logic graph

```
preds __<=__, __>=__:   Nat * Nat;
```

Casl distinguishes between Boolean functions and predicates, since the former can take three values (`true`, `false`, and be undefined), while the latter can only take two: it holds or it does not hold. Once operations and predicates have been defined, their behavior is defined by means of first-order logic formulas, where the variables are universally quantified and that can be labeled by using annotations with syntax `%(label)%`:

```
forall n,m : Nat
 . 0 + n = n                       %(add1)%
 . suc(n) + m = suc(n + m)         %(add2)%

 . min(n,m) = n when n <= m else m
 . max(n,m) = m when n <= m else n

 . 0 <= n
 . suc(n) <= suc(m) <=> n <= m
 . n >= m <=> m <= n
```

In addition to this kind of formulas, we can also use memberships of the form *t in s*, indicating that the term *t* has sort *s*. For example, we could substitute the subsort declarations by:

```
forall n : Nat
 . 0 in Zero
 . not n = 0 => n in NzNat
```

Specifications can easily be extended with new declarations and axioms by using the keyword `then`. Combining this feature and the `%implies` annotation we can state in Casl that a formula or part of the specification (depending on the annotation being in a formula or in a `then`) is redundant and thus can be proved true with the information previously introduced. For example, we can define a new specification pointing out that addition is commutative by typing:

```
spec COMM_NAT =
        My_Nat
then %implies
  forall m,n : Nat
```

```
    . m + n = n + m
end
```

where we include the previous specification by writing its name after the equal, and where the `%implies` annotation indicates that all the formulas coming after it (in that case only one) must hold given the formulas in `My_Nat`, that is, it generates *proof obligations*.

3.2 Institutions and comorphisms

Before describing institutions we recall the notions of *category*, *small category*, *dual category*, *functor*, and *natural transformation* [34].

A *category* $\mathbf{C}$ consists of:

- a class $|\mathbf{C}|$ of *objects*;
- a class $hom(\mathbf{C})$ of *morphisms* (also known as *arrows*), between the objects;
- operations assigning to each morphism f an object $dom(f)$, its *domain*, and an object $cod(f)$, its *codomain* (we write $f : A \to B$ to indicate that $dom(f) = A$ and $cod(f) = B$);
- a composition operator assigning to each pair of morphisms f and g, with $cod(f) = dom(g)$, a *composite* morphism $g \circ f : dom(f) \to cod(g)$, satisfying the associative law: for any morphisms $f : A \to B$, $g : B \to C$, and $h : C \to D$, $h \circ (g \circ f) = (h \circ g) \circ f$; and
- for each object A, an identity morphism $id_A : A \to A$ satisfying the identity law: for any morphism $f : A \to B$, $id_B \circ f = f$ and $f \circ id_A = f$.

A category $\mathbf{C}$ is called *small* if both $|\mathbf{C}|$ and $hom(\mathbf{C})$ are sets and not proper classes.

For each category $\mathbf{C}$, its *dual category* $\mathbf{C}^{op}$ is the category that has the same objects as $\mathbf{C}$ and whose arrows are the opposites of the arrows in $\mathbf{C}$, that is, if $f : A \to B$ *in* $\mathbf{C}$, then $f : B \to A$ *in* $\mathbf{C}^{op}$. Composite and identity arrows are defined in the obvious way.

Let $\mathbf{C}$ and $\mathbf{D}$ be categories. A *functor* $\mathbf{F} : \mathbf{C} \to \mathbf{D}$ is a map taking each $\mathbf{C}$-object A to a $\mathbf{D}$-object $\mathbf{F}(A)$ and each $\mathbf{C}$-morphism $f : A \to B$ to a $\mathbf{D}$-morphism $\mathbf{F}(f) : \mathbf{F}(A) \to \mathbf{F}(B)$, such that for all $\mathbf{C}$-objects A and composable $\mathbf{C}$-morphisms f and g:

- $\mathbf{F}(id_A) = id_{\mathbf{F}(A)}$,
- $\mathbf{F}(g \circ f) = \mathbf{F}(g) \circ \mathbf{F}(f)$.

A *natural transformation* $\eta : \mathbf{F} \to \mathbf{G}$ between functors $\mathbf{F}, \mathbf{G} : \mathbf{A} \to \mathbf{B}$ associates to each $X \in |\mathbf{A}|$ a morphism $\eta_X : \mathbf{F}(X) \to \mathbf{G}(X) \in \mathbf{D}$ called the *component* of η at X, such that for every morphism $f : X \to Y \in \mathbf{A}$ we have $\eta_Y \circ \mathbf{F}(f) = \mathbf{G}(f) \circ \eta_X$.

It is worth mentioning two interesting categories: **Set**, the category whose objects are sets and whose morphisms between sets A and B are all functions from A to B; and **Cat**, the category whose objects are all small categories and whose morphisms are functors between them.

An *institution* consists of:

- a category **Sign** of *signatures*;
- a functor $\mathbf{Sen} : \mathbf{Sign} \to \mathbf{Set}$ giving a set $\mathbf{Sen}(\Sigma)$ of Σ-sentences for each signature $\Sigma \in |\mathbf{Sign}|$;
- A functor $\mathbf{Mod} : \mathbf{Sign}^{op} \to \mathbf{Cat}$, giving a category $\mathbf{Mod}(\Sigma)$ of Σ-*models* for each signature $\Sigma \in |\mathbf{Sign}|$; and
- for each signature $\Sigma \in |\mathbf{Sign}|$, a *satisfaction relation* $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathbf{Sen}(\Sigma)$ between models and sentences such that for any signature morphism $\sigma : \Sigma \to \Sigma'$, Σ-sentence $\varphi \in \mathbf{Sen}(\Sigma)$ and Σ'-model $M' \in |\mathbf{Mod}(\Sigma')|$:

$$M' \models_{\Sigma'} \mathbf{Sen}(\sigma)(\varphi) \iff \mathbf{Mod}(\sigma)(M') \models_\Sigma \varphi$$

  which is called the *satisfaction condition*.

It is also important to define, for our purposes in this work, the notion of *institution comorphism* [34]. Given two institutions $\mathcal{I} = (\mathbf{Sign}, \mathbf{Mod}, \mathbf{Sen}, \models)$ and $\mathcal{I}' = (\mathbf{Sign}', \mathbf{Mod}', \mathbf{Sen}', \models')$, an *institution comorphism* from $\mathcal{I}$ to $\mathcal{I}'$ consists of a functor $\Phi : \mathbf{Sign} \to \mathbf{Sign}'$, a natural transformation $\alpha : \mathbf{Sen} \Rightarrow \Phi; \mathbf{Sen}'$, and a natural transformation $\beta : \Phi; \mathbf{Mod}' \Rightarrow \mathbf{Mod}$ (where $\mathbf{F}; \mathbf{G}$ stands for functor composition in the diagrammatic order) such that the following satisfaction condition holds for each $\Sigma \in |\mathbf{Sign}|$, $\varphi \in |\mathbf{Sen}(\Sigma')|$, and $M' \in |\mathbf{Mod}'(\Phi(\Sigma))|$:

$$\beta_\Sigma(M') \models_\Sigma \varphi \iff M' \models'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi).$$

Institutions capture in a very abstract and flexible way the notion of a logical system, by leaving open the details of signatures, models, sentences (axioms), and satisfaction (of sentences in models). The *satisfaction condition* states that *truth is invariant under change of notation* (also called enlargement of context), which is captured by the notion of signature morphism (which leads to translations of sentences and reductions of models). See [18] for formal details.

Indeed, Hets has interfaces for plugging in the different components of an institution: signatures, signature morphisms, sentences, and their translation along signature morphisms. Recently, even (some) models and model reducts have been covered, although this is not needed here. Note, however, that the model theory of an institution (including model reducts and the satisfaction condition) is essential when relating different logics via institution comorphisms. The logical correctness of their use in multi-logic proofs is ensured by model-theoretic means.

For proof management, Hets uses *development graphs* [27]. They can be defined over an arbitrary institution, and they are used to encode structured specifications in various phases of the development. Roughly speaking, each node of the graph represents a theory. The links of the graph define how theories can make use of other theories. In this way, we represent complex specifications by representing each component (e.g. each module) as a node in the development graph and the relation between them (e.g. importations) as links.

A *development graph* is an acyclic, directed graph $\mathcal{DG} = \langle \mathcal{N}, \mathcal{L} \rangle$, where:

- $\mathcal{N}$ is a set of nodes. Each node $N \in \mathcal{N}$ is a pair (Σ^N, Φ^N) such that Σ^N is a signature and $\Phi^N \subseteq \mathbf{Sen}(\Sigma^N)$ is the set of **local axioms** of N.
- $\mathcal{L}$ is a set of directed links, so-called **definition links**, between elements of $\mathcal{N}$. Each definition link from a node M to a node N is either
  - **global** (denoted $M \xRightarrow{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or
  - **local** (denoted $M \xrightarrow{\sigma} N$), again annotated with a signature morphism $\sigma : \Sigma^M \to \Sigma^N$, or
  - **hiding** (denoted $M \xRightarrow[hide]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma^N \to \Sigma^M$ *going against the direction of the link*, or
  - **free** (denoted $M \xRightarrow[free]{\sigma} N$), annotated with a signature morphism $\sigma : \Sigma \to \Sigma^M$ *where* Σ *is a subsignature of* Σ^M.

In addition to these links we add a new link, denoted $M \xRightarrow[n.p.free]{\sigma} N$, that stands for non-persistent free links and will be used when dealing with `protecting` importations in Maude modules. However, these nodes are only used for Maude specifications and thus are nonstandard; we will show in Section 5.2 how these links and the associated nodes are transformed into a new graph that only uses standard constructions. Intuitively, these links indicate that no new elements can be added to the sorts, although they can be added to the kind.

Given a node M in a development graph $\mathcal{DG}$, its associated class $\mathbf{Mod}_{\mathcal{DG}}(M)$ of models (or M-models for short) is inductively defined to consist of those Σ^M-models m for which

1. m satisfies the local axioms Φ^M,
2. for each $N \xRightarrow{\sigma} M \in \mathcal{DG}$, $m|_\sigma$ is an N-model,
3. for each $N \xrightarrow{\sigma} M \in \mathcal{DG}$, $m|_\sigma$ satisfies the local axioms Φ^N,
4. for each $N \xRightarrow[hide]{\sigma} M \in \mathcal{DG}$, m has a σ-expansion m' (i.e. $m'|_\sigma = m$) that is an N-model, and

5. for each $N \xRightarrow[\mathit{free}]{\sigma} M \in \mathcal{DG}$, m is an N-model that is persistently σ-free in $\mathbf{Mod}(N)$. The latter means that for each N-model m' and each model morphism $h : m|_\sigma \to m'|_\sigma$, there exists a unique model morphism $h^\# : m \to m'$ with $h^\#|_\sigma = h$.

Complementary to definition links, which *define* the theories of related nodes, we introduce the notion of a *theorem link* with the help of which we are able to *postulate* relations between different theories. Theorem links are the central data structure to represent proof obligations arising in formal developments. Again, we distinguish between local and global theorem links (denoted by $N \overset{\sigma}{=\,=\,\Rightarrow} M$ and $N \overset{\sigma}{-\,-\,\succ} M$ respectively). We also need theorem links $N \underset{\mathit{hide}\ \theta}{\overset{\sigma}{=\,=\,\Rightarrow}} M$ (where for some Σ, $\theta : \Sigma \to \Sigma^N$ and $\sigma : \Sigma \to \Sigma^M$) involving hiding. The semantics of theorem links is given as follows:

Let $\mathcal{DG}$ be a development graph and N, M nodes in $\mathcal{DG}$.

- $\mathcal{DG}$ **implies** a global theorem link $N \overset{\sigma}{=\,=\,\Rightarrow} M$ (denoted $\mathcal{DG} \models N \overset{\sigma}{=\,=\,\Rightarrow} M$) iff for all $m \in \mathit{Mod}(M)$, $m|_\sigma \in \mathit{Mod}(N)$.
- $\mathcal{DG}$ **implies** a local theorem link $N \overset{\sigma}{-\,-\,\succ} M$ (denoted $\mathcal{DG} \vdash N \overset{\sigma}{-\,-\,\succ} M$) iff for all $m \in \mathit{Mod}(M)$, $m|_\sigma \models \phi$ for all $\phi \in \Phi^N$.
- $\mathcal{DG}$ **implies** a hiding theorem link $N \underset{\mathit{hide}\ \theta}{\overset{\sigma}{=\,=\,\Rightarrow}} M$ (denoted $\mathcal{DG} \models N \underset{\mathit{hide}\ \theta}{\overset{\sigma}{=\,=\,\Rightarrow}} M$) iff for all $m \in \mathit{Mod}(M)$, $m|_\sigma$ has a θ-expansion to some N-model.

We refer to [28, 26] for more details on how to use development graphs.

4 Relating the Maude and Casl logics

In this section, we will relate Maude and Casl at the level of logical systems. The structuring level will be considered in the next section by means of development graphs.

4.1 Maude

As we have seen in Section 2, Maude is an efficient tool for equational reasoning and rewriting. Methodologically, Maude specifications are divided into a specification of the data objects and a specification of some concurrent transition system, the states of which are given by the data part. Two logics have been introduced and studied in the literature for this binary relation: rewriting logic [22] and preordered algebra [17]. They essentially differ in the treatment of rewrites: whereas in rewriting logic, rewrites are named, and different rewrites between two given states (terms) can be distinguished (which corresponds to equipping each carrier set with a category of rewrites), in preordered algebra, only the existence of a rewrite does matter (which corresponds to equipping each carrier set with a preorder of rewritability).

Rewriting logic has been announced as the logic underlying Maude [7]. Maude modules lead to rewriting logic theories, which can be equipped with loose semantics (`fth`/`th` specifications) or initial/free semantics (`fmod`/`mod` specifications). Although rewriting logic is not given as an institution [11], a so-called specification frame (collapsing signatures and sentences into theories) would be sufficient for our purposes.

However, after a closer look at Maude and rewriting logic, we found out that de facto, the logic underlying Maude differs from the rewriting logic as defined in [22]. The reasons are:

1. In Maude, labels of rewrites cannot (and need not) be translated along signature morphisms. This means that *e.g. Maude views do not lead to theory morphisms in rewriting logic!*
2. Although labels of rewrites are used in traces of counterexamples, they play a subsidiary role, because they cannot be used in the linear temporal logic of the Maude model checker.

Especially the first reason completely rules out a rewriting logic-based integration of Maude into Hets: if a view between two modules is specified, Hets definitely needs a theory morphism underlying the view.[2]

[2]If the Maude designers would let (and force) users to specify the action of signature morphisms on rewrite labels, it would not be difficult to switch the Hets integration of Maude to being based on rewriting logic. In that case the labels would be taken into account, while the sentences would remain unchanged.

However, the Maude user does not need to provide the action of the signature morphism on labeled rewrites, and generally, there is more than one possibility to specify this action.

The conclusion is that, for the time being, the most appropriate logic to use for Maude is preordered algebra [17]. In this logic, rewrites are neither labeled nor distinguished, only their existence is important. This implies that Maude views lead to theory morphisms in the institution of preordered algebras. Moreover, this setting also is in accordance with the above observation that in Maude rewrite labels are not first-class citizens, but are mere names of sentences that are convenient for decorating tool output (e.g. traces of the model checker). Labels of sentences play a similar role in HETS, which perfectly fits here.

