Reflexión, abstracción y simulación en la lógica de reescritura

TESIS DOCTORAL

Miguel Palomino Tarjuelo

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Noviembre 2004

Reflexión, abstracción y simulación en la lógica de reescritura

Memoria presentada para obtener el grado de
Doctor en Matemáticas

Miguel Palomino Tarjuelo

Dirigida por los doctores

Narciso Martí Oliet
José Meseguer Guaita

Departamento de Sistemas Informáticos y Programación
Facultad de Ciencias Matemáticas
Universidad Complutense de Madrid

Noviembre 2004

A Juliana y Rafael
y
a Mercedes y Pedro

Agradecimientos

Me parece muy apropiado que sea esta la primera sección de la tesis ya que considero que es la más importante. Aunque no estoy seguro de que ella por sí sola sea suficiente para conseguir el título de doctor, lo que sí sé es que sin lo que aquí se cuenta aquel sería un objetivo completamente inaccesible. A lo largo de estos años yo, como cualquier otro en una situación similar, he recibido la ayuda de muchos sin quienes no hubiese podido escribir esta tesis. En mi caso concreto han sido tres las personas cuyo concurso ha resultado fundamental, así que me parece de justicia empezar con ellos y dejar constancia de mi deuda en estos agradecimientos.

Todo empezó el martes 4 de octubre de 1994; o sea, un día antes de lo que yo esperaba. En la facultad yo había leído que la inauguración oficial del curso no sería hasta el miércoles y no pensaba aparecer por allí hasta entonces, así que no me hizo mucha gracia que mi madre me despertase a eso de las diez gritando que acababa de llamar a la facultad (claramente, no se fiaba un pelo de mi) y que le habían dicho que ya se estaban dando clases. Cuando finalmente encontré mi aula, en el plazo récord de una hora, las presentaciones de las dos primeras asignaturas ya habían pasado y estaba a punto de empezar la de la tercera. Y así fue que, como planeado por el destino si me quisiera poner melodramático, el primer profesor que conocí en la universidad fue Pepe, el profesor de Geometría I.

A lo largo de aquel primer curso, no sé si porque venía muy oxidado del colegio o por pura torpeza personal, lo cierto es que pasaba casi tanto tiempo en clase como en tutorías (bueno, quizá exagero un poco) acosando con un montón de dudas a los profesores. Pero llegó el final de las clases y los exámenes de junio, y yo, después de cómo había sido el año, más contento que unas castañuelas con mi nota en Geometría I en el segundo parcial. Y entonces Pepe me animó a hacer algo que ni se me había pasado por la cabeza: presentarme al examen final para intentar mejorarla. Y le creí. Y salió bien. La moraleja a la que llegué entonces, de manera algo retorcida me parece ahora, y que me fue muy útil para el resto de la carrera es que, si me lo proponía, podía sacar la nota que quisiese en cualquier asignatura. De forma quizá accidental, pero fue Pepe quien me lo hizo ver.

Durante segundo, salvo por algún que otro encuentro fortuito en los pasillos, no tuve ningún contacto con Pepe. Cuando terminó el curso me vi, como todos mis compañeros, en la disyuntiva de qué especialidad elegir para el curso siguiente; en mi caso, dudaba entre la sección de Ciencias de la Computación y la de Matemática Fundamental. Y no recuerdo por qué, si por la anécdota que acabo de contar o por algún detalle que se me quedara de alguna de sus tutorías, pero fui a pedir consejo a Pepe. Resulta que para él la

elección estaba clarísima, pero desafortunadamente el que tenía que elegir era yo así que me llevó a hablar con otros profesores y trató de darme la información necesaria para que yo pudiese tomar una decisión informada. Y aunque al final escogí en contra de su criterio no me lo echó en cara y a partir de entonces nuestra relación se estrechó y él se convirtió en una especie de mentor siempre dispuesto a responder mis dudas sobre las paradojas en la Teoría de Conjuntos y a "dispersarme" con libros y artículos sobre divulgación y recreaciones matemáticas. Pero es que además, resulta que entre aquellos profesores con los que me llevó a hablar cuando andaba despistado sin saber qué elegir, y que suene de nuevo la música fatalista de fondo, se encontraba el que sería profesor/tutor/director de beca/director de tesis mío, Narciso Martí.

A Narciso por tanto lo conocí en segundo, pero no fue hasta cuarto cuando me dio clase. A estas alturas supongo que ya nadie pensará que voy con ánimo pelotillero si también reconozco que Narciso fue uno de los mejores profesores que tuve durante la carrera. Y en parte por eso, en parte porque su asignatura (la fatídica Tecnología de la Programación) era de las que más me gustaban, al terminar el curso fui a pedirle información sobre cómo funcionaban las becas de colaboración y la posibilidad de pedir una bajo su supervisión. Yo no andaba muy convencido por entonces, así que quedamos en que durante el verano le echaría un vistazo a cierto libro y pensaría sobre el tema y que volveríamos a hablar a la vuelta. Y así lo hice, solo que el tiempo que dediqué a leer el libro antes de decidir que lo de la beca no iba conmigo fue de dos días. Pero claro, Narciso por su cuenta había decidido otra cosa así que al llegar septiembre y ver que yo no daba señales de vida me llamó a casa, me convocó a su despacho, me arengó y, ahora puedo decir que afortunadamente, consiguió convencerme. Y desde entonces hasta ahora. Narciso fue mi tutor durante los cursos de doctorado y el director de investigación en las sucesivas becas que tuve. Y aunque me dio demasiada cancha en determinados momentos, siempre estuvo dispuesto a guiarme y aconsejarme y ayudarme en todo lo que hiciera falta, y a decirme las cosas claramente, y también a emborronarme con letras moradas las versiones preliminares de todos los trabajos que le pasaba. Y también fue gracias a Narciso, pues fue su director de tesis, que conocí al tercer eslabón en mi cadena, José Meseguer.

Como a Narciso, a José lo conocí un tiempo antes de empezar a trabajar con él, mientras hacía los cursos de doctorado. No fue, sin embargo, hasta pasados dos años que surgió la oportunidad de viajar a la Universidad de Illinois en Urbana-Champaign, en Estados Unidos, para colaborar con él. Inicialmente yo iba con la intención de realizar una estancia de 7 meses que, gracias a sus ánimos y apoyo económico, terminaron siendo 16. Aunque por entonces yo ya tenía alguna cosita para la tesis, donde esta realmente recibió el empuje necesario para el despegue, vuelo y casi aterrizaje fue allí con José, con una conexión electrónica permanente con Narciso que llegó a ser presencial durante un mes en que incluso compartimos apartamento y en el que ya no me quedó más remedio que trabajar en ella. . . Bromas (¿bromas?) aparte, la experiencia de trabajar con un investigador de la talla de José, junto con todo lo que conlleva el vivir fuera de tu país una temporada larga por vez primera, fue extraordinaria. Y por si esto solo fuera poco, además me ofreció la oportunidad de trabajar como ayudante en sus asignaturas e impartir alguna clase que otra. (Vale, lo cierto es que lo de dar clases en inglés al principio no me pareció tan buena idea. . .)

Lo que quiero decir, porque me temo que no haya quedado claro en esta retahíla, es que sin Pepe, sin Narciso o sin José jamás habría llegado tan lejos. Muchas gracias a los tres.

Como no podía ser de otra forma, ha habido muchas otras personas que me han ayudado en esta tesis o han influido en ella de una forma u otra y, aunque de manera más breve, también quiero referirme a ellas.

El trabajo del capítulo 3 sobre reflexión está hecho, además de con los directores de esta tesis, en colaboración con Manuel Clavel. De hecho, fue él quien empezó con toda esta historia en su tesis doctoral hace ya algunos años y sigue siendo el motor fundamental que la impulsa. El plan de Manolo ahora es trabajar en mejorar el ITP (¡si la preparación de los horarios de la facultad le deja algo de tiempo libre!), lo que también tendría su reflejo en otras partes de la tesis.

Paco Durán fue mi guía durante mi primer mes en Champaign. No solo me encontró apartamento y me enseño un montón de cositas útiles sobre la vida en Estados Unidos, sino que ya me aproveché y le robé sus recetas de cocina con las que pude sobrevivir el resto de la estancia. El que también respondiera a mis preguntas sobre todas las herramientas que ha desarrollado sobre Maude no tiene en comparación ninguna importancia.

Aunque con ninguna influencia directa en el contenido de la tesis, el grupo de gente que conocí en "Chambana" siempre estuvo disponible para ayudarme en los pequeños problemas del día a día, en alguno un pelín más grande, en acabar ese trozo de pizza con el que ya no podía y, fundamentalmente, y como no podía ser de otra forma conviviendo con personas de siete nacionalidades distintas, ampliar de manera brutal mis horizontes. A todo ello contribuyeron Ambarish, Joe, Azadeh, Roberto, Peter, Mark-Oliver, Christiano y Prasanna. Especialmente, y aunque es algo injusto destacarlos solo a ellos, Peter Ölveczky (¡la próxima cita es en el estadio del Rosenborg!) y Azadeh Farzan.

Y ya de vuelta en Madrid, Isabel Pita y Alberto Verdejo. Isabel me ofreció la posibilidad de colaborar con ella cuando todavía estaba en Urbana y Alberto siempre está ahí para ayudarte en lo que sea, desde pasarte esa macro de LaTeX que necesitas a sacarte del aburrimiento de traducir una tesis con un partido de tenis; ambos se han revelado como excelentes compañeros.

Por último, me gustaría agradecer a David de Frutos y a Mario Rodríguez el tiempo que amablemente dedicaron a leer detenidamente esta tesis, y sus numerosos y acertados comentarios a la misma.

Podría seguir un poquito más, mencionando a mi familia y otros amigos, pero bueno, creo que eso es algo que, como el valor en la antigua mili, se presupone.

Índice general

Capítulo 1

Introducción

1.1 Panorámica

A nadie se le escapa que el crecimiento de la informática en las últimas décadas y su cada vez mayor influencia en la vida diaria de las personas (en algunas partes del planeta, al menos) ha sido tremendamente espectacular. Una de las consecuencias de este desarrollo es el diseño e implementación de sistemas más y más complejos para tratar las más diversas tareas. En numerosas ocasiones el buen funcionamiento de dichos sistemas es crítico porque de ellos depende la vida de centenares o miles de personas (piénsese en los sistemas de control de vuelo de un avión o de mantenimiento de una central nuclear) y en muchos otros las consecuencias de un error informático, sin ser fatales, sí pueden resultar catastróficas en términos económicos (sistemas bancarios y financieros). En realidad, no hace falta ir en busca de ejemplos tan grandilocuentes: sencillamente, cuando uno escribe un programa o diseña un sistema espera que este se comporte de una manera determinada y que cumpla un propósito concreto. A partir de cierto grado de complejidad de un proyecto, la ausencia de errores pasa a ser un objetivo inalcanzable; sin embargo, ello no debería ser óbice para intentar conseguir que su número sea lo más reducido posible. En este sentido, resulta especialmente interesante el uso de *lenguajes de especificación*, formalismos que por un lado sean flexibles y potentes para permitir manejar diseños de complejidad elevada, pero que por otro lado tengan un significado matemático preciso. Idealmente, dichos lenguajes deberían imponer una cierta disciplina metodológica de forma que se redujera el número de errores durante la etapa de codificación, mientras que disponer de una semántica matemática precisa debería servir para, aun cuando no se pueda asegurar que el sistema esté libre de errores, garantizar que al menos se cumplan las propiedades que a nosotros nos interesan.

Uno de tales mecanismos es la *lógica de reescritura* presentada por José Meseguer a comienzos de los años noventa (Meseguer, 1992). La lógica de reescritura es una extensión de la lógica ecuacional habitual que fue introducida como un modelo para la especificación de sistemas concurrentes para unificar numerosas propuestas anteriores. Desde entonces esta lógica ha demostrado ser un formalismo muy flexible, no solo para la especificación de la concurrencia, sino también como marco lógico y semántico en el que interpretar otras lógicas y modelos de computación (Martí-Oliet y Meseguer, 2002a), dando origen a

una abundante literatura (véanse las referencias en Martí-Oliet y Meseguer, 2002b). Desde su introducción, se la presentó como la base de un lenguaje declarativo de especificación y programación, llamado *Maude* (Clavel et al., 1996, 2004a).

Por lo tanto, en este marco resulta muy interesante acercarse al estudio de la lógica de reescritura desde al menos dos vertientes:

- El estudio de sus propiedades matemáticas, que justifiquen su uso como herramienta de diseño y posibiliten el desarrollo del segundo punto.
- La búsqueda de métodos de demostración que se le puedan aplicar para la verificación de sistemas que con ella se especifiquen.

Del establecimiento del primer punto se ocupó en una gran parte el trabajo original de Meseguer (1992), en el que se define la teoría de modelos de la lógica y se presenta un cálculo deductivo correcto y completo. Dichos resultados se presentaron para el caso más sencillo en el que la lógica ecuacional subyacente a la lógica de reescritura no tiene tipos ni ecuaciones condicionales; en la actualidad, Bruni y Meseguer están trabajando en generalizar aquellas construcciones para manejar una lógica ecuacional mucho más expresiva como es la *lógica de pertenencia* (Meseguer, 1998), y sus primeros resultados han quedado reflejados ya en Bruni y Meseguer (2003).

Otro tema que forma parte del ámbito de los fundamentos es la *reflexión*. La reflexión consiste en la habilidad de un sistema computacional o lógico de acceder a su propio metanivel para controlar su comportamiento. Los trabajos de Clavel y Meseguer (1996, 2002) han servido para clarificar el mismo concepto de reflexión y han demostrado que la lógica de reescritura es reflexiva. Estos trabajos, sin embargo, tampoco consideran la versión más general de la lógica de reescritura soportada por Maude, por lo que entre los objetivos de esta tesis está el demostrar detalladamente que aquellos resultados siguen siendo ciertos para esta. Un primer avance en esta dirección ya se dio en Clavel et al. (2002b). Allí se describe un esbozo de la demostración para el caso general que, a diferencia de las originales en Clavel y Meseguer (1996, 2002), presenta un enfoque más lógico y menos computacional que simplifica en gran medida las pruebas. Además, y como tarea suplementaria, se intenta presentar un enfoque unificador, aprovechando el trabajo previo para el estudio de la reflexión en algunas sublógicas de la lógica de reescritura como son la lógica de pertenencia o la lógica de Horn con igualdad.

La reflexión en la lógica de reescritura tiene, además, una vertiente claramente práctica. En particular, la tesis de Durán (1999) muestra cómo es posible utilizarla para extender Maude con un álgebra de módulos flexible y robusta tratando los módulos como datos y definiendo operaciones sobre ellos sin salirse de la lógica, y la tesis de Clavel (2000) muestra cómo se puede usar la reflexión para definir estrategias de reescritura dentro de la misma lógica y para definir herramientas de prueba. Una línea de investigación distinta fue la iniciada por Basin, Clavel y Meseguer (2004), donde se propone el uso de la reflexión para metarrazonamiento formal. Desde un punto de vista abstracto, se indican una serie de requisitos para que una lógica pueda ser utilizada como marco metalógico; mientras que desde un punto de vista concreto se presenta la lógica de reescritura, o más específicamente su sublógica ecuacional, como una lógica que satisface dichos requisitos y en la cual es posible demostrar formalmente metateoremas sobre

conjuntos de teorías ecuacionales, como teoremas de una teoría universal. Como parte del trabajo sobre reflexión, en esta tesis también se avanza en las ideas propuestas en Basin et al. (2004) y se estudia cómo aplicarlas en un ejemplo concreto.

De las dos vertientes mencionadas más arriba en que uno podría acercarse al estudio de la lógica de reescritura, la segunda, la búsqueda de métodos de demostración, se encuentra menos desarrollada. El sistema Maude dispone de un demostrador inductivo de teoremas, el ITP desarrollado por Clavel (2001, 2004), que permite demostrar teoremas sobre teorías ecuacionales. Para que el funcionamiento y los resultados del ITP sean correctos es necesario que las teorías satisfagan una serie de condiciones, como son la confluencia o la terminación; para la comprobación de estas, Durán ha desarrollado otra serie de (meta)herramientas (Durán, 2000b; Durán y Meseguer, 2000).

Para la verificación de propiedades en sistemas de reescritura, el sistema Maude incluye a partir de su versión 2.0 un *comprobador de modelos* (Eker et al., 2002). La técnica de *comprobación de modelos* (en inglés, model checking), originalmente propuesta en Clarke y Emerson (1981) y Quielle y Sifakis (1982) de manera independiente, se ha revelado como una de las historias más felices en el ámbito de la aplicación de métodos formales a la validación de propiedades de sistemas a nivel industrial. Pero a pesar de todos los éxitos que la comprobación de modelos ha conseguido en los últimos años, es una técnica que presenta una limitación fundamental: el sistema a estudiar debe tener un número finito de estados. La introducción por parte de McMillan (1993) de la comprobación de modelos simbólica ha elevado el tamaño de los sistemas susceptibles de ser tratados con esta técnica hasta límites astronómicos, del orden de 10^{120} estados. Tales límites, sin embargo, siguen siendo insuficientes en numerosas aplicaciones industriales y completamente inútiles (si se intentan aplicar los métodos directamente) en el análisis de sistemas, especialmente software, cuyo número de estados es infinito.

La limitación inherente de los comprobadores de modelos para tratar sistemas infinitos, o simplemente demasiado grandes, ha llevado a muchos investigadores a estudiar técnicas de *abstracción* con las que superarla. Estas técnicas (véase por ejemplo Clarke et al., 1994; Loiseaux et al., 1995; Graf y Saïdi, 1997; Manolios, 2001; Das, 2003) permiten reducir la comprobación de si un sistema satisface una propiedad al estudio de la misma en una versión abstracta y finita del sistema.

Nuestro interés en este campo, que pretendemos abordar en la que entendemos la parte más importante de esta tesis, reside en el estudio de cómo aplicar las técnicas de abstracción en el marco de la lógica de reescritura. Por una parte, nos interesa estudiar posibles generalizaciones de la idea de abstracción en un marco categórico posiblemente más abstracto que aquel en el que habitualmente se presentan. Por otra parte queremos saber cómo definir dichas abstracciones en la lógica de reescritura y Maude, cómo demostrar que realmente lo son, cómo extraer el sistema abstracto correspondiente, y su aplicación en el comprobador de modelos asociado. A este respecto, unos primeros resultados han sido ya publicados en Meseguer et al. (2003), donde describimos un método sencillo para definir abstracciones mediante cocientes de los sistemas originales, a través de ecuaciones que colapsan el conjunto de estados. Nuestro método produce un sistema cociente minimal (en un sentido apropiado) junto con un conjunto de condiciones que deben ser satisfechas para que el resultado sea ejecutable, las cuales se pueden resolver

utilizando herramientas disponibles en el entorno de desarrollo de Maude, como el ITP o el comprobador de confluencia mencionados anteriormente.

Un último objetivo de esta tesis que viene a unificar aún más los temas mencionados con anterioridad es el desarrollo de un prototipo que automatiza el proceso de abstracción, en concreto, el proceso de *abstracción de predicados* introducido por primera vez en Graf y Saïdi (1997). Su diseño es reflexivo y hace un uso fundamental del ITP, al que utiliza como un oráculo capaz de decidir la validez de las numerosas fórmulas en función de las cuales se construye el sistema abstracto.

1.2 Especificación algebraica

A finales de los años 60, la complejidad creciente de las aplicaciones informáticas hizo que muchos investigadores comenzaran a estudiar métodos matemáticos que contribuyeran a un desarrollo más rápido y más seguro (en el sentido de permitir demostrar formalmente propiedades de interés) de aquellas. Como se apunta en Ehrig y Mahr (1985), alrededor del año 1970 se pueden distinguir dos direcciones principales en este proceso. La primera correspondería a la definición matemática de la semántica de los lenguajes de programación, con la semántica axiomática de Hoare (1969) y la semántica denotacional de Scott y Strachey (1971) como ejemplos más destacados. La segunda dirección iría en la línea de un enfoque riguroso de los tipos abstractos de datos en lenguajes de especificación, con el uso de álgebras heterogéneas y su especificación por medio de ecuaciones como fundamento matemático. Entre los trabajos con más influencia en este área están los de Zilles (1974) y los del grupo ADJ (Goguen et al., 1975). Al poco tiempo, ante la necesidad de operaciones modulares que permitieran la especificación de sistemas a mayor nivel que directamente mediante ecuaciones, surgieron los primeros lenguajes de especificación como Clear (Burstall y Goguen, 1977) y OBJ-0 (Goguen, 1979). Desde entonces, el abanico de aplicaciones de la especificación algebraica se ha expandido hasta incluir sistemas software completos y se han desarrollado numerosos formalismos, con sólidas bases en la lógica y la teoría de categorías, para satisfacer las necesidades particulares de aquellas. En particular, además de OBJ3 (Goguen et al., 1988) y del sublenguaje ecuacional de Maude (Clavel et al., 2004a), cabe destacar el reciente desarrollo del lenguaje CASL (Bidoit y Mosses, 2004), que ha surgido con la aspiración de servir de marco común para desarrollos e investigaciones futuras en el área de la especificación algebraica.

Una descripción del estado del arte en este campo se puede encontrar en Astesiano et al. (1999) donde, junto con algunas referencias históricas que complementan las que aparecen en Ehrig y Mahr (1985), también se incluyen descripciones de aspectos más concretos, como la especificación algebraica de sistemas concurrentes o de objetos.

1.3 La lógica de reescritura

Hacia principios de los años noventa, las propuestas de numerosos autores habían dado lugar a la aparición de multitud de lenguajes y modelos para la especificación de sistemas concurrentes, como pudieran ser, entre otros muchos, las redes de Petri (1973),

CSP (Hoare, 1978), CCS (Milner, 1982) o los actores (actors) de Hewitt y Agha (Agha, 1986). Junto con la fragmentación conceptual que esto suponía, muchos de estos formalismos se limitaban a reconocer aspectos aislados de lo que, en determinados contextos, significaba la concurrencia (como por ejemplo la asociatividad y conmutatividad de los axiomas). Surgieron entonces diversas propuestas con el objetivo de unificar los enfoques de la plétora de formalismos existente por aquella época, entre las que se encontraban la máquina química abstracta (chemical abstract machine) de Berry y Boudol (1990), el π-cálculo de Milner et al. (1992) y la lógica de reescritura de Meseguer (1990a, 1992).

A diferencia de propuestas anteriores, en la lógica de reescritura la concurrencia no queda encorsetada en un mecanismo predefinido, sino que es el usuario el que puede decidir cuál es el aspecto de la misma que le interesa destacar en cada ámbito particular. Además de demostrar su flexibilidad para la especificación de sistemas concurrentes en una serie de trabajos de numerosos autores (véanse las referencias en Martí-Oliet y Meseguer, 2002b), la lógica de reescritura también se ha revelado útil como marco lógico y semántico (Martí-Oliet y Meseguer, 2002a). En contraste con las lógicas ecuacionales, que describen un mundo estático, la lógica de reescritura es una lógica *de cambio* con reglas que expresan la transformación de unos estados, descritos mediante teorías ecuacionales, en otros. Esta interpretación computacional de las reglas es lo que permite el uso de la lógica de reescritura como marco semántico para la representación de otros modelos de computación. Pero las reglas también admiten una interpretación lógica en la que los estados pasan a ser fórmulas y las transiciones corresponden a la noción de consecuencia lógica. Esta dualidad es la que en parte hace de la lógica de reescritura un formalismo flexible que se puede adaptar a muchas situaciones.

En la expansión del uso de la lógica de reescritura también ha tenido mucha importancia la existencia de herramientas como CafeOBJ (Futatsugi y Diaconescu, 1998), ELAN (Borovanský et al., 2002) y Maude (Clavel et al., 2002a, 2004a), que permiten ejecutar teorías de reescritura. En esta tesis usaremos Maude: de las tres, la que más fielmente implementa la lógica. Maude comenzó a desarrollarse a mediados de los años noventa por un grupo de investigadores dirigidos por Meseguer en el Computer Science Laboratory de SRI International, en California, y la versión 1.0 se hizo pública en 1999. Desde entonces se ha seguido mejorando y en 2003 comenzó a distribuirse la versión 2.0 del sistema, una herramienta extremadamente eficiente que ejecuta millones de reescrituras por segundo y que viene acompañada de un comprobador de modelos que se puede aplicar directamente sobre teorías de reescritura.

1.4 Reflexión

La reflexión es una característica muy útil y poderosa de la lógica de reescritura, motivadora de trabajo sobre teoremas metalógicos reflexivos. Intuitivamente, la reflexión consiste en la capacidad de un sistema, que puede ser lógico o computacional, para razonar sobre sí mismo. Ejemplos clásicos donde se observa el uso de esta propiedad, aunque en un contexto distinto al que nos va a interesar a nosotros, serían la máquina universal de Turing y la codificación del lenguaje de la aritmética de primer orden en sí mismo por parte de Gödel.

Clavel y Meseguer han dado demostraciones de esta propiedad para fragmentos cada vez más generales de la lógica de reescritura, en concreto:

1. sin tipos y sin condiciones (Clavel, 1998, 2000),
2. sin tipos y con condiciones (Clavel y Meseguer, 2002); y
3. con múltiples tipos (Clavel y Meseguer, 2002).

En esta tesis se generalizan los tres resultados anteriores al caso más general en el que las especificaciones ecuacionales subyacentes son teorías en *lógica ecuacional de pertenencia* (Meseguer, 1998). Las reglas condicionales de este último caso son muy generales puesto que en ellas pueden intervenir no solo otras reescrituras, como en los casos (2) y (3) arriba indicados, sino también ecuaciones y afirmaciones de pertenencia como parte de la condición. Este trabajo también extiende y discute con mayor detalle algunos resultados iniciales sobre reflexión en la lógica de reescritura presentados en Palomino (2001b).

¿Qué ocurre con otras lógicas? ¿Qué ocurre, por ejemplo, con la propia lógica ecuacional de pertenencia? ¿O con la lógica ecuacional heterogénea (en inglés, many-sorted)? ¿O con la lógica de Horn con igualdad? Durante mucho tiempo, y reforzado por los resultados de Clavel y Meseguer, se ha conjeturado que estas lógicas también son reflexivas y que los mismos métodos utilizados para la lógica de reescritura se podrían utilizar para obtener teoremas de reflexión para ellas. Los resultados presentados en esta tesis establecen la certidumbre de estas conjeturas. Además, nuestras construcciones arrojan un poco de luz sobre la cuestión de cómo las teorías universales de lógicas relacionadas se relacionan entre sí. Por ejemplo, la lógica ecuacional de pertenencia es una sublógica de la lógica de reescritura y esta relación se expresa al nivel de reflexión mediante el hecho de que la teoría universal de la lógica ecuacional de pertenencia es a su vez una *subteoría* de la teoría universal para la versión más general de la lógica de reescritura que consideramos en esta tesis.

Por lo tanto, estos resultados hacen patente que la reflexión está disponible como una característica muy poderosa no solo para esta variante más general de la lógica de reescritura, en concreto la que está implementada en el sistema Maude, sino también para otras lógicas computacionales de gran importancia en los ámbitos de la especificación formal y la programación declarativa, como son la lógica ecuacional de pertenencia, la lógica ecuacional heterogénea y la lógica de Horn con igualdad. Todo lo cual puede servir de base para el diseño, con una fundamentación teórica clara, de *lenguajes de programación declarativos reflexivos* en esas lógicas.

El tener una especificación explícita de las correspondientes teorías universales es de gran importancia práctica para el razonamiento metalógico. En Basin et al. (2000, 2004) se investigan las buenas propiedades de la lógica ecuacional de pertenencia y de la lógica de reescritura como *marcos metalógicos reflexivos* que combinan inducción, parametrización y reflexión para soportar razonamiento metalógico sobre las lógicas representadas en ellos. En ese metarrazonamiento, como siempre que se requieren *objetos de prueba* para justificar pruebas mediante reflexión, resulta esencial hacer un uso explícito de las correspondientes teorías de reescritura. En particular, los resultados en esta tesis tienen importantes consecuencias para el lenguaje Maude, ya que sirven tanto como

fundamento teórico para su módulo `META-LEVEL` (véase la sección 2.4.3), que implementa la reflexión, como para proporcionar un método general para combinar computaciones reflexivas eficientes, utilizando la funcionalidad del módulo `META-LEVEL` con la habilidad de generar objetos de prueba por medio de las teorías universales cuando sea necesario.

Precisamente abordando el tema de los marcos metalógicos termina el capítulo dedicado a la reflexión. Aquí extendemos los principios de metarrazonamiento de Basin et al. (2004), relajando la restricción que allí tienen de que solo se pueden aplicar a teorías relacionadas por la relación de inclusión. Esta relajación incrementa considerablemente su aplicabilidad, lo que ilustramos enseñando cómo pueden utilizarse dichos principios para el estudio de algunas relaciones semánticas entre distintas especificaciones ecuacionales mediante su formalización como teoremas en la teoría universal de la lógica ecuacional de pertenencia, cuyas demostraciones siguen fielmente las correspondientes al metanivel.

1.5 Demostración de propiedades

En el razonamiento formal sobre sistemas concurrentes intervienen habitualmente dos niveles distintos de especificación: (1) un nivel de *especificación de sistema*, en el que se recoge una descripción computacional del mismo; y (2) un nivel de *especificación de propiedades*, en el que se especifican diversas propiedades de seguridad y vivacidad que el sistema debería satisfacer. Una especificación de sistema típicamente determina un *modelo matemático* (o conjunto de modelos) sobre los que se quiere verificar que ciertas propiedades se cumplen. Entre los modelos matemáticos usados con frecuencia están los sistemas de transiciones (en inglés, transition systems) y las estructuras de Kripke—sistemas de transiciones decorados con información sobre los predicados atómicos. Para las propiedades se pueden utilizar diferentes lógicas temporales y modales: CTL* (Clarke et al., 1999) es una elección muy interesante, pues contiene como casos especiales las muy utilizadas LTL y CTL.

La cuestión entonces es cómo determinar si el sistema concurrente objeto de estudio satisface las propiedades deseadas, para lo cual existe una gama de técnicas y herramientas con muy diversos grados de complejidad y que ofrecen distintos niveles de seguridad en los resultados que devuelven:

- Pruebas (testing). Se trata de ejecutar el sistema sobre unos cuantos valores para comprobar si al hacerlo se produce algún fallo, o desviación del comportamiento previsto.
- Comprobación de modelos. A diferencia de lo que ocurre con las pruebas, un comprobador de modelos explora sistemáticamente todas las posibilidades para comprobar si una propiedad se cumple con seguridad. Requiere que el sistema sea finito y da un método de decisión.
- Demostración de teoremas (theorem proving). Es la técnica más potente y consiste en demostrar teoremas matemáticos que garanticen la corrección del sistema.

La ejecución de pruebas es una técnica sencilla y rápida de aplicar, pero aunque puede detectar fallos no puede asegurar su ausencia y resulta insuficiente para verificar

componentes críticos de un sistema; además, es más difícil de aplicar cuando el sistema es concurrente debido al no determinismo y las muchas computaciones posibles. Bajo formas más o menos elaboradas, es la técnica más extendida en el mundo de la industria.

En el extremo opuesto del espectro se encuentra la demostración de teoremas. Esta técnica es la que más potencia y flexibilidad ofrece al usuario a la hora de probar propiedades de una especificación, pero a cambio exige un gran esfuerzo y participación por parte de este, y una formación adecuada. Para auxiliar en esta tarea se han desarrollado muchos demostradores de teoremas. Entre los más importantes de los disponibles en la actualidad podemos citar ACL2 (Kaufmann et al., 2000), que se ha utilizado en la demostración de la corrección de algunos de los algoritmos usados para las operaciones de los números en coma flotante en el procesador AMD Athlon (Russinoff, 2000), el demostrador PVS (Owre et al., 1998) que, al igual que Maude, se desarrolla en el Computer Science Laboratory de SRI International en California, o el francés Coq (Bertot y Castéran, 2004). A su vez, Maude dispone del más modesto ITP (Clavel et al., 2000; Clavel, 2004).

Un compromiso entre las dos técnicas anteriores la ofrece la comprobación de modelos, introducida por Clarke y Emerson (1981) y Quielle y Sifakis (1982), donde se pueden distinguir dos clases principales de algoritmos:

- algoritmos de estados explícitos (explicit-state), que exploran exhaustivamente el conjunto de estados de un sistema en busca de un contraejemplo, y
- algoritmos de comprobación de modelos simbólica (McMillan, 1993), en los que se utiliza una representación simbólica por medio de booleanos de los conjuntos de estados, y los estados alcanzables se computan como el punto fijo de la relación de transición.

La comprobación de modelos simbólica permite verificar sistemas con más estados, pero en cambio resulta mucho más sencillo utilizar un algoritmo de estados explícitos en el caso de que se manejen sistemas que, aunque con un número potencialmente infinito de estados, realmente tienen un conjunto finito y de cardinalidad moderada de estados alcanzables.

El comprobador de modelos de Maude (Eker et al., 2002) utiliza un algoritmo de estados explícitos. A toda fórmula temporal φ se le puede asociar un autómata (de Büchi) A_φ que reconoce aquellas secuencias de estados que la satisfacen. Así, para comprobar si una fórmula φ se satisface en un estado inicial, el comprobador primero construye el autómata $A_{\neg\varphi}$ asociado a la negación de la fórmula y comprueba si es vacía la intersección del lenguaje que acepta dicho autómata con los caminos que parten del estado inicial. La eficiencia con que este algoritmo está implementado en el comprobador de modelos de Maude es comparable a la de los comprobadores de modelos de estados explícitos más avanzados disponibles en la actualidad, como puede ser SPIN (Holzmann, 2003), ofreciendo la ventaja de que el lenguaje disponible para especificar los sistemas y propiedades (la lógica de reescritura) es mucho más expresivo (en el caso de SPIN se tiene un lenguaje llamado Promela, con sintaxis similar a la de C). Sin embargo, está limitado como todos los comprobadores de modelos a la verificación de propiedades de sistemas con un número finito de estados; para poder aplicarlo en sistemas infinitos es necesario abstraer antes estos en sistemas más sencillos.

1.6 Abstracción

A la hora de especificar formalmente los modelos matemáticos asociados a un sistema concurrente existen muchas posibilidades: en esta tesis los especificaremos por medio de *teorías de reescritura*. Esta elección resulta natural porque la lógica de reescritura ofrece un marco flexible para especificar un amplio espectro de sistemas concurrentes a alto nivel (Meseguer, 1992; Martí-Oliet y Meseguer, 2002b) sin por ello perder la posibilidad de ejecutarlos en lenguajes como Maude que permiten su simulación y la comprobación de modelos (Clavel et al., 2004a). Esencialmente, los estados de un sistema se especifican como los elementos de un álgebra inicial y las transiciones como reglas de reescritura. Resulta entonces muy fácil especificar ecuacionalmente, utilizando una teoría extendida, las proposiciones atómicas que se cumplen en los estados. De esta manera podemos asociar una estructura de Kripke a una teoría de reescritura y el problema de si el sistema satisface una determinada fórmula φ se convierte en la cuestión de verificar si esa estructura de Kripke satisface φ.

Sin embargo, puede resultar considerablemente más sencillo, o incluso necesario, verificar una relación de satisfacción tal utilizando la especificación de un sistema distinto. En particular, las técnicas de abstracción permiten reducir el problema de si un sistema con un número infinito de estados alcanzables, o con un número finito pero excesivamente elevado, satisface una propiedad expresada en una lógica temporal al uso de un comprobador de modelos sobre una versión abstracta del sistema con un número finito de estados. Entre la abundante literatura sobre abstracciones podemos mencionar los trabajos de Clarke et al. (1994) y Müller y Nipkow (1995) como algunos de los iniciales en este campo, Loiseaux et al. (1995) y Kesten y Pnueli (2000a,b) como trabajos más teóricos orientados al estudio de sus propiedades y posibilidades, o los artículos de Graf y Saïdi (1997), Colón y Uribe (1998) y Saïdi y Shankar (1999) para la técnica particular de abstracción de predicados. La forma más habitual de definir tales abstracciones es mediante un *cociente* del conjunto de estados del sistema original, junto con versiones abstractas de las transiciones y los predicados utilizados para expresar las propiedades. Métodos distintos difieren en los detalles particulares pero coinciden en el uso general de una función cociente. En este marco existe siempre un sistema (estructura de Kripke) *minimal* que convierte dicha función cociente en una simulación.

En el capítulo 4 de esta tesis presentamos un método sencillo para construir abstracciones cociente minimales de manera ecuacional. Este método asume que el sistema concurrente ha sido especificado mediante una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, siendo (Σ, E) una teoría ecuacional que especifica el conjunto de estados como un tipo algebraico de datos y R un conjunto de reglas de reescritura que especifica las transiciones del sistema. El método consiste simplemente en añadir más ecuaciones, digamos el conjunto E', para obtener un sistema cociente especificado por medio de la teoría de reescritura $\mathcal{R}/E' = (\Sigma, E \cup E', R)$; a tal sistema lo llamamos *abstracción ecuacional* de $\mathcal{R}$. Esta abstracción ecuacional resulta útil desde el punto de vista de la comprobación de modelos si:

1. $\mathcal{R}/E'$ es una teoría de reescritura *ejecutable* en un sentido apropiado; y

2. los predicados de estado son *preservados* por la simulación cociente.

Los requisitos (1) y (2) son *obligaciones de prueba* (proof obligations) que se pueden resolver utilizando métodos propios de los demostradores de teoremas.

Nuestro enfoque se puede mecanizar utilizando el lenguaje Maude (Clavel et al., 2002a) y su comprobador de modelos para lógica temporal lineal asociado (Eker et al., 2002), el demostrador de teoremas inductivo ITP (Clavel, 2004) y los comprobadores de las propiedades de Church-Rosser (Durán y Meseguer, 2000) y coherencia (Durán, 2000a).

1.7 Simulación

El concepto de simulación es esencialmente el mismo que el de abstracción, solo que entendido en un ámbito más general. El objetivo de las simulaciones no se circunscribe a calcular sistemas finitos que permitan el estudio de otros más complejos, sino que va más allá e incluye relaciones como la simulación de una implementación por parte de su especificación (ambas posiblemente sistemas infinitos) y que demuestra que la implementación es correcta, o la bisimulación entre teorías semánticamente equivalentes que permite trasladar los resultados sobre una a la otra.

El capítulo 5 de esta tesis trata de avanzar en dos direcciones principales: la primera busca generalizar la noción de simulación entre estructuras de Kripke tanto como sea posible (para lo que ya se dió un primer paso en el capítulo sobre abstracciones) y la segunda persigue resultados generales de representabilidad que demuestren que las estructuras de Kripke y las simulaciones generalizadas se pueden representar en la lógica de reescritura. Estos dos objetivos están motivados por *razones prácticas*. La razón en el primer caso es que las simulaciones resultan esenciales para el *razonamiento composicional*. Una piedra angular en tal razonamiento es el resultado que nos indica que las simulaciones *reflejan* propiedades de la lógica temporal, esto es, si tenemos una simulación de estructuras de Kripke $H : \mathcal{A} \longrightarrow \mathcal{B}$ y una fórmula adecuada en lógica temporal φ, si resulta que a y b están relacionados por H y $\mathcal{B}, b \models \varphi$ entonces podemos deducir que $\mathcal{A}, a \models \varphi$. Aunque este resultado es enormemente potente, resulta interesante generalizarlo aún más pues una noción más general de simulación le dará un mayor ámbito de aplicabilidad, aún cuando para ello la clase de las fórmulas φ a las que se aplique haya de ser restringida.

El avanzar en el segundo objetivo también está motivado por razones prácticas, en concreto:

1. ejecutabilidad,
2. facilidad de especificación, y
3. facilidad de prueba.

La clave sobre (1) y (2) es que la lógica de reescritura es un marco muy flexible, de manera que un sistema concurrente normalmente se puede especificar fácilmente y a alto nivel; además, tales especificaciones pueden ser utilizadas directamente para ejecutar el sistema o para razonar sobre él, lo que es precisamente el punto (3). Ciertamente, tanto la lógica de reescritura como su lógica ecuacional subyacente pueden resultar muy útiles a la hora de razonar formalmente, ya que uno necesita a menudo razonar más allá del

nivel proposicional. Por ejemplo, incluso cuando usemos un comprobador de modelos para demostrar que un sistema con infinitos estados alcanzables satisface $\mathcal{A}, a \models \varphi$, construyendo una *simulación de abstracción* con un número finito de estados $H : \mathcal{A} \longrightarrow \mathcal{B}$ y comprobando que $\mathcal{B}, b \models \varphi$ para algún b tal que aHb, todavía nos quedará pendiente la cuestión de verificar la *corrección* de H, lo que requiere resolver las obligaciones de prueba asociadas. De forma más general, cualquier razonamiento deductivo en lógica temporal necesita incluir razonamiento de primer orden, y a menudo inductivo, al nivel de los predicados de estado. Es precisamente aquí donde la lógica de reescritura y su sublógica ecuacional y sus modelos iniciales, que soportan razonamiento inductivo, resultan útiles. Los resultados que presentamos extienden considerablemente los del capítulo anterior.

Avanzamos en el primer objetivo generalizando las simulaciones en tres direcciones. Primero, consideramos *simulaciones tartamudas* en el sentido de Browne et al. (1988) y Manolios (2001), que son bastante generales y útiles para relacionar sistemas concurrentes con distintos niveles de atomicidad; en segundo lugar relajamos la condición de preservación de las proposiciones atómicas de exigir igualdad a solo pedir inclusión; y tercero, permitimos alfabetos diferentes AP y AP' de proposiciones atómicas en las estructuras de Kripke relacionadas.

Avanzamos en el segundo objetivo demostrando varios *resultados de representabilidad* que muestran que cualquier estructura de Kripke (resp. cualquier estructura de Kripke recursiva) puede ser representada mediante una teoría de reescritura (resp. una teoría de reescritura recursiva) y que cualquier simulación generalizada (resp. simulación generalizada recursivamente enumerable) puede ser representada mediante una relación definida mediante reescritura.

Pese a que el marco categórico resulta el más natural para entender estas simulaciones generalizadas, hasta donde sabemos el mismo no ha sido explotado de manera sistemática hasta la fecha. Nosotros a medida que vayamos presentando nuestras distintas nociones de simulación las iremos organizando en categorías, junto con las estructuras de Kripke correspondientes. Posteriormente en el capítulo 6 se llevará a cabo un estudio categórico más profundo de todas ellas, agrupándolas en instituciones y describiendo sus límites y colímites.

1.8 Organización de la tesis

En el capítulo 2 se hace un rápido repaso de las principales nociones que se utilizarán en el cuerpo de la tesis, como son por un lado la lógica de reescritura y su implementación en el sistema Maude, y por otro lado la lógica temporal y las estructuras de Kripke que se utilizarán como modelos suyos.

El capítulo 3 trata sobre reflexión. En una primera parte, se presentan las demostraciones detalladas de la reflexividad de la lógica ecuacional de pertenencia y de la lógica de reescritura sobre aquella que aparecían esbozadas en Clavel et al. (2002b). Estos resultados extienden los de Clavel y Meseguer (2002) a la versión más general de la lógica de reescritura, favorecen un enfoque más lógico que simplifica enormemente las pruebas y, por vez primera, arrojan un poco de luz sobre la relación que existe entre las teorías universales correspondientes a lógicas distintas emparentadas de algún modo. La segunda parte de

este capítulo, publicada en Clavel et al. (2004b) con ligeras diferencias en la notación, hace uso de la recién definida teoría universal de la lógica ecuacional de pertenencia y muestra cómo se puede utilizar para estudiar formalmente relaciones semánticas entre especificaciones ecuacionales.

Los capítulos 4 y 5 estudian, respectivamente, las ideas de abstracción y simulación en el marco de la lógica de reescritura. En el capítulo 4 se comienza extendiendo la noción habitual de simulación entre estructuras de Kripke, eliminando la necesidad de que un estado tenga que satisfacer exactamente las mismas proposiciones que aquel al que simula, y se introduce la técnica de las abstracciones ecuacionales (Meseguer et al., 2003) para computar abstracciones de sistemas representados en la lógica de reescritura sobre las que posteriormente se pueda aplicar un comprobador de modelos. En el capítulo 5 se generaliza aún más la noción de simulación, relacionando estructuras de Kripke con conjuntos distintos de proposiciones atómicas y permitiendo el tartamudeo. Para representar estas simulaciones al nivel de la lógica de reescritura se introducen generalizaciones sucesivas de la noción de abstracción ecuacional entre teorías de reescritura: los morfismos teoroidales (recogidos en Martí-Oliet et al., 2004), simulaciones definidas ecuacionalmente y simulaciones mediante relaciones definidas utilizando reescritura. Cada uno de estos conceptos es presentado junto con ejemplos y las condiciones que han de satisfacerse para que realmente den lugar a simulaciones entre los sistemas que representan.

El concepto de simulación no es ni mucho menos nuevo; los resultados contenidos en estos dos capítulos lo extienden en algunas direcciones y, fundamentalmente, lo adaptan al marco de la lógica de reescritura, indicando cómo aplicarlo en el mismo. Junto con la herramienta descrita en el capítulo 7, las ideas que aquí se presentan contribuyen a remediar en parte el aspecto más débil en la actualidad del proceso de especificación de sistemas en la lógica de reescritura: la demostración formal de propiedades de los sistemas especificados en ella.

La tesis termina con dos capítulos de carácter contrapuesto: uno eminentemente teórico y otro de clara aplicación práctica. El capítulo 6 presenta un estudio de las propiedades básicas de las categorías de estructuras de Kripke y simulaciones introducidas en capítulos anteriores, las clasifica en instituciones y demuestra algunos resultados que respaldan el carácter "minimal" atribuido con anterioridad a algunas estructuras de Kriple. Por último, en el capítulo 7 se aborda la construcción de un prototipo para aplicar la técnica de abstracción de predicados a teorías de reescritura. Su diseño reflexivo (soportado por Maude) posibilita que, aunque su rendimiento quede por el momento por debajo del de otras herramientas similares, su implementación resulte muy sencilla, como demuestra el código en el apéndice.

Capítulo 2

Preliminares

En este capítulo repasamos todas aquellas nociones que aparecen de manera repetida a lo largo de la tesis y cuyo conocimiento es requisito para poder entender el desarrollo de capítulos posteriores.

En esta categoría se encuadra obviamente la lógica de reescritura, que utilizaremos para la especificación de sistemas concurrentes basados en estados y cuya propiedad de *reflexión* es el objeto del capítulo siguiente. Junto con la lógica de reescritura también conviene repasar su sublógica ecuacional así como su implementación en el lenguaje Maude, cuya sintaxis se utilizará en las especificaciones de muchos de los ejemplos. Pero además, en los capítulos dedicados a abstracción y simulación necesitaremos considerar los sistemas concurrentes desde un punto de vista todavía menos concreto mediante el uso de estructuras matemáticas abstractas. Por estas últimas empezamos.

2.1 Sistemas de transiciones, estructuras de Kripke y lógica temporal

A la hora de razonar sobre sistemas computacionales resulta conveniente abstraer tantos detalles como sea posible mediante el uso de modelos matemáticos sencillos que se puedan utilizar para razonar sobre ellos. Para sistemas basados en estados se puede, en un primer paso, representar su comportamiento por medio de un sistema de transiciones.

Un *sistema de transiciones* (en inglés, transition system) es un par $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$, donde A es un conjunto de estados y $\rightarrow_{\mathcal{A}} \subseteq A \times A$ es una relación binaria llamada relación de transición.

Un sistema de transiciones, sin embargo, no incluye ningún tipo de información sobre las propiedades del sistema. Para poder razonar sobre dichas propiedades se necesita añadir información sobre las propiedades atómicas que se satisfacen en cada estado. Tales propiedades atómicas pueden ser descritas mediante un conjunto AP de proposiciones atómicas.

Una *estructura de Kripke* es una terna $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$, donde $(A, \rightarrow_{\mathcal{A}})$ es un sistema de transiciones con $\rightarrow_{\mathcal{A}}$ una relación *total* y $L_{\mathcal{A}} : A \rightarrow \mathcal{P}(AP)$ es una función de etiquetado que asocia a cada estado el conjunto de las propiedades atómicas que se satisfacen en él.

Usamos la notación $a \rightarrow_{\mathcal{A}} b$ para indicar que $(a, b) \in \rightarrow_{\mathcal{A}}$. Nótese que la relación de transición de una estructura de Kripke debe ser *total*, esto es, para cada $a \in A$ existe un $b \in A$ tal que $a \rightarrow_{\mathcal{A}} b$. Esta es una decisión habitual (Clarke et al., 1999, por ejemplo) que se toma para simplificar la definición de la semántica de las lógicas temporales, de las que las estructuras de Kripke son modelos. Dada una relación arbitraria $\rightarrow$, escribimos $\rightarrow^{\bullet}$ para referirnos a la relación total que extiende $\rightarrow$ añadiendo un par $a \rightarrow^{\bullet} a$ para cada a tal que no existe ningún b tal que $a \rightarrow b$. Un *camino* en $\mathcal{A}$ es una función $\pi : \mathbb{N} \longrightarrow A$ tal que, para cada $i \in \mathbb{N}$, se cumple que $\pi(i) \rightarrow_{\mathcal{A}} \pi(i+1)$. Usamos π^i para referirnos al sufijo de π que comienza en $\pi(i)$; explícitamente, $\pi^i(n) = \pi(i+n)$.

Para especificar propiedades de sistemas utilizaremos una lógica temporal. El origen de la lógica temporal se puede rastrear hasta Aristóteles, pero su uso para la especificación de propiedades que deben ser satisfechas por programas o sistemas computacionales se debe al trabajo seminal de Pnueli (1977). De entre la variedad de lógicas temporales nosotros utilizaremos la lógica ACTL*, o ACTL$^*(AP)$ si queremos hacer explícito el conjunto de proposiciones atómicas. Esta es una sublógica de CTL$^*(AP)$, una lógica temporal ramificada de tiempo discreto en la que el momento presente se corresponde con el estado actual del sistema y cada uno de los momentos futuros se corresponde con posibles evoluciones del estado (véase por ejemplo Clarke et al., 1999, sección 3.1). Los modelos sobre los que estas lógicas se interpretan son las estructuras de Kripke.

En CTL$^*(AP)$ existen dos tipos de fórmulas: fórmulas de estado (en inglés, state formulas), agrupadas en el conjunto State(AP), y fórmulas de camino (path formulas), que denotaremos con Path(AP). Su sintaxis viene dada por las siguientes definiciones mutuamente recursivas:

$$\begin{array}{ll} \text{fórmulas de estado:} & \varphi = p \in AP \mid \top \mid \bot \mid \neg\varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{A}\psi \mid \mathbf{E}\psi \\ \text{fórmulas de camino:} & \psi = \varphi \mid \neg\psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{R}\psi \mid \mathbf{G}\psi \mid \mathbf{F}\psi\,. \end{array}$$

A es el cuantificador universal y **E** el existencial, y los operadores **X**, **U**, **R**, **G** y **F** tienen el significado de *siguiente*, *hasta que*, *libera*, *siempre* y *alguna vez*, como queda de manifiesto en la definición de la semántica.

La semántica de la lógica, que especifica las relaciones de satisfacción $\mathcal{A}, a \models \varphi$ y $\mathcal{A}, \pi \models \psi$, para una estructura de Kripke $\mathcal{A}$, un estado inicial $a \in A$, una fórmula de estado φ, un camino π y una fórmula de camino ψ, se define por inducción estructural tal y como se muestra en la figura 2.1.

ACTL$^*(AP)$ es la restricción de CTL$^*(AP)$ a aquellas fórmulas cuya formal normal negativa (con todas las negaciones al nivel de los átomos) no contiene cuantificadores existenciales.

En ocasiones nos restringiremos a la sublógica LTL de ACTL*, la lógica temporal lineal (Manna y Pnueli, 1992, 1995), que es una lógica muy utilizada y que ofrece la ventaja de que sus fórmulas son fácilmente interpretables, además de ser la lógica que implementa el comprobador de modelos de Maude (véase la sección 2.4.4). En LTL todas las fórmulas son de la forma $\mathbf{A}\psi$, donde ψ no contiene ningún cuantificador. En estos casos utilizaremos una sintaxis especial en la que omitiremos la **A** y los operadores **X**, **G** y **F** se escribirán como $\bigcirc$, $\Box$ y $\Diamond$, respectivamente.

$$
\begin{array}{lcl}
\mathcal{A}, a \models p & \Longleftrightarrow & p \in L_{\mathcal{A}}(a) \\
\mathcal{A}, a \models \top & \Longleftrightarrow & \textit{verdadero} \\
\mathcal{A}, a \models \bot & \Longleftrightarrow & \textit{falso} \\
\mathcal{A}, a \models \neg\varphi & \Longleftrightarrow & \mathcal{A}, a \not\models \varphi \\
\mathcal{A}, a \models \varphi_1 \vee \varphi_2 & \Longleftrightarrow & \mathcal{A}, a \models \varphi_1 \text{ o } \mathcal{A}, a \models \varphi_2 \\
\mathcal{A}, a \models \varphi_1 \wedge \varphi_2 & \Longleftrightarrow & \mathcal{A}, a \models \varphi_1 \text{ y } \mathcal{A}, a \models \psi_2 \\
\mathcal{A}, a \models \mathbf{A}\psi & \Longleftrightarrow & \text{para todo } \pi \text{ tal que } \pi(0) = a \text{ se tiene } \mathcal{A}, \pi \models \psi \\
\mathcal{A}, a \models \mathbf{E}\psi & \Longleftrightarrow & \text{existe } \pi \text{ con } \pi(0) = a \text{ tal que } \mathcal{A}, \pi \models \psi \\
\\
\mathcal{A}, \pi \models \varphi & \Longleftrightarrow & \mathcal{A}, \pi(0) \models \varphi \\
\mathcal{A}, \pi \models \neg\psi & \Longleftrightarrow & \mathcal{A}, \pi \not\models \psi \\
\mathcal{A}, \pi \models \psi_1 \vee \psi_2 & \Longleftrightarrow & \mathcal{A}, \pi \models \psi_1 \text{ o } \mathcal{A}, \pi \models \psi_2 \\
\mathcal{A}, \pi \models \psi_1 \wedge \psi_2 & \Longleftrightarrow & \mathcal{A}, \pi \models \psi_1 \text{ y } \mathcal{A}, \pi \models \psi_2 \\
\mathcal{A}, \pi \models \mathbf{X}\psi & \Longleftrightarrow & \mathcal{A}, \pi^1 \models \psi \\
\mathcal{A}, \pi \models \psi_1 \mathbf{U} \psi_2 & \Longleftrightarrow & \text{existe } n \in \mathbb{N} \text{ tal que } \mathcal{A}, \pi^n \models \psi_2 \text{ y, para todo } m < n \\
 & & \text{se cumple que } \mathcal{A}, \pi^m \models \psi_1 \\
\mathcal{A}, \pi \models \psi_1 \mathbf{R} \psi_2 & \Longleftrightarrow & \text{para todo } n \in \mathbb{N} \text{ se tiene que, o bien } \mathcal{A}, \pi^n \models \psi_2 \\
 & & \text{o existe } m < n \text{ tal que } \mathcal{A}, \pi^m \models \psi_1 \\
\mathcal{A}, \pi \models \mathbf{G}\psi & \Longleftrightarrow & \text{para todo } n \in \mathbb{N} \text{ se tiene } \mathcal{A}, \pi^n \models \psi \\
\mathcal{A}, \pi \models \mathbf{F}\psi & \Longleftrightarrow & \text{existe } n \in \mathbb{N} \text{ tal que } \mathcal{A}, \pi^n \models \psi
\end{array}
$$

Figura 2.1: Semántica de la lógica CTL*.

2.2 La lógica ecuacional de pertenencia

La lógica ecuacional de pertenencia fue presentada por primera vez en Meseguer (1998). Se trata de una versión muy expresiva de la lógica ecuacional que no solo extiende la lógica ecuacional heterogénea (many-sorted, en inglés) sino también (algunas de las versiones de) la lógica ecuacional de tipos ordenados (order-sorted). En ocasiones nos referiremos a ella simplemente como lógica de pertenencia o como MEL, las siglas de su nombre en inglés: membership equational logic. Para un estudio detallado de la misma referimos al lector, además de a la publicación mencionada anteriormente, a Bouhoula et al. (2000).

Una *signatura* en la lógica de pertenencia es una terna (K, Σ, S) (simplemente Σ en lo que sigue), con K un conjunto de *familias* (en inglés, kinds), Σ una signatura indizada por K, $\Sigma = \{\Sigma_{w,k}\}_{(w,k)\in K^* \times K}$ y $S = \{S_k\}_{k\in K}$ una colección de conjuntos, disjuntos dos a dos, indizados por K. A S_k se le llama el conjunto de *tipos* de la familia k. El par (K, Σ) es lo que habitualmente se llama signatura heterogénea de símbolos de funciones; sin embargo, nosotros llamamos familias a los elementos de K porque cada familia k tiene ahora un conjunto S_k de tipos asociados, que en los modelos se interpretan como subconjuntos del conjunto soporte de la familia. La familia de un tipo s se denota con $[s]$. Como es habitual (Ehrig y Mahr, 1985), escribimos $T_{\Sigma,k}$ y $T_{\Sigma,k}(X)$ para denotar, respectivamente, los conjuntos de Σ-términos cerrados (sin variables) de familia k y Σ-términos de familia k sobre las variables X, donde $X = \{x_1 : k_1, \ldots, x_n : k_n\}$ es un conjunto de variables, cada

una asociada a una familia en K. En ocasiones se utilizará la notación $t(x_1, \ldots, x_n)$ o $t(\vec{x})$ para hacer explícito el conjunto de variables que aparecen en un término t.

Las fórmulas atómicas de la lógica de pertenencia son o bien *ecuaciones* $t = t'$, donde t y t' son Σ-términos de la misma familia, o *afirmaciones de pertenencia* de la forma $t : s$, donde el término t pertenece a una familia k y s pertenece a S_k. Las sentencias son cláusulas de Horn sobre estas fórmulas atómicas, esto es, sentencias de la forma

$$(\forall X)\, A_0 \textbf{ if } A_1 \wedge \ldots \wedge A_n\,,$$

donde cada A_i es una ecuación o una afirmación de pertenencia y donde X es un conjunto de variables con familias en K que contiene todas las variables en los A_i. Según el contexto puede que también nos refiramos a estas sentencias como axiomas o simplemente fórmulas.

En la lógica de pertenencia, las relaciones de subtipado y sobrecarga de operadores no son más que una manera conveniente de escribir las correspondientes cláusulas de Horn. Por ejemplo, suponiendo que *Natural* y *Entero* son tipos de la misma familia y que se tiene un operador $\_ + \_ : [\mathit{Entero}]\ [\mathit{Entero}] \longrightarrow [\mathit{Entero}]$ en la signatura, la declaración $\mathit{Natural} < \mathit{Entero}$ es simplemente una notación para la afirmación de pertenencia $(\forall x : [\mathit{Entero}])\, x : \mathit{Entero}$ **if** $x : \mathit{Natural}$, y las declaraciones de operadores sobrecargados

$$\_ + \_ : \mathit{Natural}\ \mathit{Natural} \longrightarrow \mathit{Natural} \qquad \_ + \_ : \mathit{Entero}\ \mathit{Entero} \longrightarrow \mathit{Entero}$$

son lógicamente equivalentes a

$$(\forall x : [\mathit{Entero}], y : [\mathit{Entero}])\, x + y : \mathit{Natural} \textbf{ if } x : \mathit{Natural} \wedge y : \mathit{Natural}$$
$$(\forall x : [\mathit{Entero}], y : [\mathit{Entero}])\, x + y : \mathit{Entero} \textbf{ if } x : \mathit{Entero} \wedge y : \mathit{Entero}\,.$$

Una *teoría* en MEL es un par (Σ, E), donde E es un conjunto de sentencias en lógica ecuacional de pertenencia sobre la signatura Σ. Escribiremos $(\Sigma, E) \vdash \phi$, o simplemente $E \vdash \phi$ si la signatura Σ está clara por el contexto, para denotar que la sentencia ϕ es consecuencia lógica de (Σ, E) en el sistema de prueba de Meseguer (1998) que se incluye al final de esta sección. Intuitivamente, los términos de una familia correctos o con buen comportamiento son aquellos para los que se puede demostrar que tienen tipo, mientras que los términos de error, o no definidos, pertenecen a una familia pero no tienen un tipo asociado. Por ejemplo, si asumimos operadores de resta $\_ - \_$ y división $\_/\_$ con declaraciones apropiadas, se tendría que $3 + 2 : \mathit{Natural}$ y $3 - 4 : \mathit{Entero}$, pero $7/0$ sería un término de la familia $[\mathit{Racional}]$ sin tipo.

Una Σ-*álgebra* A consiste en un conjunto A_k para cada familia $k \in K$, una función $A_f : A_{k_1} \times \ldots \times A_{k_n} \longrightarrow A_k$ para cada operador $f \in \Sigma_{k_1 \ldots k_n, k}$ y un subconjunto $A_s \subseteq A_k$ para cada tipo $s \in S_k$. Un álgebra A y una valoración σ que asigne a cada variable $x : k$ en X un valor en A_k satisfacen una ecuación $(\forall X)\, t = t'$ si y solo si $\sigma(t) = \sigma(t')$, donde usamos la misma notación σ para la valoración y su extensión homomórfica a términos. Escribimos $A, \sigma \models (\forall X)\, t = t'$ para denotar esta relación. De manera análoga, se cumple que $A, \sigma \models (\forall X)\, t : s$ si y solo si $\sigma(t) \in A_s$. Cuando la fórmula ϕ es satisfecha por todas las valoraciones σ, escribimos $A \models \phi$ y decimos que A es un modelo de ϕ. Como es habitual, también escribimos $(\Sigma, E) \models \phi$ cuando todos los modelos del conjunto de sentencias E satisfacen ϕ.

Una teoría (Σ, E) en la lógica de pertenencia tiene un modelo inicial $T_{\Sigma/E}$ cuyos elementos son clases de E-equivalencia $[t]$ de términos cerrados. En el modelo inicial, los tipos se interpretan como los conjuntos más pequeños que satisfacen los axiomas de la teoría y la igualdad se interpreta como la menor congruencia que satisface dichos axiomas. Escribimos $E \vdash_{ind} \phi$ o $E \models \phi$ cuando ϕ se satisfaga en el modelo inicial de E (Meseguer, 1998).

Un cálculo de pruebas para MEL

Dada una teoría $T = (\Sigma, E)$ en la lógica de pertenencia, decimos que una sentencia ϕ es consecuencia lógica de T si y solo si $T \vdash \phi$ se puede obtener a través de la aplicación un número finito de veces de las siguientes *reglas de deducción* . En ocasiones, si T queda clara por el contexto simplemente diremos que existe una derivación de ϕ.

1. **Reflexividad**. Para todo $t \in T_\Sigma(X)$,

$$\frac{}{T \vdash (\forall X)\, t = t}\,.$$

2. **Simetría**. Para todo $t, t' \in T_\Sigma(X)$,

$$\frac{T \vdash (\forall X)\, t = t'}{T \vdash (\forall X)\, t' = t}\,.$$

3. **Transitividad**. Para todo $t, t', t'' \in T_\Sigma(X)$,

$$\frac{T \vdash (\forall X)\, t = t'' \qquad T \vdash (\forall X)\, t'' = t'}{T \vdash (\forall X)\, t = t'}\,.$$

4. **Congruencia**. Para todo $f : k_1 \dots k_n \longrightarrow k$ en Σ y términos $t_1, \dots, t_n$, $t'_1, \dots, t'_n$ con $t_i, t'_i \in T_{\Sigma,k_i}(X)$,

$$\frac{T \vdash (\forall X)\, t_1 = t'_1 \quad \dots \quad T \vdash (\forall X)\, t_n = t'_n}{T \vdash (\forall X)\, f(t_1, \dots, t_n) = f(t'_1, \dots, t'_n)}\,.$$

5. **Reemplazamiento**. Para todo conjunto de variables Y, sustitución $\sigma : X \longrightarrow T_\Sigma(Y)$ y axioma $(\forall X)\, A_0$ **if** $A_1 \wedge \dots \wedge A_n$ en E,

$$\frac{T \vdash (\forall Y)\, \sigma(A_1) \quad \dots \quad T \vdash (\forall Y)\, \sigma(A_n)}{T \vdash (\forall Y)\, \sigma(A_0)}\,.$$

6. **Pertenencia**. Para todo $t, u \in T_{\Sigma,k}(X)$ y $s \in S_k$,

$$\frac{T \vdash (\forall X)\, t = u \qquad T \vdash (\forall X)\, u : s}{T \vdash (\forall X)\, t : s}\,.$$

7. **Introducción de implicación**. Para cada sentencia $(\forall X)\, A_0$ **if** $A_1 \wedge \ldots \wedge A_n$ sobre la signatura de T, donde cada A_i es una ecuación o una afirmación de pertenencia,

$$\frac{(\Sigma(X), E \cup \{A_1, \ldots, A_n\}) \vdash (\forall \emptyset)\, A_0}{(\Sigma, E) \vdash (\forall X)\, A_0 \textbf{ if } A_1 \wedge \ldots \wedge A_n},$$

donde $\Sigma(X)$ es la signatura Σ extendida con los elementos de X como constantes nuevas adicionales.

La demostración de la corrección y completitud de este cálculo con respecto a la noción de satisfacción de sentencias por Σ-álgebras se puede encontrar en Meseguer (1998).

2.3 La lógica de reescritura

Los sistemas concurrentes se axiomatizan en la lógica de reescritura (en ocasiones RL, de REWRITING LOGIC) por medio de *teorías de reescritura* (Meseguer, 1992) de la forma $\mathcal{R} = (\Sigma, E, R)$, donde (Σ, E) es una teoría ecuacional. La lógica de reescritura está parametrizada por una lógica ecuacional subyacente que nosotros vamos a instanciar con la lógica de pertenencia. El conjunto de estados del sistema queda entonces descrito por la teoría (Σ, E) como el tipo de datos algebraicos $T_{\Sigma/E,k}$ asociado con el álgebra inicial $T_{\Sigma/E}$ de (Σ, E) tras la elección de una familia k de estados en Σ. Las *transiciones* del sistema quedan axiomatizadas por las *reglas de reescritura condicionales* R, que tienen la forma

$$\lambda : (\forall X)\, t \longrightarrow t' \textbf{ if } \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j \wedge \bigwedge_{l \in L} t_l \longrightarrow t'_l\,,$$

donde λ es una etiqueta, $p_i = q_i$ y $w_j : s_j$ son fórmulas atómicas en la lógica de pertenencia para cada $i \in I$ y $j \in J$, y para familias apropiadas k y k_l, $t, t' \in T_{\Sigma,k}(X)$ y $t_l, t'_l \in T_{\Sigma,k_l}(X)$ para $l \in L$.

Esta descripción difiere ligeramente de la original en Meseguer (1992) en que en aquella las teorías de reescritura venían dadas por 4-tuplas, con una componente adicional para el conjunto de las etiquetas. A su vez, en la generalización presentada en Bruni y Meseguer (2003) este conjunto no aparece explícitamente y en su lugar se introduce una función que define operadores *congelados*. En esta tesis, puesto que no usaremos los operadores congelados y las etiquetas desempeñan un papel menor, hemos decidido omitir la presentación de los primeros e incluir las etiquetas como parte constituyente de las reglas.

Un cálculo de pruebas para RL

La lógica de reescritura tiene reglas de deducción para inferir todas las posibles computaciones concurrentes en un sistema, en el sentido de que, dados dos estados $[u], [v] \in T_{\Sigma/E,k}$ podemos alcanzar $[v]$ desde $[u]$ mediante alguna, posiblemente compleja, computación concurrente si y solo si podemos demostrar que $u \longrightarrow v$ en la lógica; esta demostrabilidad se denota mediante $\mathcal{R} \vdash u \longrightarrow v$. En particular podemos definir fácilmente la *relación de $\mathcal{R}$-reescritura en un paso*, que es una relación binaria $\rightarrow^1_{\mathcal{R},k}$ sobre $T_{\Sigma,k}$

que se cumple entre dos términos $u, v \in T_{\Sigma,k}$ si y solo si existe una demostración en un solo paso del secuente $u \longrightarrow v$, esto es, si existe una prueba en la que tan solo una regla de reescritura ha sido aplicada a un único subtérmino (la definición precisa aparece más abajo).

Nuestra formulación de las reglas de deducción para la lógica de reescritura sigue el esquema utilizado en Bruni y Meseguer (2003), que generaliza la original de Meseguer (1992) mediante el uso de una regla explícita de E-igualdad para secuentes de reescritura $t \longrightarrow t'$ en lugar de absorberla en secuentes de la forma $[t] \longrightarrow [t']$ entre clases de equivalencia.

Dada una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, decimos que un secuente $t \longrightarrow t'$ es consecuencia lógica de $\mathcal{R}$, y escribimos $\mathcal{R} \vdash t \longrightarrow t'$, si y solo si $t \longrightarrow t'$ se puede obtener mediante la aplicación un número finito de veces de las siguientes *reglas de deducción*:

1. **Reflexividad**. Para todo $t \in T_\Sigma(X)$,

$$\frac{}{(\forall X)\, t \longrightarrow t}\,.$$

2. **Transitividad**. Para todo $t, t', t'' \in T_\Sigma(X)$,

$$\frac{(\forall X)\, t \longrightarrow t' \qquad (\forall X)\, t' \longrightarrow t''}{(\forall X)\, t \longrightarrow t''}\,.$$

3. **Congruencia**. Para todo $f : k_1 \dots k_n \longrightarrow k$ en Σ y términos $t_1, \dots, t_n$, $t'_1, \dots, t'_n$ con $t_i, t'_i \in T_{\Sigma,k_i}(X)$,

$$\frac{(\forall X)\, t_1 \longrightarrow t'_1 \quad \dots \quad (\forall X)\, t_n \longrightarrow t'_n}{(\forall X)\, f(t_1, \dots, t_n) \longrightarrow f(t'_1, \dots, t'_n)}\,.$$

4. **Reemplazamiento**. Para todo conjunto de variables Y, sustitución $\sigma : X \longrightarrow T_\Sigma(Y)$ y regla de reescritura $(\forall X)\, \lambda : t \longrightarrow t'$ **if** $\bigwedge_{i\in I} p_i = q_i \wedge \bigwedge_{j\in J} w_j : s_j \wedge \bigwedge_{l\in L} t_l \longrightarrow t'_l$ en R,

$$\frac{\begin{array}{c}(\Sigma, E) \vdash (\forall Y)\, \sigma(p_i) = \sigma(q_i) \;\; i \in I \qquad (\Sigma, E) \vdash (\forall Y)\, \sigma(w_j) : s_j \;\; j \in J \\ (\forall Y)\, \sigma(t_l) \longrightarrow \sigma(t'_l) \;\; l \in L\end{array}}{(\forall Y)\, \sigma(t) \longrightarrow \sigma(t')}\,.$$

5. **Igualdad**. Para todo $t, t', u, u' \in T_\Sigma(X)$,

$$\frac{(\Sigma, E) \vdash (\forall X)\, t = u \qquad (\Sigma, E) \vdash (\forall X)\, t' = u' \qquad (\forall X)\, t \longrightarrow t'}{(\forall X)\, u \longrightarrow u'}\,.$$

A diferencia de lo que ocurría en el cálculo de pruebas para MEL, aquí la teoría de reescritura no varía a lo largo de las reglas de derivación por lo que, para simplificar la notación, no se ha hecho explícita.

Ahora podemos definir de forma precisa la relación de $\mathcal{R}$-reescritura en un paso. Se tiene que $u \to^1_{\mathcal{R},k} v$ si:

- existe una derivación de $u \longrightarrow v$ con **(Reemplazamiento)** como última regla de deducción (las reescrituras que puedan aparecer en la condición no se contabilizan para el número de pasos), o
- existe una derivación de $u \longrightarrow v$ cuya última regla de deducción es **(Igualdad)** aplicada a un par de términos que ya se encontraban en la relación $\rightarrow^1_{\mathcal{R},k'}$, o bien si
- para algún $f \in \Sigma_{k_1 \ldots k_n,k}$, se tiene que $u = f(t_1, \ldots, t_n)$ y $v = f(t'_1, \ldots, t'_n)$ y existe i tal que $t_i \rightarrow^1_{\mathcal{R},k_i} t'_i$ y t_j es t'_j para todo $j \neq i$.

Esta relación da lugar a otra (con el mismo nombre) $\rightarrow^1_{\mathcal{R},k}$ sobre clases de equivalencia de términos en $T_{\Sigma/E,k}$ definiendo $[u] \rightarrow^1_{\mathcal{R},k} [v]$ si y solo si $u' \rightarrow^1_{\mathcal{R},k} v'$ para algún $u' \in [u]$ y $v' \in [v]$. Esto define un sistema de transiciones $\mathcal{T}(\mathcal{R})_k = (T_{\Sigma/E,k}, (\rightarrow^1_{\mathcal{R},k})^\bullet)$, asociado a la teoría de reescritura $\mathcal{R}$, para cada familia $k \in K$.

Antes de terminar con este apartado hay que señalar que la lógica de reescritura también tiene una teoría de modelos, los cuales vienen dados por familias de categorías y funtores, con una correspondiente noción de satisfacción con respecto a la cual el cálculo de pruebas es correcto y completo. La completitud se prueba mediante la construcción de modelos libres. Para más detalles referimos al lector a Meseguer (1992) y Meseguer (1990b); este último informe contiene ideas y construcciones que hasta el momento no han sido publicadas en ningún otro lugar. Para la completitud de la lógica de reescritura en el caso en que la lógica ecuacional subyacente es MEL, véase también Bruni y Meseguer (2003).

Ejecutabilidad de las teorías de reescritura

Bajo supuestos razonables sobre E y R, las teorías de reescritura se pueden *ejecutar*. De hecho, existen varias implementaciones del lenguaje de la lógica de reescritura, entre las que destacan CafeOBJ (Futatsugi y Diaconescu, 1998), ELAN (Borovanský et al., 2002) y Maude (Clavel et al., 1996, 2002a, 2004a).

La cuestión de cuándo una teoría de reescritura $\mathcal{R}$ es ejecutable está estrechamenente relacionada con el deseo de que tanto $T_{\Sigma/E,k}$ como $(\rightarrow^1_{\mathcal{R},k})^\bullet$ sean *computables*. Decimos que $\mathcal{R} = (\Sigma, E \cup A, R)$ es *ejecutable* si:

1. existe un *algoritmo de ajuste de patrones módulo* los axiomas ecuacionales A[1];
2. la teoría ecuacional $(\Sigma, E \cup A)$ es *Church-Rosser* y *terminante módulo* A (sobre términos cerrados) (Dershowitz y Jouannaud, 1990); y
3. las reglas R son *coherentes* (sobre términos cerrados) en el sentido de Viry (2002) en relación con las ecuaciones E módulo A.

[1]Como se verá en la sección 2.4, en el lenguaje Maude los axiomas A para los que el motor de reescritura soporta ajuste de patrones módulo A son los axiomas de *asociatividad*, *conmutatividad*, *identidad* e *idempotencia* para diferentes operadores binarios.

Las condiciones (1) y (2) garantizan que $T_{\Sigma/E\cup A,k}$ sea un conjunto computable, puesto que cada término cerrado t puede ser simplificado aplicando las ecuaciones en E de izquierda a derecha módulo A hasta alcanzar una *forma canónica* $can_{E/A}(t)$ que es única módulo los axiomas A. El problema de decidir la igualdad $[u]_{E\cup A} = [v]_{E\cup A}$ se puede reducir entonces al problema, decidible por (1), de ver si $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$. La condición (3) sobre la coherencia entre ecuaciones y reglas significa que para cada término cerrado t, siempre que tengamos $t \to^1_{\mathcal{R}} u$ debemos ser capaces de encontrar $can_{E/A}(t) \to^1_{\mathcal{R}} v$ tal que $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$.

Estas tres condiciones implican que $(\to^1_{\mathcal{R},k})^\bullet$ es una relación binaria computable sobre $T_{\Sigma/E\cup A,k}$, ya que podemos decidir si $[t]_{E\cup A} \to^1_{\mathcal{R}} [u]_{E\cup A}$ mediante la enumeración del conjunto finito de todas las $\mathcal{R}$-reescrituras en un paso módulo A del término $can_{E/A}(t)$, y para cualquiera de estas reescrituras, digamos v, podemos decidir si se tiene $[can_{E/A}(u)]_A = [can_{E/A}(v)]_A$.

La coherencia se puede comprobar mediante técnicas de pares críticos similares a las utilizadas para comprobar la confluencia de sistemas de reescritura y para realizar la complección de Knuth-Bendix; la teoría general se encuentra desarrollada en Viry (2002). Intuitivamente, la idea consiste en establecer primero que E es Church-Rosser y terminante módulo A y a continuación comprobar la coherencia de los "pares críticos relativos" (esto es, los solapamientos de subtérminos obtenidos mediante unificación que no sean variables) entre las ecuaciones en E y las reglas en R módulo los axiomas de A; en la sección 4.6 se presentan algunos ejemplos.

2.4 Maude

Como se apuntó en la sección anterior, Maude es un lenguaje de especificación y programación de alto nivel que implementa la lógica de reescritura, que comenzó a desarrollarse a mediados de la década de los noventa (Clavel et al., 1996, 1998, 1999). Maude ofrece soporte tanto para programación funcional sobre lógica ecuacional de pertenencia, por medio de *módulos funcionales*, como para la especificación de sistemas concurrentes y posiblemente no deterministas en lógica de reescritura, por medio de *módulos de sistema*. En la actualidad, Maude es un sistema estable que ha alcanzado la madurez con la versión 2.0 (Clavel et al., 2002a, 2004a).

Maude está inspirado en la familia de lenguajes OBJ (Goguen et al., 2000) y en particular en OBJ3 (Goguen et al., 1988), que puede considerarse en gran medida como un sublenguaje de la parte funcional de Maude, aunque la lógica de pertenencia que implementa Maude es más expresiva que la lógica de tipos ordenados de OBJ3.

Una prueba de la potencia y expresividad del lenguaje es Full Maude, desarrollado por Durán (1999), que es una especificación ejecutable que extiende Maude con operaciones de módulos, escrita en el propio Maude. Entre otras características ofrece soporte para *módulos orientados a objetos*, módulos parametrizados y expresiones de módulos. En esta tesis no se hace uso de Full Maude, por lo que nos limitamos a referir al lector a Clavel et al. (2004a) para más detalles.

2.4.1 Módulos funcionales

Los módulos funcionales de Maude permiten ejecutar teorías $(\Sigma, E \cup A)$ de la lógica ecuacional de pertenencia. Desde un punto de vista operacional se distingue entre A, que es un conjunto de axiomas ecuacionales para algunos de los operadores de la signatura, y E, que es un conjunto de ecuaciones que se considerarán siempre módulo A. Para que una teoría tal sea ejecutable es necesario que E sea Church-Rosser y terminante para términos cerrados módulo los axiomas en A. Además, las ecuaciones deben ser *admisibles*, esto es, cualquier variable extra que no esté en $vars(t)$ solo puede ser introducida incrementalmente mediante "ecuaciones de ajuste" en la condición de manera que todas las variables queden instanciadas mediante ajuste de patrones.

La sintaxis de los módulos funcionales queda ilustrada en el siguiente ejemplo, en el que se declaran números naturales (con nombre `MyNat` para distinguirlos de los naturales predefinidos en Maude) y listas sobre ellos.

```
fmod NAT-LIST is
  sorts MyNat List .
  subsort MyNat < List .

  op zero : -> MyNat [ctor] .
  op s : MyNat -> MyNat [ctor] .
  op _+'_ : MyNat MyNat -> MyNat .

  op nil : -> List [ctor] .
  op __ : List List -> List [ctor assoc id: nil] .

  vars N M : MyNat .

  eq N +' zero = N .
  eq N +' s(M) = s(N +' M) .
endfm
```

Una vez introducido en el sistema, el módulo se puede utilizar para reducir cualquier término utilizando sus ecuaciones por medio del comando `reduce`.

```
Maude> reduce s(zero) +' s(zero) .
result MyNat: s(s(zero))
```

Los tipos se declaran con la palabra reservada `sort`; no está permitido declarar familias explícitamente, pero una vez declarado un tipo `s` se puede utilizar la notación `[s]` para referirse a la familia correspondiente, cuyos tipos son todos los pertenecientes a la misma componente conexa del tipo `s` generada por la relación de inclusión de subtipos. Las relaciones de subtipado, que como se comentó en la sección 2.2 no son más que una notación para ciertos axiomas de pertenencia, se introducen con `subsort`; nótese que las familias de los tipos que se relacionan quedan de esta manera identificadas.

Los símbolos de función, u operadores, se declaran utilizando `op` seguido del nombre de la operación y los tipos de sus argumentos y resultado. Aquí es importante señalar algunas cuestiones:

1. Maude permite el uso de una sintaxis muy flexible, que puede ser no solo prefija sino también infija e incluso vacía. En estos dos últimos casos, los símbolos de subrayado _ indican la posición de los argumentos y deben estar en correspondencia con la aridad del operador.

2. Los operadores se pueden declarar junto con los atributos ecuacionales, llamados *A* más arriba. En el ejemplo, el operador de concatenación de listas se ha declarado asociativo (`assoc`) y con `nil` como elemento identidad (`id:`). Además de estos, existen atributos de conmutatividad (`comm`), de idempotencia (`idem`) e identidad por la izquierda y la derecha (`left id:`, `right id:`).

3. Además de los atributos ecuacionales se pueden declarar otros. En el ejemplo aparece `ctor`, que se utiliza para señalar aquellos operadores que funcionan como *constructores* de un determinado tipo (de manera que todo término cerrado de ese tipo es igual a uno con constructores). No tiene ningún uso desde un punto de vista operacional, pero en la sección 2.5 veremos que el ITP los necesita para aplicar inducción estructural y en cualquier caso resulta una buena práctica metodológica el distinguir los constructores.

Las variables se pueden declarar precediéndolas con la palabra clave `var` o se pueden utilizar directamente, simplemente escribiendo tras su nombre dos puntos y su tipo (por ejemplo, `N:MyNat`). Los axiomas ecuacionales son entonces igualdades de términos construidos con las variables y los operadores, introducidos con `eq`, o `ceq` en el caso de axiomas condicionales. Los axiomas de pertenencia se escriben de modo análogo pero utilizando `mb` y `cmb`.

Un módulo funcional puede importar otros definidos con anterioridad de tres maneras distintas, utilizando las palabras reservadas `protecting`, `extending` o `including` (o sus abreviaciones `pr`, `ex` y `inc`) seguidas del nombre del módulo a importar. Cada una de ellas impone restricciones semánticas diferentes sobre el módulo que se importa. Informalmente, `protecting` exige que los tipos presentes en el módulo importado no se alteren con la inclusión de nuevos elementos ni con la identificación de elementos que anteriormente eran distintos; `extending` permite la creación de nuevos elementos en un tipo, pero sigue sin permitir la identificación de elementos distintos ya existentes; `including` es la más liberal de las tres y no impone ningún tipo de restricciones. Las restricciones se pueden encontrar desglosadas con precisión en Clavel et al. (2004a). El sistema, sin embargo, no comprueba si estas restricciones realmente se satisfacen, por lo que desde un punto de vista estrictamente operacional no existe ninguna diferencia entre ellas.

2.4.2 Módulos de sistema

Los módulos de sistema son una extensión de los módulos funcionales que además permiten la declaración de reglas de reescritura. El conjunto de las ecuaciones en un módulo de sistema tienen que seguir siendo Church-Rosser y terminante, pero estos requisitos no se exigen de las reglas, a las que tan solo se pide que sean admisibles.

Su sintaxis es la de los módulos funcionales excepto por su cabecera y cierre, que utilizan las palabras reservadas `mod` y `endm`, y a las que se añade `rl` y `crl` para introducir reglas de reescritura. Por ejemplo, el siguiente módulo extiende `NAT-LIST` con una regla de reescritura que, aplicada repetidamente a una lista, devuelve la suma de todos sus elementos.

```
mod NAT-LIST-SUM is
  protecting NAT-LIST .

  rl N1:MyNat N2:MyNat L:List => (N1:MyNat +' N2:MyNat) L:List .
endm
```

Para reescribir un término en un módulo de sistema utilizando las reglas junto con las ecuaciones se utiliza la orden `rewrite`:

```
Maude> rewrite s(zero) zero s(s(zero)) .
result MyNat: s(s(s(zero)))
```

En este caso solo hay un posible resultado, pero en general habrá más (o ninguno si ninguna cadena de reescrituras termina), a pesar de lo cual `rewrite` tan solo devolverá un resultado, en caso de terminación.

Un módulo de sistema puede importar tanto módulos funcionales como de sistema por medio de las palabras clave `protecting`, `extending` e `including`. Una introducción mediante juegos a la programación basada en reglas utilizando los módulos de sistema en Maude se puede encontrar en Palomino et al. (2004).

2.4.3 El metanivel

La implementación de la capacidad reflexiva de la lógica de reescritura se implementó en Maude paralelamente a los estudios iniciales sobre la misma. Esta implementación, de hecho, fue más allá del desarrollo teórico al ofrecer soporte para lógicas, como eran la lógica de pertenencia y la lógica de reescritura sobre ella, para las que no existían demostraciones sobre su capacidad reflexiva.

En Maude 2.0 la reflexión está soportada a través del módulo funcional `META-LEVEL`; los términos se metarrepresentan como elementos del tipo `Term`, y los módulos funcionales y de sistema como elementos de los tipos `FModule` y `Module`, respectivamente. Para metarrepresentar las constantes y las variables se utilizan elementos del tipo `Qid`, identificadores precedidos de un apóstrofo: las constantes contienen el nombre y el tipo separados por un `.` mientras que en las variables se separa el nombre del tipo con `:`. Los restantes términos se metarrepresentan con los siguientes operadores:

```
  op _,_ : TermList TermList -> TermList [ctor assoc] .
  op _[_] : Qid TermList -> Term [ctor] .
```

Por ejemplo, el término `N:MyNat +' zero` de tipo `MyNat` se metarrepresenta mediante el término `'_+'_['N:MyNat,'zero.MyNat]` de tipo `Term`.

La sintaxis para metarrepresentar módulos es prácticamente la misma que la del nivel objeto, utilizando la metarrepresentación de los términos; para su exposición detallada nos referimos al manual de Maude (Clavel et al., 2004a) y aquí nos limitamos a ilustrarla con el módulo `NAT-LIST`.

```
fmod 'NAT-LIST is
  including 'BOOL .
  sorts 'List ; 'MyNat .
  subsort 'MyNat < 'List .
  op '_+'_ : 'MyNat 'MyNat -> 'MyNat [none] .
  op '__ : 'List 'List -> 'List [assoc ctor id('nil.List)] .
  op 'nil : nil -> 'List [ctor] .
  op 's : 'MyNat -> 'MyNat [ctor] .
  op 'zero : nil -> 'MyNat [ctor] .
  none
  eq '_+'_['N:MyNat,'zero.MyNat] = 'N:MyNat [none] .
  eq '_+'_['N:MyNat,'s['M:MyNat]] = 's['_+'_['N:MyNat,'M:MyNat]] [none] .
endfm
```

Las reglas se representan de manera análoga a las ecuaciones. Nótese que, a diferencia de la sintaxis del nivel objeto, los elementos en las distintas categorías (`sorts`, `op`, ...) tienen que aparecer agrupados y su ausencia explícitamente señalada con `none` (o `nil`).

El módulo `META-LEVEL` contiene operaciones predefinidas para realizar de manera eficiente computaciones al metanivel con módulos y términos, reduciéndolas a las correspondientes operaciones al nivel objeto. En particular, dadas las metarrepresentaciones $\overline{\mathcal{R}}$ y $\overline{t}$ de un módulo $\mathcal{R}$ y un término t respectivamente, `metaReduce`$(\overline{\mathcal{R}},\overline{t})$ utiliza la misma estrategia que `reduce` para devolver la metarrepresentación del término que resulta de aplicar las ecuaciones en $\mathcal{R}$ sobre t. De manera parecida, `metaRewrite`$(\overline{\mathcal{R}},\overline{t},n)$ reifica `rewrite`, donde n es el número de pasos de reescritura que se quieren aplicar. Además de estas, existen otras operaciones (por ejemplo, para aplicar una única regla a un término o realizar análisis sintáctico) que se pueden encontrar descritas en el manual.

2.4.4 El comprobador de modelos

A partir de su versión 2.0 Maude incluye un comprobador de modelos (model checker, en inglés), de estados explícitos, para la lógica temporal lineal (Eker et al., 2002). Dependiendo de cómo se ejecute Maude, el comprobador podría estar disponible ya desde un principio, pero en cualquier caso siempre se puede activar cargando el fichero `model-checker.maude`.

Dada una teoría de reescritura ejecutable especificada en Maude en un módulo `M` y un estado inicial, digamos que `initial` en una familia $[\texttt{State}]_{\texttt{M}}$, podemos comprobar de manera automática una propiedad cualquiera en LTL se satisface en dicho estado. Para ello debemos definir un nuevo módulo que importe tanto `M` como el módulo predefinido `MODEL-CHECKER` y que incluya la relación de subtipado

```
  subsort State_M < State .
```

(Esta declaración se puede omitir si se ha utilizado el nombre `State` para $\texttt{State}_{\texttt{M}}$; en cualquier caso, nótese que como consecuencia se tiene la identificación de las familias `[State]` y $\texttt{[State]}_{\texttt{M}}$.) A continuación, ha de declararse la sintaxis de las proposiciones atómicas que se satisfacen en cada estado, que llamaremos *predicados de estado* cuando se apliquen a teorías de reescritura. Ello se hace por medio de operadores de tipo `Prop` (que no tienen por qué ser constantes) y su semántica debe definirse mediante ecuaciones que utilicen el operador de satisfacción `op _|=_ : State Prop -> Bool`. Una vez que la semántica de los predicados de estado ha sido definida y suponiendo que el conjunto de estados alcanzables desde `initial` es finito, podemos utilizar el comprobador de modelos con cualquier fórmula `phi` en LTL sobre dichos predicados de estado invocando en Maude el comando:

```
Maude> reduce modelCheck(initial, phi) .
```

En la sección 4.1 se dará una descripción más detallada de la manera en que se asocian los predicados de estado a las teorías de reescritura para generar estructuras de Kripke.

Continuando con el ejemplo de las listas de naturales, vamos a ilustrar el uso del comprobador de modelos especificando un predicado de estado que se satisfaga exactamente por las listas con un único elemento.

```
mod NAT-LIST-CHECK is
  including NAT-LIST-SUM .
  including MODEL-CHECKER .

  subsort List < State .

  op singleton : -> Prop .

  eq (N:MyNat |= singleton) = true .
  eq (nil |= singleton) = false .
  eq (N1:MyNat N2:MyNat L:List) |= singleton = false .
endm
```

Para el correcto funcionamiento del comprobador de modelos sería suficiente con especificar solo los casos positivos en los que una proposición se cumple; sin embargo, en los desarrollos teóricos de posteriores capítulos esto deja de ser cierto por lo que desde ahora especificamos completamente todos los casos.

Ahora podemos comprobar que, dada por ejemplo la lista inicial

```
zero (s(zero) +' s(s(zero))) s(zero),
```

eventualmente se ha de alcanzar una lista unitaria a partir de la única regla introducida en `NAT-LIST-SUM`.

```
Maude> red modelCheck(zero (s(zero) +' s(s(zero))) s(zero), <> singleton) .
reduce in NAT-LIST-CHECK : modelCheck(zero (s(zero) +' s(s(zero))) s(zero), <>
    singleton) .
result Bool: true
```

La propiedad obviamente no es cierta para la lista vacía, por lo que para ella el comprobador de modelos devuelve un contraejemplo que en este caso se limita a señalar que desde `nil` no se puede avanzar más.

```
Maude> red modelCheck(nil, <> singleton) .
reduce in NAT-LIST-CHECK : modelCheck(nil, <> singleton) .
result ModelCheckResult: counterexample(nil, {nil,deadlock})
```

2.5 El ITP

El razonamiento mecánico sobre especificaciones en lógica ecuacional de pertenencia escritas en Maude está soportado por un demostrador inductivo de teoremas, de carácter experimental, llamado ITP (Inductive Theorem Prover). Este es un demostrador basado en reescritura escrito en Maude y es en sí mismo una especificación ejecutable. Una característica fundamental del ITP es su diseño reflexivo, que permite la definición de *estrategias de reescritura* distintas de la que Maude aplica por defecto; actualmente esta capacidad está siendo utilizada para extender el ITP con procedimientos de decisión para la aritmética lineal, listas y combinaciones de teorías (Clavel et al., 2004c). Desafortunadamente no existe aún un manual de uso del ITP y, aunque se puede encontrar alguna información en Clavel (2000, 2001), la mejor referencia es directamente la página web desde la que se puede descargar la herramienta (Clavel, 2004).

Aunque todavía en fase de desarrollo, el ITP ha alcanzado un grado aceptable de estabilidad en las últimas versiones (en las que el nombre ha sido cambiado por xITP). Actualmente puede trabajar con cualquier módulo presente en la base de datos de Maude, aunque no con aquellos introducidos en Full Maude; esta limitación se pretende eliminar en un futuro para soportar demostraciones sobre módulos parametrizados. La versión utilizada en esta tesis se hizo pública el 13 de abril de 2004; versiones más recientes pueden requerir ligeros cambios sintácticos en las pruebas.

Para ilustrar cómo se utiliza el ITP vamos a demostrar que la operación `_+'_` definida en el módulo `NAT-LIST` de la sección 2.4.1 es asociativa. Tras cargar el módulo que define el ITP con la instrucción `in xitp-tool.maude` e inicializar su base de datos con la orden `loop init-itp .`, la propiedad se puede demostrar con los siguientes comandos escritos entre paréntesis:

```
(goal associative : NAT-LIST |- A{N1:MyNat ; N2:MyNat ; N3:MyNat}
     (((N1:MyNat +' N2:MyNat) +' N3:MyNat) =
                    (N1:MyNat +' (N2:MyNat +' N3:MyNat))) .)

(ind on N3:MyNat .)
(auto* .)
(auto* .)
```

Las tres primeras líneas introducen el objetivo deseado: `associative` es su nombre, `NAT-LIST` el módulo sobre el que se quiere probar y el símbolo `A` (que representa $\forall$) precede una lista de variables cuantificadas universalmente. Hay que señalar que las variables

siempre aparecen anotadas con sus tipos. Tras esto se intenta demostrar la propiedad mediante inducción estructural sobre la tercera variable y el ITP genera el subobjetivo correspondiente para cada operador con rango `MyNat` que haya sido declarado con el atributo `ctor`; en este caso uno para el número `zero`, que es

```
|- A{N1:MyNat ; N2:MyNat}
   ((N1:MyNat +' N2:MyNat)+' z = N1:MyNat +'(N2:MyNat +' z))
```

y otro para el constructor `s` (con la correspondiente hipótesis de inducción):

```
|- A{V0#0:MyNat}
   ((A{N1:MyNat ; N2:MyNat}
    ((N1:MyNat +' N2:MyNat)+' V0#0:MyNat = N1:MyNat +'(N2:MyNat +' V0#0:MyNat)))
    ==>
    (A{N1:MyNat ; N2:MyNat}
    ((N1:MyNat +' N2:MyNat)+' s(V0#0:MyNat) =
                  N1:MyNat +'(N2:MyNat +' s(V0#0:MyNat)))))
```

Finalmente, ambos casos se pueden demostrar automáticamente con la orden `auto*` que transforma todas las variables en constantes nuevas y a continuación reescribe los términos a ambos lados de la ecuación tanto como sea posible y comprueba si son iguales.

Capítulo 3

Reflexión

Presentamos en este capítulo el concepto de *reflexión* y damos una demostración detallada que muestra que la lógica de pertenencia y la lógica de reescritura en su versión más general son reflexivas. El concepto de reflexión, aunque estudiado y explotado en la lógica de reescritura de especial manera, ciertamente no es específico de esta. En efecto, el concepto de reflexión se puede expresar de manera que se aplique a lógicas arbitrarias (Clavel y Meseguer, 1996). Para poder hacerlo, sin embargo, es necesario definir antes de manera precisa lo que se entiende por una *lógica*. Esta tarea fue llevada a cabo en Meseguer (1989), donde se propone la *teoría de lógicas generales* como un marco muy abstracto en el que lógicas muy distintas pueden ser representadas y relacionadas entre sí. Comenzamos entonces presentando un breve resumen de los conceptos de esta teoría necesarios para formalizar la idea de reflexión, antes de abordar la demostración de la propiedad en las lógicas que nos interesan. Por último avanzamos algunas ideas sobre el uso de la reflexión para el razonamiento metalógico, extendiendo los principios de razonamiento presentados en Basin et al. (2004) y mostrando cómo se pueden aplicar para estudiar algunas relaciones semánticas entre teorías distintas expresadas en la lógica de pertenencia.

3.1 Reflexión en el marco de las lógicas generales

Presentamos aquí de manera resumida los axiomas que caracterizan la noción de *lógica reflexiva* (Clavel y Meseguer, 1996). En primer lugar introducimos las nociones de *sintaxis* y *sistema de derivación*, que son utilizadas en la axiomatización; esta última puede considerarse como una idea "hermana" del concepto de institución de Goguen y Burstall (1992), correspondiente al ámbito de la sintaxis en lugar de al de la semántica. Estas nociones se definen utilizando el lenguaje de la teoría de categorías, pero no requieren ningún conocimiento de esta teoría matemática más allá de las nociones básicas de categoría y funtor.

Sintaxis. La sintaxis típicamente viene dada por una *signatura* Σ que proporciona una gramática sobre la que construir *sentencias*. Para la lógica de primer orden, por ejemplo,

las signaturas consisten de un conjunto de símbolos de función y un conjunto de símbolos de predicados, cada uno con un número determinado de argumentos, que se utilizan para construir las sentencias. Nosotros suponemos que para cada lógica existe una categoría **Sign** de signaturas posibles para ella y un funtor $sen : \mathbf{Sign} \longrightarrow \mathbf{Set}$ que asigna a cada signatura Σ el conjunto $sen(\Sigma)$ de todas sus sentencias. Al par $(\mathbf{Sign}, sen)$ lo llamamos *sintaxis*.

Sistemas de derivación. Dada una signatura Σ en **Sign**, la *derivabilidad* (también llamada *deducibilidad*) de una sentencia $\varphi \in sen(\Sigma)$ a partir de un conjunto de axiomas $\Gamma \subseteq sen(\Sigma)$ es una relación $\Gamma \vdash_\Sigma \varphi$ que se cumple si y solo si podemos demostrar φ a partir de los axiomas Γ utilizando las reglas de la lógica. Esta relación es siempre relativa a una signatura.

En lo que sigue, $|C|$ denota la colección de objetos de una categoría C.

Definición 3.1 *Un* sistema de derivación *es una terna* $\mathcal{E} = (\mathbf{Sign}, sen, \vdash)$ *tal que*

- $(\mathbf{Sign}, sen)$ *es una sintaxis,*
- $\vdash$ *es una función que asocia a cada* $\Sigma \in |\mathbf{Sign}|$ *una relación binaria* $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$, *llamada* Σ-derivación, *que satisface las siguientes propiedades:*
  1. reflexividad*: para todo* $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$,
  2. monotonía*: si* $\Gamma \vdash_\Sigma \varphi$ *y* $\Gamma' \supseteq \Gamma$ *entonces* $\Gamma' \vdash_\Sigma \varphi$,
  3. transitividad*: si* $\Gamma \vdash_\Sigma \varphi$ *para todo* $\varphi \in \Delta$ *y* $\Gamma \cup \Delta \vdash_\Sigma \psi$, *entonces* $\Gamma \vdash_\Sigma \psi$,
  4. $\vdash$-traducción*: si* $\Gamma \vdash_\Sigma \varphi$, *entonces para todo* $H : \Sigma \longrightarrow \Sigma'$ *en* **Sign** *tenemos* $sen(H)(\Gamma) \vdash_{\Sigma'} sen(H)(\varphi)$.

La relación de derivación $\vdash$ induce una función que lleva cada conjunto de sentencias Γ al conjunto $\Gamma^\bullet = \{\varphi \mid \Gamma \vdash \varphi\}$. Llamamos a $\Gamma^\bullet$ el conjunto de *teoremas* demostrables a partir de Γ.

Definición 3.2 *Dado un sistema de derivación* $\mathcal{E}$, *su categoría* **Th** *de* teorías *tiene como objetos los pares* $T = (\Sigma, \Gamma)$ *con* Σ *una signatura y* $\Gamma \subseteq sen(\Sigma)$. *Un* morfismo de teorías $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ *es un morfismo de signaturas* $H : \Sigma \longrightarrow \Sigma'$ *tal que si* $\varphi \in \Gamma$, *entonces* $\Gamma' \vdash_{\Sigma'} sen(H)(\varphi)$.

Nótese que el funtor *sen* se puede extender a un funtor $sen : \mathbf{Th} \longrightarrow \mathbf{Set}$ sin más que definir $sen(\Sigma, \Gamma) = sen(\Sigma)$.

3.1.1 Lógicas reflexivas

Una lógica reflexiva es una lógica en la que aspectos importantes de su metateoría pueden ser representados al nivel objeto de manera consistente, de manera que esta representación simula correctamente los aspectos metateóricos relevantes. Dos nociones metateóricas que pueden ser reflejadas son las nociones de teoría y la relación de derivación $\vdash$. Para capturar formalmente esta idea utilizamos la noción de teoría universal,

teniendo en cuenta que la universalidad puede no ser absoluta sino tan solo relativa a una clase C de teorías *representables*.

Normalmente, para que una teoría sea representable al nivel objeto debe tener una descripción finitaria de alguna manera—digamos, siendo recursivamente enumerable—para que pueda ser representada como un fragmento de lenguaje. En la terminología del enfoque axiomático de la computabilidad de Shoenfield (1971), deberíamos exigir que las teorías T en C sean *objetos finitos*; en las propias palabras de Shoenfield, "object(s) which can be specified by a finite amount of information", esto es, objetos que pueden ser especificados con una cantidad finita de información. A su vez, la clase de teorías C debería ser un *espacio*, una clase X de objetos finitos tales que, dado un objeto finito x, podemos decidir si x pertenece a X o no. Los informáticos llaman habitualmente a tales objetos finitos *estructuras de datos* y a tales espacios *tipos de datos*. Naturalmente, en lógicas finitarias típicas, si la signatura de una teoría T es un objeto finito el conjunto $sen(T)$ de sus sentencias también es un espacio en el sentido de más arriba; esto es, tales sentencias son objetos finitos y podemos determinar de manera efectiva cuándo son sentencias legales en T. Dados espacios X e Y, la noción de Shoenfield de *función recursiva* $f : X \longrightarrow Y$ es entonces una función (total) que puede ser computada por un *algoritmo*, es decir, por un programa de ordenador cuando no tenemos en cuenta limitaciones de espacio ni tiempo.

Definición 3.3 *Dado un sistema de derivación $\mathcal{E}$ y un conjunto de teorías $C \subseteq |\mathbf{Th}|$, una teoría U es C-*universal *si existe una función*

$$\overline{\_ \vdash \_} : \bigcup_{T \in C} \{T\} \times sen(T) \longrightarrow sen(U)\,,$$

llamada función de representación*, tal que para toda teoría $T \in C$ y $\varphi \in sen(T)$,*

$$T \vdash \varphi \quad \Longleftrightarrow \quad U \vdash \overline{T \vdash \varphi}\,.$$

*Si, además, $U \in C$, el sistema de derivación $\mathcal{E}$ recibe el nombre de C-*reflexivo*.*

Para tener en cuenta cuestiones de computabilidad, debemos exigir también que la función de representación $\overline{\_ \vdash \_}$ sea recursiva[1]*. Por último, para descartar representaciones que no sean fidedignas exigimos que la función sea* inyectiva*.*

Hay que señalar que en un sistema de derivación reflexivo, como la propia teoría U es representable, la representación se puede iterar de manera que inmediatamente obtenemos una "torre reflexiva":

$$T \vdash \varphi \quad \Longleftrightarrow \quad U \vdash \overline{T \vdash \varphi} \quad \Longleftrightarrow \quad U \vdash \overline{U \vdash \overline{T \vdash \varphi}} \quad \ldots$$

Nosotros vamos a trabajar con lo que llamamos teorías *finitamente presentables*, que son aquellas cuya signatura y conjunto de sentencias son finitos. Nuestros argumentos también se aplican de manera más general a teorías que, sin ser finitas, tienen una descripción que sí lo es; sin embargo, las codificaciones que resultan en este caso son enrevesadas y no resultan de utilidad ni en Maude ni para el razonamiento metalógico.

[1] Nótese que, bajo las suposiciones de efectividad mencionadas con anterioridad, $\bigcup_{T \in C} \{T\} \times sen(T)$ es un espacio, ya que sus elementos deben ser pares de elementos finitos de la forma (T, φ), donde podemos decidir en primer lugar si T está en el espacio C y a continuación si φ está en el espacio $sen(T)$.

3.2 Reflexión en la lógica de pertenencia

3.2.1 Ajustes en la presentación de la lógica de pertenencia

Antes de pasar a la demostración de la propiedad de reflexión para la lógica de pertenencia vamos a realizar una serie de pequeñas asunciones y cambios técnicos con respecto a la presentación dada en la sección 2.2. Estas modificaciones no alteran en ningún modo fundamental el carácter de la lógica, pero ayudan a que la prueba resulte más simple.

En primer lugar, para simplificar la definición de la teoría universal para la lógica de pertenencia, en lo que sigue trabajaremos con teorías con *familias no vacías*, esto es, existe al menos un término cerrado para cada familia. Esta es una restricción relativamente menor que facilita el desarrollo de las secciones posteriores y no tiene ninguna influencia en el resto de la tesis, y que evita las conocidas complicaciones con la cuantificación en la lógica ecuacional heterogénea (Goguen y Meseguer, 1985). Por lo tanto, desde ahora hasta que terminemos la demostración omitiremos los cuantificadores en todas las sentencias ya que no son necesarios bajo esta hipótesis.

Normalmente denotamos las signaturas en la lógica de pertenencia por su conjunto de operadores Σ. Aquí, sin embargo, habrá ocasiones en las que será necesario distinguir entre la signatura completa y sus operadores por lo que utilizaremos la letra Ω para referirnos a la primera y mantendremos Σ para los segundos.

Las sustituciones se presentaron en la sección 2.2 como funciones que aplican variables en términos. Aunque conceptualmente esta es la definición correcta, por razones técnicas nosotros preferimos definir aquí las sustituciones como un caso especial de listas de pares de variables y términos. Dada una signatura Σ y un conjunto de variables $X = \{x_1 : k_1, \ldots, x_n : k_n\}$, definimos $\mathcal{S}_{(\Sigma,X)}$ como el conjunto de *sustituciones*

$$\mathcal{S}_{(\Sigma,X)} = \{(x_1 \mapsto w_1, \ldots, x_n \mapsto w_n) \mid x_i \neq x_j \text{ si } i \neq j, \text{ y } x_i \text{ y } w_i \text{ son de la misma familia}\}.$$

Dado un término t y una sustitución $\sigma = (x_1 \mapsto w_1, \ldots, x_n \mapsto w_n)$, denotamos mediante $\sigma(t)$ el término $t(w_1/x_1, \ldots, w_n/x_n)$ obtenido a partir de t sustituyendo simultáneamente cada w_i por x_i, $i = 1, \ldots, n$.

A continuación presentamos nuestra noción de contextos y la correspondiente notación que utilizaremos. Dada una signatura $\Sigma = (K, \Sigma, S)$, un conjunto de variables X, cada una asociada a una familia en K, y un conjunto K-indizado de constantes nuevas $\{\iota_k\}_{k\in K}$, un *contexto* es un término C^k en $T_{\Sigma\cup\{\iota_k\}_{k\in K}}(X)$ que contiene exactamente un subtérmino ι_k, llamado su "agujero", para una familia $k \in K$. Definimos $\mathrm{C}^{\iota}_{\Sigma}(X)$ como el conjunto de todos los contextos. Dado un contexto C^k y un término $t \in T_\Sigma(X)$ en la familia k, $C^k[t] \in T_\Sigma(X)$ es el término que resulta de reemplazar el "agujero" ι_k en C^k por t. Cuando no sea necesario omitiremos mencionar la familia del "agujero" en el contexto.

Por último, el cálculo de pruebas que utilizaremos en este capítulo es ligeramente distinto, pero equivalente y más simple para nuestros propósitos, al original de Meseguer (1998) presentado en la sección 2.2. Esta versión elimina la regla **(Congruencia)**, que pasa a tenerse en cuenta en la siguiente versión modificada de la regla de **(Reemplazamiento)**.

- **Reemplazamiento**. Para cada ecuación $t = t'$ **if** $C_{mb} \wedge C_{eq}$ en E, con t, t' en la familia k, contexto $C^k \in C^{\iota}_{\Sigma}(X)$ y sustitución σ, donde $C_{mb} \triangleq (u_1 : s_1 \wedge \ldots \wedge u_j : s_j)$ y $C_{eq} \triangleq (v_1 = v'_1 \wedge \ldots \wedge v_k = v'_k)$,

$$\frac{T \vdash \sigma(u_1) : s_1 \quad \cdots \quad T \vdash \sigma(u_j) : s_j \quad T \vdash \sigma(v_1) = \sigma(v'_1) \quad \cdots \quad T \vdash \sigma(v_k) = \sigma(v'_k)}{T \vdash C[\sigma(t)] = C[\sigma(t')]}\,.$$

  De manera análoga, para cada axioma de pertenencia $t : s$ **if** $C_{mb} \wedge C_{eq}$ en E y sustitución σ, donde C_{mb} y C_{eq} son como antes,

$$\frac{T \vdash \sigma(u_1) : s_1 \quad \cdots \quad T \vdash \sigma(u_j) : s_j \quad T \vdash \sigma(v_1) = \sigma(v'_1) \quad \cdots \quad T \vdash \sigma(v_k) = \sigma(v'_k)}{T \vdash \sigma(t) : s}\,.$$

3.2.2 Una teoría universal para MEL

Comenzamos ahora a presentar la teoría universal $\mathrm{U}_{\mathrm{MEL}}$ y la función de representación $\overline{\_ \vdash \_}$ que codifica pares formados por una teoría y una sentencia sobre su signatura por medio de una sentencia en $\mathrm{U}_{\mathrm{MEL}}$. En lo que sigue solo trabajaremos con teorías finitamente presentables en MEL.

La signatura de $\mathrm{U}_{\mathrm{MEL}}$

La signatura de la teoría $\mathrm{U}_{\mathrm{MEL}}$ contiene constructores para representar términos, contextos, sustituciones, familias, tipos, signaturas, axiomas y teorías. Para mejorar la legibilidad presentamos la signatura de $\mathrm{U}_{\mathrm{MEL}}$ como una especificación de Maude; como Maude soporta sintaxis "misfija" (misfix syntax) definida por el usuario, podemos así especificar un operador en su forma más legible. En lo que sigue tan solo enumeramos los operadores de la signatura; las familias son aquellas que aparezcan en alguna declaración de operadores y no se utilizan tipos sino solo familias.

Los principales operadores en $\mathrm{U}_{\mathrm{MEL}}$ son los siguientes. En primer lugar, la signatura de $\mathrm{U}_{\mathrm{MEL}}$ contiene un pequeño subconjunto de los operadores booleanos básicos.

```
op true : -> [Bool] .
op false : -> [Bool] .
op _and_ : [Bool] [Bool] -> [Bool] .
op _or_ : [Bool] [Bool] -> [Bool] .
```

Para representar los símbolos que aparecen en una teoría de la lógica de pertenencia utilizaremos listas de caracteres ASCII precedidos de un apóstrofo, construidas con los constructores `nilM` y `consM`. Tenemos también dos predicados de igualdad que devuelven `true` si dos caracteres o dos listas son sintácticamente iguales y `false` en caso contrario.

```
ops 'a 'b 'c ... : -> [ASCII] .
op equalASCII : [ASCII] [ASCII] -> [Bool] .
op nilM : -> [MetaExp] .
op consM : [ASCII] [MetaExp] -> [MetaExp] .
op equalM : [MetaExp] [MetaExp] -> [Bool] .
```

Para representar familias y listas de familias:

```
op k : [MetaExp] -> [Kind] .
op nilK : -> [KindList] .
op consK : [Kind] [KindList] -> [KindList] .
```

Variables, términos arbitrarios y listas de términos:

```
op v : [MetaExp] [Kind]  -> [Term] .
op _[_] : [MetaExp] [TermList] -> [Term] .
op nilTL : -> [TermList] .
op consTL : [Term] [TermList] -> [TermList] .
```

Nótese que la familia de una variable se metarrepresenta junto con su nombre. Durante el resto de la demostración, como la metarrepresentación hace uso tanto del nombre x como de la familia k, también nos referiremos a las variables al nivel objeto como pares $x : k$.

Por ejemplo, el término $x : [Nat] + (y : [Nat] - 0)$ se metarrepresenta en $\mathrm{U}_{\mathrm{MEL}}$ como:

```
consM('_,consM('+,consM('_,nilM)))
    [consTL(v('x,k(consM('N,consM('a,consM('t,nilM))))),
            consTL(consM('_,consM('-,consM('_,nilM)))
                       [consTL(v('y,k(consM('N,consM('a,consM('t,nilM))))),
                               consTL('0[nilTL],nilTL))],nilTL))]
```

Los tipos y operadores se representan utilizando

```
op s : [MetaExp] [Kind] -> [Sort] .
op _:_->_ : [MetaExp] [KindList] [Kind] -> [Operator] .
```

Las sustituciones se representan con los siguientes constructores, donde '-' representa la sustitución vacía que no asigna ningún término a ninguna variable y por tanto actúa como la identidad sobre términos.

```
op - : -> [Substitution] .
op v(_,_)->_ : [MetaExp] [Kind] [Term] -> [Assignment] .
op consS : [Assignment] [Substitution] -> [Substitution] .
```

Para los contextos tenemos los siguientes constructores:

```
op * : -> [Context] .
op _[_] : [MetaExp] [CTermList] -> [Context] .
op consCTL1 : [Context] [TermList] -> [CTermList] .
op consCTL2 : [Term] [CTermList] -> [CTermList] .
```

La familia `[CTermList]` se introduce para representar listas de términos tales que solo haya un "agujero" entre todos, esto es, solo uno de los términos es un contexto. Debido a esto existen dos operadores cons diferentes para construir estas listas: uno para añadir términos arbitrarios por la izquierda y otro para añadir un contexto cuando todavía no hay ninguno. Por ejemplo, el contexto $0 + (\iota - x : [Nat])$ se metarrepresenta como:

```
consM('_,consM('+,consM('_,nilM)))
    [consCTL2('0[nilTL],
              consCTL1(consM('_,consM('-,consM('_,nilM)))
                         [consCTL1(*,
                            consTL(v('x,k(consM('N,consM('a,consM('t,nilM))))),
                            nilTL))],
                       nilTL))]
```

Para representar ecuaciones y axiomas de pertenencia (posiblemente condicionales), la signatura de U_{MEL} incluye los constructores

```
op _=_ : [Term] [Term] -> [Atom] .
op _:_ : [Term] [Sort] -> [Atom] .
op none : -> [Condition] .
op _/\_ : [Atom] [Condition] -> [Condition] .
op _if_ : [Atom] [Condition] -> [Axiom] .
```

donde la constante none se utiliza para representar la ausencia de condiciones. Los constructores

```
op (_,_,_) : [KindSet] [OperatorSet] [SortSet] -> [Signature] .
op (_,_) : [Signature] [AxiomSet] -> [MelTheory] .
```

son los que se utilizan para representar signaturas y teorías, respectivamente. Finalmente, la signatura de U_{MEL} contiene el operador booleano

```
op _|-_ : [MelTheory] [Axiom] -> [Bool] .
```

para representar la derivación de sentencias en una teoría dada en la lógica de pertenencia; los axiomas principales de U_{MEL}, incluidos los de la figura 3.1 en la página 44, definen el comportamiento de estos dos operadores.

Además, la signatura de U_{MEL} contiene constructores para conjuntos de familias, de axiomas, de tipos, de variables y de operadores.

```
op emptyK : -> [KindSet] .
op unionK : [Kind] [KindSet] -> [KindSet] .
op emptyA : -> [AxiomSet] .
op unionA : [Axiom] [AxiomSet] -> [AxiomSet] .
op emptyS : -> [SortSet] .
op unionS : [Sort] [SortSet] -> [SortSet] .
op emptyV : -> [VarSet] .
op unionV : [Term] [VarSet] -> [VarSet] .
op emptyOp -> [OperatorSet] .
op unionOp : [Operator] [OperatorSet] -> [OperatorSet] .
```

Nótese que un nombre más apropiado para [VarSet] sería [TermSet] puesto que los constructores permiten construir conjuntos de términos arbitrarios. Sin embargo nosotros solo utilizaremos conjuntos de variables y creemos que este nombre resulta más intuitivo en algunas declaraciones.

En la signatura de U_{MEL} se incluye también:

- un operador booleano `parse` para decidir si un término está bien formado con respecto a una signatura;
- un operador `vars` para extraer las variables que aparecen en una sentencia;
- un operador `addVarAsCnst` para extender una signatura añadiendo algunas variables como nuevas constantes;
- un operador `makeGround` que transforma las variables de un término en constantes con el mismo nombre;
- un operador `addEq` que tiene como argumentos un conjunto de axiomas y una condición, y añade al conjunto las fórmulas atómicas que aparecen en esta última; y
- operadores `applyC` y `applyS` para aplicar un contexto y una sustitución respectivamente a un término.

```
op parse : [Term] [Signature] -> [Bool] .
op vars : [Axiom] -> [VarSet] .
op addVarAsCnst : [Signature] [VarSet] -> [Signature] .
op makeGround : [Term] -> [Term] .
op addEq : [AxiomSet] [Condition] -> [AxiomSet] .
op applyC : [Context] [Term] -> [Term] .
op applyC : [CTermList] [Term] -> [TermList] .
op applyS : [Substitution] [Term] -> [Term] .
op applyS : [Substitution] [TermList] -> [TermList] .
```

Por último, los operadores

```
op satisfyC : [MelTheory] [Condition] [Substitution] -> [Bool] .
op satisfyA : [MelTheory] [Atom] [Substitution] -> [Bool] .
```

se utilizan para decidir, dadas las representaciones de una teoría T, de una condición C y de una sustitución σ, si todas las fórmulas atómicas en $\sigma(C)$ se pueden derivar en T.

Hay algunos otros operadores auxiliares que se irán introduciendo en las siguientes secciones según se vayan necesitando.

La función de representación

A continuación definimos la función de representación $\overline{\_ \vdash \_}$. Para toda teoría T en la lógica de pertenencia y sentencia ϕ sobre la signatura de T,

$$\overline{T \vdash \phi} \triangleq (\overline{T} \,|\!-\, \overline{\phi}) = \texttt{true}\,,$$

donde $\overline{(\_)}$ es una función definida recursivamente de la siguiente manera:

1. Para una teoría,
$$\overline{(\Omega, E)} \triangleq (\overline{\Omega}, \overline{E})\,.$$

2. Para un conjunto de sentencias,

$$\overline{\{\varphi_1,\ldots,\varphi_n\}} \triangleq \texttt{unionA(}\overline{\varphi_1}\texttt{,}\ldots\texttt{,unionA(}\overline{\varphi_n}\texttt{,emptyA)}\ldots\texttt{)}\,.$$

3. Para una fórmula atómica,

$$\overline{t = t'} \triangleq \overline{t} = \overline{t'} \quad \text{y} \quad \overline{t : s} \triangleq \overline{t} : \overline{s}\,.$$

4. Para una conjunción de fórmulas atómicas,

$$\overline{A_1 \wedge A_2 \wedge \ldots \wedge A_n} \triangleq \overline{A_1}\,\texttt{/\textbackslash}\,(\overline{A_2}\,\texttt{/\textbackslash}\,\ldots\,\texttt{/\textbackslash}\,(\overline{A_n}\,\texttt{/\textbackslash}\,\texttt{none})\ldots)\,.$$

5. Para una sentencia,

$$\overline{A_0\ \textbf{if}\ A_1 \wedge \ldots \wedge A_n} \triangleq \overline{A_0}\ \texttt{if}\ \overline{A_1 \wedge \ldots \wedge A_n}\,.$$

6. Para una lista no vacía de caracteres ASCII,

$$\overline{(a_1,\ldots,a_n)} \triangleq \texttt{consM('}a_1\texttt{,}\ldots\texttt{,consM('}a_n\texttt{,nilM)}\ldots\texttt{)}\,.$$

7. Para una variable x de la familia k,

$$\overline{x : k} \triangleq \texttt{v(}\overline{x}\texttt{,}\overline{k}\texttt{)}\,.$$

8. Para t un término de la forma $f(t_1,\ldots,t_n)$,

$$\overline{t} \triangleq \overline{f}\texttt{[}\overline{t_1,\ldots,t_n}\texttt{]}\,.$$

9. Para una lista de términos,

$$\overline{(t_1,\ldots,t_n)} \triangleq \texttt{consTL(}\overline{t_1}\texttt{,}\ldots\texttt{,consTL(}\overline{t_n}\texttt{,nilTL)}\ldots\texttt{)}\,.$$

10. Para una sustitución,

$$\overline{x_1 : k_1 \mapsto w_1,\ldots,x_n : k_n \mapsto w_n} \triangleq$$
$$\texttt{consS(v(}\overline{x_1}\texttt{,}\overline{k_1}\texttt{) -> }\overline{w_1}\texttt{,}\ldots\texttt{,consS(v(}\overline{x_n}\texttt{, }\overline{k_n}\texttt{) -> }\overline{w_n}\texttt{,-)}\ldots\texttt{)}\,.$$

Y análogamente para el resto de elementos: contextos, signaturas y conjuntos de familias, de tipos y de operadores.

Nótese que, tal y como están escritas, algunas definiciones por el momento son ambiguas. Por ejemplo, en el punto (2), dependiendo de cómo ordenemos los elementos del conjunto podemos obtener una metarrepresentación u otra. Este problema desaparecerá en la siguiente sección una vez que demos ecuaciones apropiadas para todos los constructores de conjuntos.

Los axiomas de $\mathrm{U}_{\mathrm{MEL}}$

Definimos ahora los axiomas de $\mathrm{U}_{\mathrm{MEL}}$, que incluyen ecuaciones que corresponden a las reglas de inferencia de MEL junto con ecuaciones que definen los operadores introducidos con anterioridad: `parse`, `vars`, `satisfyC`, ... Para facilitar la comprensión de estas ecuaciones, reemplazamos la notación habitual para variables por la correspondiente representación de las entidades que se van a colocar en tal posición. Por ejemplo, $\overline{\Omega}$ es una variable normal pero la notación sugiere que los términos que tal variable ajustará serán normalmente representaciones de signaturas.

En lo que sigue, asumimos una teoría $T = (\Omega, E)$ finitamente presentable en la lógica de pertenencia, donde $\Omega = (K, \Sigma, S)$.

Empezamos con las ecuaciones obvias para los operadores booleanos:

```
eq true and B = B .
eq false and B = false .
eq true or B = true .
eq false or B = B .
```

A continuación, las ecuaciones para los constructores de conjuntos; para conjuntos de axiomas:

```
eq unionA(φ̄, unionA(ψ̄,ĀS)) = unionA(ψ̄, unionA(φ̄,ĀS)) .
eq unionA(φ̄, unionA(φ̄,ĀS)) = unionA(φ̄,ĀS) .
```

y de forma análoga para familias, tipos, variables y operadores, y para el operador _/_ que construye condiciones:

```
eq φ̄ /\ (ψ̄ /\ C̄ond) = ψ̄ /\ (φ̄ /\ C̄ond) .
eq φ̄ /\ (φ̄ /\ C̄ond) = φ̄ /\ C̄ond .
```

En particular, se tiene el siguiente resultado, con el que la definición (2) anterior deja de ser ambigua.

Proposición 3.1 *Para todo conjunto E de sentencias y $\varphi \in E$,*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \overline{E} = \mathtt{union}(\overline{\varphi}, \overline{E \setminus \{\varphi\}}) .$$

A lo largo de esta sección van a aparecer bastantes resultados auxiliares como este. Sus demostraciones suelen ser inmediatas por inducción estructural, pero tediosas, por lo que no se darán excepto en aquellos casos que requieran alguna explicación adicional.

Para definir el predicado de igualdad para los caracteres ASCII simplemente tenemos que considerar todos los posibles casos:

```
eq equalASCII('a, 'a) = true .
eq equalASCII('a, 'b) = false .
eq equalASCII('a, 'c) = false .
...
```

y su extensión a listas de caracteres es inmediata:

```
eq equalM(nilM, nilM) = true .
eq equalM(nilM, consM(c,cl)) = false .
eq equalM(consM(c,cl), nilM) = false .
eq equalM(consM(c,cl), consM(c',cl')) = equalASCII(c,c') and equalM(cl,cl') .
```

En realidad, también necesitaremos un predicado de igualdad para familias, cuya definición no ofrece ahora ningún problema:

```
op equalK : [Kind] [Kind] -> [Bool] .
eq equalK(k(cl), k(cl')) = equalM(cl,cl') .
```

Se demuestra entonces por inducción estructural sobre términos que:

Proposición 3.2 *Para cualesquiera términos w, w' en $\mathrm{U}_{\mathrm{MEL}}$ pertenecientes, respectivamente, a las familias* `[ASCII]`, `[MetaExp]` *y* `[Kind]`, *y para f igual, respectivamente, a* `equalASCII`, `equalM` *y* `equalK`*:*

- *w y w' son iguales sintácticamente si y solo si $\mathrm{U}_{\mathrm{MEL}} \vdash f(w,w') = \mathtt{true}$, que a su vez es equivalente a $\mathrm{U}_{\mathrm{MEL}} \vdash w = w'$, y*
- *w y w' son sintácticamente diferentes si y solo si $\mathrm{U}_{\mathrm{MEL}} \vdash f(w,w') = \mathtt{false}$, que a su vez es equivalente a $\mathrm{U}_{\mathrm{MEL}} \nvdash w = w'$.*

Demostración. Basta con observar que los constructores son libres (no se han dado, ni se darán en el resto de la sección, ecuaciones entre términos de estas familias) y que los predicados de igualdad solo igualan términos construidos de la misma forma. □

A continuación presentamos las ecuaciones que definen la aplicación de contextos y sustituciones a términos. Primero los contextos:

```
eq applyC(*,t) = t .
eq applyC(f[ctl], t) = f[applyC(ctl,t)] .

eq applyC(consCTL1(C,tl), t) = consTL(applyC(C,t),tl) .
eq applyC(consCTL2(t',ctl), t) = consTL(t',applyC(ctl,t)) .
```

Proposición 3.3 *Para todo término $t \in T_\Sigma(X)$ y contexto $C \in \mathrm{C}^{\iota}_{\Sigma}$, se tiene que*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \mathtt{applyC}(\overline{C}, \bar{t}) = \overline{C[t]}\,.$$

Las ecuaciones que definen la aplicación de una sustitución a un término dado son similares.

```
eq applyS(-,t̄) = t̄ .
eq applyS(consS(v(x̄,k̄)-> w̄,σ̄), v(x̄,k̄)) = w̄ .
ceq applyS(consS(v(x̄′,k̄′)-> w̄,σ̄), v(x̄,k̄)) = applyS(σ̄, v(x̄,k̄))
    if equalM(x̄,x̄′) and equalK(k̄,k̄′) = false .
eq applyS(σ̄,f̄[tl̄]) = f̄[applyS(σ̄,tl̄)] .

eq applyS(σ̄, nilTL) = nilTL .
eq applyS(σ̄, consTL(t̄,tl̄)) = consTL(applyS(σ̄,t̄), applyS(σ̄,tl̄)) .
```

Proposición 3.4 *Para todo término $t \in T_\Sigma(X)$ y sustitución σ, se tiene que*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{applyS}(\overline{\sigma},\overline{t}) = \overline{\sigma(t)}\,.$$

Para comprobar si un término está bien formado introducimos dos operadores auxiliares y las siguientes ecuaciones:

```
op parse : [Term] [Signature] [Kind] -> [Bool] .
op parse : [TermList] [Signature] [KindList] -> [Bool] .

ceq parse(t̄,(K̄,Σ̄,S̄)) = true
    if K̄ = unionK(k̄,K̄) /\ parse(t̄, (K̄, Σ̄,S̄),k̄) = true .
eq parse(v(x̄,k̄), Ω̄, k̄) = true .
ceq parse(f̄[tl̄], (K̄,Σ̄,S̄), k̄) = true
    if Σ̄ = unionOp(f̄ : k̄l -> k̄, Σ̄) /\ parse(tl̄, (K̄,Σ̄,S̄), k̄l) = true .
eq parse(nilTL, Ω̄, nilK) = true .
ceq parse(consTL(t̄, tl̄), Ω̄, consK(k̄, k̄l)) = true
    if parse(t̄, Ω̄, k̄) = true /\ parse(tl̄, Ω̄, k̄) = true .
```

La siguiente proposición es entonces una consecuencia inmediata.

Proposición 3.5 *Para todo término $t \in T_\Sigma(X)$,*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{parse}(\overline{t},\overline{\Omega}) = \texttt{true}\,.$$

Además, para todo término w en $\mathrm{U}_{\mathrm{MEL}}$ perteneciente a la familia [Term] *se cumple que si*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{parse}(w,\overline{\Omega}) = \texttt{true}\,,$$

entonces existe un término $t \in T_\Sigma(X)$ tal que $\mathrm{U}_{\mathrm{MEL}} \vdash w = \overline{t}$.

Demostración. La primera parte es inmediata por inducción estructural sobre t. En la segunda, que es bastante más tediosa, nótese que no podemos concluir que w es $\overline{t}$ debido a los operadores `applyC` y `applyS`. □

El operador auxiliar `satisfyC` comprueba si una condición se cumple en una teoría bajo una sustitución dada. Las ecuaciones simplemente iteran sobre todas las fórmulas atómicas en la condición y dejan a (_|-_) el trabajo de comprobar si se satisfacen.

```
eq satisfyC(T,none,σ) = true .
eq satisfyC(T,A /\ C,σ) = satisfyA(T,A,σ) and satisfyC(T,C,σ) .

eq satisfyA(T,t = t',σ) = (T |- applyS(σ,t) = applyS(σ,t') if none) .
eq satisfyA(T,t : s,σ) = (T |- applyS(σ,t) : s if none) .
```

Proposición 3.6 *Para cualesquiera fórmulas atómicas $A_1, \ldots, A_n$ y sustitución σ, son equivalentes:*

1. $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T \vdash \sigma(A_i)}$ *para cada i; esto es, para cada A_i de la forma $t_i = t_i'$,*

$$\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{\sigma(t_i)} = \overline{\sigma(t_i')} \texttt{ if none}) = \texttt{true}\,,$$

   y para cada A_i de la forma $t_i : s_i$,

$$\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{\sigma(t_i)} : \overline{s_i} \texttt{ if none}) = \texttt{true}\,.$$

2. *Se cumple que*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{satisfyC}(\overline{T}, \overline{A_1 \wedge \ldots \wedge A_n}, \overline{\sigma}) = \texttt{true}\,.$$

Demostración. Se demuestra fácilmente por inducción sobre n que (2) es equivalente a: para cada A_i de la forma $t_i = t_i'$,

$$\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \texttt{applyS}(\overline{\sigma}, \overline{t_i}) = \texttt{applyS}(\overline{\sigma}, \overline{t_i}') \texttt{ if none}) = \texttt{true}\,,$$

y para cada A_i de la forma $t_i : s_i$,

$$\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \texttt{applyS}(\overline{\sigma}, \overline{t_i}) : \overline{s_i} \texttt{ if none}) = \texttt{true}\,.$$

El resultado se sigue ahora de la proposición 3.4. □

Por último damos las ecuaciones que especifican los operadores auxiliares `vars`, `addEq`, `addVarAsCnst` y `makeGround`. Para definir `vars` necesitamos un operador `uniteV` que compute la unión de dos conjuntos de variables, así como operadores auxiliares que calculen las variables en fórmulas atómicas, condiciones, términos y listas de términos.

```
op uniteV : [VarSet] [VarSet] -> [VarSet] .
eq uniteV(emptyV, VS) = VS .
eq uniteV(unionV(V,VS), VS') = unionV(V, uniteV(VS,VS')) .

op vars : [Atom] -> [VarSet] .
op vars : [Condition] -> [VarSet] .
op vars : [Term] -> [VarSet] .
op vars : [TermList] -> [VarSet] .

eq vars(A if C) = uniteV(vars(A), vars(C)) .
eq vars(A /\ C) = uniteV(vars(A), vars(C)) .
eq vars(none) = emptyV .
```

```
eq vars(t = t') = uniteV(vars(t), vars(t')) .
eq vars(t : s) = vars(t) .
eq vars(f[tl]) = vars(tl) .
eq vars(v(x,k)) = uniteV(v(x,k), emptyV) .
eq vars(nilTL) = emptyV .
eq vars(consTL(t, tl)) = uniteV(vars(t), vars(tl)) .
```

Proposición 3.7 *Si X es el conjunto de variables que aparecen en la sentencia φ, entonces*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \mathtt{vars}(\overline{\varphi}) = \overline{X}.$$

La especificación de `addEq`, en cambio, es inmediata.

```
eq addEq(E, none) = E .
eq addEq(E, A /\ C) = unionA(A, addEq(E,C)) .
```

Proposición 3.8 *Dado un conjunto de ecuaciones E y una conjunción de fórmulas atómicas $A_1 \wedge \ldots \wedge A_n$,*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \mathtt{addEq}(\overline{E}, \overline{A_1 \wedge \ldots \wedge A_n}) = \overline{E \cup \{A_1, \ldots, A_n\}}.$$

Como las variables se metarrepresentan junto con su familia, añadirlas como constantes a una signatura consiste en poco más que sustituir el constructor `v` por `_:_->_` y `unionV` por `unionOp`.

```
eq addVarAsCnst(Ω, emptyV) = Ω .
eq addVarAsCnst((K, Σ, S), unionV(v(x,k), VS)) =
   addVarAsCnst((K, unionOp(x : nilK -> k,Σ), S), VS) .
```

Proposición 3.9 *Si $\Omega(X)$ es la signatura que resulta de añadir un conjunto finito de variables X a la signatura Ω como constantes,*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \mathtt{addVarAsCnst}(\overline{\Omega}, \overline{X}) = \overline{\Omega(X)}.$$

Al nivel objeto no existe ninguna diferencia en la manera como variables y constantes se representan al construir términos, pero esto deja de ser cierto al metanivel. La función `makeGround` transforma la metarrepresentación de un término en otro en el que las variables han pasado a ser constantes; para poder especificarla es necesario un operador auxiliar que tome listas de términos como argumento.

```
op makeGround : [TermList] -> [TermList] .

eq makeGround(f[tl]) = f[makeGround(tl)] .
eq makeGround(v(x,k)) = x[nilTL] .
eq makeGround(nilTL) = nilTL .
eq makeGround(consTL(t, tl)) = consTL(makeGround(t), makeGround(tl)) .
```

Proposición 3.10 *Si t un término sobre una signatura Ω con variables en X y t' es el mismo término visto sobre la signatura $\Omega(X)$, se tiene que*

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{makeGround}(\bar{t}) \;=\; \bar{t}' \,.$$

La especificación en $\mathrm{U}_{\mathrm{MEL}}$ de las reglas de inferencia de MEL es ahora inmediata y aparecen listadas en la figura 3.1; nótese el uso de los operadores `addVarAsCnst` y `makeGround` en las ecuaciones que capturan las reglas de `introduccion de implicacion`.

3.2.3 Corrección de la teoría universal $\mathrm{U}_{\mathrm{MEL}}$

La teoría $\mathrm{U}_{\mathrm{MEL}}$ presentada en la sección anterior es universal en el sentido de la definición 3.3.

Teorema 3.1 *Para todas las sentencias $t : s$ **if** $A_1 \wedge \ldots \wedge A_n$ sobre la signatura Ω de T, donde cada A_i es una fórmula atómica,*

$$T \vdash t : s \;\textbf{if}\; A_1 \wedge \ldots \wedge A_n \iff \mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \bar{t} : \bar{s} \;\texttt{if}\; \overline{A_1 \wedge \ldots \wedge A_n}) = \texttt{true}\,.$$

*De forma análoga, para todas las sentencias $t = t'$ **if** $A_1 \wedge \ldots \wedge A_n$ sobre la signatura Ω de T, donde cada A_i es una fórmula atómica,*

$$T \vdash t = t' \;\textbf{if}\; A_1 \wedge \ldots \wedge A_n \iff \mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \bar{t} = \bar{t}' \;\texttt{if}\; \overline{A_1 \wedge \ldots \wedge A_n}) = \texttt{true}\,.$$

Demostración. La dirección ($\Rightarrow$) es el corolario 3.1 más abajo, mientras que la dirección ($\Leftarrow$) se sigue del teorema 3.3. □

Vamos a comenzar con la dirección ($\Rightarrow$) del teorema, demostrando para ello un resultado a partir del cual se obtendrá muy fácilmente el corolario 3.1.

Teorema 3.2 *Para cualesquiera términos $t, t' \in T_\Sigma(X)$ y tipos s en Ω,*

$$T \vdash t = t' \implies \mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \bar{t} = \bar{t}' \;\texttt{if none}) = \texttt{true}$$

y

$$T \vdash t : s \implies \mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \bar{t} : \bar{s} \;\texttt{if none}) = \texttt{true}\,.$$

Demostración. Por inducción estructural sobre la derivación de $T \vdash t = t'$ o $T \vdash t : s$. Según la última regla de derivación utilizada al razonar con T, tenemos:

- **(Reflexividad)**. El resultado se sigue utilizando **(Reemplazamiento)** para razonar con $\mathrm{U}_{\mathrm{MEL}}$, aplicada a la ecuación `reflexividad` de la figura 3.1.
- **(Simetría)**. Por hipótesis de inducción, $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \bar{t}' = \bar{t} \;\texttt{if none}) = \texttt{true}$, y podemos usar **(Reemplazamiento)** aplicada a la ecuación `simetria`.

```
*** reflexividad
eq ((Ω̄,Ē) |- t̄ = t̄ if none) = true .

*** reemplazamiento
ceq ((Ω̄,Ē) |- applyC(C̄, applyS(σ̄,t̄)) = applyC(C̄, applyS(σ̄,t̄′)) if none) = true
    if Ē = unionA(t̄ = t̄′ if Cond̄,Ē′)
    /\ satisfyC((Ω̄,Ē), Cond̄, σ̄) = true .

*** reemplazamiento
ceq ((Ω̄,Ē) |- applyS(σ̄,t̄) : s̄ if none) = true
    if Ē = unionA(t̄ : s̄ if Cond̄,Ē′)
    /\ satisfyC((Ω̄,Ē), Cond̄, σ̄) = true .

*** simetria
ceq ((Ω̄,Ē) |- t̄ = t̄′ if none) = true
    if ((Ω̄,Ē) |- t̄′ = t̄ if none) = true .

*** transitividad
ceq ((Ω̄,Ē) |- t̄ = t̄′ if none) = true
    if parse(t̄′′, Ω̄) = true
    /\ ((Ω̄,Ē) |- t̄ = t̄′′ if none) = true
    /\ ((Ω̄,Ē) |- t̄′′ = t̄′ if none) = true .

*** pertenencia
ceq ((Ω̄, Ē) |- t̄ : s̄ if none) = true
    if parse(ū, Ω̄) = true
    /\ ((Ω̄,Ē) |- t̄ = ū if none) = true
    /\ ((Ω̄,Ē)|- ū : s̄ if none) = true .

*** introduccion de implicacion
ceq ((Ω̄,Ē) |- t̄ = t̄′ if Cond̄) = true
    if
    ((addVarAsCnst(Ω̄, vars(t̄ = t̄′ if Cond̄)), addEq(Ē,Cond̄))
         |- makeGround(t̄) = makeGround(t̄′) if none) =
    true .
ceq ((Ω̄,Ē) |- t̄ : s̄ if Cond̄) = true
    if
    ((addVarAsCnst(Ω̄, vars(t̄ : s̄ if Cond̄)), addEq(Ē,Cond̄))
         |- makeGround(t̄) : s̄ if none) =
    true .
```

Figura 3.1: La teoría universal U_{MEL} (fragmento).

- **(Transitividad)**. En caso de que

$$\frac{T \vdash t = t'' \qquad T \vdash t'' = t'}{T \vdash t = t'},$$

tenemos, por hipótesis de inducción, $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{t} = \overline{t''}\ \texttt{if none}) = \texttt{true}$ y $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{t''} = \overline{t'}\ \texttt{if none}) = \texttt{true}$, y por la proposición 3.5, $\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{parse}(\overline{t''}, \overline{\Omega}) = \texttt{true}$, por lo que podemos aplicar **(Reemplazamiento)** a la ecuación `transitividad` para obtener el resultado.

- **(Pertenencia)**. Si

$$\frac{T \vdash t = t' \qquad T \vdash t' : s}{T \vdash t : s},$$

tenemos, por hipótesis de inducción, $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{t} = \overline{t'}\ \texttt{if none}) = \texttt{true}$ y $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{t'} : \overline{s}\ \texttt{if none}) = \texttt{true}$, y por la proposición 3.5, $\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{parse}(\overline{t'}, \overline{\Omega}) = \texttt{true}$, con lo que se obtiene el resultado aplicando **(Reemplazamiento)** a la ecuación `pertenencia`.

- **(Reemplazamiento)**. Supongamos que $t = t'$ **if** $C_{mb} \wedge C_{eq} \in E$, donde $C_{mb} \triangleq (u_1 : s_1 \wedge \ldots \wedge u_j : s_j)$ y $C_{eq} \triangleq (v_1 = v_1' \wedge \ldots \wedge v_k = v_k')$, y tenemos que, para algún contexto C y sustitución σ,

$$\frac{T \vdash \sigma(u_1) : s_1 \quad \cdots \quad T \vdash \sigma(u_j) : s_j \quad T \vdash \sigma(v_1) = \sigma(v_1') \quad \cdots \quad T \vdash \sigma(v_k) = \sigma(v_k')}{T \vdash C[\sigma(t)] = C[\sigma(t')]}.$$

Por hipótesis de inducción, $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{\sigma(u_i)} : \overline{s_i}\ \texttt{if none}) = \texttt{true}$, para $i = 1, \ldots, j$, y $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{\sigma(v_i)} = \overline{\sigma(v_i')}\ \texttt{if none}) = \texttt{true}$, para $i = 1, \ldots, k$. Por lo tanto, aplicando la proposición 3.6 podemos usar la primera ecuación `reemplazamiento` para obtener

$$\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \texttt{applyC}(\overline{C}, \texttt{applyS}(\overline{\sigma}, \overline{t})) = \texttt{applyC}(\overline{C}, \texttt{applyS}(\overline{\sigma}, \overline{t'}))\ \texttt{if none}) = \texttt{true}$$

y se sigue el resultado ya que, por las proposiciones 3.3 y 3.4,

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{applyC}(\overline{C},\ \texttt{applyS}(\overline{\sigma}, \overline{t})) = \overline{C[\sigma(t)]}$$

y

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{applyC}(\overline{C},\ \texttt{applyS}(\overline{\sigma}, \overline{t'})) = \overline{C[\sigma(t')]}.$$

En caso de que la ecuación de E que se aplicó hubiera sido de la forma $t : s$ **if** $C_{mb} \wedge C_{eq}$ el argumento sería el mismo, pero ahora se utilizaría la segunda de las ecuaciones `reemplazamiento`. □

Corolario 3.1 *Para términos cualesquiera $t, t' \in T_\Sigma(X)$, tipos s en Ω y fórmulas atómicas $A_1, \ldots, A_n$, se tiene que*

$$T \vdash t = t' \textbf{ if } A_1 \wedge \ldots \wedge A_n \implies \mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{t} = \overline{t'}\ \texttt{if}\ \overline{A_1 \wedge \ldots \wedge A_n}) = \texttt{true}$$

y

$$T \vdash t : s \textbf{ if } A_1 \wedge \ldots \wedge A_n \implies \mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T} \mid\!- \overline{t} : \overline{s}\ \texttt{if}\ \overline{A_1 \wedge \ldots \wedge A_n}) = \texttt{true}.$$

Demostración. Demostramos la primera implicación; la otra es análoga. Sea X el conjunto de variables que aparecen en $t = t'$ **if** $A_1 \wedge \ldots \wedge A_n$.

$$\begin{aligned}(\Omega, E) \vdash t = t' \textbf{ if } A_1 \wedge \ldots \wedge A_n & \\ \Longleftrightarrow (\Omega(X), E \cup \{A_1, \ldots, A_n\}) \vdash t = t' & \\ \Longrightarrow \mathrm{U}_{\mathrm{MEL}} \vdash ((\overline{\Omega(X)}, \overline{E \cup \{A_1, \ldots, A_n\}}) \mid\!- \bar{t} = \bar{t}' \texttt{ if none}) = \texttt{true} & \\ \Longrightarrow \mathrm{U}_{\mathrm{MEL}} \vdash ((\overline{\Omega}, \overline{E}) \mid\!- \bar{t} = \bar{t}' \texttt{ if } \overline{A_1 \wedge \ldots \wedge A_n}) = \texttt{true} &\end{aligned}$$

La segunda implicación es consecuencia del teorema 3.2 y la tercera resulta, por las proposiciones 3.9, 3.10, 3.7 y 3.8, de aplicar **(Reemplazamiento)** con la primera de las ecuaciones `introduccion de implicacion`. □

Para demostrar la dirección ($\Leftarrow$) del teorema principal necesitamos el siguiente lema auxiliar. En esencia, nos va a permitir trabajar con derivaciones "normalizadas" de profundidad mínima y descartar aquellas otras espurias que aparecen utilizando la regla de **(Transitividad)**. De forma más precisa, obsérvese que una vez se tengan derivaciones de las ecuaciones $\overline{T_1 \vdash t_1 = t_1'} = \texttt{true}$ y $\overline{T_2 \vdash t_2 = t_2'} = \texttt{true}$, se puede utilizar **(Transitividad)** para obtener primero $\overline{T_1 \vdash t_1 = t_1'} = \overline{T_2 \vdash t_2 = t_2'}$ y a continuación combinarlas para conseguir un número infinito de derivaciones de las mismas ecuaciones; son precisamente estas últimas las que no queremos considerar.

Dada una derivación δ en MEL de una ecuación o afirmación de pertenencia, denotamos mediante *prof*(δ) su *profundidad*, definida de la forma habitual como la profundidad del árbol correspondiente.

Lema 3.1 *Para cualesquiera términos t_1, t_2 y fórmulas atómicas $A_1, \ldots, A_n$ sobre una teoría T en lógica de pertenencia, y términos w_1, w_2, w_3 y w_4 en $\mathrm{U}_{\mathrm{MEL}}$ tales que $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_1} = w_1$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_2} = w_2$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{A_1 \wedge \ldots \wedge A_n} = w_3$ y $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T} = w_4$, siempre que se tenga una derivación δ en $\mathrm{U}_{\mathrm{MEL}}$ de la ecuación*

$$(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = m$$

o de su simétrica

$$m = (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3)$$

para algún término m sobre la signatura de $\mathrm{U}_{\mathrm{MEL}}$, se cumple alguna de las siguientes alternativas:

1. *m es $w_4' \mid\!- w_1' = w_2' \texttt{ if } w_3'$, para términos w_1', w_2', w_3' y w_4' tales que $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_1} = w_1'$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_2} = w_2'$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{A_1 \wedge \ldots \wedge A_n} = w_3'$ y $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T} = w_4'$; o*
2. *existe una derivación δ' de la ecuación $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}$ o de $\texttt{true} = (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3)$, tal que prof($\delta'$) $\leq$ prof(δ).*

Un resultado análogo también se cumple cuando $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3)$ se sustituye por $(w_4 \mid\!- w_1 : w_2 \texttt{ if } w_3)$.

Demostración. Por inducción estructural sobre la derivación δ; consideramos la última regla de deducción utilizada.