Actually, the switch from rewriting logic to preordered algebras has effects on the consequence relation, contrary to what is said in [22]. Consider the following Maude theory:

```
th A is
 sorts S T .
 op a : -> S .
 eq X:S = a .
 ops h k : S -> T .
 rl [r] : a => a .
 rl [s] : h(a) => k(a) .
endth
```

This logically implies $h(x) \Rightarrow k(x)$ in preordered algebra, but not in rewriting logic, since in the latter logic it is easy to construct models in which the naturality condition $r; k(r) = h(r); s$ fails to hold.

Thus, we will work with preordered algebra semantics for Maude. We will define an institution, that we will denote Maude^{pre}, which can be, like in the case of Maude's logic, parametric over the underlying equational logic. Following the Maude implementation, we have used membership equational logic [23]. Notice that the resulting institution Maude^{pre} is very similar to the one defined in the context of CafeOBJ [17, 11] for preordered algebra (the differences are mainly given by the discussion about operation profiles below, but this is only a matter of representation). This allows us to make use of some results without giving detailed proofs.

Signatures of Maude^{pre} are tuples $(K, F, \mathit{kind} : (S, \leq) \to K)$, where K is a set of *kinds*, kind is a function assigning a kind to each *sort* in the poset $(S, \leq)$, and F is a set of function symbols of the form $F = \{F_{k_1 \ldots k_n \to k} \mid k_i, k \in K\} \cup \{F_{s_1 \ldots s_n \to s} \mid s_i, s \in S\}$ such that if $f \in F_{s_1 \ldots s_n \to s}$, there is a symbol $f \in F_{\mathit{kind}(s_1) \ldots \mathit{kind}(s_n) \to \mathit{kind}(s)}$. Notice that there is actually no essential difference between our putting operation profiles on sorts into the signatures and Meseguer's original formulation putting them into the sentences.

Given two signatures $\Sigma_i = (K_i, F_i, \mathit{kind}_i)$, $i \in \{1, 2\}$, a signature morphism $\phi : \Sigma_1 \to \Sigma_2$ consists of a function $\phi^{kind} : K_1 \to K_2$ which preserves $\leq_1$, a function between the sorts $\phi^{sort} : S_1 \to S_2$ such that $\phi^{sort}; \mathit{kind}_2 = \mathit{kind}_1; \phi^{kind}$ and the subsorts are preserved, and a function $\phi^{op} : F_1 \to F_2$ which maps operation symbols compatibly with the types. Moreover, the overloading of symbol names must be preserved, i.e. the name of $\phi^{op}(\sigma)$ must be the same both when mapping the operation symbol σ on sorts and on kinds. With composition defined component-wise, we get the category of signatures.

For a signature Σ, a model M interprets each kind k as a preorder $(M_k, \leq)$, each sort s as a subset M_s of $M_{\mathit{kind}(s)}$ that is equipped with the induced preorder, with M_s a subset of $M_{s'}$ if $s < s'$, and each operation symbol $f \in F_{k_1 \ldots k_n, k}$ as a function $M_f : M_{k_1} \times \ldots \times M_{k_n} \to M_k$ which has to be monotonic and such that for each function symbol f on sorts, its interpretation must be a restriction of the interpretation of the corresponding function on kinds. For two Σ-models A and B, a homomorphism of models is a family $\{h_k : A_k \to B_k\}_{k \in K}$ of preorder-preserving functions which is also an algebra homomorphism and such that $h_{\mathit{kind}(s)}(A_s) \subseteq B_s$ for each sort s.

The sentences of a signature Σ are Horn clauses of the form $\forall X \,.\, A \Leftarrow A_1 \ \wedge \ \cdots \ \wedge \ A_n$, where the set of variables X used for quantification is K-sorted, built with three types of atoms: equational atoms $t = t'$, membership atoms $t : s$, and rewrite atoms $t \Rightarrow t'$, where t, t' are Σ-terms and s is a sort in S.[3]

Given a Σ-model M and a valuation $\eta = \{\eta_k\}_{k \in K}$, i.e., a K-sorted family of functions assigning elements in M to variables, M^η_t is inductively defined as usual. An equational atom $t = t'$ holds in M if $M^\eta_t = M^\eta_{t'}$, a membership atom $t : s$ holds when M^η_t is an element of M_s, and a rewrite atom $t \Rightarrow t'$ holds when $M^\eta_t \leq M^\eta_{t'}$. Satisfaction of atoms is extended to satisfaction of sentences in the obvious way. Finally, we use $M, \eta \models A$ to indicate that the model M satisfies the sentence A under the valuation η.

[3]Note that this is slightly more general than Maude's version, because rewrite conditions are allowed in equations and membership axioms.

We prove that the satisfaction condition holds for atoms, and then the extension to Horn clauses is straightforward. To do so, we will use the following lemma:

Lemma 1 *Given a signature morphism $\sigma : \Sigma \to \Sigma'$, inducing the function $\sigma : \mathbf{Sen}(\Sigma) \to \mathbf{Sen}(\Sigma')$ and the functor $\_|_\sigma : \mathbf{Mod}(\Sigma') \to \mathbf{Mod}(\Sigma)$, a Σ'-model M, the sets of variables $X = \{x_1 : k_1, \ldots, x_l : k_l\}$ and $X' = \{x_1 : \sigma(k_1), \ldots, x_l : \sigma(k_l)\}$, with $k_i \in K$, $1 \le i \le l$, a valuation $\eta : X \to M|_\sigma$, which induces a valuation $\eta' : X' \to M$ with $\eta'(x) = \eta(x)$, and a Σ-term t with variables in X, we have $M^{\eta'}_{\sigma(t)} = (M|_\sigma)^\eta_t$.*

Proof. By structural induction on t. For $t = x$ a variable on kinds it is trivial because $\sigma(x) = x$ and $\eta(x) = \eta'(x)$. Similarly, for $t = c$ a constant it is trivial by applying the definition of morphism to operators. If $t = f(t_1, \ldots, t_n)$, then we have $M^{\eta'}_{\sigma(t_i)} = (M|_\sigma)^\eta_{t_i}$, $1 \le i \le n$, by induction hypothesis, and

$$\begin{array}{rcll}
M^{\eta'}_{\sigma(f(t_1,\ldots,t_n))} & = & M^{\eta'}_{\sigma(f)(\sigma(t_1),\ldots,\sigma(t_n))} & \text{(by definition of } \sigma \text{ on terms)} \\
& = & M_{\sigma(f)}(M^{\eta'}_{\sigma(t_1)}, \ldots, M^{\eta'}_{\sigma(t_n)}) & \text{(meaning of the term in the model)} \\
& = & M_{\sigma(f)}((M|_\sigma)^\eta_{t_1}, \ldots, (M|_\sigma)^\eta_{t_n}) & \text{(by induction hypothesis)} \\
& = & (M|_\sigma)_f((M|_\sigma)^\eta_{t_1}, \ldots, (M|_\sigma)^\eta_{t_n}) & \text{(by definition of } \sigma \text{ on models)} \\
& = & (M|_\sigma)^\eta_{f(t_1,\ldots,t_n)}. & \text{(meaning of the term in the model)}
\end{array}$$

□

We can use this result, combined with the bijective correspondence between η and η', to check the satisfaction condition for a Σ-equation $t = t'$:

$$\begin{array}{rcl}
M \models_{\Sigma'} \sigma(t = t') & \iff & M \models_{\Sigma'} \sigma(t) = \sigma(t') \\
& \iff & M, \eta' \models_{\Sigma'} \sigma(t) = \sigma(t') \text{ for all } \eta' \\
& \iff & M^{\eta'}_{\sigma(t)} = M^{\eta'}_{\sigma(t')} \text{ for all } \eta' \\
& \iff & (M|_\sigma)^\eta_t = (M|_\sigma)^\eta_{t'} \text{ for all } \eta \\
& \iff & M|_\sigma, \eta \models_\Sigma t = t' \text{ for all } \eta \\
& \iff & M|_\sigma \models_\Sigma t = t'
\end{array}$$

and similarly for memberships and rules.

4.2 Casl

Casl, the Common Algebraic Specification Language [4, 31], has been designed by CoFI, the international *Common Framework Initiative for algebraic specification and development.* Its underlying logic combines first-order logic and induction (the latter is expressed using so-called sort generation constraints, which express term-generatedness of a part of a model; this is needed for the specification of the usual inductive datatypes) with subsorts and partial functions. The institution underlying Casl is introduced in two steps: first, many-sorted partial first-order logic with sort generation constraints and equality (*PCFOL*$^=$) is introduced, and then, subsorted partial first-order logic with sort generation constraints and equality (*SubPCFOL*$^=$) is described in terms of *PCFOL*$^=$ [25]. Basically this institution is composed of:

- A *subsorted signature* $\Sigma = (S, TF, PF, P, \le_S)$, where S is a set of sorts, TF and PF are two $S^* \times S$-sorted families $TF = (TF_{w,s})_{w \in S^*, s \in S}$ and $PF = (PF_{w,s})_{w \in S^*, s \in S}$ of total function symbols and partial function symbols, respectively, such that $TF_{w,s} \cap PF_{w,s} = \emptyset$, for each $(w, s) \in S^* \times S$, $P = (P_w)_{w \in S^*}$ a family of predicates, and $\le_S$ is a reflexive and transitive subsort relation on the set S. Given two signatures $\Sigma = (S, TF, PF, P)$ and $\Sigma' = (S', TF', PF', P')$, a signature morphism $\sigma : \Sigma \to \Sigma'$ consists of:
  - a map $\sigma^S : S \to S'$ preserving the subsort relation,
  - a map $\sigma^F_{w,s} : TF_{w,s} \cup PF_{w,s} \to TF_{\sigma^{S^*}(w), \sigma^S(s)} \cup PF'_{\sigma^{S^*}(w), \sigma^S(s)}$ preserving totality, for each $w \in S^*, s \in S$, and
  - a map $\sigma^P_w : P_w \to P'_{\sigma^{S^*}(w)}$ for each $w \in S^*$.

  Identities and composition are defined in the obvious way.

  With each subsorted signature $\Sigma = (S, TF, PF, P, \le_S)$ we associate a many-sorted signature $\hat{\Sigma}$, which is the extension of the underlying many-sorted signature (S, TF, PF, P) with:

  - a total *injection* function symbol $inj : s \to s'$, for each pair of sorts $s \leq_S s'$,
  - a partial *projection* function symbol $pr : s' \to ? s$, for each pair of sorts $s \leq_S s'$, and
  - a unary *membership* predicate symbol $\in^s : s'$, for each pair of sorts $s \leq_S s'$.

  Signature morphisms $\sigma : \Sigma \to \Sigma'$ are extended to signature morphisms $\hat{\sigma} : \hat{\Sigma} \to \hat{\Sigma}'$ by just mapping the injections, projections, and memberships in $\hat{\Sigma}$ to the corresponding injections, projections, and memberships in $\hat{\Sigma}'$.

  For a subsorted signature $\Sigma = (S, TF, PF, P, \leq_S)$, we define *overloading relations* (also called *monotonicity orderings*), $\sim_F$ and $\sim_P$, for function and predicate symbols, respectively:

  Let $f : w_1 \to s_1, f : w_2 \to s_2 \in TF \cup PF$, then $f : w_1 \to s_1 \sim_F f : w_2 \to s_2$ iff there exist $w \in S^*$ with $w \leq_{S^*} w_1$ and $w \leq_{S^*} w_2$ and $s \in S$ with $s_1 \leq_S s$ and $s_2 \leq_S s$.

  Let $p : w_1, p : w_2 \in P$, then $p : w_1 \sim_P p : w_2$ iff there exists $w \in S^*$ with $w \leq_{S^*} w_1$ and $w \leq_{S^*} w_2$.

- A set of *subsorted* Σ*-sentences*, that correspond to ordinary $\hat{\Sigma}$-many-sorted sentences, that is, closed many-sorted first-order $\hat{\Sigma}$-formulas or sort generation constraints over Σ. Sentence translation along a subsorted signature morphism σ is just sentence translation along the many-sorted signature morphism $\hat{\sigma}$.

- *Subsorted* Σ*-models* M are ordinary many-sorted $\hat{\Sigma}$-models, which are composed of:

  - a non-empty carrier set M_s for each sort $s \in S$,
  - a partial function f_M from M_w to M_s for each function symbol $f \in TF_{w,s} \cup PF_{w,s}$, $w \in S^*$, $s \in S$, the function being total if $f \in TF_{w,s}$, and
  - a predicate $p_M \subseteq M_w$ for each predicate symbol $p \in P_w$, $w \in S^*$,

  satisfying the following set of axioms $\hat{J}(\Sigma)$:

  - $inj_{(s,s)}(x) \overset{e}{=} x$ (identity), where $\overset{e}{=}$ stands for existential equation.
  - $inj_{(s,s')}(x) \overset{e}{=} inj_{(s,s')}(x) \implies x \overset{e}{=} y$ for $s \leq_S s'$ (embedding-injectivity),
  - $inj_{(s',s'')}(inj_{s,s'}(x)) \overset{e}{=} inj_{(s,s'')}(x)$ for $s \leq_S s' \leq_S s''$ (transitivity),
  - $pr_{(s',s)}(inj_{(s,s')}(x)) \overset{e}{=} x$ for $s \leq_S s'$ (projection),
  - $pr_{(s',s)}(x) \overset{e}{=} pr_{(s',s)}(y) \implies x \overset{e}{=} y$ for $s \leq_S s'$ (projection-injectivity),
  - $\in^s_{s'} (x) \iff pr_{(s',s)}(x)$ for $s \leq_S s'$ (membership),
  - $inj_{(s',s)}(f_{w',s'}(inj_{s_1,s'_1}(x_1), \ldots, inj_{s_n,s'_n}(x_n))) =$ $inj_{(s'',s)}(f_{w'',s''}(inj_{(s_1,s''_1)}(x_1), \ldots, inj_{(s_n,s''_n)}(x_n)))$ for $f_{w',s'} \sim_F f_{w'',s''}$, where $w \leq w', w''$, $s', s'' \leq s$, $w = s_1, \ldots, s_n$, $w' = s'_1, \ldots, s'_n$, and $w'' = s''_1, \ldots, s''_n$ (function-monotonicity), and
  - $p_{w'}(inj_{(s_1,s'_1)}(x_1), \ldots, inj_{(s_n,s'_n)}(x_n)) \iff p_{w''}(inj_{(s_1,s''_1)}(x_1), \ldots, inj_{(s_n,s''_n)}(x_n))$ for $p_{w'} \sim_P p_{w''}$, where $w \leq w', w''$, $w = s_1 \ldots s_n$, $w' = s'_1 \ldots s'_n$, and $w'' = s''_1 \ldots s''_n$ (predicate-monotonicity).

- *Satisfaction* and the *satisfaction condition* are inherited from the many-sorted institution. Roughly speaking, a formula φ is satisfied in a model M iff it is satisfied w.r.t. all variable valuations into M.

More details on this institution can be found in [25].

In contrast to Maude, Casl's subsort relations may be interpreted by arbitrary injections $inj_{s,t}$, not only by subsets. We refer to [31] for details. We will only need the Horn clause fragment of first-order logic. For freeness, we will also need sort generation constraints, as well as the *second-order* extension of Casl with quantification over predicates; we show the details in Section 5.2.

4.3 Encoding Maude into Casl

We now present an encoding of Maude into Casl, which is formalized as an institution comorphism. The idea of the encoding of Maude^{pre} in Casl is that we represent rewriting as a binary predicate and we axiomatize it as a preorder compatible with operations.

Every Maude signature $\Sigma = (K, F, \mathit{kind} : (S, \leq) \rightarrow K)$ is translated to the Casl theory $\Phi(\Sigma) = ((S', \leq', F, P), E)$, where S' is the disjoint union of K and S, $\leq'$ extends the relation $\leq$ on sorts with pairs $(s, \mathit{kind}(s))$, for each $s \in S$, $\mathit{rew} \in P_{s,s}$ for any $s \in S'$ is a binary predicate and E contains axioms stating that for any kind k, $\mathit{rew} \in P_{k,k}$ is a preorder compatible with the operations. The latter means that for any $f \in F_{s_1 \ldots s_n, s}$ and any x_i, y_i of sort $s_i \in S'$, $i = 1, \ldots, n$, if $\mathit{rew}(x_i, y_i)$ holds, then $\mathit{rew}(f(x_1, \ldots, x_n), f(y_1, \ldots, y_n))$ also holds.

Let $\Sigma_i,\ i = 1, 2$ be two Maude signatures and let $\varphi : \Sigma_1 \rightarrow \Sigma_2$ be a Maude signature morphism. Then its translation $\Phi(\varphi) : \Phi(\Sigma_1) \rightarrow \Phi(\Sigma_2)$ denoted ϕ, is defined as follows:

- for each $s \in S$, $\phi(s) = \varphi^{sort}(s)$ and for each $k \in K$, $\phi(k) = \varphi^{kind}(k)$.
- the subsort preservation condition of ϕ follows from the similar condition for φ.
- for each operation symbol σ, $\phi(\sigma) = \varphi^{op}(\sigma)$.
- *rew* is mapped identically.

The sentence translation map for each signature is obtained in two steps. While the equational atoms are translated as themselves, membership atoms $t : s$ are translated to Casl memberships $t\ \mathit{in}\ s$ and rewrite atoms of the form $t \Rightarrow t'$ are translated as $\mathit{rew}(t, t')$. Then, any sentence of Maude of the form $(\forall x_i : k_i) H \implies C$, where H is a conjunction of Maude atoms and C is an atom is translated as $(\forall x_i : k_i) H' \implies C'$, where H' and C' are obtained by mapping all the Maude atoms as described before.

Given a Maude signature Σ, a model M' of its translated theory (Σ', E) is reduced to a Σ-model denoted M where:

- for each kind k, define $M_k = M'_k$ and the preorder relation on M_k is *rew*;
- for each sort s, define M_s to be the image of M'_s under the injection $\mathit{inj}_{s,\mathit{kind}(s)}$ generated by the subsort relation;
- for each f on kinds, let $M_f(x_1, \ldots, x_n) = M'_f(x_1, \ldots, x_n)$ and for each f on sorts of result sort s, let $M_f(x_1, \ldots, x_n) = \mathit{inj}_{s,\mathit{kind}(s)}(M'_f(x_1, \ldots, x_n))$. M_f is monotone because axioms ensure that M'_f is compatible with *rew*.

The reduct of model homomorphisms is the expected one.

Let Σ be a Maude signature, M', N' be two $\Phi(\Sigma)$-models (in Casl) and let $h' : M' \rightarrow N'$ be a model homomorphism. Let us denote $M = \beta_\Sigma(M')$, $N = \beta_\Sigma(N')$ and let us define $h : M \rightarrow N$ as follows: for any kind k of Σ, $h_k = h'_k$ (this is correct because the domain and the codomain match, by definition of M and N). We need to show that h is indeed a Maude model homomorphism. For this, we need to show three things:

1. h_k is preorder preserving for any kind k.

   Assume $x \leq_k^M y$. By definition, the preorder on M_k is the one given by *rew*, so this means $M_{rew}(x, y)$ holds. By the homomorphism condition for h' we have $N_{rew}(h'(x), h'(y))$ holds, which means by definition of the preorder on N that $h'(x) \leq_k^N h'(y)$.

2. h is an algebra homomorphism.

   This follows directly from the definition of M_f where f is an operation symbol and from the homomorphism condition for operation symbols for h'.

3. for any sort s, $h_{\mathit{kind}(s)}(M_s) \subseteq N_s$.

   By definition, $M_s = \mathit{inj}_{s,\mathit{kind}(s)}(M'_s)$. By the homomorphism condition for $\mathit{inj}_{s,\mathit{kind}(s)}$, which is an explicit operation symbol in Casl, we have that

   $$h_{\mathit{kind}(s)}(M_s) = h_{\mathit{kind}(s)}(\mathit{inj}_{s,\mathit{kind}(s)}(M'_s)) = \mathit{inj}_{s,\mathit{kind}(s)}(h_s(M'_s)).$$

   Since $h_s(M'_s) \subseteq N'_s$ by definition, we have that $\mathit{inj}_{s,\mathit{kind}(s)}(h_s(M'_s)) \subseteq \mathit{inj}_{s,\mathit{kind}(s)}(N'_s)$, which by definition is N_s.

5 Building development graphs

We describe in this section how Maude structuring mechanisms described in Section 2 are translated into development graphs. Then, we explain how these development graphs are normalized to deal with freeness constraints.

5.1 Creating the development graph

We describe here how Maude modules, theories, and views are translated into development graphs, illustrating it with an example.

5.1.1 Modules

Each Maude module generates two nodes in the development graph. The first one contains the theory equipped with the usual loose semantics. The second one, linked to the first one with a free definition link (whose signature morphism is detailed in Section 5.2), contains the same signature but no local axioms and stands for the free models of the theory. Note that Maude theories only generate one node, since their initial semantics is not used by Maude specifications.

The model class of parameterized modules consists of free extensions of the models of their parameters, that are persistent on sorts, but not on kinds. This notion of freeness has been studied in [5] under assumptions like existence of top sorts for kinds and sorted variables in formulas; our results hold under similar hypotheses. We use non-persistent free links to link these modules with their corresponding theories.

5.1.2 Module expressions

Maude module expressions allow to combine and modify the information contained in Maude modules:

- When the module expression is a simple identifier the development graph remains unchanged.
- The summation of the module expressions ME_1 and ME_2 generates a new node in the development graph ($ME_1 + ME_2$) with the union of the information in both summands. A definition link is also created between the original expressions and the resulting one.
- The renaming expression $ME * (R)$ creates a morphism with the information given in R that will be used to label the link between the node standing for the module expression and the node importing it.

5.1.3 Importations

As explained above, each Maude module generates two nodes in the development graph; when importing a module, we will select between these nodes depending on the chosen importation mode:

- The `protecting` mode generates a non-persistent free link between the current node and the node standing for the free semantics of the included one. We use the same links for the parameters in parameterized modules.
- The `extending` mode generates a global link with the annotation `PCons?`, that stands for proof-theoretic conservativity and that can be checked with a special conservativity checker that is integrated into Hets.
- The `including` mode generates a global definition link between the current node and the node standing for the loose semantics of the included one.

5.1.4 Views

Maude views have a theory as source and either a module or a theory as target. All the sorts and the operators declared in the source theory have to be mapped to sorts and operators in the target.

As seen in Section 2.5.3, a particular case of mapping between operators is the mapping between terms, that has the general form `op` e `to term` t. Since this shortcut allows to map operators with different profiles, in these cases it generates an auxiliary node with the signature of the target specification extended by an extra operator of the appropriate arity; this node will be used as new target.

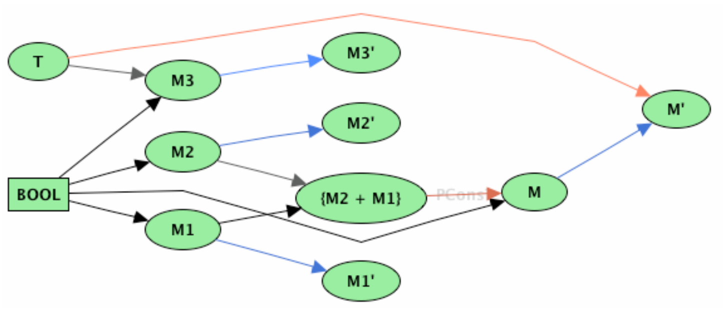


Figure 3: Development graph for Maude specifications

Views generate a theorem link between the theory and the module satisfying it. Note that an instantiation generates some implicit morphisms and modifies the ones stated in the views, see Section 2.5 for details:

- Sorts and labels are qualified by the parameter name in order to distinguish different labels/sorts with the same name defined in different theories. Thus, the mapping indicated by the view (more specifically, the source sorts) is modified depending on the name of the parameter.
- As explained in Section 2.5.4, parameterized modules can define parameterized sorts, that is, sorts that use the parameters as part of the sort name and hence they are modified by the mapping in the view. Moreover, when the target of a view is a theory the identifiers of these sorts are extended with the name of the view and the name of the new parameter. Thus, the sort morphism is extended with these new renamings.

5.1.5 An example of development graph

We illustrate how to build the development graph with an example. Consider the following Maude specification:

```
fmod M1 is
 sort S1 .
 op _+_ : S1 S1 -> S1 [comm] .
endfm
```

```
fmod M2 is
  sort S2 .
endfm
```

```
th T is
 sort S1 .
 op _._ : S1 S1 -> S1 .
 eq V1:S1 . V2:S1 = V2:S1 . V1:S1 [nonexec] .
endth
```

```
mod M3{X :: T} is
 sort S4 .
endm
```

```
mod M is
 ex M1 + M2 * (sort S2 to S) .
endm
```

```
view V from T to M is
 op _._ to _+_ .
endv
```

HETS builds the graph shown in Figure 3, where the following steps take place:

- First, the modules in the predefined Maude prelude generate their own graph. These nodes can be used by the ones of the current specification, like the `BOOL` node in the image (where the node has rectangular form because it hides part of its structure, like the modules it imports).
- Each module has generated a node with its name and another primed one that contains the initial model, while both of them are linked with a non-persistent free link (in blue in the illustration). Note that theory `T` did not generate this primed node.
- The summation expression has created a new node that includes the theories of `M1` and `M2`, importing the latter with a renaming; this new node, since it is imported in `extending` mode, uses a link with the PCons? annotation. This is the (unfortunately blurry) label in the link from `{M2 + M1}` to `M`.

- There is a theorem link (red link in the figure) between `T` and the free (here, initial) model of `M`. This link is labeled with the mapping defined in the view `V`, namely `op _._ to _+_ .`.
- The parameterized module `M3` includes the theory of its parameter with a renaming, that qualifies the sort. Note that these nodes are connected by means of a non-persistent free link.

It is straightforward to show:

Theorem 1 *The translation of Maude modules into development graphs is semantics-preserving.*

Once the development graph is built, we can apply the (logic independent) calculus rules that reduce global theorem links to local theorem links, which are in turn discharged by local theorem proving [27]. This can be used to prove Maude views, like "natural numbers are a total order." For example, we can automatically prove the view `V` above correct by using the first-order automated provers SPASS or Vampire.

We show in the next section how we deal with the freeness constraints imposed by free definition links.

5.2 Normalization of free definition links

Maude uses initial and free semantics intensively. The semantics of freeness is, as mentioned, different from the one used in CASL in that the free extensions of models are required to be persistent only on sorts and new error elements can be added on the interpretation of kinds. As explained before, attempts to design the translation to CASL in such a way that Maude free links would be translated to usual free definition links in CASL have been unsuccessful, and thus we decided to use non-persistent free links. Hence, in order not to break the development graph calculus, we need a way to normalize them, by replacing them with a semantically equivalent development graph in CASL. The main idea is to make a free extension persistent by duplicating parameter sorts appropriately, such that the parameter is always explicitly included in the free extension.

For any Maude signature Σ, let us define an extension $\Sigma^{\#} = (S^{\#}, \leq^{\#}, F^{\#}, P^{\#})$ of the translation $\Phi(\Sigma)$ of Σ to CASL as follows:

- $S^{\#}$ adds the sorts of $\Phi(\Sigma)$ to the set $\{[s] \mid s \in \mathit{Sorts}(\Sigma)\}$;
- $\leq^{\#}$ extends the subsort relation $\leq$ with pairs $(s, [s])$ for each sort s and $([s], [s'])$ for any sorts $s \leq s'$;
- $F^{\#}$ adds the function symbols $\{f : [w] \to [s]\}$ for all function symbols on sorts $f : w \to s$;[4] and
- $P^{\#}$ adds the predicate symbol *rew* on all new sorts.

Now, we consider a Maude non-persistent free definition link and let $\sigma : \Sigma \to \Sigma'$ be the morphism labeling it.[5] We define a CASL signature morphism $\sigma^{\#} : \Phi(\Sigma) \to \Sigma'^{\#}$: on sorts, $\sigma^{\#}(s) = \sigma^{sort}(s)$ and $\sigma^{\#}([s]) = [\sigma^{sort}(s)]$; on operation symbols, we can define $\sigma^{\#}(f) = \sigma^{op}(f)$ and this is correct because the operation symbols were introduced in $\Sigma'^{\#}$; *rew* is mapped identically.

The normalization of Maude freeness is then illustrated in Figure 4. Given a non-persistent free definition link $M \xRightarrow[n.p.free]{\sigma} N$, with $\sigma : \Sigma \to \Sigma_N$, we first take the translation of the nodes to CASL (nodes M' and N'), and create the node M'', an extension (the morphism ι is a renaming to make the signature distinct from M) of M' where the signature has been extended with sorts $[s]$ for each sort $s \in \Sigma_M$, such that $s \leq [s]$ and $[s] \leq [s']$ if $s \leq s'$; function symbols have been extended with $f : [w] \to [s]$ for each $f : w \to s \in \Sigma_M$; and new *rew* predicates have been added for these sorts. Then, we introduce a new node, K, labeled with $\Sigma_N^{\#}$, a free definition link from M'' to K labeled with $\sigma^{\#}$ and a hiding definition link from K to N' labeled with the inclusion ι_N.[6]

Notice that the models of N are Maude reducts of CASL models of K, reduced along the inclusion ι_N.

Now we show how to eliminate CASL free definition links in a logic-independent way. The idea is to use a transformation specific to the second-order extension of CASL to normalize freeness. The intuition behind this construction is that it mimics the quotient term algebra construction, that is, the free model is specified as the homomorphic image of an absolutely free model (i.e. term model).

We are going to make use of the following known facts [35]:

[4] $[s_1 \ldots s_n]$ is defined to be $[s_1] \ldots [s_n]$.

[5] In Maude, this would usually be an injective renaming.

[6] The arrows without labels in Figure 4 correspond to heterogeneous links from Maude to CASL.

$$\begin{tikzcd}
M \arrow[rr, Rightarrow, "\sigma", "n.p.free"'] \arrow[d, Rightarrow] & & N \arrow[dd, Rightarrow] \\
M' \arrow[d, Rightarrow, "\iota_N"] & & \\
M'' \arrow[r, Rightarrow, "\sigma^{\#}", "free"'] & K \arrow[r, Rightarrow, "\iota_N", "hide"'] & N'
\end{tikzcd}$$

Figure 4: Normalization of Maude free links

Fact 1 *Extensions of theories in Horn form admit free extensions of models.*

Fact 2 *Extensions of theories in Horn form are monomorphic.*[7]

Given a free definition link $M \xRightarrow[free]{\sigma} N$, with $\sigma : \Sigma \to \Sigma^N$ such that $Th(M)$ is in Horn form, replace it with $M \xRightarrow{incl} K \xRightarrow[hide]{incl} N'$, where N' has the same signature and axioms as N, *incl* denote inclusions and the node K is constructed as follows.

The signature Σ^K consists of the signature Σ^M disjointly united with a copy of Σ^M, denoted $\iota(\Sigma_M)$ which makes all function symbols total (let us denote $\iota(f)$ the corresponding symbol in this copy for each symbol f from the signature Σ^M) and augmented with new operations $h : \iota(s) \to ?\, s$, for any sort s of Σ^M and $make_s : s \to \iota(s)$, for any sort s of the source signature Σ of the morphism σ labelling the free definition link.