- **(Reflexividad)**. Debe cumplirse (1).
- **(Simetría)**. El resultado se sigue de la hipótesis de inducción.
- **(Pertenencia)**. No se puede dar.
- **(Reemplazamiento)**. Tiene que ocurrir que o bien:
  - una de las ecuaciones correspondientes a las reglas de deducción en MEL (figura 3.1) haya sido utilizada, en cuyo caso m es `true` y se tiene (2) tomando como derivación δ' la propia δ; o
  - una ecuación haya sido aplicada sobre w_1, w_2, w_3 o w_4 (o sobre alguno de sus subtérminos, a través de un operador como pudiera ser `vars` o `applyS`), y se cumple (1).
- **(Transitividad)**. Supongamos que δ demuestra $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = m$ (el otro caso es análogo). Existe un término n tal que

$$\frac{\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = n \qquad \mathrm{U}_{\mathrm{MEL}} \vdash n = m}{\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = m}\,.$$

  Aplicando la hipótesis de inducción a la derivación δ' de $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = n$, tenemos una de las siguientes posibilidades:

  - n es $w_4' \mid\!- w_1' = w_2' \texttt{ if } w_3'$, con derivaciones de $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_1} = w_1'$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_2} = w_2'$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{A_1 \wedge \ldots \wedge A_n} = w_3'$ y $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T} = w_4'$. En este caso aplicamos la hipótesis de inducción a la derivación δ'' de $n = m$ y distinguimos los siguientes casos:
    - m es $w_4'' \mid\!- w_1'' = w_2'' \texttt{ if } w_3''$, con $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_1} = w_1''$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_2} = w_2''$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{A_1 \wedge \ldots \wedge A_n} = w_3''$ y $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T} = w''$, y se cumple (1).
    - Hay una derivación δ''' de $n = \texttt{true}$ tal que $\mathit{prof}(\delta''') \leq \mathit{prof}(\delta'')$. Pero entonces

$$\frac{\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = n \qquad \mathrm{U}_{\mathrm{MEL}} \vdash n = \texttt{true}}{\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}}$$

      es una derivación de $\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}$ cuya profundidad es igual a $1 + \max(\mathit{prof}(\delta'), \mathit{prof}(\delta''')) \leq 1 + \max(\mathit{prof}(\delta'), \mathit{prof}(\delta'')) = \mathit{prof}(\delta)$, y tenemos (2).
  - Existe una derivación de $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}$ cuya profundidad es menor o igual que la de δ' y por lo tanto menor que la de δ, y se cumple (2). □

Teorema 3.3 *Para cualesquiera términos t_1 y t_2 y fórmulas atómicas $A_1, \ldots, A_n$ sobre una teoría T en la lógica de pertenencia, y términos w_1, w_2, w_3 y w_4 en $\mathrm{U}_{\mathrm{MEL}}$ tales que $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_1} = w_1$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_2} = w_2$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{A_1 \wedge \ldots \wedge A_n} = w_3$ y $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T} = w_4$, si se tiene que*

$$\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}$$

o

$$\mathrm{U}_{\mathrm{MEL}} \vdash \texttt{true} = (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3)\,,$$

entonces, si $n > 0$

$$T \vdash t_1 = t_2 \textbf{ if } A_1 \wedge \ldots \wedge A_n \,,$$

y en caso contrario,

$$T \vdash t_1 = t_2 \,.$$

Un resultado análogo se tiene también para afirmaciones de pertenencia.

Demostración. Por inducción sobre la profundidad de la derivación. Según la última regla de derivación utilizada, tenemos:

- **(Reflexividad)** y **(Pertenencia)**. No se pueden dar.
- **(Simetría)**. Por la hipótesis de inducción.
- **(Transitividad)**. Supongamos que tenemos $\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}$ (los otros casos son análogos). Hay un término m sobre la signatura de $\mathrm{U}_{\mathrm{MEL}}$ tal que

$$\frac{\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = m \qquad \mathrm{U}_{\mathrm{MEL}} \vdash m = \texttt{true}}{\mathrm{U}_{\mathrm{MEL}} \vdash (w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}} \,.$$

  Por el lema 3.1 aplicado a la derivación δ de $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = m$ tenemos una de las siguientes alternativas:

  1. m es $w'_4 \mid\!- w'_1 = w'_2 \texttt{ if } w'_3$, con derivaciones de $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_1} = w'_1$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{t_2} = w'_2$, $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{A_1 \wedge \ldots \wedge A_n} = w'_3$ y $\mathrm{U}_{\mathrm{MEL}} \vdash \overline{T} = w'_4$: el resultado se obtiene aplicando la hipótesis de inducción a la derivación de $m = \texttt{true}$.
  2. Hay una derivación de $(w_4 \mid\!- w_1 = w_2 \texttt{ if } w_3) = \texttt{true}$ cuya profundidad es menor o igual que la de δ y por tanto menor que la de la derivación original, y aplicamos de nuevo la hipótesis de inducción.

- **(Reemplazamiento)**. Nótese que el contexto C con el que se ha debido utilizar la regla necesariamente debe haber sido el vacío, por lo que las únicas ecuaciones que se han podido aplicar son las correspondientes a las reglas de deducción de MEL en la figura 3.1. Consideremos cada caso por separado:
  - `reflexividad`. Trivial.
  - `reemplazamiento`. Por la proposición 3.6 hay un axioma $t = t'$ **if** *Cond* (resp. $t : s$ **if** *Cond*) en E tal que todas las fórmulas atómicas en $\sigma(\mathit{Cond})$ se pueden probar en T. En este caso, el término w_1 es $\texttt{applyC}(\overline{C}, \texttt{applyS}(\overline{\sigma}, \overline{t}))$, w_2 es $\texttt{applyC}(\overline{C}, \texttt{applyS}(\overline{\sigma}, \overline{t'}))$ y w_3 es `none`, y, por las proposiciones 3.3 y 3.4, t_1 es $C[\sigma(t)]$ y t_2 es $C[\sigma(t')]$. Pero entonces $T \vdash t_1 = t_2$ se tiene por **(Reemplazamiento)**.
  - `simetria`. Por la hipótesis de inducción y **(Simetría)**.
  - `transitividad`. El resultado se sigue de la proposición 3.5, la hipótesis de inducción y **(Transitividad)**.
  - `pertenencia`. El resultado se sigue de la proposición 3.5, la hipótesis de inducción y **(Pertenencia)**.

– `introduccion de implicacion`. Por las proposiciones 3.9, 3.10, 3.7 y 3.8, la hipótesis de inducción e **(Introducción de implicación)**. □

Como corolario de los resultados sobre reflexión que hemos demostrado, vamos a mostrar en las siguientes dos secciones la naturaleza reflexiva de otras dos lógicas relacionadas: la lógica ecuacional heterogénea y la lógica de Horn con igualdad.

3.3 Reflexión en la lógica ecuacional heterogénea

La lógica ecuacional heterogénea, a la que también nos referiremos como MSEL por sus siglas en inglés (de many-sorted equational logic), es una sublógica de la lógica de pertenencia, concretamente la sublógica que se obtiene al hacer que el conjunto de tipos sea vacío (y renombrando "familia" como "tipo"); en particular, para toda teoría T en MSEL y sentencias ϕ sobre la signatura de T, se cumple que $T \vdash_{\text{MSEL}} \phi \iff T \vdash_{\text{MEL}} \phi$. Pero entonces, como en la definición de U_{MEL} solo hemos utilizado familias y ecuaciones condicionales en las que no intervienen afirmaciones de pertenencia y estas últimas tampoco son introducidas por la función de representación, tenemos que U_{MEL} es una teoría en la lógica MSEL, que por lo tanto resulta ser reflexiva.

Teorema 3.4 U_{MEL} *es una teoría universal en* MSEL *para la clase de teorías finitamente presentables que no tienen tipos vacíos.*

Demostración. Para todas las teorías T finitamente presentables en MSEL con tipos no vacíos y sentencias ϕ sobre la signatura de T, como por definición $\overline{T \vdash \phi}$ es una sentencia en MSEL, se tiene

$$T \vdash_{\text{MSEL}} \phi \iff T \vdash_{\text{MEL}} \phi \iff \mathrm{U}_{\text{MEL}} \vdash_{\text{MEL}} \overline{T \vdash \phi} \iff \mathrm{U}_{\text{MEL}} \vdash_{\text{MSEL}} \overline{T \vdash \phi}\,.$$

□

3.4 Reflexión en la lógica de Horn con igualdad

En Meseguer (1998) se demuestra que la lógica de pertenencia es equivalente a la lógica de Horn heterogénea con igualdad, o $\text{MSHORN}^{=}$. No resulta sorprendente, entonces, que los resultados reflexivos sobre MEL se puedan traducir de manera inmediata a $\text{MSHORN}^{=}$.

Una signatura en $\text{MSHORN}^{=}$ es una terna (L, Σ, Π), donde L es un conjunto de tipos, $\Sigma = \{\Sigma_{w,l}\}_{(w,l)\in L^*\times L}$ es una colección de conjuntos de símbolos de función y $\Pi = \{\Pi_w\}_{w\in L^*}$ es una colección de conjuntos de símbolos de predicados. Entonces una signatura $\Omega = (K, \Sigma, S)$ en MEL se puede transformar en una signatura $\Omega^\sharp = (K, \Sigma, S^\sharp)$ en $\text{MSHORN}^{=}$ haciendo que cada conjunto $S^\sharp_k$ sea igual a S_k para cada familia $k \in K$ y que $S^\sharp_w$ sea el conjunto vacío para todo $w \in K^* \setminus K$. De esta manera, si adoptamos una notación postfija $\_ : s$ para cada predicado en $\Omega^\sharp$ que corresponde a un tipo s en Ω, cada sentencia sobre Ω en MEL puede ser interpretada como una sentencia sobre $\Omega^\sharp$ en $\text{MSHORN}^{=}$. Escribiremos $(\Omega, E)^\sharp$ para $(\Omega^\sharp, E)$. Se tiene entonces, para toda sentencia ϕ sobre Σ, que $(\Omega, E) \vdash_{\text{MEL}} \phi \iff (\Omega, E)^\sharp \vdash_{\text{MSHORN}^{=}} \phi$.

Además, en Meseguer (1998) también se define una traducción de MSHORN$^=$ en MEL. Una signatura (L, Σ, Π) en MSHORN$^=$ se aplica en una teoría $J(L, \Sigma, \Pi)$ en MEL cuya signatura está formada por:

- un conjunto de familias $K = L \uplus \{p(w) \mid w \in L^* \setminus L \text{ y } \Pi_w \neq \emptyset\}$;
- para cada clase $k \in K$, el conjunto de tipos S_k es Π_k si $k \in L$ o Π_w si k es $p(w)$;
- un conjunto de operadores

$$\begin{aligned}\Delta = \Sigma \cup \{\langle \_, \ldots, \_ \rangle : l_1 \ldots l_n \longrightarrow p(l_1, \ldots, l_n) \mid p(l_1, \ldots, l_n) \in K \setminus L\} \\ \cup \{\pi_i : p(l_1, \ldots, l_n) \longrightarrow l_i \mid 1 \leq i \leq n, p(l_1, \ldots, l_n) \in K \setminus L\}\,.\end{aligned}$$

La idea es representar el producto cartesiano de las familias $l_1, \ldots, l_n$ mediante la familia $p(l_1, \ldots, l_n)$. Para ello, los axiomas de la teoría $J(L, \Sigma, \Pi)$ son

$$\begin{gathered}(\forall x_1 : l_1, \ldots, x_n : l_n)\ \pi_i(\langle x_1, \ldots, x_n \rangle) = x_i \qquad (1 \leq i \leq n) \\ (\forall y : p(l_1, \ldots, l_n))\ y = \langle \pi_1(y), \ldots, \pi_n(y) \rangle\end{gathered}$$

para cada $p(l_1, \ldots, l_n)$.

La traducción α de sentencias deja sin variar toda ecuación $t = t'$ y envía cada fórmula atómica $P(t_1, \ldots, t_n)$ al axioma de pertenencia $\langle t_1, \ldots, t_n \rangle : P$ y cada cláusula de Horn

$$(\forall X)\, at \Leftarrow u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge P_1(\overline{w_1}) \wedge \ldots \wedge P_m(\overline{w_m})$$

a la sentencia

$$(\forall X)\, \alpha(at) \textbf{ if } u_1 = v_1 \wedge \ldots \wedge u_n = v_n \wedge \langle \overline{w_1} \rangle : P_1 \wedge \ldots \wedge \langle \overline{w_m} \rangle : P_m\,.$$

Una teoría $T = (L, \Sigma, \Pi, \Gamma)$ en MSHORN$^=$ se traduce entonces a la teoría $J(T)$ que resulta de añadir a $J(L, \Sigma, \Pi)$ la traducción de las cláusulas en Γ. Se cumple entonces $T \vdash_{\text{MSHORN}^=} \phi \iff J(T) \vdash_{\text{MEL}} \alpha(\phi)$ y tenemos el siguiente resultado.

Teorema 3.5 $\mathrm{U}^\sharp_{\text{MEL}}$ *es una teoría universal en* MSHORN$^=$ *para la clase de teorías finitamente presentables con tipos no vacíos.*

Demostración. Para toda teoría finitamente presentable T con tipos no vacíos en MSHORN$^=$ y sentencias ϕ sobre T,

$$\begin{aligned}T \vdash_{\text{MSHORN}^=} \phi \quad &\iff \quad J(T) \vdash_{\text{MEL}} \alpha(\phi) \\ &\iff \quad \mathrm{U}_{\text{MEL}} \vdash_{\text{MEL}} \overline{J(T) \vdash_{\text{MEL}} \alpha(\phi)} \\ &\iff \quad \mathrm{U}^\sharp_{\text{MEL}} \vdash_{\text{MSHORN}^=} \overline{J(T) \vdash_{\text{MEL}} \alpha(\phi)}\,.\end{aligned}$$

□

3.5 Reflexión en la lógica de reescritura

3.5.1 Ajustes en el cálculo de derivación

De nuevo, tal y como hicimos para la lógica de pertenencia, eliminamos la regla **(Congruencia)**, que pasa a tenerse en cuenta en una nueva versión de la regla **(Reemplazamiento)**.

- **Reemplazamiento**. Para cada regla de reescritura $(t \longrightarrow t' \textbf{ if } C_{mb} \wedge C_{eq} \wedge C_{rl})$ en R, con t, t' en la familia k, contexto $C^k \in \mathrm{C}^{\iota}_{\Sigma}(X)$ y sustitución σ, donde $C_{mb} \triangleq (u_1 : s_1 \wedge \ldots \wedge u_j : s_j)$, $C_{eq} \triangleq (v_1 = v'_1 \wedge \ldots \wedge v_k = v'_k)$ y $C_{rl} \triangleq (w_1 \longrightarrow w'_1 \wedge \ldots \wedge w_h \longrightarrow w'_h)$,

$$\frac{\begin{array}{ccc}(\Omega, E) \vdash \sigma(u_1) : s_1 & \cdots & (\Omega, E) \vdash \sigma(u_j) : s_j \\ (\Omega, E) \vdash \sigma(v_1) = \sigma(v'_1) & \cdots & (\Omega, E) \vdash \sigma(v_k) = \sigma(v'_k) \\ \sigma(w_1) \longrightarrow \sigma(w'_1) & \cdots & \sigma(w_h) \longrightarrow \sigma(w'_h)\end{array}}{C[\sigma(t)] \longrightarrow C[\sigma(t')]}\,.$$

3.5.2 Una teoría universal para RL

Presentamos aquí la teoría universal U_{RL} y una función de representación $\overline{\_ \vdash \_}$ que codifica pares formados por una teoría de reescritura $\mathcal{R}$ y una sentencia sobre su signatura como una sentencia en U_{RL}, y demostramos la universalidad de U_{RL}. La observación fundamental es que U_{RL} es una extensión de $\mathrm{U}_{\mathrm{MEL}}$, con lo que podemos utilizar la universalidad de $\mathrm{U}_{\mathrm{MEL}}$ en la demostración de la universalidad de U_{RL}. En lo que sigue trabajaremos con teorías finitamente presentables en RL.

La signatura de U_{RL}

La signatura de la teoría U_{RL} es una extensión de la signatura de $\mathrm{U}_{\mathrm{MEL}}$. Para representar reglas (posiblemente condicionales), la signatura de U_{RL} incluye además los constructores:

```
op _=>_ : [Term] [Term] -> [AtomR] .
op noneR : -> [RuleCondition] .
op _/\_ : [Atom] [RuleCondition] -> [RuleCondition] .
op _/\_ : [AtomR] [RuleCondition] -> [RuleCondition] .
op _=>_if_ : [Term] [Term] [RuleCondition] -> [Rule] .
```

Las teorías se representan utilizando:

```
op (_,_,_) : [Signature] [AxiomSet] [RuleSet] -> [RLTheory] .
```

Por último, la signatura de U_{RL} contiene un operador booleano

```
op _|-_ : [RLTheory] [AtomR] -> [Bool] .
```

para representar la derivabilidad de sentencias en una teoría de reescritura dada; los axiomas principales de U_{RL} definen este operador.

Como ocurría con U_{MEL}, tenemos también una serie de operadores auxiliares. En primer lugar tenemos constructores para conjuntos de reglas de reescritura

```
op emptyR : -> [RuleSet] .
op unionR : [Rule] [RuleSet] -> [RuleSet] .
```

así como operadores para comprobar si una condición se satisface en una teoría bajo una sustitución dada:

```
op satisfyRC : [RLTheory] [RuleCondition] [Substitution] -> [Bool] .
op satisfyR : [RLTheory] [Rule] [Substitution] -> [Bool] .
```

La función de representación

A continuación definimos la función de representación $\overline{\_ \vdash \_}$. Para toda teoría de reescritura $\mathcal{R}$ finitamente presentable con familias no vacías y sentencias $t \longrightarrow t'$ sobre la signatura de $\mathcal{R}$,

$$\overline{\mathcal{R} \vdash t \longrightarrow t'} \triangleq (\overline{\mathcal{R}} \mid\!- \bar{t} \Rightarrow \bar{t}') \longrightarrow \texttt{true}\,.$$

donde $\overline{(\_)}$ es una extensión de la función de representación para U_{MEL}, que se define de la manera esperada.

1. Para una teoría de reescritura

$$\overline{(\Omega, E, R)} \triangleq (\overline{\Omega}, \overline{E}, \overline{R})\,.$$

2. Para un conjunto de reglas de reescritura,

$$\overline{\{r_1, \ldots, r_n\}} \triangleq \texttt{consR}(\overline{r_1}, \ldots, \texttt{consR}(\overline{r_n}, \texttt{emptyR}) \ldots)\,.$$

3. Para una reescritura,

$$\overline{t \longrightarrow t'} \triangleq \bar{t} \Rightarrow \bar{t}'\,.$$

4. Para una conjunción de ecuaciones, afirmaciones de pertenencia y reescrituras,

$$\overline{A_1 \wedge A_2 \wedge \ldots \wedge A_n} \triangleq \overline{A_1} \,\texttt{/\textbackslash}\, (\overline{A_2} \,\texttt{/\textbackslash}\, \ldots \,\texttt{/\textbackslash}\, (\overline{A_n} \,\texttt{/\textbackslash}\, \texttt{noneR}) \ldots)\,.$$

5. Para una regla de reescritura,

$$\overline{t \longrightarrow t' \textbf{ if } A_1 \wedge \ldots \wedge A_n} \triangleq \bar{t} \;\Rightarrow\; \bar{t}' \;\texttt{ if }\; \overline{A_1 \wedge \ldots \wedge A_n}\,.$$

Los axiomas de U_{RL}

Finalmente definimos los axiomas de U_{RL}, que incluyen reglas que corresponden a las reglas de inferencia de RL y todas las ecuaciones en $\mathrm{U}_{\mathrm{MEL}}$. Los comentarios de la página 38 sobre la notación a emplear siguen siendo válidos.

En lo que sigue asumimos una teoría de reescritura finitamente presentable con tipos no vacíos, $\mathcal{R} = (\Omega, E, R)$, donde $\Omega = (K, \Sigma, S)$.

En primer lugar, necesitamos de nuevo ecuaciones entre los constructores de conjuntos.

```
eq unionR(r̄, unionR(r̄',ĀS)) = unionR(r̄', unionR(r̄,ĀS)) .
eq unionR(r̄, unionR(r̄,ĀS)) = unionR(r̄,ĀS) .
```

Y análogamente para el constructor _/_ de condiciones.

Las ecuaciones para `satisfyRC`, como las de `satisfyC`, simplemente extraen los componentes de una condición.

```
eq satisfyRC(R̄,none,σ̄) = true .
eq satisfyRC(R̄,Ā /\ C̄,σ̄) = satisfyR(R̄,Ā,σ̄) and satisfyRC(R̄,C̄,σ̄) .

eq satisfyR(R̄,t̄ => t̄',σ̄) = (R̄ |- applyS(σ̄,t̄) => applyS(σ̄,t̄')) .
eq satisfyR(R̄,t̄ = t̄',σ̄) = (R̄ |- applyS(σ̄,t̄) = applyS(σ̄,t̄') if none) .
eq satisfyR(R̄,t̄ : s̄,σ̄) = (R̄ |- applyS(σ̄,t̄) : s̄ if none) .
```

Proposición 3.11 *Para cualesquiera fórmulas atómicas y reescrituras $A_1, \ldots, A_n$, y sustituciones σ, son equivalentes:*

1. $\mathrm{U}_{\mathrm{RL}} \vdash \overline{\mathcal{R} \vdash \sigma(A_i)}$ *para cada i; esto es, para cada A_i de la forma $t_i \longrightarrow t_i'$,*

$$\mathrm{U}_{\mathrm{RL}} \vdash (\overline{\mathcal{R}} \mid\!- \overline{\sigma(t_i)} \Rightarrow \overline{\sigma(t_i')}) \longrightarrow \mathtt{true}\,,$$

para cada A_i de la forma $t_i = t_i'$,

$$\mathrm{U}_{\mathrm{RL}} \vdash (\overline{\mathcal{R}} \mid\!- \overline{\sigma(t_i)} = \overline{\sigma(t_i')}\ \mathtt{if\ none}) = \mathtt{true}\,,$$

y para cada A_i de la forma $t_i : s_i$,

$$\mathrm{U}_{\mathrm{RL}} \vdash (\overline{\mathcal{R}} \mid\!- \overline{\sigma(t_i)} : \overline{s_i}\ \mathtt{if\ none}) = \mathtt{true}\,.$$

2. *Se cumple que*

$$\mathrm{U}_{\mathrm{RL}} \vdash \mathtt{satisfyRC}(\overline{\mathcal{R}}, \overline{A_1 \wedge \ldots \wedge A_n}, \overline{\sigma}) = \mathtt{true}\,.$$

Por último, las reglas de derivación del cálculo de la lógica de reescritura se especifican tal y como aparecen en la figura 3.2.

```
including U_MEL .
*** reflexividad
rl ((Ω̄,Ē,R̄) |- t̄ => t̄) => true.

*** reemplazamiento
crl ((Ω̄,Ē,R̄) |- applyC(C̄, applyS(σ̄,t̄)) => applyC(C̄, applyS(σ̄,t̄'))) => true
    if R̄ = unionR(t̄ = t̄' if Cond,R̄)
    /\ satisfyR((Ω̄,Ē,R̄),Cond,σ̄) = true .

*** transitividad
crl ((Ω̄,Ē,R̄) |- t̄ => t̄') => true
    if parse(t̄'',Ω̄) = true
    /\ ((Ω̄,Ē,R̄) |- t̄ => t̄'') => true
    /\ ((Ω̄,Ē,R̄) |- t̄'' => t̄') => true .

*** igualdad
crl ((Ω̄,Ē,R̄) |- ū => ū') => true
    if parse(t̄,Ω̄) = true
    /\ parse(t̄',Ω̄) = true
    /\ ((Ω̄,Ē) |- t̄ = ū if none) = true
    /\ ((Ω̄,Ē) |- t̄' = ū' if none) = true
    /\ ((Ω̄,Ē,R̄) |- t̄ => t̄') => true .
```

Figura 3.2: La teoría universal U_{RL} (fragmento).

3.5.3 La corrección de la teoría universal RL

Tenemos ya todos los ingredientes necesarios para probar la corrección de la teoría universal U_{RL}. La demostración es análoga a la del teorema 3.1, haciendo uso en determinados pasos del hecho de que U_{RL} es una extensión de $\mathrm{U}_{\mathrm{MEL}}$, por lo que no incluimos todos los detalles y nos concentramos en los casos más complicados o interesantes.

Como ya ocurría en la sección anterior, son necesarios algunos lemas auxiliares antes de pasar a demostrar el resultado principal. En las demostraciones que siguen escribiremos $\mathrm{U}_{\mathrm{RL}} \vdash w = w'$ para denotar que la ecuación $w = w'$ se puede derivar en la teoría ecuacional de U_{RL}.

Lema 3.2 *Para cualesquiera términos w, w_1, w_2 y w_3 en U_{RL} tales que*

$$\mathrm{U}_{\mathrm{RL}} \vdash w = (w_3 \mid\!- w_1 \Rightarrow w_2) \quad o \quad \mathrm{U}_{\mathrm{RL}} \vdash (w_3 \mid\!- w_1 \Rightarrow w_2) = w\,,$$

se tiene que w es de la forma $w'_3 \mid\!- w'_1 \Rightarrow w'_2$ para términos w'_1, w'_2 y w'_3 tales que se verifica $\mathrm{U}_{\mathrm{RL}} \vdash w_i = w'_i$, $1 \leq i \leq 3$.

Demostración. Por inducción estructural sobre la derivación. En el caso de la regla **(Reemplazamiento)** basta con darse cuenta de que no hay ecuaciones que se apliquen al operador $\_ \mid\!- \_ \Rightarrow \_$ que representa la derivabilidad; los demás casos son inmediatos. □

El siguiente lema es análogo al lema 3.1.

Lema 3.3 *Para cualesquiera términos t_1, t_2 sobre una teoría $\mathcal{R}$ en la lógica de reescritura, y términos w_1, w_2 y w_3 en U_{RL} tales que $\mathrm{U}_{\mathrm{RL}} \vdash \overline{t_1} = w_1$, $\mathrm{U}_{\mathrm{RL}} \vdash \overline{t_2} = w_2$ y $\mathrm{U}_{\mathrm{RL}} \vdash \overline{\mathcal{R}} = w_3$, siempre que se tenga una derivación δ en U_{RL} de la reescritura*

$$(w_3 \mid\!- w_1 \Rightarrow w_2) \longrightarrow m$$

para algún término m sobre la signatura de U_{RL}, se cumple alguna de las siguientes alternativas:

1. *m es $w'_3 \mid\!- w'_1 \Rightarrow w'_2$, para términos w'_1, w'_2 y w'_3 tales que $\mathrm{U}_{\mathrm{RL}} \vdash \overline{t_1} = w'_1$, $\mathrm{U}_{\mathrm{RL}} \vdash \overline{t_2} = w'_2$ y $\mathrm{U}_{\mathrm{RL}} \vdash \overline{\mathcal{R}} = w'_3$; o*
2. *existe una derivación δ' de la reescritura $(w_3 \mid\!- w_1 \Rightarrow w_2) \longrightarrow \mathtt{true}$ tal que $prof(\delta') \leq prof(\delta)$.*

Demostración. Como en el lema 3.1, la demostración procede por inducción estructural sobre la derivación. El único caso que se trata de forma distinta a los allí descritos es el correspondiente a la regla **(Igualdad)**; supongamos entonces que el último paso en la derivación es

$$\frac{\mathrm{U}_{\mathrm{RL}} \vdash t = (w_3 \mid\!- w_1 \Rightarrow w_2) \qquad \mathrm{U}_{\mathrm{RL}} \vdash t' = u' \qquad t \longrightarrow t'}{(w_3 \mid\!- w_1 \Rightarrow w_2) \longrightarrow u'}\,.$$

Por el lema 3.2, t es de la forma $w'_3 \mid\!- w'_1 \Rightarrow w'_2$, con $\mathrm{U}_{\mathrm{RL}} \vdash w_i = w'_i$, $1 \leq i \leq 3$. Aplicando ahora la hipótesis de inducción a la derivación de $(w'_3 \mid\!- w'_1 \Rightarrow w'_2) \longrightarrow t'$, tenemos una de las siguientes posibilidades:

- t' es de la forma $w''_3 \mid\!- w''_1 \Rightarrow w''_2$, con $\mathrm{U}_{\mathrm{RL}} \vdash \overline{t_1} = w''_1$, $\mathrm{U}_{\mathrm{RL}} \vdash \overline{t_2} = w''_2$ y $\mathrm{U}_{\mathrm{RL}} \vdash \overline{\mathcal{R}} = w''_3$. Pero, de nuevo por el lema 3.2, u' es de la misma forma y tenemos que se cumple (1).
- Hay una derivación de $(w'_3 \mid\!- w'_1 \Rightarrow w'_2) \longrightarrow \mathtt{true}$ cuya profundidad es menor o igual que la de $(w'_3 \mid\!- w'_1 \Rightarrow w'_2) \longrightarrow t'$. Podemos modificar entonces la derivación original de $(w_3 \mid\!- w_1 \Rightarrow w_2) \longrightarrow u'$ utilizando esta derivación y reemplazando el término u' por `true` y así obtenemos una derivación que satisface (2). □

Ahora el teorema principal se demuestra imitando la prueba del teorema 3.1.

Teorema 3.6 *Para toda teoría de reescritura finitamente presentable y con familias no vacías $\mathcal{R} = (\Omega, E, R)$, con $\Omega = (K, \Sigma, S)$, y términos t, t' en $T_\Sigma(X)$,*

$$\mathcal{R} \vdash t \longrightarrow t' \iff \mathrm{U}_{\mathrm{RL}} \vdash (\overline{\mathcal{R}} \mid\!- \overline{t} \Rightarrow \overline{t'}) \longrightarrow \mathtt{true}\,.$$

Demostración. La demostración de la implicación ($\Rightarrow$) sigue los pasos de la del teorema 3.2: se procede por inducción estructural sobra la derivación de $\mathcal{R} \vdash t \longrightarrow t'$ y el único caso nuevo que aparece es el correspondiente a la regla **(Igualdad)**. Supongamos entonces que el último paso de la derivación tiene la forma

$$\frac{(\Omega, E) \vdash t = u \qquad (\Omega, E) \vdash t' = u' \qquad t \longrightarrow t'}{u \longrightarrow u'}\,.$$

Como $\mathrm{U}_{\mathrm{MEL}}$ es universal tenemos $\mathrm{U}_{\mathrm{MEL}} \vdash ((\overline{\Omega}, \overline{E}) \mid\!- \overline{t} = \overline{u}\ \texttt{if none}) = \texttt{true}$ y $\mathrm{U}_{\mathrm{MEL}} \vdash ((\overline{\Omega}, \overline{E}) \mid\!- \overline{t'} = \overline{u}'\ \texttt{if none}) = \texttt{true}$ y por la hipótesis de inducción, $\mathrm{U}_{\mathrm{RL}} \vdash (\overline{\mathcal{R}} \mid\!- \overline{t} \Rightarrow \overline{t'}) \longrightarrow \texttt{true}$; como U_{RL} contiene a $\mathrm{U}_{\mathrm{MEL}}$, en virtud de la proposición 3.5 podemos aplicar **(Reemplazamiento)** con la regla `igualdad` para obtener $\mathrm{U}_{\mathrm{RL}} \vdash (\overline{\mathcal{R}} \mid\!- \overline{u} \Rightarrow \overline{u}') \longrightarrow \texttt{true}$.

La implicación ($\Leftarrow$) se sigue de una generalización análoga a la del teorema 3.3: para términos w_1, w_2 y w_3 sobre la signatura de U_{RL} tales que son ecuacionalmente iguales, respectivamente, a $\overline{t_1}$, $\overline{t_2}$ y $\overline{\mathcal{R}}$, si se tiene que

$$\mathrm{U}_{\mathrm{RL}} \vdash (w_3 \mid\!- w_1 \Rightarrow w_2) \longrightarrow \texttt{true}\,,$$

entonces

$$\mathcal{R} \vdash t_1 \longrightarrow t_2\,.$$

De nuevo la demostración sigue los pasos de la del teorema 3.3. El caso correspondiente a la regla **(Transitividad)** se apoya ahora en el lema 3.3 y el de la regla **(Reemplazamiento)** presenta un caso que no aparecía antes, que corresponde a la regla `igualdad`. En esta última situación el resultado se tiene por la proposición 3.5, la hipótesis de inducción y el propio teorema 3.3. Además, también hay que considerar el caso en el que la última regla aplicada ha sido **(Igualdad)**, en la forma

$$\frac{\mathrm{U}_{\mathrm{RL}} \vdash t = (w_3 \mid\!- w_1 \Rightarrow w_2) \qquad \mathrm{U}_{\mathrm{RL}} \vdash t' = \texttt{true} \qquad t \longrightarrow t'}{(w_3 \mid\!- w_1 \Rightarrow w_2) \longrightarrow \texttt{true}}\,.$$

Por el lema 3.2, t es de la forma $w_3' \mid\!- w_1' \Rightarrow w_2'$ con $\mathrm{U}_{\mathrm{RL}} \vdash w_i = w_i'$, $1 \leq i \leq 3$, y podemos aplicar el lema 3.3 a la derivación de $t \longrightarrow t'$ para distinguir los siguientes casos:

- t' es de la forma $w_3'' \mid\!- w_1'' \Rightarrow w_2''$. Esta situación no se puede dar por el lema 3.2, porque tenemos $\mathrm{U}_{\mathrm{RL}} \vdash t' = \texttt{true}$.
- Hay una derivación de $(w_3' \mid\!- w_1' \Rightarrow w_2') \longrightarrow \texttt{true}$ de profundidad menor o igual que la de $t \longrightarrow t'$ y por tanto menor que la de la derivación original, con lo que el resultado se obtiene de nuevo por la hipótesis de inducción. □

3.6 Una aplicación

El contenido de esta sección está basado en ideas propuestas en Basin et al. (2004) sobre metarrazonamiento formal utilizando reflexión, donde se discuten detalladamente las ventajas, inconvenientes y limitaciones de los marcos metalógicos reflexivos. Un marco metalógico, al igual que uno lógico, consiste en una lógica junto con una metodología asociada que se puede utilizar para representar otros sistemas formales. Sin embargo, mientras que en un marco lógico el énfasis se pone en razonar *en* una lógica, en un marco metalógico lo que se pretende es razonar *sobre* una lógica o incluso sobre relaciones entre lógicas distintas. La propuesta en Basin et al. (2004) consiste en que aquellos marcos lógicos que son reflexivos se pueden utilizar como marcos metalógicos reflejando al metanivel los principios inductivos de las lógicas formalizadas.

Aquí vamos a extender los principios de metarrazonamiento que allí se propusieron, lo que aumenta la clase de metateoremas a los que se pueden aplicar, y vamos a ilustrar su

uso con un ejemplo que muestra cómo utilizarlos para realizar metarrazonamiento sobre relaciones semánticas entre especificaciones ecuacionales. En este apartado utilizamos las palabras "teoría" y "especificación" como sinónimos.

3.6.1 Relaciones semánticas entre especificaciones

La formalización y prueba de ciertas relaciones semánticas entre especificaciones ecuacionales son aspectos importantes de la metodología de especificación algebraica. En este sentido una noción clásica es la de *enriquecimiento*, que es una herramienta conceptual básica en la metodología de especificación paso a paso—véase por ejemplo Ehrig y Mahr (1985) o Loeckx et al. (1996). Aquí consideramos la siguiente definición de la relación de enriquecimiento entre especificaciones en la lógica ecuacional de pertenencia.

Definición 3.4 *Sean* $T = (\Omega, E)$ *y* $T' = (\Omega', E')$ *especificaciones en la lógica de pertenencia, con* $\Omega = (K, \Sigma, S)$ *y* $\Omega' = (K', \Sigma', S')$ *tales que* $T \subseteq T'$ *componente a componente. Sean* k *una familia en* K *y* s *un tipo en* S_k*. Decimos que* T' *es un* s-enriquecimiento *de* T *si y solo si:*

1. *para todo* $t \in T_{\Omega'}$*, si* $T' \vdash t : s$ *entonces existe* $t' \in T_\Omega$ *tal que* $T \vdash t' : s$ *y* $T' \vdash t = t'$*;*
2. *para todo* $t, t' \in T_\Omega$*, si* $T \vdash t : s$*,* $T \vdash t' : s$ *y* $T' \vdash t = t'$ *entonces* $T \vdash t = t'$*.*

Nótese que nuestra definición es ligeramente diferente a la de Ehrig y Mahr (1985)—(1) y (2) corresponden, respectivamente, a sus nociones de *extensión completa* y *consistente*. En concreto, nosotros definimos la relación de enriquecimiento relativa a un tipo particular mientras que en Ehrig y Mahr (1985) se define simultáneamente sobre todos los tipos utilizando la noción de homomorfismo. La idea capturada por nuestra definición es que cada término cerrado en la especificación T' con tipo s puede probarse igual a un término cerrado en la especificación T con tipo s, y también que T' no impone nuevas igualdades entre términos cerrados de tipo s de la especificación T. Estas propiedades corresponden, en la terminología de Burstall y Goguen, a las propiedades *sin basura* y *sin confusión* (Burstall y Goguen, 1982).

La relación de enriquecimiento asume que una de las especificaciones está incluida en la otra. Hay otras relaciones semánticas, sin embargo, que no exigen tal inclusión. Considérese, por ejemplo, las especificaciones `INT1` e `INT2` presentadas en la figura 3.3 utilizando una extensión de la sintaxis de Maude en la que se hace explícita la declaración de familias junto con sus tipos. Las especificaciones `INT1` e `INT2` están claramente relacionadas ya que ambas especifican los números enteros—y, en ese sentido, son *intercambiables*—pero ninguna está incluida en la otra. Nosotros proponemos la siguiente definición para caracterizar esta relación entre especificaciones en lógica de pertenencia. Para simplificar la presentación restringimos nuestra definición a especificaciones que tengan en común el conjunto de las familias.

Definición 3.5 *Sean* $T = (\Omega, E)$ *y* $T' = (\Omega', E')$ *especificaciones con* $\Omega = (K, \Sigma, S)$ *y* $\Omega' = (K, \Sigma', S')$*. Sean* k *una familia en* K *y* s *un tipo en* $S_k \cap S'_k$*. Decimos que* T *y* T' *son* s-intercambiables *si y solo si:*

```
fmod INT1 is
  kind Num[Neg, Nat, Int] .
  op 0 : -> Num .
  op s : Num -> Num .
  op p : Num -> Num .
  var N : Num .

  --- Numeros no positivos
  mb 0 : Neg .
  cmb p(N) : Neg if N : Neg .

  --- Numeros naturales
  mb 0 : Nat .
  cmb s(N) : Nat if N : Nat .

  --- Enteros
  cmb N : Int if N : Neg .
  cmb N : Int if N : Nat .
endfm
```

```
fmod INT2 is
  kind Num[Int] .
  op 0 : -> Num .
  op s : Num -> Num .
  op p : Num -> Num .
  var N : Num .

  --- Enteros
  mb 0 : Int .
  cmb s(N) : Int if N : Int .
  cmb p(N) : Int if N : Int .
  eq p(s(N)) = N .
  eq s(p(N)) = N .
endfm
```

Figura 3.3: Las especificaciones `INT1` e `INT2`.

1. *para todo* $t \in T_\Omega$, *si* $T \vdash t : s$ *entonces existe* $t' \in T_{\Omega'}$ *tal que* $T' \vdash t' : s$ *y* $T \vdash t = t'$;
2. *para todo* $t, t' \in T_{\Omega'}$, *si* $T \vdash t : s$, $T' \vdash t' : s$ *y* $T \vdash t = t'$, *entonces* $T' \vdash t = t'$;
3. *para todo* $t \in T_{\Omega'}$, *si* $T' \vdash t : s$ *entonces existe* $t' \in T_\Omega$ *tal que* $T \vdash t' : s$ *y* $T' \vdash t = t'$;
4. *para todo* $t, t' \in T_\Omega$, *si* $T \vdash t : s$, $T \vdash t' : s$ *y* $T' \vdash t = t'$ *entonces* $T \vdash t = t'$.

La idea capturada por la definición anterior es que para cada término cerrado en la especificación T' (resp. T) con tipo s se puede demostrar que es igual a un término cerrado en la especificación T (resp. T') con tipo s, y también que T' (resp. T) no impone nuevas igualdades entre términos cerrados de tipo s en la especificación T (resp. T').

Nótese que para probar la propiedad (2) en la definición 3.4 y las propiedades (2) y (4) en la definición 3.5 necesitamos, en general, examinar la forma de los axiomas en T y T'. Pero para demostrar (1) en la definición 3.4 o (1) y (3) en la definición 3.5 podemos usar técnicas inductivas sobre los términos. Ciertamente, en Basin et al. (2004) se propone un principio inductivo de razonamiento para demostrar lógicamente la primera de estas. Sin embargo, la ausencia de una relación de inclusión entre T y T' invalida el uso de este principio para demostrar las otras dos propiedades.

En la sección 3.6.2 proponemos un principio inductivo de razonamiento (ind^+) para razonar metalógicamente sobre estas propiedades y en la sección 3.6.4 mostramos cómo este principio se puede transformar, utilizando reflexión, en un principio de razonamiento deductivo ($\overline{ind^+}$) para demostrarlas lógicamente. La diferencia entre ambos estriba en que (ind^+) es un principio, comparable al de inducción sobre los números naturales, que se puede utilizar para razonar matemáticamente sobre propiedades de conjuntos de

teorías en la lógica de pertenencia. Por el contrario, ($\overline{ind^+}$) es una sentencia en esta lógica que se puede utilizar en el cálculo de pruebas para derivar otras que metarrepresenten propiedades de dichos conjuntos de teorías.

3.6.2 Un principio inductivo para razonamiento metalógico

Para presentar una formulación y prueba de corrección más sencillas y compactas del principio inductivo (ind^+) que nos sirva para demostrar *metalógicamente* relaciones semánticas entre especificaciones ecuacionales, empezamos extendiendo las definiciones de término, fórmula atómica y relación de derivación para la lógica de pertenencia. Básicamente, estas extensiones nos permiten razonar sobre clases de equivalencia de términos en lugar de sobre términos individuales, lo que resulta esencial para razonar con familias de teorías que no están relacionadas por una relación de inclusión. Sin embargo, este cambio de marco lógico es transitorio. En la sección 3.6.4 mostramos cómo el principio inductivo (ind^+) se puede transformar, utilizando reflexión, en un principio inductivo ($\overline{ind^+}$) para demostrar *lógicamente*, en la lógica ecuacional de pertenencia estándar, relaciones semánticas entre especificaciones ecuacionales. Naturalmente, como nuestro objetivo final es obtener principios para realizar metarrazonamiento formal, estamos interesados en ($\overline{ind^+}$) más que en (ind^+), pero presentamos este último como una herramienta técnica para simplificar la presentación y demostración del primero.

En lo que sigue, $\mathcal{MT}$ será la clase de multiconjuntos finitos de teorías finitamente presentables

$$\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$$

con un conjunto común no vacío de familias de forma que $\Omega_i = (K, \Sigma_i, S_i)$ para $i \in [1..p]$. Consideramos multiconjuntos en lugar de listas o conjuntos por un motivo meramente técnico, pues ello simplifica nuestra definición del principio inductivo (ind^+).

Definición 3.6 *Dado $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]} \in \mathcal{MT}$, donde $\Omega_i = (K, \Sigma_i, S_i)$, definimos el conjunto $T^{\mathcal{T}}_{\Sigma_i}(X)$ de $(\Sigma_i, \mathcal{T})$-términos con variables en X como sigue:*

- $x \in T^{\mathcal{T}}_{\Sigma_i,k}(X)$ *si y solo si* $x \in X_k$, $k \in K$;
- $f(t_1, \ldots, t_n) \in T^{\mathcal{T}}_{\Sigma_i,k}(X)$ *si y solo si* $f \in (\Sigma_i)_{k_1\ldots k_n,k}$ *y* $t_j \in T^{\mathcal{T}}_{\Sigma_i,k_j}(X)$, *para* $j = 1, \ldots, n$,
- $[t]_{T_j} \in T^{\mathcal{T}}_{\Sigma_i,k}(X)$ *si y solo si* $T_j \in \mathcal{T}$ *y* $t \in T^{\mathcal{T}}_{\Sigma_j,k}(X)$.

Obsérvese que la definición es puramente sintáctica, en el sentido de que $[t]_{T_j}$ se corresponde con la aplicación a t de un operador más que podríamos haber denotado con notación prefija como $[]_j(t)$. Si hemos utilizado la notación $[t]$ reminiscente del concepto de clase de equivalencia es precisamente porque como se formaliza en la definición 3.8, el significado del término $[t]_{T_i}$, en el caso particular en que $t \in T_{\Sigma_i}$, es la clase de equivalencia de los términos que se pueden demostrar iguales a t en la teoría T_i. Por supuesto, en general puede ocurrir que el término t contenga a su vez términos de la forma $[t']_{T_j}$, en cuyo caso su significado deja de estar tan claro.

Definición 3.7 *Dado* $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]} \in \mathcal{MT}$, *donde* $\Omega_i = (K, \Sigma_i, S_i)$, *una* $(\Sigma_i, \mathcal{T})$*-fórmula atómica es o bien una ecuación* $t = t'$, *donde* t *y* t' *son* $(\Sigma_i, \mathcal{T})$*-términos de la misma familia, o una afirmación de pertenencia de la forma* $t : s$, *donde el* $(\Sigma_i, \mathcal{T})$*-término* t *pertenece a la familia* k *y* $s \in (S_i)_k$.

Definición 3.8 *Dado* $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]} \in \mathcal{MT}$, *donde* $\Omega_i = (K, \Sigma_i, S_i)$, *para todas las teorías* $T_i \in \mathcal{T}$ *y* $(\Sigma_i, \mathcal{T})$*-fórmulas atómicas* ϕ, *definimos recursivamente la relación de derivación* $\vdash^{\mathcal{T}}$ *como sigue:*

- *si hay una posición* p *en* ϕ *(para una definición apropiada de posiciones en fórmulas atómicas) y un término* $t \in T_{\Sigma_j,k}$, *con* $T_j \in \mathcal{T}$, *tal que* $[t]_{T_j}$ *ocupa la posición* p *en* ϕ, *entonces*

$$T_i \vdash^{\mathcal{T}} \phi \iff \text{existe } t' \in T_{\Sigma_i,k} \cap T_{\Sigma_j,k} \text{ tal que } T_i \vdash^{\mathcal{T}} \phi[t']_p \text{ y } T_j \vdash t = t' \,,$$

  donde $\phi[t']_p$ *resulta de reemplazar en* ϕ *el término en la posición* p *por* t'*;*

- *en caso contrario,* $T_i \vdash^{\mathcal{T}} \phi \iff T_i \vdash \phi$.

Según esta definición, una $(\Sigma_i, \mathcal{T})$-fórmula atómica ϕ es derivable en una teoría T_i si ϕ se puede derivar en T_i tras reemplazar recursivamente todas las apariciones de términos $[t]_{T_j}$, tales que $t \in T_{\Sigma_j}$, con términos cerrados adecuados en las clases de equivalencia correspondientes. Los resultados en la proposición siguiente son entonces inmediatos.

Proposición 3.12 *Dado* $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]} \in \mathcal{MT}$, *con* $\Omega_i = (K, \Sigma_i, S_i)$, *para todas las teorías* $T_i, T_j \in \mathcal{T}$, $(\Sigma_i, \mathcal{T})$*-fórmulas atómicas* $\phi(x)$ *con variable* x *de la familia* k, *y términos* $t, t' \in T_{\Sigma_i,k} \cap T_{\Sigma_j,k}$, *se cumplen los siguientes resultados:*

1. *si* $T_j \vdash t = t'$, *entonces* $T_i \vdash^{\mathcal{T}} \phi([t]_{T_j})$ *si y solo si* $T_i \vdash^{\mathcal{T}} \phi([t']_{T_j})$*;*
2. $T_i \vdash^{\mathcal{T}} \phi(t)$ *si y solo si* $T_i \vdash^{\mathcal{T}} \phi([t]_{T_i})$*;*
3. *si* $T_i \vdash^{\mathcal{T}} \phi(t)$, *entonces* $T_i \vdash^{\mathcal{T}} \phi([t]_{T_j})$*;*
4. *si* $E_j \subseteq E_i$, *entonces* $T_i \vdash^{\mathcal{T}} \phi(t)$ *si y solo si* $T_i \vdash^{\mathcal{T}} \phi([t]_{T_j})$*; y*
5. *si* E_j *no incluye ninguna ecuación, entonces* $T_i \vdash^{\mathcal{T}} \phi(t)$ *si y solo si* $T_i \vdash^{\mathcal{T}} \phi([t]_{T_j})$.

Demostración. (1) se sigue de la definición; (2) se tiene porque claramente $T_i^{\mathcal{T}} \vdash t = [t]_{T_i}$; (3) es cierto porque $T_j \vdash t = t$; (4) pues si $T_j \vdash t = t'$ entonces $T_i \vdash t = t'$; (5) se cumple porque en T_j el término t solo es igual a sí mismo. □

Por ejemplo, utilizando estas definiciones de términos, fórmulas atómicas y relación de derivación, podemos expresar de manera simple y compacta la propiedad (3) en la definición 3.5 de teorías intercambiables cuando T es `INT2` y T' es `INT1` mediante la siguiente afirmación metalógica:

$$\forall t \in T_{\texttt{INT2},\texttt{Num}}.(\texttt{INT2} \vdash t : \texttt{Int} \Longrightarrow \texttt{INT1} \vdash^{\mathcal{I}} [t]_{\texttt{INT2}} : \texttt{Int}) \,, \tag{3.1}$$

donde $\mathcal{I}$ es el multiconjunto $\{\texttt{INT1}, \texttt{INT2}\}$.

Decimos que se trata de una afirmación metalógica pues no se trata de una fórmula en la lógica de pertenencia o en alguna otra lógica, sino de una propiedad matemática que se cumple entre dos teorías lógicas, `INT1` e `INT2`. De hecho, para evitar confusiones, la propiedad se debería haber expresado sin utilizar símbolos lógicos en la siguiente forma:

> para todo término t en $T_{\mathtt{INT2,Num}}$,
> si existe una derivación de $\mathtt{INT2} \vdash t : \mathtt{Int}$
> entonces también existe otra de $\mathtt{INT1} \vdash^{\mathcal{I}} [t]_{\mathtt{INT2}} : \mathtt{Int}$).

En contextos como el presente es práctica común utilizar símbolos lógicos para abreviar sentencias como la anterior. En ocasiones el uso de tales convenciones puede dar lugar a situaciones en las que no esté claro si se está hablando del metanivel o del nivel objeto. Obviamente, la alternativa más segura sería eliminar los símbolos lógicos en todas las afirmaciones metalógicas y utilizar en su lugar las expresiones que abrevian, pero pensamos que el estilo resultante es recargado, por lo que preferimos limitarnos a poner al lector sobre aviso y dejar al contexto la tarea de resolver las posibles ambigüedades.

Ahora estamos preparados para dar, en la proposición 3.13 más abajo, un principio inductivo (ind^+) para demostrar metateoremas sobre multiconjuntos finitos de teorías $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ en $\mathcal{MT}$. Como el resultado en cuestión es bastante técnico, avanzamos primero informalmente su contenido.

- El principio inductivo (ind^+) se puede aplicar a enunciados metalógicos de la forma "para todo término t con tipo s en una teoría T_i de $\mathcal{T}$ en la lógica de pertenencia, alguna propiedad P se cumple". Aquí, P es una expresión booleana $bexp(B_1, \ldots, B_p)$, cuyas variables proposicionales se instancian con enunciados metalógicos de la forma: "una $(\Sigma_j, \mathcal{T})$-fórmula atómica $\phi([t]_{T_i})$ se cumple en T_j", con respecto a nuestra definición extendida de la relación de derivación. Por ejemplo, el enunciado metalógico (3.1) pertenece a esta clase de metateoremas para los que nuestro principio inductivo se puede aplicar.

- Los casos inductivos generados por (ind^+) se obtienen directamente de la definición inductiva del tipo s en la teoría T_i. Por lo tanto, nuestros casos inductivos reflejan los casos inductivos generados por el principio de inducción estructural habitual. Por ejemplo, los tres casos inductivos generados por (ind^+) cuando se aplica al enunciado (3.1) corresponden a los tres casos en la definición inductiva del tipo `Int` en `INT2`: concretamente, `0` es un `Int`; `s(`n`)` es un `Int` si n es un `Int`; y `p(`n`)` es un `Int` si n es un `Int`.

En lo que sigue, dados un término $u(x_1, \ldots, x_n) \in T^{\mathcal{T}}_{\Sigma}(X)$ y un conjunto $\{t_1, \ldots, t_n\}$ de *metavariables* representando términos cualesquiera de la familia apropiada, denotamos mediante $u(\vec{t})$ la sustitución simultánea de x_i por t_i en u, para $i = 1, \ldots, n$. De forma similar, dada una fórmula atómica $\phi(\vec{x})$ con variables en $\vec{x}$, denotamos mediante $\phi(\vec{t})$ la sustitución simultánea de x_i por t_i en ϕ, para $i = 1, \ldots, n$.

Proposición 3.13 *Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$, donde $\Omega_i = (K, \Sigma_i, S_i)$. Sean s un tipo en algún $(S_e)_k$, $e \in [1..p]$ y $k \in K$, y sea $C_{[T_e,s]} = \{C_1, \ldots, C_n\}$ el conjunto de sentencias en E_e que especifican s, es decir, aquellas C_i de la forma*

$$\forall(x_1, \ldots, x_{r_i}).\, A_0 \textbf{ if } A_1 \wedge \ldots \wedge A_{q_i}\,, \tag{3.2}$$

donde, para $1 \leq j \leq r_i$, x_j *pertenece a una familia* k_{i_j}, *y* A_0 *es* $w : s$ *para algún término* w *de la familia* k.

Entonces, para todo multiconjunto de fórmulas atómicas $\{\phi_l(x)\}_{l\in[1..p]}$, *donde cada* $\phi_l(x)$ *es una* $(\Sigma_l, \mathcal{T})$*-fórmula atómica con variable* x *de la familia* k, *y expresión booleana* $bexp$, *se cumple el siguiente enunciado metalógico:*

$$\begin{aligned} &\psi_1 \wedge \ldots \wedge \psi_n \\ &\quad \Longrightarrow \forall t \in T_{\Sigma_e,k}.\, (T_e \vdash t : s \Longrightarrow bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_e}))) \,, \end{aligned} \tag{3.3}$$

donde, para $1 \leq i \leq n$ *y* C_i *en* E_e *de la forma* (3.2), ψ_i *es*

$$\forall t_1 \in T_{\Sigma_e,k_{i_1}} \ldots \forall t_{r_i} \in T_{\Sigma_e,k_{i_{r_i}}}.\, [A_1]^\sharp \wedge \ldots \wedge [A_{q_i}]^\sharp \Longrightarrow [A_0]^\sharp$$

y, para $0 \leq j \leq q_i$,

$$[A_j]^\sharp \triangleq \begin{cases} bexp\left(T_1 \vdash^{\mathcal{T}} \phi_1([u(\vec{t})]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([u(\vec{t})]_{T_e})\right) & si\ A_j = u : s \\ T_e \vdash A_j(\vec{t}) & en\ otro\ caso. \end{cases}$$

El enunciado metalógico (3.3) *introduce un principio inductivo de metarrazonamiento* (ind^+) *donde cada* ψ_i *corresponde a un caso inductivo y la línea superior en la definición de* $[A_j]^\sharp$ *suministra la correspondiente hipótesis de inducción.*

Demostración. Supongamos que se tiene $\psi_1 \wedge \ldots \wedge \psi_n$: debemos probar entonces que

$$\forall t \in T_{\Sigma_e,k}.\, (T_e \vdash t : s \Longrightarrow bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_e})))$$

también se cumple. Sea $t \in T_{\Sigma_e,k}$ un término tal que $T_e \vdash t : s$; procedemos por inducción estructural sobre esta derivación. Si $T_e \vdash t : s$ entonces existe una sentencia C_i en E_e de la forma $\forall(x_1, \ldots, x_{r_i}).\, A_0$ **if** $A_1 \wedge \ldots \wedge A_{q_i}$ donde, para $1 \leq j \leq r_i$, x_j pertenece a la familia k_{i_j} y para algún término w en la familia k, A_0 es $w : s$, y una sustitución $\sigma : \{x_1, \ldots, x_{r_i}\} \longrightarrow T_{\Sigma_e}$ tal que

- $T_e \vdash t = \sigma(w)$, y
- $T_e \vdash \sigma(A_j)$, para $1 \leq j \leq q_i$.

Debemos probar que se cumple $bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_e}))$ bajo la hipótesis de inducción que dice que, para $1 \leq j \leq q_i$, si $A_j = u_j : s$ entonces se tiene $bexp\left(T_1 \vdash^{\mathcal{T}} \phi_1([\sigma(u_j)]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([\sigma(u_j)]_{T_e})\right)$. Como por la suposición se tiene ψ_i, también se cumple $[A_1]^\sharp_\sigma \wedge \ldots \wedge [A_{q_i}]^\sharp_\sigma \Longrightarrow [A_0]^\sharp_\sigma$ donde, para $0 \leq j \leq q_i$,

$$[A_j]^\sharp_\sigma \triangleq \begin{cases} bexp\left(T_1 \vdash^{\mathcal{T}} \phi_1([\sigma(u_j)]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([\sigma(u_j)]_{T_e})\right) & si\ A_j = u_j : s \\ T_e \vdash \sigma(A_j) & en\ otro\ caso. \end{cases}$$

Nótese que, para $1 \leq j \leq q_i$,

- Si $A_j = (u_j : s)$, entonces $[A_j]^\sharp_\sigma$ se cumple por la hipótesis de inducción.

- Si $A_j \neq (u_j : s)$, entonces $[A_j]^\sharp_\sigma$ se cumple por la suposición.

De esta manera $[A_0]^\sharp_\sigma$, esto es, $bexp(T_1 \vdash^{\mathcal{T}} \phi_1([\sigma(w)]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([\sigma(w)]_{T_e}))$, también se cumple. Finalmente, como $T_e \vdash t = \sigma(w)$, por la proposición 3.12, tenemos que $bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_e}))$ como se debía demostrar. □

Un principio de análisis por casos ($case^+$) puede definirse de manera completamente análoga a (ind^+), con casi idéntica demostración. Para probar un enunciado metalógico utilizando este principio debemos demostrar que el enunciado se cumple para todos los casos sin valernos de asunciones como las de la hipótesis de inducción; esto se refleja modificando ligeramente la definición de $[A_j]^\sharp$:

$$[A_j]^\flat \triangleq \begin{cases} bexp\left(T_1 \vdash^{\mathcal{T}} \phi_1([u(\vec{t})]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([u(\vec{t})]_{T_e})\right) & si\ j = 0 \\ T_e \vdash A_j(\vec{t}) & en\ otro\ caso. \end{cases}$$

Proposición 3.14 *Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$. Sea s un tipo en algún $(S_e)_k$, $e \in [1..p]$ y $k \in K$, y sea $C_{[T_e,s]} = \{C_1, \ldots, C_n\}$ el conjunto de sentencias en E_e que especifican s, es decir, aquellas C_i de la forma*

$$\forall(x_1, \ldots, x_{r_i}).\, A_0 \textbf{ if } A_1 \wedge \ldots \wedge A_{q_i}\,,$$

donde, para $1 \leq j \leq r_i$, x_j pertenece a una familia k_{i_j}, y A_0 es $w : s$ para algún término w de la familia k.

Entonces, para todo multiconjunto de fórmulas atómicas $\{\phi_l(x)\}_{l\in[1..p]}$, donde cada $\phi_l(x)$ es una $(\Sigma_l, \mathcal{T})$-fórmula atómica con variable x de la familia k, y expresión booleana $bexp$, se cumple el siguiente enunciado metalógico:

$$\psi_1 \wedge \ldots \wedge \psi_n \\ \quad \Longrightarrow \forall t \in T_{\Sigma_e,k}.\, (T_e \vdash t : s \Longrightarrow bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_e})))\,,$$

donde, para $1 \leq i \leq n$ y C_i en E_e, ψ_i es

$$\forall t_1 \in T_{\Sigma_e,k_{i_1}} \ldots \forall t_{r_i} \in T_{\Sigma_e,k_{i_{r_i}}}.\, [A_1]^\flat \wedge \ldots \wedge [A_{q_i}]^\flat \Longrightarrow [A_0]^\flat$$

y, para $0 \leq j \leq q_i$,

$$[A_j]^\flat \triangleq \begin{cases} bexp\left(T_1 \vdash^{\mathcal{T}} \phi_1([u(\vec{t})]_{T_e}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p([u(\vec{t})]_{T_e})\right) & si\ j = 0 \\ T_e \vdash A_j(\vec{t}) & en\ otro\ caso. \end{cases}$$

Por último, enunciamos y demostramos una proposición que resulta muy útil para probar enunciados metalógicos ya que permite, normalmente en conjunción con la proposición 3.12, reducirlos a una forma en la que poder aplicar (ind^+).

Proposición 3.15 *Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$. Sean T_m, T_n teorías en $\mathcal{T}$ y s un tipo en algún $(S_m)_k$, $k \in K$. Entonces, para todo multiconjunto*

finito de fórmulas atómicas $\{\phi_l(x)\}_{l\in[1..p]}$*, donde cada* $\phi_l(x)$ *es una* $(\Sigma_l,\mathcal{T})$*-fórmula atómica con variable* x *de la familia* k*, y expresión booleana* $bexp$*, el siguiente enunciado metalógico se cumple:*

$$\forall t \in T_{\Sigma_m,k}.\left[T_m \vdash t : s \Longrightarrow bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_n}),\ldots,T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_n}))\right]$$
$$\Longrightarrow \forall t \in T_{\Sigma_n,k}.\left[T_m \vdash^{\mathcal{T}} [t]_{T_n} : s \Longrightarrow bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_n}),\ldots,T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_n}))\right].$$

Demostración. Supongamos que se tiene el antecedente del enunciado y veamos que se ha de cumplir el consecuente. Sea $t \in T_{\Sigma_n,k}$ un término tal que $T_m \vdash^{\mathcal{T}} [t]_{T_n} : s$. Nótese que $T_m \vdash^{\mathcal{T}} [t]_{T_n} : s$ implica que existe un término $t' \in T_{\Sigma_m,k} \cap T_{\Sigma_n,k}$ tal que $T_m \vdash t' : s$ y $T_n \vdash t = t'$. Como $t' \in T_{\Sigma_m,k}$ y $T_m \vdash^{\mathcal{T}} t' : s$, y estamos suponiendo que el antecedente se cumple,

$$bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t']_{T_n}),\ldots,T_p \vdash^{\mathcal{T}} \phi_p([t']_{T_n}))$$

lo que, como $t' \in T_{\Sigma_n,k}$ y $T_n \vdash^{\mathcal{T}} t = t'$, por la proposición 3.12 implica que

$$bexp(T_1 \vdash^{\mathcal{T}} \phi_1([t]_{T_n}),\ldots,T_p \vdash^{\mathcal{T}} \phi_p([t]_{T_n})).$$

□

Ejemplo 3.1 Para ilustrar el uso del principio inductivo vamos a demostrar la propiedad (3) en la definición de teorías intercambiables con respecto a `INT2` e `INT1`, que recordemos se expresaba mediante la afirmación metalógica

$$\forall t \in T_{\mathtt{INT2},\mathtt{Num}}.(\mathtt{INT2} \vdash t : \mathtt{Int} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [t]_{\mathtt{INT2}} : \mathtt{Int}),$$

donde $\mathcal{I}$ es el multiconjunto $\{\mathtt{INT1},\mathtt{INT2}\}$.

Demostración. Por (ind^+), podemos probar este enunciado mostrando:

$$\mathtt{INT1} \vdash^{\mathcal{I}} [0]_{\mathtt{INT2}} : \mathtt{Int} \wedge \tag{3.4}$$
$$\forall N \in T_{\mathtt{INT2},\mathtt{Num}}.(\mathtt{INT1} \vdash^{\mathcal{I}} [N]_{\mathtt{INT2}} : \mathtt{Int} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [\mathtt{s}(N)]_{\mathtt{INT2}} : \mathtt{Int}) \wedge \tag{3.5}$$
$$\forall N \in T_{\mathtt{INT2},\mathtt{Num}}.(\mathtt{INT1} \vdash^{\mathcal{I}} [N]_{\mathtt{INT2}} : \mathtt{Int} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [\mathtt{p}(N)]_{\mathtt{INT2}} : \mathtt{Int}). \tag{3.6}$$

Por la proposición 3.12(3), se tiene (3.4). Las demostraciones de (3.5) y (3.6) son parecidas y aquí solo mostramos la de (3.5). Nótese que, por la proposición 3.15, (3.5) se cumple si

$$\forall N \in T_{\mathtt{INT1},\mathtt{Num}}.(\mathtt{INT1} \vdash N : \mathtt{Int} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [\mathtt{s}(N)]_{\mathtt{INT2}} : \mathtt{Int}),$$

que, por la proposición 3.12(2), es equivalente a

$$\forall N \in T_{\mathtt{INT1},\mathtt{Num}}.(\mathtt{INT1} \vdash N : \mathtt{Int} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [\mathtt{s}([N]_{\mathtt{INT1}})]_{\mathtt{INT2}} : \mathtt{Int}).$$

Para demostrar esta última implicación podemos usar de nuevo (ind^+) (pero nótese que ahora estamos razonando sobre `INT1`) y reducir su prueba a demostrar

$$\forall N \in T_{\mathtt{INT1},\mathtt{Num}}.(\mathtt{INT1} \vdash N : \mathtt{Nat} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [\mathtt{s}([N]_{\mathtt{INT1}})]_{\mathtt{INT2}} : \mathtt{Int}) \wedge \tag{3.7}$$
$$\forall N \in T_{\mathtt{INT1},\mathtt{Num}}.(\mathtt{INT1} \vdash N : \mathtt{Neg} \Longrightarrow \mathtt{INT1} \vdash^{\mathcal{I}} [\mathtt{s}([N]_{\mathtt{INT1}})]_{\mathtt{INT2}} : \mathtt{Int}). \tag{3.8}$$

Las demostraciones de (3.7) y de (3.8) son parecidas y aquí mostramos solo la de (3.8). Por ($case^+$) y la proposición 3.12(2), podemos reducir (3.8) a:

$$\texttt{INT1} \vdash^{\mathcal{I}} [\texttt{s(0)}]_{\texttt{INT2}} : \texttt{Int} \wedge \tag{3.9}$$

$$\forall N \in T_{\texttt{INT1,Num}}.(\texttt{INT1} \vdash N : \texttt{Neg} \Longrightarrow \texttt{INT1} \vdash^{\mathcal{I}} [\texttt{s(p}(N)\texttt{)}]_{\texttt{INT2}} : \texttt{Int}). \tag{3.10}$$

Por la proposición 3.12(3), se tiene (3.9). En cuanto a (3.10), sea $N \in T_{\texttt{INT1,Num}}$ un término tal que $\texttt{INT1} \vdash N : \texttt{Neg}$. Nótese que $N \in T_{\texttt{INT2,Num}}$ ya que INT1 e INT2 tienen la misma signatura a nivel de familias. Finalmente, como $\texttt{INT2} \vdash s(p(N)) = N$ se sigue que $\texttt{INT1} \vdash^{\mathcal{I}} [\texttt{s(p}(N)\texttt{)}]_{\texttt{INT2}} : \texttt{Int}$. □

3.6.3 Reflexión en la lógica de pertenencia extendida

Para representar y razonar sobre nuestra definición de relación de derivación extendida, definimos una nueva teoría $\mathrm{U}^{\star}_{\mathrm{MEL}}$ que extiende la teoría universal $\mathrm{U}_{\mathrm{MEL}}$ con un operador binario

```
op _in_ : [Term] [MelTheory] -> [Term] .
```

que nos sirve para representar la clase de equivalencia de un término en una teoría en la lógica de pertenencia.

Con este operador podemos ahora definir una función de representación $\overline{(\_)}$ para términos extendidos que, como muestra la proposición 3.16, cumple la propiedad esperada. Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$. Para todo término $t \in T^{\mathcal{T}}_{\Sigma}(X)$:

$$\overline{t} \triangleq \begin{cases} \mathrm{v}(\overline{x},\overline{k}) & \text{si } t \text{ es una variable } x \text{ en la familia } k \\ \overline{f}[\overline{t_1},\ldots,\overline{t_n}] & \text{si } t = f(t_1,\ldots,t_n) \\ \overline{t'} \texttt{ in } \overline{T} & \text{si } t = [t']_T. \end{cases}$$

Con el operador _in_ podemos escoger el término que más nos convenga en una clase de equivalencia; ahora, la relación de derivación extendida se refleja en $\mathrm{U}^{\star}_{\mathrm{MEL}}$ por medio de las ecuaciones:

```
ceq (Ω̄,Ē) |- applyC(C̄,t̄ in (Ω̄',Ē')) : s̄ if none = true
    if parse(t̄',Ω̄,k̄) = true
    /\ parse(t̄',Ω̄',k̄) = true
    /\ (Ω̄',Ē') |- t̄ = t̄' if none = true
    /\ (Ω̄,Ē) |- applyC(C̄,t̄') : s̄ if none = true .

ceq (Ω̄,Ē) |- applyC(C̄,t̄ in (Ω̄',Ē')) = t̄' if none = true
    if parse(t̄'',Ω̄,k̄) = true
    /\ parse(t̄'',Ω̄',k̄) = true
    /\ (Ω̄',Ē') |- t̄ = t̄'' if none = true
    /\ (Ω̄,Ē) |- applyC(C̄,t̄'') = t̄' if none = true .
```

```
ceq (Ω̄,Ē) |- t⃗' = applyC(C̄,t̄ in (Ω̄',Ē')) if none = true
    if parse(t⃗'',Ω̄,k̄) = true
    /\ parse(t⃗'',Ω̄',k̄) = true
    /\ (Ω̄',Ē') |- t̄ = t⃗'' if none = true
    /\ (Ω̄,Ē) |- t⃗' = applyC(C̄,t⃗'') if none = true .
```

Proposición 3.16 *Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$, donde $\Omega_i = (K, \Sigma_i, S_i)$. Se tiene entonces, para toda teoría $T_i \in \mathcal{T}$, término $t \in T^{\mathcal{T}}_{\Sigma_i,k}$ y tipo s en $(S_i)_k$, $k \in K$,*

$$T_i \vdash^{\mathcal{T}} t : s \iff \mathrm{U}^{\star}_{\mathrm{MEL}} \vdash (\overline{T_i} \mid\!- \bar{t} : \bar{s} \ \texttt{if none}) = \texttt{true}\,.$$

Del mismo modo, para cualesquiera términos $t, t' \in T^{\mathcal{T}}_{\Sigma_i,k'}$ $k \in K$,

$$T_i \vdash^{\mathcal{T}} t = t' \iff \mathrm{U}^{\star}_{\mathrm{MEL}} \vdash (\overline{T_i} \mid\!- \bar{t} = \bar{t}' \ \texttt{if none}) = \texttt{true}\,.$$

Demostración. Por inducción sobre el número de términos de la forma $[t]_{T_j}$ en la fórmula atómica ϕ. Si no hay ninguno, el resultado se tiene por la universalidad de $\mathrm{U}_{\mathrm{MEL}}$. En el caso inductivo, por la definición 3.8 se tiene $T_i \vdash^{\mathcal{T}} \phi$ si y solo si existe $t' \in T_{\Sigma_i,k} \cap T_{\Sigma_j,k}$ tal que $T_i \vdash^{\mathcal{T}} \phi[t']_p$ y $T_j \vdash t = t'$. Por la hipótesis de inducción, $\mathrm{U}^{\star}_{\mathrm{MEL}} \vdash (\overline{T_i} \mid\!- \overline{\phi[t']_p} \ \texttt{if none}) = \texttt{true}$, y por el teorema 3.1, $\mathrm{U}_{\mathrm{MEL}} \vdash (\overline{T_j} \mid\!- \bar{t} = \bar{t}' \ \texttt{if none}) = \texttt{true}$. El resultado se sigue entonces aplicando una de las tres ecuaciones que se han añadido en $\mathrm{U}^{\star}_{\mathrm{MEL}}$. □

3.6.4 Un principio inductivo para razonamiento lógico

Estamos ya preparados para demostrar el resultado principal de esta sección, concretamente, que hay una clase de metateoremas sobre la lógica ecuacional de pertenencia que pueden ser representados y demostrados lógicamente como teoremas sobre el modelo inicial de la teoría $\mathrm{U}^{\star}_{\mathrm{MEL}}$. Como corolario obtendremos un principio de razonamiento inductivo ($\overline{ind^+}$) para demostrar *lógicamente* metateoremas sobre clases de teorías en la lógica de pertenencia.