The axioms ψ^K of the node K consist of:

- sentences imposing the bijectivity of *make*;
- axiomatization of the sorts in $\iota(\Sigma_M)$ as free types with all operations as constructors, including *make* for the sorts in $\iota(\Sigma)$;
- homomorphism conditions for h:

  $$h(\iota(f)(x_1, \ldots, x_n)) = f(h(x_1), \ldots, h(x_n))$$

  and

  $$\iota(p)(t_1, \ldots, t_n) \Rightarrow p(h(t_1), \ldots, h(t_n))$$

- surjectivity of homomorphisms:

  $$\forall y : s.\exists x : \iota(s).h(x) \overset{e}{=} y$$

- a second-order formula saying that the kernel of h ($ker(h)$) is the least partial predicative congruence[8] satisfying $Th(M)$. This is done by quantifying over a predicate symbol for each sort for the binary relation and one predicate symbol for each relation symbol as follows:

  $$\begin{array}{l}\forall \{P_s : \iota(s), \iota(s)\}_{s \in Sorts(\Sigma_M)}, \{P_{p:w} : \iota(w)\}_{p:w \in \Sigma_M} \\ .\ symmetry \wedge transitivity \wedge congruence \wedge satThM \implies largerThanKerH\end{array}$$

  where *symmetry* stands for

  $$\bigwedge_{s \in Sorts(\Sigma^M)} \forall x : \iota(s), y : \iota(s).P_s(x, y) \implies P_s(y, x),$$

[7]That is, if N is an extension of K under a morphism σ, then every K-model has a σ-expansion to an N-model that is unique up to isomorphism.

[8]A *partial predicative congruence* consists of a symmetric and transitive binary relation for each sort and a relation of appropriate type for each predicate symbol.

transitivity stands for

$$\bigwedge_{s\in Sorts(\Sigma^M)} \forall x:\iota(s), y:\iota(s), z:\iota(s).P_s(x,y)\wedge P_s(y,z) \implies P_s(x,z),$$

congruence is the conjunction of

$$\begin{array}{l}\bigwedge_{f_{w\to s}\in\Sigma^M} \forall x_1\ldots x_n:\iota(w), y_1\ldots y_n:\iota(w)\,.\\ D(\iota(f_{w,s})(\bar{x}))\wedge D(\iota(f_{w,s})(\bar{y}))\wedge P_w(\bar{x},\bar{y}) \implies P_s(\iota(f_{w,s})(\bar{x}),\iota(f_{w,s})(\bar{y}))\end{array}$$

and

$$\begin{array}{l}\bigwedge_{p_w\in\Sigma^M} \forall x_1\ldots x_n:\iota(w), y_1\ldots y_n:\iota(w)\,.\\ D(\iota(f_{w,s})(\bar{x}))\wedge D(\iota(f_{w,s})(\bar{y}))\wedge P_w(\bar{x},\bar{y}) \implies P_{p:w}(\bar{x})\Leftrightarrow P_{p:w}(\bar{y})\end{array}$$

where D indicates definedness. *satThM* stands for

$$Th(M)[\overset{e}{=}/P_s; p:w/P_{p:w}; D(t)/P_s(t,t); t=u/P_s(t,u)\vee(\neg P_s(t,t)\wedge\neg P_s(u,u))]$$

where, for a set of formulas Ψ, $\Psi[sy_1/sy'_1;\ldots;sy_n/sy'_n]$ denotes the simultaneous substitution of sy'_i for sy_i in all formulas of Ψ (while possibly instantiating the meta-variables t and u). Finally *largerThanKerH* stands for

$$\begin{array}{l}\bigwedge_{s\in Sorts(\Sigma^M)} \forall x:\iota(s), y:\iota(s).h(x)\overset{e}{=}h(y) \implies P_s(x,y)\\ \bigwedge\wedge_{p_w\in\Sigma^M}\forall\bar{x}:\iota(w).\iota(p:w)(\bar{x}) \implies P_{p:w}(\bar{x})\end{array}$$

Proposition 1 *The models of the nodes N and N' are the same.*

Proof. Let n be an N-model. To prove that n is also an N'-model, we need to show that it has a K-expansion.

Let us define the following Σ_K model, denoted k:

- on Σ_M, k coincides with n;
- on $\iota(\Sigma_M)$, the interpretation of sorts and function symbols is given by the free types axioms (i.e., sorts are interpreted as set of terms, operations $\iota(f)$ map terms $t_1,\ldots,t_n$ to the term $\iota(f)(t_1,\ldots,t_n)$). We define the interpretation of predicates after defining h;
- *make* assigns to each x the term $make(x)$;
- the homomorphism h is defined inductively as follows:
  - $h(make(x)) = x$, if $x\in n_s$ and $s\in Sorts(\Sigma)$;
  - $h(make(t)) = h(t)$, otherwise;
  - $h(\iota(f)(t_1,\ldots,t_n))$ is defined iff $f(h(t_1),\ldots,h(t_n))$ is defined in n and then $h(\iota(f)(t_1,\ldots,t_n)) = f(h(t_1),\ldots,h(t_n))$;
- for predicates in $\iota(\Sigma_M)$ we define $\iota(p)(t_1,\ldots,t_n)$ iff $p(h(t_1),\ldots,h(t_n))$.

Notice that the first three types of axioms of the node K hold by construction and also notice that $ker(h)$ satisfies $Th(M)$ because n is an M-model. The surjectivity of h and the minimality of $ker(h)$ are exactly the "no junk" and the "no confusion" properties of the free model n.

For the other inclusion, let n' be a model of N', n_0 be its Σ-reduct and k' a K-expansion of n'. Using the fact that the theory of M is in Horn form, we get an expansion of n_0 to a σ-free model n. We have seen that all free models are also models of N' and moreover we have seen that $ker(k_h)$ is the least predicative congruence satisfying $Th(M)$. The free types axioms of K fix the interpretation of $\iota(\Sigma_M)$ and therefore $ker(k'_h)$ and $ker(k_h)$ are both minimal on the same set, and must be the same. This and the surjectivity of k_h and k'_h allow us to define easily an isomorphism between n and n' and because n' is isomorphic with a free model it must be free as well.

□

6 An example: Reversing lists

The example we are going to present is a standard specification of lists with empty lists, concatenation, and reversal. We want to prove that by reversing a list twice we obtain the original list. Since Maude syntax does not support marking sentences of a theory as theorems, the methodology to state proof obligations in Maude would normally be to write a view (*PROVEIDEM* in Figure 5, left side) from a theory containing the theorem (*REVIDEM*) to the module with the axioms defining *reverse* (*LISTREV*).

```
fmod MYLIST is
 sorts Elt List .
 subsort Elt < List .
 op nil : -> List [ctor] .
 op __ : List List -> List
    [ctor assoc id: nil] .
endfm
fmod MYLISTREV is
 pr MYLIST .
 op reverse : List -> List .
 var L : List .
 var E : Elt .
 eq reverse(nil) = nil .
 eq reverse(E L) = reverse(L) E .
endfm
fth REVIDEM  is
 pr MYLIST .
 op reverse : List -> List .
 var L : List .
 eq reverse(reverse(L)) = L .
endfth
view PROVEIDEM from REVIDEM
               to MYLISTREV is
 sort List to List .
 op reverse to reverse .
endv
```

logic MAUDE
spec PROVEIDEM =
  free
  {*sorts Elt List .*
   subsort Elt < List .
   op nil : −> List [ctor] .
   op __ : List List −> List [ctor assoc id: nil] .
  }
then {*op reverse : List −> List .*
   var L : List . *var E : Elt .*
   eq reverse(nil) = nil .
   eq reverse(E L) = reverse(L) E .
  } **then %implies**
  {*var L : List .*
   eq reverse(reverse(L)) = L .
  }

Figure 5: Lists with reverse, in Maude (left) and CASL (right) syntax.

The first advantage that the integration of Maude in HETS brings in is that we can use heterogeneous CASL structuring mechanisms and the *%implies* annotation to obtain the same development graph in a shorter way—see the right side of Figure 5, whose development graph is shown in Figure 6, where the blue link[9] stands for freeness and the red one[10] for proof obligations, and only the rightmost node has name, while all the others are intermediate nodes introduced by the HETS constructors (`free`, used to indicate that the specification is free, and `then`, used to import the previous specification without assuming anything about it) used to structure the specification.

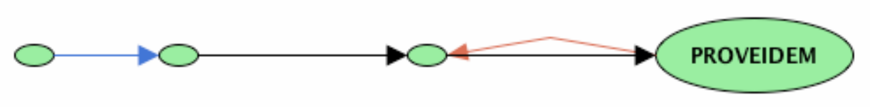


Figure 6: Development graph for the lists example

For our example, the development calculus rules are applied as follows.[11] First, the whole graph is translated to CASL; during this step, Maude non-persistent free links are normalized. The next step is to normalize CASL free links, using the `Freeness` rule. We then apply the `Normal-Form` rule which introduces normal forms for the nodes with incoming hiding links (introduced at the previous step) and then the `Theorem-Hide-Shift` rule which moves the target of any theorem link targeting a node with incoming hiding links to the normal form of the latter. Finally, calling `Automatic` the development graph

[9]The leftmost link, if you are reading a black-and-white paper.
[10]The one starting in `PROVEIDEM`.
[11]All the rules listed below are accessible in the Edit/Proof menu.

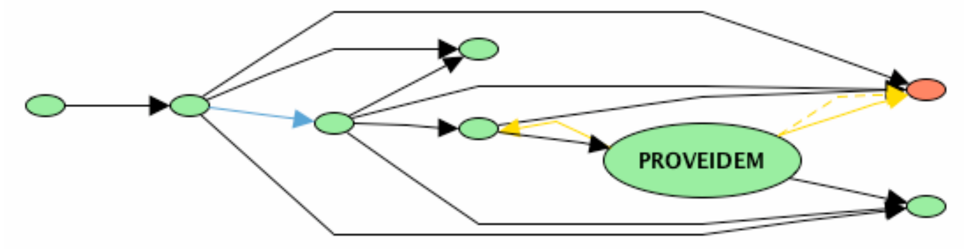


Figure 7: Development graph for the lists example after the transformation rules

in Figure 7 is obtained, where the proof obligation has been delegated to the normal form node (the red node in the upper right corner).

In this node, we now have a proof goal for a second-order theory. It can be discharged using the interactive theorem prover Isabelle/HOL [32]. We have set up a series of lemmas easing such proofs, and that can be adapted to other proofs over Maude specifications. First of all, normalization of freeness introduces sorts for the free model which are axiomatized to be the homomorphic image of a set of the absolutely free (i.e. term) model. A transfer lemma (that exploits surjectivity of the homomorphism) enables us to transfer any proof goal from the free model to the absolutely free model. Since the absolutely free model is term generated, we can use induction proofs here. For the case of datatypes with total constructors (like lists), we prove by induction that the homomorphism is total as well. Once these lemmas have been proved we prove the main theorem; to do that, two further lemmas on lists are proved by induction: (1) associativity of concatenation and (2) the reverse of a concatenation is the concatenation (in reverse order) of the reversed lists. This infrastructure then allows us to prove (again by induction) that $reverse(reverse(L)) = L$.

While proof goals in Horn clause form often can be proved by induction, other proof goals like the inequality of certain terms or extensionality of sets cannot. Here, we need to prove inequalities or equalities with more complex premises, and this calls for use of the special axiomatization of the kernel of the homomorphism. This axiomatization is rather complex, and we are currently setting up the infrastructure for easing such proofs in Isabelle/HOL.

6.1 An easier proof: Revisiting our initial example

Recalling the example in Section 5.1.5, we stated in a theory `T` that an operator `_._` over the sort `S1` fulfilling the equation

```
  eq V1:S1 . V2:S2 = V2:S1 . V1:S2 [nonexec] .
```

was required. We mapped this operator by means of a view to another one with syntax `_+_` and declared with the commutativity attribute `comm`, and thus we want to check that this view is correct. This easy proof can be automatically discarded by using provers such as SPASS just by transforming the development graph with the `Automatic` command, that "pushes" proof obligations to the appropriate nodes. We show in Figure 8 how it was proved (the `+` symbol indicates it was proved) by just using the `Run` button. This straightforward strategy can be applied to most Maude views (such as the ones from theories requiring different orders over the elements), making these proofs straightforward.

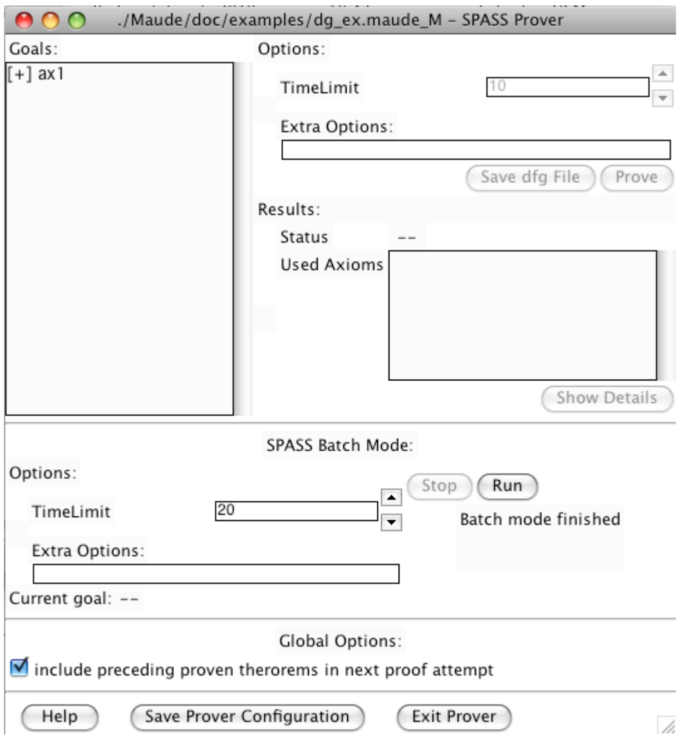
Of course, this proof can also be done in Isabelle. The code needed to prove it is:

```
theorem Ax1 :
"ALL (v1 :: kind_S1). ALL (v2 :: kind_S1). v1 +'' v2 = v2 +'' v1"
apply auto
apply (rule comm_Plus'_kind_S1)
done
```

where the theorem was introduced by HETS and `comm_Plus'_kind_S1` is the axiom automatically generated by the system for the commutative operator `_+_` in `M1`.

7 Implementation

We describe in this section how the integration described in the previous sections has been implemented. We have used Maude to parse Maude modules, taking advantage of the reflective capabilities of rewriting



[+] ax1

TimeLimit 10

Status --

Used Axioms

Options:

TimeLimit 20

Batch mode finished

Extra Options:

Current goal: --

Global Options:

[✓] include preceding proven therorems in next proof attempt

Figure 8: Axiom proved by SPASS

logic [8], while the rest of the system has been implemented in Haskell, the implementation language of Hets. Section 7.1 shows the abstract syntax used to represent Maude modules in Haskell, while Section 7.2 presents how these data structures are generated in Maude. Section 7.3 shows how Maude signatures, sentences, and morphisms are obtained, Section 7.4 explains how they are introduced into a development graph, and Section 7.5 describes the implementation of the comorphism between Maude and Casl. Finally, Section 7.6 outlines how the freeness constraints are implemented.

7.1 Abstract syntax

In this section we show how the abstract syntax for Maude specifications is defined in Haskell. This abstract syntax is based in the Maude grammar presented in [7, Chapter 24].