Para simplificar la presentación del material siguiente vamos a introducir alguna notación adicional. Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$, donde $\Omega_i = (K, \Sigma_i, S_i)$. Para toda teoría $T_i \in \mathcal{T}$ y término $t \in T^{\mathcal{T}}_{\Sigma_i}(X)$, denotamos mediante $\overline{t}^{[X]}$ la metarrepresentación de t, pero ahora cada variable $x \in X$ ha sido sustituida por otra variable $\overline{x}^{[X]}$ en la familia `[Term]` (para la que normalmente escribiremos simplemente x) y denotamos por $\overline{X}^{[X]}$ el conjunto $\overline{X}^{[X]} \triangleq \{\overline{x}^{[X]} \mid x \in X\}$. Por ejemplo, para el término $x : [Nat] + (y : [Nat] - 0)$, cuya metarrepresentación se da en la página 34, se obtiene

```
consM('_,consM('+,consM('_,nilM)))
      [consTL(x : [Term],
              consTL(consM('_,consM('-,consM('_,nilM)))
                     [consTL(y : [Term],
                             consTL('0[nilTL],nilTL))],nilTL))]
```

Ahora para toda teoría $T_i \in \mathcal{T}$ y afirmación de pertenencia $t : s$, con t un término en $T_{\Sigma_i}^{\mathcal{T}}(X)$ y s en algún $(S_i)_k$, definimos

$$\overline{t : s}^{[T_i,X]} \triangleq (\overline{T_i} \,|\!-\, \overline{t}^{[X]} : \overline{s} \;\texttt{if none}) = \texttt{true}\,,$$

y, del mismo modo, para toda ecuación $t = t'$, con t, t' en $T_{\Sigma_i}^{\mathcal{T}}(X)$,

$$\overline{t = t'}^{[T_i,X]} \triangleq (\overline{T_i} \,|\!-\, \overline{t}^{[X]} = \overline{t'}^{[X]} \;\texttt{if none}) = \texttt{true}\,.$$

Ahora podemos definir una función de representación para enunciados metalógicos que satisfacerá la propiedad esperada, como se demostrará en la proposición 3.17 más abajo. Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$, donde $\Omega_i = (K, \Sigma_i, S_i)$. Sea $\{k_1, \ldots, k_n\}$ un multiconjunto finito de familias, donde cada k_i pertenece a K, sea $\vec{x} = \{x_1, \ldots, x_n\}$ un conjunto finito de variables, donde cada x_i pertenece a la familia k_i, y sea τ un enunciado metalógico de la forma

$$\forall t_1 \in T_{\Sigma_1,k_1}. \ldots \forall t_n \in T_{\Sigma_n,k_n}. \mathit{bexp}(T_1 \vdash^{\mathcal{T}} \phi_1(\vec{t}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p(\vec{t}))\,, \tag{3.11}$$

donde cada $\phi_l(\vec{x})$ es una $(\Sigma_l, \mathcal{T})$-fórmula atómica con variables en $\vec{x}$. Definimos entonces

$$\begin{aligned}\overline{\tau} \triangleq \forall x_1. \ldots \forall x_n. ((&\texttt{parse}(x_1, \overline{T_1}, \overline{k_1}) = \texttt{true} \wedge \ldots \wedge \texttt{parse}(x_n, \overline{T_n}, \overline{k_n}) = \texttt{true}) \\ &\Longrightarrow \mathit{bexp}(\overline{\phi_1(\vec{x})}^{[T_1,\vec{x}]}, \ldots, \overline{\phi_p(\vec{x})}^{[T_p,\vec{x}]}))\,,\end{aligned}$$

donde $\{x_1, \ldots, x_n\}$ son ahora variables en la familia `[Term]`. La fórmula $\overline{\tau}$ es una fórmula en la lógica (heterogénea) de primer orden, pero no en la lógica de pertenencia.

Nótese que la clase de los enunciados metalógicos de la forma (3.11) incluye, por ejemplo, todas las instancias de la propiedad (1) en la definición 3.4, y de las propiedades (1) y (3) en la definición 3.5. En particular, el enunciado metalógico (3.1) se representa en $\mathrm{U}^{\star}_{\mathrm{MEL}}$ como la fórmula

$$\begin{aligned}&\forall \texttt{N}.((\texttt{parse}(\texttt{N}, \overline{\texttt{INT2}}, \overline{\texttt{Num}}) = \texttt{true}) \\ &\Longrightarrow (\overline{\texttt{INT2}} \,|\!-\, \texttt{N} : \overline{\texttt{Int}} \;\texttt{if none} = \texttt{true}) \Longrightarrow (\overline{\texttt{INT1}} \,|\!-\, (\texttt{N in } \overline{\texttt{INT2}}) : \overline{\texttt{Int}} \;\texttt{if none} = \texttt{true}))\,,\end{aligned}$$

donde `N` es una variable en la familia `[Term]`.

Proposición 3.17 *Sea $\mathcal{T} = \{T_i\}_{i\in[1..p]} = \{(\Omega_i, E_i)\}_{i\in[1..p]}$ un multiconjunto finito de teorías en $\mathcal{MT}$, donde $\Omega_i = (K, \Sigma_i, S_i)$. Para todo enunciado metalógico τ de la forma (3.11), τ se cumple si y solo si $\mathrm{U}^{\star}_{\mathrm{MEL}} \models \overline{\tau}$.*

Demostración. Primero demostramos la implicación de izquierda a derecha. Supongamos que se cumple τ y sea $\sigma : \{x_1, \ldots, x_n\} \longrightarrow T_{\mathrm{U}^{\star}_{\mathrm{MEL}}}$ una sustitución tal que, para $1 \leq i \leq n$,

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \models \texttt{parse}(\sigma(x_i), \overline{T_i}, \overline{k_i}) = \texttt{true}\,;$$

debemos demostrar que

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \models \sigma\left(\mathit{bexp}(\overline{\phi_1(\vec{x})}^{[T_1,\vec{x}]}, \ldots, \overline{\phi_p(\vec{x})}^{[T_p,\vec{x}]})\right).$$

Nótese que, como $\mathtt{parse}(\sigma(x_i), \overline{T_i}, \overline{k_i})$ es un término cerrado,

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \models \mathtt{parse}(\sigma(x_i), \overline{T_i}, \overline{k_i}) = \mathtt{true},$$

lo que por la completitud de la lógica de pertenencia implica

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \vdash \mathtt{parse}(\sigma(x_i), \overline{T_i}, \overline{k_i}) = \mathtt{true}.$$

De esta manera, por la proposición 3.5 sabemos[2] que, para $1 \leq i \leq n$, $\mathrm{U}^{\star}_{\mathrm{MEL}} \vdash \sigma(x_i) = \overline{w_i}$ para algún $w_i \in T_{\Sigma_i, k_i}$ y que entonces

$$\sigma\left(bexp(\overline{\phi_1(\vec{x})}^{[T_1, \vec{x}]}, \ldots, \overline{\phi_p(\vec{x})}^{[T_p, \vec{x}]})\right)$$

es igual a

$$bexp(\overline{\phi_1(\vec{w})}^{[T_1, \emptyset]}, \ldots, \overline{\phi_p(\vec{w})}^{[T_p, \emptyset]}),$$

en el modelo inicial de $\mathrm{U}^{\star}_{\mathrm{MEL}}$. Por la proposición 3.16, para $1 \leq l \leq p$, $T_l \vdash^{\mathcal{T}} \phi_l(\vec{w})$ si y solo si $\mathrm{U}^{\star}_{\mathrm{MEL}} \vdash \overline{\phi_l(\vec{w})}^{[T_l, \emptyset]}$ que, a su vez, al ser $\overline{\phi_l(\vec{w})}^{[T_l, \emptyset]}$ una fórmula atómica cerrada, es equivalente a $\mathrm{U}^{\star}_{\mathrm{MEL}} \models \overline{\phi_l(\vec{w})}^{[T_l, \emptyset]}$; como estamos asumiendo que $bexp(T_1 \vdash^{\mathcal{T}} \phi_1(\vec{w}), \ldots, T_p \vdash^{\mathcal{T}} \phi_p(\vec{w}))$ se cumple, tenemos finalmente que

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \models bexp(\overline{\phi_1(\vec{w})}^{[T_1, \emptyset]}, \ldots, \overline{\phi_p(\vec{w})}^{[T_p, \emptyset]}),$$

como se exigía.

La demostración de la otra implicación es esencialmente la inversa de la anterior, considerando para términos $w_1, \ldots, w_n$ cualesquiera la sustitución $\sigma : \{x_1, \ldots, x_n\} \longrightarrow T_{\mathrm{U}^{\star}_{\mathrm{MEL}}}$ dada por $\sigma(x_i) = \overline{w_i}$, para $1 \leq i \leq n$. □

Como corolarios de la proposición 3.17 podemos demostrar las versiones "reflexivas" de las proposiciones 3.12 y 3.13, que denotamos respectivamente como proposiciones $\overline{3.12}$ y $\overline{3.13}$. Ambas se obtienen reemplazando cada enunciado metalógico ϕ por la representación $\overline{\phi}$ en $\mathrm{U}^{\star}_{\mathrm{MEL}}$ definida en la página 67. Naturalmente, esto resulta fundamental para nuestros propósitos porque nos da automáticamente un principio de razonamiento inductivo ($\overline{ind^+}$) y otro de análisis por casos ($\overline{case^+}$) para demostrar enunciados metalógicos sobre teorías en la lógica de pertenencia representadas como enunciados lógicos en $\mathrm{U}^{\star}_{\mathrm{MEL}}$. En particular, ($\overline{ind^+}$) resulta ser la fórmula

$$\begin{aligned}&\overline{\psi_1} \wedge \ldots \overline{\psi_n}\\ &\quad \Longrightarrow \forall x.(\mathtt{parse}(x, \overline{T_e}, \overline{k}) = \mathtt{true} \Longrightarrow (\overline{t : s}^{[T_e, x]} \Longrightarrow bexp(\overline{\phi_1([x]_{T_e})}^{[T_1, x]}, \ldots, \overline{\phi_p([x]_{T_e})}^{[T_p, x]}))),\end{aligned}$$

siendo cada $\overline{\psi_i}$ a su vez la representación en $\mathrm{U}^{\star}_{\mathrm{MEL}}$ de cada caso inductivo.

[2] La proposición 3.5 en realidad se aplica a $\mathrm{U}_{\mathrm{MEL}}$, pero como $\mathrm{U}^{\star}_{\mathrm{MEL}}$ la extiende con solo tres ecuaciones que no afectan a parse, en ella se cumple un resultado análogo.

Además, como las proposiciones $\overline{3.12}$ y $\overline{3.13}$ son reflejo de (3.12) y (3.13), las pruebas metalógicas que hagan uso de estas últimas también serán reflejadas por las correspondientes pruebas lógicas. Como ejemplo, a continuación probamos *lógicamente*, utilizando el principio ($\overline{ind^+}$), que `INT2` satisface la propiedad (3) de intercambiabilidad con respecto a `INT1`. Para aligerar la notación, utilizaremos $\overline{T}$ `|-` $\bar{t}$ `:` $\bar{t}$ como abreviatura de $\overline{T}$ `|-` $\bar{t}$ `:` $\bar{t}'$ `if none` y omitiremos `consTL` y `nilTL` en las listas de términos.

Ejemplo 3.2

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \models \forall \mathtt{N}.((\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT2}},\overline{\mathtt{Num}}) = \mathtt{true}) \Longrightarrow (\overline{\mathtt{INT2}} \mid\!\!- \mathtt{N} : \overline{\mathtt{Int}} = \mathtt{true}) \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\mathtt{N}\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}))\,,$$

donde `N` es una variable de la familia `[Term]`.

Demostración. Aplicando ($\overline{ind^+}$) es suficiente demostrar

$$\begin{aligned}
\mathrm{U}^{\star}_{\mathrm{MEL}} \models\ & (\overline{\mathtt{INT1}} \mid\!\!- (\overline{\mathtt{0}}\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}) && (3.12)\\
& \wedge\\
& \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT2}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.13)\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\mathtt{N}\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true})\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\overline{\mathtt{s}}[\mathtt{N}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}))\\
& \wedge\\
& \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT2}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.14)\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\mathtt{N}\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true})\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\overline{\mathtt{p}}[\mathtt{N}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}))\,,
\end{aligned}$$

donde `N` es una variable de la familia `[Term]`. Nótese que (3.12) se cumple por la proposición 3.16 (utilizando la corrección del cálculo de derivación de la lógica de pertenencia).

En cuanto a (3.13) y (3.14), sus demostraciones son parecidas por lo que solo mostramos la de (3.13). Por la proposición $\overline{3.15}$, se tiene (3.13) si

$$\begin{aligned}
\mathrm{U}^{\star}_{\mathrm{MEL}} \models\ & \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT1}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.15)\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- \mathtt{N} : \overline{\mathtt{Int}} = \mathtt{true})\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\overline{\mathtt{s}}[\mathtt{N}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}))\,,
\end{aligned}$$

que, por la proposición $\overline{3.12}$, es equivalente a

$$\begin{aligned}
\mathrm{U}^{\star}_{\mathrm{MEL}} \models\ & \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT1}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.16)\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- \mathtt{N} : \overline{\mathtt{Int}} = \mathtt{true})\\
& \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!\!- (\overline{\mathtt{s}}[\mathtt{N}\ \mathtt{in}\ \overline{\mathtt{INT1}}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}))\,.
\end{aligned}$$

Para demostrar (3.16) podemos usar de nuevo $(\overline{ind^+})$ y reducirlo a:

$$\begin{aligned} \mathrm{U}^{\star}_{\mathrm{MEL}} \models\ & \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT1}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.17)\\ & \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!- \mathtt{N} : \overline{\mathtt{Nat}} = \mathtt{true})\\ & \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!- (\overline{\mathtt{s}}[\mathtt{N}\ \mathtt{in}\ \overline{\mathtt{INT1}}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}))\\ & \wedge\\ & \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT1}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.18)\\ & \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!- \mathtt{N} : \overline{\mathtt{Neg}} = \mathtt{true})\\ & \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!- (\overline{\mathtt{s}}[\mathtt{N}\ \mathtt{in}\ \overline{\mathtt{INT1}}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true})) \end{aligned}$$

Las pruebas de (3.17) y (3.18) son parecidas y solo damos la de (3.18). Aplicando $(\overline{case^+})$ y la proposición 3.12, podemos reducir (3.18) a:

$$\begin{aligned} \mathrm{U}^{\star}_{\mathrm{MEL}} \models\ & \overline{\mathtt{INT1}} \mid\!- (\overline{\mathtt{s}}[\overline{\mathtt{0}}]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true} && (3.19)\\ & \wedge\\ & \forall \mathtt{N}.(\mathtt{parse}(\mathtt{N},\overline{\mathtt{INT1}},\overline{\mathtt{Num}}) = \mathtt{true} && (3.20)\\ & \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!- \mathtt{N} : \overline{\mathtt{Neg}} = \mathtt{true})\\ & \quad \Longrightarrow (\overline{\mathtt{INT1}} \mid\!- (\overline{\mathtt{s}}[\overline{\mathtt{p}}[\mathtt{N}]]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true})) \end{aligned}$$

Nótese que (3.19) se cumple por la proposición 3.16. En cuanto a (3.20), sea $\sigma : \{\mathtt{N}\} \longrightarrow T_{\mathrm{U}^{\star}_{\mathrm{MEL}}}$ una sustitución tal que $(\mathtt{parse}(\sigma(\mathtt{N}),\overline{\mathtt{INT1}},\overline{\mathtt{Num}}) = \mathtt{true})$ y $(\overline{\mathtt{INT1}} \mid\!- \sigma(\mathtt{N}) : \overline{\mathtt{Neg}} = \mathtt{true})$ se cumplen en el modelo inicial de $\mathrm{U}^{\star}_{\mathrm{MEL}}$. Entonces, por la proposición 3.5, $\sigma(\mathtt{N}) = \overline{N}$ para algún término N en la familia $\mathtt{Num}$ y, por el teorema 3.1, $\mathtt{INT1} \vdash N : \mathtt{Neg}$. Por último, como $\mathtt{INT2} \vdash \mathtt{s(p}(N)) = N$ por la proposición 3.16 se tiene que

$$\mathrm{U}^{\star}_{\mathrm{MEL}} \models \overline{\mathtt{INT1}} \mid\!- (\overline{\mathtt{s}}[\overline{\mathtt{p}}[\overline{N}]]\ \mathtt{in}\ \overline{\mathtt{INT2}}) : \overline{\mathtt{Int}} = \mathtt{true}\,.$$

□

3.7 Conclusiones y comparación con resultados anteriores

El trabajo sobre reflexión aquí discutido generaliza y extiende el trabajo previo sobre reflexión en la lógica de reescritura de Clavel (2000), Clavel y Meseguer (2002) y Palomino (2001b).

Los resultados presentados aquí *generalizan* los resultados previos sobre reflexión en la lógica de reescritura a su variante más general, concretamente al caso de teorías de reescritura condicionales cuya lógica ecuacional subyacente es la lógica de pertenencia. Para simplificar la demostración de la corrección de la teoría universal, sin embargo, hemos adoptado un enfoque distinto en su definición. Esencialmente, tanto en Clavel (2000) como en Clavel y Meseguer (2002) una derivación de la forma $T \vdash t \longrightarrow t'$ se reflejaba como $U \vdash \langle \overline{T}, \overline{t} \rangle \longrightarrow \langle \overline{T}, \overline{t'} \rangle$. De acuerdo con esto, la regla **(Transitividad)** no tenía que ser reificada explícitamente en la teoría universal porque, si $T \vdash t_1 \longrightarrow t_3$ se demostraba por transitividad a partir de $T \vdash t_1 \longrightarrow t_2$ y $T \vdash t_2 \longrightarrow t_3$, entonces se

seguía que $U \vdash \langle \overline{T}, \bar{t}_1 \rangle \longrightarrow \langle \overline{T}, \bar{t}_3 \rangle$ también se podía probar por transitividad a partir de $U \vdash \langle \overline{T}, \bar{t}_1 \rangle \longrightarrow \langle \overline{T}, \bar{t}_2 \rangle$ y $U \vdash \langle \overline{T}, \bar{t}_2 \rangle \longrightarrow \langle \overline{T}, \bar{t}_3 \rangle$. En nuestro enfoque actual, en cambio, una derivación de la forma $T \vdash t \longrightarrow t'$ se refleja como $U \vdash (\overline{T} \mid\!- \bar{t} \Rightarrow \bar{t}') \longrightarrow \texttt{true}$ y la regla **(Transitividad)** (y también la regla **(Simetría)** en el caso de la lógica de pertenencia) tiene que ser explícitamente reificada en la teoría universal.

Los resultados presentados *extienden* de manera natural resultados previos en lógica de reescritura a otras lógicas relacionadas como son las lógica de pertenencia, heterogénea y de Horn con igualdad. Las extensiones son muy naturales en el sentido de que las teorías universales propuestas están a su vez relacionadas. En este sentido estos resultados arrojan algo de luz sobre la cuestión de cómo se relacionan las teorías universales de lógicas relacionadas. Además, estos resultados proporcionan un fundamento lógico para lenguajes reflexivos y herramientas basadas en estas lógicas, y en particular para el propio lenguaje Maude.

Por último, el trabajo de la sección 3.6 constituye tanto un desarrollo como una aplicación de la metodología reflexiva propuesta en Basin et al. (2004) para metarrazonar formalmente utilizando la lógica de pertenencia. Los principios de metarrazonamiento allí introducidos exigen trabajar sobre teorías relacionadas por la relación de inclusión; nosotros hemos eliminado esa restricción, lo que incrementa considerablemente su aplicabilidad.

Una de las ventajas del metarrazonamiento formal utilizando reflexión es que se puede llevar a cabo utilizando herramientas de razonamiento lógico ya existentes. En esta dirección, existen planes para extender el ITP con capacidades de metarrazonamiento y, de hecho, ya se han realizado algunos experimentos de este tipo utilizando un interfaz para interactuar con el ITP.

Capítulo 4

Abstracción

A la hora de especificar un sistema se pueden distinguir dos niveles diferentes:

- un nivel de *especificación de sistema*, en el que se especifica el sistema computacional en el que uno esté interesado, y
- un nivel de *especificación de propiedades*, en el que se especifican todas aquellas propiedades que sean relevantes.

El principal interés que tiene la lógica de reescritura es que proporciona un marco muy flexible para la especificación a nivel de sistema de sistemas concurrentes, como demuestra la abundante bibliografía que se puede encontrar en Martí-Oliet y Meseguer (2002b).

Un problema surge, sin embargo, a la hora de verificar qué propiedades satisfacen dichos sistemas. Para aquellos cuyo conjunto de estados alcanzables es finito se puede utilizar una herramienta como el comprobador de modelos de Maude para estudiar sus propiedades en la lógica temporal lineal. Desafortunadamente, esta solución no está disponible para sistemas infinitos o con un número muy grande de estados. En este capítulo presentamos una técnica muy sencilla para resolver este problema en determinadas situaciones, que consiste en añadir ecuaciones a la especificación original para colapsar el conjunto de estados y hacer que pase a ser finito.

4.1 Especificación de sistemas en lógica de reescritura y comprobación de modelos

Recordemos del capítulo 2 que los sistemas computacionales se axiomatizan en la lógica de reescritura por medio de teorías de reescritura $\mathcal{R} = (\Sigma, E, R)$; sus estados vienen dados por el tipo de datos $T_{\Sigma,k}$ asociado a la teoría ecuacional (Σ, E) y una familia k distinguida, y las transiciones quedan descritas por las reglas condicionales en R. Estas teorías, además, se pueden especificar y ejecutar en Maude utilizando módulos de sistema.

Como se avanzó en la sección 2.4.4, esta descripción se puede complementar con información sobre las distintas propiedades que se satisfacen en cada estado. De hecho, una característica atractiva de la lógica de reescritura es que ofrece una integración fluida

entre los niveles de especificación de sistema y de propiedades porque podemos especificar los predicados de estado relevantes *ecuacionalmente*, lo que determina la función de etiquetado L_Π y la semántica de las fórmulas en lógica temporal de manera única. Para asociar propiedades en una lógica temporal a una teoría de reescritura $\mathcal{R} = (\Sigma, E \cup A, R)$ con una familia distinguida k de estados solo necesitamos hacer explícitos los predicados de estado relevantes Π, que no tienen por qué ser parte de la especificación del sistema $\mathcal{R}$. Los predicados de estado Π pueden definirse mediantes ecuaciones D en una teoría ecuacional $(\Sigma', E \cup A \cup D)$ que extienda $(\Sigma, E \cup A)$ de manera conservadora; de manera precisa, el único Σ-homomorfismo $T_{\Sigma/E \cup A} \to T_{\Sigma'/E \cup A \cup D}$ inducido por la inclusión de teorías $(\Sigma, E \cup A) \subseteq (\Sigma', E \cup A \cup D)$ debería ser biyectivo para cada tipo s en Σ. También asumimos que $(\Sigma', E \cup D)$ contenga la teoría *BOOL* de valores booleanos de la misma manera conservadora. La sintaxis que define los predicados de estado consiste en una subsignatura $\Pi \subseteq \Sigma'$ de operadores p con la forma general $p : s_1 \dots s_n \longrightarrow \mathit{Prop}$ (donde *Prop* es el tipo de las proposiciones), lo que refleja el hecho de que los predicados de estado pueden ser *paramétricos*. La semántica de los predicados de estado Π se define con las ecuaciones D mediante un operador $\_ \models \_ : k \;\; \mathit{Prop} \longrightarrow \mathit{Bool}$ en Σ'. Por definición, dados los términos cerrados $u_1, \dots, u_n$ decimos que el predicado de estado $p(u_1, \dots, u_n)$ *se cumple* en el estado $[t]$ si y solo si

$$E \cup A \cup D \vdash (\forall \emptyset)\, t \models p(u_1, \dots, u_n) = \mathit{true}\,.$$

Ahora podemos asociar a $\mathcal{R}$ una estructura de Kripke $\mathcal{K}(\mathcal{R}, k)_\Pi$, cuyas proposiciones atómicas son las contenidas en el conjunto

$$AP_\Pi = \{\theta(p) \mid p \in \Pi \text{ y } \theta \text{ sustitución cerrada}\}\,,$$

donde, por convención, si p tiene n parámetros, $\theta(p)$ denota el término $\theta(p(x_1, \dots, x_n))$.

La estructura de Kripke asociada a una teoría de reescritura $\mathcal{R}$ y predicados de estado Π queda entonces definida por $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E,k}, (\to^1_{\mathcal{R},k})^\bullet, L_\Pi)$, donde

$$L_\Pi([t]) = \{\theta(p) \in AP_\Pi \mid \theta(p) \text{ se cumple en } [t]\}\,.$$

En la práctica querremos que la igualdad $t \models p(u_1, \dots, u_n) = \mathit{true}$ sea *decidible*. Esto se puede conseguir especificando ecuaciones $E \cup D$ que sean Church-Rosser y terminantes módulo A. En tal caso, si comenzamos con una teoría de reescritura *ejecutable* $\mathcal{R}$ y definimos predicados de estado decidibles en la forma que acabamos de mencionar, lo que obtenemos es una estructura de Kripke *computable* que, en caso de tener un conjunto de estados alcanzable finito, se puede utilizar con un comprobador de modelos. (Sobre la computabilidad de las estructuras de Kripke se dirá mucho más en el capítulo 5.)

Para continuar ilustrando el uso de la lógica de reescritura como marco para la especificación tanto a nivel de sistema como de propiedades, así como para motivar el desarrollo posterior sobre abstracciones, vamos a utilizar como ejemplo el protocolo de la panadería de Lamport (1974).

El protocolo de la panadería es un protocolo con un número infinito de estados que consigue exclusión mutua entre procesos que compiten por acceder a una región crítica; a cada proceso se le asigna un número y se les atiende de manera secuencial comenzando por aquellos cuyo número sea menor. Una especificación sencilla en Maude para el caso en el que se tienen dos procesos es la siguiente:

```
mod BAKERY is
  protecting NAT .
  sorts Mode State .

  ops sleep wait crit : -> Mode .
  op <_,_,_,_> : Mode Nat Mode Nat -> State .
  op initial : -> State .

  vars P Q : Mode .
  vars X Y : Nat .

  eq initial = < sleep, 0, sleep, 0 > .

  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  rl [p1_wait] : < wait, X, Q, 0 > => < crit, X, Q, 0 > .
  crl [p1_wait] : < wait, X, Q, Y > => < crit, X, Q, Y > if not (Y < X) .
  rl [p1_crit] : < crit, X, Q, Y > => < sleep, 0, Q, Y > .
  rl [p2_sleep] : < P, X, sleep, Y > => < P, X, wait, s X > .
  rl [p2_wait] : < P, 0, wait, Y > => < P, 0, crit, Y > .
  crl [p2_wait] : < P, X, wait, Y > => < P, X, crit, Y > if Y < X .
  rl [p2_crit] : < P, X, crit, Y > => < P, X, sleep, 0 > .
endm
```

Esta especificación corresponde a una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, donde (Σ, E) importa la teoría ecuacional de los números naturales `NAT` y la signatura Σ contiene además los tipos `Mode` y `State`, con las constantes `sleep`, `wait` y `crit` como únicos operadores cuyo rango es `Mode`. Los estados se representan mediante términos de tipo `State`, que son construidos por un operador `<_,_,_,_>`; las dos primeras componentes describen la situación en que se encuentra el primer proceso (el modo de funcionamiento en el que está en estos momentos y su prioridad, dada por el número según el cual será atendido) y las dos últimas la situación del segundo. E consiste simplemente de las ecuaciones importadas de `NAT` más la ecuación que define el estado `initial`. R contiene ocho reglas de reescritura, cuatro para cada proceso. Estas reglas describen cómo cada proceso pasa de estar durmiendo (`sleep`) a estar esperando (`wait`), de esperando a su sección crítica (`crit`) y de nuevo vuelta a dormir. Naturalmente, en este caso la familia escogida para los estados es `[State]`.

Dos propiedades básicas que podemos querer verificar son:

1. *exclusión mutua*: los dos procesos no se encuentran nunca simultáneamente en sus regiones críticas; y
2. *vivacidad*: cualquier proceso que se encuentre en modo de espera terminará entrando en su región crítica en algún momento.

Para especificar estas propiedades es suficiente con especificar en Maude el siguiente conjunto Π de predicados de estado:

```
mod BAKERY-CHECK is
  inc MODEL-CHECKER .
```

```
  inc BAKERY .

  ops 1wait 2wait 1crit 2crit : -> Prop .

  vars P Q : Mode .
  vars X Y : Nat .

  eq (< P, X, Q, Y > |= 1wait) = (P == wait) .
  eq (< P, X, Q, Y > |= 2wait) = (Q == wait) .
  eq (< P, X, Q, Y > |= 1crit) = (P == crit) .
  eq (< P, X, Q, Y > |= 2crit) = (Q == crit) .
endm
```

donde _==_ es el predicado de igualdad, que se cumple para dos términos cerrados bajo las hipótesis de Church-Rosser y terminación si sus formas canónicas son iguales y es falso en caso contrario[1]. La propiedad de exclusión mutua se expresa entonces mediante la fórmula temporal

```
[]~ (1crit /\ 2crit)
```

y la de vivacidad mediante la fórmula temporal

```
(1wait |-> 1crit) /\ (2wait |-> 2crit),
```

donde `[]`, `~` y `|->` son respectivamente los símbolos que utiliza el comprobador de modelos para representar los operadores "siempre" $\Box$, negación $\neg$ y "lleva a" $\rightsquigarrow$, este último definido como $\varphi \rightsquigarrow \psi$ si y solo si $\varphi \rightarrow \Diamond\psi$.

Desafortunadamente, sin embargo, no podemos utilizar directamente el comprobador de modelos de Maude con estas formulas ya que el conjunto de estados alcanzables desde `initial` (que se definió en el módulo `BAKERY`) es *infinito*. Un proceso podría querer acceder a la región crítica y para ello recibiría el número 1. Antes de que acceda, el otro proceso podría solicitar entrar también y se le asignaría el número 2. Una vez que el primer proceso deje libre la región crítica pierde su número (el contador se reinicia a 0), pero si volviera a solicitar entrar antes de que el otro proceso deje libre a su vez la región crítica se le asignará otro número, que será el 3. El esquema podría repetirse indefinidamente, ahora con el segundo proceso recibiendo el número 4, luego el primero recibiendo 5, ...

La alternativa que nos queda entonces es definir una *abstracción* del sistema en la cual solo se pueda alcanzar un número finito de estados desde `initial` y entonces, ya sí, aplicar el comprobador de modelos. La manera de definir esa abstracción será el objeto del resto del presente capítulo.

4.2 Simulaciones

Presentamos aquí una noción de simulación similar a la de Clarke et al. (1999), pero algo más general (las simulaciones en Clarke et al. (1999) se corresponden esencialmen-

[1] Por un metateorema de Bergstra y Tucker (1980), bajo tales hipótesis siempre se puede axiomatizar _==_ mediante un conjunto de ecuaciones Church-Rosser y terminante. Por conveniencia y eficiencia, Maude computa de manera interna los predicados de igualdad mediante simplificación ecuacional.

te con nuestras simulaciones *estrictas*). Primero tenemos que definir simulaciones entre sistemas de transiciones.

Definición 4.1 *Dados dos sistemas de transiciones* $\mathcal{A} = (\mathcal{A}, \rightarrow_{\mathcal{A}})$ *y* $\mathcal{B} = (\mathcal{B}, \rightarrow_{\mathcal{B}})$*, una* simulación de sistemas de transiciones $H : \mathcal{A} \longrightarrow \mathcal{B}$ *es una relación binaria* $H \subseteq A \times B$ *tal que si* $a \rightarrow_{\mathcal{A}} a'$ *y* aHb *entonces existe un* $b' \in B$ *tal que* $b \rightarrow_{\mathcal{B}} b'$ *y* $a'Hb'$*.*

Decimos que H *es una* simulación total *si la relación* H *es total. Un* morfismo de sistemas de transiciones H *es una simulación total tal que* H *es una función*[2]*. Si tanto* H *como* H^{-1} *son simulaciones, entonces decimos que* H *es una* bisimulación*.*

Gráficamente, el requisito de simulación se puede representar así:

$$\begin{array}{ccc} a & \longrightarrow_{\mathcal{A}} & a' \\ H & & H \\ b & \longrightarrow_{\mathcal{B}} & b' \end{array}$$

Las simulaciones de sistemas de transiciones H se extienden de manera natural a los caminos definiendo $\pi H \rho$ si $\pi(i) H \rho(i)$ para cada $i \in \mathbb{N}$.

Definición 4.2 *Dadas dos estructuras de Kripke* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ *y* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ *sobre el mismo conjunto AP de proposiciones atómicas, una AP-*simulación $H : \mathcal{A} \longrightarrow \mathcal{B}$ *de* $\mathcal{A}$ *por* $\mathcal{B}$ *viene dada por una simulación* $H : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ *entre los sistemas de transiciones subyacentes tal que si* aHb *se cumple también* $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$*.*

Decimos que H *es un AP-*morfismo *si su simulación de sistemas de transiciones subyacente es un morfismo. Decimos que* H *es una AP-*bisimulación *si* H *y* H^{-1} *son AP-simulaciones. También, llamamos a* H estricta *si* aHb *implica que* $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$*. Nótese que toda AP-bisimulación es necesariamente estricta.*

El hecho de que $H : \mathcal{A} \longrightarrow \mathcal{B}$ sea una simulación de sistemas de transiciones garantiza que, para cada camino concreto en $\mathcal{A}$ que empiece en un estado relacionado con otro en $\mathcal{B}$, existe un camino que lo simula en $\mathcal{B}$. La segunda condición implica que un estado en $\mathcal{B}$ puede satisfacer como mucho aquellas proposiciones atómicas que se cumplen en todos los estados de $\mathcal{A}$ que simula.

Antes de continuar es interesante señalar que esta definición de simulación entre sistemas de transiciones, y por lo tanto la de *AP*-simulaciones, es muy general y ni siquiera requiere que la relación H sea total. Esto tiene algunas consecuencias quizá inesperadas: por ejemplo, ¡la relación vacía es una bisimulación de manera vacua! La noción es natural, sin embargo, en el sentido de que cada *AP*-simulación surge de una *AP*-simulación total restringida a un cierto dominio de interés.

Definición 4.3 *Dados sistemas de transiciones* $\mathcal{A}$ *y* $\mathcal{B}$*, decimos que* $\mathcal{A}$ *es un* subsistema *de* $\mathcal{B}$ *si* $A \subseteq B$ *y* $\rightarrow_{\mathcal{A}} \subseteq \rightarrow_{\mathcal{B}}$*, y escribimos entonces que* $\mathcal{A} \subseteq \mathcal{B}$*. Decimos que un subsistema* $\mathcal{A}$ *es* completo *en* $\mathcal{B}$ *si para todo* $a \in A$*, siempre que* $a \rightarrow_{\mathcal{B}} a'$ *se cumple que* $a' \in A$ *y* $a \rightarrow_{\mathcal{A}} a'$*.*

[2] En lo que sigue todas nuestras funciones serán siempre totales, a menos que se indique explícitamente lo contrario.

Una estructura de Kripke $\mathcal{A}$ es una subestructura de Kripke *de $\mathcal{B}$ si el sistema de transiciones subyacente en $\mathcal{A}$ es un subsistema del subyacente en $\mathcal{B}$ y además se tiene que $L_{\mathcal{A}} = L_{\mathcal{B}}|_A$. Diremos que es una subestructura* completa *si lo es al nivel de los sistemas de transiciones.*

Observación 4.1 Nótese que si $\mathcal{A}$ es una subestructura de Kripke completa de $\mathcal{B}$ entonces la inclusión $i : \mathcal{A} \longrightarrow \mathcal{B}$ es una *AP*-bisimulación.

Proposición 4.1 *Sea $H : \mathcal{A} \longrightarrow \mathcal{B}$ una AP-simulación. Entonces, para toda subestructura de Kripke completa $\mathcal{B}' \subseteq \mathcal{B}$ se tiene que $H^{-1}(\mathcal{B}') = (H^{-1}(B'), \rightarrow_{\mathcal{A}} \cap\, H^{-1}(B') \times H^{-1}(B'), L_{\mathcal{A}}|_{H^{-1}(B')})$ es una subestructura de Kripke completa de $\mathcal{A}$. En particular, $H^{-1}(\mathcal{B})$ es una subestructura de Kripke completa de $\mathcal{A}$.*

Demostración. Tenemos que demostrar tanto que la relación de transición es total como que $H^{-1}(\mathcal{B}')$ es completa en $\mathcal{A}$. Sea a un elemento de $H^{-1}(B')$ tal que $a \rightarrow_{\mathcal{A}} a'$ (que tiene que existir porque $\rightarrow_{\mathcal{A}}$ es total). Por definición, existe un elemento $b \in B'$ tal que aHb. Ahora, puesto que H es una simulación, existe $b' \in B$ tal que $a'Hb'$ y $b \rightarrow_{\mathcal{B}} b'$ y además, como $\mathcal{B}'$ es completa en $\mathcal{B}$, se tiene que $b' \in B'$. De esta forma $a' \in H^{-1}(B')$, $\rightarrow_{H^{-1}(\mathcal{B}')}$ es total y $H^{-1}(\mathcal{B}')$ es completa en $\mathcal{A}$. □

Por lo tanto, cualquier *AP*-simulación $H : \mathcal{A} \longrightarrow \mathcal{B}$ se puede entender alternativamente como una simulación total $H : H^{-1}(\mathcal{B}) \longrightarrow \mathcal{B}$.

Como consecuencias sencillas de las correspondiente definiciones se tienen los siguientes resultados sobre simulaciones, que presentamos sin demostración.

Lema 4.1 *Si $\{H_i : \mathcal{A} \longrightarrow \mathcal{B}\}_{i \in I}$ es un conjunto de simulaciones de sistemas de transiciones (resp. AP-simulaciones) se tiene que $\bigcup_{i \in I} H_i : \mathcal{A} \longrightarrow \mathcal{B}$ es una simulación de sistemas de transiciones (resp. una AP-simulación).*

Corolario 4.1 *Para dos sistemas de transiciones cualesquiera (resp. estructuras de Kripke) $\mathcal{A}$ y $\mathcal{B}$ existe una simulación de sistemas de transiciones (resp. AP-simulación) máxima, en el sentido de que contiene a cualquier otra tal. En ocasiones podría tratarse de la simple simulación degenerada vacía.*

Lema 4.2 *Si $F : \mathcal{A} \longrightarrow \mathcal{B}$ y $G : \mathcal{B} \longrightarrow \mathcal{C}$ son simulaciones de sistemas de transiciones (resp. AP-simulaciones) entonces su composición $G \circ F$ es también una simulación de sistemas de transiciones (resp. AP-simulación).*

Corolario 4.2 *Para cualquier sistema de transiciones (resp. estructura de Kripke) $\mathcal{A}$ existe una bisimulación máxima $H : \mathcal{A} \longrightarrow \mathcal{A}$ de sistemas de transiciones (resp. AP-bisimulación), que es siempre una relación de equivalencia.*

Nótese que la función identidad $1_{\mathcal{A}} : \mathcal{A} \longrightarrow \mathcal{A}$ es trivialmente un morfismo de sistemas de transiciones y un *AP*-morfismo para todo conjunto de proposiciones atómicas *AP*. Por lo tanto, por el lema 4.2, los sistemas de transiciones junto con sus simulaciones definen

una categoría **TSys**. De la misma manera, las estructuras de Kripke junto con las AP-simulaciones definen una categoría[3] $\mathbf{KSim}_{AP}$, con dos correspondientes subcategorías $\mathbf{KMap}_{AP}$y $\mathbf{KBSim}_{AP}$ cuyos morfismos son, respectivamente, los AP-morfismos y las AP-bisimulaciones. Naturalmente también hay una subcategoría $\mathbf{KSim}^{\mathrm{str}}_{AP}$ de AP-simulaciones estrictas y correspondientes subcategorías $\mathbf{KMap}^{\mathrm{str}}_{AP}$ y $\mathbf{KBSim}^{\mathrm{str}}_{AP} = \mathbf{KBSim}_{AP}$. Nótese que si H es un isomorfismo en $\mathbf{KSim}_{AP}$ entonces debe ser simultáneamente un AP-morfismo y una AP-bisimulación. Por último, la aplicación $(A, \to_{\mathcal{A}}, L_{\mathcal{A}}) \mapsto (A, \to_{\mathcal{A}})$ se puede extender a un funtor de olvido $\mathbf{TS} : \mathbf{KSim}_{AP} \longrightarrow \mathbf{TSys}$.

4.3 Las simulaciones preservan propiedades

La característica fundamental que nos va a interesar de las AP-simulaciones es que reflejan la satisfacción de una clase apropiada de fórmulas. (¡No confundir esta noción con la idea de reflexión del capítulo 3!) Como se mencionó en el capítulo 2, la lógica que vamos a utilizar para expresar propiedades de sistemas es ACTL*, lo que apunta a que sean precisamente las fórmulas de esta lógica las que se vean reflejadas. A veces, sin embargo, para evitar introducir de manera implícita cuantificadores universales resulta más conveniente restringirse al fragmento ACTL$^*\backslash\neg(AP)$ de ACTL$^*(AP)$ libre de negaciones, que se define como sigue:

$$\begin{array}{ll} \text{fórmulas de estado:} & \varphi = p \in AP \mid \top \mid \bot \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathbf{A}\psi \\ \text{fórmulas de camino:} & \psi = \varphi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbf{X}\psi \mid \psi\mathbf{U}\psi \mid \psi\mathbf{R}\psi \mid \mathbf{G}\psi \mid \mathbf{F}\psi\,. \end{array}$$

Nuestra generalización de la definición de simulaciones en la página 77 y la distinción entre AP-simulaciones estrictas y no estrictas se hizo ya con esta lógica en mente.

Escribimos State$\backslash\neg(AP)$ y Path$\backslash\neg(AP)$ para denotar los conjuntos de fórmulas de estado y de camino en ACTL$^*\backslash\neg(AP)$, respectivamente. Al ser ACTL$^*\backslash\neg$ una sublógica de ACTL*, su semántica sigue siendo la misma. Nótese que desde el punto de vista práctico la restricción a fórmulas en ACTL$^*\backslash\neg$ no supone una pérdida real de generalidad porque siempre podemos transformar una fórmula cualquiera en ACTL* en otra semánticamente equivalente en ACTL$^*\backslash\neg$, simplemente mediante la introducción de proposiciones atómicas nuevas para representar la negación de las originales. Para ello, consideramos el conjunto de proposiciones atómicas extendido $\widehat{AP} = AP \cup \overline{AP}$, donde $\overline{AP} = \{\bar{p} \mid p \in AP\}$, y construimos $\hat{\varphi}$ formando primero la forma normal negativa de φ (en la que todas las negaciones se llevan al nivel de los átomos) y reemplazando a continuación cada átomo negado $\neg p$ por $\bar{p}$. Dada una estructura de Kripke $\mathcal{A} = (A, \to_{\mathcal{A}}, L_{\mathcal{A}})$, definimos $\widehat{\mathcal{A}} = (A, \to_{\mathcal{A}}, L_{\widehat{\mathcal{A}}})$ con $L_{\widehat{\mathcal{A}}}(a) = L_{\mathcal{A}}(a) \cup \{\bar{p} \in \overline{AP} \mid p \notin L_{\mathcal{A}}(a)\}$. Se tiene entonces que $\mathcal{A}, a \models \varphi \iff \widehat{\mathcal{A}}, a \models \hat{\varphi}$.

Definición 4.4 *Una AP-simulación $H : \mathcal{A} \longrightarrow \mathcal{B}$* refleja *la satisfacción de una fórmula $\varphi \in$ CTL$^*(AP)$ cuando:*

- *φ es una fórmula de estado y $\mathcal{B}, b \models \varphi$ y aHb implican que $\mathcal{A}, a \models \varphi$; o bien*

[3] Un lector que piense en términos de la teoría de categorías puede reconocer la categoría de estructuras de Kripke y AP-simulaciones como la categoría de morfismos *parciales* asociada a la categoría de las estructuras de Kripke y AP-simulaciones *totales* tomando las subestructuras de Kripke completas como los subobjetos.

- *φ es una fórmula de camino y $\mathcal{B}, \rho \models \varphi$ y $\pi H \rho$ implican que $\mathcal{A}, \pi \models \varphi$.*

El siguiente teorema generaliza ligeramente el teorema 16 en Clarke et al. (1999):

Teorema 4.1 *Las AP-simulaciones reflejan siempre la satisfacción de las fórmulas en la lógica* ACTL$^*\backslash\neg(AP)$. *Además, las simulaciones estrictas también reflejan la satisfacción de las fórmulas en* ACTL$^*(AP)$.

Demostración. Consideremos primero el caso no estricto. Sean $H : \mathcal{A} \longrightarrow \mathcal{B}$ una *AP*-simulación y $a \in A$ y $b \in B$ elementos tales que aHb. Si π es un camino en $\mathcal{A}$ que comienza en a es inmediato demostrar por inducción sobre la longitud de segmentos iniciales que existe un camino ρ en $\mathcal{B}$ que comienza en b tal que $\pi H \rho$. Para toda fórmula de estado φ y fórmula de camino ψ en ACTL$^*\backslash\neg(AP)$ se demuestra por inducción estructural simultánea que $\mathcal{B}, b \models \varphi$ implica $\mathcal{A}, a \models \varphi$ y que $\mathcal{B}, \rho \models \psi$ implica $\mathcal{A}, \pi \models \psi$.

Cada uno de los casos es inmediato, por lo que consideramos solo algunos de ellos. Para una proposición atómica p, si $\mathcal{B}, b \models p$ se tiene que $p \in L_{\mathcal{B}}(b)$ y como $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$ al ser H una simulación se sigue que $\mathcal{A}, a \models p$. Para probar los casos correspondientes a los operadores $\top$, $\vee$ y $\wedge$ basta con aplicar la hipótesis de inducción. Si $\mathcal{B}, b \models \mathbf{A}\psi$, entonces $\mathcal{B}, \rho' \models \psi$ para todo camino ρ' que comience en b. Sea entonces π' un camino que comience en a y, abusando de la notación, sea $H(\pi')$ un camino en $\mathcal{B}$ que comience en b tal que $\pi' \, H \, H(\pi')$. Se sigue que $\mathcal{B}, H(\pi') \models \psi$ y por hipótesis de inducción $\mathcal{A}, \pi' \models \psi$; como esta relación se satisface para todo camino que comience en a llegamos a que $\mathcal{A}, a \models \mathbf{A}\psi$. Para los demás operadores temporales el esquema es el mismo. Por ejemplo, si $\mathcal{B}, \rho \models \mathbf{G}\psi$ se tiene que $\mathcal{B}, \rho^i \models \psi$ para todo i de donde por hipótesis de inducción se sigue que $\mathcal{A}, \pi^i \models \psi$ y por lo tanto $\mathcal{A}, \pi \models \mathbf{G}\psi$.

En el caso estricto, puesto que toda fórmula es semánticamente equivalente a otra en formal normal negativa, es suficiente con demostrar el teorema para estas. La demostración es como la del caso no estricto, aunque ahora además hay que considerar el caso $\neg p$. Para el mismo tenemos que $\mathcal{B}, b \models \neg p$ implica que $p \notin L_{\mathcal{B}}(b)$ y como H es estricta $p \notin L_{\mathcal{A}}(a)$ y $\mathcal{A}, a \models \neg p$. □

Este teorema es la base fundamental del método de comprobación de modelos mediante abstracción: dado un sistema $\mathcal{M}$ infinito (o demasiado grande), hay que encontrar un sistema $\mathcal{A}$ con un conjunto de estados alcanzable finito que lo simule y usar un comprobador de modelos para intentar demostrar que φ se cumple en $\mathcal{A}$; si eso es así, entonces por el teorema 4.1, φ también se cumple en $\mathcal{M}$. En general, sin embargo, uno tan solo tiene el sistema concreto $\mathcal{M}$ y una función *sobreyectiva* $h : M \longrightarrow A$ que lleva estados concretos a un dominio abstracto simplificado A (normalmente finito). Entonces estamos interesados en encontrar una estructura de Kripke a partir de h que sea la que mejor simula $\mathcal{M}$ bajo determinadas condiciones. Clarke et al. (1994) definen el *sistema de transiciones minimal* asociado a un sistema de transiciones $\mathcal{M}$ y a una función sobreyectiva $h : M \longrightarrow A$; usando nuestra noción de simulación esta se puede extender al nivel de las estructuras de Kripke.

Definición 4.5 *La* estructura de Kripke minimal $\mathcal{M}^h_{\min}$ *asociada a la estructura de Kripke* $\mathcal{M}$ *y a la función sobreyectiva* $h : M \longrightarrow A$ *viene dada por la terna* $(A, (h \times h)(\rightarrow_{\mathcal{M}}), L_{\mathcal{M}^h_{\min}})$, *donde* $(h \times h)(\rightarrow_{\mathcal{M}})$ *es la imagen de* $\rightarrow_{\mathcal{M}}$ *por* h *y* $L_{\mathcal{M}^h_{\min}}(a) = \bigcap_{x \in h^{-1}(a)} L_{\mathcal{M}}(x)$.

La siguiente proposición es una consecuencia inmediata de las definiciones.

Proposición 4.2 *Para cualesquiera estructura de Kripke* $\mathcal{M}$ *y función sobreyectiva* h, $h : \mathcal{M} \longrightarrow \mathcal{M}^h_{\min}$ *es un AP-morfismo.*

El uso del adjetivo "minimal" es apropiado porque, tal y como se señala en Clarke et al. (1994), $\mathcal{M}^h_{\min}$ es la aproximación más exacta a $\mathcal{M}$ que es consistente con h. Dentro de nuestro marco esta afirmación se puede expresar de manera precisa mediante la noción de *morfismo opcartesiano* de la teoría de categorías, como se verá en la sección 6.2.

Los sistemas minimales también se pueden ver como cocientes. Sean $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ una estructura de Kripke sobre AP y $\equiv$ una relación de equivalencia arbitraria sobre A. Podemos entonces usar $\equiv$ para definir una nueva estructura de Kripke, $\mathcal{A}/\equiv$, dada por $(A/\equiv, \rightarrow_{\mathcal{A}/\equiv}, L_{\mathcal{A}/\equiv})$, donde:

- $[a_1] \rightarrow_{\mathcal{A}/\equiv} [a_2]$ si y solo si existen $a'_1 \in [a_1]$ y $a'_2 \in [a_2]$ tales que $a'_1 \rightarrow_{\mathcal{A}} a'_2$;
- $L_{\mathcal{A}/\equiv}([a]) = \bigcap_{x \in [a]} L_{\mathcal{A}}(x)$.

Es trivial comprobar que la función de proyección en clases de equivalencia $q_{\equiv} : a \mapsto [a]$ es un AP-morfismo $q_{\equiv} : \mathcal{A} \longrightarrow \mathcal{A}/\equiv$, que llamamos *abstracción cociente* definida por $\equiv$. De esta forma, una presentación equivalente del sistema minimal es la expresada por la siguiente proposición.

Proposición 4.3 *Sean* $\mathcal{M} = (M, \rightarrow_{\mathcal{M}}, L_{\mathcal{M}})$ *una estructura de Kripke y* $h : M \longrightarrow A$ *una función sobreyectiva. Entonces existe una AP-bisimulación biyectiva entre las estructuras de Kripke* $\mathcal{M}^h_{\min}$ *y* $\mathcal{M}/\equiv_h$, *donde por definición* $x \equiv_h y$ *si y solo si* $h(x) = h(y)$.

Demostración. Definimos $f : \mathcal{M}^h_{\min} \longrightarrow \mathcal{M}/\equiv_h$ como $f(h(x)) = [x]$; por definición de $\equiv_h$ y ya que h es sobreyectiva, f es una función biyectiva bien definida: necesitamos comprobar que tanto f como $f^{-1}([x]) = h(x)$ son simulaciones estrictas.

Si $a \rightarrow_{\mathcal{M}^h_{\min}} b$, existen elementos x e y en $\mathcal{M}$ tales que $h(x) = a$, $h(y) = b$ y $x \rightarrow_{\mathcal{M}} y$, y por lo tanto $f(a) = [x] \rightarrow_{\mathcal{M}/\equiv_h} [y] = f(b)$. De manera similar, si $[x] \rightarrow_{\mathcal{M}/\equiv_h} [y]$ entonces existen x' tal que $h(x) = h(x')$ e y' tal que $h(y) = h(y')$, con $x' \rightarrow_{\mathcal{M}} y'$, y por lo tanto $f^{-1}([x]) = h(x') \rightarrow_{\mathcal{M}^h_{\min}} h(y') = f^{-1}([y])$.

Finalmente, $p \in L_{\mathcal{M}/\equiv_h}([x])$ si y solo si $p \in L_{\mathcal{M}}(x')$ para todo x' con $h(x') = h(x)$, si y solo si $p \in L_{\mathcal{M}^h_{\min}}(h(x))$, y por lo tanto f y f^{-1} son estrictas. □

De esta forma, podemos realizar abstracciones bien mediante la aplicación de estados concretos en un dominio abstracto o bien, como haremos en la sección 4.4, mediante la identificación de estados y trabajando de ahí en adelante con las correspondientes clases de equivalencia.

Por último, hay que señalar que aunque $\mathcal{M}^h_{\min}$ es la aproximación más precisa no siempre es posible conseguir una descripción *computable* de $\mathcal{M}^h_{\min}$. La definición de $\to_{\mathcal{M}^h_{\min}}$ se puede reescribir como $x \to_{\mathcal{M}^h_{\min}} y$ si y solo si existen a y b tales que $h(a) = x$, $h(b) = y$ y $a \to_{\mathcal{M}} b$. Esta relación, incluso en el caso de que $\to_{\mathcal{M}}$ sea recursiva, tan solo es en general recursivamente enumerable (r.e. para abreviar). Sin embargo, en la sección 4.4 se desarrollan métodos ecuacionales que, cuando tienen éxito, dan lugar a una descripción computable de $\mathcal{M}^h_{\min}$.

4.4 Abstracciones ecuacionales

Sea $\mathcal{R} = (\Sigma, E \cup A, R)$ una teoría de reescritura. Un método bastante general para definir abstracciones de la estructura de Kripke $\mathcal{K}(\mathcal{R}, k)_\Pi = (T_{\Sigma/E\cup A,k}, (\to^1_{\mathcal{R},k})^\bullet, L_\Pi)$ es mediante la especificación de una teoría ecuacional extendida de la forma

$$(\Sigma, E \cup A) \subseteq (\Sigma, E \cup A \cup E') \,.$$

Como esto define una relación de equivalencia $\equiv_{E'}$ sobre $T_{\Sigma/E\cup A,k}$, dada por

$$[t]_{E\cup A} \equiv_{E'} [t']_{E\cup A} \iff E \cup A \cup E' \vdash t = t' \iff [t]_{E\cup A\cup E'} = [t']_{E\cup A\cup E'} \,,$$

obviamente podemos obtener nuestra abstracción cociente como $\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'}$ y la llamamos *abstracción cociente ecuacional* de $\mathcal{K}(\mathcal{R}, k)_\Pi$ definida por E'.

Pero, ¿puede $\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'}$, que acabamos de definir en términos de la estructura de Kripke subyacente $\mathcal{K}(\mathcal{R}, k)_\Pi$, ser entendida como la estructura de Kripke asociada a *otra* teoría de reescritura? Echemos un vistazo más detallado a la terna

$$\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'} = (T_{\Sigma/E\cup A,k}/\equiv_{E'}, (\to^1_{\mathcal{R},k})^\bullet/\equiv_{E'}, L_{\Pi/\equiv_{E'}}).$$

La primera observación que puede hacerse es que, por definición, tenemos $T_{\Sigma/E\cup A,k}/\equiv_{E'} \cong T_{\Sigma/E\cup A\cup E',k}$. Una segunda observación es que si $\mathcal{R}$ está *libre de k-bloqueo* (k-deadlock free, en inglés), esto es, si se tiene que $(\to^1_{\mathcal{R},k})^\bullet = \to^1_{\mathcal{R},k}$, entonces la teoría de reescritura $\mathcal{R}/E' = (\Sigma, E \cup A \cup E', R)$ está también libre de k-bloqueo y tenemos, bajo algunos requisitos no muy exigentes (véase el lema 4.4 más adelante, en la página 85):

$$(\to^1_{\mathcal{R}/E',k})^\bullet = \to^1_{\mathcal{R}/E',k} = (\to^1_{\mathcal{R},k})^\bullet/\equiv_{E'} \,.$$

Por lo tanto, si $\mathcal{R}$ está libre de k-bloqueo, el candidato obvio para una teoría de reescritura que tenga $\mathcal{K}(\mathcal{R}, k)_\Pi/\equiv_{E'}$ como estructura de Kripke subyacente es la teoría de reescritura $\mathcal{R}/E' = (\Sigma, E \cup A \cup E', R)$. Esto es, simplemente añadimos a $\mathcal{R}$ las ecuaciones E' y no modificamos las reglas R en modo alguno.

¿Cómo de restrictivo es el requisito de que $\mathcal{R}$ esté libre de k-bloqueo? Afortunadamente, no existe una pérdida esencial de generalidad: en la sección 4.5 mostramos que siempre podemos asociar a una teoría de reescritura ejecutable otra teoría $\mathcal{R}_{df}$ que es semánticamente equivalente (desde el punto de vista de ACTL*), libre de bloqueo y que sigue siendo ejecutable.

De esta manera, a un nivel puramente matemático $\mathcal{R}/E'$ parece ser exactamente lo que andamos buscando. Asumiendo que tenemos un algoritmo de A-ajuste de patrones, el problema se plantea en relación con las dos siguientes *cuestiones de ejecutabilidad* sobre $\mathcal{R}/E'$, que son esenciales para que $\mathcal{K}(\mathcal{R},k)_\Pi/\equiv_{E'}$ sea computable y, por lo tanto, para que se pueda utilizar un comprobador de modelos:

- ¿Son las ecuaciones $E \cup E'$ Church-Rosser y terminantes módulo A sobre términos cerrados?
- ¿Son las reglas de R coherentes para términos cerrados en relación con $E \cup E'$ módulo A?

La respuesta a cada una de estas preguntas puede ser tanto afirmativa como negativa. En la práctica, el usuario debería tener el suficiente cuidado a la hora de especificar E' para que la respuesta a la primera pregunta sea afirmativa. En cualquier caso, siempre podemos intentar comprobar si la propiedad se cumple con una herramienta como el comprobador Church-Rosser de Maude (Durán y Meseguer, 2000); si la comprobación falla, podemos intentar completar las ecuaciones con una herramienta de complección Knuth-Bendix, por ejemplo las descritas en Contejean y Marché (1996) o Durán (2000b), para así obtener una teoría $(\Sigma, E'' \cup A)$ equivalente a $(\Sigma, E \cup A \cup E')$ para la que la primera pregunta tiene una respuesta positiva. Del mismo modo, podemos intentar comprobar si las reglas de R son coherentes para términos cerrados en relación con $E \cup E'$ (o con E'') módulo A usando las herramientas descritas en Durán (2000a). Si la comprobación falla, podemos intentar de nuevo completar las reglas R para alcanzar un conjunto de reglas R' semánticamente equivalente, utilizando las mismas herramientas (Durán, 2000a). Mediante este proceso podemos esperar alcanzar una teoría de reescritura *ejecutable* $\mathcal{R}' = (\Sigma, E'' \cup A, R')$ que sea semánticamente equivalente a $\mathcal{R}/E'$. Y entonces podremos usar $\mathcal{R}'$ para intentar demostrar propiedades de $\mathcal{R}$ utilizando un comprobador de modelos.

Pero aún no hemos terminado. ¿Qué ocurre con los predicados de estado Π? Tenemos que recordar (ver la sección 4.1) que estos predicados de estado (posiblemente parametrizados) habrán sido definidos mediantes ecuaciones D en un módulo de Maude que importa la especificación de $\mathcal{R}$ junto con el módulo `MODEL-CHECKER`. La pregunta es si los predicados de estado Π se *preservan* bajo las ecuaciones en E'. Esto realmente podría ser un problema. Antes de responder a la pregunta necesitamos esclarecer cómo queda la definición de la función de etiquetado $L_{\Pi/\equiv_{E'}}$, que viene definida por la fórmula

$$L_{\Pi/\equiv_{E'}}([t]_{E\cup A\cup E'}) = \bigcap_{[x]_{E\cup A}\subseteq[t]_{E\cup A\cup E'}} L_\Pi([x]_{E\cup A})\,.$$

En general, computar esta intersección y conseguir nuevas definiciones ecuacionales D' que capturen la nueva función de etiquetado $L_{\Pi/\equiv_{E'}}$ no va a ser una tarea sencilla. Todo resulta mucho más fácil si los predicados de estado Π son *preservados* por las ecuaciones en E'. Por definición, decimos que los predicados de estado se preservan bajo las ecuaciones E' si para todo $[t]_{E\cup A}, [t']_{E\cup A} \in T_{\Sigma/E\cup A,k}$ tenemos la implicación

$$[t]_{E\cup A\cup E'} = [t']_{E\cup A\cup E'} \quad\Longrightarrow\quad L_\Pi([t]_{E\cup A}) = L_\Pi([t']_{E\cup A})\,.$$

Hay que señalar que en este caso, asumiendo que las ecuaciones $E \cup E' \cup D$ (o $E'' \cup D$) son Church-Rosser y terminantes para términos cerrados módulo A, *no necesitamos cambiar las ecuaciones* D para definir los predicados de estado Π sobre $\mathcal{R}/E'$ (o sobre la semánticamente equivalente $\mathcal{R}'$). Por lo tanto, tenemos un isomorfismo (dado por un par de funciones de bisimulación invertibles)

$$\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'} \cong \mathcal{K}(\mathcal{R}/E', k)_\Pi ,$$

o, en caso de que necesitemos la teoría $\mathcal{R}'$ semánticamente equivalente, un isomorfismo

$$\mathcal{K}(\mathcal{R}, k)_\Pi / \equiv_{E'} \cong \mathcal{K}(\mathcal{R}', k)_\Pi .$$

El punto crucial en ambos isomorfismos es que la función de etiquetado de la estructura de Kripke en el lado derecho está ahora definida ecuacionalmente por las mismas ecuaciones D que antes. Puesto que, por construcción, bien $\mathcal{R}/E'$ o $\mathcal{R}'$ son teorías ejectubles, para un estado inicial $[t]_{E \cup A \cup E'}$ que tenga un conjunto *finito* de estados alcanzables podemos usar el comprobador de modelos de Maude para comprobar cualquier fórmula en LTL en esta abstracción cociente ecuacional. Más aún, puesto que el AP_Π-morfismo cociente

$$\mathcal{K}(\mathcal{R}, k)_\Pi \longrightarrow \mathcal{K}(\mathcal{R}/E', k)_\Pi$$

es por construcción *estricto*, por el teorema 4.1 va a reflejar la satisfacción de fórmulas arbitrarias en $\mathrm{ACTL}^*(AP_\Pi)$.

Pero sigue quedando pendiente un problema práctico: ¿cómo podemos *probar* la implicación

$$[t]_{E \cup A \cup E'} = [t']_{E \cup A \cup E'} \quad \Longrightarrow \quad L_\Pi([t]_{E \cup A}) = L_\Pi([t']_{E \cup A})$$

para mostrar la deseada preservación de los predicados de estado? Un caso particularmente sencillo es el de teorías de reescritura *k-encapsuladas*. Se trata de teorías en las que la familia k de los estados solo aparece como codominio de un único operador $f : k_1 \ldots k_n \longrightarrow k$ y no aparece como argumento en ningún operador de Σ. Esta no es una restricción muy exigente ya que cualquier teoría de reescritura puede ser transformada en otra semánticamente equivalente que además es k'-encapsulada, simplemente encerrando los estados originales en la familia k en nuevos estados de una familia k' a través de un operador $\{\_\} : k \longrightarrow k'$, como se explica en el siguiente lema.

Lema 4.3 *Dada una teoría de reescritura* $\mathcal{R} = (\Sigma, E, R)$ *y una familia* $k \in \Sigma$, *definimos la teoría de reescritura* $\mathcal{R}' = (\Sigma', E, R)$ *con* Σ' *extendiendo* Σ *con una nueva familia* k' *y un operador* $\{\_\} : k \longrightarrow k'$. *La teoría* $\mathcal{R}'$ *así definida es* k'*-encapsulada.*

Además, si Π *es un conjunto de predicados de estado para* $\mathcal{R}$ *definido por un conjunto de ecuaciones* D, *definimos el conjunto de predicados de estado* Π *para* $\mathcal{R}'$ *transformando cada ecuación* $(\forall \vec{x})\,(t \models p) = b$ **if** C *en* D *en* $(\forall \vec{x})\,(\{t\} \models p) = b$ **if** C. *Se tiene entonces que la función* $h : T_{\Sigma'/E,k'} \longrightarrow T_{\Sigma/E,k}$ *dada por* $h([\{t\}]_E) = [t]_E$ *define una bisimulación biyectiva* $\mathcal{K}(\mathcal{R}, k)_\Pi \cong \mathcal{K}(\mathcal{R}', k')_\Pi$.

Demostración. Dado que no se han añadido nuevas ecuaciones o reglas a $\mathcal{R}'$, resulta inmediato que $\{t\} \rightarrow^1_{\mathcal{R}',k'} \{t'\}$ si y solo si $t \rightarrow^1_{\mathcal{R},k} t'$. Pero entonces, ya que h lleva el término $\{t\}$ a t, se tiene que la relación de transición se preserva en ambas direcciones. En cuanto

a los predicados de estado, por la transformación aplicada a las ecuaciones en D y, de nuevo, ya que no se han añadido ecuaciones nuevas a $\mathcal{R}'$, se tiene que $L_\Pi(\{t\}) = L_\Pi(t)$ de donde se sigue el resultado. □

Hacemos ahora explícitas las condiciones bajo las que una teoría de reescritura libre de k-bloqueo lo continúa siendo tras añadir un conjunto de ecuaciones.

Lema 4.4 *Supongamos que $\mathcal{R} = (\Sigma, E \cup A, R)$ es una teoría de reescritura k-encapsulada y que E' es un conjunto de ecuaciones de la forma $(\forall \vec{x})\, t = t'$ **if** C, con $t, t' \in T_{\Sigma,k}(\vec{x})$. Entonces, si $\mathcal{R}$ está libre de k-bloqueo y en las condiciones de las reglas de reescritura en R no aparecen términos de la familia k, la teoría de reescritura $\mathcal{R}/E' = (\Sigma, E \cup A \cup E', R)$ está también libre de k-bloqueo y se tiene que*

$$\rightarrow^1_{\mathcal{R}/E',k'} = \rightarrow^1_{\mathcal{R},k'} /\equiv_{E'} .$$

para todas las familias k' en Σ.

Demostración. Está claro que $\mathcal{R}/E'$ está libre de k-bloqueo porque cada reescritura en $\mathcal{R}$ es también una reescritura en $\mathcal{R}/E'$. Por la misma razón, la segunda relación está incluida en la primera. Ahora asumimos que $[t] \rightarrow^1_{\mathcal{R}/E',k'} [t']$; por inducción sobre la definición de $\rightarrow^1_{\mathcal{R}/E',k'}$:

- Si existe una regla $l \longrightarrow r$ **if** C en R y una sustitución cerrada θ tal que $[t] = [\theta(l)]$, $[t'] = [\theta(r)]$ y $E \cup A \cup E' \vdash (\forall \emptyset)\, \theta(C)$ entonces, debido a las restricciones sobre E' y R, se tiene que $E \cup A \vdash (\forall \emptyset)\, \theta(C)$ (ver lema 4.5 para los detalles de una demostración similar) y por lo tanto $[t] = [\theta(l)] \rightarrow^1_{\mathcal{R},k'} /\equiv_{E'} [\theta(r)] = [t']$.

- Si $[t] = [f(u_1, \ldots, u_n)]$, $[t'] = [f(u'_1, \ldots, u'_n)]$ y $[u_i] \rightarrow^1_{\mathcal{R}/E',k_i} [u'_i]$ para algún i, el resultado se sigue por la hipótesis de inducción. □

Ahora ya podemos dar una condición suficiente bajo la cual la preservación de proposiciones atómicas está garantizada.

Proposición 4.4 *Supongamos que $\mathcal{R} = (\Sigma, E \cup A \cup E', R)$ es una teoría de reescritura k-encapsulada en la que todas las ecuaciones en E' son de la forma $(\forall \vec{x})\, t = t'$ **if** C, con $t, t' \in T_{\Sigma,k}(\vec{x})$, y que las ecuaciones $E \cup E' \cup D$ son Church-Rosser y terminantes módulo los axiomas ecuacionales A. Además, supongamos que no existen ecuaciones entre términos de la familia k en las condiciones de las ecuaciones en E'. Si para cada ecuación $(\forall \vec{x})\, t = t'$ **if** C en E' y cada predicado de estado p podemos demostrar la propiedad inductiva*

$$E \cup A \cup D \vdash_{ind} (\forall \vec{x}, \vec{y})\; C \rightarrow (t(\vec{x}) \models p(\vec{y}) = true \;\leftrightarrow\; t'(\vec{x}) \models p(\vec{y}) = true) \qquad (\dagger)$$

habremos establecido la preservación de los predicados de estado Π por las ecuaciones E'.

Demostración. Para demostrar que, para u y v términos cerrados de la familia k, $[u]_{E \cup A \cup E'} = [v]_{E \cup A \cup E'}$ implica $L_\Pi([u]_{E \cup A}) = L_\Pi([v]_{E \cup A})$, procedemos mediante inducción estructural sobre la derivación de $E \cup A \cup E' \vdash (\forall \emptyset)\, u = v$:

- **(Reflexividad)**, **(Simetría)**, **(Transitividad)** y **(Pertenencia)** son inmediatos.
- **(Congruencia)**. Por el lema 4.5 más abajo, si

$$\frac{E \cup A \cup E' \vdash (\forall \emptyset)\, u_1 = v_1 \quad \ldots \quad E \cup A \cup E' \vdash (\forall \emptyset)\, u_n = v_n}{E \cup A \cup E' \vdash (\forall \emptyset)\, f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)}$$

  es el último paso en una derivación en $E \cup A \cup E'$, entonces $E \cup A \vdash (\forall \emptyset)\, u_i = v_i$ para todo $1 \leq i \leq n$ y por lo tanto $E \cup A \vdash (\forall \emptyset)\, f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)$, de donde se sigue el resultado.
- **(Reemplazamiento)**. Supongamos que

$$\frac{E \cup A \cup E' \vdash (\forall \emptyset)\, \theta(C)}{E \cup A \cup E' \vdash (\forall \emptyset)\, \theta(t = t')}\,,$$

  con $u = \theta(t)$ y $v = \theta(t')$, es el último paso en una derivación en $E \cup A \cup E'$ para alguna ecuación $(\forall \vec{x})\, t = t'$ **if** C en $E \cup A \cup E'$ y sustitución cerrada θ. Por nuestras hipótesis y el lema 4.5 más abajo se tiene que $E \cup A \vdash (\forall \emptyset)\, \theta(C)$. Así, si la ecuación pertenece a $E \cup A$ se sigue que $E \cup A \vdash (\forall \emptyset)\, \theta(t = t')$ y se tiene el resultado. En caso contrario, puesto que $E \cup A \cup D \vdash (\forall \emptyset)\, \theta(C)$ obviamente se cumple, usando la hipótesis

$$E \cup A \cup D \vdash_{ind} (\forall \vec{x}, \vec{y})\, C \rightarrow (t(\vec{x}) \models p(\vec{y}) = true \;\leftrightarrow\; t'(\vec{x}) \models p(\vec{y}) = true)$$

  para cada sustitución cerrada θ' que extienda θ a las variables en $\vec{y}$, se tiene

$$E \cup A \cup D \vdash_{ind} (\forall \emptyset)\, \theta'(t \models p = true \;\leftrightarrow\; t' \models p = true)$$

  de manera que $L_\Pi([\theta(t)]_{E \cup A}) = L_\Pi([\theta(t')]_{E \cup A})$, como se quería. □

Lema 4.5 *Bajo las condiciones de la proposición 4.4, para todos los términos $u, v \in T_{\Sigma,k'}(\vec{x})$ con k' distinto de k, se cumple que*

$$E \cup A \cup E' \vdash (\forall \vec{x})\, u = v \quad \textit{implica} \quad E \cup A \vdash (\forall \vec{x})\, u = v\,.$$

Demostración. Por inducción estructural sobre la derivación de $E \cup A \cup E' \vdash (\forall \vec{x})\, u = v$:

- **(Reflexividad)**, **(Simetría)**, **(Transitividad)** y **(Pertenencia)**. Trivial.
- **(Congruencia)**. Si

$$\frac{E \cup A \cup E' \vdash (\forall \vec{x})\, u_1 = v_1 \quad \ldots \quad E \cup A \cup E' \vdash (\forall \vec{x})\, u_n = v_n}{E \cup A \cup E' \vdash (\forall \vec{x})\, f(u_1, \ldots, u_n) = f(v_1, \ldots, v_n)}$$

  es el último paso de una derivación en $E \cup A \cup E'$ entonces, dado que la teoría es k-encapsulada, ninguno de los u_i o los v_i pertenece a la familia k y podemos aplicar la hipótesis de inducción para obtener $E \cup A \vdash (\forall \vec{x})\, u_i = v_i$, de donde se sigue el resultado.

- **(Reemplazamiento)**. Si

$$\frac{E \cup A \cup E' \vdash (\forall \vec{x})\, \theta(C)}{E \cup A \cup E' \vdash (\forall \vec{x})\, \theta(t = t')}$$

es el último paso en una derivación en $E \cup A \cup E'$ para alguna ecuación $(\forall \vec{y})\, t = t'$ **if** C en E (nótese que por la hipótesis no puede pertenecer a E') y sustitución $\theta : \vec{y} \longrightarrow T_{\Sigma,k}(\vec{x})$, podemos aplicar la hipótesis de inducción sobre $\theta(C)$ ya que no puede contener ecuaciones entre términos de la familia k, y se tiene el resultado. □

Como consecuencia de la proposición 4.4, para demostrar que los predicados de estado Π se preservan en una abstracción ecuacional se puede, bajo las hipótesis señaladas más arriba sobre $\mathcal{R}$, E' y D, utilizar una herramienta como el ITP para resolver las obligaciones de prueba (proof obligations) de la forma (†) en el enunciado de dicha proposición.

Resulta interesante destacar que el hecho de que la familia del estado esté encapsulada no excluye el uso de estructuras de datos recursivas en las componentes del estado como una variable de historia. Por ejemplo, el caso práctico que se discute en la sección 4.6.2 ciertamente muestra estados encapsulados en los que aparecen dichas estructuras recursivas. En realidad, el requisito de encapsulamiento no impone en la práctica ninguna restricción real ya que el lema 4.3 permite transformar cualquier teoría de reescritura $\mathcal{R}$ con k como familia del estado en otra equivalente que sea k'-encapsulada.

4.5 La dificultad con el bloqueo

El motivo por el que nos hemos concentrado en teorías de reescritura libres de bloqueo es porque los bloqueos pueden suponer un problema, debido a un detalle técnico en la semántica de estructuras de Kripke de la lógica temporal. Como se enfatizó en su definición en la sección 2.1, la relación de transición de una estructura de Kripke es *total* y este requisito también se impone en las estructuras de Kripke que surgen a partir de teorías de reescritura.