The main datatype of this abstract syntax is `Spec`, that distinguishes between the different specifications available in Maude: modules, theories, and views. Although both modules and theories contain the same information, their semantics are different and need different constructors:

```
data Spec = SpecMod Module
          | SpecTh Module
          | SpecView View
          deriving (Show, Read, Ord, Eq)
```

A `Module` is composed of the identifier of the module, a list of parameters, and a list of statements:

```
data Module = Module ModId [Parameter] [Statement]
            deriving (Show, Read, Ord, Eq)
```

while a `View` is composed of a module identifier, the source and target module expressions, and a list of renamings:

```
data View = View ModId ModExp ModExp [Renaming]
            deriving (Show, Read, Ord, Eq)
```

The `Parameter` type contains the identifier of the parameter, a sort (used as the parameter identifier), and its type (which is a module expression):

```
data Parameter = Parameter Sort ModExp
               deriving (Show, Read, Ord, Eq)
```

A `Statement` can be any of the Maude statements: importation, sort, subsort, and operator declarations, and equation, membership axiom, and rule statements:

```
data Statement = ImportStmnt Import
               | SortStmnt Sort
               | SubsortStmnt SubsortDecl
               | OpStmnt Operator
               | EqStmnt Equation
               | MbStmnt Membership
               | RlStmnt Rule
               deriving (Show, Read, Ord, Eq)
```

Importations consist of a module expression qualified by the type of import:

```
data Import = Including ModExp
            | Extending ModExp
            | Protecting ModExp
            deriving (Show, Read, Ord, Eq)
```

A subsort declaration keeps single relations between sorts, being the first one the subsort and the second one the supersort:

```
data SubsortDecl = Subsort Sort Sort
                 deriving (Show, Read, Ord, Eq)
```

Operator declarations are composed of the identifier of the operator, a list of types giving the arity of the operator, a type for its coarity, and a list of attributes:

```
data Operator = Op OpId [Type] Type [Attr]
              deriving (Show, Read, Ord, Eq)
```

Membership statements consist of a term, its sort, a list of conditions, and a list of statement attributes:

```
data Membership = Mb Term Sort [Condition] [StmntAttr]
                deriving (Show, Read, Ord, Eq)
```

Equations and rules share the same elements: the lefthand and righthand terms of the statement, a list of conditions, and a list of statement attributes:

```
data Equation = Eq Term Term [Condition] [StmntAttr]
              deriving (Show, Read, Ord, Eq)

data Rule = Rl Term Term [Condition] [StmntAttr]
          deriving (Show, Read, Ord, Eq)
```

We distinguish between the following module expressions:

- A single identifier:

  ```
  data ModExp = ModExp ModId
  ```

- A summation, that keeps the two module expressions involved:

  ```
              | SummationModExp ModExp ModExp
  ```

- A renaming, that contains the module expression renamed and the list of renamings:

  ```
              | RenamingModExp ModExp [Renaming]
  ```

- An instantiation, composed of the module instantiated and the list of view identifiers applied:

```
               | InstantiationModExp ModExp [ViewId]
               deriving (Show, Read, Ord, Eq)
```

The `Renaming` type distinguishes the different renamings available in Maude:

- Renaming of sorts, that indicates that the first sort identifier is changed to the second one:

```
data Renaming = SortRenaming Sort Sort
```

- Renaming of labels, where the first label is renamed to the second one:

```
                | LabelRenaming LabelId LabelId
```

- Renaming of operators, that can be of three kinds: renaming of operators without profile, with profile, or a map between terms, as explained in Section 2.5.3:

```
                | OpRenaming1 OpId ToPartRenaming
                | OpRenaming2 OpId [Type] Type ToPartRenaming
                | TermMap Term Term
                deriving (Show, Read, Ord, Eq)
```

  where `ToPartRenaming` specifies the new operator identifier and the new attributes:

```
data ToPartRenaming = To OpId [Attr]
                      deriving (Show, Read, Ord, Eq)
```

The `Condition` type distinguishes between the different conditions available in Maude, namely equational conditions, membership conditions, matching conditions, and rewriting conditions:

```
data Condition = EqCond Term Term
               | MbCond Term Sort
               | MatchCond Term Term
               | RwCond Term Term
               deriving (Show, Read, Ord, Eq)
```

We define the type `Qid`, a synonym of `Token` that will be used for identifiers:

```
type Qid = Token
```

Terms are always represented in prefix notation. Notice that the case of an operator applied to a list of terms is slightly different to the Maude grammar because it also includes the type of the term. It will be used later in the implementation to rename operators whose profile has been specified:

```
data Term = Const Qid Type
          | Var Qid Type
          | Apply Qid [Term] Type
          deriving (Show, Read, Ord, Eq)
```

Finally, `Type` distinguishes between sorts and kinds:

```
data Type = TypeSort Sort
          | TypeKind Kind
          deriving (Show, Read, Ord, Eq)
```

7.2 Maude parsing

In this section we explain how the Maude specifications introduced in HETS are parsed in order to obtain a term following the abstract syntax described in the previous section. We are able to implement this parsing in Maude itself thanks to Maude's metalevel [7, Chapter 14], a module that allows the programmer to use Maude entities such as modules, equations, or rules as usual data by efficiently implementing the *reflective* capabilities of rewriting logic [8].

The function `haskellify` receives a module (the first parameter stands for the original module, while the second one contains the flattened one) and returns a list of quoted identifiers creating an object of type `Spec`, that can be read by Haskell since this data type derives the class `Read`:

```
op haskellify : Module Module -> QidList .
ceq haskellify(M, M') =
    'SpecMod '`( 'Module haskellifyHeader(H) ' '
    '`[ haskellifyImports(IL) comma(IL, SS)
        haskellifySorts(SS) comma(IL, SS, SSDS)
        haskellifySubsorts(SSDS) comma(IL, SS, SSDS, ODS)
        haskellifyOpDeclSet(M', ODS) comma(IL, SS, SSDS, ODS, MAS)
        haskellifyMembAxSet(M', MAS) comma(IL, SS, SSDS, ODS, MAS, EqS)
        haskellifyEqSet(M', EqS) '`] '`) '\n '@#$endHetsSpec$#@ '\n
 if fmod H is IL sorts SS . SSDS ODS MAS EqS endfm := M .
```

This function prints the keyword `SpecMod` and uses the `haskellify` auxiliary functions to print the different parts of the module. The functions `comma` introduce a comma whenever it is necessary. Since all the "haskellify" functions are very similar, we describe them by using `haskellifyImports` as example. This function traverses all the imports in the list and applies the auxiliary function `haskellifyImport` to each of them:

```
op haskellifyImports : ImportList -> QidList .
eq haskellifyImports(nil) = nil .
eq haskellifyImports(I IL) = 'ImportStmnt ' '`( haskellifyImport(I) '`)
                             comma(IL) haskellifyImports(IL) .
```

This auxiliary function distinguishes between the importation modes, using the appropriate keyword for each of them:

```
op haskellifyImport : Import -> QidList .
eq haskellifyImport(protecting ME .) = 'Protecting haskellifyME(ME) .
eq haskellifyImport(including ME .) = 'Including haskellifyME(ME) .
eq haskellifyImport(extending ME .) = 'Extending haskellifyME(ME) .
```

where `haskellifyME` is in charge of printing the module expression. When it is just an identifier, it prints it preceded by the word `ModId`:

```
op haskellifyME : ModuleExpression -> QidList .
eq haskellifyME(Q) = ' '`( 'ModExp ' '`( 'ModId qid2token(Q) '`) '`) ' .
```

The summation module expression recursively prints the summands, and uses the keyword `SummationModExp` to indicate the type of module expression:

```
eq haskellifyME(ME + ME') = ' '`( 'SummationModExp haskellifyME(ME)
                            haskellifyME(ME')  '`) ' .
```

To print a renaming we recursively apply `haskellifyME` for the inner module expression and then we use the auxiliary function `haskellifyMaps` to print the renamings. In this case we use the constant `no-module` as argument because it will only be used when parsing mappings from views, since it may be needed to parse terms:

```
eq haskellifyME(ME * (RNMS)) = ' '`( 'RenamingModExp haskellifyME(ME)
                               '`[ haskellifyMaps(no-module, no-module, RNMS) '`]  '`) ' .
```

Finally, an instantiation is printed by using an auxiliary function `haskellifyPL` in charge of the parameters:

```
eq haskellifyME(ME {PL}) = ' '`( 'InstantiationModExp haskellifyME(ME)
                           '`[ haskellifyPL(PL)  '`]  '`) ' .
```

7.3 Logic

Once the Maude modules have been translated into their abstract syntax, we must implement the type classes `Language` and `Logic` provided by Hets, that define the types needed to represent each logic as an institution and the comorphisms between them. In the following section we describe the most important of these datatypes, while more details about them can be found in [21].

7.3.1 Signature

The signature defines types for the sorts, kinds, subsort relations, operators, and sentences, and a map relating each sort to its corresponding kind:

```
type SortSet = SymbolSet
type KindSet = SymbolSet
type SubsortRel = SymbolRel
type OpDecl = (Set Symbol, [Attr])
type OpDeclSet = Set OpDecl
type OpMap = Map Qid OpDeclSet
type Sentences = Set Sentence
type KindRel = Map Symbol Symbol
```

These types are used to define `Sign`, that stands for Maude signatures:

```
data Sign = Sign {
        sorts :: SortSet,
        kinds :: KindSet,
        subsorts :: SubsortRel,
        ops :: OpMap,
        sentences :: Sentences,
        kindRel :: KindRel
        } deriving (Show, Ord, Eq)
```

The function `fromSpec` extracts the signature from a module:

```
fromSpec :: Module -> Sign
```

This type class also provides functions to join and intersect signatures:

```
union :: Sign -> Sign -> Sign
intersection :: Sign -> Sign -> Sign
```

7.3.2 Sentences

The type for sentences distinguishes between membership axioms, equations, and rules:

```
data Sentence = Membership Membership
              | Equation Equation
              | Rule Rule
              deriving (Show, Read, Ord, Eq)
```

The sentences can be extracted from a module with `fromSpec`, that uses an auxiliary function to obtain them from the statements:

```
fromSpec :: Module -> [Sentence]
fromSpec (Module _ _ stmts) = fromStatements stmts
```

This auxiliary function generates the sentences inferred from the operator attributes such as `assoc` or `comm` with `fromOperator`, while the rest of statements are translated identically:

```
fromStatements :: [Statement] -> [Sentence]
fromStatements stmts = let
   convert stmt = case stmt of
       OpStmnt op -> fromOperator op
       MbStmnt mb -> [Membership mb]
       EqStmnt eq -> [Equation eq]
       RlStmnt rl -> [Rule rl]
       _ -> []
   in concatMap convert stmts
```

`fromOperator` traverses the attributes and generates the appropriate sentence for each of them:

```
fromOperator :: Operator -> [Sentence]
fromOperator (Op op dom cod attrs) = let
   name = getName op
   first = head dom
   second = head $ tail dom
   convert attr = case attr of
       Comm -> commEq name first second cod
       Assoc -> assocEq name first second cod
       Idem -> idemEq name first cod
       Id term -> identityEq name first term cod
       LeftId term -> leftIdEq name first term cod
       RightId term -> rightIdEq name first term cod
       _ -> []
   in concatMap convert attrs
```

We show how the sentence for commutativity is created; the rest of them are produced analogously. The function `commEq` generates two variables of the given sort, which are provided as arguments to the operator in different order, thus creating two terms that are made equal by means of an equation:

```
commEq :: Qid -> Type -> Type -> Type -> [Sentence]
commEq op ar1 ar2 co = [Equation $ Eq t1 t2 [] []]
   where v1 = mkVar "v1" $ type2Kind ar1
         v2 = mkVar "v2" $ type2Kind ar2
         t1 = Apply op [v1, v2] $ type2Kind co
         t2 = Apply op [v2, v1] $ type2Kind co
```

7.3.3 Morphisms

To define the Maude morphisms we first declare maps for sorts, kinds (induced from the previous maps), operators, and labels as map of symbols, a generic identifier for the different data that appears in Maude modules:

```
type SortMap = SymbolMap
type KindMap = SymbolMap
type OpMap = SymbolMap
type LabelMap = SymbolMap
```

We create our morphisms by adding to these data types the source and target signatures:

```
data Morphism = Morphism {
        source :: Sign,
        target :: Sign,
        sortMap :: SortMap,
        kindMap :: KindMap,
        opMap :: OpMap,
        labelMap :: LabelMap
    } deriving (Show, Ord, Eq)
```

Morphisms can be obtained from a list of renamings with `fromSignRenamings`:

```
fromSignRenamings :: Sign -> [Renaming] -> Morphism
```

The type class also provides functions to join and compose morphisms:

```
union :: Morphism -> Morphism -> Morphism
compose :: Morphism -> Morphism -> Result Morphism
```

and to generate an inclusion morphism, that is, an identity morphism between two signatures:

```
inclusion :: Sign -> Sign -> Morphism
inclusion src tgt = Morphism {
        source = src,
        target = tgt,
        sortMap  = Map.empty,
        kindMap  = Map.empty,
        opMap    = Map.empty,
        labelMap = Map.empty
    }
```

7.4 Development graph

We describe in this section the main functions used to draw the development graph for Maude specifications. The most important function is `anaMaudeFile`, that receives a record of all the options received from the command line (of type `HetcatsOpts`) and the path of the Maude file to be parsed and returns a pair with the library name and its environment. This environment contains two development graphs, the first one containing the modules used in the Maude prelude and another one with the user specification:

```
anaMaudeFile :: HetcatsOpts -> FilePath -> IO (Maybe (LibName, LibEnv))
anaMaudeFile _ file = do
    (dg1, dg2) <- directMaudeParsing file
    let ln = emptyLibName file
        lib1 = Map.singleton preludeLib $
                 computeDGraphTheories Map.empty $ markFree Map.empty $
                 markHiding Map.empty dg1
        lib2 = Map.insert ln
                 (computeDGraphTheories lib1 $ markFree lib1 $
                 markHiding lib1 dg2) lib1
    return $ Just (ln, lib2)
```

This environment is computed with the function `directMaudeParsing`, that receives the path introduced by the user and returns a pair of development graphs. These graphs are obtained with the function `maude2DG`, that receives the predefined specifications (obtained with `predefinedSpecs`) and the user defined specifications (obtained with `traverseSpecs`):

```
directMaudeParsing :: FilePath -> IO (DGraph, DGraph)
directMaudeParsing fp = do
  ml <- getEnvDef "MAUDE_LIB" ""
  if null ml then error "environment variable MAUDE_LIB is not set" else do
    ns <- parse fp
    let ns' = either (const []) id ns
    (hIn, hOut, hErr, procH) <- runMaude
    exitCode <- getProcessExitCode procH
    case exitCode of
      Nothing -> do
              hPutStrLn hIn $ "load " ++ fp
              hFlush hIn
              hPutStrLn hIn "."
              hFlush hIn
              hPutStrLn hIn "in Maude/hets.prj"
              psps <- predefinedSpecs hIn hOut
              sps <- traverseSpecs hIn hOut ns'
              (ok, errs) <- getErrors hErr
              if ok
                  then do
                        hClose hIn
                        hClose hOut
                        hClose hErr
                        return $ maude2DG psps sps
                  else do
                        hClose hIn
                        hClose hOut
                        hClose hErr
```

```
                       error errs
      Just ExitSuccess -> error "maude terminated immediately"
      Just (ExitFailure i) -> error $ "calling maude failed with exitCode: " ++ show i
```

The function maude2DG first computes the data structures associated to the predefined specifications and then uses them to compute the development graph related to the specifications introduced by the user. These data structures are computed with insertSpecs:

```
maude2DG :: [Spec] -> [Spec] -> (DGraph, DGraph)
maude2DG psps sps = (dg1, dg2)
   where (_, tim, vm, tks, dg1) = insertSpecs psps emptyDG Map.empty
                                               Map.empty Map.empty [] emptyDG
         (_,_, _, _, dg2) = insertSpecs sps dg1 tim Map.empty vm tks emptyDG
```

Before describing this function, we briefly explain the data structures used during the generation of the development graph:

- The type ParamSort defines a pair with a symbol representing a sort and a list of tokens indicating the parameters present in the sort, so for example the sort List{X, Y} generates the pair (List{X, Y}, [X,Y]):

  ```
  type ParamSort = (Symbol, [Token])
  ```

- The information of each node introduced in the development graph is stored in the tuple ProcInfo, that contains the following information:
  - The identifier of the node.
  - The signature of the node.
  - A list of symbols standing for the sorts that are not instantiated.
  - A list of triples with information about the parameters of the specification, namely the name of the parameter, the name of the theory, and the list of not instantiated sorts from this theory.
  - A list with information about the parameterized sorts.

  ```
  type ProcInfo = (Node, Sign, Symbols, [(Token, Token, Symbols)], [ParamSort])
  ```

- Each ProcInfo tuple is associated to its corresponding module expression in the TokenInfoMap map:

  ```
  type TokenInfoMap = Map.Map Token ProcInfo
  ```

- When a module expression is parsed a ModExpProc tuple is returned, containing the following information:
  - The identifier of the module expression.
  - The TokenInfoMap structure updated with the data in the module expression.
  - The morphism associated to the module expression.
  - The list of sorts parameterized in this module expression.
  - The development graph thus far.

  ```
  type ModExpProc = (Token, TokenInfoMap, Morphism, [ParamSort], DGraph)
  ```

- When parsing a list of importation statements we return a ParamInfo tuple, containing:
  - The list of parameter information: the name of the parameter, the name of the theory, and the sorts that are not instantiated.
  - The updated TokenInfoMap map.
  - The list of morphisms associated with each parameter.