Consideremos la siguiente especificación de una teoría de reescritura, junto con la declaración de dos predicados de estado.

```
mod FOO is
  inc MODEL-CHECKER .
  ops a b c : -> State .
  ops p1 p2 : -> Prop .

  eq (a |= p1) = true .
  eq (b |= p2) = true .
  eq (c |= p1) = true .

  rl a => b .
  rl b => c .
endm
```

La relación de transición de la estructura de Kripke correspondiente tiene tres elementos: `a` $\rightarrow$ `b`, `b` $\rightarrow$ `c` y `c` $\rightarrow$ `c`, el último de los cuales se añade de manera consistente como una transición de bloqueo de acuerdo con la definición de $(\rightarrow^1_{\mathcal{R},\texttt{[State]}})^\bullet$.

Supongamos ahora que quisiéramos abstraer este sistema y que decidiéramos identificar `a` y `c` por medio de un simulación h. Para ello, de acuerdo con las secciones anteriores sería suficiente con añadir la ecuación

```
eq c = a .
```

a la especificación anterior. El sistema resultante es coherente y `a` y `c` satisfacen los mismos predicados de estado. Nótese que la correspondiente estructura de Kripke tiene solo dos pares en su relación de transición: uno desde la clase de equivalencia de `a` a la de `b` y otro en sentido opuesto. Ahora, puesto que no se puede dar una situación de bloqueo en ninguno de los estados, para el conjunto actual de ecuaciones {`c` = `a`} tenemos $(\rightarrow_{\mathcal{R}/E',\texttt{[State]}})^\bullet = \rightarrow_{\mathcal{R}/E',\texttt{[State]}}$ de manera que no hay que añadir transiciones de bloqueo adicionales. En particular, no existe una transición desde la clase de equivalencia de `a` en sí misma, lo que significa que la especificación resultante no se corresponde con el sistema minimal asociado a h, en el que tal transición sí que existe. La ausencia de esta transición "perezosa" es un problema serio porque ahora es posible demostrar propiedades en el sistema que supuestamente está simulando el original que en realidad son falsas en este, como por ejemplo $\Box\Diamond$`p2`.

Una forma sencilla de tratar esta dificultad consiste simplemente en añadir transiciones perezosas para cada uno de los estados en la especificación resultante por medio de una regla de la forma `x => x`. El sistema resultante, además de contener todas las reglas del sistema minimal, puede de hecho incluir algunas transiciones "basura" que no son parte de aquel. Por lo tanto, terminaríamos con un sistema que se puede utilizar de manera segura para inferir propiedades del sistema original (es inmediato ver que se tiene una simulación) pero que en general sería más tosco que el sistema minimal.

Una mejor alternativa de encarar el problema es caracterizar el conjunto de estados con bloqueo. Para ello, dada una teoría de reescritura $\mathcal{R}$ introducimos un nuevo operador *enabled* : $k \longrightarrow$ [*Bool*] para cada familia k en $\mathcal{R}$ que será *true* para un término si y solo si existe una regla que se le puede aplicar al mismo.

Proposición 4.5 *Dada una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$, definimos una extensión (Σ', E') de su parte ecuacional añadiendo:*

1. *para cada familia k en Σ, un nuevo operador enabled : $k \longrightarrow$ [Bool] en Σ';*
2. *para cada regla $(\forall \vec{x})\, l \rightarrow r$ **if** C en R, una ecuación $(\forall \vec{x})\, enabled(l) = true$ **if** C en E', y*
3. *para cada operador $f : k_1 \ldots k_n \longrightarrow k$ en Σ y para cada i con $1 \leq i \leq n$, la ecuación $(\forall \vec{x})\, enabled(f(x_1, \ldots, x_n)) = true$ **if** $enabled(x_i) = true$.*

Entonces, para cada término $t \in T_\Sigma$,

$$E' \vdash_{ind} (\forall \emptyset)\, enabled(t) = true \iff \text{existe } t' \in T_\Sigma \text{ tal que } t \rightarrow^1_{\mathcal{R},k} t' .$$

Demostración. Señalemos en primer lugar que, puesto que los términos son cerrados, la ecuación se cumple en el modelo inicial si y solo se cumple en todos los modelos. Demostramos la implicación de izquierda a derecha por inducción sobre la derivación. Los únicos casos no triviales ocurren cuando la última regla de inferencia usada es o bien **(Reemplazamiento)** o bien **(Transitividad)**. En el caso de **(Reemplazamiento)**, la ecuación utilizada debe haber sido una de las añadidas a E porque *enabled* es un operador nuevo. Asumimos entonces que para $(\forall \vec{x})\, enabled(l) = true$ **if** C en E' y una sustitución cerrada θ,

$$\frac{E' \vdash (\forall \emptyset)\, \theta(C)}{E' \vdash (\forall \emptyset)\, \theta(enabled(l)) = true}$$

es el último paso en una derivación en E' donde $l \longrightarrow r$ **if** C es una regla de R. Por el lema 4.6 que veremos más abajo, $E \vdash (\forall \emptyset)\, \theta(C)$ y por lo tanto $\theta(l) \rightarrow^1_{\mathcal{R},k} \theta(r)$. Cuando la ecuación utilizada es $(\forall \vec{x})\, enabled(f(x_1, \ldots, x_n)) = true$ **if** $enabled(x_i) = true$, el resultado se sigue por la hipótesis de inducción y la regla de **(Congruencia)** del cálculo de pruebas de la lógica de reescritura. Finalmente, en el caso de **(Transitividad)**,

$$\frac{E' \vdash (\forall \emptyset)\, enabled(t) = t' \qquad E' \vdash (\forall \emptyset)\, t' = true}{E' \vdash (\forall \emptyset)\, enabled(t) = true}\,.$$

Por el lema 4.7 más abajo podemos distinguir dos casos. Si t' es *true* o si existe una derivación más pequeña de $E' \vdash (\forall \emptyset)\, enabled(t) = true$, podemos aplicar la hipótesis de inducción. Si t' es $enabled(t'')$ para algún t'' tal que $E' \vdash (\forall \emptyset)\, t = t''$, el resultado se cumple aplicando la hipótesis de inducción sobre $E' \vdash (\forall \emptyset)\, t' = true$ y el hecho de que $E \vdash (\forall \emptyset)\, t = t''$ por el lema 4.6.

La implicación en el otro sentido se demuestra por inducción sobre la definición de $\rightarrow^1_{\mathcal{R},k}$. Si t es $\theta(l)$ y t' es $\theta(r)$ para alguna sustitución θ y regla $l \longrightarrow r$ **if** C en R, el resultado se obtiene instanciando la ecuación correspondiente de entre aquellas añadidas a E'. Si $E \vdash (\forall \emptyset)\, t = u$, $E \vdash (\forall \emptyset)\, t' = v$ y $u \rightarrow^1_{\mathcal{R},k} v$, por hipótesis de inducción $E' \vdash (\forall \emptyset)\, enabled(u) = true$ y por lo tanto $E' \vdash (\forall \emptyset)\, enabled(t) = true$. Finalmente, si t es $f(t_1, \ldots, t_n)$, t' es $f(t'_1, \ldots, t'_n)$ y $t_i \rightarrow^1_{\mathcal{R},k} t'_i$ para algún i, por la hipótesis de inducción tenemos $E' \vdash (\forall \emptyset)\, enabled(t_i) = true$ y, de nuevo, el resultado se obtiene instanciando la ecuación apropiada en E'. □

Lema 4.6 *Bajo las condiciones de la proposición 4.5, para todos los términos* $t, t' \in T_\Sigma(X)$,

$$E' \vdash (\forall X)\, t = t' \quad \textit{implica} \quad E \vdash (\forall X)\, t = t'\,.$$

Demostración. Es inmediato demostrar por inducción que si existe una derivación de $E' \vdash (\forall X)\, t = t'$ entonces también existe una derivación, sin ocurrencias de *enabled*, de $E' \vdash (\forall X)\, u = u'$, donde u y u' se obtienen a partir de t y t' sustituyendo todos los subtérminos de la forma $enabled(w)$ por *true*. Así, cuando $t, t' \in T_\Sigma(X)$ lo que obtenemos es una derivación en E. □

Lema 4.7 *Bajo las condiciones de la proposición 4.5, para todos los términos cerrados* t *y* t'*, si existe una derivación de* $E' \vdash (\forall \emptyset)\, enabled(t) = t'$ *o de* $E' \vdash (\forall \emptyset)\, t' = enabled(t)$*, se cumple alguna de las siguientes posibilidades:*

(a) t' *es true,*

(b) *existe una derivación de* $E' \vdash (\forall \emptyset)\, enabled(t) = true$ *cuya profundidad es menor o igual a la de la anterior, o*

(c) t' *es* $enabled(t'')$ *para algún* t'' *tal que* $E' \vdash (\forall \emptyset)\, t = t''$.

Demostración. Por inducción sobre la derivación. El único caso que no es inmediato es el correspondiente a **(Transitividad)**. Dado

$$\frac{E' \vdash (\forall \emptyset)\, enabled(t) = t'' \qquad E' \vdash (\forall \emptyset)\, t'' = t'}{E' \vdash (\forall \emptyset)\, enabled(t) = t'},$$

aplicamos la hipótesis de inducción a $E' \vdash (\forall \emptyset)\, enabled(t) = t''$. Si bien (a) o (b) se cumplen, entonces (b) también se cumple para la ecuación original. En caso contrario t'' es $enabled(t''')$ y podemos aplicar la hipótesis de inducción a $E' \vdash (\forall \emptyset)\, t'' = t'$: los casos (a) y (c) son inmediatos; y si se cumple (b) existe una derivación de $E' \vdash (\forall \emptyset)\, enabled(t''') = true$ cuya profundidad es menor o igual que la de $E' \vdash (\forall \emptyset)\, enabled(t''') = t'$ y podemos usarla junto con $E' \vdash (\forall \emptyset)\, enabled(t) = enabled(t''')$ para construir una derivación de $E' \vdash (\forall \emptyset)\, enabled(t) = true$ que no sea más profunda que la derivación original. □

El predicado *enabled* y sus propiedades son el ingrediente fundamental en la demostración de la siguiente proposición, que nos permite transformar una teoría de reescritura ejecutable en otra semánticamente equivalente que es tanto ejecutable como libre de bloqueo.

Teorema 4.2 *Sea* $\mathcal{R} = (\Sigma, E \cup A, R)$ *una teoría de reescritura ejecutable. Dada una familia* k *de estados, se puede construir una teoría de reescritura ejecutable extendida* $\mathcal{R} \subseteq \mathcal{R}^k_{df} = (\Sigma', E' \cup A, R')$ *tal que:*

- $\mathcal{R}^k_{df}$ *está libre de* k'*-bloqueo y es* k'*-encapsulada para una cierta familia* k'*;*
- *existe una función* $h : T_{\Sigma',k'} \longrightarrow T_{\Sigma,k}$ *que induce una biyección* $h : T_{\Sigma'/E' \cup A,k'} \longrightarrow T_{\Sigma/E \cup A,k}$ *tal que para cada* $t, t' \in T_{\Sigma',k'}$ *se tiene*

$$h(t)(\rightarrow^1_{\mathcal{R},k})^\bullet h(t') \iff t \rightarrow^1_{\mathcal{R}^k_{df},k'} t'.$$

Más aún, si Π *es un conjunto de predicados de estado para* $\mathcal{R}$ *y* k *definido por las ecuaciones* D*, podemos definir predicados de estado* Π *para* $\mathcal{R}^k_{df}$ *y* k' *mediante ecuaciones* D' *tales que la función* h *se transforma en una* AP_Π*-bisimulación biyectiva*

$$h : \mathcal{K}(\mathcal{R}^k_{df}, k')_\Pi \longrightarrow \mathcal{K}(\mathcal{R}, k)_\Pi.$$

Demostración. $\mathcal{R}^k_{df}$ se define extendiendo la teoría ecuacional (Σ, E) en $\mathcal{R}$ con un predicado *enabled* de la manera explicada en la proposición 4.5 y añadiendo una nueva familia k',

un nuevo operador $\{\_\} : k \longrightarrow k'$ y la regla[4]

$$(\forall \{x\})\, \{x\} \rightarrow \{x\} \textbf{ if } enabled(x) \neq true$$

al conjunto R.

Por construcción, está claro que $\mathcal{R}^k_{df}$ es k'-encapsulada. Dado un término cerrado $\{t\}$ con t perteneciente a la familia k, si existe t' en $\mathcal{R}$ tal que $t \rightarrow^1_{\mathcal{R},k} t'$ entonces $\{t\} \rightarrow^1_{\mathcal{R}^k_{df},k'} \{t'\}$; en caso contrario, por la proposición 4.5 se cumple la condición de la regla que acabamos de añadir y se tiene $\{t\} \rightarrow^1_{\mathcal{R}^k_{df}} \{t\}$. De esta manera $\mathcal{R}^k_{df}$ está libre de k'-bloqueo. La función h se puede definir como $h(\{t\}) = t$ y, debido a que no se han introducido ecuaciones entre términos de la familia k', induce una biyección que claramente satisface la equivalencia del segundo punto.

Para terminar, en relación con los predicados de estado, transformamos cada ecuación $(\forall \vec{x})\, (t \models p) = b \textbf{ if } C$ en $(\forall \vec{x})\, (\{t\} \models p) = b \textbf{ if } C$, como en el lema 4.3. Esto implica que $L_\Pi(\{t\}) = L_\Pi(t)$ y, junto con los resultados previos, que h es una bisimulación estricta. □

4.6 Casos prácticos

En esta sección mostramos en detalle la aplicación de las técnicas introducidas hasta ahora con cuatro ejemplos: el protocolo de la panadería y tres protocolos de comunicación.

4.6.1 El protocolo de la panadería otra vez

Podemos utilizar el protocolo de la panadería para ilustrar cómo se pueden usar las abstracciones cociente ecuacionales para verificar sistemas con un número infinito de estados alcanzables. Una abstracción tal se puede definir añadiendo a las ecuaciones en la especificación de `BAKERY-CHECK` en la página 75 un conjunto E' de ecuaciones adicionales que definan un cociente del conjunto de estados. Esto lo hacemos en el siguiente módulo que extiende `BAKERY-CHECK` con algunas ecuaciones y no cambia las reglas de reescritura que expresan las transiciones.

```
mod ABSTRACT-BAKERY-CHECK is
  inc BAKERY-CHECK .

  vars P Q : Mode .
  vars X Y : Nat .

  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
```

[4] Es importante señalar que se utiliza una *desigualdad* en la condición de la regla. Esto está permitido en la implementación de la lógica de reescritura en Maude bajo supuestos de Church-Rosser y terminación adecuados, pero no en la lógica de reescritura en sí misma. Sin embargo, gracias al metateorema de Bergstra y Tucker (1980) mencionado en la nota al pie de la página 76, bajo las condiciones de la proposición es posible definir esta desigualdad de un modo ecuacional. El motivo para no hacerlo explícito aquí es porque es mucho más conveniente y conciso expresar la regla de esta manera, que además está soportada internamente por Maude mediante el predicado de desigualdad `_=/=_`, que es la negación del predicado de igualdad `_==_`.

```
  eq < P, s s X, Q, 0 > = < P, s 0, Q, 0 > .
  ceq < P, s X, Q, s Y > = < P, s s 0, Q, s 0 >
      if (Y < X) /\ not(Y == 0 and X == s 0) .
  ceq < P, s X, Q, s Y > = < P, s 0, Q, s 0 >
      if not (Y < X) /\ not (Y == 0 and X == 0) .
endm
```

Nótese que de acuerdo con las ecuaciones de arriba $< P,N,Q,M > \equiv < P',N',Q',M' >$ si y solo si:

1. $P = P'$ y $Q = Q'$,
2. $N = 0$ si y solo si $N' = 0$,
3. $M = 0$ si y solo si $M' = 0$,
4. $M < N$ si y solo si $M' < N'$.

Intuitivamente, no nos importa el valor concreto de las variables sino tan solo cuál de ellas es mayor y si alguna es igual a cero.

Tres cuestiones fundamentales son:

- ¿Es ahora el conjunto de estados finito?
- ¿Corresponde esta abstracción con una teoría de reescritura cuyas ecuaciones son Church-Rosser y terminantes para términos cerrados?
- ¿Siguen siendo las reglas coherentes para términos cerrados?

Las ecuaciones son ciertamente Church-Rosser y terminantes para términos cerrados porque no existe solapamiento entre los lados izquierdos de las ecuaciones y los lados derechos tienen valores más pequeños para los números asociados a los procesos. También está claro que el conjunto de estados es ahora finito, ya que en las formas canónicas que se obtienen con estas ecuaciones los números naturales posibles en el estado no pueden ser nunca mayores que `s(s(0))`. Esto nos deja solo con la cuestión de la coherencia. Tenemos que analizar los posibles "pares críticos relativos" entre las reglas y las ecuaciones. Por ejemplo, consideremos el siguiente par formado por una regla y una ecuación.

```
  rl [p1_sleep] : < sleep, X, Q, Y > => < wait, s Y, Q, Y > .
  eq < P, 0, Q, s s Y > = < P, 0, Q, s 0 > .
```

El único solapamiento posible corresponde a la unificación (después de hacer que los conjuntos de variables sean disjuntos) de los dos lados izquierdos, lo que da lugar al término `< sleep, 0, Q, s s Y >`, que es reescrito por la regla en `< wait, s s s Y, Q, s s Y >` y por la ecuación en `< sleep, 0, Q, s 0 >`, y ambos términos se reducen finalmente a `< wait, s s 0, Q, s 0 >`, en el primer caso mediante la tercera ecuación y en el segundo caso mediante la regla `[p1_sleep]`. De la misma forma se demuestra que todos los demás pares regla-ecuación también son coherentes para términos cerrados. Por lo tanto el módulo es coherente.

Una cuarta cuestión que ha quedado pendiente es si BAKERY-CHECK está libre de bloqueo. Para demostrar que ciertamente este es el caso podemos especificar un predicado *enabled*, tal y como se explicó en la sección 4.5, que devuelva *true* cuando se aplique a un término si ese término representa un estado libre de bloqueo. Necesitamos las siguientes ecuaciones:

```
eq enabled(< sleep, X, Q, Y >) = true .
eq enabled(< wait, X, Q, 0 >) = true .
ceq enabled(< wait, X, Q, Y >) = true if not (Y < X) .
eq enabled(< crit, X, Q, Y >) = true .
eq enabled(< P, X, sleep, Y >) = true .
eq enabled(< P, 0, wait, Y >) = true .
ceq enabled(< P, X, wait, Y >) = true if Y < X .
eq enabled(< P, X, crit, Y >) = true .
```

La ecuación que tenemos que probar para asegurar que la teoría está libre de bloqueo es entonces

```
eq enabled(S:State) = true .
```

La prueba se puede hacer en el ITP por inducción sobre la primera y la tercera componentes del ITP. O alternativamente, podríamos probar el resultado de una manera más automática utilizando un *comprobador de completitud suficiente* para Maude desarrollado recientemente por Hendrix (2003). Esta herramienta toma como entrada un módulo y comprueba si es suficientemente completo, en el sentido intuitivo de que hay suficientes ecuaciones para que cada término pueda ser reducido a una forma canónica en la que solo aparezcan constructores. En nuestro caso la herramienta nos dice que el módulo es suficientemente completo lo que significa, en particular, que todos los términos de la forma *enabled*(*t*) pueden ser reducidos a un término canónico de tipo *Bool* y, debido a las ecuaciones utilizadas, este término debe ser *true* como se pedía.

¿Qué ocurre con los predicados de estado? ¿Son preservados por la abstracción? Señalemos que como la teoría de reescritura es [State]-encapsulada y todas las ecuaciones son entre términos de la familia [State], según la proposición 4.4 solo necesitamos comprobar que cada una de las ecuaciones preserva los predicados de estado. Pero esto es trivial ya que los predicados solo dependen de los componentes Mode, que no son alterados por las ecuaciones. Esto se puede comprobar de manera mecánica con el ITP. Por ejemplo, para demostrar que la tercera ecuación preserva 1wait se haría:

```
(goal pred1 : BAKERY-PROOF |- A{P:Mode ; Q:Mode ; X:Nat ; Y:Nat}
      (((Y:Nat < X:Nat) = (true)) =>
       (((< P:Mode, (s s 0), Q:Mode, (s 0) > |= 1wait) = (true)) =>
        ((< P:Mode, (s X:Nat), Q:Mode, (s Y:Nat) > |= 1wait) = (true)))) .)
(auto* .)


(goal pred2 : BAKERY-PROOF |- A{P:Mode ; Q:Mode ; X:Nat ; Y:Nat}
      (((Y:Nat < X:Nat) = (true)) =>
```

```
      (((< P:Mode, (s X:Nat), Q:Mode, (s Y:Nat) > |= 1wait) = (true)) =>
       ((< P:Mode, (s s 0), Q:Mode, (s 0) > |= 1wait) = (true)))) .)
(auto* .)
```

Esto demuestra las dos implicaciones que constituyen la equivalencia en la proposición 4.4.

En otras palabras, acabamos de demostrar que, para Π el conjunto de predicados de estado declarado en el módulo BAKERY-CHECK (página 75), tenemos un Π-morfismo cociente estricto,

$$\mathcal{K}(\texttt{BAKERY-CHECK}, \texttt{State})_{\Pi} \longrightarrow \mathcal{K}(\texttt{ABSTRACT-BAKERY-CHECK}, \texttt{State})_{\Pi}.$$

Por lo tanto, podemos establecer la propiedad de exclusión mutua de BAKERY-CHECK utilizando el comprobador de modelos de Maude sobre ABSTRACT-BAKERY-CHECK con:

```
Maude> reduce modelCheck(initial, []~ (1crit /\ 2crit)) .
result Bool: true
```

Del mismo modo podemos establecer la propiedad de vivacidad de BAKERY-CHECK comprobando el modelo ABSTRACT-BAKERY-CHECK.

```
Maude> reduce modelCheck(initial, (1wait |-> 1crit) /\ (2wait |-> 2crit)) .
result Bool: true
```

4.6.2 Un protocolo de comunicación

Nuestro segundo ejemplo es un protocolo para la comunicación ordenada de mensajes entre un emisor y un receptor en un medio asíncrono (Meseguer, 2002, lección 28). Para garantizar que los mensajes se reciben en el orden correcto, los mensajes contienen un número de secuencia y tanto el emisor como el receptor mantienen un contador que se refiere al mensaje con el que están trabajando actualmente. El emisor puede, en cualquier momento, elegir de manera no determinista el siguiente valor (dentro del conjunto {a, b, c} en esta presentación) que es entonces emparejado con el contador del emisor para componer un mensaje que a continuación se libera en el medio; el propio valor también se añade al final de una lista de valores enviados propiedad del emisor. El receptor mantiene a su vez una lista análoga de valores recibidos. El propósito de estas listas es permitirnos articular la propiedad que nos interesa demostrar. Cuando el receptor "ve" un mensaje con un número de secuencia igual a su contador actual lo retira del medio y añade su valor a su lista de valores recibidos.

La siguiente es la especificación en Maude del protocolo, donde solo hay tres clases distintas de mensajes. Los estados se representan mediante ternas < S, MS, R >, donde S representa el estado local del emisor, R el del receptor y MS el medio asíncrono (una "sopa de mensajes").

```
mod PROTOCOL is
  protecting NAT .
```

```
  sorts Value ValueList LocalState Message MessageSoup State .
  subsort Value < ValueList .
  subsort Message < MessageSoup .

  ops a b c : -> Value .
  op nil : -> ValueList .
  op _:_ : ValueList ValueList -> ValueList [assoc id: nil] .
  op ls : Nat ValueList -> LocalState .

  op msg : Nat Value -> Message .
  op null : -> MessageSoup .
  op _;_ : MessageSoup MessageSoup -> MessageSoup [assoc comm id: null] .

  op <_,_,_> : LocalState MessageSoup LocalState -> State .
  op initial : -> State .

  vars N M : Nat .
  var X : Value  .
  vars L L1 L2 : ValueList .
  var MS : MessageSoup .
  vars R S : LocalState .

  eq initial = < ls(0, nil), null, ls(0, nil) > .

  rl < ls(N, L), MS, R > => < ls(s(N), L : a), MS ; msg(N, a), R > .
  rl < ls(N, L), MS, R > => < ls(s(N), L : b), MS ; msg(N, b), R > .
  rl < ls(N, L), MS, R > => < ls(s(N), L : c), MS ; msg(N, c), R > .
  rl < S, msg(N, X) ; MS, ls(N, L) > => < S, MS, ls(s(N), L : X) > .
endm
```

En esta especificación, los términos de tipo `LocalState`, construidos con el operador `ls`, se usan para representar el estado local tanto del emisor como del receptor. El primer argumento de `ls` corresponde al contador mientras que el segundo es la lista de mensajes que ya han sido enviados o recibidos. Destacamos el importante uso del ajuste de patrones y la reescritura módulo los axiomas de asociatividad (`assoc`) y de identidad (`id`) para el operador `_:_` de concatenación de listas, y módulo asociatividad (`assoc`), conmutatividad (`comm`) e identidad (`id`) para el operador de unión de multiconjuntos `_;_` que construye sopas de mensajes. Estos axiomas corresponden a los axiomas A en la descripción teórica de una teoría de reescritura $\mathcal{R} = (\Sigma, E \cup A, R)$ y son utilizados por Maude para aplicar las ecuaciones y reglas módulo los mismos.

La propiedad que nos gustaría que nuestro sistema satisfaciera es la de que los mensajes se entregan en el orden correcto. Gracias a las listas del emisor y del receptor, este requisito se puede expresar formalmente mediante la fórmula $\Box$`prefix`, donde `prefix` es una proposición atómica que se cumple en aquellos estados en los que la lista del receptor es un prefijo de la lista del emisor. En Maude esto se expresa como sigue:

```
mod PROTOCOL-CHECK is
  inc MODEL-CHECKER .
  inc PROTOCOL .
```

```
  op prefix : -> Prop .
  vars M N : Nat .
  vars L1 L2 : ValueList .
  var MS : MessageSoup .

  eq (< ls(N, L1 : L2), MS, ls(M, L1) > |= prefix) = true .
endm
```

Como ya ocurría para el protocolo de la panadería, no se puede aplicar directamente un comprobador de modelos porque el conjunto de estados alcanzables desde `initial` es infinito. Existen dos razones distintas para la infinitud en este ejemplo. La primera corresponde a los contadores, que son números naturales que pueden tomar valores arbitrariamente grandes. La segunda es el medio de comunicación, que no está limitado y puede contener un número arbitrario de mensajes. De nuevo, para tratar la infinitud y poder aplicar un comprobador de modelos necesitamos definir una abstracción.

En primer lugar, un estado en el que las correspondientes listas del emisor y del receptor tengan como primer elemento el mismo valor puede ser identificado con el estado que resulta de eliminar dicho valor de ambas listas. Esto se puede expresar por medio de la ecuación:

```
  eq < ls(N, X : L1), MS, ls(M, X : L2) > = < ls(N, L1), MS, ls(M, L2) > .
```

En segundo lugar, si en un determinado momento ambos contadores son iguales y no hay mensajes en el medio de comunicación los contadores se pueden reiniciar a cero.

```
  ceq < ls(N, L1), null, ls(N, L2) > = < ls(0, L1), null, ls(0, L2) >
      if N =/= 0 .
```

(La condición en esta ecuación es irrelevante desde un punto de vista matemático, pero resulta necesaria para garantizar la terminación.)

Por último, si en el medio del estado actual hay un mensaje `msg(N, X)` y el contador del receptor es `N`, podemos identificar este estado con otro en el que el mensaje ya ha sido leído por el receptor.

```
  eq < ls(M, L1 : X : L2), msg(N, X) ; MS, ls(N, L1) > =
     < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .
```

La ecuación es incondicional, pero nótese que para forzar que o bien los dos estados satisfagan `prefix` o que ninguno lo haga, se requiere que el término correspondiente al emisor se ajuste a un cierto patrón en el lado izquierdo de la ecuación.

Antes de poder aplicar un comprobador de modelos a este nuevo sistema debemos de nuevo preguntarnos si las ecuaciones siguen siendo Church-Rosser y terminantes, las reglas coherentes y si los predicados se preservan. La propiedad de terminación es clara porque el número de mensajes siempre decrece.

La propiedad de Church-Rosser no es tan inmediata debido al solapamiento entre la primera y la tercera ecuaciones: si el siguiente mensaje que vaya a ser entregado aparece

también como cabeza de la lista de mensajes asociados al emisor y al receptor, podríamos tanto añadirlo al final de la lista del receptor usando la tercera ecuación, como eliminarlo de ambas listas usando la primera. Sin embargo en este último caso no parece posible seguir reduciendo (ecuacionalmente) el estado, pese a lo cual la propiedad de Church-Rosser se cumple. Informalmente, lo que sucede es que para que la tercera ecuación se pueda aplicar el emisor y el receptor han de ser tales que se puedan eliminar todos los mensajes de la lista del receptor; tras esto, el mensaje se puede añadir al final de la lista del receptor como se quería. Nos hubiera gustado confirmar esta intuición con la herramienta de Maude para la comprobación de la propiedad Church-Rosser (Durán y Meseguer, 2000), pero la herramienta en cuestión no soporta de momento unificación módulo asociatividad y conmutatividad; en su lugar hemos adaptado la especificación para poder comprobarlo utilizando C*i*ME (Contejean et al., 2004).

También, por inspección de las ecuaciones resulta claro que ningún estado que satisfaga `prefix` es identificado con otro que no lo haga.

Sin embargo, la teoría de reescritura resultante *no es coherente*. Por un lado, nótese que la última ecuación en la abstracción es realmente un caso particular de la última regla de reescritura. Por ejemplo, el término

$$< \texttt{ls}(5, \texttt{a} : \texttt{b} : \texttt{c}), \texttt{msg}(3, \texttt{b}), \texttt{ls}(3, \texttt{a}) >$$

se puede reducir a

$$< \texttt{ls}(5, \texttt{a} : \texttt{b} : \texttt{c}), \texttt{null}, \texttt{ls}(3, \texttt{a} : \texttt{b}) >$$

aplicando tanto la ecuación como la regla, pero este término, a su vez, no puede ser reescrito por ninguna regla a un término al que se pueda demostrar que es igual ecuacionalmente, como debería ser el caso para tener la deseada coherencia. Afortunadamente, para resolver el problema basta con añadir la siguiente regla perezosa:

```
rl < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > =>
   < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .
```

Por otro lado, la segunda ecuación también puede dar lugar a un problema de coherencia. Por ejemplo:

$$\begin{array}{ccc} < \texttt{ls}(5,\texttt{L1}), \texttt{null}, \texttt{ls}(5,\texttt{L2}) > & \rightarrow & < \texttt{ls}(6,\texttt{L1} : \texttt{a}), \texttt{msg}(5,\texttt{a}), \texttt{ls}(5,\texttt{L2}) > \\ \| & & \| \; ? \\ < \texttt{ls}(\texttt{0},\texttt{L1}), \texttt{null}, \texttt{ls}(\texttt{0},\texttt{L2}) > & \rightarrow & < \texttt{ls}(1,\texttt{L1} : \texttt{a}), \texttt{msg}(\texttt{0},\texttt{a}), \texttt{ls}(\texttt{0},\texttt{L2}) > \end{array}$$

Supongamos ahora que `L1` y `L2` fueran iguales: en tal caso los dos términos en el lado derecho pueden ser reducidos aplicando las ecuaciones al término

$$< \texttt{ls}(\texttt{0}, \texttt{nil}), \texttt{null}, \texttt{ls}(\texttt{0}, \texttt{nil}) >$$

con lo que logramos coherencia. Aunque dicha igualdad no se tiene en general, se puede forzar exigiendo que tanto `L1` como `L2` sean `nil`, y por lo tanto iguales, para que la ecuación se pueda aplicar:

```
  ceq < ls(N, nil), null, ls(N, nil) > = < ls(0, nil), null, ls(0, nil) >
      if N =/= 0 .
```

La abstracción resultante viene entonces dada por:

```
mod ABSTRACT-PROTOCOL-CHECK is
  inc PROTOCOL-CHECK .

  vars M N : Nat .
  vars L1 L2 : ValueList .
  var MS : MessageSoup .
  var X : Value .

  eq < ls(N, X : L1), MS, ls(M, X : L2) > = < ls(N, L1), MS, ls(M, L2) > .
  ceq < ls(N, nil), null, ls(N, nil) > = < ls(0, nil), null, ls(0, nil) >
      if N =/= 0 .
  eq < ls(M, L1 : X : L2), msg(N, X) ; MS, ls(N, L1) > =
     < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .

  --- coherencia
  rl < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > =>
     < ls(M, L1 : X : L2), MS, ls(s(N), L1 : X) > .
endm
```

Y ahora la propiedad que deseamos puede ser finalmente comprobada:

```
Maude> red modelCheck(initial, [] prefix) .
result Bool: true
```

La siguiente observación sobre la abstracción anterior es digna de mención: ¡la razón por la que alcanzamos la coherencia es porque la abstracción colapsa casi todo! En particular, cada estado *alcanzable* es simplificado por las ecuaciones de la abstracción en el término

$$< \texttt{ls(0,nil)}, \texttt{null}, \texttt{ls(0,nil)} > .$$

4.6.3 El protocolo ABP

Pasamos ahora a considerar la corrección del protocolo del bit alternante o ABP (del inglés Alternating Bit Protocol), que apareció por primera vez descrito en Bartlett et al. (1969). El objetivo del protocolo ABP es asegurarse de que los mensajes enviados desde un emisor a un receptor en un canal defectuoso en el que se pueden perder o duplicar mensajes se reciben en el orden correcto. Nuestra presentación está adaptada de la de Müller y Nipkow (1995).

El estado del sistema en un momento dado viene dado por la 9-tupla

$$< \texttt{NEXT}, \texttt{MS}, \texttt{HS}, \texttt{MR}, \texttt{HR}, \texttt{QS}, \texttt{QR}, \texttt{LMR}, \texttt{LMS} >,$$

donde:

- `MS` y `MR` son los mensajes con los que el emisor y el receptor, respectivamente, se encuentran actualmente trabajando;
- `HS` y `HR` son variables booleanas utilizadas por el emisor y el receptor con própositos de sincronización;
- `QS` es el canal utilizado por el emisor para enviar los mensajes (junto con el bit alternante);
- `QR` es el canal utilizado por el receptor para confirmar la recepción de un mensaje;
- la variable booleana `NEXT` modela la asunción de que el entorno solo enviará un mensaje si se le pide que lo haga (véase Müller y Nipkow, 1995);
- `LMR` y `LMS` guardan, respectivamente, las listas de mensajes que ya han sido enviados y recibidos, y se introducen solo para poder llevar a cabo la verificación.

La infinitud se filtra en el protocolo de dos formas distintas: por un lado, el alfabeto de mensajes puede ser infinito; por otro, los canales no están acotados y pueden contener un número arbitrario de mensajes. Tal y como se hace en Müller y Nipkow (1995), gracias a los resultados sobre independencia de datos utilizados en Aggarwal et al. (1990) podemos descartar la primera fuente de infinitud y asumir un alfabeto de mensajes finito (de hecho, con solo tres elementos).

La especificación en Maude de los tipos de datos que intervienen en el protocolo es la siguiente.

```
fmod QBOOL is
  sort QBool .
  subsort Bool < QBool .

  op nil : -> QBool [ctor] .
  op _;_ : QBool QBool -> QBool [ctor assoc id: nil] .
endfm

fmod MESSAGE is
  sorts Message NonEMessage LMessage .
  subsort NonEMessage < Message .
  subsort NonEMessage < LMessage .

  op none : -> Message [ctor] .
  ops a b c : -> NonEMessage [ctor] .
  op nil : -> LMessage [ctor] .
  op _;_ : LMessage LMessage -> LMessage [ctor assoc id: nil] .
endfm
```

El tipo `Message` contiene mensajes de dos clases: genuinos (`a`, `b` y `c`) y una constante `none` que se utiliza para avisar de que no hay ningún mensaje disponible. En el modulo siguiente se emparejan con un booleano (el bit alternante), para conformar los paquetes (elementos de `QPair`) que el emisor envía por el canal `QS`.

```
fmod QPAIR is
  protecting MESSAGE .
  sorts Pair QPair .
  subsort Pair < QPair .

  op [_,_] : Bool Message -> Pair [ctor] .

  op nil : -> QPair [ctor] .
  op _;_ : QPair QPair -> QPair [ctor assoc id: nil] .
endfm

mod ABP is
  protecting MESSAGE .
  protecting QBOOL .
  protecting QPAIR .

  sort State .

  op <_,_,_,_,_,_,_,_,_> : Bool Message Bool Message Bool QPair
                           QBool LMessage LMessage -> State [ctor] .
```

El comportamiento del sistema se especifica mediante doce reglas de reescritura. Por ejemplo, si el emisor actualmente no dispone de mensajes que enviar (su buffer se encuentra vacío), la variable booleana `NEXT` se hace igual a `true` para permitir al entorno que introduzca un nuevo mensaje.

```
  rl [3] : < NEXT, none, HS, MR, HR, QS, QR, LMS, LMR > =>
           < true, none, HS, MR, HR, QS, QR, LMS, LMR > .
```

Entonces el emisor puede elegir entre tomar una `a`, una `b` o una `c`.

```
  rl [0] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < false, a, HS, MR, HR, QS, QR, LMS, LMR > .
  rl [1] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < false, b, HS, MR, HR, QS, QR, LMS, LMR > .
  rl [2] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < false, c, HS, MR, HR, QS, QR, LMS, LMR > .
```

El conjunto completo de reglas es el siguiente (nótese que la regla `[4]` permite la duplicación de mensajes y la regla `[7]` su eliminación).

```
  rl [0] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < false, a, HS, MR, HR, QS, QR, LMS, LMR > .
  rl [1] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < false, b, HS, MR, HR, QS, QR, LMS, LMR > .
  rl [2] : < true, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < false, c, HS, MR, HR, QS, QR, LMS, LMR > .
  rl [3] : < NEXT, none, HS, MR, HR, QS, QR, LMS, LMR > =>
           < true, none, HS, MR, HR, QS, QR, LMS, LMR > .
```

```
  rl [4] : < NEXT, M, HS, MR, HR, QS, QR, LMS, LMR > =>
           < NEXT, M, HS, MR, HR, QS ; [HS, M], QR, LMS, LMR > .
  crl [5] : < NEXT, MS, HS, none, HR, [B, M] ; QS, QR, LMS, LMR > =>
            < NEXT, MS, HS, M, not(HR), [B, M] ; QS, QR, LMS, LMR >
            if (HR =/= B) .
  crl [6] : < NEXT, MS, HS, none, HR, [B, M] ; QS, QR, LMS, LMR > =>
            < NEXT, MS, HS, M, not(HR), QS, QR, LMS, LMR >
            if (HR =/= B) .
  crl [7] : < NEXT, MS, HS, MR, HR, [B, M] ; QS, QR, LMS, LMR > =>
            < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR >
            if (HR == B) or (MS =/= none) .
  rl [8] : < NEXT, MS, HS, M, HR, QS, QR, LMS, LMR > =>
           < NEXT, MS, HS, none, HR, QS, QR, LMS, LMR ; M > .
  rl [9] : < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR > =>
           < NEXT, MS, HS, MR, HR, QS, QR ; HR, LMS, LMR > .
  crl [10] : < NEXT, MS, HS, MR, HR, QS, B ; QR, LMS, LMR > =>
             < NEXT, none, not(HS), MR, HR, QS, B ; QR, LMS ; MS, LMR >
             if (HS == B) .
  crl [11] : < NEXT, MS, HS, MR, HR, QS, B ; QR, LMS, LMR > =>
             < NEXT, none, not(HS), MR, HR, QS, QR, MS ; LMS, LMR >
             if (HS == B) .
  crl [12] : < NEXT, MS, HS, MR, HR, QS, B ; QR, LMS, LMR > =>
             < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR >
             if (HS =/= B) .
endm
```

La propiedad en la que estamos interesados es la de que los mensajes se entregan en el orden correcto. Para demostrarla, vamos a comprobar que en todo momento o bien la lista de los mensajes enviados es un prefijo de la lista de los mensajes recibidos, o viceversa. Obsérvese que no es cierto siempre que la lista de los mensajes recibidos sea un prefijo de la lista de los mensajes enviados, ya que los mensajes no son incluídos en esta segunda lista hasta que el emisor ha recibido la confirmación. Lo contrario tampoco es cierto en general, ya que el receptor puede enviar la confirmación antes de meter el correspondiente mensaje. En el comprobador de modelos de Maude, la propiedad se expresa mediante el término `[] pref`, donde el predicado de estado `pref` se especifica de la siguiente manera:

```
mod CHECK is
  inc ABP .

  op init : -> State .
  op pref : -> Prop [ctor] .

  vars NEXT HS HR : Bool .
  vars MS MR : Message .
  vars LM LMR LMS : LMessage .
  var QS : QPair .
  var QR : QBool .

  eq init = < false, none, true, none, false, nil, nil, nil, nil > .
```

```
  eq (< NEXT, MS, HS, MR, HR, QS, QR, LMS ; LM, LMS > |= pref) = true .
  eq (< NEXT, MS, HS, MR, HR, QS, QR, LMR, LMR ; LM > |= pref) = true .
endm
```

Esto completa la especificación del protocolo; sin embargo, como el conjunto de estados alcanzables es infinito, el comprobador de modelos no se puede utilizar.

La forma de resolver el problema consiste en simplificar el sistema utilizando una abstracción que aparece en Müller y Nipkow (1995). La idea es la siguiente: aunque los canales no están acotados, su contenido, como se puede observar inspeccionando las trazas de unas cuantas ejecuciones que comiencen en `init`, son siempre de la forma $m_1^* m_2^*$, con $m_1, m_2 \in \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. Entonces podemos obtener un sistema abstracto yuxtaponiendo mensajes adyacentes iguales. Nótese que no necesitamos demostrar que los contenidos de los canales sean realmente de esta forma: esto simplemente se usa como intuición para obtener un sistema más simple. Además también tenemos que abstraer las listas de mensajes que ya han sido enviados y recibidos y lo haremos eliminando de las cabeceras de las listas los mensajes comunes. Obsérvese que esta abstracción no produce un sistema con un número finito de estados; en principio, es todavía posible tener cadenas infinitas de mensajes de la forma $m_1 m_2 m_1 m_2 \ldots$ que no pueden ser reducidas (abstraídas) más. Sin embargo, va a resultar que el conjunto de estados alcanzables es finito y eso es todo lo que el comprobador de modelos necesita.

La identificación impuesta por la abstracción puede definirse mediante las siguientes ecuaciones:

```
eq < NEXT, MS, HS, MR, HR, QS, QR ; B ; B ; QR', LMS, LMR > =
   < NEXT, MS, HS, MR, HR, QS, QR ; B ; QR', LMS, LMR > .
eq < NEXT, MS, HS, MR, HR, QS ; [B, M] ; [B, M] ; QS', QR, LMS, LMR > =
   < NEXT, MS, HS, MR, HR, QS ; [B, M] ; QS', QR, LMS, LMR  > .
eq < NEXT, MS, HS, MR, HR, QS, QR, M ; LMS, M ; LMR > =
   < NEXT, MS, HS, MR, HR, QS, QR, LMS, LMR > .
```

Las dos primeras ecuaciones juntan mensajes adyacentes iguales en los canales, mientras que la última corresponde a la eliminación de aquellos mensajes que ya han sido enviados y recibidos.

Es inmediato darse cuenta de que si dos estados son identificados por las ecuaciones anteriores, entonces o bien ambos o ninguno satisfacen `pref`. En definitiva, todas las obligaciones de prueba se satisfacen (en el ejemplo de la sección siguiente se dan las demostraciones detalladas para un caso similar) con lo que podemos ejecutar el comprobador de modelos de Maude para comprobar que la propiedad `[] pref` se cumple.

```
Maude> red init |= [] pref .
Result Bool: true
```

4.6.4 El protocolo BRP

En esta última sección vamos a discutir en detalle un ejemplo algo más complejo, el protocolo de retransmisión acotada (Havelund y Shankar, 1996; D'Argenio et al., 1997),

al que nos referiremos por sus siglas en inglés: BRP (Bounded Retransmission Protocol). El protocolo BRP es una extensión del protocolo del bit alternante en el que se establece un límite en el número de veces que se pueden transmitir los mensajes; la siguiente descripción está tomada de Abdulla et al. (1999).

En el lado del emisor el protocolo requiere una secuencia de datos $s = d_1, \ldots, d_n$ (acción REQ) y devuelve una señal de confirmación que puede ser SOK, SNOK o SDNK. La confirmación SOK significa que el fichero ha sido transmitido con éxito, SNOK significa que el fichero no ha sido transmitido completamente y SDNK significa que el fichero puede que no se haya transmitido completamente. Esta última señal se produce cuando el último dato d_n es enviado pero no se recibe confirmación de que ha sido recibido.

Del lado del receptor el protocolo entrega cada dato que ha sido recibido correctamente junto con una indicación que puede ser RFST, RINC, ROK o RNOK. La indicación RFST significa que el dato entregado es el primero y que le seguirán otros, RINC significa que el dato es uno de los intermedios y ROK que este era el último dato y que el fichero ha sido completado. Sin embargo, cuando se pierde la conexión con el servidor la indicación que se presenta es RNOK (sin ningún dato).

En Maude, los distintos estados locales del emisor y del receptor, los mensajes, las secuencias de mensajes y las etiquetas de las transiciones se pueden representar de la manera siguiente:

```
fmod DATA is
  sorts Sender Receiver .
  sort Label .
  sorts Msg MsgL .
  subsort Msg < MsgL .

  ops 0s 1s 2s 3s 4s 5s 6s 7s : -> Sender .
  ops 0r 1r 2r 3r 4r : -> Receiver .

  ops none req snok sok sdnk rfst rnok rinc rok : -> Label .

  ops 0 1 fst last : -> Msg .
  op nil : -> MsgL .
  op _;_ : MsgL MsgL -> MsgL [assoc id: nil] .
endfm
```

Entre las propiedades que el protocolo debería satisfacer se encuentran las siguientes:

1. Una petición REQ debe ir seguida de una señal de confirmación (SOK, SNOK o SDNK) antes de la siguiente petición.
2. Una indicación RFST debe ser seguida de una de las indicaciones ROK o RNOK antes del comienzo de una nueva transmisión (nueva petición a un emisor).
3. Una señal de confirmación SOK debe ir precedida de una señal ROK.
4. Una indicación RNOK debe ir precedida de una señal de confirmación SNOK o SDNK (cancelación de la transmisión).

El protocolo BRP es modelado en Abdulla et al. (1999), tras algunas simplificaciones para eliminar todas las referencias al *tiempo* del sistema, como un sistema con un canal de comunicación con fallos. Nuestra especificación en Maude está adaptada de la suya. Los estados del sistema se representan mediante un tipo `State` que se construye con el operador 7-ario <_,_,_,_,_,_,_>. Las componentes primera y quinta describen la situación actual del emisor y del receptor, respectivamente. La segunda y la sexta son valores booleanos utilizados por el emisor y por el receptor por motivos de sincronización. Las componentes tercera y cuarta de la tupla se corresponden con los dos canales con fallos a través de los cuales el emisor y el receptor se comunican. La última componente lleva el registro del nombre de la última transición utilizada para alcanzar el estado actual (de ahí los nombres de las constantes del tipo `Label`: `req`, `snok`, `sok`, ...). Solo hacemos explícito el nombre de estas transiciones para los casos en los que estamos interesados (es decir, aquellos requeridos por las propiedades); en los demás usaremos el valor `none`.

Para una descripción más detallada del protocolo nos referimos al trabajo de Abdulla et al. (1999). En Maude, el protocolo queda como sigue:

```
mod BRP is
  protecting DATA .
  inc MODEL-CHECKER .

  op <_,_,_,_,_,_,_> : Sender Bool MsgL MsgL Receiver Bool Label -> State .
  op initial : -> State .

  var S : Sender .
  var R : Receiver .
  var M : Msg .
  vars K L KL : MsgL .
  vars A RT : Bool .
  var LA : Label .

  eq initial = < 0s, false, nil, nil, 0r, false, none > .

  rl [REQ] : < 0s, A, nil, nil, R, false, LA > =>
             < 1s, false, nil, nil, R, false, req > .
  rl [K!fst] : < 1s, A, K, L, R, RT, LA > =>
               < 2s, A, K ; fst, L, R, RT, none > .
  rl [K!fst] : < 2s, A, K, L, R, RT, LA > =>
               < 2s, A, K ; fst, L, R, RT, none > .
  rl [L?fst] : < 2s, A, K, fst ; L, R, RT, LA > =>
               < 3s, A, K, L, R, RT, none > .
  crl [L?-fst] : < 2s, A, K, M ; L, R, RT, LA > =>
                 < 2s, A, K, L, R, RT, none > if M =/= fst .
  rl [K!0] : < 3s, A, K, L, R, RT, LA > =>
             < 4s, A, K ; 0, L, R, RT, none > .
  rl [K!last] : < 3s, A, K, L, R, RT, LA > =>
                < 7s, A, K ; last, L, R, RT, none > .
  rl [K!0] : < 4s, A, K, L, R, RT, LA > =>
             < 4s, A, K ; 0, L, R, RT, none > .
  crl [L?-0] : < 4s, A, K, M ; L, R, RT, LA > =>
```

```
             < 4s, A, K, L, R, RT, none > if M =/= 0 .
rl [L?0] : < 4s, A, K, 0 ; L, R, RT, LA > =>
           < 5s, A, K, L, R, RT, none > .
rl [SNOK] : < 4s, A, K, nil, R, RT, LA > =>
            < 0s, true, K, nil, R, RT, snok > .
rl [K!1] : < 5s, A, K, L, R, RT, LA > =>
           < 6s, A, K ; 1, L, R, RT, none > .
rl [K!last] : < 5s, A, K, L, R, RT, LA > =>
              < 7s, A, K ; last, L, R, RT, none > .
rl [K!1] : < 6s, A, K, L, R, RT, LA > =>
           < 6s, A, K ; 1, L, R, RT, none > .
crl [L?-1] : < 6s, A, K, M ; L, R, RT, LA > =>
             < 6s, A, K, L, R, RT, none > if M =/= 1 .
rl [SNOK] : < 6s, A, K, nil, R, RT, LA > =>
            < 0s, true, K, nil, R, RT, snok > .
rl [K!last] : < 7s, A, K, L, R, RT, LA > =>
              < 7s, A, K ; last, L, R, RT, none > .
crl [L?-last] : < 7s, A, K, M ; L, R, RT, LA > =>
                < 7s, A, K, L, R, RT, none > if M =/= last .
rl [SOK] : < 7s, A, K, last ; L, R, RT, LA > =>
           < 0s, A, K, L, R, RT, sok > .
rl [SDNK] : < 7s, A, K, nil, R, RT, LA > =>
            < 0s, true, K, nil, R, RT, sdnk > .

rl [RFST] : < S, false, fst ; K, L, 0r, RT, LA > =>
            < S, false, K, L ; fst, 1r, true, rfst > .
rl [K?fstL!fst] : < S, A, fst ; K, L, 1r, RT, LA > =>
                  < S, A, K, L ; fst, 1r, RT, none > .
rl [RNOK] : < S, true, nil, L, 1r, RT, LA > =>
            < S, true, nil, L, 1r, false, rnok > .
rl [RINC] : < S, false, 0 ; K, L, 1r, RT, LA > =>
            < S, false, K, L ; 0, 2r, RT, rinc > .
rl [ROK] : < S, false, last ; K, L, 1r, RT, LA > =>
           < S, false, K, L ; last, 4r, RT, rok > .
rl [K?0L!0] : < S, A, 0 ; K, L, 2r, RT, LA > =>
              < S, A, K, L ; 0, 2r, RT, none > .
rl [RINC] : < S, false, 1 ; K, L, 2r, RT, LA > =>
            < S, false, K, L ; 1, 3r, true, rinc > .
rl [RNOK] : < S, true, nil, L, 2r, RT, LA > =>
            < S, true, nil, L, 0r, false, rnok > .
rl [ROK] : < S, false, last ; K, L, 2r, RT, LA > =>
           < S, false, K, L ; last, 4r, RT, rok > .
rl [RINC] : < S, false, 0 ; K, L, 3r, RT, LA > =>
            < S, false, K, L ; 0, 2r, RT, rinc > .
rl [K?1L!1] : < S, A, 1 ; K, L, 3r, RT, LA > =>
              < S, A, K, L ; 1, 3r, RT, none > .
rl [ROK] : < S, false, last ; K, L, 3r, RT, LA > =>
           < S, false, K, L ; last, 4r, RT, rok > .
rl [RNOK] : < S, true, nil, L, 3r, RT, LA > =>
            < S, true, nil, L, 0r, false, rnok > .
rl [K?lastL!last] : < S, A, last ; K, L, 4r, RT, LA > =>
```

```
                    < S, A, K, L ; last, 4r, RT, none > .
  rl [empty] : < S, A, last ; K, L, 4r, RT, LA > =>
               < S, A, nil, L, 0r, false, none > .
endm
```

Las propiedades que el sistema debería satisfacer imponen requisitos que suelen exigir que ciertas transiciones ocurran antes que otras. Para formular requisitos de esta forma general vamos a declarar una proposición atómica *paramétrica*, `tr(`LA`)`, que será cierta en aquellos estados que resulten de la aplicación de una transición etiquetada con LA.

```
mod BRP-CHECK is
  inc BRP .

  op tr : Label -> Prop .

  var S : Sender .      var R : Receiver .
  var M : Msg .         vars K L : MsgL .
  vars A RT : Bool .    vars LA : Label .

  eq (< S, A, K, L, R, RT, LA > |= tr(LA)) = true .
endm
```

Ahora las cuatro propiedades pedidas se pueden expresar en la lógica temporal de Maude como sigue:

1. `[](tr(req) -> O (~ tr(req) W (tr(sok) \/ tr(snok) \/ tr(sdnk))));`

2. `[](tr(rfst) -> (~ tr(req) W (tr(rok) \/ tr(rnok))));`

3. `[](tr(req) -> (~ tr(sok) W tr(rok)));`

4. `[](tr(req) -> (~ tr(rnok) W (tr(snok) \/ tr(sdnk)))).`

Nótese que en estas fórmulas aparecen tanto negaciones como implicaciones. Por lo tanto, para que podamos aplicar el teorema 4.1 debemos asegurarnos de que la abstracción que se defina sea estricta, esto es, que preserve las proposiciones atómicas.

El sistema es infinito pero uno se da cuenta en seguida de que, como ya ocurría en el protocolo ABP, el contenido de los canales siempre se ajusta a la expresión regular $m_1^* m_2^*$, donde m_1 y m_2 toman valores sobre {`first`, `last`, `0`, `1`}. Por lo tanto, para colapsar el conjunto de estados a un número finito podemos utilizar la idea de mezclar mensajes adyacentes que sean iguales, lo que se puede especificar por medio de las dos siguientes ecuaciones:

```
  eq < S, A, KL ; M ; M ; K, L, R, RT, LA > =
     < S, A, KL ; M ; K, L, R, RT, LA > .
  eq < S, A, K, KL ; M ; M ; L, R, RT, LA > =
     < S, A, K, KL ; M ; L, R, RT, LA > .
```

Nótese que, de nuevo, no necesitamos probar que los contenidos de los canales de comunicación son realmente de la forma $m_1^* m_2^*$. Esto solo se usa como intuición para guiarnos a la hora de elegir las ecuaciones que definan la abstracción y puede usarse con independencia de la validez de la afirmación (aunque probablemente no sería muy útil si la afirmación no fuese cierta).

Se comprueba inmediatamente, debido a que las ecuaciones de la abstracción no afectan a la etiqueta del estado, que tan solo se identifican estados que satisfacen las mismas proposiciones atómicas. Por lo tanto, los requisitos del teorema 4.1 se cumplen. Y puesto que las ecuaciones se aplican a componentes disjuntas del estado y solo hay un número finito de mensajes que puedan ser eliminados, también tenemos las propiedades de Church-Rosser y terminación.

¿Qué ocurre con la dificultad del bloqueo? Mediante la inspección de los lados izquierdos de las reglas en BRP se ve fácilmente que la ecuación

```
enabled(< S, A, KL ; M ; K, L, R, RT, LA >) = true
```

no se cumple (piénsese en el caso en el que S es igual a 0s) para el operador enabled definido en la sección 4.5, por lo que la regla

```
rl < S, A, KL ; M ; K, L, R, RT, LA > => < S, A, KL ; M ; K, L, R, RT, LA > .
```

debería ser añadida; algo parecido ocurre con la segunda ecuación que define la abstracción. Nótese que sin embargo podemos hacer algo aún mejor. Por inspección directa de las reglas es fácil comprobar que, con la excepción del caso en el que S es igual a 0s, en realidad todos los términos de la forma < S, A, KL ; M ; K, L, R, RT, LA > están habilitados (existe alguna regla que se les puede aplicar). Así, en lugar de la anterior podemos añadir simplemente las reglas

```
rl [deadlock] : < 0s, A, KL ; M ; K, L, R, RT, LA > =>
                < 0s, A, KL ; M ; K , L, R, RT, LA > .
rl [deadlock] : < 0s, A, K, KL ; M ; L, R, RT, LA > =>
                < 0s, A, K, KL ; M ; L, R, RT, LA > .
```

Finalmente, la última de las obligaciones de prueba por comprobar es la de la coherencia, que también falla. Considérese por ejemplo el término

```
< 2s, true, nil, fst ; fst, 0r, true, none >
```

Este término se puede reescribir utilizando la primera de las reglas etiquetadas con [L?fst] dando lugar a otro término t de la forma

```
< 3s, true, nil, fst, 0r, true, none >
```

Sin embargo, si lo hubiéramos reducido primero utilizando las ecuaciones habríamos obtenido

```
< 2s, true, nil, fst, 0r, true, none >
```

que ya no puede ser reescrito a t ni a ningún otro término ecuacionalmente igual (un mensaje `fst` adicional ha sido consumido siguiendo este camino). Para solucionar este problema se tiene que añadir la siguiente regla:

```
rl [L?fst'] : < 2s, A, K, fst ; L, R, RT, LA > =>
              < 3s, A, K, fst ; L, R, RT, none > .
```

Con ella queda resuelto el problema de coherencia.

La misma situación se plantea con todas aquellas reglas en las que un mensaje es eliminado de una de las listas; la solución en todos los casos es la misma, lo que resulta en la adición de *todas* las siguientes reglas:

```
crl [L?-fst'] : < 2s, A, K, M ; L, R, RT, LA > =>
                < 2s, A, K, M ; L, R, RT, none > if M =/= fst .
crl [L?-0'] : < 4s, A, K, M ; L, R, RT, LA > =>
              < 4s, A, K, M ; L, R, RT, none > if M =/= 0 .
rl [L?0'] : < 4s, A, K, 0 ; L, R, RT, LA > =>
            < 5s, A, K, 0 ; L, R, RT, none > .
crl [L?-1] : < 6s, A, K, M ; L, R, RT, LA > => < 6s, A, K, M ; L, R, RT, none >
             if M =/= 1 .
crl [L?-last] : < 7s, A, K, M ; L, R, RT, LA > =>
                < 7s, A, K, M ; L, R, RT, none >
                if M =/= last .
rl [SOK] : < 7s, A, K, last ; L, R, RT, LA > =>
           < 0s, A, K, last ; L, R, RT, sok > .
rl [RFST'] : < S, false, fst ; K, L, 0r, RT, LA > =>
             < S, false, fst ; K, L ; fst, 1r, true, rfst > .
rl [K?fstL!fst'] : < S, A, fst ; K, L, 1r, RT, LA > =>
                   < S, A, fst ; K, L ; fst, 1r, RT, none > .
rl [RINC'] : < S, false, 0 ; K, L, 1r, RT, LA > =>
             < S, false, 0 ; K, L ; 0, 2r, RT, rinc > .
rl [ROK'] : < S, false, last ; K, L, 1r, RT, LA > =>
            < S, false, last ; K, L ; last, 4r, RT, rok > .
rl [K?0L!0'] : < S, A, 0 ; K, L, 2r, RT, LA > =>
               < S, A, 0 ; K, L ; 0, 2r, RT, none > .
rl [RINC'] : < S, false, 1 ; K, L, 2r, RT, LA > =>
             < S, false, 1 ; K, L ; 1, 3r, true, rinc > .
rl [ROK'] : < S, false, last ; K, L, 2r, RT, LA > =>
            < S, false, last ; K, L ; last, 4r, RT, rok > .
rl [RINC'] : < S, false, 0 ; K, L, 3r, RT, LA > =>
             < S, false, 0 ; K, L ; 0, 2r, RT, rinc > .
rl [K?1L!1'] : < S, A, 1 ; K, L, 3r, RT, LA > =>
               < S, A, 1 ; K, L ; 1, 3r, RT, none > .
rl [ROK'] : < S, false, last ; K, L, 3r, RT, LA > =>
            < S, false, last ; K, L ; last, 4r, RT, rok > .
rl [K?lastL!last'] : < S, A, last ; K, L, 4r, RT, LA > =>
                     < S, A, last ; K, L ; last, 4r, RT, none > .
```

El deseado módulo `ABSTRACT-BRP-CHECK` con la abstracción ejecutable se obtiene ahora importando `BRP-CHECK` e incluyendo las ecuaciones que definen la abstracción, las dos reglas `[deadlock]` y las reglas que acabamos de enumerar más arriba.

Podemos entonces utilizar el comprobador de modelos de Maude sobre el sistema especificado por ABSTRACT-BRP-CHECK para verificar que todas las propiedades se cumplen en él. Como todas las obligaciones de prueba se cumplen, podemos inferir correctamente que las propiedades también se cumplen en el sistema original.

```
Maude> red modelCheck(initial,
                      [](tr(req) ->
                         O (~ tr(req) W (tr(sok) \/ tr(snok) \/ tr(sdnk))))) .
reduce in ABSTRACT-BRP-CHECK : modelCheck(initial, [](tr(req) ->
    O (~ tr(req) W tr(sdnk) \/ (tr(snok) \/ tr(sok))))) .
result Bool: true

Maude> red modelCheck(initial,
                      [](tr(rfst) -> (~ tr(req) W (tr(rok) \/ tr(rnok))))) .
reduce in ABSTRACT-BRP-CHECK : modelCheck(initial, [](tr(rfst) ->
    ~ tr(req) W tr(rnok) \/ tr(rok))) .
result Bool: true

Maude> red modelCheck(initial, [](tr(req) -> (~ tr(sok) W tr(rok)))) .
reduce in ABSTRACT-BRP-CHECK : modelCheck(initial, [](tr(req) ->
    ~ tr(sok) W tr(rok))) .
result Bool: true

Maude> red modelCheck(initial,tr(req) -> (~ tr(rnok) W (tr(snok) \/ tr(sdnk)))) .
reduce in ABSTRACT-BRP-CHECK : modelCheck(initial, tr(req) ->
    ~ tr(rnok) W tr(snok) \/ tr(sdnk)) .
result Bool: true
```

4.7 Conclusiones

En Clarke et al. (1994) se definió la simulación de un sistema $\mathcal{M}$ por medio de otro $\mathcal{M}'$ a través de una función sobreyectiva y se identificó el sistema simulador óptimo $\mathcal{M}^h_{\text{min}}$. La idea de simular por medio de un cociente ha sido también explorada en los trabajos de Clarke et al. (1999, 2000), de Bensalem et al. (1998), Kesten y Pnueli (2000b), Manolios (2001) y Dams et al. (1997) entre otros, aunque la construcción en Dams et al. (1997) requiere el uso de una conexión de Galois en lugar de una simple función. Bensalem et al. (1998) proponen el uso de demostradores de teoremas para construir la relación de transición del sistema abstracto y también lo hace Manolios (2001), quien los utiliza para probar que una función es *representativa*, lo que implica que puede ser usada como entrada de un algoritmo que extrae $\mathcal{M}^h_{\text{min}}$ a partir de $\mathcal{M}$. Estos usos de los demostradores de teoremas se concentran en la corrección de la relación de transición abstracta, mientras que nuestro método se centra en lograr que la relación de transición minimal (que es correcta por construcción) sea *computable* y en demostrar la preservación de la función de etiquetado. En Clarke et al. (1994, 1999), por otro lado, el modelo minimal $\mathcal{M}^h_{\text{min}}$ se descarta en favor de aproximaciones menos precisas pero más fáciles de computar; bajo nuestra óptica esto correspondería a añadir reglas de reescritura a la especificación para simplificar las demostraciones de las obligaciones de prueba (lo que ciertamente puede

ser una alternativa razonable a la hora de aplicar las técnicas de este capítulo con una metodología más "ligera"). En todos los trabajos mencionados, dos estados se identifican si y solo si satisfacen las mismas proposiciones atómicas; nuestra definición de simulación es más general.

Capítulo 5

Simulaciones algebraicas

Tras estudiar la idea de abstracción de sistemas y presentar una técnica para realizarla en la lógica de reescritura, pasamos ahora a estudiar la noción de *simulación* en sentido más amplio, como un medio para relacionar dos sistemas arbitrarios, sin que necesariamente uno sea finito o más simple que el otro. Nos interesa así una definición de simulación lo más general posible, por lo que vamos a extender la del capítulo 4 en dos direcciones de forma que se puedan relacionar sistemas sobre proposiciones atómicas distintas y no sea necesario preservar las transiciones estrictamente, sino solo *salvo tartamudeo*. Todas nuestras construcciones tendrán su contrapartida al nivel de la lógica de reescritura, que veremos se puede utilizar para especificar cualquiera de ellas mediante unos resultados generales de representabilidad.

Como ya se hizo en el capítulo anterior, a medida que vayamos introduciendo nuestras diferentes nociones de simulación las iremos organizando en categorías junto con las correspondientes estructuras de Kripke. Aunque se darán ya algunos resultados sobre ellas, un estudio categórico en sí mismo no se llevará a cabo hasta el capítulo 6.

5.1 Moviéndonos entre distintos niveles

Estamos interesados en una definición de simulación más general, una en la que estructuras de Kripke sobre diferentes conjuntos AP y AP' puedan ser relacionadas. Esto ofrecería una manera muy flexible de relacionar estructuras de Kripke y nos permitiría agrupar todas las categorías anteriores $\mathbf{KSim}_{AP}$ en una única. En primer lugar necesitamos la siguiente definición para trasladar las propiedades de una estructura de Kripke a un conjunto distinto de proposiciones atómicas.

Definición 5.1 *Dada una función* $\alpha : AP \longrightarrow \mathrm{State}(AP')$ *y una estructura de Kripke* $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ *sobre* AP'*, definimos la* estructura de Kripke reducto $\mathcal{A}|_\alpha = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}|_\alpha})$ *sobre* AP *como aquella cuya función de etiquetado viene dada por* $L_{\mathcal{A}|_\alpha}(a) = \{p \in AP \mid \mathcal{A}, a \models \alpha(p)\}$.

La definición de α se extiende homomórficamente a fórmulas $\varphi \in \mathrm{CTL}^*(AP)$ de la manera esperada, reemplazando cada proposición atómica p que aparece en φ por $\alpha(p)$; denotamos esta extensión mediante $\overline{\alpha}(\varphi)$.

La noción de reducto de una estructura de Kripke está inspirada en la de reducto de un álgebra; como para aquella, tenemos el siguiente resultado relacionando la satisfacción de fórmulas por estructuras de Kripke con sus correspondientes traducciones y reductos.

Proposición 5.1 *Sean $\alpha : AP \longrightarrow \mathrm{State}(AP')$ una función y φ una fórmula en $\mathrm{CTL}^*(AP)$. Entonces, para toda estructura de Kripke $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ sobre AP', estado $a \in A$ y camino π:*

- *si φ es una fórmula de estado, $\mathcal{A}, a \models \overline{\alpha}(\varphi) \iff \mathcal{A}|_\alpha, a \models \varphi$, y*
- *si φ es una fórmula de camino, $\mathcal{A}, \pi \models \overline{\alpha}(\varphi) \iff \mathcal{A}|_\alpha, \pi \models \varphi$.*

Demostración. Demostraremos ambas afirmaciones al tiempo mediante inducción simultánea sobre la estructura de las fórmulas.

El resultado se sigue por definición de $\mathcal{A}|_\alpha$ si p es una proposición atómica y es trivial en el caso de $\top$ y $\bot$. Para $\mathbf{A}\varphi$ se tiene que

$$\begin{array}{rcll} \mathcal{A}, a \models \overline{\alpha}(\mathbf{A}\varphi) & \iff & \mathcal{A}, \pi \models \overline{\alpha}(\varphi) & \text{para todo camino } \pi \text{ que comience en } a \\ & \iff & \mathcal{A}|_\alpha, \pi \models \varphi & \text{para todo camino } \pi \text{ que comience en } a \\ & \iff & \mathcal{A}|_\alpha, a \models \mathbf{A}\varphi & \end{array}$$

donde la primera equivalencia se cumple porque $\overline{\alpha}(\mathbf{A}\varphi) = \mathbf{A}\overline{\alpha}(\varphi)$ y la segunda gracias a la hipótesis de inducción. Para $\mathbf{F}\varphi$,

$$\begin{array}{rcl} \mathcal{A}, \pi \models \overline{\alpha}(\mathbf{F}\varphi) & \iff & \mathcal{A}, \pi^n \models \overline{\alpha}(\varphi) \text{ para algún } n \in \mathbb{N} \\ & \iff & \mathcal{A}|_\alpha, \pi^n \models \varphi \\ & \iff & \mathcal{A}|_\alpha, \pi \models \mathbf{F}\varphi \end{array}$$

donde la segunda equivalencia se sigue de la hipótesis de inducción. La demostración procede de manera análoga para el resto de los operadores temporales. □

Nótese que no tiene ningún sentido llevar una proposición atómica, que es una fórmula de estado, en una fórmula arbitraria de ACTL^* que podría ser una fórmula de camino. Por lo tanto, la elección de $\mathrm{State}(AP')$ como el rango de las funciones α es tan general como resulta posible.

Hay que señalar también que cuando trabajemos con simulaciones no estrictas, como las fórmulas reflejadas estarán en $\mathrm{ACTL}^* \backslash \neg(AP)$ querremos que nuestras funciones α tengan su rango en el fragmento libre de negación $\mathrm{State} \backslash \neg(AP')$, esto es, usaremos funciones $\alpha : AP \longrightarrow \mathrm{State} \backslash \neg(AP')$.

Ahora la definición generalizada de simulación resulta inmediata. Nótese que simplemente estamos elaborando los detalles de la construcción de Grothendieck (Tarlecki et al., 1991) sobre una categoría indexada particular (véase la sección 6.4 del capítulo 6).

Definición 5.2 *Dada una estructura de Kripke $\mathcal{A}$ sobre un conjunto AP de proposiciones atómicas y una estructura de Kripke $\mathcal{B}$ sobre un conjunto AP', una* simulación *(resp.* simulación estricta*)* $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ *consiste en una función $\alpha : AP \longrightarrow \mathrm{State} \backslash \neg(AP')$ (resp. $\alpha : AP \longrightarrow \mathrm{State}(AP')$) y una AP-simulación (resp. AP-simulación estricta) $H : \mathcal{A} \longrightarrow \mathcal{B}|_\alpha$. Decimos que (α, H) es un* morfismo *(resp.* morfismo estricto*) o una* bisimulación *si H lo es en la categoría $\mathbf{KSim}_{AP}$ (resp. $\mathbf{KSim}^{\mathrm{str}}_{AP}$).*

Estas simulaciones generalizadas también se pueden componer.

Proposición 5.2 *Si* $(\alpha, F) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ *y* $(\beta, G) : (AP', \mathcal{B}) \longrightarrow (AP'', C)$ *son simulaciones entonces* $(\beta, G) \circ (\alpha, F) = (\overline{\beta} \circ \alpha, G \circ F)$ *también es una simulación.*

Demostración. Tenemos que comprobar que $F : \mathcal{A} \longrightarrow C_{\overline{\beta}\circ\alpha}$ es una AP-simulación. Sean $a \in A$ y $c \in C$ tales que $a(G \circ F)c$; entonces existe un elemento $b \in \mathcal{B}$ tal que aFb y bGc.

Vamos a comprobar primero que $G \circ F$ es una simulación de los sistemas de transiciones subyacentes. Si $a \rightarrow_{\mathcal{A}} a'$, como $F : \mathcal{A} \longrightarrow \mathcal{B}_\alpha$ es una AP-simulación existe $b' \in B$ tal que $a'Fb'$ y $b \rightarrow_{\mathcal{B}} b'$; de manera análoga, existe $c' \in C$ tal que $b'Gc'$ y $c \rightarrow_C c'$. Tenemos así que $a'(G \circ F)c'$ con $c \rightarrow_C c'$ como se necesitaba.

Sea ahora $p \in L_{C|_{\overline{\beta}\circ\alpha}}(c)$:

$$\begin{array}{lcll} p \in L_{C|_{\overline{\beta}\circ\alpha}}(c) & \iff & C, c \models \overline{\beta}(\alpha(p)) & \text{(por definición)} \\ & \iff & C|_\beta, c \models \alpha(p) & \text{(por la proposición 5.1)} \\ & \implies & \mathcal{B}, b \models \alpha(p) & \text{(por el teorema 4.1)} \\ & \iff & \mathcal{B}|_\alpha, b \models p & \text{(por la proposición 5.1)} \\ & \implies & A, a \models p & (F \text{ es una } AP\text{-simulación}) \\ & \iff & p \in L_{\mathcal{A}}(a) & \end{array}$$

Esto es, $L_{C|_{\overline{\beta}\circ\alpha}}(c) \subseteq L_{\mathcal{A}}(a)$ y $G \circ F$ es una AP-simulación. □

Por lo tanto, usando pares $(AP, \mathcal{A})$, con AP un conjunto de proposiciones atómicas y $\mathcal{A}$ una estructura de Kripke sobre AP, como objetos y los morfismos adecuados en cada caso obtenemos las categorías **KSim**, **KMap** y **KBSim**. De nuevo, si (α, H) es un isomorfismo en **KSim** entonces H debe ser un morfismo y una bisimulación.

Hay que señalar que, por el contrario, las simulaciones estrictas *no* se pueden componer. En la demostración anterior, si F y G fueran estrictas la última implicación se convertiría en una equivalencia, pero no ocurriría lo mismo con la primera. Por ejemplo, consideremos las siguientes tres estructuras de Kripke sobre el mismo conjunto $AP = \{p\}$ de proposiciones atómicas, con $L_{\mathcal{A}}(a) = L_{\mathcal{B}}(b) = L_C(c) = \{p\}$ y $L_C(d) = \emptyset$:

Entonces, si definimos $\alpha(p) = \mathbf{AG}p$, $\beta(p) = p$, $f(a) = b$ y $g(b) = c$, es fácil comprobar que $(\alpha, f) : (AP, \mathcal{A}) \longrightarrow (AP, \mathcal{B})$ y $(\beta, g) : (AP, \mathcal{B}) \longrightarrow (AP, C)$ son simulaciones estrictas, pero $(\overline{\beta} \circ \alpha, g \circ f) : (AP, \mathcal{A}) \longrightarrow (AP, C)$ no lo es: $p \notin L_{C|_{\overline{\beta}\circ\alpha}}(c)$ porque $C, c \not\models \mathbf{AG}p$. Claramente, el motivo reside en el hecho de que α aplica una proposición atómica en una fórmula arbitraria. Las AP-simulaciones estrictas se pueden componer porque los elementos que relacionan satisfacen exactamente las mismas proposiciones; ahora que nos movemos entre distintos niveles y transformamos proposiciones atómicas en fórmulas generales, para que las simulaciones se pudieran componer sería necesario que elementos relacionados satisfacieran dichas fórmulas, lo cual en general no es cierto.

De esta manera, si estuviéramos interesados en tener simulaciones estrictas que se pudieran componer habría que restringir el rango de la función α. Bastaría con exigir que α solo llevase proposiciones atómicas a proposiciones atómicas, pero una ligera generalización resulta posible: es suficiente pedir que el rango de α sea de la forma Bool(AP), el subconjunto de las fórmulas de estado en ACTL*(AP) que no contienen el operador **A** (esto es, las expresiones booleanas sobre AP). Se tiene entonces que la composición de simulaciones estrictas es consecuencia de la siguiente especialización sencilla del teorema 4.1.

Proposición 5.3 *Las AP-simulaciones estrictas, además de reflejar, siempre preservan las fórmulas en* Bool(AP)*; es decir, si $H : \mathcal{A} \longrightarrow \mathcal{B}$ es una AP-simulación estricta, aHb y $\varphi \in$ Bool(AP),*

$$\mathcal{A}, a \models \varphi \quad \textit{si y solo si} \quad \mathcal{B}, b \models \varphi\,.$$

Esta es una situación que se repetirá de nuevo, aunque bajo una apariencia distinta, en las secciones 6.3 y 6.4.

Para simplificar la notación, de ahora en adelante escribiremos $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ en lugar de $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ excepto en aquellas situaciones en las que podría ser motivo de confusión.

Definición 5.3 *Dadas dos estructuras de Kripke $\mathcal{A}$ sobre AP y $\mathcal{B}$ sobre AP', una simulación $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$* refleja *la satisfacción de una fórmula $\varphi \in$ CTL*(AP) cuando:*

- *φ es una fórmula de estado y $\mathcal{B}, b \models \overline{\alpha}(\varphi)$ y aHb implica que $\mathcal{A}, a \models \varphi$; o bien*
- *φ es una fórmula de camino y $\mathcal{B}, \rho \models \overline{\alpha}(\varphi)$ y $\pi H \rho$ implica que $\mathcal{A}, \pi \models \varphi$.*

Los principales resultados que obtuvimos para estructuras de Kripke sobre un conjunto fijo de proposiciones atómicas se extienden de manera natural a simulaciones generalizadas.

Teorema 5.1 *Las simulaciones siempre reflejan la satisfacción de fórmulas en* ACTL$^*\backslash\neg$. *Además, las simulaciones estrictas también reflejan la satisfacción de fórmulas* ACTL*.

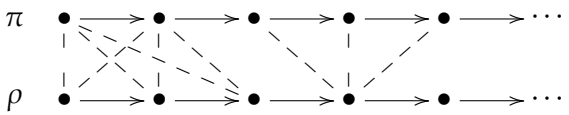
Demostración. El resultado es una consecuencia directa de la proposición 5.1 y del teorema 4.1. □

5.2 Simulaciones tartamudas

Otra dirección en la que la definición original de simulación se puede extender es la de las bisimulaciones tartamudas utilizadas en Browne et al. (1988) y Namjoshi (1997) o, con mayor generalidad, las simulaciones tartamudas de Manolios (2001).

Definición 5.4 *Sean $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ y $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ sistemas de transiciones y sea $H \subseteq A \times B$ una relación. Dado un camino π en $\mathcal{A}$ y un camino ρ en $\mathcal{B}$, decimos que ρ H-*encaja *(H-matches) con π si existen dos funciones estrictamente crecientes $\alpha, \beta : \mathbb{N} \longrightarrow \mathbb{N}$ con $\alpha(0) = \beta(0) = 0$ tales que, para todo $i, j, k \in \mathbb{N}$, si $\alpha(i) \leq j < \alpha(i+1)$ y $\beta(i) \leq k < \beta(i+1)$, se cumple que $\pi(j) H \rho(k)$.*

Por ejemplo, el siguiente diagrama muestra el comienzo de dos caminos que encajan, con elementos relacionados unidos por líneas discontinuas y donde $\alpha(0) = \beta(0) = 0$, $\alpha(1) = 2$, $\beta(1) = 3$, $\alpha(2) = 5$.



Definición 5.5 *Dados dos sistemas de transiciones $\mathcal{A}$ y $\mathcal{B}$, una* simulación tartamuda de sistemas de transiciones *$H : \mathcal{A} \longrightarrow \mathcal{B}$ es una relación binaria $H \subseteq A \times B$ tal que si aHb entonces para cada camino π en $\mathcal{A}$ que comience en a existe un camino ρ en $\mathcal{B}$ que comienza en b que H-encaja con π.*

Si H es una función decimos que H es un morfismo tartamudo de sistemas de transiciones. *Si tanto H como H^{-1} son simulaciones tartamudas, entonces decimos que H es una* bisimulación tartamuda.

Las simulaciones tartamudas de sistemas de transiciones se pueden componer (Manolios, 2001, lema 4) y junto con los sistemas de transiciones definen una categoría que denotamos **STSys**.

La extensión a las estructuras de Kripke es obvia:

Definición 5.6 *Dadas dos estructuras de Kripke $\mathcal{A} = (A, \rightarrow_{\mathcal{A}}, L_{\mathcal{A}})$ y $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ sobre AP, una AP-*simulación tartamuda *$H : \mathcal{A} \longrightarrow \mathcal{B}$ es una simulación tartamuda de sistemas de transiciones $H : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ tal que si aHb entonces $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$. Decimos que la AP-simulación tartamuda H es* estricta *si aHb implica que $L_{\mathcal{B}}(b) = L_{\mathcal{A}}(a)$.*

De nuevo, las AP-simulaciones tartamudas se pueden componer y definen una categoría $\mathbf{KSSim}_{AP}$ con las correspondientes subcategorías para simulaciones tartamudas estrictas y morfismos.

Observación 5.1 Nuestra definición de simulación tartamuda está estrechamente relacionada con la utilizada por Manolios (2001, 2003), pero con algunas diferencias técnicas y metodológicas. Él define sus simulaciones sobre un único conjunto obtenido mediante la unión disjunta de las dos estructuras de Kripke, y constan de dos ingredientes distintos: la relación de simulación y una función de refinado que "toma prestada" para la estructura de origen la información de etiquetado de la estructura destino.

Ya comentamos en la sección 4.2 que la noción de AP-simulación era muy general y que incluso permitía relaciones vacías. El mismo comentario se aplica ahora a las AP-simulaciones tartamudas y también, tal y como ocurrió entonces, se puede demostrar que todas las AP-simulaciones tartamudas surgen a partir de relaciones totales.

Proposición 5.4 *Sea $H : \mathcal{A} \longrightarrow \mathcal{B}$ una AP-simulación tartamuda. Se tiene entonces que, para toda subestructura de Kripke completa $\mathcal{B}' \subseteq \mathcal{B}$, la terna $H^{-1}(\mathcal{B}') = (H^{-1}(B'), \rightarrow_{\mathcal{A}} \cap H^{-1}(B') \times H^{-1}(B'), L_{\mathcal{A}}|_{H^{-1}(B')})$ es una subestructura de Kripke completa de $\mathcal{A}$. En particular, $H^{-1}(\mathcal{B})$ es una subestructura de Kripke completa de $\mathcal{A}$.*

Demostración. Tenemos que mostrar que la relación de transición es total y que $H^{-1}(\mathcal{B}')$ es completa en $\mathcal{A}$.

Sea a un elemento de $H^{-1}(B')$ tal que $a \rightarrow_{\mathcal{A}} a'$ (que existe porque $\rightarrow_{\mathcal{A}}$ es total) y sea π un camino en $\mathcal{A}$ tal que $\pi(0) = a$ y $\pi(1) = a'$. Por definición, existe $b \in B'$ tal que aHb. Ahora, como H es una AP-simulación tartamuda existe un camino ρ en $\mathcal{B}$ que comienza en b y que H-encaja con π, y puesto que $\mathcal{B}'$ es completa en $\mathcal{B}$, ρ es en realidad un camino en $\mathcal{B}'$. Como ρ H-encaja con π, se tiene $a'H\rho(i)$ para algún i; así $a' \in H^{-1}(B')$, $\rightarrow_{H^{-1}(\mathcal{B}')}$ es total y $H^{-1}(\mathcal{B}')$ es completa en $\mathcal{A}$. □

La definición de cuándo una simulación refleja la satisfacción de una fórmula tiene que ser ligeramente modificada en este contexto para el caso de los caminos.

Definición 5.7 *Una AP-simulación tartamuda $H : \mathcal{A} \longrightarrow \mathcal{B}$* refleja *la satisfacción de una fórmula $\varphi \in \mathrm{CTL}^*(AP)$ cuando:*

- *φ es una fórmula de estado y $\mathcal{B}, b \models \varphi$ y aHb implica que $\mathcal{A}, a \models \varphi$; o bien*
- *φ es una fórmula de camino y $\mathcal{B}, \rho \models \varphi$ y ρ H-encaja con π implica que $\mathcal{A}, \pi \models \varphi$.*

Es claro que el operador "siguiente" **X** de la lógica temporal no es reflejado por las AP-simulaciones tartamudas; sin embargo, si restringimos nuestra atención a $\mathrm{ACTL}^* \setminus \mathbf{X}(AP)$ y $\mathrm{ACTL}^* \setminus \{\neg, \mathbf{X}\}(AP)$, es decir, a los fragmentos de la lógica que no contienen **X**, las fórmulas sí se reflejan. En la práctica la eliminación del operador **X** no es una gran pérdida porque, como se defiende en Lamport (1983), las propiedades en las que uno está interesado no suelen centrarse en que algo ocurra en el siguiente paso sino tan solo en que termine ocurriendo, por lo que no utilizan **X** en sus enunciados.

Teorema 5.2 *Las AP-simulaciones tartamudas reflejan la satisfacción de las fórmulas en la lógica* $\mathrm{ACTL}^* \setminus \{\neg, \mathbf{X}\}(AP)$. *Además, las AP-simulaciones tartamudas estrictas también reflejan la satisfacción de las fórmulas en* $\mathrm{ACTL}^* \setminus \mathbf{X}(AP)$.

Demostración. Sea $H : \mathcal{A} \longrightarrow \mathcal{B}$ una AP-simulación tartamuda y supongamos que aHb y que ρ H-encaja con π por medio de α y β. Procedemos por inducción simultánea sobre la estructura de las fórmulas de estado y de camino.

Para una proposición atómica p, si $\mathcal{B}, b \models p$ entonces $p \in L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$ y por lo tanto $\mathcal{A}, a \models p$. El resultado es trivial para $\top$ y $\bot$. Para una fórmula de estado $\mathbf{A}\varphi$, si $\mathcal{B}, b \models \mathbf{A}\varphi$ entonces $\mathcal{B}, \rho' \models \varphi$ para todos los caminos ρ' en $\mathcal{B}$ que comienzan en b. Sea entonces π' un camino en $\mathcal{A}$ que comience en a y, abusando de la notación, escribimos $H(\pi')$ para uno cualquiera de los caminos que H-encajan con π' en $\mathcal{B}$ y que comienzan en b. Entonces, $\mathcal{B}, H(\pi') \models \varphi$ y, por la hipótesis de inducción, $\mathcal{A}, \pi' \models \varphi$ y por lo tanto $\mathcal{A}, a \models \mathbf{A}\varphi$. El resultado para los operadores lógicos $\vee$ y $\wedge$, tanto para fórmulas de estado como de camino, se sigue inmediatamente a partir de la hipótesis de inducción.

Si $\mathcal{B}, \rho \models \mathbf{F}\varphi$ entonces existe $n \in \mathbb{N}$ tal que $\mathcal{B}, \rho^n \models \varphi$. Sea i el único número natural tal que $\beta(i) \leq n < \beta(i+1)$. Se tiene entonces que $\rho^{\beta(i)}$, pero también ρ^n, H-encaja con $\pi^{\alpha(i)}$ y, por la hipótesis de inducción, $\mathcal{A}, \pi^{\alpha(i)} \models \varphi$ y por lo tanto $\mathcal{A}, \pi \models \mathbf{F}\varphi$.

Si $\mathcal{B}, \rho \models \varphi_1 \mathbf{U} \varphi_2$ entonces existe $n \in \mathbb{N}$ tal que $\mathcal{B}, \rho^n \models \varphi_2$ y, para todo $m < n$, $\mathcal{B}, \rho^m \models \varphi_1$. Sea i el único número natural tal que $\beta(i) \leq n < \beta(i+1)$. Entonces ρ^n H-encaja con $\pi^{\alpha(i)}$ y, por la hipótesis de inducción, $\mathcal{A}, \pi^{\alpha(i)} \models \varphi_2$. Sea ahora $m < \alpha(i)$. Si j es el único número natural tal que $\alpha(j) \leq m < \alpha(j+1)$, como α es estrictamente creciente debe tenerse que $j < i$, y como β también es estrictamente creciente, $\beta(j) < \beta(i) \leq n$ con lo que $\mathcal{B}, \rho^{\beta(j)} \models \varphi_1$. Como $\rho^{\beta(j)}$ H-encaja con π^m, por hipótesis de inducción, $\mathcal{A}, \pi^m \models \varphi_1$ y por lo tanto $\mathcal{A}, \pi \models \varphi_1 \mathbf{U} \varphi_2$.

Las demostraciones para **R** y **G** son similares.

En el caso de las AP-simulaciones tartamudas estrictas es suficiente con consider tan solo fórmulas en forma normal negativa. La demostración es exactamente como la anterior, aunque ahora hay que considerar también el caso adicional en el que la fórmula es de la forma $\neg p$ para una proposición atómica p. En este caso, si $\mathcal{B}, b \models \neg p$ entonces $p \notin L_{\mathcal{B}}(b)$ y, como H es estricta, se sigue que $p \notin L_{\mathcal{A}}(a)$ y $\mathcal{A}, a \models \neg p$ como se exigía. □

Finalmente podemos combinar las dos extensiones de la noción de simulación (sobre distintos conjuntos de proposiciones atómicas y tartamudas) en una única definición.

Definición 5.8 *Dada una estructura de Kripke $\mathcal{A}$ sobre un conjunto AP de proposiciones atómicas y una estructura de Kripke $\mathcal{B}$ sobre un conjunto AP', una* simulación tartamuda *(resp.* simulación tartamuda estricta*)* $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$ *consiste en una función* $\alpha : AP \longrightarrow \text{State}\backslash\{\neg, \mathbf{X}\}(AP')$ *(resp.* $\alpha : AP \longrightarrow \text{State} \setminus \mathbf{X}(AP')$*) y una AP-simulación tartamuda (resp. AP-simulación tartamuda estricta)* $H : \mathcal{A} \longrightarrow \mathcal{B}|_\alpha$.

Nótese que hemos restringido el rango de α prohibiendo el uso de **X**. Esto está en consonancia con el hecho de que el operador "siguiente" no tiene ningún sentido en presencia de tartamudeo. Por otra parte, las nociones de (AP)-morfismo tartamudo, (AP)-morfismo tartamudo estricto y (AP)-bisimulación tartamuda se definirían de la manera esperada. De nuevo, escribiremos habitualmente $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ en lugar de $(\alpha, H) : (AP, \mathcal{A}) \longrightarrow (AP', \mathcal{B})$.

Proposición 5.5 *Si $(\alpha, F) : \mathcal{A} \longrightarrow \mathcal{B}$ y $(\beta, G) : \mathcal{B} \longrightarrow C$ son simulaciones tartamudas, la composición $(\beta, G) \circ (\alpha, F) = (\overline{\beta} \circ \alpha, G \circ F)$ también es una simulación tartamuda.*

Demostración. Supongamos que $\mathcal{A}$ es una estructura de Kripke sobre el conjunto de proposiciones atómicas AP: tenemos que comprobar que $G \circ F : \mathcal{A} \longrightarrow C|_{\overline{\beta} \circ \alpha}$ es una AP-simulación tartamuda. F y G son simulaciones tartamudas de los sistemas de transiciones subyacentes y por lo tanto, como se demuestra en Manolios (2001, lema 4), su composición también es una simulación tartamuda de sistemas de transiciones. Sean ahora $a \in A$ y $c \in C$ tales que $a(G \circ F)c$, y sea $p \in L_{C|_{\overline{\beta} \circ \alpha}}(c)$. Entonces existe $b \in B$ tal que aFb y bGc y tenemos la siguiente cadena de equivalencias e implicaciones:

$$
\begin{array}{rcll}
p \in L_{C|_{\overline{\beta} \circ \alpha}}(c) & \Longleftrightarrow & C, c \models \overline{\beta}(\alpha(p)) & \text{(por definición)} \\
& \Longleftrightarrow & C|_\beta, c \models \alpha(p) & \text{(por la proposición 5.1)} \\
& \Longrightarrow & \mathcal{B}, b \models \alpha(p) & \text{(por el teorema 5.2)} \\
& \Longleftrightarrow & \mathcal{B}|_\alpha, b \models p & \text{(por la proposición 5.1)} \\
& \Longrightarrow & A, a \models p & \text{(por el teorema 5.2)} \\
& \Longleftrightarrow & p \in L_{\mathcal{A}}(a) &
\end{array}
$$

Esto es, $L_{C|_{\overline{\beta}\circ\alpha}}(c) \subseteq L_{\mathcal{A}}(a)$, con lo que $G \circ F$ es una AP-simulación tartamuda. □

Por lo tanto, tenemos una categoría **KSSim** de estructuras de Kripke y simulaciones tartamudas, con las correspondientes subcategorías para AP-simulaciones tartamudas, bisimulaciones tartamudas, etc. Como ya vimos cuando presentamos por primera vez simulaciones entre distintos niveles en la sección 5.1, para que las simulaciones tartamudas estrictas se puedan componer es necesario que el rango de las funciones α sea de la forma Bool(AP).

El siguiente teorema generaliza todos los resultados anteriores sobre reflexión de la relación de satisfacción por parte de diversos tipos de simulaciones.

Teorema 5.3 *Las simulaciones tartamudas siempre reflejan la satisfacción de las fórmulas en* ACTL$^*\backslash\{\neg, \mathbf{X}\}$. *Además, las simulaciones tartamudas estrictas también reflejan la satisfacción de fórmulas en* ACTL$^* \setminus \mathbf{X}$.

Demostración. Es una consecuencia sencilla del teorema 5.2 y la proposición 5.1. □

Observación 5.2 En realidad este resultado es cierto incluso si permitimos funciones generales de la forma $\alpha : AP \to \text{State}\backslash\neg(AP')$ en la definición de las simulaciones tartamudas; la restricción a fórmulas sin el operador "siguiente" solo es necesaria para que la composición quede bien definida.

La definición 5.5 de más arriba caracteriza las simulaciones tartamudas en términos de caminos infinitos. En Manolios (2001) se presenta una caracterización alternativa, más finitaria, llamada simulación bien fundada, que también se puede adaptar a nuestro contexto.

Definición 5.9 *Sean $\mathcal{A} = (A, \to_{\mathcal{A}})$ y $\mathcal{B} = (B, \to_{\mathcal{B}})$ sistemas de transiciones. Una relación $H \subseteq A \times B$ es una* simulación bien fundada de sistemas de transiciones *de $\mathcal{A}$ en $\mathcal{B}$ si existen funciones $\mu : A \times B \longrightarrow W$ y $\mu' : A \times A \times B \longrightarrow \mathbb{N}$, con $(W, <)$ un orden bien fundado, tales que si aHb y $a \to_{\mathcal{A}} a'$ se tenga que:*

1. *existe b' tal que $b \to_{\mathcal{B}} b'$ y $a'Hb'$, o*
2. *$a'Hb$ y $\mu(a', b) < \mu(a, b)$, o*
3. *existe b' tal que $b \to_{\mathcal{B}} b'$, aHb' y $\mu'(a, a', b') < \mu'(a, a', b)$.*

Observación 5.3 Nótese que si H es una función tan solo se aplican las condiciones (1) y (2) y la función μ' resulta innecesaria.

Definición 5.10 *Dadas dos estructuras de Kripke $\mathcal{A} = (A, \to_{\mathcal{A}}, L_{\mathcal{A}})$ y $\mathcal{B} = (B, \to_{\mathcal{B}}, L_{\mathcal{B}})$ sobre AP, una relación $H \subseteq A \times B$ es una AP-*simulación bien fundada *si H es una simulación bien fundada de sistemas de transiciones y aHb implica que $L_{\mathcal{B}}(b) \subseteq L_{\mathcal{A}}(a)$.*

Teorema 5.4 (Manolios, 2001, teorema 4) *Sean $\mathcal{A} = (A, \to_{\mathcal{A}}, L_{\mathcal{A}})$ y $\mathcal{B} = (B, \to_{\mathcal{B}}, L_{\mathcal{B}})$ dos estructuras de Kripke sobre AP y $H \subseteq A \times B$. Entonces, H es una AP-simulación bien fundada si y solo si es una AP-simulación tartamuda.*

5.3 Resultados generales de representabilidad

¿Cuál es el motivo por el que quereremos utilizar teorías de reescritura para especificar estructuras de Kripke? Se trata de una razón *lógica*: de esta forma tenemos a nuestra disposición dos lógicas para especificar un sistema y sus predicados, concretamente la lógica ecuacional de pertenencia para especificar el tipo de datos de los estados y sus proposiciones atómicas, y la lógica de reescritura para especificar las transiciones del sistema. Esto resulta muy útil para razonar sobre las propiedades de un sistema especificado de esta manera. Por ejemplo, al hacer razonamiento deductivo sobre propiedades en lógica temporal podemos utilizar un sinnúmero de técnicas ecuacionales combinadas, con razonamiento en lógica temporal para demostrar que determinadas fórmulas se cumplen. Del mismo modo, para la comprobación de modelos es posible especificar muchas estructuras de Kripke a alto nivel y (asumiendo que sus conjuntos de estados alcanzables sean finitos) verificar sus propiedades utilizando un comprobador de modelos como el de Maude (Eker et al., 2002).

¿Hasta qué punto es *general* la lógica de reescritura con vistas a especificar estructuras de Kripke? Esto es, ¿podemos especificar de esta manera cualquier estructura de Kripke que nos pueda interesar? La respuesta es *sí*; es más, si la estructura de Kripke es *recursiva* entonces la correspondiente teoría de reescritura será finita y también recursiva en un sentido adecuado.

Esto nos lleva a las nociones de sistema de transiciones y estructura de Kripke recursivos. En lo que sigue, usamos la noción de conjunto recursivo y función recursiva en el sentido de Shoenfield (1971).

Definición 5.11 *Un sistema de transiciones* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ *se llama* recursivo *si* B *es un conjunto recursivo y existe una función recursiva* $next : B \longrightarrow \mathcal{P}_{\text{fin}}(B)$ *(donde* $\mathcal{P}_{\text{fin}}(B)$ *es el conjunto recursivo de los subconjuntos finitos de* B*) tal que* $a \rightarrow_{\mathcal{B}} b$ *si y solo si* $b \in next(a)$.

Definición 5.12 *Una estructura de Kripke* $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ *se llama* recursiva *si* $(B, \rightarrow_{\mathcal{B}})$ *es un sistema de transiciones recursivo,* AP *es un conjunto recursivo y la función* $\hat{L}_{\mathcal{B}} : B \times AP \longrightarrow Bool$ *que aplica un par* (a, p) *en* ***true*** *si* $p \in L_{\mathcal{B}}(a)$ *y en* ***false*** *en caso contrario, es recursiva.*

Las nociones de sistema de transiciones y estructura de Kripke recursivos de arriba capturan la clase de sistemas para los que podemos determinar de manera efectiva todos los sucesores de un estado cualquiera por la relación de reescritura en un paso. Esto es más fuerte que simplemente exigir que la relación de transición $\rightarrow_{\mathcal{B}}$ sea recursiva, ya que en tal caso el conjunto de sucesores de un estado en general solo sería en principio recursivamente enumerable (r.e.). Nótese que el ser una estructura de Kripke recursiva es una condición necesaria en la aplicación efectiva de un comprobador de modelos para comprobar la satisfacción de fórmulas en lógica temporal a partir de un estado inicial. En general, sin embargo, la recursividad no es una condición suficiente para la comprobación de modelos a menos que el conjunto de estados alcanzables desde el estado inicial sea finito.

Por el conocido teorema de Bergstra y Tucker (1980), los conjuntos y funciones recursivos coinciden con aquellos conjuntos y funciones que pueden ser especificados en una

signatura finita Σ utilizando un conjunto finito de ecuaciones E que sean Church-Rosser y terminantes. Los conjuntos soporte subyacentes del álgebra inicial $T_{\Sigma/E}$ son los conjuntos recursivos deseados y las operaciones subyacentes del álgebra suministran las funciones recursivas. En el contexto de las estructuras de Kripke, esto significa que si $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ es una estructura de Kripke recursiva entonces B, AP y $\hat{L}_{\mathcal{B}}$ pueden ser especificados por medio de una signatura y conjunto de ecuaciones finitos. En nuestro método, esto se consigue especificando B como el soporte de una familia k en un álgebra inicial $T_{\Sigma/E}$ con Σ finita y E Church-Rosser y terminante, y especificando $\hat{L}_{\mathcal{B}}$ (que se denota $\_ \models \_$ en nuestra terminología) en una extensión conservadora $(\Sigma', E \cup D) \supseteq (\Sigma, E)$, también Church-Rosser y terminante, en la que los predicados de estado Π han sido especificados.

¿Qué ocurre con la especificación de la relación de transición $\rightarrow_{\mathcal{B}}$? Es aquí donde las teorías de reescritura entran en juego.

Definición 5.13 *Sea $\mathcal{R} = (\Sigma, E \cup A, R)$ una teoría de reescritura finita tal que todas sus reglas son de la forma*

$$\lambda : (\forall X)\, t \longrightarrow t' \textbf{ if } \bigwedge_{i \in I} p_i = q_i \wedge \bigwedge_{j \in J} w_j : s_j\,, \qquad (\ddagger)$$

donde o bien $\bigcup_i vars(p_i) \cup vars(q_i) \cup \bigcup_j vars(w_j) \cup vars(t') \subseteq vars(t)$ o bien, de forma más general, las reglas (‡) *son* admisibles *(recuérdese la sección 2.4.1).*

Decimos que $\mathcal{R}$ es recursiva *si:*

1. *existe un* algoritmo de ajuste de patrones módulo *los axiomas ecuacionales A;*
2. *la teoría ecuacional $(\Sigma, E \cup A)$ es* Church-Rosser *y* terminante *(para términos cerrados) módulo A (Dershowitz y Jouannaud, 1990); y*
3. *las reglas R son* coherentes *(para términos cerrados) en el sentido de Viry (2002), relativas a las ecuaciones E módulo A.*

La última condición significa que no se pierde ninguna reescritura por el hecho de reducir un término a su forma canónica $can_{E/A}(t)$ con respecto a E (única módulo A) antes de aplicar alguna de las reglas, como se explicó al final de la sección 2.3.

Nótese que si $\mathcal{R}$ es una teoría de reescritura recursiva, entonces para toda familia k la relación de transición $\rightarrow^1_{\mathcal{R},k} \subseteq T_{\Sigma/E \cup A,k} \times T_{\Sigma/E \cup A,k}$ es recursiva. Ciertamente, dados $[u], [v] \in T_{\Sigma/E \cup A,k}$, por coherencia tenemos que

$$u \rightarrow^1_{\mathcal{R},k} v \iff \text{existe } w \text{ tal que } can_{E/A}(u) \rightarrow^1_{\mathcal{R},k} w \text{ y } can_{E/A}(w) = can_{E/A}(v)\,.$$

Por lo tanto, para decidir si $[u] \rightarrow^1_{\mathcal{R},k} [v]$ primero reducimos u a su forma canónica y a continuación intentamos ajustar cualquiera de las reglas (‡) en R con $can_{E/A}(u)$. Para cada sustitución θ que ajuste módulo A podemos entonces intentar encontrar una sustitución ρ módulo A que extienda θ a las variables en $vars(t') \setminus vars(t)$ (que puede no ser el conjunto vacío para reglas admisibles) y tal que

$$E \vdash \bigwedge_{i \in I} \rho(p_i) = \rho(q_i) \wedge \bigwedge_{j \in J} \rho(w_j) : s_j\,,$$

que es un problema decidible bajo los supuestos de que E sea Church-Rosser y terminante módulo A. Debido a la hipótesis de que todas las variables extras en $vars(t') \setminus vars(t)$ son introducidas incrementalmente por ecuaciones de ajuste en la condición y la existencia de un algoritmo de ajuste de patrones módulo A, existe solo un número finito $\rho_1, \ldots, \rho_n$ de sustituciones que extienden θ y satisfacen la condición de la regla, y además se pueden computar. Por ejemplo, para calcular los sucesores del término $f(a)$ por la regla

$$(\forall \{x, y, z\})\, f(x) \longrightarrow g(x, y, z) \textbf{ if } h(y, z) := h(x, b)\,,$$

donde h es un operador binario con atributo de conmutatividad y $h(y, z) := h(x, b)$ es una ecuación de ajuste, primero se obtendría la sustitución $\theta = \{x \mapsto a\}$. Las variables y y z quedarían entonces sin instanciar, pero como la regla es admisible y asumimos que disponemos de un algoritmo de ajuste módulo conmutatividad, a partir de la instanciación de $h(y, z) := h(x, b)$ con θ resulta que tan solo hay dos posible maneras de asignar valores a y y a z y dan lugar a las sustituciones $\rho_1 = \{x \mapsto a, y \mapsto a, z \mapsto b\}$ y $\rho_2 = \{x \mapsto a, y \mapsto b, z \mapsto a\}$.

Los estados sucesores de la regla (‡) son entonces descritos de manera efectiva como las formas canónicas de las reescrituras en un solo paso en las que el subtérmino $\theta(t)$ de $can_{E/A}(u)$ es reemplazado por $\rho_i(t')$. De esta forma, dado $[u] \in T_{\Sigma/E \cup A,k}$ tenemos una función recursiva $next_{\mathcal{R}} : T_{\Sigma/E \cup A,k} \longrightarrow \mathcal{P}_{\text{fin}}(T_{\Sigma/E \cup A,k})$.

Como consecuencia, si $\mathcal{R}$ es recursiva entonces $\mathcal{T}(\mathcal{R})_k$ también lo es. Además, si la extensión $(\Sigma', E \cup D) \supseteq (\Sigma, E)$ es conservadora, donde $E \cup D$ es Church-Rosser y terminante, $\mathcal{K}(\mathcal{R}, k)_\Pi$ es una estructura de Kripke recursiva.

El recíproco también es cierto, esto es, todo sistema y estructura de Kripke recursivos se pueden especificar de esta forma. Por el metateorema de Bergstra y Tucker (1980), cualquier estructura de Kripke recursiva $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ (y análogamente para un sistema de transiciones) se puede especificar mediante una teoría ecuacional finita en la lógica de pertenencia $(\Sigma, E \cup A)$, donde E es Church-Rosser y terminante módulo A. Se tienen familias k, $Prop$ y $Set\text{-}k$, con B y AP isomorfos a $T_{\Sigma/E,k}$ y $T_{\Sigma/E,Prop}$ respectivamente, un operador $holds : k\ Prop \longrightarrow Bool$ que corresponde a $\hat{L}_{\mathcal{B}}$ y un operador $next : k \longrightarrow Set\text{-}k$ que corresponde a la función que devuelve el sucesor de cada estado. También hay operadores

$$\{\_\} : k \longrightarrow Set\text{-}k\,, \quad \emptyset :\longrightarrow Set\text{-}k \quad \text{y} \quad \_,\_ : Set\text{-}k\ Set\text{-}k \longrightarrow Set\text{-}k$$

de manera que $\_,\_$ es la unión de conjuntos finitos que satisface los axiomas de asociatividad, conmutatividad y de elemento identidad ($\emptyset$) en A, así como la ecuación $(\forall x : k)\, \{x\}, \{x\} = \{x\}$. Para definir $\mathcal{R}$, añadimos una nueva familia $System$ y operadores $\langle \_ \rangle : k \longrightarrow System$ y $\_ \models \_ : System\ Prop \longrightarrow Bool$, la ecuación

$$(\forall x : k, y : k)\, (\langle x \rangle \models p) = holds(x, p)\,,$$

y la regla admisible

$$(\forall x : k, y : k, s : Set\text{-}k)\, \langle x \rangle \longrightarrow \langle y \rangle \textbf{ if } \{y\}, s = next(x)\,,$$

donde la ecuación en la condición es una ecuación de ajuste (Clavel et al., 2002a) que puede ser decidida ajustando el patrón $\{y\}, s$ con el conjunto finito computado por $next(x)$.

Entonces cada sustitución de ajuste θ suministra una reescritura en un solo paso que genera cada uno de los estados siguientes.

Al precio de permitir signaturas infinitas y perder la computabilidad, existe un resultado general de representabilidad que dice que *cualquier* sistema de transiciones y *cualquier* estructura de Kripke pueden ser modelados en la lógica de reescritura. Para ello, dado un sistema de transiciones $\mathcal{B} = (B, \rightarrow_{\mathcal{B}})$ podemos definir $\mathcal{R}_{\mathcal{B}}$ con una única familia *State*, $\Sigma_{nil,State} = B$ y reglas $a \longrightarrow b$ si y solo $a \rightarrow_{\mathcal{B}} b$. Y si extendemos $\mathcal{B}$ a una estructura de Kripke, la función de etiquetado $L_{\mathcal{B}}$ se puede modelar con ecuaciones $(a \models p) = \mathit{true}$ si $p \in L_{\mathcal{B}}(a)$, y $(a \models p) = \mathit{false}$ si $p \notin L_{\mathcal{B}}(a)$.

La cuestión interesante, sin embargo, no es si podemos o no representar cualquier estructura de Kripke: siempre podemos. La cuestión es que tenemos una forma general de definir cualquier estructura de Kripke recursiva por medio de una teoría de reescritura *finita* que es Church-Rosser, terminante y coherente.

Como veremos en la sección 5.4, observaciones similares también se aplican a las simulaciones. Así, tendremos categorías de teorías de reescritura que pueden representar todos los sistemas de transiciones (resp. estructuras de Kripke) y todas las simulaciones entre ellos, y categorías de teorías de reescritura que pueden representar todos los sistemas de transiciones (resp. estructuras de Kripke) y todos los morfismos recursivos entre ellos.

5.4 Simulaciones algebraicas

Ya hemos señalado que, para razonar sobre sistemas computacionales, estos se pueden describir de manera abstracta por medio de sistemas de transiciones y estructuras de Kripke. Como ya se ha explicado varias veces a lo largo de la tesis, la lógica de reescritura se puede utilizar para especificar ambas clases de estructuras de manera natural y modular. Nuestro objetivo ahora es estudiar cómo relacionar teorías de reescritura diferentes y cómo elevar al nivel de especificación todos los resultados anteriores sobre simulaciones de estructuras de Kripke. Para esto se pueden considerar cuatro maneras sucesivamente más generales de definir simulaciones entre teorías de reescritura que especifican sistemas concurrentes:

1. La manera más fácil de definir un morfismo de simulación para una teoría de reescritura (Σ, E, R) es por medio de una *abstracción ecuacional*, añadiendo nuevas ecuaciones, digamos E', para obtener un sistema cociente especificado por $(\Sigma, E \cup E', R)$.

2. El método anterior se puede generalizar considerando, en lugar de simplemente inclusiones de teorías $(\Sigma, E) \subseteq (\Sigma, E \cup E')$, *interpretaciones de teorías* arbitrarias $H : (\Sigma, E) \longrightarrow (\Sigma', E')$ que permitan transformaciones en la representación de los estados.

3. Una tercera alternativa consiste en definir un *morfismo* entre teorías de reescritura $\mathcal{R}$ y $\mathcal{R}'$ directamente al nivel de sus estructuras de Kripke asociadas, mediante *funciones definidas ecuacionalmente*.

4. Finalmente, el caso más general se obtiene definiendo simulaciones cualesquiera entre teorías de reescritura $\mathcal{R}$ y $\mathcal{R}'$ por medio de *relaciones definidas mediante reescritura*.

Para cada uno de estos métodos de definir simulaciones existen las correspondientes *condiciones de corrección* asociadas que deben ser verificadas. Las abstracciones ecuacionales fueron ya tratadas en el capítulo 4; en lo que sigue nos ocuparemos de los casos restantes.

5.4.1 Morfismos de simulación como funciones definidas ecuacionalmente

En esta sección describimos los detalles de las categorías que fueron mencionadas al final de la sección 5.3. Consideremos en primer lugar los sistemas de transiciones. Para ello, definimos una categoría **SRWTh** cuyos objetos son los pares $(\mathcal{R}, k)$, con $\mathcal{R}$ una teoría de reescritura y k una familia distinguida en $\mathcal{R}$. Los objetos en la subcategoría **RecSRWTh** también son pares $(\mathcal{R}, k)$, pero ahora, como estamos interesados en estructuras recursivas, exigimos que la teoría $\mathcal{R}$ sea recursiva.

¿Qué ocurre con los morfismos? Al final de la sección 5.3 mostramos que cualquier sistema de transiciones se puede definir en la lógica de reescritura. Del mismo modo, cualquier morfismo tartamudo de sistemas de transiciones (y en particular aquellos que no sean tartamudos) $h : \mathcal{A} \longrightarrow \mathcal{B}$ se puede describir ecuacionalmente en una extensión conservadora de $\mathcal{R}_\mathcal{A}$ y $\mathcal{R}_\mathcal{B}$ sin más que añadir una ecuación $h(a) = b$ para todo $a \in A$ que h aplique en b. Por lo tanto, la siguiente definición no implica ninguna pérdida de generalidad.

Definición 5.14 *Una flecha* $(\mathcal{R}_1, k_1) \longrightarrow (\mathcal{R}_2, k_2)$ *en* **SRWTh**, *llamada* morfismo tartamudo algebraico de sistemas de transiciones, *es un morfismo tartamudo* $h : \mathcal{T}(\mathcal{R}_1)_{k_1} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{k_2}$ *tal que existe una teoría extendida* (Ω, G) *que contiene de forma disjunta las partes ecuacionales de* $\mathcal{R}_1$ *y* $\mathcal{R}_2$ *en la que* h *se puede definir ecuacionalmente a través de un operador* $h : k'_1 \longrightarrow k'_2$ *(donde los apóstrofos indican los correspondientes nombres para las copias disjuntas de las familias).*

Nótese que tan solo pedimos la existencia de (Ω, G); para definir la categoría no necesitamos elegir ninguna extensión en particular. Las flechas en **RecSRWTh** se definen análogamente pero con el requisito añadido de que las ecuaciones que definen h en la extensión (Ω, G) sean Church-Rosser y terminantes.

Ahora podemos mostrar que la construcción definida en la sección 2.3 que asocia un sistema de transiciones a una teoría de reescritura con una familia escogida de estados es en realidad un *funtor*. De manera más precisa, definimos $\mathcal{T} : \mathbf{SRWTh} \longrightarrow \mathbf{STSys}$ como sigue:

- para los objetos, $\mathcal{T}(\mathcal{R}, k) = \mathcal{T}(\mathcal{R})_k$;
- para las flechas $h : (\mathcal{R}_1, k_1) \longrightarrow (\mathcal{R}_2, k_2)$, $\mathcal{T}(h) = h$.

Denotemos con **RecSTSys** la categoría cuyos objetos son los sistemas de transiciones recursivos y cuyas flechas $h : \mathcal{A} \longrightarrow \mathcal{B}$ son los morfismos tartamudos de sistemas de

transiciones tales que h es recursiva; el siguiente resultado se sigue inmediatamente de las definiciones y construcciones anteriores.

Proposición 5.6 *El funtor* $\mathcal{T}$: **SRWTh** $\longrightarrow$ **STSys** *es sobreyectivo sobre objetos, completo y fiel, con la restricción obvia para morfismos no tartamudos. Del mismo modo,* $\mathcal{T}$: **RecSRWTh** $\longrightarrow$ **RecSTSys** *es sobreyectivo sobre objetos salvo isomorfismo, completo y fiel (de nuevo con la correspondiente restricción).*

Centremos ahora nuestra atención sobre las estructuras de Kripke. Para ello vamos a necesitar considerar una teoría $BOOL_{\models}$ que extienda $BOOL$ con dos nuevas familias, $State$ y $Prop$, y un nuevo operador $\_ \models \_ : State\ Prop \longrightarrow Bool$.

Los objetos de la categoría $\mathbf{SRWTh}_{\models}$ son especificaciones de estructuras de Kripke en lógica de reescritura y las flechas definen morfismos tartamudos entre ellas. Como ya se explicó, en la descripción de una estructura de Kripke intervienen tanto un nivel de especificación de sistema como un nivel de especificación de propiedades, concretamente el nivel del sistema de transiciones y el nivel en el que se añaden las proposiciones. Por lo tanto, los objetos en $\mathbf{SRWTh}_{\models}$ serán pares formados por una teoría de reescritura especificando el sistema de transiciones subyacente y una teoría ecuacional especificando las proposiciones atómicas relevantes. Sin embargo, también añadiremos una tercera componente cuya función será la de distinguir la familia escogida para los estados, asegurándonos así de que la teoría $BOOL$ permanece fija a lo largo de las simulaciones. De manera más precisa, los objetos de la categoría $\mathbf{SRWTh}_{\models}$ vienen dados por ternas $(\mathcal{R}, (\Sigma', E \cup D), J)$ donde:

1. $\mathcal{R} = (\Sigma, E, R)$ es una teoría de reescritura especificando el sistema de transiciones.
2. $(\Sigma, E) \subseteq (\Sigma', E \cup D)$ es una extensión conservadora, que también contiene y extiende de forma conservadora a la teoría $BOOL$ de los booleanos, que define las proposiciones atómicas satisfechas por los estados. Definimos $\Pi \subseteq \Sigma'$ como la subsignatura de los operadores de coaridad $Prop$.
3. $J : BOOL_{\models} \longrightarrow (\Sigma', E \cup D)$ es un morfismo de teorías ecuacionales de pertenencia que selecciona la familia distinguida de los estados $J(State)$ tal que:
   a) es la identidad cuando se restringe a $BOOL$,
   b) $J(Prop) = Prop$, y
   c) $J(\_ \models \_ : State\ Prop \to Bool) = \_ \models \_ : J(State)\ Prop \to Bool$.

Como se explicó en la sección 4.5 podemos asumir, sin pérdida de generalidad, que $\mathcal{R}$ está libre de $J(State)$-bloqueo, esto es, que la relación $\to^1_{\mathcal{R}, J(State)}$ es total.

Los objetos en la subcategoría $\mathbf{RecSRWTh}_{\models}$ son también ternas $(\mathcal{R}, (\Sigma', E \cup D), J)$ pero ahora exigimos que la teoría de reescritura $\mathcal{R}$ sea recursiva y la extensión conservadora $(\Sigma, E) \subseteq (\Sigma', E \cup D)$ finita, Church-Rosser y terminante.

¿Qué ocurre con las flechas? De nuevo, cualquier morfismo tartamudo (y no tartamudo) de estructuras de Kripke $(\alpha, h) : \mathcal{A} \longrightarrow \mathcal{B}$ se puede definir ecuacionalmente en una extensión conservadora de $\mathcal{R}_{\mathcal{A}}$ y $\mathcal{R}_{\mathcal{B}}$, con lo que la siguiente definición no conlleva ninguna pérdida de generalidad.

Definición 5.15 *Las flechas* $(\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *en* $\mathbf{SRWTh}_{\models}$*, que llamaremos* morfismos tartamudos algebraicos*, son los pares* (α, h) *tales que:*

1. (α, h) *es un morfismo tartamudo de estructuras de Kripke* $(\alpha, h) : \mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$.

2. *Existe una teoría* (Ω, G) *que contiene de manera conservadora copias disjuntas de* $(\Sigma'_1, E_1 \cup D_1)$ *y* $(\Sigma'_2, E_2 \cup D_2)$ *en la que* α *y* h *se pueden definir ecuacionalmente a través de operadores* $\alpha : Prop_1 \longrightarrow StateForm$ *y* $h : J_1(State)_1 \longrightarrow J_2(State)_2$ *en* Ω*; los subíndices 1 y 2 indican los correspondientes nombres para las copias disjuntas de las familias y StateForm es una nueva familia para representar fórmulas de estado sobre* $Prop_2$.

Nótese de nuevo el cuantificador existencial en relación a la extensión (Ω, G).

Como por los resultados generales de representabilidad siempre podemos encontrar una extensión (Ω, G) en la que tal función h pueda ser definida ecuacionalmente, la categoría está bien definida, pues para cada composición podemos hacer lo mismo.

El detalle importante es que en el caso en que α y h son *recursivos* y $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$ y $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ son objetos en $\mathbf{RecSRWTh}_{\models}$, entonces por el metarresultado de Bergstra y Tucker (1980) siempre podemos encontrar una extensión *finita* (Ω, G) que es tanto una extensión conservadora de los componentes como Church-Rosser y terminante, y en la que α y h se pueden especificar mediante ecuaciones Church-Rosser y terminantes. Por lo tanto, definimos las flechas en $\mathbf{RecSRWTh}_{\models}$, llamadas *morfismos tartamudos algebraicos recursivos*, como pares (α, h) igual que antes, pero ahora con el requisito añadido de que tanto α como h puedan ser definidos mediante ecuaciones Church-Rosser y terminantes en la extensión (Ω, G).

Naturalmente, todas estas construcciones también se pueden aplicar sobre simulaciones no tartamudas, dando lugar a las subcategorías $\mathbf{RWTh}_{\models}$ y $\mathbf{RecRWTh}_{\models}$.

$$\begin{array}{ccc} \mathbf{RecRWTh}_{\models} & \hookrightarrow & \mathbf{RecSRWTh}_{\models} \\ \downarrow & & \downarrow \\ \mathbf{RWTh}_{\models} & \hookrightarrow & \mathbf{SRWTh}_{\models} \end{array}$$

Ahora ya podemos demostrar que la construcción definida en la sección 4.1, que asocia una estructura de Kripke a una teoría de reescritura con una familia de estados y predicados de estado escogidos, es un *funtor*. De forma precisa, definimos $\mathcal{K} : \mathbf{SRWTh}_{\models} \longrightarrow \mathbf{KSMap}$ como sigue:

- para objetos, $\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(State))_{\Pi}$;
- para flechas $(\alpha, h) : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$, $\mathcal{K}(\alpha, h) = (\alpha, h)$.

Ahora, si denotamos por **RecKSMap** la subcategoría cuyos objetos son estructuras de Kripke recursivas y cuyas flechas son morfismos tartamudos $(\alpha, h) : \mathcal{A} \longrightarrow \mathcal{B}$ tales que α y h son ambas funciones recursivas, la discusión previa se puede resumir así:

Proposición 5.7 *El funtor* $\mathcal{K}$: **SRWTh**$_{\models}$ $\longrightarrow$ **KSMap** *es sobreyectivo sobre los objetos, completo y fiel, con las restricciones obvias para morfismos no tartamudos. De manera similar,* $\mathcal{K}$: **RecSRWTh**$_{\models}$ $\longrightarrow$ **RecKSMap** *es sobreyectivo sobre los objetos salvo isomorfismo, completo y fiel (de nuevo con las correspondientes restricciones). Gráficamente:*

$$\begin{array}{ccc} \textbf{RecSRWTh}_{\models} & \hookrightarrow & \textbf{SRWTh}_{\models} \\ \downarrow \mathcal{K} & & \downarrow \mathcal{K} \\ \textbf{RecKSMap} & \hookrightarrow & \textbf{KSMap} \end{array}$$

El hecho de que $\mathcal{K}$ sea sobreyectiva sobre los objetos, completa y fiel, es un *resultado general de representabilidad* que dice que *todas* las estructuras de Kripke (resp. *todas las estructuras de Kripke recursivas*) y morfismos tartamudos se pueden representar mediante teorías de reescritura y funciones definidas ecuacionalmente (resp. teorías de reescritura recursivas y funciones recursivas definidas ecuacionalmente).

Una cuestión importante es cómo verificar que una simulación tartamuda algebraica es "correcta", esto es, identificar un conjunto de obligaciones de prueba que aseguren que determinadas funciones α y h definidas ecuacionalmente definen de hecho una simulación tartamuda algebraica. Algunos criterios al respecto se discuten en la sección 5.8.

5.4.2 Simulaciones como relaciones definidas mediante reescritura

La construcción anterior, pese a ser muy general y aplicable en muchas situaciones, nos restringe a trabajar solo con funciones. Este inconveniente se puede evitar con una simple extensión de las ideas introducidas en la sección previa. Vamos a considerar tan solo el caso de las estructuras de Kripke, teniendo presente que todo lo que se cuente se puede trasladar también a sistemas de transiciones simplemente olvidando la estructura adicional introducida por las proposiciones atómicas.

Definimos una categoría **SRelRWTh**$_{\models}$ cuyos objetos son los de **SRWTh**$_{\models}$ y con las flechas que se describen a continuación.

Definición 5.16 *Una flecha* $(\mathcal{R}_1, (\Sigma_1', E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma_2', E_2 \cup D_2), J_2)$ *en* **SRelRWTh**$_{\models}$, *llamada* simulación tartamuda algebraica, *es un par* (α, H) *tal que:*

1. $(\alpha, H) : \mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1} \longrightarrow \mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ *es una simulación tartamuda de estructuras de Kripke.*

2. *Existe una teoría de reescritura* $\mathcal{R}_3$ *que contiene de manera conservadora copias disjuntas de* $(\Sigma_1', E_1 \cup D_1, R_1)$ *y* $(\Sigma_2', E_2 \cup D_2, R_2)$ *en la que* α *se puede definir ecuacionalmente a través de un operador* $\alpha : Prop_1 \longrightarrow StateForm$, *y* H *se define mediante reglas de reescritura que utilizan un operador* $H : J_1(State)_1\ J_2(State)_2 \longrightarrow Bool$ *tal que* xHy *si y solo si* $\mathcal{R}_3 \vdash H(x, y) \longrightarrow true$. *Aquí los subíndices 1 y 2 indican los correspondientes nombres para las copias disjuntas de las familias, y StateForm es una nueva familia para representar las fórmulas de estado sobre* $Prop_2$.

La subcategoría **RecSRelRWTh**$_{\models}$ de teorías recursivas y *simulaciones tartamudas algebraicas r.e.* se define de manera análoga, pero ahora exigimos que la teoría extendida $\mathcal{R}_3$ sea finita y *admisible* en el sentido de Clavel et al. (2002a). Esto es, $\mathcal{R}_3$ satisface requisitos parecidos a los de una teoría de reescritura recursiva, pero las condiciones de las reglas ahora pueden contener reescrituras en tanto en cuanto no introduzcan nuevas variables en sus lados izquierdos. Nótese que esto es equivalente a pedir que la relación H sea r.e.

Observación 5.4 Resulta interesante señalar que cuando trabajamos con funciones en **RecSRWTh**$_{\models}$ solo consideramos funciones recursivas, mientras que en **RecSRelRWTh**$_{\models}$ admitimos relaciones r.e. arbitrarias. Para nosotros esta es una extensión natural puesto que en general la composición de relaciones recursivas no es recursiva, mientras que la composición de relaciones r.e. es r.e.

Denotaremos por **RecKSSim** la categoría de estructuras de Kripke recursivas y simulaciones tartamudas $(\alpha, H) : \mathcal{A} \longrightarrow \mathcal{B}$ tales que α es recursiva y H es r.e. El funtor de olvido $\mathcal{K}$ se extiende de la manera obvia a las nuevas categorías, con lo que tenemos el siguiente resultado:

Proposición 5.8 *Con las definiciones anteriores,* $\mathcal{K}$: **SRelRWTh**$_{\models}$ $\longrightarrow$ **KSSim** *es sobreyectivo sobre los objetos, completo y fiel, y* $\mathcal{K}$: **RecSRelRWTh**$_{\models}$ $\longrightarrow$ **RecKSSim** *es sobreyectivo sobre los objetos salvo isomorfismo, completo y fiel. Gráficamente:*

$$\begin{array}{ccc} \textbf{RecSRelRWTh}_{\models} & \hookrightarrow & \textbf{SRelRWTh}_{\models} \\ \downarrow \mathcal{K} & & \downarrow \mathcal{K} \\ \textbf{RecKSSim} & \hookrightarrow & \textbf{KSSim} \end{array}$$

Este es el resultado de representabilidad más general posible para simulaciones tartamudas, tal y como las hemos definido. Muestra que en la lógica de reescritura podemos representar tanto estructuras de Kripke como simulaciones tartamudas, y que podemos utilizar la lógica de reescritura y la lógica de pertenencia para razonar sobre ellas.

5.5 Algunos ejemplos

Vamos a dedicar esta sección a ilustrar con tres ejemplos algunas situaciones en las que aparecen los morfismos y simulaciones tartamudos algebraicos y en los que se pueden utilizar estos para trasladar propiedades temporales de un sistema a otro.

5.5.1 La semántica de un lenguaje funcional

En Hennessy (1990) se define un lenguaje funcional simple llamado *Fpl* junto con tres semánticas diferentes: una semántica de evaluación bastante abstracta, una semántica de computación y una semántica más concreta que utiliza una máquina de pila.

La implementación en Maude de esas tres semánticas aparece descrita en Verdejo y Martí-Oliet (2003). La semántica de evaluación es muy abstracta y poco interesante desde

- Semántica de computación para *Fpl*.

```
rl [VarRc] : < rho, x > => < rho, rho(x) > .
rl [OpRc] : < rho, v op v' > => < rho, Ap(op,v,v') > .
crl [OpRc] : < rho, e op e' > => < rho', e'' op e' >
             if < rho, e > => < rho', e'' > .
crl [OpRc] : < rho, e op e' > => < rho', e op e'' >
             if < rho, e' > => < rho', e'' > .
crl [IfRc] : < rho, If be Then e Else e' > =>
             < rho', If be' Then e Else e' >
             if < rho, be > => < rho', be' > .
rl [IfRc] : < rho, If T Then e Else e' > => < rho, e > .
rl [IfRc] : < rho, If F Then e Else e' > => < rho, e' > .
crl [LocRc] : < rho, let x = e in e' > => < rho', let x = e'' in e' >
              if < rho, e > => < rho', e'' > .
rl [LocRc] : < rho, let x = v in e' > => < rho, e'[v / x] > .
```

- Semántica de computación para expresiones booleanas.

```
rl [BVarRc] : < rho, bx > => < rho, rho(bx) > .
rl [BOpRc] : < rho, bv bop bv' > => < rho, Ap(bop,bv,bv') > .
crl [BOpRc] : < rho, be bop be' > => < rho', be'' bop be' >
              if < rho, be > => < rho', be'' > .
crl [BOpRc] : < rho, be bop be' > => < rho', be bop be'' >
              if < rho, be' > => < rho', be'' > .
crl [NotRc] : < rho, Not be > => < rho', Not be' >
              if < rho, be > => < rho', be' > .
rl [NotRc] : < rho, Not T > => < rho, F > .
rl [NotRc] : < rho, Not F > => < rho, T > .
crl [EqRc] : < rho, Equal(e,e') > => < rho, Equal(e'',e') >
             if < rho, e > => < rho', e'' > .
crl [EqRc] : < rho, Equal(e,e') > => < rho, Equal(e,e'') >
             if < rho, e' > => < rho', e'' > .
crl [EqRc] : < rho, Equal(v,v') > => < rho, T >
             if v = v' .
crl [EqRc] : < rho, Equal(v,v') > => < rho, F > if v =/= v'.
```

Figura 5.1: Reglas semánticas para la semántica de computación.

el punto de vista de los sistemas de transiciones: todas las expresiones se evalúan en un único paso. Sin embargo, las otras dos semánticas son mucho más concretas de modo que la evaluación de una sola expresión requiere la ejecución de bastantes pasos. Por lo tanto, tiene sentido estudiar la relación entre las ejecuciones en cada una de ellas y expresar su sintonía mediante una simulación tartamuda.

Un estado de la máquina de pila, utilizando la sintaxis de Maude con la que la vamos a especificar, es una terna

```
< ST, rho, e >,
```

- Reglas de análisis para la máquina de pila.

```
rl [Opm1] : < ST, rho, e op e' . C > => < ST, rho, e . e' . op . C > .
rl [Opm1] : < ST, rho, be op be' . C > =>
            < ST, rho, be . be' . bop . C > .
rl [Ifm1] : < ST, rho, If be Then e Else e' . C > =>
            < ST, rho, be . if(e, e') . C > .
rl [Locm1] : < ST, rho, let x = e in e' . C > =>
             < ST, rho, e . < x, e' > . C > .
rl [Notm1] : < ST, rho, Not be . C > => < ST, rho, be . not . C > .
rl [Eqm1] : < ST, rho, Equal(e, e') . C > =>
            < ST, rho, e . e' . equal . C > .
```

- Reglas de aplicación para la máquina abstracta.

```
rl [Opm2] : < v' . v . ST, rho, op . C > =>
            < Ap(op,v,v') . ST, rho, C > .
rl [Opm2] : < bv' . bv . ST, rho, bop . C > =>
            < Ap(bop,bv,bv') . ST, rho, C > .
crl [Varm] : < ST, rho, x . C > => < v . ST, rho, C >
             if v := lookup(rho,x) .
crl [Varm] : < ST, rho, bx . C > => < bv . ST, rho, C >
             if bv := lookup(rho,bx) .
rl [Valm] : < ST, rho, v . C > => < v . ST, rho, C > .
rl [Valm] : < ST, rho, bv . C > => < bv . ST, rho, C > .
rl [Notm2] : < T . ST, rho, not . C > => < F . ST, rho, C > .
rl [Notm2] : < F . ST, rho, not . C > => < T . ST, rho, C > .
crl [Eqm2] : < v . v' . ST, rho, equal . C > => < T . ST, rho, C >
             if v = v' .
crl [Eqm2] : < v . v' . ST, rho, equal . C > => < F . ST, rho, C >
             if v =/= v' .
rl [Ifm2] : < T . ST, if(e, e') . C > => < ST, rho, e . C > .
rl [Ifm2] : < F . ST, rho, if(e, e') . C > => < ST, rho, e' . C > .
rl [Locm2] : < v . ST, rho, < x, e > . C > =>
             < ST, (x,v) . rho, e . pop . C > .
rl [Pop] : < ST, (x,v) . rho, pop . C > => < ST, rho, C > .
```

Figura 5.2: Reglas semánticas para la máquina de pila.

donde `ST` es una pila de valores, `rho` es un entorno que asigna valores a las variables y `e` es una expresión. Un estado para la semántica de computación es un par

$$\texttt{< rho, e >}$$

con `rho` un entorno y `e` una expresión. Las relaciones de transición

$$\texttt{< ST, rho, e >} \longrightarrow \texttt{< ST', rho', e' >} \quad \text{y} \quad \texttt{< rho, e >} \longrightarrow \texttt{< rho',e' >}$$

definidas en Hennessy (1990) fueron traducidas a la lógica de reescritura por Verdejo y

Martí-Oliet (2003). Aparecen en las figuras 5.1 y 5.2, utilizando la notación de Maude. Siguiendo a Verdejo y Martí-Oliet (2003), para poder definir un sistema de transiciones extendemos el lado derecho de la relación de transición de la semántica de computación, que tan solo contiene el valor numérico en Hennessy (1990). A diferencia de los dos trabajos citados, aquí no vamos a considerar funciones.

Estas definiciones dan lugar a dos sistemas de transiciones, $\mathcal{S} = (S, \rightarrow_{\mathcal{S}})$ y $\mathcal{C} = (C, \rightarrow_{\mathcal{C}})$, para la máquina de pila y la semántica de computación respectivamente. Para demostrar la corrección de la implementación de la máquina de pila con relación a la semántica de computación vamos a mostrar que existe una simulación tartamuda algebraica de sistemas de transiciones $h : \mathcal{S} \longrightarrow \mathcal{C}$.

Intuitivamente, `< empty, rho, e >`, donde `empty` se utiliza tanto para representar una pila de valores vacía como el entorno que no asocia ningún valor a ninguna variable, debería estar relacionado con `< rho, e >`. Consideremos la siguiente derivación:

```
< empty, empty, 2 + 3 >  →S  < empty, empty, 2 . 3 . + >
                         →S  < 2, empty, 3 . + >
                         →S  < 3 . 2, empty, + >
                         →S  < 5, empty, empty >
```

Los estados segundo, tercero y cuarto en la derivación arrastran exactamente la misma información, aunque en distinto orden. Las reglas utilizadas para alcanzarlos son ejemplos de lo que en Hennessy (1990) se llaman *reglas de análisis*. Por lo tanto parece apropiado relacionarlos con el mismo estado que el primero, concretamente `< empty, 2 + 3 >`. La situación es distinta para el último estado: parte de la información se ha perdido, por lo que parece apropiado relacionar este estado con `< empty, 5 >`. Este último paso es un ejemplo de *regla de aplicación*.

Definimos así $h : \mathcal{S} \longrightarrow \mathcal{C}$ mediante $h(a) =$ `< rho, e >` si a se puede obtener a partir de `< empty, rho, e >` mediante cero o más aplicaciones de las reglas de análisis para la máquina de pila junto con `Valm` y `Locm2`. Nótese que h es una función precisamente porque no todas las reglas se pueden aplicar en esta definición. Además, h es parcial: solo está definida para los estados alcanzables, que constituyen una subestructura completa de $\mathcal{S}$ en la que h es total (recuérdese la discusión en relación con la definición 4.3).

De modo alternativo, "deshaciendo" los pasos dados por las reglas, h se puede definir mediante el siguiente conjunto de ecuaciones:

```
eq [Base] : h(< empty, rho, e>) = < rho, e > .
eq [Opm1] : h(< ST, rho, e . e' . op . C >) = h(< ST, rho, e op e' . C >) .
eq [Opm1] : h(< ST, rho, be . be' . bop . C >) =
            h(< ST, rho, be bop be' . C >) .
eq [Ifm1] : h(< ST, rho, be . if(e, e') . C >) =
            h(< ST, rho, If be Then e Else e' . C >) .
eq [Locm1] : h(< ST, rho, e. <x, e'> . C >) =
             h(< ST, rho, let x = e in e' . C >) .
eq [Notm1] : h(< ST, rho, be . not. C >) = h(< ST, rho, Not be . C >) .
eq [Eqm1] : h(< ST, rho, e . e' . equal . C >) =
            h(< ST, rho, Equal(e, e') . C >) .
eq [Locm2] : h(< ST, (x, v) . rho, e . pop . C >) =
```

```
              h(< v . ST, rho, < x, e > . C >) .
ceq [Valm] : h(< v . ST, rho, C >) = h(< ST, rho, v . C >) if not(enabled(C)) .
ceq [Valm] : h(< bv . ST, rho, C >) = h(< ST, rho, bv . C >)
              if not(enabled(C)) .
```

El predicado auxiliar `enabled` utilizado en `Valm` comprueba que no se puede aplicar ninguna otra ecuación, de forma análoga a como se hizo en la sección 4.5.

Lema 5.1 *Si* $h(<\ \texttt{ST, rho, e . C}\ >) = <\ \texttt{rho, e'}\ >$ *entonces existe una posición* p *en* `e'` *tal que* $\texttt{e'}|_p = \texttt{e}$ *y, en el caso de que* `e` *no sea un valor será una subexpresión que se puede reducir con las reglas de la semántica de computación dando* `e'` *en el siguiente paso.*

Demostración. Nótese que la relación de transición $\rightarrow_{\mathcal{S}}$ es determinista y que para cada estado $<\ \texttt{ST, rho, C}\ >$ hay una única manera de deshacer todos los pasos hasta alcanzar un estado de la forma $<\ \texttt{empty, rho, e}\ >$. Por lo tanto, para el propósito de la prueba podemos considerar que las ecuaciones que definen h están orientadas y proceder por inducción en el número de pasos usados para alcanzar $<\ \texttt{rho, e'}\ >$.

Cuando el número de pasos es 1 tenemos $h(<\ \texttt{empty, rho, e}\ >) \rightarrow <\ \texttt{rho, e}\ >$ y el resultado es trivial. Supongamos que n es mayor que 1; distinguimos casos de acuerdo con la ecuación (vista como regla) utilizada para el primer paso:

- Si la primera de las reglas etiquetadas con `Opm1` ha sido aplicada,

  $$h(<\ \texttt{ST, rho, e1 . e2 . op . C}\ >) \rightarrow h(<\ \texttt{ST, rho, e1 op e2 . C}\ >).$$

  Por la hipótesis de inducción existe una posición p tal que $\texttt{e'}|_p$ es `e1 op e2`, por lo que la posición buscada es $p.1$. El mismo razonamiento se aplica a la otra regla `Opm1` y a `Ifm1`, `Notm1` y `Eqm1`.

- Si se ha aplicado `Locm1`,

  $$h(<\ \texttt{ST, rho, e1 . < x, e2 > . C}\ >) \rightarrow h(<\ \texttt{ST, rho, let x = e1 in e2 . C}\ >).$$

  Por la hipótesis de inducción, $\texttt{e'}|_p$ es `let x = e1 in e2` y podemos tomar $p.1$ como la posición que buscamos.

- Para `Locm2`,

  $$\begin{aligned} h(<\ \texttt{ST, (x,v). rho, e . pop . C}\ >) &\rightarrow h(<\ \texttt{v . ST, rho, < x, e > . C}\ >) \\ &\rightarrow h(<\ \texttt{ST, rho, v . < x, e > . C}\ >) \\ &\rightarrow h(<\ \texttt{ST, rho, let x = v in e . C}\ >). \end{aligned}$$

  Por la hipótesis de inducción, $\texttt{e'}|_p$ es `let x = v in e` y podemos tomar $p.3$.

- Para `Valm` tenemos $h(<\ \texttt{v . ST, rho, e . C}\ >) \rightarrow h(<\ \texttt{ST, rho, v . e . C}\ >)$. Ahora las únicas reglas que se pueden aplicar al último término son `Opm1` y `Eqm1`; `Valm` no es una alternativa válida, pues daría lugar a tres expresiones consecutivas, lo que no es posible ya que no hay operadores ternarios. Supongamos que se usase la regla `Eqm1` (el razonamiento es análogo para las otras dos). Entonces C será de la forma `equal . C'` y $h(<\ \texttt{ST, rho, v . e . equal . C'}\ >) \rightarrow h(<\ \texttt{ST, rho, Equal(v,e) . C'}\ >)$. Ahora, por la hipótesis de inducción $\texttt{e'}|_p$ es `Equal(v,e)` y la posición buscada es $p.2$. □

Teorema 5.5 *La función $h : \mathcal{S} \longrightarrow C$ define una simulación tartamuda algebraica y recursiva de sistemas de transiciones.*

Demostración. Usaremos la caracterización finitaria de las simulaciones tartamudas dada en la definición 5.9. Como h es una función (parcial), solo es necesario definir una función $\mu : S \times C \longrightarrow \mathbb{N}$, y asignamos a $\mu(a, c)$ la longitud del camino más largo que comienza en a que solo utilice reglas de análisis.

Supongamos que $a \rightarrow_{\mathcal{S}} a'$ y que $h(a) = c$. Si a' se ha obtenido aplicando una regla de análisis, entonces $h(a') = c$ y $\mu(a', c) < \mu(a, c)$. En caso contrario debemos encontrar un elemento c' tal que $c \rightarrow_C c'$ y $h(a') = c'$; distinguimos casos dependiendo de la regla utilizada:

- `Opm2`. En este caso a es igual a `< v' . v . ST, rho, op . C >` y $h(a)$ es igual a h(`< ST, rho, v op v' . C >`) = `< rho, e >` donde, por el lema 5.1, existe una posición p en `e` tal que $\mathtt{e}|_p$ es `v op v'` y `v op v'` es una subexpresión de `e` que se puede reducir con las reglas de la semántica computacional en el siguiente paso. Podemos entonces tomar c' como `< rho, e[Ap(op,v,v')]`$_p$`>`. De modo similar haríamos para `Notm2`, `Eqm2` y `Ifm2`.

- `Varm`. Debe ser a igual a `< ST, rho, x . C >` y $h(a)$ a `< rho, e >` con $\mathtt{e}|_p$ = `x` una expresión en `e` que se puede reducir. De esta forma podemos hacer que c' sea `< rho, e[rho(x)]`$_p$`>`.

- `Pop`. En este caso a debe ser de la forma `< ST, (x,v) . rho, pop . C >`. La única ecuación que se puede aplicar a $h(a)$ es `Valm` y por lo tanto existe un valor `v'` tal que `ST` es `v' . ST'`. Aplicando ahora el resto de las ecuaciones resulta que $h(a)$ = h(`< ST', rho, let x = v in v' . C >`), que tiene que ser igual a `< rho, e >` con $\mathtt{e}|_p$ = `let x = v in v'` una subexpresión de `e` que se puede reducir. De modo que podemos hacer que c' sea `< rho, e[v']`$_p$`>`.

En consecuencia, las condiciones de la definición 5.9 se cumplen y, por el teorema 5.4, h es una simulación tartamuda de sistemas de transiciones. También está claro que las ecuaciones introducidas justo antes del lema 5.1 que definen h son Church-Rosser y terminantes con lo que h es una simulación tartamuda algebraica y recursiva de sistemas de transiciones. □

Es interesante destacar que *h no* es una bisimulación. En la semántica de computación, para una expresión `e op e'` podemos elegir entre evaluar `e` antes que `e'` o viceversa, mientras que la máquina de pila siempre evalúa `e` primero. Eso significa que, por ejemplo, la transición `< empty, (1 + 2) + (3 + 4) >` → `< empty, (1 + 2) + 7 >` no puede ser simulada por la máquina de pila.

La simulación h se puede elevar al nivel de las estructuras de Kripke. Para ello, basta con considerar como conjunto de proposiciones AP atómicas el conjunto de todos los valores posibles y extender los sistemas de transiciones $\mathcal{S}$ y C con $L_{\mathcal{S}}$(`< empty, rho, v >`) = $L_{\mathcal{S}}$(`< v, rho, empty >`) = {`v`}, L_C(`< rho,v >`) = {`v`} y tanto $L_{\mathcal{S}}(a)$ como $L_C(c)$ vacíos en otro caso. Entonces, por el teorema 5.2, para todas las expresiones `e` y entornos `rho`,

$$C, \texttt{< rho, e >} \models \mathbf{AFv} \quad \Longrightarrow \quad \mathcal{S}, \texttt{< empty, rho, e >} \models \mathbf{AFv}\,.$$

Esto es, $\mathcal{S}$ es una correcta implementación de C.

5.5.2 Un ejemplo de protocolo de comunicación

Aunque un mecanismo de comunicación no garantice la entrega en orden de mensajes, puede ser necesario generar este servicio usando la base dada a pesar de que no se pueda confiar en ella. En Meseguer (1993) se propone una posible solución al problema que vamos a adaptar aquí, retocando ligeramente la especificación para considerar un sistema con tan solo tres tipos de contenidos para transmitir. En ella, tanto el emisor como el receptor almacenan un contador de secuencia que utilizan para mantener la sincronización; el emisor libera los mensajes junto con dicho número (regla `send`) y no envía un nuevo mensaje hasta que recibe la confirmación de que el receptor lo ha recibido.

```
mod PROTOCOL is
  protecting NAT .
  protecting QID .

  sorts Object Msg Config .
  subsort Object Msg < Config .

  op null : -> Config .
  op __ : Config Config -> Config [assoc comm id: null] .

  sorts Elem List Contents .
  subsort Elem < Contents List .

  op empty : -> Contents .
  ops a b c : -> Elem .
  op nil : -> List .
  op _:_ : List List -> List [assoc id: nil] .

  op to:_(_,_) : Qid Elem Nat -> Msg .
  op to:_ack_ : Qid Nat -> Msg .

  op <_: Sender | rec:_, sendq:_ , sendbuff:_, sendcnt:_ > :
     Qid Qid List Contents Nat -> Object .

  --- rec es el receptor, sendq es la cola de salida, sendbuff es
  --- o bien vacio o el dato actual, sendcnt es el numero de secuencia
  --- del emisor

  op <_: Receiver | sender:_, recq:_, reccnt:_ > :
     Qid Qid List Nat -> Object .

  --- sender es el emisor, recq es la cola de entrada,
  --- reccnt es el numero de secuencia del receptor


  vars S R : Qid .      vars M N : Nat .
  var E : Elem .        var L : List .
  var C : Contents .

  --- reglas para el emisor
```

```
  rl [produce-a] :
     < S : Sender | rec: R, sendq: L, sendbuff: empty, sendcnt: N > =>
     < S : Sender | rec: R, sendq: L : a, sendbuff: a, sendcnt: N + 1 > .
  rl [produce-b] :
     < S : Sender | rec: R, sendq: L, sendbuff: empty, sendcnt: N > =>
     < S : Sender | rec: R, sendq: L : b, sendbuff: b, sendcnt: N + 1 > .
  rl [produce-c] :
     < S : Sender | rec: R, sendq: L, sendbuff: empty, sendcnt: N > =>
     < S : Sender | rec: R, sendq: L : c, sendbuff: c, sendcnt: N + 1 > .
  rl [send] : < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N >
           => < S : Sender | rec: R, sendq: L, sendbuff: E, sendcnt: N >
              (to: R (E,N)) .
  rl [rec-ack] :
     < S : Sender | rec: R, sendq: L, sendbuff: C, sendcnt: N >
     (to: S ack M) =>
     < S : Sender | rec: R, sendq: L,
                    sendbuff: (if N == M then empty else C fi),
                    sendcnt: N > .

  --- reglas para el receptor

  rl [receive] :
     < R : Receiver | sender: S, recq: L, reccnt: M > (to: R (E,N)) =>
     (if N == M + 1 then
        < R : Receiver | sender: S, recq: L : E, reccnt: M + 1 >
      else
        < R : Receiver | sender: S, recq: L, reccnt: M >
      fi)
     (to: S ack N) .
endm
```

Bajo hipótesis razonables de *justicia* (concretamente, que el receptor no estará indefinidamente sin leer un mensaje disponible), estas definiciones generan un mecanismo fiable de comunicación en orden a partir de uno no fiable. Los modos de fallo del canal de comunicación se pueden modelar explícitamente como sigue.

```
mod PROTOCOL-FAULTY is
  including PROTOCOL .

  op <_: Destroyer | sender:_, rec:_, cnt:_, cnt':_, rate:_ > :
     Qid Qid Qid Nat Nat Nat -> Object .

  var M : Msg .       vars K N N' : Nat .
  var E : Elem .      vars S R D : Qid .

  rl [destroy1] :
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N'), rate: K >
     (to: R (E,N)) =>
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N', rate: K > .
  rl [destroy2] :
     < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N'), rate: K >
```

```
    (to: R ack N) =>
    < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N', rate: K > .
  rl [limited-injury] :
    < D : Destroyer | sender: S, rec: R, cnt: N, cnt': 0, rate: K > =>
    < D : Destroyer | sender: S, rec: R, cnt: s(N), cnt': K, rate: K > .
endm
```

Los mensajes pueden ser destruidos por objetos de la clase `Destroyer`. El primer contador representa el número que identifica los mensajes que un objeto determinado puede destruir y el segundo es la cantidad restante de mensajes con ese número que todavía se le permite eliminar. El atributo `rate` se utiliza para reiniciar el valor de `cnt'` una vez que alcance el valor cero.