  - The updated development graph.

```
type ParamInfo = ([(Token, Token, Symbols)], TokenInfoMap, [Morphism], DGraph)
```

- Data about views is kept in a separated way from data about theories and modules. The `ViewMap` map associates to each view identifier a tuple with:
  - The identifier of the target node of the view.
  - The morphism generated by the view.
  - The list of renamings that generated the morphism.
  - A Boolean value indicating whether the target is a theory (`True`) or a module (`False`).

```
type ViewMap = Map.Map Token (Node, Token, Morphism, [Renaming], Bool)
```

- Finally, we describe the tuple `InsSpecRes`, used to return the data structures updated when a specification or a view is introduced in the development graph. It contains:
  - Two values of type `TokenInfoMap`. The first one includes all the information related to the specification, including the one from the predefined modules, while the second one only contains information related to the current development graph.
  - The updated `ViewMap`.
  - A list of tokens indicating the theories introduced thus far.
  - The new development graph.

```
type InsSpecRes = (TokenInfoMap, TokenInfoMap, ViewMap, [Token], DGraph)
```

The function `insertSpecs` traverses the specifications updating the data structures and the development graph with `insertSpec`:

```
insertSpecs :: [Spec] -> DGraph -> TokenInfoMap -> TokenInfoMap -> ViewMap -> [Token] -> DGraph
               -> InsSpecRes
insertSpecs [] _ ptim tim vm tks dg = (ptim, tim, vm, tks, dg)
insertSpecs (s : ss) pdg ptim tim vm ths dg = insertSpecs ss pdg ptim' tim' vm' ths' dg'
              where (ptim', tim', vm', ths', dg') = insertSpec s pdg ptim tim vm ths dg
```

The behavior of `insertSpec` is different for each type of Maude specification. When the introduced specification is a module, the following actions are performed:

- The parameters are parsed:
  - The list of parameter declarations is obtained with the auxiliary function `getParams`.
  - These declarations are processed with `processParameters`, that returns a tuple of type `ParamInfo` described above.
  - Given the parameters names, we traverse the list of sorts to check whether the module defines parameterized sorts with `getSortsParameterizedBy`.
  - The links between the theories in the parameters and the current module are created with `createEdgesParams`.
- The importations are handled:
  - The importation statements are obtained with `getImportsSorts`. Although this function also returns the sorts declared in the module, in this case they are not needed and its value is ignored.
  - These importations are handled by `processImports`, that returns a list containing the information of each parameter.
  - The definition links generated by the imports are created with `createEdgesImports`.

- The final signature is obtained with sign_union_morphs by merging the signature in the current module with the ones obtained from the morphisms from the parameters and the imports.

```
insertSpec :: Spec -> DGraph -> TokenInfoMap -> TokenInfoMap -> ViewMap -> [Token] -> DGraph
              -> InsSpecRes
insertSpec (SpecMod sp_mod) pdg ptim tim vm ths dg = (ptimUp, tim5, vm, ths, dg6)
      where ps = getParams sp_mod
            (il, _) = getImportsSorts sp_mod
            up = incPredImps il pdg (ptim, tim, dg)
            (ptimUp, timUp, dgUp) = incPredParams ps pdg up
            (pl, tim1, morphs, dg1) = processParameters ps timUp dgUp
            top_sg = Maude.Sign.fromSpec sp_mod
            paramSorts = getSortsParameterizedBy (paramNames ps) (Set.toList $ sorts top_sg
            ips = processImports tim1 vm dg1 il
            (tim2, dg2) = last_da ips (tim1, dg1)
            sg = sign_union_morphs morphs $ sign_union top_sg ips
            ext_sg = makeExtSign Maude sg
            nm_sns = map (makeNamed "") $ Maude.Sentence.fromSpec sp_mod
            sens = toThSens nm_sns
            gt = G_theory Maude ext_sg startSigId sens startThId
            tok = HasName.getName sp_mod
            name = makeName tok
            (ns, dg3) = insGTheory dg2 name DGBasic gt
            tim3 = Map.insert tok (getNode ns, sg, [], pl, paramSorts) tim2
            (tim4, dg4) = createEdgesImports tok ips sg tim3 dg3
            dg5 = createEdgesParams tok pl morphs sg tim4 dg4
            (_, tim5, dg6) = insertFreeNode tok tim4 morphs dg5
```

When the specification inserted is a theory the process varies slightly:

- Theories cannot be parameterized in Core Maude, so the parameter handling is not required.
- The specified sorts have to be qualified with the parameter name when used in a parameterized module. These sorts are extracted with getImportsSorts and kept in the corresponding field of TokenInfoMap.

```
insertSpec (SpecTh sp_th) pdg ptim tim vm ths dg = (ptimUp, tim3, vm, tok : ths, dg3)
      where (il, ss1) = getImportsSorts sp_th
            (ptimUp, timUp, dgUp) = incPredImps il pdg (ptim, tim, dg)
            ips = processImports timUp vm dgUp il
            ss2 = getThSorts ips
            (tim1, dg1) = last_da ips (tim, dg)
            sg = sign_union (Maude.Sign.fromSpec sp_th) ips
            ext_sg = makeExtSign Maude sg
            nm_sns = map (makeNamed "") $ Maude.Sentence.fromSpec sp_th
            sens = toThSens nm_sns
            gt = G_theory Maude ext_sg startSigId sens startThId
            tok = HasName.getName sp_th
            name = makeName tok
            (ns, dg2) = insGTheory dg1 name DGBasic gt
            tim2 = Map.insert tok (getNode ns, sg, ss1 ++ ss2, [], []) tim1
            (tim3, dg3) = createEdgesImports tok ips sg tim2 dg2
```

The introduction of views into the development graph follows these steps:

- The function isInstantiated checks whether the target of the view is a theory or a module. This value will be used to decide whether the sorts have to be qualified when this view is used.
- A morphism is generated between the signatures of the source and target specifications.
- If there is a renaming between terms the function sign4renamings generates the extra signature and sentences needed. These values, kept in new_sign and new_sens are used to create an inner node with the function insertInnerNode.

- Finally, a theorem link stating the proof obligations generated by the view is introduced between the source and the target of the view with `insertThmEdgeMorphism`.

```
insertSpec (SpecView sp_v) pdg ptim tim vm ths dg = (ptimUp, tim3, vm', ths, dg4)
      where View name from to rnms = sp_v
            (ptimUp, timUp, dgUp) = incPredView from to pdg (ptim, tim, dg)
            inst = isInstantiated ths to
            tok_name = HasName.getName name
            (tok1, tim1, morph1, _, dg1) = processModExp timUp vm dgUp from
            (tok2, tim2, morph2, _, dg2) = processModExp tim1 vm dg1 to
            (n1, _, _, _, _) = fromJust $ Map.lookup tok1 tim2
            (n2, _, _, _, _) = fromJust $ Map.lookup tok2 tim2
            morph = fromSignsRenamings (target morph1) (target morph2) rnms
            morph' = fromJust $ maybeResult $ compose morph1 morph
            (new_sign, new_sens) = sign4renamings (target morph1) (sortMap morph) rnms
            (n3, tim3, dg3) = insertInnerNode n2 tim2 tok2 morph2 new_sign new_sens dg2
            vm' = Map.insert (HasName.getName name) (n3, tok2, morph', rnms, inst) vm
            dg4 = insertThmEdgeMorphism tok_name n3 n1 morph' dg3
```

We describe now the main auxiliary functions used above. Module expressions are parsed following the guidelines outlined in Section 5.1.2:

- When the module expression is a simple identifier its signature and its parameterized sorts are extracted from the `TokenInfoMap` and returned, while the generated morphism is an inclusion:

```
processModExp :: TokenInfoMap -> ViewMap -> DGraph -> ModExp -> ModExpProc
processModExp tim _ dg (ModExp modId) = (tok, tim, morph, ps, dg)
                     where tok = HasName.getName modId
                           (_, sg, _, _, ps) = fromJust $ Map.lookup tok tim
                           morph = Maude.Morphism.inclusion sg sg
```

- The parsing of the summation expression performs the following steps:
  - The information about the module expressions is recursively computed with `processModExp`.
  - The signature of the resulting module expression is obtained with the `union` of signatures.
  - The morphism generated by the summation is just an inclusion.
  - A new node for the summation is introduced with `insertNode`.
  - The target signature of the obtained morphisms is substituted by this new signature with `setTarget`.
  - These new morphisms are used to generate the links between the summation and its summands in `insertDefEdgeMorphism`.

```
processModExp tim vm dg (SummationModExp modExp1 modExp2) = (tok, tim3, morph, ps', dg5)
          where (tok1, tim1, morph1, ps1, dg1) = processModExp tim vm dg modExp1
                (tok2, tim2, morph2, ps2, dg2) = processModExp tim1 vm dg1 modExp2
                ps' = deleteRepeated $ ps1 ++ ps2
                tok = mkSimpleId $ concat ["{", show tok1, " + ", show tok2, "}"]
                (n1, _, ss1, _, _) = fromJust $ Map.lookup tok1 tim2
                (n2, _, ss2, _, _) = fromJust $ Map.lookup tok2 tim2
                ss1' = translateSorts morph1 ss1
                ss2' = translateSorts morph1 ss2
                sg1 = target morph1
                sg2 = target morph2
                sg = Maude.Sign.union sg1 sg2
                morph = Maude.Morphism.inclusion sg sg
                morph1' = setTarget sg morph1
                morph2' = setTarget sg morph2
                (tim3, dg3) = insertNode tok sg tim2 (ss1' ++ ss2') [] dg2
                (n3, _, _, _, _) = fromJust $ Map.lookup tok tim3
                dg4 = insertDefEdgeMorphism n3 n1 morph1' dg3
                dg5 = insertDefEdgeMorphism n3 n2 morph2' dg4
```

- The renaming module expression recursively parses the inner expression, computes the morphism from the given renamings with `fromSignRenamings`, taking special care of the renaming of the parameterized sorts with `applyRenamingParamSorts`. Once the values are computed, the final morphism is obtained from the composition of the morphisms computed for the inner expression and the one computed from the renamings:

```
processModExp tim vm dg (RenamingModExp modExp rnms) = (tok, tim', comp_morph, ps', dg')
              where (tok, tim', morph, ps, dg') = processModExp tim vm dg modExp
                    morph' = fromSignRenamings (target morph) rnms
                    ps' = applyRenamingParamSorts (sortMap morph') ps
                    comp_morph = fromJust $ maybeResult $ compose morph morph'
```

- The parsing of the instantiation module expression works as follows:
  - The information of the instantiated parameterized module is obtained with `processModExp`.
  - The parameter names are obtained by applying `fstTpl`, that extracts the first component of a triple, to the information about the parameters of the parameterized module.
  - Parameterized sorts are instantiated with `instantiateSorts`, that returns the new parameterized sorts, in case the target of the view is a theory, and the morphism associated.
  - The view identifiers are processed with `processViews`. This function returns the token identifying the list of views, the morphism to be applied from the parameterized module, a list of pairs of nodes and morphisms, indicating the morphism that has to be used in the link from each view, and a list with the updated information about the parameters due to the views with theories as target.
  - The morphism returned is the inclusion morphism.
  - The links between the targets of the views and the expression are created with `updateGraphViews`.

```
processModExp tim vm dg (InstantiationModExp modExp views) =
                                  (tok'', tim'', final_morph, new_param_sorts, dg'')
       where (tok, tim', morph, paramSorts, dg') = processModExp tim vm dg modExp
             (_, _, _, ps, _) = fromJust $ Map.lookup tok tim'
             param_names = map fstTpl ps
             view_names = map HasName.getName views
             (new_param_sorts, ps_morph) = instantiateSorts param_names
                                                 view_names vm morph paramSorts
             (tok', morph1, ns, deps) = processViews views (mkSimpleId "") tim'
                                              vm ps (ps_morph, [], [])
             tok'' = mkSimpleId $ concat [show tok, "{", show tok', "}"]
             sg2 = target morph1
             final_morph = Maude.Morphism.inclusion sg2 sg2
             (tim'', dg'') = if Map.member tok'' tim
                             then (tim', dg')
                             else updateGraphViews tok tok'' sg2 morph1 ns tim' deps dg'
```

We present the function `insertNode` to describe how the nodes are introduced into the development graph. This function receives the identifier of the node, its signature,[12] the `TokenInfoMap` map, a list of sorts, and information about the parameters, and returns the updated map and the new development graph. First, it checks whether the node is already in the development graph. If it is in the graph, the current map and graph are returned. Otherwise, the extended signature is computed with `makeExtSign` and used to create a graph theory that will be inserted with `insGTheory`, obtaining the new node information and the new development graph. Finally, the map is updated with the information received as parameter and the node identifier obtained when the node was introduced:

```
insertNode :: Token -> Sign -> TokenInfoMap -> Symbols -> [(Token, Token, Symbols)]
            -> DGraph -> (TokenInfoMap, DGraph)
insertNode tok sg tim ss deps dg = if Map.member tok tim
                    then (tim, dg)
```

[12]Note that when the function `insertNode` is used there are no sentences.

```
                    else let
                           ext_sg = makeExtSign Maude sg
                           gt = G_theory Maude ext_sg startSigId noSens startThId
                           name = makeName tok
                           (ns, dg') = insGTheory dg name DGBasic gt
                           tim' = Map.insert tok (getNode ns, sg, ss, deps, []) tim
                         in (tim', dg')
```