Para comprobar si los mensajes se entregan en el orden correcto definimos un predicado de estado `prefix(S,R)` que se cumple para un emisor `S` y un receptor `R` en comunicación si la cola asociada a `R` es un prefijo de aquella asociada a `S`. Tanto en `PROTOCOL` como en `PROTOCOL-FAULTY` se haría por medio de:

```
op prefix : Qid Qid -> Prop .

var CO : Config .

eq < S : Sender | rec: R, sendq: L1 : L2, sendbuff: C, sendcnt: N >
   < R : Receiver | sender: S, recq: L1, reccnt: M >
   CO |= prefix(S, R) = true .
```

El nuevo sistema satisface las mismas condiciones de corrección que `PROTOCOL`, con independencia de que los mensajes sean destruidos o lleguen desordenados. En particular, el estado inicial

```
eq init = < 'A : Sender | rec: 'B, sendq: nil, sendbuff: empty, sendcnt: 0 >
          < 'B : Receiver | sender: 'A, recq: nil, reccnt: 0 > .
```

debería satisfacer la fórmula **AG** `prefix('A,'B)`. Para demostrarlo definimos una simulación tartamuda

$$H : \mathcal{K}(\texttt{PROTOCOL-FAULTY}, \texttt{Config})_{\Pi} \longrightarrow \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_{\Pi}\,,$$

donde Π únicamente contiene el predicado `prefix`. Dadas las configuraciones (estados) a en `PROTOCOL-FAULTY` y b en `PROTOCOL`, tendremos aHb si y solo si:

- b se obtiene a partir de a eliminando todos los objetos de la clase `Destroyer`, o
- existe a' tal que $a'Hb$ y a se puede obtener a partir de a' aplicando solo las reglas que pertenecen a `PROTOCOL-FAULTY`.

Veamos que podemos definir H mediante reescritura en una teoría de reescritura admisible que extienda `PROTOCOL` y `PROTOCOL-FAULTY`. En esta especificación, las familias de los estados han sido renombrados como `Config1` y `Config2`, y `removeD` y `messages` son funciones auxiliares que, dada una configuración, eliminan todos los objetos de la clase `Destroyer` y devuelven todos los mensajes en ella, respectivamente.

```
op H : Config1 Config2 -> Bool .

op undo-d1 : Qid Elem Nat -> Msg .
op undo-d2 : Qid Nat -> Msg .
op undo-injury : -> Msg .

rl [destroy1-inv] :
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N' > undo-d1(R,E,N) =>
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N') > (to: R (E,N)) .
rl [destroy2-inv] :
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': N' > undo-d2(R,N) =>
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': s(N') > (to: R ack N) .
rl [limited-injury-inv] :
   < D : Destroyer | sender: S, rec: R, cnt: s(N), cnt': K, rate: K >
                                                          undo-injury =>
   < D : Destroyer | sender: S, rec: R, cnt: N, cnt': 0 > .

crl H(C, C') => true if removeD(C) = C' .
crl H(C, C') => true if M (to: R (E,N)) := messages(C') /\
                        (to: R (E,N)) in messages(C) = false /\
                        C undo-d1(R,E,N) => C'' /\ H(C'', C') => true .
crl H(C, C') => true if M (to: R ack N) := messages(C') /\
                        (to: R ack N) in messages(C) = false /\
                        C undo-d2(R,E) => C'' /\ H(C'', C') => true .
crl H(C, C') => true if C undo-injury => C'' /\ H(C'', C') => true .
```

Teorema 5.6 $H : \mathcal{K}(\texttt{PROTOCOL-FAULTY}, \texttt{Config})_\Pi \longrightarrow \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_\Pi$ *es una simulación tartamuda algebraica r.e.*

Demostración. H así definida preserva claramente las proposiciones atómicas porque los contenidos de las colas del emisor y del receptor, `sendq` y `recq`, no cambian. Sea R_1 el conjunto de reglas en `PROTOCOL` y sea R_2 el conjunto de aquellas que se han añadido en `PROTOCOL-FAULTY`, y definamos $\mu(a, b)$como la longitud de la secuencia de reescritura más larga que comienza en a que usa las reglas de R_2. Nótese que μ está bien definida, ya que R_2 es terminante. Si aHb y $a \to^1_{R_1} a'$ entonces, como la clase `Destroyer` no juega ningún papel en R_1, se tiene que $b \to^1_{R_1} b'$ con $a'Hb'$. Y si $a \to^1_{R_2} a'$, por definición de H tendremos $a'Hb$ y $\mu(a', b) < \mu(a, b)$. Gracias a la regla `send` no hay bloqueos en el sistema y por lo tanto estas dos alternativas cubren todas las posibilidades. Así, por el teorema 5.4 H es una Π-simulación tartamuda. Y puesto que las reglas que definen H en la lógica de reescritura son admisibles, H es una simulación tartamuda algebraica r.e. □

Por el teorema 5.2, la existencia de H muestra que si **AG** `prefix('A, 'B)` se cumple en `PROTOCOL` entonces también debe cumplirse en `PROTOCOL-FAULTY`. Pero todavía no hemos demostrado que la propiedad se cumpla en `PROTOCOL`. Para verlo, vamos a definir ahora una abstracción ecuacional finita

$$G : \mathcal{K}(\texttt{PROTOCOL}, \texttt{Config})_\Pi \longrightarrow \mathcal{K}(\texttt{ABS-PROTOCOL}, \texttt{Config})_\Pi$$

y a comprobar utilizando el comprobador de modelos que la fórmula es cierta en ABS-PROTOCOL; componiendo G con H esto también demuestra que la misma propiedad es cierta en PROTOCOL-FAULTY.

Este ejemplo es similar al visto en la sección 4.6.2. Como allí, la infinitud procede tanto de los contadores como del medio de comunicación; por lo tanto la abstracción será también muy parecida. Sin embargo, a diferencia de lo que sucedía entonces el módulo PROTOCOL no es Config-encapsulado, por lo que para poder aplicarle una abstracción ecuacional tenemos que modificarlo ligeramente según el enunciado del lema 4.3, introduciendo un nuevo operador

```
sort EConfig .
op {_} : Config -> EConfig .
```

y alterando ligeramente la ecuación de `prefix` que quedaría como

```
eq { < S : Sender | rec: R, sendq: L1 : L2, sendbuff: C, sendcnt: N >
     < R : Receiver | sender: S, recq: L1, reccnt: M >
     CO } |= prefix(S, R) = true .
```

Una vez hecho esto, los pasos vistos en el ejemplo de la sección 4.6.2 se aplican casi sin ningún cambio, produciéndosesde nuevo un problema de coherencia con la misma solución que allí. En definitiva, el módulo con la abstracción ecuacional al que se puede aplicar el comprobador de modelos de Maude queda:

```
mod ABS-PROTOCOL is
  protecting PROTOCOL .

  sort EConfig .

  op `{_`} : Config -> EConfig .

  var CO : Config .      vars M N : Nat .
  vars L1 L2 : List .    vars R S : Qid .
  vars E E1 E2 : Elem .
  var C : Contents .

  eq { < S : Sender | rec: R, sendq: E : L1, sendbuff: C, sendcnt: N >
       < R : Receiver | sender: S, recq: E : L2, reccnt: M >
       CO } =
     { < S : Sender | rec: R, sendq: L1, sendbuff: C, sendcnt: N >
       < R : Receiver | sender: S, recq: L2, reccnt: M >
       CO } .

  ceq { < S : Sender | rec: R, sendq: nil, sendbuff: empty, sendcnt: N >
        < R : Receiver | sender: S, recq: nil, reccnt: N > } =
      { < S : Sender | rec: R, sendq: nil, sendbuff: empty, sendcnt: 0 >
        < R : Receiver | sender: S, recq: nil, reccnt: 0 > }
      if N =/= 0 .
```

```
  ceq { < S : Sender | rec: R, sendq: E, sendbuff: E, sendcnt: s(N) >
        < R : Receiver | sender: S, recq: nil, reccnt: N > } =
      { < S : Sender | rec: R, sendq: E, sendbuff: E, sendcnt: 1 >
        < R : Receiver | sender: S, recq: nil, reccnt: 0 > }
      if N =/= 0 .

  eq { < S : Sender | rec: R, sendq: L1 : E : L2, sendbuff: C, sendcnt: s(M) >
       < R : Receiver | sender: S, recq: L1, reccnt: M >
       (to: R (E,s(M))) CO } =
     { < S : Sender | rec: R, sendq: L1 : E : L2, sendbuff: C, sendcnt: s(M) >
       < R : Receiver | sender: S, recq: L1 : E, reccnt: s(M) >
       (to: S ack s(M)) CO } .

  eq { < S : Sender | rec: R, sendq: L1, sendbuff: C, sendcnt: N >
       < R : Receiver | sender: S, recq: L2, reccnt: M >
       (to: S ack N) CO } =
     { < S : Sender | rec: R, sendq: L1, sendbuff: empty, sendcnt: N >
       < R : Receiver | sender: S, recq: L2, reccnt: M >
       CO } .

  --- para la coherencia
  rl [idle] :
     { < S : Sender | rec: R, sendq: L1, sendbuff: empty, sendcnt: N >
       < R : Receiver | sender: S, recq: L2, reccnt: s(M) >
       CO } =>
     { < S : Sender | rec: R, sendq: L1, sendbuff: empty, sendcnt: N >
       < R : Receiver | sender: S, recq: L2, reccnt: s(M) >
       CO } .
endm
```

Si `CHECK` es el módulo que contiene tanto `ABS-PROTOCOL` como la especificación del predicado de estado `prefix`, la propiedad se comprueba mediante:

```
Maude> red in CHECK : modelCheck(init, [] prefix('S,'R))  .
reduce in CHECK : modelCheck(init, []prefix('S, 'R)) .
result Bool: true
```

5.5.3 Una máquina canalizada simple

Consideramos ahora un ejemplo adaptado de Manolios (2001) sobre la corrección de una máquina canalizada (pipelined machine, en inglés).

La especificación utilizada para demostrar la corrección es una *arquitectura de conjunto de instrucciones* o ISA, por sus siglas en inglés: Instruction Set Architecture. Un estado ISA es una terna que consiste en un contador de programa, un archivo de registros y una memoria en la que (solo) se almacenan instrucciones. Las instrucciones consisten en un código de operación, el registro destino al que se aplica la operación y dos registros origen; hay códigos de operación para la suma, la resta y una operación "hacer nada" llamada *noop*. En cada paso, la máquina ejecuta la instrucción a la que apunta el contador de programa y actualiza de manera apropiada dicho contador y el archivo de registros.

Podemos representar una máquina ISA en la lógica de reescritura mediante una teoría $\mathcal{R}_{\text{ISA}}$ que extiende conservadoramente los números naturales utilizados para representar los registros y sus valores. Para representar todos los elementos de la máquina necesitamos los siguientes operadores:

```
op {_,_,_} : ProgramCounter RegFile Memory -> StateISA .
op inst : OpCode Nat Nat Nat -> Instruction .
ops add sub noop : -> OpCode .
op reg : Nat Nat -> Register .
op _;_ : RegFile RegFile -> RegFile [assoc comm] .
op update : RegFile Instruction -> RegFile .
op cell : Nat Instruction -> MemCell .
op _:_ Memory Memory -> Memory .
op applyOp : OpCode Nat Nat -> Nat .
op getValue : RegisterFile Nat -> Nat .
```

Entonces, su comportamiento está gobernado por la regla

```
rl { PC, RF, cell(PC, I) : M } => { PC + 1, update(RF, I), cell(PC, I) : M } .
```

y las ecuaciones

```
eq update(reg(R1, V1) ; RF, inst(OC, R1, R2, R3)) =
   reg(R1, applyOP(OC, getValue(reg(R1, V1) ; RF, R2),
                       getValue(reg(R1, V1) ; RF, R3))) ; RF .
eq getValue(reg(R, V) ; RF, R) = V .
eq applyOp(+, N1, N2) = N1 + N2 .
eq applyOp(*, N1, N2) = N1 * N2 .
```

Siguiendo a Manolios (2001), nos centramos en la relación de transición, olvidándonos de las proposiciones atómicas.

La máquina ISA está implementada por una *máquina con microarquitectura* (MA), una máquina canalizada con tres estapas. Un estado de una máquina MA es una 5-tupla consistente en un contador de programa, un archivo de registros, una memoria y dos almacenes auxiliares. Durante la etapa de recogida, la instrucción a la que apunta el contador del programa se almacena en el primer almacén. Durante la etapa de iniciación, la instrucción en el primer almacén se pasa al segundo junto con los valores en los registros de origen a los que hace referencia aquella. En la etapa de escritura, la instrucción en el segundo almacén se ejecuta y el archivo de registro se actualiza.

De nuevo, la máquina MA se puede representar en la lógica de reescritura como una teoría $\mathcal{R}_{\text{MA}}$ que extiende conservadoramente la teoría de los números naturales. Los operadores necesarios incluyen los introducidos más arriba exceptuando el constructor `{_,_,_}` y añadiendo los siguientes:

```
subsort Instruction < Latch1 .

op {_,_,_,_,_} : ProgramCounter RegisterFile Memory Latch1 Latch2 -> StateMA .
```

```
op empty1 : -> Latch1 .
op empty2 : -> Latch2 .
op latch : OpCode Nat Nat Nat -> Latch2 .
op nextPC : StateMA -> ProgramCounter .
op nextRF : StateMA -> RegisterFile .
op nextM : StateMA -> Memory .
op nextL1 : StateMA -> Latch1 .
op nextL2 : StateMA -> Latch2 .
op stalled : Latch1 Latch2 -> Bool .
```

El comportamiento de la máquina queda definido por la regla

```
rl S => { nextPC(S), nextRF(S), nextM(S), nextL1(S), nextL2(S) } .
```

donde `S` es una variable de tipo sort `StateMA`. Si no ocurre un atasco el contador del programa es incrementado; en caso contrario se mantiene fijo. Un atasco ocurre cuando ambos almacenes son no vacíos y el registro de destino del segundo es uno de los registros de origen del primero.

```
eq nextPC({ PC, RF, M, L1, L2 }) = PC if stalled(L1, L2) = true .
eq nextPC({ PC, RF, M, L1, L2 }) = PC + 1 if stalled(L1, L2) = false .
eq stalled(empty1, L2) = false .
eq stalled(L1, empty2) = false .
eq stalled(inst(OC, R1, R2, R3), latch(OC', R, N1, N2)) =
   (R == R2) or (R == R3) .
```

El contenido de la memoria, esto es, las instrucciones a ejecutar, se mantiene fijo a lo largo de toda la ejecución,

```
eq nextM({ PC, RF, M, L1, L2 }) = M .
```

y el archivo de registros es actualizado con el valor resultante de ejecutar la instrucción en el segundo almacén, siempre que este no sea vacío:

```
eq nextRF({ PC, RF, M, L1, empty }) = RF .
eq nextRF({ PC, reg(R, V) ; RF, M, L1, latch(OC, R, N1, N2) }) =
   reg(R, applyOp(OC, N1, N2)) ; RF .
```

Si no se produce un atasco, el primer almacén se actualiza leyendo de la memoria la instrucción a la que apunta el contador de programa.

```
ceq nextL1({ PC, RF, cell(PC, I) : M, L1, L2 }) = I
    if stalled(L1, L2) = false .
eq nextL1({ PC, RF, M, L1, L2 }) = L1 if stalled(L1, L2) = true .
```

De forma similar, si no se produce un atasco el segundo almacén se actualiza con la instrucción en el primero, junto con los valores en los registros de origen.

```
ceq nextL2({ PC, RF, M, empty1, L2 }) = empty2 if stalled(L1, L2) = false .
ceq nextL2({ PC, RF, M, inst(OC, R1, R2, R3), L2 }) =
    inst(OC, R1, getValue(RF, R2), getValue(RF, R3))
    if stalled(L1, L2) = false .
eq nextL2({ PC, RF, M, L1, L2 }) = empty2 if stalled(L1, L2) = true .
```

Acabamos de dar las especificaciones en lógica de reescritura para las máquinas ISA y MA. Ahora queremos relacionarlas por medio de un morfismo tartamudo algebraico y recursivo de sistemas de transiciones $(\mathcal{R}_{\text{MA}}, [\texttt{State}]_{\text{MA}}) \longrightarrow (\mathcal{R}_{\text{ISA}}, [\texttt{State}]_{\text{ISA}})$. Nótese el sentido de la flecha, desde la implementación a la especificación.

Las instrucciones en la máquina ISA se ejecutan de manera inmediata, mientras que en MA pasan por tres etapas distintas; por lo tanto, la simulación necesariamente será tartamuda. Dado un estado de la máquina MA, para obtener un estado ISA solo tenemos que olvidar la información en los almacenes. Nótese, sin embargo, que el contador de programa en la máquina MA apunta a la siguiente instrucción que hay que ejecutar, de manera que ahora que vamos a eliminar aquellas instrucciones que ya han sido trasladadas a los almacenes tenemos que decrementar convenientemente el contador.

La simulación se puede especificar entonces sobre la unión disjunta de las teorías de reescritura $\mathcal{R}_{\text{ISA}}$ y $\mathcal{R}_{\text{MA}}$. El contador de programa es actualizado de forma apropiada por el operador

```
op commit : StateRA -> ProgramCounter .
```

definido por las ecuaciones

```
eq commit({ PC, RF, M, empty1, empty2}) = PC .
eq commit({ PC, RF, M, inst(OC, R1, R2, R3), empty2}) = PC - 1 .
eq commit({ PC, RF, M, empty1, latch(OC, R, N1, N2) }) = PC - 1 .
eq commit({ PC, RF, M, inst(OC, R1, R2, R3), latch(OC, R, N1, N2) }) = PC - 2 .
```

Finalmente, el morfismo se define por medio de un operador

```
op sim : StateMA -> StateISA .
```

junto con la ecuación

```
eq sim({ PC, RF, M, L1, L2 }) = { commit({ PC, RF, M, L1, L2 }, RF, M) } .
```

Que estas definiciones dan lugar a un morfismo tartamudo algebraico y recursivo de sistemas de transiciones se demuestra en Manolios (2001, sección 12.3).

Teorema 5.7 *El operador* `sim` *especifica un morfismo tartamudo algebraico y recursivo de sistemas de transiciones, sim* : $\mathcal{T}(\mathcal{R}_{\text{MA}})_{\text{State}_{\text{MA}}} \longrightarrow \mathcal{T}(\mathcal{R}_{\text{ISA}})_{\text{State}_{\text{ISA}}}$.

5.6 Morfismos de simulación teoroidales

Al comienzo de la sección 5.4 señalamos que las abstracciones ecuacionales del capítulo 4 podían ser generalizadas bien considerando interpretaciones de teorías o, de forma más general, funciones definidas ecuacionalmente o relaciones definidas por reescritura. Las secciones anteriores han estado dedicadas a presentar los casos más generales. Sin embargo, no siempre es necesario usar esa mayor generalidad: hay muchos ejemplos interesantes que pueden ser explicados simplemente por medio de interpretaciones de teorías; a ellos dedicamos ahora nuestra atención.

5.6.1 Morfismos generalizados de signaturas

Lo primero que tenemos que hacer es precisar el significado de la *interpretación de teorías*. La idea es utilizar los conceptos estándar de morfismos de signaturas y de teorías. Sin embargo, tal y como veremos en algunos de los ejemplos, la definición habitual de morfismo de signaturas no es en ocasiones suficientemente expresiva. Por esta razón introducimos la siguiente generalización del concepto de morfismo de signaturas, en el que una familia o un operador pueden ser *eliminados*.

Definición 5.17 *Dadas dos signaturas,* $\Sigma = (K, \Sigma, S)$ *y* $\Sigma' = (K', \Sigma', S')$*, en la lógica de pertenencia, un* morfismo generalizado de signaturas $H : \Sigma \longrightarrow \Sigma'$ *viene dado por:*

- *funciones parciales* $H : K \longrightarrow K'$ *y* $H : S \longrightarrow S'$ *tales que, para todos los tipos* $s \in \Sigma$*, si* $H(s)$ *está definido entonces también lo está* $H([s])$ *y se tiene* $H([s]) = [H(s)]$*.*
- *una función parcial* H *que asigna, a cada* $f \in \Sigma_{k_1 \ldots k_n, k}$ *tal que* $H(k)$ *está definido, un* Σ'*-término* $H(f)$ *de la familia* $H(k)$ *tal que* $vars(H(f)) \subseteq \{x_{i_1} : H(k_{i_1}), \ldots, x_{i_m} : H(k_{i_m})\}$*, donde* $k_{i_1}, \ldots, k_{i_m}$ *es la subsecuencia (posiblemente vacía) de* $k_1, \ldots, k_n$ *determinada por aquellos* k_i *tales que* $H(k_i)$ *está definido. Por otra parte, si* $H(k)$ *está indefinido también lo está* $H(f)$*.*

Todas las construcciones y resultados habituales sobre los morfismos de signaturas también se aplican a estos morfismos generalizados. Dados $H : \Sigma \longrightarrow \Sigma'$ y una Σ'-álgebra A, su *reducto* $U_H(A)$ sobre Σ está definido por:

- Para cada familia k, $U_H(A)_k = A_{H(k)}$ si $H(k)$ está definido; en caso contrario, $U_H(A)_k = \{*\}$.
- Para cada tipo s, $U_H(A)_s = A_{H(s)}$ si $H(s)$ está definido; en caso contrario, $U_H(A)_s = \{*\}$.
- Para cada operador $f : k_1 \ldots k_n \longrightarrow k$, si $k_{i_1}, \ldots, k_{i_m}$ es la subsecuencia de aquellas familias en $k_1, \ldots, k_n$ para las que H está definido,

  $$U_H(A)_f(a_1, \ldots, a_n) = A_{H(f)}(a_{i_1}, \ldots, a_{i_m})\,;$$

  en caso contrario

  $$U_H(A)_f(a_1, \ldots, a_n) = *\,.$$

Dados dos morfismos generalizados de signaturas $F : \Sigma \longrightarrow \Sigma'$ y $G : \Sigma' \longrightarrow \Sigma''$, su composición $G \circ F$ está definida para una familia k solo si tanto $F(k)$ como $G(F(k))$ están definidos, en cuyo caso se tiene $(G \circ F)(k) = G(F(k))$; lo mismo ocurre para cada tipo s y cada operador f.

Los morfismos generalizados de signaturas también se pueden extender homomórficamente a términos, pero hay que señalar que para todo término t en la familia k, si $H(k)$ no está definido entonces $H(t)$ tampoco lo está. Esta traducción se extiende de la manera esperada a sentencias, donde por convención $H(t = t') = H(t : s) = \top$ si H no está definido para la familia de t (que es la misma que la de t' y s). La noción "interpretación de teorías" buscada es capturada entonces por la siguiente definición.

Definición 5.18 *Dadas dos teorías en la lógica de pertenencia,* (Σ, E) *y* (Σ', E')*, un* morfismo generalizado de teorías *(resp. un* morfismo generalizado de teorías con semántica inicial*)* $H : (\Sigma, E) \longrightarrow (\Sigma', E')$ *es un morfismo generalizado de signaturas* $H : \Sigma \longrightarrow \Sigma'$ *tal que para cada* $\varphi \in E$ *se tiene* $E' \models H(\varphi)$ *(resp.* $T_{\Sigma'/E'} \models H(\varphi)$*).*

Nótese que, puesto que $T_{\Sigma'/E'} \models E'$, todo morfismo generalizado de teorías es a posteriori un morfismo generalizado de teorías con semántica inicial, pero el recíproco no es cierto. Por ejemplo, si (Σ, E) es la teoría con una familia *Natural*, un operador binario $+$ y la ecuación $(\forall \{x, y : Natural\})\ x + y = y + x$, (Σ', E') es la definición ecuacional habitual de la suma en la aritmética de Peano y H es la inclusión de signaturas obvia, tenemos que $T_{\Sigma'/E'} \models (\forall \{x, y : Natural\})\ x + y = y + x$, pero $E' \not\models (\forall \{x, y : Natural\})\ x + y = y + x$.

De nuevo los morfismos generalizados de teorías se pueden componer y, junto con las teorías en la lógica de pertenencia, dan lugar a la categoría $\mathbf{GTh}_{\mathrm{MEL}}$.

La nueva característica de los morfismos generalizados de signaturas, heredada por los morfismos generalizados de teorías, es que las familias y los operadores pueden ser eliminados. Esto podría haber sido "implementado" utilizando la noción estándar de morfismo de teorías de la siguiente manera alternativa:

Proposición 5.9 *Un morfismo generalizado de teorías* $H : T \longrightarrow T'$ *es simplemente un morfismo ordinario de teorías* $H : T \longrightarrow T' \oplus ONE$*, donde* $\oplus$ *denota la unión disjunta de teorías y ONE es la teoría con una única familia* [*One*] *y tipo One, una constante* $*$ *de esa familia y la ecuación* $(\forall \{x\})\ x = *$*.*

Demostración. El dejar una familia o tipo indefinido en un morfismo generalizado de signaturas corresponde respectivamente a llevarlo a [*One*] o a *One* en $T' \oplus ONE$, mientras que el dejar la imagen de un operador sin definir corresponde a aplicarlo al término $*$. □

Se trata pues de la construcción habitual para convertir las funciones parciales en totales por medio de un valor indefinido o $\perp$, que en este caso se representa por *.

Nótese que existe una equivalencia de categorías entre los modelos de T' y los de $T' \oplus ONE$ porque, aunque hemos introducido una nueva familia [*One*], todos sus elementos son colapsados por la ecuación $(\forall \{x\})\ x = *$ en la constante $*$ y no pueden desempeñar ningún papel distinguido.

Ejemplo. Un caso especial de morfismo generalizado de teorías lo constituyen las funciones de proyección desde n-tuplas en $(n-k)$-tuplas, con $0 \leq k \leq n$. Consideremos una teoría *3-TUPLA* para ternas con familias *3-Tupla*, *Elt@x*, *Elt@y*, *Elt@z*, un operador $\langle \_, \_, \_ \rangle : \mathit{Elt@x}\ \mathit{Elt@y}\ \mathit{Elt@z} \longrightarrow \mathit{3\text{-}Tuple}$ y operadores de proyección p_1, p_2 y p_3, con las ecuaciones obvias. De forma parecida, la teoría *2-TUPLA* tiene familias *2-Tuple*, *Elt@x*, *Elt@z*, un operador $\langle \_, \_ \rangle : \mathit{Elt@x}\ \mathit{Elt@z} \longrightarrow \mathit{2\text{-}Tuple}$, los correspondientes operadores de proyección p_1 y p_2, y las ecuaciones para emparejar. La proyección desde una terna a un par eliminando la segunda componente en el proceso se puede representar por medio del morfismo generalizado de teorías $H : \mathit{3\text{-}TUPLA} \longrightarrow \mathit{2\text{-}TUPLA}$ que lleva las familias *Elt@x* y *Elt@z* a sí mismas, *3-Tuple* a *2-Tuple* y el operador $\langle \_, \_, \_ \rangle$ al término $\langle x_1 : \mathit{Elt@x}, x_3 : \mathit{Elt@z} \rangle$; las imágenes de la familia *Elt@y* y el operador p_2 se dejan sin definir.

5.6.2 Morfismos teoroidales como morfismos de simulación

Ahora tenemos disponibles todos los ingredientes necesarios para definir una categoría **SRWThHom**$_{\models}$ en la que las morfismos tartamudos se especifiquen mediante interpretaciones de teorías.

Los objetos en **SRWThHom**$_{\models}$ son los mismos que los de **SRWTh**$_{\models}$, es decir, ternas $(\mathcal{R}, (\Sigma', E'), J)$ que satisfacen todos los requisitos dados en la página 124.

Una flecha

$$H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$$

en **SRWThHom**$_{\models}$ es un *morfismo generalizado de signaturas* $H : \Sigma_1 \cup \Pi_1 \longrightarrow \Sigma_2 \cup \Pi_2$ tal que:

1. $H \circ J_1 = J_2$ (de manera que *BOOL* es preservada y los estados en $\mathcal{R}_1$ son llevados a estados en $\mathcal{R}_2$).

2. $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ es un morfismo generalizado de teorías en lógica de pertenencia con semántica inicial, de manera que tenemos un único Σ_1-homomorfismo

$$\eta^H : T_{\Sigma_1/E_1} \longrightarrow U_H(T_{\Sigma_2/E_2}) : [t] \mapsto [H(t)]\,.$$

3. (Preservación de transiciones) $\eta^H_{J_1(\mathit{State})} : \mathcal{T}(\mathcal{R}_1)_{J_1(\mathit{State})} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{J_2(\mathit{State})}$, la componente correspondiente a la familia $J_1(\mathit{State})$ en η^H, que aplica $[t]$ en $[H(t)]$, es un morfismo tartamudo de sistemas de transiciones.

4. (Preservación de predicados) Para todo término $t \in T_{\Sigma_1, J_1(\mathit{State})}$ y predicado de estado $p(u_1, \ldots, u_n)$, $H(p(u_1, \ldots, u_n))$ es un predicado de estado y tenemos

$$E_2 \cup D_2 \vdash \Big(H(t) \models H(p(u_1, \ldots, u_n))\Big) = \mathit{true} \implies E_1 \cup D_1 \vdash \Big(t \models p(u_1, \ldots, u_n)\Big) = \mathit{true}\,.$$

Como H no puede aplicar un predicado de estado en una fórmula cualquiera, el problema comentado en la sección 5.1 no aparece aquí y podemos construir una subcategoría **SRWThHom**$^{\mathrm{str}}_{\models}$ de *morfismos estrictos* de manera análoga a como hemos definido la versión no estricta. La definición es exactamente la misma excepto por el punto (4), donde la implicación pasa a ser una equivalencia. De forma similar, para obtener una subcategoría

RWThHom$_{\models}$ de *morfismos no tartamudos* simplemente reemplazamos la condición (3) por el requisito de que para todo t, t' en $T_{\Sigma_1, J_1(\text{State})}$:

$$t \rightarrow^1_{\mathcal{R}_1, J_1(State)} t' \implies H(t) \rightarrow^1_{\mathcal{R}_2, J_2(State)} H(t')$$

Que H así constreñida da lugar a un morfismo de estructuras de Kripke se demuestra en la proposición 5.10 que veremos a continuación.

Definimos un funtor $\mathcal{K}$: **SRWThHom**$_{\models}$ $\longrightarrow$ **KSMap** como sigue:

- sobre los objetos, $\mathcal{K}(\mathcal{R}, (\Sigma', E \cup D), J) = \mathcal{K}(\mathcal{R}, J(State))_{\Pi}$;
- sobre las flechas, para $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ definimos $\mathcal{K}(H) = (H|_{\Pi_1}, \eta^H_{J_1(State)})$, donde $H|_{\Pi_1}$ es la restricción de H a los predicados de estado Π_1.

Proposición 5.10 *Con las definiciones anteriores,* $\mathcal{K}$: **SRWThHom**$_{\models}$ $\longrightarrow$ **KSMap** *es un funtor con restricciones* $\mathcal{K}$: **SRWThHom**$^{\text{str}}_{\models}$ $\longrightarrow$ **KSMap**$^{\text{str}}$ *y* $\mathcal{K}$: **RWThHom**$_{\models}$ $\longrightarrow$ **KMap**.

Demostración. $\mathcal{K}$ está bien definido sobre los objetos y es inmediato ver que preserva las identidades y la composición de flechas; lo único que necesitamos comprobar es que, para todo H, $\mathcal{K}(H)$ es en efecto un morfismo de estructuras de Kripke.

Sea entonces $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ una flecha en la categoría **SRWThHom**$_{\models}$. Por el punto (3), $\eta^H_{J_1(State)} : \mathcal{T}(\mathcal{R}_1)_{J_1(State)} \longrightarrow \mathcal{T}(\mathcal{R}_2)_{J_2(State)}$ es un morfismo tartamudo de sistemas de transiciones. Para demostrar que los predicados de estado se reflejan, sea $p(u_1, \ldots, u_n) \in L_{\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}|_{H|_{\Pi_1}}}([H(t)])$. Por definición del reducto de una estructura de Kripke $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}, [H(t)] \models H(p(u_1, \ldots, u_n))$ que, por definición de $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$ y la condición (4) en la definición de flecha en **SRWThHom**$_{\models}$, implica que $p(u_1, \ldots, u_n) \in L_{\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}}([t])$, como se pedía. Es claro que si H pertenece a **SRWThHom**$^{\text{str}}_{\models}$ el recíproco también es cierto y $\mathcal{K}(H)$ es un morfismo estricto.

Finalmente, para la segunda restricción mencionada en el enunciado de la proposición, sea $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ una flecha en **RWThHom**$_{\models}$. Tenemos que demostrar que $\mathcal{K}(H) = (H|_{\Pi_1}, \eta^H_{J_1(State)})$ es un morfismo de $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$ en $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$, esto es, que $\eta^H_{J_1(State)}$ es un Π_1-morfismo de $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$ al reducto $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}|_{H|_{\Pi_1}}$. Sea $[t] \rightarrow [t']$ una transición en $\mathcal{K}(\mathcal{R}_1, J_1(State))_{\Pi_1}$. Por la suposición sobre la ausencia de bloqueo (recuérdese la definición de los objetos en **SRWTh**$_{\models}$ en la sección 5.4.1), esto significa que $t_0 \rightarrow^1_{\mathcal{R}_1, J_1(State)} t'_0$ para algún $t_0 \in [t]$ y $t'_0 \in [t']$. Como H preserva reescrituras, $H(t_0) \rightarrow^1_{\mathcal{R}_2, J_2(State)} H(t'_0)$ y por lo tanto $[H(t)] \rightarrow [H(t')]$ en $\mathcal{K}(\mathcal{R}_2, J_2(State))_{\Pi_2}$. Que los predicados se reflejan se demuestra como en el párrafo anterior. □

Una consecuencia importante de este resultado y de los teoremas 5.1 y 5.3 es la siguiente:

Teorema 5.8 *Dado un morfismo* $H : (\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1) \longrightarrow (\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *en* $\mathbf{SRWThHom}_{\models}$, $\mathbf{SRWThHom}^{\mathrm{str}}_{\models}$ *o* $\mathbf{RWThHom}_{\models}$, *y una fórmula* φ *en* $\mathrm{ACTL}^*\backslash\{\neg, \mathbf{X}\}(\Pi_1)$, $\mathrm{ACTL}^*(\Pi_1) \setminus \mathbf{X}$ *o* $\mathrm{ACTL}^*\backslash\neg(\Pi_1)$ *respectivamente, si* $H(\varphi)$ *se cumple en* $\mathcal{K}(\mathcal{R}_2, (\Sigma'_2, E_2 \cup D_2), J_2)$ *entonces* φ *también se cumple en* $\mathcal{K}(\mathcal{R}_1, (\Sigma'_1, E_1 \cup D_1), J_1)$.

Nótese que existe un funtor de olvido $\mathbf{SRWThHom}_{\models} \hookrightarrow \mathbf{SRWTh}_{\models}$ obvio. De la misma forma se puede definir una categoría $\mathbf{RecSRWThHom}_{\models}$ con morfismos recursivos y funtor de inclusión $\mathbf{RecSRWThHom}_{\models} \hookrightarrow \mathbf{RecSRWTh}_{\models}$.

Con esto, la traducción de las estructuras de Kripke al marco de la lógica de reescritura se puede representar gráficamente por medio del siguiente diagrama conmutativo. En él, las flechas horizontales entre categorías asociadas a estructuras de Kripke son inclusiones y las que llegan a aquellas asociadas con sistemas de transiciones son funtores de olvido.

$$\begin{array}{ccccccc}
\mathbf{SRWThHom}_{\models} & \longrightarrow & \mathbf{SRWTh}_{\models} & \longrightarrow & \mathbf{SRelRWTh}_{\models} & \longrightarrow & \mathbf{SRWTh} \\
\downarrow{\scriptstyle \mathcal{K}} & & \downarrow{\scriptstyle \mathcal{K}} & & \downarrow{\scriptstyle \mathcal{K}} & & \downarrow{\scriptstyle \mathcal{T}} \\
\mathbf{KSMap} & \longrightarrow & \mathbf{KSMap} & \longrightarrow & \mathbf{KSSim} & \longrightarrow & \mathbf{STSys}
\end{array}$$

Naturalmente, existe un diagrama análogo sobre las correspondientes estructuras recursivas.

5.7 Algunos ejemplos más

En esta sección presentamos algunos otros ejemplos más de simulaciones que ilustran la definición de morfismo teoroidal.

5.7.1 Un ejemplo sencillo

Consideremos las dos especificaciones siguientes en las que se representan "sopas" de relojes con y sin etiquetas, respectivamente.

```
mod CLOCK is
  protecting NAT .
  protecting QID .
  sorts Object Configuration Prop .
  subsort Object < Configuration .
  op <_ : Clock | time: _> : Qid Nat -> Object .
  op null : -> Configuration .
  op __ : Configuration Configuration -> Configuration [assoc comm id:null].
  op time? : Nat -> Prop .
  op _|=_ : Configuration Prop -> Bool .
  var N : Nat . var A : Qid . var C : Configuration .

  eq (< A : Clock | time : N > C |= time?(N)) = true .
  rl [tick] : < A : Clock | time : N > => < A : Clock | time : s(N) > .
endm
```

```
mod ABSTRACT-CLOCK is
  protecting NAT .
  sort Conf Prop .
  subsort Nat < Conf .
  op null : -> Conf .
  op __: Conf Conf -> Conf [assoc comm id: null] .
  op time? : Nat -> Prop [frozen] .
  op _|=_ : Conf Prop -> Bool .
  var N : Nat . var C : Config .

  eq (N |= time?(N)) = true .
  rl [tick-0] : 0 => s(0) .
  rl [tick-s] : s(N) => s(s(N)) .
endm
```

Nos gustaría relacionarlas por medio de una simulación. La idea es que tan solo estamos interesados en la hora pero no en los nombres de los relojes, por lo que podemos borrar estos de manera segura. Para ello definimos un morfismo generalizado de signaturas $H : \Sigma_{\texttt{CLOCK}} \longrightarrow \Sigma_{\texttt{ABSTRACT-CLOCK}}$ que deja los tipos y los operadores en `NAT` fijos, lleva el tipo `Object` a `Nat`, el tipo `Configuration` a `Conf` y los operadores `null` y de unión de multiconjuntos `Configuration` a los de `Conf`; el tipo `Qid` es eliminado. Finalmente, el operador

```
  op <_: Clock | time : _> : Qid Nat -> Object .
```

se lleva al término x_2 : `Nat` y `time?` se lleva a sí mismo.

Es claro que si $t \rightarrow^1_{\texttt{CLOCK,Configuration}} t'$ entonces $H(t) \rightarrow^1_{\texttt{ABSTRACT-CLOCK,Conf}} H(t')$; análogamente, los predicados de estado son reflejados. Es así que H da lugar a un morfismo en $\mathbf{RWThHom}_{\models}$, de manera que las propiedades temporales de `CLOCK`, como por ejemplo **AF**`time?(10)`, se pueden estudiar en `ABSTRACT-CLOCK`.

5.7.2 Abstracción de predicados

Una instancia particular de la metodología de abstracción es la *abstracción de predicados*, propuesta por primera vez en Graf y Saïdi (1997) y desarrollada en numerosos trabajos como los de Colón y Uribe (1998), Saïdi y Shankar (1999) o Das et al. (1999). En este método el dominio abstracto es un álgebra booleana sobre un conjunto de predicados y la función de abstracción, que típicamente forma parte de una conexión de Galois, es construida simbólicamente como la conjunción de todas las expresiones que satisfacen una cierta condición que se suele probar utilizando un demostrador de teoremas. A continuación mostramos cómo la abstracción de predicados puede ser presentada como un ejemplo más de nuestra noción de simulación algebraica.

Vamos a centrarnos en primer lugar en la relación de transición. Dado un sistema computacional, un conjunto $\phi_1, \ldots, \phi_n$ de predicados sobre los estados determina una función abstracta que aplica un estado S en la tupla booleana $\langle \phi_1(S), \ldots, \phi_n(S) \rangle$. Supongamos que las transiciones del sistema vienen especificadas por una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$ cuya familia de estados es *State*. Entonces, si $\mathcal{R}$ es *State*-encapsulada con

constructor $st : k_1 \dots k_m \longrightarrow State$ (recordemos de la sección 4.4 que esto significa que la familia *State* solo es usada en el operador *st*, y solo como su coaridad), la abstracción de predicados anterior puede representarse en la lógica de reescritura por medio de una teoría de reescritura $\mathcal{R}_A = (\Sigma_A, E_A, R_A)$ donde:

- Σ_A contiene Σ y la signatura de *BOOL*, junto con una nueva familia *BState*, un nuevo operador $bst : Bool^n \longrightarrow BState$ y, para cada predicado ϕ_i, $i = 1, \dots, n$, un operador $p_i : State \longrightarrow Bool$ que lo representa. Tenemos entonces un morfismo de signaturas $H : \Sigma \longrightarrow \Sigma_A$ que lleva *State* a *BState*, el constructor *st* al término

  $$bst(p_1(st(x_1, \dots, x_m)), \dots, p_n(st(x_1, \dots, x_m)))$$

  y es la identidad sobre los demás operadores.

- E_A contiene $H(E)$ y las ecuaciones de *BOOL*, junto con ecuaciones para $p_1, \dots, p_n$ que especifican los predicados $\phi_1, \dots, \phi_n$.

- $R_A = H(R)$.

Por construcción, $H : (\Sigma, E) \longrightarrow (\Sigma_A, E_A)$ es un morfismo de teorías tal que $t \rightarrow^1_{\mathcal{R},State} t'$ implica $H(t) \rightarrow^1_{\mathcal{R}_A,BState} H(t')$, preservando así la relación de transición.

Podemos ahora dedicar nuestra atención a la preservación de las propiedades. Gráficamente, la relación entre las distintas teorías consideradas se ilustra en el siguiente diagrama,

$$\begin{array}{ccc} (\Sigma, E) & \hookrightarrow & (\Sigma', E \cup D) \\ {\scriptstyle H}\downarrow & & \downarrow \\ (\Sigma_A, E_A) & \hookrightarrow & (\Sigma'_A, E_A \cup D_A) \end{array}$$

donde $(\Sigma', E \cup D)$ es la teoría ecuacional que especifica las propiedades del sistema dado y $(\Sigma'_A, E_A \cup D_A)$ es la teoría que tenemos que asociar a $\mathcal{R}_A$ para definir sus proposiciones atómicas.

La sintaxis para los predicados de estado q (que asumimos son constantes) en el sistema original viene dada por una subsignatura Π de Σ'. Es habitual que para cada uno de estos q uno de los predicados ϕ_i en la base que define la abstracción tenga el significado "el estado S satisface q". Sean $q_1, \dots, q_k$ los predicados de estado en Π. Suponemos que $k \leq n$ y los predicados están ordenados de forma que cada q_j, $1 \leq j \leq k$, corresponde al predicado ϕ_j en la base de la abstracción (nótese que podemos tener $n > k$, en cuyo caso los predicados $\phi_{k+1}, \dots, \phi_n$ no tendrán contrapartida en Π). Bajo estas hipótesis tenemos entonces que para un ϕ_j con su correspondiente q_j en Π su especificación en E_A a través de $p_j(S)$ es esencialmente la misma (módulo renombramiento) que la de $S \models q_j$ en D, de manera que $E \cup D \vdash (S \models q_j) = true \iff E_A \vdash p_j(S) = true$. Para la abstracción utilizamos entonces el mismo conjunto de predicados de estado Π que se especifican en la teoría extendida $(\Sigma_A, E_A) \subseteq (\Sigma'_A, E_A \cup D_A)$, de forma que $\Sigma'_A = \Sigma_A \cup \Sigma'$ y D_A contiene, para cada q_j en Π asociado a ϕ_j, la ecuación

$$(\forall \{x_1, \dots, x_n\})\, (bst(x_1, \dots, x_i, \dots, x_n) \models q_j) = x_j\,.$$

Extendemos H a $\Sigma \cup \Pi$ aplicando cada predicado de estado en sí mismo. De este modo, para todo término cerrado t de la familia *State* y predicado de estado q_j, si

$$E_A \cup D_A \vdash (H(t) \models q_j) = \mathit{true}$$

entonces, por la ecuación que define q_j en $E_A \cup D_A$ y como $H(t) = \mathit{bst}(p_1(t), \ldots, p_n(t))$, tenemos $E_A \cup D_A \vdash p_j(t) = \mathit{true}$ e incluso $E_A \vdash p_j(t) = \mathit{true}$, ya que p_j está completamente especificado en E_A. Así, debido a la relación entre las ecuaciones que definen $p_j(S)$ y las que definen $S \models q_j$, se tiene $E \cup D \vdash (t \models q_j) = \mathit{true}$ con lo que se garantiza la preservación de predicados.

Finalmente podemos poner todas las piezas juntas y resumir la discusión anterior como sigue.

Teorema 5.9 *Sea un sistema concurrente especificado como un objeto $(\mathcal{R}, (\Sigma', E \cup D), J)$ de* **RWThHom**$_{\models}$*, donde $\mathcal{R}$ es $J(State)$-encapsulada, y sea $\{\phi_1, \ldots, \phi_n\}$ un conjunto de predicados de estado sobre la familia $J(State)$, donde cada predicado de estado $q_j \in \Pi$ (que suponemos que son constantes) corresponde a ϕ_j, $1 \leq j \leq k$. El resultado de aplicar abstracción de predicados es el sistema dado por $(\mathcal{R}_A, (\Sigma'_A, E_A \cup D_A), J_A)$, donde $(\Sigma'_A, E_A \cup D_A)$ y $\mathcal{R}_A$ están definidos como se ha explicado más arriba y donde $J_A(State) = BState$. Entonces, $H : (\mathcal{R}, (\Sigma', E \cup D), J) \longrightarrow (\mathcal{R}_A, (\Sigma'_A, E_A \cup D_A), J_A)$ es una flecha en* **RWThHom**$_{\models}$*, donde H es el morfismo de signaturas $\Sigma \cup \Pi \longrightarrow \Sigma'_A \cup \Pi$.*

Ejemplo. Para ilustrar estas ideas vamos a aplicarlas al caso del protocolo de la panadería de la sección 4.1. Recordemos que este protocolo, que garantiza el acceso mutuamente exclusivo a un recurso asignando números a los procesos y atendiéndolos según el orden que establecen estos, se especificó mediante una teoría de reescritura $\mathcal{R} = (\Sigma, E, R)$ página 74 y fue extendido con predicados de estado en una teoría extendida $(\Sigma, E) \subseteq (\Sigma', E \cup D)$ en la página 75.

Vamos a definir la abstracción de predicados mediante los siguiente siete predicados (que están en estrecha correspondencia con los que se utilizaron en la abstracción ecuacional que se aplicó al protocolo):

$$\begin{aligned}
\phi_1(\texttt{< P, X, Q, Y >}) &\iff \texttt{P} = \texttt{wait}\\
\phi_2(\texttt{< P, X, Q, Y >}) &\iff \texttt{P} = \texttt{crit}\\
\phi_3(\texttt{< P, X, Q, Y >}) &\iff \texttt{Q} = \texttt{wait}\\
\phi_4(\texttt{< P, X, Q, Y >}) &\iff \texttt{Q} = \texttt{crit}\\
\phi_5(\texttt{< P, X, Q, Y >}) &\iff \texttt{X} = \texttt{0}\\
\phi_6(\texttt{< P, X, Q, Y >}) &\iff \texttt{Y} = \texttt{0}\\
\phi_7(\texttt{< P, X, Q, Y >}) &\iff \texttt{X} < \texttt{Y} = \texttt{true}
\end{aligned}$$

Intuitivamente, solo nos preocupamos de si los procesos están en modo de espera (`wait`) o crítico (`crit`), si sus contadores son iguales a cero y cuál de los contadores es el mayor.

Nótese que los predicados de estado de la signatura Σ' están en correspondencia con los predicados 1–4. En términos de la notación utilizada más arriba, q_1 sería `1wait` y estaría asociado al predicado ϕ_1; q_2 sería `1crit` y estaría asociado a ϕ_2; y q_3 y q_4 serían `2wait` y `2crit`, asociados a ϕ_3 y ϕ_4. Ahora la teoría de reescritura abstracta $\mathcal{R}_A = (\Sigma_A, E_A, R_A)$ se construye añadiendo a $\mathcal{R}$ los siguientes elementos:

- Operadores `p1 : State -> Bool`, ..., `p7 : State -> Bool`, junto con una nueva familia `BState` y el constructor de estados abstractos

  ```
  op bst : Bool Bool Bool Bool Bool Bool Bool -> BState .
  ```

  Esto determina el morfismo de signaturas H, que lleva el constructor `st` al término

  ```
  bst(p1(< P, X, Q, Y >),...,p7(< P, X, Q, Y >))
  ```

- Ecuaciones asociadas a pi que especifican ϕ_i, para $i = 1, \ldots, 7$. Como los predicados $\phi_1, \ldots, \phi_4$ corresponden a los predicados de estado, las ecuaciones que los definen son "las mismas" que antes:

  ```
  eq p1(< P, X, Q, Y >) = (P == wait) .
  eq p2(< P, X, Q, Y >) = (P == crit) .
  eq p3(< P, X, Q, Y >) = (Q == wait) .
  eq p4(< P, X, Q, Y >) = (Q == crit) .
  ```

  Las tres ecuaciones restantes también son inmediatas:

  ```
  eq p5(< P, X, Q, Y)) = (X == 0) .
  eq p6(< P, X, Q, Y)) = (Y == 0) .
  eq p7(< P, X, Q, Y >) = (Y < X) .
  ```

- La traducción de las reglas en R por el morfismo de signaturas H; por ejemplo, las reglas `[p2_sleep]` y `[p2_wait]` se convierten respectivamente en:

  ```
  rl bst(p1(< P, X, sleep, Y >), ..., p7(< P, X, sleep, Y >)) =>
     bst(p1(< P, X, wait, Y >), ..., p7(< P, X, wait, s X >)) .

  crl bst(p1(< P, X, wait, Y >), ..., p7(< P, X, wait, Y >)) =>
      bst(p1(< P, X, crit, Y >), ..., p7(< P, X, crit, Y >))
      if Y < X .
  ```

Finalmente tenemos que escribir las ecuaciones de D_A que definen los predicados de estado en el modelo abstracto, lo que es inmediato.

```
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 1wait) = B1 .
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 1crit) = B2 .
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 2wait) = B3 .
eq (bst(B1, B2, B3, B4, B5, B6, B7) |= 2crit) = B4 .
```

Por construcción, este modelo es una abstracción de predicados del protocolo de la panadería, con respecto a la base $\phi_1, \ldots, \phi_7$.

Es importante darse cuenta de que este método algebraico de definir la abstracción de predicados no puede ser expresado en el marco de las abstracciones ecuacionales del

capítulo 4 porque la especificación de los predicados ϕ_i requiere, en general, introducir operadores auxiliares y por lo tanto una signatura distinta $\Sigma_A \neq \Sigma$. También hay que señalar que la teoría de reescritura resultante *no es en general ejecutable*. Esto significa que esta teoría de reescritura no se puede utilizar directamente en una herramienta como el comprobador de modelos de Maude. La abstracción de predicados se puede considerar como un caso particular de nuestro marco de simulaciones algebraicas desde un punto de vista conceptual o de fundamentos, lo que resulta útil al ofrecer una justificación del método.

Casi todas las aproximaciones al método de abstracción de predicados no trabajan directamente con la relación de transición minimal (que viene dada en nuestra descripción por $\mathcal{R}_A$). En su lugar, computan una aproximación más grosera de $\mathcal{R}_A$, aunque correcta. En el capítulo 7 exploramos cómo computar tales aproximaciones en nuestro marco mediante un prototipo que combina el uso de reflexión y el ITP para realizar la abstracción de predicados.

5.7.3 Un ejemplo de justicia

En muchas situaciones estamos interesados en el comportamiento de un sistema bajo ciertas hipótesis de *justicia*, como la de requerir que una regla termine ejecutándose si está habilitada siempre desde un cierto punto en adelante. En la actualidad, el comprobador de modelos de Maude no posee la capacidad de concentrarse solo en aquellos caminos que satisfacen las hipótesis de justicia. Sin embargo, no todo está perdido y vamos a ilustrar el uso de morfismos de (bi)simulación teoroidales para razonar sobre justicia. El mismo tratamiento se puede aplicar para clases muy generales de teorías de reescritura y nociones más flexibles de justicia (Meseguer, 2004). Aquí nos vamos a limitar a ilustrar algunas de las ideas principales, incluyendo el uso de morfismos teoroidales, por medio de un sencillo protocolo de comunicación. También es importante señalar que las mismas ideas se pueden utilizar para representar y estudiar sistemas de transiciones *etiquetados* en la lógica de reescritura.

Consideremos un sistema formado por un emisor, un canal y un receptor. El objetivo es enviar un multiconjunto de números (en un orden arbitrario) desde el emisor al receptor a través del canal. El canal puede contener en un momento dado varios de estos números. Además de las acciones normales de envío y recepción, el canal puede atascarse un número arbitrario de ocasiones al enviar algún dato. Podemos modelar los estados de un sistema tal por medio de la signatura

```
ops snd ch rcv : Nat -> Conf .
op null : -> Conf .
op __ Conf Conf -> Conf [assoc comm id: null] .
```

donde el operador `__` denota la unión de multiconjuntos, satisface las ecuaciones de asociatividad y conmutatividad y tiene a `null` como elemento identidad. Por ejemplo, el término

```
snd(7) snd(3) snd(7) ch(2) ch(3) rcv(1) rcv(9)
```

describe el estado en el que 3 y dos copias de 7 no han sido enviadas todavía, 2 y otra copia de 3 están en el canal y 1 y 9 ya han sido recibidos. El comportamiento del sistema se especifica por medio de las siguientes tres reglas de reescritura:

```
var N : Nat .

rl [send] : snd(N) => ch(N) .
rl [stall] : ch(N) => ch(N) .
rl [receive] : ch(N) => rcv(N) .
```

¿Es este sistema terminante? No sin hipótesis adicionales, puesto que la regla `stall` se podría aplicar continuamente. Para hacerlo terminante es suficiente con asumir la siguiente propiedad de *justicia débil* sobre la regla `receive`, descrita por la fórmula

$$\textit{wf-receive} = \mathbf{FG}\,\texttt{enabled-receive} \rightarrow \mathbf{GF}\,\texttt{taken-receive}\,;$$

es decir, si la regla `receive` termina por estar habilitada continuamente en un camino, entonces tiene que ser ejecutada un número infinito de veces. La especificación del predicado `enabled-receive` es bastante fácil (solo necesitamos comprobar que hay algún valor en el canal) pero la especificación del predicado `taken-receive` es más problemática. Por ejemplo, ¿se satisface el predicado `taken-receive` en el estado descrito anteriormente? No lo sabemos; quizá la última acción fue recibir el valor 1, en cuyo caso sí se cumpliría, pero en cambio se podría haber atascado en 3 o haber enviado 2 y entonces no se tendría. En este punto es donde entra en juego una transformación de teorías correspondiente a un morfismo teoroidal que nos permita definir un sistema bisimilar donde el predicado `taken-receive` pueda definirse. La nueva teoría extiende la signatura de arriba con los siguientes nuevos tipos y operadores:

```
ops send stall receive * : -> Label .
op {_|_} : Configuration Label -> State .
```

Ahora un estado consiste en un par configuración-etiqueta, indicando la última regla que fue aplicada. Como inicialmente ninguna regla ha sido aplicada, añadimos la etiqueta * para todos los estados iniciales. Las reglas de la teoría transformada son ahora:

```
var Conf : Configuration .
var L : Label .

rl [send] : { Conf snd(n) | L } => { conf ch(n) | send } .
rl [stall] : { Conf ch(n) | L } => { conf ch(n) | stall } .
rl [receive] : { Conf ch(n) | L } => { conf rcv(n) | receive } .
```

Los predicados `enabled-send`, `enabled-receive` y `taken-receive` se pueden definir entonces mediante las ecuaciones

```
eq ({ Conf snd(N) | L } |= enabled-send) = true .
eq ({ Conf ch(N) | L } |= enabled-receive) = true .
eq ({ Conf | receive } |= taken-receive) = true .
```

La propiedad de "terminación justa" se puede definir ahora con la siguiente fórmula, que ciertamente se cumple en la estructura de Kripke asociada a esta teoría transformada para cualquier estado inicial:

$$\mathbf{A}(\mathit{wf\text{-}receive} \rightarrow \mathbf{F}(\neg\texttt{enabled-send} \wedge \neg\texttt{enabled-receive}))$$

Denotemos por $(\Sigma_{Comm}, E_{Comm})$ la teoría ecuacional subyacente en nuestra teoría de reescritura original y sea $(\Sigma_{LComm}, E_{Comm})$ la signatura de la teoría transformada (tiene las mismas ecuaciones E_{Comm}). Definimos un morfismo generalizado de teorías $H : (\Sigma_{LComm}, E_{Comm}) \longrightarrow (\Sigma_{Comm}, E_{Comm})$ como sigue: los tipos, familias implícitas y operadores en Σ_{Comm} se llevan a sí mismos; el tipo `State` se aplica en `Conf`; el tipo `Label` no se lleva a ninguna parte; el operador `{_|_}` se lleva a la variable `conf` de tipo `Conf`; y finalmente, la imagen de las etiquetas se deja indefinida. Sea Π_0 el conjunto de predicados `enabled-send` y `enabled-receive`, que en la teoría original están definidos por las ecuaciones

```
eq (Conf snd(N) |= enabled-send) = true .
eq (Conf ch(N) |= enabled-receive) = true .
```

Entonces, si *Comm* y *LComm* denotan nuestras teorías de reescritura, H induce una *bisimución* teoroidal (por lo tanto, estricta) de estructuras de Kripke

$$H : \mathcal{K}(\mathit{LComm}, [\texttt{State}])_{\Pi_0} \longrightarrow \mathcal{K}(\mathit{Comm}, [\texttt{Conf}])_{\Pi_0} .$$

Además, en el caso de *LComm* podemos extender Π_0 a Π con el predicado `taken-receive`, de manera que la propiedad de terminación justa puede ser adecuadamente especificada y verificada.

5.8 Demostrando la corrección de simulaciones algebraicas

En esta sección discutimos algunos métodos para demostrar que un morfismo teoroidal dado o una función definida ecuacionalmente está realmente especificando una simulación entre dos sistemas computacionales. Comenzamos considerando el caso más sencillo de simulaciones no tartamudas y trataremos las tartamudas después.

5.8.1 Preservación de la transición de relación en $\mathbf{RWThHom}_{\models}$ y $\mathbf{RWTh}_{\models}$

Vamos a empezar con la categoría $\mathbf{RWThHom}_{\models}$. Un sencillo criterio para saber si un morfismo teoroidal $H : \mathcal{R}_1 \longrightarrow \mathcal{R}_2$ realmente preserva la relación de transición es comprobar que, para cada regla $t \longrightarrow t'$ **if** C en $\mathcal{R}_1$, hay una regla $H(t) \longrightarrow H(t')$ **if** $H(C)$ correspondiente en $\mathcal{R}_2$. Este requisito, sin embargo, es demasiado estricto.

Otra posibilidad reside en utilizar un demostrador de teoremas. En Maude tenemos el ITP, pero desafortunadamente por el momento no permite el razonamiento sobre reglas de reescritura. Sin embargo, utilizando una construcción descrita en Bruni y Meseguer (2003) ese razonamiento es posible de manera indirecta. En Bruni y Meseguer (2003), a cada teoría de reescritura $\mathcal{R}$ se le asocia una teoría $\mathit{Reach}(\mathcal{R})$ en la lógica de pertenencia, con

tipos Ar_k, Ar^1_k y operadores $\_ \to \_$ para cada familia k en $\mathcal{R}$, de tal forma que $\mathcal{R} \vdash t \longrightarrow t'$ si y solo si $Reach(\mathcal{R}) \vdash (t \to t') : Ar_k$, y $t \to^1_{\mathcal{R},k} t'$ si y solo si $Reach(\mathcal{R}) \vdash (t \to t') : Ar^1_k$. Basándonos en este resultado, la siguiente proposición ofrece un criterio para comprobar si la relación de transición se preserva.

Proposición 5.11 *Sean $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ y $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$ dos teorías de reescritura y sea $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ un morfismo generalizado de teorías con semántica inicial tal que, para todo $f \in \Sigma_1$, el término $H(f)$ no contiene múltiples ocurrencias de una misma variable. Sea T una teoría en la lógica de pertenencia que extiende la unión disjunta de (Σ_1, E_1) y (Σ_2, E_2) de manera conservadora con operadores y sentencias definiendo $\to^1_{\mathcal{R}_1,k}$ y $\to^1_{\mathcal{R}_2,k}$ como tipos $Ar1^1_k$ y $Ar2^1_k$, y en la que el morfismo H está especificado ecuacionalmentee a través de operadores h. Entonces, si para todas las reglas $(\forall X)\, t \longrightarrow t'$ **if** C en $\mathcal{R}_1$, con t en la familia k, podemos demostrar inductivamente que*

$$T \vdash_{ind} (\forall X)\, (h(t) \to h(t')) : Ar2^1_{H(k)} \textbf{ if } C^\sharp\,,$$

(donde $C^\sharp$ es como C pero con todas las reescrituras $t \longrightarrow t'$ sustituidas por $(t \to t') : Ar1_k$), se sigue que para todas las familias k en $\mathcal{R}_1$ y $t, t' \in T_{\Sigma_1,k}$,

$$t \to^1_{\mathcal{R}_1,k} t' \implies H(t) \to^1_{\mathcal{R}_2,H(k)} H(t')\,.$$

Demostración. Supongamos que $t \to^1_{\mathcal{R}_1,k} t'$. Entonces, o bien existe una regla de reescritura $(\forall X)\, l \longrightarrow r$ **if** C en $\mathcal{R}_1$ y una sustitución θ tal que $E_1 \vdash (\forall \emptyset)\, t = \theta(l)$, $E_1 \vdash (\forall \emptyset)\, t' = \theta(r)$ y $\theta(C)$ se cumple en $\mathcal{R}_1$, o t es $f(t_1, \ldots, t_i, \ldots, t_n)$, t' es $f(t_1, \ldots, t'_i, \ldots, t_n)$ y $t_i \to^1_{\mathcal{R}_1,k_i} t'_i$.

En el primer caso, por la forma en que T ha sido construida tenemos $T \vdash (\forall \emptyset)\, \theta(C^\sharp)$, y a partir de las hipótesis se sigue que $T \vdash (h(t) \to h(t')) : Ar2_k$, lo que implica que $H(t) \to^1_{\mathcal{R}_2,H(k)} H(t')$.

En el segundo caso, por la hipótesis de inducción $H(t_i) \to^1_{\mathcal{R}_2,H(k_i)} H(t'_i)$ y, por nuestra suposición de que $H(f)$ no tiene variables repetidas,

$$H(t) = H(f)(H(t_1), \ldots, H(t_i), \ldots, H(t_n)) \to^1_{\mathcal{R}_2,H(k_i)} H(f)(H(t_1), \ldots, H(t'_i), \ldots, H(t_n)) = H(t')\,.$$

□

Nótese que esta proposición sigue siendo válida incluso si H es solo una función arbitraria, con tal de que pueda ser definida ecuacionalmente. Por lo tanto, el resultado también se aplica a los morfismos en $\mathbf{RWTh}_\models$ y obviamente a aquellos en $\mathbf{RecRWThHom}_\models$ y $\mathbf{RecRWTh}_\models$.

Un sencillo protocolo. Vamos a ilustrar esta idea con un ejemplo. Consideramos el siguiente (¡otro más!) protocolo adaptado de Dams et al. (1997):

```
mod PROTOCOL is
  protecting NAT .
  sorts State Mode .
  ops think eat : -> Mode .
```

```
  op st : Mode Mode Nat -> State .
  op odd : Nat -> Bool .

  vars M N : Mode .  var X : Nat .

  eq odd(0) = false .
  eq odd(s(X)) = not(odd(X)) .

  crl st(think, N, X) => st(eat, N, X) if odd(X) = true .
  rl  st(eat, N, X) => st(think, N, 3 * X + 1) .
  crl st(M, think, X) => st(M, eat, X) if odd(X) = false .
  crl st(M, eat, X) => st(M, think, X quo 2) if odd(X) = false .
endm
```

Siguiendo a Dams et al. (1997), esta especificación se puede entender como un protocolo que controla el acceso mutuamente exclusivo a un recurso común de dos procesos, que modelan el comportamiento de dos matemáticos, correspondientes a las dos primeras componentes en un estado representado por `st(M, N, X)`. Alternan fases de pensar (`think`) y comer (`eat`), reguladas por el valor actual de `X` en la tercera componente del estado: si `X` es impar, el primer matemático tiene derecho a disfrutar de la comida, en caso contrario el turno le corresponde al segundo. Tras terminar de comer, cada matemático deja el comedor y modifica el valor de `N` siguiendo un criterio propio.

Consideremos ahora el siguiente módulo, que afirmamos especifica una abstracción correcta del sistema especificado por `PROTOCOL`, que reemplaza el tercer argumento del estado por su paridad.

```
mod PROTOCOL-ABS is
  sorts State Mode Parity .
  ops think eat : -> Mode .
  ops o e : -> Parity .
  op st : Mode Mode Parity -> State .

  vars M N : Mode .

  rl st(think, N, o) => st(eat, N, o) .
  rl st(eat, N, o) => st(think, N, e) .
  rl st(eat, N, e) => st(think, N, o) .
  rl st(M, think, e) => st(M, eat, e) .
  rl st(M, eat, e) => st(M, think, e) .
  rl st(M, eat, e) => st(M, think, o) .
endm
```

La teoría T en la proposición 5.11 correspondiente a estos dos módulos queda descrita como sigue. (En realidad, como los módulos son bastante simples podemos tomar como tal la teoría de abajo, que es una simplificación de la teoría que resultaría de aplicar la construcción general dada en Bruni y Meseguer (2003).)

```
fmod ABSTRACTION is protecting NAT .
  sorts AR1 AR1? AR2 AR2? .
```

```
  sorts Mode1 Mode2 State1 State2 Parity .

  subsort AR1 < AR1? .
  subsort AR2 < AR2? .

  op odd : Nat -> Bool .

  ops think1 eat1 : -> Mode1 [ctor] .
  op st1 : Mode1 Mode1 Nat -> State1 [ctor] .

  ops think2 eat2 : -> Mode2 [ctor] .
  ops o e : -> Parity [ctor] .
  op st2 : Mode2 Mode2 Parity -> State2 [ctor] .

  op _->_ : State1 State1 -> AR1? [ctor] .
  op _->_ : State2 State2 -> AR2? [ctor] .

  op abs : State1 -> State2 .
  op absMode : Mode1 -> Mode2 .

  vars M1 N1 : Mode1 .
  vars M2 N2 : Mode2 .
  vars X Y : Nat .

  eq odd(0) = false .
  eq odd(s(X)) = not(odd(X)) .

  eq absMode(think1) = think2 .
  eq absMode(eat1) = eat2 .

  ceq abs(st1(M1, N1, X)) = st2(absMode(M1), absMode(N1), o)
      if odd(X) = true  .
  ceq abs(st1(M1, N1, X)) = st2(absMode(M1), absMode(N1), e)
      if odd(X) = false .

  cmb st1(think1, N1, X) -> st1(eat1, N1, X) : AR1 if odd(X) = true .
  mb st1(eat1, N1, X) -> st1(think1, N1, (3 * X) + 1) : AR1  .
  cmb st1(M1, think1, X) -> st1(M1, eat1, X) : AR1 if odd(X) = false .
  cmb st1(M1, eat1, X) -> st1(M1, think1, X quo 2) : AR1 if odd(X) = false .

  mb st2(think2, N2, o) -> st2(eat2, N2, o) : AR2 .
  mb st2(eat2, N2, o) -> st2(think2, N2, e) : AR2 .
  mb st2(eat2, N2, e) -> st2(think2, N2, o) : AR2 .
  mb st2(M2, think2, e) -> st2(M2, eat2, e) : AR2 .
  mb st2(M2, eat2, e) -> st2(M2, think2, e) : AR2 .
  mb st2(M2, eat2, e) -> st2(M2, think2, o) : AR2 .
endfm
```

Nótese cómo cada una de las reglas en las especificaciones originales ha dado lugar a un axioma de pertenencia que define uno de los dos tipos `AR1` o `AR2`.

Ahora podemos demostrar con el ITP que la transición dada por la cuarta regla en

PROTOCOL se preserva en PROTOCOL-ABS:

```
(goal abstract4 : ABSTRACTION |- A{ M1:Mode1 ; X:Nat }
      (((odd(X:Nat)) = (false)) =>
      ((abs(st1(M1:Mode1, eat1, X:Nat)) ->
        abs(st1(M1:Mode1, think1, X:Nat quo 2))) : AR2)) .)

(auto* .)
(split on (odd(X*Nat quo 2)) .)
(auto* .)
(auto* .)
```

La demostración para las demás reglas se haría de forma parecida.

Hay que señalar que, sin embargo, la proposición 5.11 no resulta muy útil cuando la condición C contiene reglas. Consideremos por ejemplo una teoría de reescritura $\mathcal{R}_1$ con dos operadores unarios f_1 y g_1, y la regla $(\forall\{x,y\})\, f_1(x) \to g_1(y)$ **if** $x \to y$. Escribamos $\mathcal{R}_2$ para referirnos a la teoría de reescritura que se obtiene a partir de $\mathcal{R}_1$ renombrando f_1 y g_1 como f_2 y g_2, respectivamente. $\mathcal{R}_1$ y $\mathcal{R}_2$ están claramente relacionadas por un morfismo teoroidal H (un renombramiento) y la relación de transición es preservada trivialmente. Pero para demostrarlo utilizando el resultado anterior tendríamos que mostrar

$$T \vdash_{ind} (\forall x, y)\, (h(f_1(x)) \to h(g_1(x))) : Ar2^1_{H(k)} \textbf{ if } (x \to y) : Ar1_k\,,$$

que requiere el uso de alguna clase de hipótesis de inducción sobre $(x \to y) : Ar1_k$, que no está disponible. Afortunadamente, muchas especificaciones no requieren el uso de reglas en las condiciones; en particular, las teorías de reescritura recursivas pertenecen a esta clase (recordemos la definición 5.13).

5.8.2 Preservación de la relación de transición en SRWThHom$_\models$ y SRWTh$_\models$

La definición de simulaciones tartamudas requiere el cumplimiento de una propiedad en la que intervienen caminos infinitos, que en general no es fácil de comprobar. En la sección 5.2 presentamos una caracterización alternativa que también se puede utilizar para morfismos entre teorías de reescritura; sin embargo, obviamente preferiríamos tener condiciones que se aplicaran directamente a los conjuntos de ecuaciones y reglas de dichas teorías.

Supongamos que tenemos dos teorías de reescritura $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ y $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$, y un morfismo generalizado de teorías con semántica inicial $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$. Para que H sea un morfismo tartamudo es suficiente con mostrar que algunas de las reglas en R_1 dan lugar a pasos de reescritura en $\mathcal{R}_2$ mientras que el resto se convierten en tartamudeo cuando son traducidas. Esto es, R_1 se puede descomponer como la unión disjunta de R_1' y R_1'' de manera que si $t \to^1_{R_1',k} t'$ entonces $H(t) \to^1_{\mathcal{R}_2,H(k)} H(t')$ y si $t \to^1_{R_1'',k} t'$ entonces $H(t)$ y $H(t')$ se pueden probar iguales en $\mathcal{R}_2$. Solo falta un pequeño detalle: para evitar un tartamudeo infinito necesitamos exigir que R_1'' sea terminante. Esta idea queda formalizada en la siguiente proposición.