The function insertDefEdgeMorphism describes how the definition links are introduced into the development graph. It receives the identifier of the source and target nodes, the morphism to be used in the link, and the current development graph. The morphism is transformed into a development graph morphism indicating the current logic (Maude) and the type (globalDef) and is introduced in the development graph with insLEdgeDG:

```
insertDefEdgeMorphism :: Node -> Node -> Morphism -> DGraph -> DGraph
insertDefEdgeMorphism n1 n2 morph dg = snd $ insLEdgeDG (n2, n1, edg) dg
                    where mor = G_morphism Maude morph startMorId
                          edg = globDefLink (gEmbed mor) SeeTarget
```

Theorem links are introduced with insertThmEdgeMorphism in the same way, but specifying with globalThm that the link is a theorem link. This function receives as extra argument the name of the view generating the proof obligations, that is used to name the link:

```
insertThmEdgeMorphism :: Token -> Node -> Node -> Morphism -> DGraph -> DGraph
insertThmEdgeMorphism name n1 n2 morph dg = snd $ insLEdgeDG (n2, n1, edg) dg
                    where mor = G_morphism Maude morph startMorId
                          edg = defDGLink (gEmbed mor) globalThm
                                (DGLinkView name $ Fitted [])
```

The function insertFreeEdge receives the names of the nodes and the TokenInfoMap and builds an inclusion morphism to use it in the FreeOrCofreeDefLink link:

```
insertFreeEdge :: Token -> Token -> TokenInfoMap -> DGraph -> DGraph
insertFreeEdge tok1 tok2 tim dg = snd $ insLEdgeDG (n2, n1, edg) dg
          where (n1, _, _, _, _) = fromJust $ Map.lookup tok1 tim
                (n2, sg2, _, _, _) = fromJust $ Map.lookup tok2 tim
                mor = G_morphism Maude (Maude.Morphism.inclusion Maude.Sign.empty sg2) startMorId
                dgt = FreeOrCofreeDefLink NPFree $ EmptyNode (Logic Maude)
                edg = defDGLink (gEmbed mor) dgt SeeTarget
```

7.5 Comorphism

We show in this section how the comorphism from Maude to CASL described in Section 4 is implemented. The function in charge of computing the comorphism is maude2casl, that returns the CASL signature and sentences given the Maude signature and sentences:

```
maude2casl :: MSign.Sign -> [Named MSentence.Sentence] -> (CSign.CASLSign,
                                                          [Named CAS.CASLFORMULA])
```

This function splits the work into different stages:

- The function rewPredicates generates the rew predicates for each sort to simulate the rewrite rules in the Maude specification.
- The function rewPredicatesSens creates the formulas associated to the rew predicates created above, stating that they are reflexive and transitive.
- The CASL operators are obtained from the Maude operators:
  - The function translateOps splits the Maude operator map into a tuple of CASL operators and CASL associative operators, (which are required for parsing purposes).
  - Since CASL does not allow the definition of polymorphic operators, these operators are removed from the map with deleteUniversal and for each one of these Maude operators we create a set of CASL operators with all the possible profiles with universalOps.

- Casl sentences are obtained from the Maude sentences and from predefined Casl libraries:
  - In the computation of the Casl formulas we split Maude sentences in equations defined without the `owise` attribute, equations defined with `owise`, and the rest of statements with the function `splitOwiseEqs`.
  - The equations defined without the `owise` attribute are translated as universally quantified equations, as shown in Section 4, with `noOwiseSen2Formula`.
  - Equations with the `owise` attribute are translated using a negative existential quantification, as we will show later, with the function `owiseSen2Formula`. This function requires as additional parameter the definition of the formulas defined without the `owise` attribute, in order to state that the equations defined with `owise` are applied when the rest of possible equations cannot.
  - The rest of statements, namely memberships and rules, are translated with the function `mb_rl2formula`.
  - There are some built-in operators in Maude that are not defined by means of equations. To allow the user to reason about them we provide some libraries with the definitions of these operators as Casl formulas, obtained with `loadLibraries`.
- Finally, the Casl symbols are created:
  - The kinds are translated to symbols with `kinds2syms`.
  - The operators are translated with `ops2symbols`.
  - The symbol predicates are obtained with `preds2syms`.

```
maude2casl msign nsens = (csign { CSign.sortSet = cs,
                            CSign.sortRel = sbs',
                            CSign.opMap = cops',
                            CSign.assocOps = assoc_ops,
                            CSign.predMap = preds,
                            CSign.declaredSymbols = syms }, new_sens)
   where csign = CSign.emptySign ()
         ss = MSign.sorts msign
         ss' = Set.map sym2id ss
         mk = kindMapId $ MSign.kindRel msign
         sbs = MSign.subsorts msign
         sbs' = maudeSbs2caslSbs sbs mk
         cs = Set.union ss' $ kindsFromMap mk
         preds = rewPredicates cs
         rs = rewPredicatesSens cs
         ops = deleteUniversal $ MSign.ops msign
         ksyms = kinds2syms cs
         (cops, assoc_ops, _) = translateOps mk ops
         cops' = universalOps cs cops $ booleanImported ops
         rs' = rewPredicatesCongSens cops'
         pred_forms = loadLibraries (MSign.sorts msign) ops
         ops_syms = ops2symbols cops'
         (no_owise_sens, owise_sens, mbs_rls_sens) = splitOwiseEqs nsens
         no_owise_forms = map (noOwiseSen2Formula mk) no_owise_sens
         owise_forms = map (owiseSen2Formula mk no_owise_forms) owise_sens
         mb_rl_forms = map (mb_rl2formula mk) mbs_rls_sens
         preds_syms = preds2syms preds
         syms = Set.union ksyms $ Set.union ops_syms preds_syms
         new_sens = concat [rs, rs', no_owise_forms, owise_forms,
                            mb_rl_forms, pred_forms]
```

The `rew` predicates are declared with the function `rewPredicates`, that traverses the set of sorts applying the function `rewPredicate`:

```
rewPredicates :: Set.Set Id -> Map.Map Id (Set.Set CSign.PredType)
rewPredicates = Set.fold rewPredicate Map.empty
```

This function defines a binary predicate using as identifier the constant `rewID` and the sort as type of the arguments:

```
rewPredicate :: Id -> Map.Map Id (Set.Set CSign.PredType)
               -> Map.Map Id (Set.Set CSign.PredType)
rewPredicate sort m = Map.insertWith (Set.union) rewID ar m
   where ar = Set.singleton $ CSign.PredType [sort, sort]
```

Once these predicates have been declared, we have to introduce formulas to state their properties. The function `rewPredicatesSens` accomplishes this task by traversing the set of sorts and applying `rewPredicateSens`:

```
rewPredicatesSens :: Set.Set Id -> [Named CAS.CASLFORMULA]
rewPredicatesSens = Set.fold rewPredicateSens []
```

This function generates the formulas for each sort:

```
rewPredicateSens :: Id -> [Named CAS.CASLFORMULA] -> [Named CAS.CASLFORMULA]
rewPredicateSens sort acc = ref : trans : acc
        where ref = reflSen sort
              trans = transSen sort
```

We describe the formula for reflexivity, being the formula for transitivity analogous. A new variable of the required sort is created with the auxiliary function `newVar`, then the qualified predicate name is created with the `rewID` constant and applied to the variable. Finally, the formula is named with the prefix `rew_refl_` followed by the name of the sort:

```
reflSen :: Id -> Named CAS.CASLFORMULA
reflSen sort = makeNamed name $ quantifyUniversally form
        where v = newVar sort
              pred_type = CAS.Pred_type [sort, sort] nullRange
              pn = CAS.Qual_pred_name rewID pred_type nullRange
              form = CAS.Predication pn [v, v] nullRange
              name = "rew_refl_" ++ show sort
```

The function `translateOps` traverses the map of Maude operators, applying to each of them the function `translateOpDeclSet`:

```
translateOps :: IdMap -> MSign.OpMap -> OpTransTuple
translateOps im = Map.fold (translateOpDeclSet im) (Map.empty, Map.empty, Set.empty)
```

Since the values in the Maude operator map are sets of operator declarations, the auxiliary function `translateOpDeclSet` has to traverse these sets, applying `translateOpDecl` to each operator declaration:

```
translateOpDeclSet :: IdMap -> MSign.OpDeclSet -> OpTransTuple -> OpTransTuple
translateOpDeclSet im ods tpl = Set.fold (translateOpDecl im) tpl ods
```

The function `translateOpDecl` receives an operator declaration, that consists of all the operators declared with the same profile at the kind level. The function traverses these operators, transforming them into Casl operators with the function `ops2pred` and returning a tuple containing the operators, the associative operators, and the constructors:

```
translateOpDecl :: IdMap -> MSign.OpDecl -> OpTransTuple -> OpTransTuple
translateOpDecl im (syms, ats) (ops, assoc_ops, cs) = case tl of
                     [] -> (ops', assoc_ops', cs')
                     _ -> translateOpDecl im (syms', ats) (ops', assoc_ops', cs')
      where sym = head $ Set.toList syms
            tl = tail $ Set.toList syms
            syms' = Set.fromList tl
            (cop_id, ot, _) = fromJust $ maudeSym2CASLOp im sym
            cop_type = Set.singleton ot
            ops' = Map.insertWith (Set.union) cop_id cop_type ops
            assoc_ops' = if any MAS.assoc ats
                         then Map.insertWith (Set.union) cop_id cop_type assoc_ops
                         else assoc_ops
            cs' = if any MAS.ctor ats
                  then Set.insert (Component cop_id ot) cs
                  else cs
```

As said above, Maude equations that are not defined with the `owise` attribute are translated to CASL with `noOwiseSen2Formula`. This function extracts the current equation from the named sentence, translates it with `noOwiseEq2Formula` and creates a new named sentence with the resulting formula:

```
noOwiseSen2Formula ::  IdMap -> Named MSentence.Sentence -> Named CAS.CASLFORMULA
noOwiseSen2Formula im s = s'
       where MSentence.Equation eq = sentence s
             sen' = noOwiseEq2Formula im eq
             s' = s { sentence = sen' }
```

The function `noOwiseEq2Formula` distinguishes whether the equation is conditional or not. In both cases, the Maude terms in the equation are translated into CASL terms with `maudeTerm2caslTerm`, and a strong equation is used to create a formula. If the equation has no conditions this formula is universally quantified and returned as result, while if it has conditions each of them generates a formula and their conjunction, computed with `conds2formula`, will be used as premise of the equational formula:

```
noOwiseEq2Formula :: IdMap -> MAS.Equation -> CAS.CASLFORMULA
noOwiseEq2Formula im (MAS.Eq t t' [] _) = quantifyUniversally form
      where ct = maudeTerm2caslTerm im t
            ct' = maudeTerm2caslTerm im t'
            form = CAS.Strong_equation ct ct' nullRange
noOwiseEq2Formula im (MAS.Eq t t' conds@(_:_) _) = quantifyUniversally form
      where ct = maudeTerm2caslTerm im t
            ct' = maudeTerm2caslTerm im t'
            conds_form = conds2formula im conds
            concl_form = CAS.Strong_equation ct ct' nullRange
            form = createImpForm conds_form concl_form
```

`maudeTerm2caslTerm` is defined for each Maude term:

- Variables are translated into qualified CASL variables, and their type is translated to the corresponding type in CASL:

  ```
  maudeTerm2caslTerm :: IdMap -> MAS.Term -> CAS.CASLTERM
  maudeTerm2caslTerm im (MAS.Var q ty) = CAS.Qual_var q ty' nullRange
          where ty' = maudeType2caslSort ty im
  ```

- Constants are translated as functions applied to the empty list of arguments:

  ```
  maudeTerm2caslTerm im (MAS.Const q ty) = CAS.Application op [] nullRange
          where name = token2id q
                ty' = maudeType2caslSort ty im
                op_type = CAS.Op_type CAS.Total [] ty' nullRange
                op = CAS.Qual_op_name name op_type nullRange
  ```

- The application of an operator to a list of terms is translated into another application, translating recursively the arguments into valid CASL terms:

  ```
  maudeTerm2caslTerm im (MAS.Apply q ts ty) = CAS.Application op tts nullRange
          where name = token2id q
                tts = map (maudeTerm2caslTerm im) ts
                ty' = maudeType2caslSort ty im
                types_tts = getTypes tts
                op_type = CAS.Op_type CAS.Total types_tts ty' nullRange
                op = CAS.Qual_op_name name op_type nullRange
  ```

The conditions are translated into a conjunction with `conds2formula`, that traverses the conditions applying `cond2formula` to each of them, and then creates the conjunction of the obtained formulas:

```
conds2formula :: IdMap -> [MAS.Condition] -> CAS.CASLFORMULA
conds2formula im conds = CAS.Conjunction forms nullRange
        where forms = map (cond2formula im) conds
```

- Both equality and matching conditions are translated into strong equations:

  ```
  cond2formula :: IdMap -> MAS.Condition -> CAS.CASLFORMULA
  cond2formula im (MAS.EqCond t t') = CAS.Strong_equation ct ct' nullRange
         where ct = maudeTerm2caslTerm im t
               ct' = maudeTerm2caslTerm im t'
  cond2formula im (MAS.MatchCond t t') = CAS.Strong_equation ct ct' nullRange
         where ct = maudeTerm2caslTerm im t
               ct' = maudeTerm2caslTerm im t'
  ```

- Membership conditions are translated into Casl memberships by translating the term and the sort:

  ```
  cond2formula im (MAS.MbCond t s) = CAS.Membership ct s' nullRange
        where ct = maudeTerm2caslTerm im t
              s' = token2id $ getName s
  ```

- Rewrite conditions are translated into formulas by using both terms as arguments of the corresponding `rew` predicate:

  ```
  cond2formula im (MAS.RwCond t t') = CAS.Predication pred_name [ct, ct'] nullRange
         where ct = maudeTerm2caslTerm im t
               ct' = maudeTerm2caslTerm im t'
               ty = token2id $ getName $ MAS.getTermType t
               kind = Map.findWithDefault (errorId "rw cond to formula") ty im
               pred_type = CAS.Pred_type [kind, kind] nullRange
               pred_name = CAS.Qual_pred_name rewID pred_type nullRange
  ```

The equations defined with the `owise` attribute are translated with `owiseSen2Formula`, that traverses them and applies `owiseEq2Formula` to the inner equation:

```
owiseSen2Formula ::  IdMap -> [Named CAS.CASLFORMULA] -> Named MSentence.Sentence
                     -> Named CAS.CASLFORMULA
owiseSen2Formula im owise_forms s = s'
       where MSentence.Equation eq = sentence s
             sen' = owiseEq2Formula im owise_forms eq
             s' = s { sentence = sen' }
```

This function receives all the formulas defined without the `owise` attribute and, for each formula with the same operator in the lefthand side as the current equation (obtained with `getLeftApp`), it generates with `existencialNegationOtherEqs` a negative existential quantification stating that the arguments do not match or the condition does not hold that is used as premise of the equation:

```
owiseEq2Formula :: IdMap -> [Named CAS.CASLFORMULA] -> MAS.Equation -> CAS.CASLFORMULA
owiseEq2Formula im no_owise_form eq = form
      where (eq_form, vars) = noQuantification $ noOwiseEq2Formula im eq
            (op, ts, _) = fromJust $ getLeftApp eq_form
            ex_form = existencialNegationOtherEqs op ts no_owise_form
            imp_form = createImpForm ex_form eq_form
            form = CAS.Quantification CAS.Universal vars imp_form nullRange
```

The translation from sorts, operators, and predicates to symbols works in a similar way to the transformations shown above, so we only describe how the predicate symbols are obtained. The function `preds2syms` traverses the map of predicates and inserts each obtained symbol into the set with `pred2sym`:

```
preds2syms :: Map.Map Id (Set.Set CSign.PredType) -> Set.Set CSign.Symbol
preds2syms = Map.foldWithKey pred2sym Set.empty
```

This function traverses the set of predicate types and creates the symbol corresponding to each one with `createSym4id`:

```
pred2sym :: Id -> Set.Set CSign.PredType -> Set.Set CSign.Symbol -> Set.Set CSign.Symbol
pred2sym pn spt acc = Set.fold (createSym4id pn) acc spt
```

`createSym4id` generates the symbol and inserts it into the accumulated set:

```
createSym4id :: Id -> CSign.PredType -> Set.Set CSign.Symbol -> Set.Set CSign.Symbol
createSym4id pn pt acc = Set.insert sym acc
      where sym = CSign.Symbol pn $ CSign.PredAsItemType pt
```

7.6 Freeness constraints

We describe here how the freeness constraints introduced in Section 5.2 have been implemented. This implementation has been performed for general CASL theories, so it is not specific to Maude specifications. Since this transformation is quite complex, we focus in this section on the second-order formula for the kernel of h (the functions added for each $\iota(s)$). As explained before, this formula is split into several subformulas:

- The formula for *symmetry* of each sort is implemented by the function `symmetry_ax`. It first generates two fresh variables, `v1` and `v2`, of the given sort, and a binary predicate `ps` ranging in this sort; then we create the righthand (`rhs`) and lefthand (`lhs`) sides of the implication, that finally is universally quantified and returned:

```
symmetry_ax :: SORT -> CASLFORMULA
symmetry_ax s = quant
     where free_sort = mkFreeName s
           v1@(Qual_var n1 _ _) = newVarIndex 1 free_sort
           v2@(Qual_var n2 _ _) = newVarIndex 2 free_sort
           pt = Pred_type [free_sort, free_sort] nullRange
           name = phiName s
           ps = Qual_pred_name name pt nullRange
           lhs = Predication ps [v1, v2] nullRange
           rhs = Predication ps [v2, v1] nullRange
           inner_form = Implication lhs rhs True nullRange
           vd = [Var_decl [n1, n2] free_sort nullRange]
           quant = Quantification Universal vd inner_form nullRange
```

- The formulas for *transitivity* are generated with `transitivity_ax`, that works in a similar way to the function above: it first creates the fresh variables `v1`, `v2`, and `v3`, which are used in the `ps` predicate to build the formulas `fst_form` (Φ(`v1`, `v2`)), `snd_form` (Φ(`v2`, `v3`)), and `thr_form` (Φ(`v1`, `v3`)). The first two formulas are put together in the conjunction `conj` and imply the third one in `imp`. Finally, the formula is universally quantified and returned as `quant`:

```
transitivity_ax :: SORT -> CASLFORMULA
transitivity_ax s = quant
     where free_sort = mkFreeName s
           v1@(Qual_var n1 _ _) = newVarIndex 1 free_sort
           v2@(Qual_var n2 _ _) = newVarIndex 2 free_sort
           v3@(Qual_var n3 _ _) = newVarIndex 3 free_sort
           pt = Pred_type [free_sort, free_sort] nullRange
           name = phiName s
           ps = Qual_pred_name name pt nullRange
           fst_form = Predication ps [v1, v2] nullRange
           snd_form = Predication ps [v2, v3] nullRange
           thr_form = Predication ps [v1, v3] nullRange
           conj = mk_conj [fst_form, snd_form]
           imp = Implication conj thr_form True nullRange
           vd = [Var_decl [n1, n2, n3] free_sort nullRange]
           quant = Quantification Universal vd imp nullRange
```

- Formulas stating *congruence* are more complex. The function `congruence_ax` computes the axioms for each operator identifier by using the auxiliary function `congruence_ax_aux`:

```
congruence_ax :: Id -> Set.Set OpType -> [CASLFORMULA] -> [CASLFORMULA]
congruence_ax name sot acc = set_forms
     where set_forms = Set.fold ((:) . (congruence_ax_aux name)) acc sot
```

  This auxiliary function, after renaming the operator and the sorts to obtain the free identifiers, creates the arrays of variables, `xs` and `ys`, and the two terms where they are applied, `fst_term` and `snd_term`. Definedness formulas of the form $D(t)$ are built as $P_s(t, t)$ in `fst_form` and `snd_form`, while the third part of the conjunction is built with `congruence_ax_vars` (not shown here) and kept in `vars_forms`. These formulas are put together as a conjunction and used in the final implication, which is returned once it is universally quantified:

```
congruence_ax_aux :: Id -> OpType -> CASLFORMULA
congruence_ax_aux name ot = cong_form'
     where OpType _ args res = ot
           free_name = mkFreeName name
           free_args = map mkFreeName args
           free_res = mkFreeName res
           free_ot = Op_type Total free_args free_res nullRange
           free_os = Qual_op_name free_name free_ot nullRange
           lgth = length free_args
           xs = createVars 1 free_args
           ys = createVars (1 + lgth) free_args
           fst_term = Application free_os xs nullRange
           snd_term = Application free_os ys nullRange
           phi = phiName res
           pt = Pred_type [free_res, free_res] nullRange
           ps = Qual_pred_name phi pt nullRange
           fst_form = Predication ps [fst_term, fst_term] nullRange
           snd_form = Predication ps [snd_term, snd_term] nullRange
           vars_forms = congruence_ax_vars args xs ys
           conj = mk_conj $ fst_form : snd_form : vars_forms
           concl = Predication ps [fst_term, snd_term] nullRange
           cong_form = Implication conj concl True nullRange
           cong_form' = quantifyUniversally cong_form
```

- The formulas for *satThM* are obtained by applying `free_formula`, which performs the substitutions shown in Section 5.2, to the formulas in the current theory:

```
sat_thm_ax :: [Named CASLFORMULA] -> CASLFORMULA
sat_thm_ax forms = final_form
     where forms' = map (free_formula . sentence) forms
           final_form = mk_conj forms'
```

- Finally, *largerThanKerH* formulas are computed with `larger_than_ker_h`. We split the work between the function `ltkh_sorts`, in charge of the formulas related to the sorts, and `ltkh_preds`, in charge of the formulas related to the predicates:

```
larger_than_ker_h :: Set.Set SORT -> Map.Map Id (Set.Set PredType) -> CASLFORMULA
larger_than_ker_h ss mis = conj
     where ltkhs = ltkh_sorts ss
           ltkhp = ltkh_preds mis
           conj = mk_conj (ltkhs ++ ltkhp)
```

We describe how the `ltkh_sorts` function works. This function traverses all the sorts and applies `ltkh_sort` to each of them:

```
ltkh_sorts :: Set.Set SORT -> [CASLFORMULA]
ltkh_sorts = Set.fold ((:) . ltkh_sort) []
```

This auxiliary function creates variables of the given sort and uses them in the homomorphism function. The terms thus obtained are used in an existential equation to create the premise of the implication, while the conclusion is a predicate with these variables as arguments. Finally, the formula is universally quantified and returned:

```
ltkh_sort :: SORT -> CASLFORMULA
ltkh_sort s = imp'
     where free_s = mkFreeName s
           v1 = newVarIndex 1 free_s
           v2 = newVarIndex 2 free_s
           phi = phiName s
           pt = Pred_type [free_s, free_s] nullRange
           ps = Qual_pred_name phi pt nullRange
```

```
ot_hom = Op_type Partial [free_s] s nullRange
name_hom = Qual_op_name homId ot_hom nullRange
t1 = Application name_hom [v1] nullRange
t2 = Application name_hom [v2] nullRange
prem = Existl_equation t1 t2 nullRange
concl = Predication ps [v1, v2] nullRange
imp = Implication prem concl True nullRange
imp' = quantifyUniversally imp
```

8 Concluding remarks and future work

We have presented how Maude has been integrated into Hets, a parsing, static analysis, and proof management tool that combines various tools for different specification languages. To achieve this integration, we consider preordered algebra semantics for Maude and define an institution comorphism from Maude to Casl. This integration allows to prove properties of Maude specifications like those expressed in Maude views. We have also implemented a normalization of the development graphs that allows us to prove freeness constraints. We have used this transformation to connect Maude to Isabelle [32], a Higher Order Logic prover, and have demonstrated a small example proof about reversal of lists. Moreover, this encoding is suited for proofs of e.g. extensionality of sets, which require first-order logic, going beyond the abilities of existing Maude provers like ITP.

Since interactive proofs are often not easy to conduct, future work will make proving more efficient by adopting automated induction strategies like rippling [12]. We also have the idea to use the automatic first-order prover SPASS for induction proofs by integrating special induction strategies directly into Hets.

We have also studied the possible comorphisms from Casl to Maude. We distinguish whether the formulas in the source theory are confluent and terminating or not. In the first case, that we plan to check with the Maude termination [13] and confluence checker [14], we map formulas to equations, whose execution in Maude is more efficient, while in the second case we map formulas to rules.

Finally, we also plan to relate Hets' Modal Logic and Maude models in order to use the Maude model checker [7, Chapter 13] for linear temporal logic.

Acknowledgments We wish to thank Francisco Durán for discussions regarding freeness in Maude, Martin Kühl for cooperation on the implementation of the system presented here, and Maksym Bortin for help with the Isabelle proofs. This work has been supported by the German Federal Ministry of Education and Research (Project 01 IW 07002 FormalSafe), the German Research Council (DFG) under grant KO-2428/9-1, the Comunidad de Madrid project *PROMETIDOS* (S2009/TIC–1465), the MICINN Spanish project *DESAFIOS10* (TIN2009-14599-C03-01), and the Spanish Ministerio de Educación grant AP2005-007.

References

[1] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. D. Mosses, D. Sannella, and A. Tarlecki. Casl: the common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[2] S. Autexier, D. Hutter, B. Langenstein, H. Mantel, G. Rock, A. Schairer, W. Stephan, R. Vogt, and A. Wolpers. VSE: formal methods meet industrial needs. *International Journal on Software Tools for Technology Transfer*, 3(1):66–77, 2009.

[3] T. Baar, A. Strohmeier, A. M. D. Moreira, and S. J. Mellor, editors. *Proceedings of UML 2004 - The Unified Modelling Language: Modelling Languages and Applications. 7th International Conference*, volume 3273 of *Lecture Notes in Computer Science*. Springer, 2004.

[4] M. Bidoit and P. D. Mosses. Casl *User Manual*, volume 2900 of *Lecture Notes in Computer Science*. Springer, 2004.

[5] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[6] R. Bruni and J. Meseguer. Semantic foundations for generalized rewrite theories. *Theoretical Computer Science*, 360(1):386–414, 2006.

[7] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] M. Clavel, J. Meseguer, and M. Palomino. Reflection in membership equational logic, many-sorted equational logic, Horn logic with equality, and rewriting logic. *Theoretical Computer Science*, 373(1-2):70–91, 2007.

[9] M. Clavel, M. Palomino, and A. Riesco. Introducing the ITP tool: a tutorial. *Journal of Universal Computer Science*, 12(11):1618–1650, 2006. Programming and Languages. Special Issue with Extended Versions of Selected Papers from PROLE 2005: The Fifth Spanish Conference on Programming and Languages.

[10] L. A. Dennis, G. Collins, M. Norrish, R. J. Boulton, K. Slind, and T. F. Melham. The PROSPER toolkit. *International Journal on Software Tools for Technology Transfer (STTT)*, 4(2):189–210, 2002.

[11] R. Diaconescu. *Institution-Independent Model Theory.* Birkhäuser Basel, 2008.

[12] L. Dixon and J. D. Fleuriot. Higher order rippling in IsaPlanner. In K. Slind, A. Bunker, and G. Gopalakrishnan, editors, *Proceedings of the 17th International Conference on Theorem Proving in Higher Order Logics, TPHOLs 2004*, volume 3223 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2004.

[13] F. Durán, S. Lucas, and J. Meseguer. MTT: The Maude Termination Tool (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Proceedings of the 4th International Joint Conference on Automated Reasoning, IJCAR 2008, Sydney, Australia, August 12-15*, volume 5195 of *Lecture Notes in Computer Science*, pages 313–319. Springer, 2008.

[14] F. Durán and J. Meseguer. A Church-Rosser checker tool for conditional order-sorted equational Maude specifications. In *Proceedings of the 8th International Workshop on Rewriting Logic and its Applications, WRLA 2010*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85, 2010.

[15] N. Een and N. Sörensson. An extensible SAT-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003, Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[16] A. Fuchs. Darwin: A Theorem Prover for the Model Evolution Calculus. Master's thesis, University of Koblenz-Landau, 2004.

[17] K. Futatsugi and R. Diaconescu. *CafeOBJ Report.* World Scientific, AMAST Series, 1998.

[18] J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39:95–146, 1992.

[19] C. T. T. Group. CoFI, the common framework initiative for algebraic specification and development. `http://www.cofi.info`.

[20] M. Kohlhase. OMDoc: An infrastructure for OpenMath content dictionary information. *Bulletin of the ACM Special Interest Group on Symbolic and Automated Mathematics (SIGSAM)*, 34(2):43–48, 2000.

[21] M. Kühl. Integrating Maude into Hets. Master's thesis, Universität Bremen, 2010.

[22] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[23] J. Meseguer. Membership algebra as a logical framework for equational specification. In F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3–7, 1997, Selected Papers*, volume 1376 of *Lecture Notes in Computer Science*, pages 18–61. Springer, 1998.

[24] M. W. Moskewicz and C. F. Madigan. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535. ACM Press, 2001.

[25] T. Mossakowski. Relating Casl with other specification languages: the institution level. *Theoretical Computer Science*, 286(2):367–475, 2002.

[26] T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Technical report, Universität Bremen, 2005. Habilitation thesis.

[27] T. Mossakowski, S. Autexier, and D. Hutter. Development graphs - proof management for structured specifications. *Journal of Logic and Algebraic Programming, special issue on Algebraic Specification and Development Techniques*, 67(1-2):114–145, 2006.

[28] T. Mossakowski, C. Maeder, M. Codescu, and D. Lücke. Hets user guide – Version 0.97 –, January 2011.

[29] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In O. Grumberg and M. Huth, editors, *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer, 2007.

[30] T. Mossakowski, C. Maeder, and K. Lüttich. The Heterogeneous Tool Set. In B. Beckert, editor, *VERIFY 2007, 4th International Verification Workshop*, volume 259 of *CEUR Workshop Proceedings*, pages 119–135, 2007.

[31] P. Mosses, editor. Casl *Reference Manual*, volume 2960 of *Lecture Notes in Computer Science*. Springer, 2004.

[32] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.

[33] M. Norrish and K. Slind. The HOL system tutorial, September 2010.

[34] B. C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[35] H. Reichel. *Initial Computability, Algebraic Specifications, and Partial Algebras.* Oxford University Press, 1987.

[36] A. Tarlecki. Institutions: An abstract framework for formal specifications. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specifications*, pages 105–130. Springer, 1999.

[37] D. Tsarkov, A. Riazanov, S. Bechhofer, and I. Horrocks. Using Vampire to reason with OWL. In S. A. McIlraith, D. Plexousakis, and F. van Harmelen, editors, *Proceedings of the 3rd International Semantic Web Conference, ISWC 2004*, volume 3298 of *Lecture Notes in Computer Science*, pages 471–485. Springer, 2004.

[38] P. Viry. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517, 2002.

[39] C. Weidenbach, D. Dimova, A. Fietzke, R. Kumar, M. Suda, and P. Wischnewski. SPASS version 3.5. In R. A. Schmidt, editor, *Proceedings of the 22nd International Conference on Automated Deduction, CADE 2009*, volume 5663 of *Lecture Notes in Artificial Intelligence*, pages 140–145. Springer, 2009.