Proposición 5.12 *Sean $\mathcal{R}_1 = (\Sigma_1, E_1, R_1)$ y $\mathcal{R}_2 = (\Sigma_2, E_2, R_2)$ dos teorías de reescritura y sea $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ un morfismo generalizado de teorías con semántica inicial tal que, para todo $f \in \Sigma_1$, el término $H(f)$ no contiene múltiples ocurrencias de una misma variable. Sea T una teoría en lógica de pertenencia que extiende la unión disjunta de (Σ_1, E_1) y (Σ_2, E_2) con operadores y sentencias que definen $\rightarrow^1_{\mathcal{R}_1,k}$ y $\rightarrow^1_{\mathcal{R}_2,k}$ como tipos $Ar1^1_k$ y $Ar2^1_k$, y en la que el morfismo H se especifica ecuacionalmente a través de operadores h. Supongamos que R_1 es la unión disjunta de R'_1 y R''_1, con R''_1 terminante módulo E_1. Entonces, si para todas las reglas $(\forall X)\, t \longrightarrow t'$ **if** C en R'_1, con t en la familia k, podemos demostrar inductivamente*

$$T \vdash_{ind} (\forall X)\, (h(t) \rightarrow h(t')) : Ar2^1_{H(k)} \textbf{ if } C^\sharp\,,$$

*(donde $C^\sharp$ es como C pero con las reescrituras $t \longrightarrow t'$ sustituidas por $(t \rightarrow t') : Ar1_k$), y para todas las reglas $(\forall X)\, t \longrightarrow t'$ **if** C en R''_1 con t en la familia k,*

$$T \vdash_{ind} (\forall X)\, (h(t) = h(t')) \textbf{ if } C^\sharp\,,$$

se sigue que todo camino en $\mathcal{R}_1$ H-encaja con algún camino en $\mathcal{R}_2$.

Demostración. Sea π un camino $t = t_0 \rightarrow_{\mathcal{R}_1} t_1 \rightarrow_{\mathcal{R}_1} t_2 \rightarrow_{\mathcal{R}_1} \cdots$ que comienza en $t \in T_{\Sigma_1,k_1}$: tenemos que demostrar que existe un camino ρ en $\mathcal{R}_2$ que empieza en $H(t)$ y que H-encaja con π. Para ello definimos $\alpha(0) = 0$ y $\alpha(i+1)$ como la primera posición en π mayor que $\alpha(i)$ que resulte de aplicar una regla en R'_1; como R''_1 es terminante, α está bien definida y es estrictamente creciente. Definimos entonces ρ mediante $\rho(i) = H(t_{\alpha(i)})$. Resulta que $t_{\alpha(i)}$ alcanza $t_{\alpha(i+1)}$ tras un número finito de reescrituras en R''_1 y una única transición en R'_1: $t_{\alpha(i)} \rightarrow^1_{R''_1,k} u_1 \rightarrow^1_{R''_1,k} u_2 \rightarrow^1_{R''_1,k} \cdots \rightarrow^1_{R''_1,k} u_n \rightarrow^1_{R'_1,k} t_{\alpha(i+1)}$. Por los supuestos de la proposición, aplicando el mismo razonamiento que en la demostración de la proposición 5.11 tenemos que $H(u_n) \rightarrow^1_{\mathcal{R}_2,H(k)} H(t_{\alpha(i+1)}) = \rho(i+1)$. Análogamente, $t \rightarrow^1_{R''_1,k} t'$ implica $E_2 \vdash (\forall \emptyset)\, H(t) = H(t')$ y así $\rho(i) = H(t_{\alpha(i)})$, $u_1, \ldots, u_n$ se pueden probar iguales en E_2. Se sigue que $\rho(i) \rightarrow^1_{\mathcal{R}_2,H(k)} \rho(i+1)$ y por lo tanto ρ es un camino válido en $\mathcal{R}_2$ que H-encaja con π por construcción. □

Como en el caso de las simulaciones no tartamudas, esta proposición se aplica no solo a morfismos generalizados de teorías sino también a cualquier función definible ecuacionalmente, con lo que nos proporciona un criterio para comprobar la preservación de transiciones en $\mathbf{SRWTh}_{\models}$.

Un sistema de lectores y escritores. Para ilustrar el uso de esta última proposición vamos a utilizar la siguiente especificación de un sistema de lectores y escritores:

```
mod R&W is
  protecting NAT .
  sort Config .
  op <_,_> : Nat Nat -> Config .  --- lectores/escritores

  vars R W : Nat .
```

```
  rl < 0, 0 > => < 0, s(0) > .
  rl < R, s(W) > => < R, W > .
  rl < R, 0 > => < s(R), 0 > .
  rl < s(R), W > => < R, W > .
endm
```

Los estados se representan por medio de pares `< R, W >` que indican el número `R` de lectores y el número `W` de escritores que están accediendo al recurso crítico. Los lectores y los escritores pueden abandonar el recurso en cualquier momento, pero los escritores solo pueden acceder a él si nadie más lo está usando, y los lectores solo cuando no hay escritores.

Ahora consideremos la siguiente implementación del sistema en la que los lectores y los escritores "piden permiso" antes de entrar en la sección crítica.

```
mod R&W-STUTTERING is
  protecting NAT .
  sorts Key Config .

  ops reader writer empty : -> Key .
  op <_,_,_> : Nat Nat Key -> Config .

  vars R W : Nat .
  var K : Key .

  rl [writer-ask] : < 0, 0, empty > => < 0, 0, writer > .
  rl [writer-out] : < R, s(W), K > => < R, W, K > .
  rl [reader-ask] : < R, 0, empty > => < R, 0, reader > .
  rl [reader-out] : < s(R), W, K > => < R, W, K > .
  rl [writer-in] : < R, W, writer > => < R, s(W), empty > .
  rl [reader-in] : < R, W, reader > => < s(R), W, empty > .
endm
```

La tercera componente de la terna de un estado indica si un lector (`reader`) o un escritor (`writer`) ha pedido permiso para entrar en la región crítica; su valor es `empty` si no se ha hecho ninguna petición.

Podemos demostrar que `R&W-STUTTERING` es una implementación correcta de `R&W` construyendo un morfismo tartamudo de sistemas de transiciones

$$h : \mathcal{T}(\texttt{R\&W-STUTTERING})_{\texttt{Config}} \longrightarrow \mathcal{T}(\texttt{R\&W})_{\texttt{Config}} .$$

Para ello, si la teoría T en la proposición 5.12 renombra el tipo `Config` en los módulos `R&W` y `R&W-STUTTERING` como `Config1` y `Config2`, respectivamente, el morfismo tartamudo h puede definirse ecuacionalmente en T como sigue. (De nuevo se trata de una simplificación de la construcción en Bruni y Meseguer (2003).)

```
fmod R&W-SIMULATION is
  protecting NAT .
```

```
  sorts AR1 AR1? AR2 AR2? .
  sorts Config1 Key Config2 .

  subsort AR1 < AR1? .
  subsort AR2 < AR2? .

  op <_,_> : Nat Nat -> Config1 [ctor] .  --- lectores/escritores

  ops reader writer empty : -> Key [ctor] .
  op <_,_,_> : Nat Nat Key -> Config2 [ctor] .

  op _->_ : Config1 Config1 -> AR1? [ctor] .
  op _->_ : Config2 Config2 -> AR2? [ctor] .

  op h : Config2 -> Config1 .

  vars R W : Nat .
  var K : Key .

  eq h(< R, W, empty >) = < R, W > .
  eq h(< R, W, reader >) = < s(R), W > .
  eq h(< R, W, writer >) = < R, s(W) > .

  mb < 0, 0 > -> < 0, s(0) > : AR1 .
  mb < R, s(W) > -> < R, W > : AR1 .
  mb < R, 0 > -> < s(R), 0 > : AR1 .
  mb < s(R), W > -> < R, W > : AR1 .

  mb < 0, 0, empty > -> < 0, 0, writer > : AR2 .
  mb < R, s(W), K > -> < R, W, K > : AR2 .
  mb < R, 0, empty > -> < R, 0, reader > : AR2 .
  mb < s(R), W, K > -> < R, W, K > : AR2 .
  mb < R, W, writer > -> < R, s(W), empty > : AR2 .
  mb < R, W, reader > -> < s(R), W, empty > : AR2 .
endfm
```

Tomando como R_1 las primeras cuatro reglas de `R&W-STUTTERING` y como R_2 las dos restantes, tenemos que R_1 es terminante porque el número total de lectores, escritores y "`empty`s" decrece. Consideremos ahora `writer-out`, la segunda regla en R_1, y recordemos que la relación de reescritura en `R&W` es representada por un tipo `AR1` en T. Podemos entonces demostrar

```
(goal abstract2 : R&W-SIMULATION |- A{ R:Nat ; W:Nat ; K:Key }
      ((h(< R:Nat, s(W:Nat), K:Key >) -> h(< R:Nat, W:Nat, K:Key >)) : AR1) .)
```

en el ITP utilizando los comandos

```
(auto* .)
(ctor-split on K*Key .)
(auto* .)
```

```
(auto* .)
(auto* .)
```

y de forma parecida para las otras reglas en R_1. En el caso de R_2 también podemos demostrar de manera inmediata

```
(goal stutt1 : R&W-SIMULATION |- A { R:Nat ; W:Nat }
      ((h(< R:Nat, W:Nat, writer >)) = (h(< R:Nat, s(W:Nat), empty >))) .)
```

y

```
(goal stutt2 : R&W-SIMULATION |- A { R:Nat ; W:Nat }
      ((h(< R:Nat, W:Nat, reader >)) = (h(< s(R:Nat), W:Nat, empty >))) .)
```

Por lo tanto, las condiciones en la proposición 5.12 se cumplen con lo que h es un morfismo tartamudo de sistemas de transiciones.

5.8.3 Preservación de proposiciones atómicas

Para demostrar que las proposiciones atómicas son preservadas es conveniente asumir que están completamente especificadas, en el sentido de que siempre podemos probar si son ciertas o falsas con respecto a un estado concreto. (Recordemos que, como se comentó en la sección 2.4.4, el comprobador de modelos de Maude solo necesita para funcionar que se especifiquen los casos positivos.) Para el caso de propiedades decidibles podemos realizar esta suposición sin pérdida de generalidad. Tenemos entonces el siguiente resultado, parecido a los de preservación de la relación de transición.

Proposición 5.13 *Sean* $(\Sigma_1, E_1) \subseteq (\Sigma_1', E_1 \cup D_1)$ *y* $(\Sigma_2, E_2) \subseteq (\Sigma_2', E_2 \cup D_2)$ *las teorías ecuacionales correspondientes a dos objetos en* $\mathbf{SRWThHom}_{\models}$ *y sea* $H : \Sigma_1 \cup \Pi_1 \longrightarrow \Sigma_2 \cup \Pi_2$ *un morfismo generalizado de signaturas tal que* $H : (\Sigma_1, E_1) \longrightarrow (\Sigma_2, E_2)$ *es un morfismo de teorías. Sea* T *una teoría en lógica de pertenencia que extiende la unión disjunta de* $(\Sigma_1', E_1 \cup D_1)$ *y* $(\Sigma_2', E_2 \cup D_2)$ *en la que el morfismo* H *está especificado ecuacionalmente a través de operadores* h*. Entonces, si podemos demostrar que*

$$T \vdash_{ind} (\forall X)\,(h(t) \models h(p)) = \mathit{false}\ \mathbf{if}\ C$$

para todas las ecuaciones $(\forall X)\,(t \models p) = \mathit{false}\ \mathbf{if}\ C$ *en* D_1*, se sigue que, para cualesquiera términos cerrados* t' *y* p'*,*

$$E_2 \cup D_2 \vdash (h(t') \models h(p') = \mathit{true}) \implies E_1 \cup D_1 \vdash (t' \models p' = \mathit{true})\,.$$

Además, si en su lugar podemos demostrar

$$T \vdash_{ind} (\forall X)\,(x \models p) = (h(x) \models h(p))$$

entonces la implicación anterior se convierte en una equivalencia.

Demostración. Por nuestra suposición sobre la completitud de las especificaciones, se tiene que o bien $E_1 \cup D_1 \vdash (\forall \emptyset)\,(t \models p) = \mathit{true}$ o $E_1 \cup D_1 \vdash (\forall \emptyset)\,(t \models p) = \mathit{false}$. Pero este segundo caso no puede ocurrir a menos que $\mathit{true} = \mathit{false}$ en $E_2 \cup D_2$, porque se puede demostrar por inducción estructural y utilizando la hipótesis de la proposición que $E_1 \cup D_1 \vdash (\forall \emptyset)\,(t \models p) = \mathit{false}$ implica $E_2 \cup D_2 \vdash (\forall \emptyset)\,(h(t) \models h(p)) = \mathit{false}$.

Es claro que la condición añadida en el último apartado del enunciado implica que h es estricta, con lo que se tendría la equivalencia indicada. □

De nuevo, este resultado se aplica también a la categoría $\mathbf{SRWTh}_{\models}$.

Un protocolo revisitado. Para el protocolo de los "matemáticos pensadores" del ejemplo de la sección 5.8.1 las proposiciones atómicas se especifican como sigue:

```
eq st1(think1, N1, X) |= nmexcl1 = true .
eq st1(M1, think1, X) |= nmexcl1 = true .
eq st1(eat1, eat1, X) |= nmexcl1 = false .

eq st2(think2, N2, P) |= nmexcl2 = true .
eq st2(M2, think2, P) |= nmexcl2 = true .
eq st2(eat2, eat2, P) |= nmexcl2 = false .
```

Entonces la condición que tenemos que comprobar es

```
(goal abstract : ABSTRACTION |- A{ X:Nat }
      ((abs(st1(eat1, eat1, X:Nat)) |= nmexcl2) = (false)) .)
```

que se puede demostrar en el ITP con los comandos:

```
(auto* .)
(split on (odd(X*Nat)) .)
(auto* .)
(auto* .)
```

De forma parecida, para ver que la simulación es estricta tendríamos que mostrar que

```
(goal abstract-st : ABSTRACTION |- A{ S1:State1 }
      ((S1:State1 |= nmexcl1) = (abs(S1:State1) |= nmexcl2)) .)
```

lo que también se puede demostrar en el ITP de la siguiente forma:

```
(ind on S1:State1 .)
(ind on V0#0:Mode1 .)

(ind on V0#1:Mode1 .)
(auto* .)
(split on (odd(V0#2*Nat)) .)
(auto* .)
```

```
(auto* .)
(auto* .)
(split on (odd(V0#2*Nat)) .)
(auto* .)
(auto* .)

(ind on V0#1:Mode1 .)
(auto* .)
(split on (odd(V0#2*Nat)) .)
(auto* .)
(auto* .)
(auto* .)
(split on (odd(V0#2*Nat)) .)
(auto* .)
(auto* .)
```

5.9 Conclusiones

Hemos presentado una noción muy general de simulación tartamuda entre estructuras de Kripke que relaja los requisitos sobre preservación de predicados, no requiriendo que la preservación sea estricta y permitiendo que las fórmulas se puedan traducir. También hemos demostrado resultados de representabilidad generales que demuestran que tanto las estructuras de Kripke como sus simulaciones pueden ser representadas fructíferamente en la lógica de reescritura. Direcciones de investigación futura incluyen: (i) una búsqueda continuada de simulaciones todavía más generales y técnicas de preservación y composicionalidad relacionadas; (ii) métodos de prueba y soporte de herramientas para demostrar que las simulaciones son correctas; y (iii) experimentación y más casos prácticos.

Capítulo 6

Algunos resultados categóricos

En este capítulo realizamos un estudio categórico de las estructuras que se han desarrollado en capítulos anteriores, centrándonos en las categorías sobre un conjunto fijo de proposiciones atómicas. Empezaremos utilizando el lenguaje de la teoría de categorías para justificar el uso del adjetivo "minimal" en las estructuras de Kripke de la sección 4.3 y para estudiar cómo transferir la función de etiquetado de una estructura de Kripke a un sistema de transiciones. Tras ello agruparemos las categorías en instituciones, lo que nos llevará a una construcción alternativa de las categorías de Grothendieck introducidas en el capítulo 5, para a continuación estudiar sus límites, colímites y factorizaciones. En las categorías de Grothendieck los colímites se pueden obtener imitando las construcciones que damos aquí; en el caso de los límites, conjeturamos que en general estos no existen.

El trabajo presentado aquí es en gran medida ortogonal a los desarrollos previos y, hasta cierto punto, no está tan desarrollado como aquellos. Por ejemplo, debería ser posible continuar con el trabajo sobre instituciones para la lógica de reescritura comenzado en Palomino (2001a) y establecer relaciones entre las correspondientes instituciones que generalicen el diagrama en la sección 5.6.2; todo esto queda pendiente como trabajo futuro.

6.1 Conceptos fundamentales

Casi todas las nociones de la teoría de categorías que vamos a utilizar son bastante básicas y se pueden encontrar en numerosas fuentes, como en el libro clásico de Mac Lane (1998) o el más orientado a (determinado sector dentro de) la informática de Barr y Wells (1999). En esta sección nos limitamos a repasar aquellos conceptos que pueden ser algo menos conocidos o que desempeñan un papel importante en el desarrollo del capítulo. Para intentar evitar confusiones con los morfismos de simulaciones, nos referiremos en muchas ocasiones a los morfismos de una categoría simplemente como flechas.

Opfibraciones. Lo que determina una *opfibración* (por ejemplo, Jacobs, 1999) es la propiedad de poder "elevar" una flecha en una categoría base a otra categoría de manera "inicial" (y por lo tanto, mínima) en un sentido apropiado.

Sea $F : \mathcal{C} \longrightarrow \mathcal{D}$ un funtor. Una flecha $f : X \longrightarrow Y$ en $\mathcal{C}$ es *opcartesiana* sobre u si $F(f) = u$ y cada flecha $g : X \longrightarrow Z$ en $\mathcal{C}$ para la que se tiene $F(g) = v \circ u$ para algún $v : Y \longrightarrow F(Z)$ determina una única flecha $h : Y \longrightarrow Z$ tal que $g = h \circ f$ y $F(h) = v$. El funtor F es una opfibración si para todas las flechas $u : F(X) \longrightarrow J$ existe un morfismo opcartesiano. Las nociones duales son las de morfismo cartesiano y fibración.

Instituciones. La noción de *institución* se debe al trabajo seminal de Goguen y Burstall (1992); el objetivo era capturar la noción de modelo de manera independiente a cualquier formalismo concreto. Una institución es una 4-tupla $\mathcal{I} = (\mathbf{Sign}, \mathit{sen}, \mathbf{Mod}, \models)$ tal que:

- **Sign** es una categoría cuyos objetos se llaman *signaturas*,
- sen : **Sign** $\longrightarrow$ **Set** es un funtor que asocia a cada signatura Σ un conjunto de Σ-*sentencias*,
- **Mod** : **Sign** $\longrightarrow$ $\mathbf{Cat}^{\mathrm{op}}$ es un funtor que asocia a cada signatura Σ una categoría cuyos objetos se llaman Σ-*modelos*, y
- $\models$ es una función que asocia a cada $\Sigma \in |\mathbf{Sign}|$ una relación binaria $\models_\Sigma \subseteq |\mathbf{Mod}(\Sigma)| \times \mathit{sen}(\Sigma)$ llamada Σ-*satisfacción*, de manera que la siguiente propiedad se satisface para todo $H : \Sigma \longrightarrow \Sigma'$, $M' \in |\mathbf{Mod}(\Sigma')|$ y todo $\varphi \in \mathit{sen}(\Sigma)$:

$$M' \models_{\Sigma'} \mathit{sen}(H)(\varphi) \iff \mathbf{Mod}(H)(M') \models_\Sigma \varphi.$$

Un morfismo de teorías $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ es un morfismo $H : \Sigma \longrightarrow \Sigma'$ tal que todo modelo en $\mathbf{Mod}(\Sigma')$ que satisfaga Γ' a su vez satisface $\mathit{sen}(H)(\varphi)$, para toda $\varphi \in \Gamma$.

La *liberalidad* es una propiedad importante que expresa la posibilidad de construcciones libres y generaliza el principio de "semántica algebraica inicial". Una institución es *liberal* si los funtores de olvido $\mathbf{Mod}(H) : \mathbf{Mod}(\Sigma', \Gamma') \longrightarrow \mathbf{Mod}(\Sigma, \Gamma)$ inducidos por los morfismos de teorías $H : (\Sigma, \Gamma) \longrightarrow (\Sigma', \Gamma')$ tienen adjuntos por la izquierda.

Otra propiedad que expresa la posibilidad de "poner teorías juntas" mediante colímites es la *exactitud* de una institución. Una institución es *exacta* si su categoría de signaturas es cocompleta y el funtor de modelos **Mod** preserva colímites, y es *semiexacta* si **Sign** tiene sumas amalgamadas (en inglés, pushouts) y **Mod** la preserva.

Mónadas y categorías de Kleisli. Una *mónada* (llamada *triple* en Barr y Wells, 1999) es una terna (T, η, μ), donde $T : \mathcal{C} \longrightarrow \mathcal{C}$ es un funtor y $\eta : 1_{\mathcal{C}} \longrightarrow T$ y $\mu : T \circ T \longrightarrow T$ son transformaciones naturales que satisfacen $\mu \circ \eta T = \mu \circ T\eta = 1_T$ y $\mu \circ \mu T = \mu \circ T\mu$.

Todas las mónadas se pueden obtener a partir de adjunciones y una de las posibles construcciones hace uso de la *categoría de Kleisli*. La categoría de Kleisli de una mónada (T, η, μ) tiene por objetos los de $\mathcal{C}$. Si X e Y son objetos de $\mathcal{C}$, una flecha $X \longrightarrow Y$ en la categoría de Kleisli es una flecha $X \longrightarrow T(Y)$ en $\mathcal{C}$. La composición de dos flechas $f : X \longrightarrow T(Y)$ y $g : Y \longrightarrow T(Z)$ viene dada por $\mu_C \circ Tg \circ f$.

La construcción de Grothendieck. A menudo nos interesa considerar las componentes de una *categoría indexada* todas juntas en una única categoría "aplanada" que se obtenga formando la unión disjunta de las componentes y añadiendo algunas flechas nuevas. Es la llamada, por ejemplo en Tarlecki et al. (1991), *construcción de Grothendieck*.

Dada una categoría indexada $\mathbf{C} : \mathcal{I}^{\mathrm{op}} \longrightarrow \mathbf{Cat}$, la *categoría de Grothendieck* asociada está definida por:

- los objetos son pares (I, X), donde I es un objeto de $\mathcal{I}$ y X es un objeto de $\mathbf{C}(I)$;
- una flecha $(I, X) \longrightarrow (J, Y)$ es un par (u, f) con $u : I \longrightarrow J$ en $\mathcal{I}$ y $f : X \longrightarrow \mathbf{C}(u)(Y)$ en $\mathbf{C}(I)$;
- la composición de flechas $(u, f) : (I, X) \longrightarrow (J, Y)$ y $(v, g) : (J, Y) \longrightarrow (K, Z)$ viene dada por
$$(v, g) \circ (u, f) = (v \circ u, \mathbf{C}(u)(g) \circ f)\,.$$

Epis y monos regulares. Tal y como se define en Herrlich y Strecker (1973), una flecha $m : X \longrightarrow Y$ es un *monomorfismo regular* si existen flechas f y g tales que m es el igualador (equalizer) de f y g. Dualmente, $e : X \longrightarrow Y$ es un *epimorfismo regular* si es el coigualador de dos flechas.

Dadas dos clases $\mathcal{E}$ y $\mathcal{M}$ de epimorfismos y monomorfimos respectivamente, cerradas bajo composición con isomorfismos, una $(\mathcal{E}, \mathcal{M})$-*factorización* de una flecha f es una factorización $f = m \circ e$ con e en $\mathcal{E}$ y m en $\mathcal{M}$. Una categoría es $(\mathcal{E}, \mathcal{M})$-*factorizable* (unívocamente) si toda flecha tiene una $(\mathcal{E}, \mathcal{M})$-factorización (única). Una categoría es una $(\mathcal{E}, \mathcal{M})$-*categoría* si es unívocamente factorizable y tanto $\mathcal{E}$ como $\mathcal{M}$ son cerradas bajo composición.

6.2 Estructuras de Kripke minimales

Recordemos que en la sección 4.3 se discutió que, a menudo, dada una estructura de Kripke sobre un conjunto de proposiciones atómicas AP, estamos interesados en encontrar otra sobre el mismo conjunto de proposiciones atómicas que sea la que mejor la simule bajo determinadas condiciones. En particular, definimos la noción de *estructura de Kripke minimal* $\mathcal{M}^h_{\mathrm{min}}$ asociada a una estructura de Kripke $\mathcal{M}$ y función sobreyectiva $h : M \to A$.

Entonces se dijo que el adjetivo "minimal" era apropiado y que ya había sido utilizado en Clarke et al. (1994) aplicado a sistemas de transiciones, pero dejamos pendiente ver que esta noción de minimalidad también se aplica a las estructuras de Kripke y que, en nuestro marco categórico, queda capturada por el concepto de morfismo opcartesiano. Ese es el objeto de la siguiente proposición.

Proposición 6.1 *Para una estructura de Kripke $\mathcal{M}$ y una función sobreyectiva $h : M \longrightarrow A$, el AP-morfismo inducido $h : \mathcal{M} \longrightarrow \mathcal{M}^h_{\mathrm{min}}$ es un morfismo opcartesiano en el contexto del funtor de olvido $U : \mathbf{KMap}_{AP} \longrightarrow \mathbf{Set}$ que lleva la estructura de Kripke $\mathcal{M} = (M, \to_{\mathcal{M}}, L_{\mathcal{M}})$ a su conjunto M subyacente y cada AP-morfismo sobre sí mismo.*

Demostración. Dado $f : \mathcal{M} \longrightarrow \mathcal{N}$ en $\mathbf{KMap}_{AP}$ que se pueda factorizar en **Set** como $f = g \circ h$ para alguna función $g : A \longrightarrow N$, tenemos que encontrar un único g' en $\mathbf{KMap}_{AP}$ tal que $g' : \mathcal{M}^h_{\min} \longrightarrow \mathcal{N}$, $f = g' \circ h$ y $U(g') = g$. Por definición de U, debe ser $g' = g$; tenemos que comprobar que g es realmente un AP-morfismo.

Por definición de $\mathcal{M}^h_{\min}$, si $a \rightarrow_{\mathcal{M}^h_{\min}} b$ existen x e y en M tales que $h(x) = a$, $h(y) = b$ y $x \rightarrow_{\mathcal{M}} y$. Entonces, como f es un AP-morfismo, $g(a) = g(h(x)) = f(x) \rightarrow_{\mathcal{N}} f(y) = g(h(y)) = g(b)$. Además, utilizando de nuevo el hecho de que f es un AP-morfismo, si $p \in L_{\mathcal{N}}(s)$ entonces $p \in L_{\mathcal{M}}(x)$ para todo x en M tal que $f(x) = s$. Sea entonces $a \in A$ tal que $g(a) = s$: para todo y en M tal que $h(y) = a$, como $f(y) = g(h(y)) = s$ se tiene que $p \in L_{\mathcal{M}}(y)$. Por lo tanto, $p \in L_{\mathcal{M}^h_{\min}}(a)$ y para todo a con $g(a) = s$ tenemos $L_{\mathcal{N}}(s) \subseteq L_{\mathcal{M}^h_{\min}}(a)$. □

El resultado también se tiene para simulaciones generalizadas en las que el conjunto de proposiciones atómicas puede variar.

Proposición 6.2 *La simulación $(\eta_{AP}, h) : \mathcal{M} \longrightarrow \mathcal{M}^h_{\min}$, con $h : M \longrightarrow A$ una función sobreyectiva y η_{AP} la inclusión $AP \hookrightarrow \text{State}\backslash\neg(AP)$, es un morfismo opcartesiano para el funtor de olvido U : **KMap** $\longrightarrow$ **Set** que lleva un par $(AP, \mathcal{M})$ al conjunto subyacente M y un morfismo de simulación (α, h) a su correspondiente función h.*

Demostración. La demostración sigue los mismos pasos que la de la proposición 6.1, a pesar de que el conjunto de proposiciones atómicas puede ahora variar de una estructura de Kripke a otra. □

6.3 Préstamo

Las simulaciones, en todas sus distintas variantes, requieren alguna forma de preservación de transiciones y proposiciones atómicas. A veces, sin embargo, resulta más fácil y natural pensar simplemente en términos de los sistemas de transiciones subyacentes; en tales casos todavía podemos recuperar una simulación *tomando prestada* la estructura de Kripke del dominio a través de la función de etiquetado del codominio.

Definición 6.1 *Sea $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ un sistema de transiciones y sea $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ una estructura de Kripke sobre un conjunto AP de proposiciones atómicas. Si $h : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$ es un morfismo tartamudo de sistemas de transiciones, $\mathcal{A}$ se puede extender a una estructura de Kripke sobre AP definiendo $L_{\mathcal{A}} = L_{\mathcal{B}} \circ h$. Se dice que $\mathcal{A}$* toma prestadas *sus propiedades de $\mathcal{B}$.*

Proposición 6.3 *Si $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ toma prestadas sus propiedades de una estructura de Kripke $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ sobre un conjunto AP de proposiciones atómicas por medio de una simulación tartamuda de sistemas de transiciones $h : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$, entonces h se convierte en un AP-morfismo tartamudo estricto. Más aún, h es un morfismo cartesiano para el funtor de olvido U : **KSMap**$_{AP}$ $\longrightarrow$ **STSys** que aplica cada estructura de Kripke en su sistema de transiciones subyacente.*

Demostración. Claramente h es un AP-morfismo tartamudo estricto porque, por definición de $L_{\mathcal{A}}$, las proposiciones atómicas son preservadas.

Para mostrar que es un morfismo cartesiano, sean $f : C \longrightarrow \mathcal{B}$ un AP-morfismo tartamudo y $g : U(C) \longrightarrow (A, \rightarrow_{\mathcal{A}})$ un morfismo tartamudo de sistemas de transiciones tales que $f = h \circ g$: tenemos que mostrar que existe un único AP-morfismo tartamudo g' tal que $h \circ g' = f$ y $U(g') = g$. El único candidato posible es g y lo que tenemos que comprobar es que $g : C \longrightarrow \mathcal{A}$ es realmente un AP-morfismo tartamudo. Por hipótesis, g es un morfismo de sistemas de transiciones. Ahora, supongamos que $g(c) = a$ y $p \in L_{\mathcal{A}}(a)$. Se sigue que $p \in L_{\mathcal{B}}(h(a))$ y como $f(c) = (h \circ g)(c) = h(a)$ y f es un AP-morfismo tartamudo, $p \in L_C(c)$ como se necesitaba. □

Es interesante señalar que esta proposición también se cumple incluso si h no es una función (aunque en tal caso la AP-simulación resultante podría no ser estricta).

Uno podría preguntarse si este resultado se extiende a la categoría de Grothendieck **KSMap** de manera que (η_{AP}, h) se convierta en un morfismo cartesiano para el funtor de olvido U : **KSMap** $\longrightarrow$ **STSys**. La respuesta es *no* y el motivo, como ya ocurría cuando señalábamos que las simulaciones estrictas no se podían componer, reside de nuevo en la generalidad de las funciones $\alpha : AP \longrightarrow \text{State}(AP')$ usadas para relacionar estructuras de Kripke sobre conjuntos de proposiciones atómicas diferentes. Sin embargo, como entonces, se puede recuperar el resultado trabajando en la subcategoría **KSMap**^{bool} de **KSMap** en la que el codominio de las funciones α se restringe a $\text{Bool}(AP')$. Ese precisamente es el contenido de la siguiente proposición.

Proposición 6.4 *Si $\mathcal{A} = (A, \rightarrow_{\mathcal{A}})$ toma prestadas sus propiedades de $\mathcal{B} = (B, \rightarrow_{\mathcal{B}}, L_{\mathcal{B}})$ por medio de un morfismo tartamudo de sistemas de transiciones $h : (A, \rightarrow_{\mathcal{A}}) \longrightarrow (B, \rightarrow_{\mathcal{B}})$, entonces (η_{AP}, h) se convierte en un morfismo tartamudo estricto. Más aún, (η_{AP}, h) es un morfismo cartesiano para el funtor de olvido U : **KSMap**^{bool} $\longrightarrow$ **STSys** que lleva cada estructura de Kripke a su sistema de transiciones subyacente.*

Demostración. Claramente, (η_{AP}, h) es un morfismo tartamudo estricto porque por definición de $L_{\mathcal{A}}$ las proposiciones atómicas son preservadas.

Para mostrar que es un morfismo cartesiano, supongamos que $(\alpha, f) : C \longrightarrow \mathcal{B}$ es un morfismo tartamudo y $g : U(C) \longrightarrow (A, \rightarrow_{\mathcal{A}})$ un morfismo tartamudo de sistemas de transiciones tales que $f = h \circ g$. Tenemos que demostrar que existe una única simulación tartamuda (α', g') tal que $(\eta_{AP}, h) \circ (\alpha', g') = (\alpha, f)$ y $U(\alpha', g') = g$. El único candidato posible es (α, g) y lo que tenemos que comprobar es que $g : C \longrightarrow \mathcal{A}|_{\alpha}$ es realmente un AP'-morfismo tartamudo, donde AP' es el conjunto de proposiciones atómicas de C. Por hipótesis, g es un morfismo de sistemas de transiciones. Ahora, supongamos que $g(c) = a$ y $p \in L_{\mathcal{A}|_{\alpha}}(a)$; se sigue que $\mathcal{A}, a \models \alpha(p)$. Por la proposición 5.3 se tiene que $\mathcal{A}, a \models \varphi$ si y solo $\mathcal{B}, h(a) \models \varphi$ para todo $\varphi \in \text{Bool}(AP)$. Por lo tanto, $\mathcal{B}, h(a) \models \alpha(p)$ y como $f(c) = (h \circ g)(c) = h(a)$ y (α, f) es un morfismo tartamudo, $C, c \models p$ por el teorema 5.3, esto es, $p \in L_C(c)$ como se necesitaba. □

6.3.1 Préstamo entre teorías de reescritura

El préstamo al nivel de las teorías de reescritura se puede especificar como sigue. Inicialmente comenzamos con una teoría de reescritura $\mathcal{R}_1$ con una familia distinguida $J_1(State)$, un objeto $(\mathcal{R}_2, (\Sigma_2', E_2 \cup D_2), J_2)$ en $\mathbf{SRWTh}_{\models}$ y una función ecuacionalmente definida $h : T_{\Sigma_1/E_1, J_1(State)} \longrightarrow T_{\Sigma_2/E_2, J_2(State)}$ que da lugar a un morfismo tartamudo de sistemas de transiciones. Para extender la teoría $\mathcal{R}_1$ a un objeto en $\mathbf{SRWTh}_{\models}$ hemos de:

1. seleccionar una extensión conservadora (Ω, G) de $(\Sigma_1, E_1) \oplus (\Sigma_2', E_2 \cup D_2)$ en la que podamos definir ecuacionalmente h; y
2. extender esta teoría con un nuevo operador $\_ \models \_ : J_1(State)\ Prop \longrightarrow Bool$, donde $Prop$ es el tipo con el mismo nombre en Σ_2', y con la ecuación

$$(\forall x, y)\,(x \models y) = (h(x) \models y)\,.$$

Esto define un morfismo desde $(\mathcal{R}_1, (\Omega, G), J_1)$ en $(\mathcal{R}_2, (\Sigma_2', E_2 \cup D_2), J_2)$ en $\mathbf{SRWTh}_{\models}$.

6.4 Instituciones de la lógica temporal

El lector familiarizado con nociones categóricas seguramente haya notado que la proposición 5.1 en la página 112 posee un característico "sabor institucional". Ciertamente, las estructuras de Kripke se pueden organizar como los modelos de una institución para la lógica temporal en la que la proposición 5.1 corresponde al lema de satisfacción. Otras instituciones para lógicas temporales, concretamente para LTL y CTL*, fueron discutidas por Arrais y Fiadeiro (1996), pero tanto su noción de morfismo de signaturas como la de simulación (que corresponde aproximadamente a nuestra noción de morfismo bisimilar) son más limitados.

Volvamos entonces a nuestro propósito de definir una institución para la lógica temporal que utilice nuestras nociones de simulación y modelo. En primer lugar definimos la categoría de signaturas. Para ello, sea $\mathrm{State}\backslash\neg : \mathbf{Set} \longrightarrow \mathbf{Set}$ el funtor que lleva un conjunto (que se puede entender de proposiciones atómicas) AP al conjunto de fórmulas de estado $\mathrm{State}\backslash\neg(AP)$, y una función $\alpha : AP \longrightarrow AP'$ a su extensión homomórfica $\overline{\alpha} : \mathrm{State}\backslash\neg(AP) \longrightarrow \mathrm{State}\backslash\neg(AP')$. Se tiene entonces que la terna $\langle \mathrm{State}\backslash\neg, \eta, \mu \rangle$ es una *mónada*, donde $\eta : Id_{\mathbf{Set}} \Rightarrow \mathrm{State}\backslash\neg$ y $\mu : \mathrm{State}\backslash\neg \circ \mathrm{State}\backslash\neg \Rightarrow \mathrm{State}\backslash\neg$ son transformaciones naturales tales que $\eta_{AP}(p) = p$ y μ "desenvuelve" una fórmula en sus proposiciones atómicas básicas.

Nuestra categoría de signaturas será $\mathbf{Set}_{\mathrm{State}\backslash\neg}$, la *categoría de Kleisli* de la mónada. Sus objetos son simplemente conjuntos y las flechas $AP \longrightarrow AP'$ son funciones $\alpha : AP \longrightarrow \mathrm{State}\backslash\neg(AP')$. Con estos datos podemos ya definir la deseada institución.

Definición 6.2 *La institución de estructuras de Kripke,* $\mathcal{I}_{\mathbf{K}} = (\mathbf{Sign}_{\mathbf{K}}, sen_{\mathbf{K}}, \mathbf{Mod}_{\mathbf{K}}, \models)$, *viene dada por:*

- $\mathbf{Sign}_{\mathbf{K}} = \mathbf{Set}_{\mathrm{State}\backslash\neg}$.

- *$sen_{\mathbf{K}}$ es el funtor que lleva cada conjunto AP al conjunto* $\mathrm{State}\backslash\neg(AP)$ *y cada función* $\alpha : AP \longrightarrow \mathrm{State}\backslash\neg(AP')$ *a su extensión homomórfica* $\overline{\alpha} : \mathrm{State}\backslash\neg(AP) \longrightarrow \mathrm{State}\backslash\neg(AP')$.
- $\mathbf{Mod_K} : \mathbf{Set}_{\mathrm{State}\backslash\neg} \longrightarrow \mathbf{Cat}^{\mathrm{op}}$ *viene dada por* $\mathbf{Mod_K}(AP) = \mathbf{KSim}_{AP}$ *y, para* $\alpha : AP \longrightarrow AP'$ *en* $\mathbf{Set}_{\mathrm{State}\backslash\neg}$, *por* $\mathbf{Mod_K}(\alpha)(\mathcal{A}) = \mathcal{A}|_\alpha$ *y* $\mathbf{Mod_K}(\alpha)(H) = H$.
- *La relación de satisfacción se define como* $\mathcal{A} \models \varphi$ *si y solo si* $\mathcal{A}, a \models \varphi$ *para todo* $a \in A$.

Proposición 6.5 *$\mathcal{I}_{\mathbf{K}}$ es una institución.*

Demostración. Es un ejercicio rutinario comprobar que los llamados funtores efectivamente lo son. Por ejemplo, vamos a comprobar que $\mathbf{Mod_K}$ está bien definido. Dada $\alpha : AP \longrightarrow AP'$, $\mathbf{Mod_K}(\alpha)$ es un funtor. Está bien definido sobre los objetos y preserva las identidades y la composición: tan solo tenemos que ver si $\mathbf{Mod_K}(\alpha)(H) : \mathcal{A}|_\alpha \longrightarrow \mathcal{B}|_\alpha$ es una AP-simulación siempre que $H : \mathcal{A} \longrightarrow \mathcal{B}$ sea una AP'-simulación. Como los sistemas de transiciones no han cambiado, $\mathbf{Mod_K}(\alpha)(H)$ preserva la relación de transición. Ahora, si aHb y $p \in L_{\mathcal{B}|_\alpha}(b)$ se tiene por definición que $\mathcal{B}, b \models \alpha(p)$ y, por la proposición 5.2, $\mathcal{A}, a \models \alpha(p)$, lo que de nuevo por definición implica que $p \in L_{\mathcal{A}|_\alpha}(a)$ como se pedía. Así, $\mathbf{Mod_K}$ está bien definido tanto sobre los objetos como sobre las flechas. Claramente preserva las identidades por lo que tan solo resta demostrar que preserva la composición, para lo que resulta suficiente con mostrar que dadas dos flechas $\alpha : AP \longrightarrow AP'$ y $\beta : AP' \longrightarrow AP''$, y una estructura de Kripke $\mathcal{A}$ sobre AP'', se tiene $\mathcal{A}_{\beta\circ\alpha} = (\mathcal{A}|_\beta)|_\alpha$. La igualdad se cumple al nivel de los sistemas de transiciones de manera inmediata. Para la función de etiquetado, $p \in L_{\mathcal{A}|_{\beta\circ\alpha}}(a)$ si y solo si $\mathcal{A}, a \models \overline{\beta}(\alpha(p))$ (por definición) si y solo si $\mathcal{A}|_\beta \models \alpha(p)$ (por la proposición 5.1) si y solo si $p \in L_{(\mathcal{A}|_\beta)|_\alpha}(a)$ (por definición). Y el lema de satisfacción se sigue de la proposición 5.1. □

De manera análoga podríamos pensar en definir una institución que correspondiera a las estructuras de Kripke y las simulaciones estrictas. Sin embargo, tal y como ya ocurría en la sección 5.1, el hecho de que α pueda aplicar una proposición atómica en una fórmula de estado arbitraria hace que esto no sea posible. El problema ahora no es que las simulaciones estrictas no se puedan componer, sino que el funtor de modelos putativo no es tal: el reducto de una simulación estricta no tiene por qué ser estricta. Para verlo basta con considerar la función α y la AP-simulación g del ejemplo de la página 113; g es estricta pero $g|_\alpha$ no lo es. También de manera análoga a como sucedió en casos anteriores, para solucionar el problema y conseguir una institución para las simulaciones estrictas sería suficiente con restringir los morfismos de signatura a funciones de la forma $\alpha : AP \longrightarrow \mathrm{Bool}(AP)$.

Nótese que la categoría **KSim** se puede obtener por medio de la construcción de Grothendieck. En realidad, **KSim** no es nada más que la categoría de Grothendieck asociada a la categoría indexada $\mathbf{Mod_K}$. (Lo mismo ocurriría para las simulaciones estrictas si trabajáramos con las funciones α restringidas.) De manera similar, las categorías **KMap** y **KBSim** también se pueden obtener modificando $\mathbf{Mod_K}$ de manera que aplique AP en $\mathbf{KMap}_{AP}$ y $\mathbf{KBSim}_{AP}$, respectivamente.

Naturalmente, existen resultados análogos para el caso de las simulaciones tartamudas. Ahora el funtor utilizado para definir la categoría de Kleisli es $\mathrm{State}\backslash\{\neg, \mathbf{X}\}$, que lleva

AP al conjunto de fórmulas de estado State\{¬, **X**}(AP) (y Bool en el caso estricto). El funtor de modelos asigna al conjunto de proposiciones atómicas AP la correspondiente categoría de AP-simulaciones tartamudas, $\mathbf{KSSim}_{AP}$.

Las instituciones que acabamos de presentar hacen uso de la noción de morfismo de signaturas más general que es compatible con el hecho de que ciertas fórmulas temporales sean reflejadas. Pero precisamente por esta generalidad, la propiedad de *exactitud* no se cumple en ellas. Para verlo, es suficiente con considerar un conjunto de proposiciones atómicas $AP = \{p, q\}$ y morfismos de signaturas $\alpha_1, \alpha_2 : AP \longrightarrow \text{State}\backslash\neg(AP)$ tales que $\alpha_1(p) = p \wedge q$ y $\alpha_2(p) = p \vee q$. Entonces, para cualesquiera morfismos de signaturas $\beta_1, \beta_2 : AP \longrightarrow \text{State}\backslash\neg(AP')$, se tiene $(\beta \circ \alpha_1)(p) = \beta_1(p) \wedge \beta_1(q)$, que es distinto de $(\beta \circ \alpha_2)(p) = \beta_2(p) \vee \beta_2(q)$. Esto demuestra que $\mathbf{Sign}_{\mathbf{K}}$ no tiene sumas amalgamadas. Sin embargo podemos lograr que ello suceda, aunque para hacerlo hemos de imponer que los morfismos de signaturas se restrinjan a aplicar proposiciones atómicas en proposiciones atómicas; esta vez no sería suficiente con imponer que los morfismos devolvieran valores en Bool(AP).

Proposición 6.6 *Sea $\mathcal{I}'_{\mathbf{K}}$ obtenida a partir de la institución $\mathcal{I}_{\mathbf{K}}$ reemplazando $\mathbf{Set}_{\text{State}\backslash\neg}$ por* **Set** *como la categoría de signaturas. Entonces se tiene que $\mathcal{I}'_{\mathbf{K}}$ es una institución semiexacta.*

Demostración. Que $\mathcal{I}'_{\mathbf{K}}$ es una institución es inmediato y como la categoría de signaturas es **Set** sabemos que tiene sumas amalgamadas. Por lo tanto, nos queda la tarea de comprobar que aquellas son transformadas en productos fibrados (pullbacks) por el funtor de los modelos. Sea la suma amalgamada

$$\begin{array}{ccc} AP_0 & \xrightarrow{\quad\alpha_2\quad} & AP_2 \\ {\scriptstyle\alpha_1}\downarrow & & \downarrow{\scriptstyle\beta_2} \\ AP_1 & \xrightarrow{\quad\beta_1\quad} & AP_3 = (AP_1 \uplus AP_2)/\equiv \end{array}$$

donde $\equiv$ es la menor relación de equivalencia sobre la unión disjunta $AP_1 \uplus AP_2$ que satisface $\alpha_1(p) \equiv \alpha_2(p)$, y β_1 y β_2 llevan cada elemento a su clase de equivalencia. Para ver que es transformada en un producto fibrado, consideremos $F_1 : C \longrightarrow \mathbf{KSim}_{AP_1}$ y $F_2 : C \longrightarrow \mathbf{KSim}_{AP_2}$ funtores tales que $\_|_{\alpha_1} \circ F_1 = \_|_{\alpha_2} \circ F_2$; tenemos que encontrar un funtor único $F : C \longrightarrow \mathbf{KSim}_{AP_3}$ tal que $\_|_{\beta_1} \circ F = F_1$ y $\_|_{\beta_2} \circ F = F_2$.

Sea c un objeto en C y $f : c \longrightarrow c'$ una flecha, con $F_1(c) = \mathcal{A}$ y $F_2(c) = \mathcal{B}$. De la hipótesis se sigue que A es igual a B, $\rightarrow_{\mathcal{A}}$ es igual a $\rightarrow_{\mathcal{B}}$ y $F_1(f)$ es igual a $F_2(f)$. Esto nos lleva a definir $F(c) = (A, \rightarrow_{\mathcal{A}}, L_{F(c)})$ y $F(f) = F_1(f)$, donde decidimos elegir la función de etiquetado como $L_{F(c)} = \beta_1(L_{\mathcal{A}}) \cup \beta_2(L_{\mathcal{B}})$. Es inmediato comprobar que $F(f)$ es una AP_3-simulación, por lo que F está bien definido, y es un funtor porque F_1 (o F_2) lo es.

Nos queda tan solo comprobar que F satisface la condición de conmutatividad y demostrar que es el único funtor que lo hace. Para la primera parte, nótese que por definición de suma amalgamada no es posible tener dos proposiciones p y p' en AP_1 tales que $\beta_1(p) = \beta_1(p')$ y $p \in L_{\mathcal{A}}(a)$ pero $p' \notin L_{\mathcal{A}}(a)$ (una demostración detallada procedería por inducción sobre la definición de $\equiv$). Usararemos esta propiedad para mostrar que

$F(c)|_{\beta_1} = F_1(c)$. Ya sabemos que sus objetos y las relaciones de transición son las mismas; en cuanto a las proposiciones atómicas:

$$p \in L_{F(c)|_{\beta_1}}(a) \iff F(c), a \models \beta_1(p) \iff \beta_1(p) \in L_{F(c)}(a) \iff p \in L_{F_1(c)}(a)\,,$$

donde hemos utilizado la propiedad anterior para probar la última implicación hacia la derecha. El resultado para F_2 es simétrico. La unicidad se sigue de las definiciones de los funtores y las equivalencias previas. □

El mismo resultado se aplica a toda la familia de instituciones descrita más arriba; en particular, a las construcciones comentadas para las simulaciones estrictas. Señalemos también que ni siquiera estas instituciones restringidas, en las que los morfismos de signaturas solo pueden devolver proposiciones, son liberales. Sin embargo, los funtores de olvido **Mod**(α) tienen adjunto por la izquierda para todo morfismo de signaturas α, que aplique cada estructura de Kripke $\mathcal{A}$ en $\mathcal{A}^* = (A, \to_{\mathcal{A}}, L_{\mathcal{A}^*})$, con $L_{\mathcal{A}^*}(a) = \{q \mid$ si p es tal que $\alpha(p) = q$, entonces $p \in L_{\mathcal{A}}(a)\}$, y cada simulación en sí misma.

6.5 Productos

Proposición 6.7 *Para todos los conjuntos de proposiciones atómicas AP, la categoría* $\mathbf{KMap}_{AP}$ *tiene productos finitos.*

Demostración. Dadas dos estructuras de Kripke $\mathcal{A}$ y $\mathcal{B}$, definimos $\mathcal{A} \times \mathcal{B} = (A \times B, \to_{\mathcal{A}\times\mathcal{B}}, L_{\mathcal{A}\times\mathcal{B}})$, donde $(a, b) \to_{\mathcal{A}\times\mathcal{B}} (a', b')$ si y solo si $a \to_{\mathcal{A}} a'$ y $b \to_{\mathcal{B}} b'$, y $L_{\mathcal{A}\times\mathcal{B}}(a, b) = L_{\mathcal{A}}(a) \cup L_{\mathcal{B}}(b)$, con las proyecciones habituales $\pi_{\mathcal{A}} : \mathcal{A} \times \mathcal{B} \longrightarrow \mathcal{A}$ y $\pi_{\mathcal{B}} : \mathcal{A} \times \mathcal{B} \longrightarrow \mathcal{B}$. La relación $\to_{\mathcal{A}\times\mathcal{B}}$ es total de manera que $\mathcal{A} \times \mathcal{B}$ está bien definida y es inmediato comprobar que $\pi_{\mathcal{A}}$ y $\pi_{\mathcal{B}}$ son AP-morfismos.

Ahora, si $f : C \longrightarrow \mathcal{A}$ y $g : C \longrightarrow \mathcal{B}$ son AP-morfismos, la única flecha $\langle f, g\rangle : C \longrightarrow \mathcal{A}\times\mathcal{B}$ tal que $\pi_{\mathcal{A}} \circ \langle f, g\rangle = f$ y $\pi_{\mathcal{B}} \circ \langle f, g\rangle = g$ viene dada por $\langle f, g\rangle(c) = (f(c), g(c))$. La unicidad está clara; tenemos que comprobar que $\langle f, g\rangle$ es realmente un AP-morfismo. Si se tiene que $c \to_C c'$ entonces $f(c) \to_{\mathcal{A}} f(c')$ y $g(c) \to_{\mathcal{B}} g(c')$, y por lo tanto $\langle f, g\rangle(c) \to_{\mathcal{A}\times\mathcal{B}} \langle f, g\rangle(c')$. Y si $p \in L_{\mathcal{A}\times\mathcal{B}}(\langle f, g\rangle(c))$ se sigue que $p \in L_{\mathcal{A}}(f(c))$ o $p \in L_{\mathcal{B}}(g(c))$: en cualquier caso, $p \in L_C(c)$. □

Esta construcción es lo que en ocasiones se llama en la literatura el *producto síncrono* de estructuras de Kripke. Nótese que esta construcción se puede extender a productos infinitos de la forma esperada y que, como la estructura de Kripke con un único estado * y transición $* \longrightarrow *$, y etiquetado $L(*) = \emptyset$ es un objeto final en la categoría, resulta que $\mathbf{KMap}_{AP}$ tiene productos arbitrarios.

Este resultado sigue siendo cierto para la categoría de AP-morfismos estrictos, pero la construcción es ligeramente más complicada. El objeto final en $\mathbf{KMap}^{\mathrm{str}}_{AP}$ es $(\mathcal{P}(AP), \mathcal{P}(AP)\times\mathcal{P}(AP), id_{\mathcal{P}(AP)})$. La construcción de productos finitos se detalla en la demostración de la siguiente proposición.

Proposición 6.8 *Para todos los conjuntos de proposiciones atómicas AP, la categoría* $\mathbf{KMap}^{\mathrm{str}}_{AP}$ *tiene productos finitos.*

Demostración. Dadas dos estructuras de Kripke $\mathcal{A}$ y $\mathcal{B}$, sea $\mathcal{A} \times \mathcal{B}$ la estructura de Kripke definida en la demostración de la proposición 6.7. Definamos ahora

$$\text{Path}(\mathcal{A} \times \mathcal{B})^{=} = \{\pi \in \text{Path}(\mathcal{A} \times \mathcal{B}) \mid \text{para todo } i,\, L_{\mathcal{A}}(\pi_{\mathcal{A}}(\pi(i))) = L_{\mathcal{B}}(\pi_{\mathcal{B}}(\pi(i)))\}\,.$$

Entonces, si

$$D = \{(a,b) \mid \text{existe } \pi \in \text{Path}(\mathcal{A} \times \mathcal{B})^{=} \text{ e } i \in \mathbb{N} \text{ tales que } (a,b) = \pi(i)\}\,,$$

el producto de $\mathcal{A}$ y $\mathcal{B}$ en $\mathbf{KMap}^{\text{str}}_{AP}$ viene dado por $\mathcal{A} \times^{\text{st}} \mathcal{B} = (D, \to_{\mathcal{A}\times\mathcal{B}}|_{D^2}, L_{\mathcal{A}\times\mathcal{B}}|_D)$ con las proyecciones esperadas. Por construcción, $\to_{\mathcal{A}\times\mathcal{B}}|_{D^2}$ es total. Nótese que para AP-morfismos arbitrarios $f : C \longrightarrow \mathcal{A}$ y $g : C \longrightarrow \mathcal{B}$, la función $\langle f, g\rangle : C \longrightarrow \mathcal{A} \times^{\text{st}} \mathcal{B}$ está bien definida. Para cada $c \in C$, sea π un camino con $\pi(0) = c$ (debe existir porque $\to_C$ es total). Como tanto f como g son estrictos, $L_{\mathcal{A}}(f(\pi(i))) = L_{\mathcal{B}}(g(\pi(i)))$ y el camino

$$(f(\pi(0)), g(\pi(0))) \to_{\mathcal{A}\times\mathcal{B}} (f(\pi(1)), g(\pi(1))) \to_{\mathcal{A}\times\mathcal{B}} \cdots$$

pertenece a $\text{Path}(\mathcal{A} \times \mathcal{B})^{=}$; así, $(f(c), g(c)) \in D$. Y $\langle f, g\rangle$ es claramente estricto. □

Es interesante señalar que en algunos casos podemos tener $\mathcal{A} \times^{\text{str}} \mathcal{B} = \emptyset$ aunque ni $\mathcal{A}$ ni $\mathcal{B}$ sean vacíos, pero eso solo significa que la *única* estructura de Kripke C desde la que existen AP-morfismos tanto hacia $\mathcal{A}$ como hacia $\mathcal{B}$ es la estructura vacía. Notemos también que la construcción se puede extender a productos infinitos en la manera esperada.

Si miramos qué ocurre cuando consideramos AP-simulaciones en lugar de solo morfismos, resulta que en $\mathbf{KSim}_{AP}$ también existen productos finitos aunque su definición es bastante distinta de las anteriores.

Proposición 6.9 *Para todos los conjuntos de proposiciones atómicas AP, la categoría* $\mathbf{KSim}_{AP}$ *tiene productos finitos.*

Demostración. Definimos el producto de $\mathcal{A}$ y $\mathcal{B}$ como $\mathcal{A} \times \mathcal{B} = (A \uplus B, \to_{\mathcal{A}} \uplus \to_{\mathcal{B}}, L_{\mathcal{A}\times\mathcal{B}})$, donde $L_{\mathcal{A}\times\mathcal{B}}(x)$ es $L_{\mathcal{A}}(x)$ si $x \in A$ o $L_{\mathcal{B}}(x)$ si $x \in B$, con proyecciones $\Pi_{\mathcal{A}}$ y $\Pi_{\mathcal{B}}$ definidas mediante $a\Pi_{\mathcal{A}}a$ para todo $a \in A$ y $b\Pi_{\mathcal{B}}b$ para todo $b \in B$. Entonces, para AP-simulaciones $F : C \longrightarrow \mathcal{A}$ y $G : C \longrightarrow \mathcal{B}$, la única AP-simulación $\langle F, G\rangle$ está definida por $c\langle F, G\rangle a$ si y solo si cFa, y $c\langle F, G\rangle b$ si y solo si cGb. □

De nuevo, la construcción que se acaba de describir se puede extender a familias arbitrarias de estructuras de Kripke y como la estructura de Kripke vacía es un objeto final de manera trivial, la categoría $\mathbf{KSim}_{AP}$ tiene productos arbitrarios.

Finalmente, y a diferencia de lo que hemos visto en todos los casos anteriores, en la categoría $\mathbf{KSMap}_{AP}$ de AP-simulaciones tartamudas no existen productos en general. Para verlo, consideremos las estructuras de Kripke $\mathcal{A}$ y $\mathcal{B}$ determinadas respectivamente por las relaciones de transición $a_1 \to_{\mathcal{A}} a_2 \to_{\mathcal{A}} \cdots$ y $b_1 \to_{\mathcal{B}} b_2 \to_{\mathcal{B}} \cdots$ y con funciones de etiquetado vacías. Consideremos ahora una tercera estructura C dada por $c_1 \to_C c_2 \to_C \cdots$, y la AP-simulación tartamuda $f : C \longrightarrow \mathcal{A}$ tal que $f(c_1) = a_1$, $f(c_{2*i}) = a_{i+1}$ y $f(c_{2*i+1}) = a_{i+1}$ para $i \geq 1$, y $g : C \longrightarrow \mathcal{B}$ tal que $g(c_{2*i+1}) = a_{i+1}$ y $g(c_{2*i+2}) = a_{i+1}$ para $i \geq 0$. Supongamos que

$\mathcal{D}$ fuera el producto de $\mathcal{A}$ y $\mathcal{B}$ con proyecciones $\pi_{\mathcal{A}}$ y $\pi_{\mathcal{B}}$, y sea $d_1 \to_{\mathcal{D}} d_2 \to_{\mathcal{D}} \cdots$ el camino en $\mathcal{D}$ que $\langle f, g\rangle$-encaja con el camino en C que comienza en c_1. Tenemos $\langle f, g\rangle(c_1) = d_1$, $\pi_{\mathcal{A}}(d_1) = a_1$ y $\pi_{\mathcal{B}}(d_1) = b_1$. Ahora, $\langle f, g\rangle(c_2)$ debe ser igual a d_1 o a d_2. Pero la primera alternativa no se puede cumplir porque se tendría que $\pi_{\mathcal{A}}(\langle f, g\rangle(c_2)) \neq f(c_2)$; por lo tanto $\langle f, g\rangle(c_2) = d_2$, y $\pi_{\mathcal{A}}(d_2)$ y $\pi_{\mathcal{B}}(d_2)$ tienen que ser a_2 y b_1, respectivamente. Y hemos terminado, porque si intercambiamos las definiciones de f y g el mismo argumento conduce a que $\pi_{\mathcal{A}}(d_1) = \pi_{\mathcal{B}}(d_2) = a_1$, lo que supone una contradicción.

6.6 Coproductos

Todas las categorías mencionadas en la sección anterior tienen coproductos y su definición es la misma en todos los casos. Aquí presentamos los detalles para $\mathbf{KSim}_{AP}$.

Proposición 6.10 *Para todos los conjuntos de proposiciones atómicas AP, la categoría* $\mathbf{KSim}_{AP}$ *tiene coproductos finitos.*

Demostración. Dadas dos estructuras de Kripke $\mathcal{A}$ y $\mathcal{B}$, definimos $\mathcal{A} + \mathcal{B}$ como $(A \uplus B, \to_{\mathcal{A}} \uplus \to_{\mathcal{B}}, L_{\mathcal{A}+\mathcal{B}})$, donde $L_{\mathcal{A}+\mathcal{B}}(x)$ es $L_{\mathcal{A}}(x)$ si $x \in A$ o $L_{\mathcal{B}}(x)$ si $x \in B$, y con inclusiones $I_{\mathcal{A}}$ y $I_{\mathcal{B}}$ definidas mediante $aI_{\mathcal{A}}a$ para todo $a \in A$ y $bI_{\mathcal{B}}b$ para todo $b \in B$. $\mathcal{A} + \mathcal{B}$ está claramente bien definida y resulta trivial comprobar que $I_{\mathcal{A}}$ y $I_{\mathcal{B}}$ son AP-simulaciones. Ahora, para $F : \mathcal{A} \longrightarrow C$ y $G : \mathcal{B} \longrightarrow C$ AP-simulaciones arbitrarias, definimos $[F, G] : \mathcal{A} + \mathcal{B}$ mediante $a[F, G]c$ si y solo si aFc, y $b[F, G]c$ si y solo si bGc. Se comprueba fácilmente que $[F, G]$ así definida es la única AP-simulación que satisface $I_{\mathcal{A}} \circ [F, G]$ y $I_{\mathcal{B}} \circ [F, G]$. □

Nótese que la estructura de Kripke $\mathcal{A} + \mathcal{B}$ es la misma que la estructura de Kripke $\mathcal{A} \times \mathcal{B}$ de la proposición 6.9 y que la construcción también se puede aplicar a familias infinitas. El objeto inicial corresponde a la estructura de Kripke vacía.

6.7 Igualadores

Proposición 6.11 *Para todos los conjuntos de proposiciones atómicas AP, la categoría* $\mathbf{KMap}_{AP}$ *tiene igualadores.*

Demostración. Sean $f, g : \mathcal{A} \longrightarrow \mathcal{B}$ dos AP-morfismos y definamos

$$\mathrm{Path}(\mathcal{A})_{f,g} = \{\pi \in \mathrm{Path}(\mathcal{A}) \mid f \circ \pi = g \circ \pi\}.$$

Entonces, si

$$E = \{a \in A \mid \text{existen } \pi \in \mathrm{Path}(\mathcal{A})_{f,g} \text{ e } i \in \mathbb{N} \text{ tales que } a = \pi(i)\},$$

el igualador de f y g viene dado por la estructura de Kripke $\mathcal{E} = (E, \to_{\mathcal{A}}|_{E^2}, L_{\mathcal{A}}|_E)$ y la inclusión $e : \mathcal{E} \longrightarrow \mathcal{A}$. Por definición, la relación $\to_{\mathcal{E}}$ es total con lo que $\mathcal{E}$ es una estructura de Kripke bien definida; e es trivialmente un AP-morfismo (estricto). Supongamos ahora que $h : \mathcal{D} \longrightarrow \mathcal{A}$ es un AP-morfismo tal que $f \circ h = g \circ h$, y definamos $m : \mathcal{D} \longrightarrow \mathcal{E}$

mediante $m(d) = h(d)$. Obviamente $f(h(d)) = g(h(d))$ y, como $\rightarrow_{\mathcal{D}}$ es total, existe un camino π en $\mathcal{D}$ tal que $\pi(0) = d$: su imagen por h pertenece a $\mathrm{Path}(\mathcal{A})_{f,g}$ y por lo tanto $h(d) \in E$ y m está bien definido. Es claro que m es único y que $h = e \circ m$. Finalmente, m es un AP-morfismo: si $d \rightarrow_{\mathcal{D}} d'$ entonces $h(d) \rightarrow_{\mathcal{A}} h(d')$ y por definición de m y $\rightarrow_{\mathcal{E}}$ se tiene que $m(d) \rightarrow_{\mathcal{E}} m(d')$; y si $p \in L_{\mathcal{E}}(m(d))$ entonces $p \in L_{\mathcal{A}}(h(d))$ de donde se sigue que $p \in L_{\mathcal{D}}(d)$. □

Es fácil comprobar que la misma construcción también devuelve igualadores en las categorías $\mathbf{KMap}^{\mathrm{str}}_{AP}$ and $\mathbf{KSMap}_{AP}$. En cuanto a $\mathbf{KSim}_{AP}$, por el momento no hemos sido capaces de demostrar o refutar la existencia en general de igualadores.

6.8 Coigualadores

Proposición 6.12 *Para todos los conjuntos de proposiciones atómicas AP, la categoría* $\mathbf{KMap}_{AP}$ *tiene coigualadores.*

Demostración. Supongamos que $f, g : \mathcal{A} \longrightarrow \mathcal{B}$ son AP-simulaciones y definamos $\equiv$ como la menor relación de equivalencia sobre B que contiene $\{(f(a), g(a)) \mid a \in A\}$. Entonces el coigualador de f y g viene dado por la estructura de Kripke cociente $\mathcal{B}/\equiv$ y la proyección $c : \mathcal{B} \longrightarrow \mathcal{B}/\equiv$. Para verlo, sea $h : \mathcal{B} \longrightarrow \mathcal{D}$ un AP-morfismo tal que $h \circ f = h \circ g$; podemos definir $m : \mathcal{B}/\equiv \longrightarrow \mathcal{D}$ mediante $m([b]) = h(b)$ donde $h = m \circ c$. Ahora tenemos que comprobar que m está bien definido y que es un AP-morfismo. La primera parte se prueba mostrando que si $b_1 \equiv b_2$ entonces $h(b_1) = h(b_2)$, por inducción sobre la definición de $\equiv$. El caso base corresponde a $f(a) \equiv g(a)$, y por hipótesis se tiene que $h(f(a)) = h(g(a))$. Y es inmediato darse cuenta de que el resultado también se cumple para $b \equiv b$, para $b_2 \equiv b_1$ si se cumple para $b_1 \equiv b_2$, y para $b_1 \equiv b_3$ si se cumple para $b_1 \equiv b_2$ y $b_2 \equiv b_3$. Para la segunda parte, si $[b_1] \rightarrow_{\mathcal{B}/\equiv} [b_2]$ debe de tenerse $b'_1 \rightarrow_{\mathcal{B}} b'_2$ para algún $b'_1 \equiv b_1$ y $b'_2 \equiv b_2$ con lo que $h(b'_1) \rightarrow_{\mathcal{D}} h(b'_2)$. Y si $p \in L_{\mathcal{D}}(m([b]))$ entonces $p \in L_{\mathcal{B}}(b')$ para todo $b' \in [b]$ y por lo tanto $p \in L_{\mathcal{B}/\equiv}([b])$. □

De nuevo, esta construcción también se puede realizar en la categoría $\mathbf{KMap}^{\mathrm{str}}_{AP}$, pero no sabemos lo que ocurre ni en $\mathbf{KSim}_{AP}$ ni en $\mathbf{KSMap}_{AP}$.

6.9 Epis y monos

En esta sección caracterizamos las clases de AP-simulaciones que corresponden a los epimorfismos y monomorfismos (regulares).

Proposición 6.13 *Una flecha en* $\mathbf{KMap}_{AP}$, $\mathbf{KMap}^{\mathrm{str}}_{AP}$ *o* $\mathbf{KSMap}_{AP}$ *es un epimorfismo si y solo si es una función sobreyectiva.*

Demostración. Supongamos que $f : \mathcal{A} \longrightarrow \mathcal{B}$ es sobreyectiva. Tenemos que si $g \circ f = h \circ f$ entonces debe tenerse $g = h$, de donde se sigue que f es epi porque el rango de f es B.

En la otra dirección, supongamos ahora que f es un epimorfismo. Si f no fuera sobreyectiva existiría un elemento $b \in B$ que no estaría en la imagen de f. Definimos una

estructura de Kripke $\mathcal{B}'$ que es como $\mathcal{B}$ pero con b sustituida por b_1 y b_2 con el mismo valor para la función de etiquetado que b y tales que $b_i \to_{\mathcal{B}'} b'$ si y solo $b \to_{\mathcal{B}} b'$ y $b' \to_{\mathcal{B}'} b_i$ si y solo si $b' \to_{\mathcal{B}} b$. Ahora, si $g : \mathcal{B} \longrightarrow \mathcal{B}'$ lleva b a b_1 y los demás elementos a sí mismos, y si $h : \mathcal{B} \longrightarrow \mathcal{B}'$ lleva b a b_2 y es la identidad para el resto de elementos, tenemos que g y h son AP-morfismos (estrictos/tartamudos) bien definidos, $g \circ f = h \circ f$, pero $g \neq h$: contradiciendo la hipótesis de que f fuera un epimorfismo. □

Proposición 6.14 *Una flecha en* $\mathbf{KMap}_{AP}$, $\mathbf{KMap}^{\mathrm{str}}_{AP}$ *o* $\mathbf{KSMap}_{AP}$ *es un monomorfismo si y solo si es inyectiva sobre caminos.*

Demostración. Supongamos que $f : \mathcal{A} \longrightarrow \mathcal{B}$ es inyectiva sobre caminos, es decir, la función f que va de $\mathrm{Path}(\mathcal{A})$ en $\mathrm{Path}(\mathcal{B})$ definida mediante $f(\pi) = f \circ \pi$ es inyectiva. Sean $g, h : \mathcal{C} \longrightarrow \mathcal{A}$ flechas tales que $f \circ g = f \circ h$, $c \in C$ y π un camino que empiece en c: se tiene que $f(g(\pi)) = f(h(\pi))$ de donde se sigue que $g(\pi) = h(\pi)$ y por lo tanto $g(c) = h(c)$.

En el otro sentido, supongamos que f es mono pero que existen caminos π y π' en $\mathcal{A}$ tales que $f(\pi) = f(\pi')$ y $\pi \neq \pi'$. Definamos una estructura de Kripke $\mathcal{C}$ con un único camino $c_1 \to_{\mathcal{C}} c_2 \to_{\mathcal{C}} \cdots$ y con función de etiquetado $L_{\mathcal{C}}(c_i) = L_{\mathcal{A}}(\pi(i)) \cup L_{\mathcal{A}}(\pi'(i))$. Entonces, si $g, h : \mathcal{C} \longrightarrow \mathcal{A}$ se definen mediante $f(c_i) = \pi(i)$ y $g(c_i) = \pi'(i)$, se tiene que g y h son AP-morfismos por construcción (y estrictos, si f lo es) con $f \circ g = f \circ h$ y $g \neq h$, lo que supone una contradicción. □

La caracterización de los monomorfismos regulares resulta ahora inmediata.

Proposición 6.15 *Una flecha* $f : \mathcal{A} \longrightarrow \mathcal{B}$ *es mono regular en las categorías* $\mathbf{KMap}_{AP}$, $\mathbf{KMap}^{\mathrm{str}}_{AP}$ *o* $\mathbf{KSMap}_{AP}$ *si y solo si* f *es inyectiva,* $L_{\mathcal{A}}(a) = L_{\mathcal{B}}(f(a))$ *para todo* $a \in A$, *y* $a \to_{\mathcal{A}} a'$ *si y solo si* $f(a) \to_{\mathcal{B}} f(a')$.

Demostración. La implicación de izquierda a derecha se sigue de la construcción en la demostración de la proposición 6.11.

En el otro sentido, sea $\mathcal{C}$ la estructura de Kripke obtenida a partir de $\mathcal{B}$ partiendo cada estado b en b_1 y b_2, con función de etiquetado $L_{\mathcal{C}}(b_1) = L_{\mathcal{C}}(b_2) = L_{\mathcal{B}}(b)$ y transiciones $b_i \to_{\mathcal{C}} b'_j$ si y solo si $b \to_{\mathcal{B}} b'$. Ahora, definamos $f, g : \mathcal{B} \longrightarrow \mathcal{B}'$ mediante $g(b) = b_1$ y $h(b) = b_1$ si $b \in f(A)$ y $h(b) = b_2$ en caso contrario: f y g así definidas son AP-morfismos (estrictos) y, como $f(\mathcal{A})$ es una subestructura de Kripke de $\mathcal{B}$ isomorfa a $\mathcal{A}$ debido a las suposiciones, se comprueba fácilmente que el resultado de la construcción del igualador en la proposición 6.11 es (isomorfo a) f. □

Existe también una caracterización de los epimorfismos regulares, bastante más complicada, que se describe en la demostración de la siguiente proposición.

Proposición 6.16 *Una flecha* $f : \mathcal{A} \longrightarrow \mathcal{B}$ *es epi regular en las categorías* $\mathbf{KMap}_{AP}$ *o* $\mathbf{KMap}^{\mathrm{str}}_{AP}$ *si y solo si:*

1. $f(a) = f(a')$ *implica que existen caminos* π *y* π' *tales que* $\pi(0) = a$, $\pi'(0) = a'$ *y* $f(\pi) = f(\pi')$;

2. *si* $b \to_{\mathcal{B}} b'$ *entonces existen* $a, a' \in A$ *que verifican* $f(a) = b$, $f(a') = b'$ *y* $a \to_{\mathcal{A}} a'$;

3. *para todo* b *se cumple* $L_{\mathcal{B}}(b) = \bigcap_{f(a)=b} L_{\mathcal{A}}(a)$.

Demostración. La implicación de izquierda a derecha se sigue de la proposición 6.12; el punto (2) se demuestra por inducción sobre $\equiv$.

En el otro sentido, definimos una estructura de Kripke C y dos AP-morfismos $g, h : C \longrightarrow \mathcal{A}$ como sigue. Para cada par de estados a, a' tales que $f(a) = f(a')$ consideremos los caminos π y π' que (1) nos facilita. Ahora, añadimos a C un nuevo camino ρ que verifique $g(\rho(i)) = \pi(i)$ y $h(\rho(i)) = \pi'(i)$. Entonces, si aplicamos la construcción en la proposición 6.12 que devuelve el coigualador de g y h utilizando una estructura de Kripke cociente $\mathcal{A}/\equiv$, el resultado que se obtiene es, por los puntos (2) y (3), isomorfo a $f : \mathcal{A} \longrightarrow \mathcal{B}$. □

6.10 Factorizaciones

Proposición 6.17 $\mathbf{KMap}_{AP}$, $\mathbf{KMap}^{\mathrm{str}}_{AP}$ *y* $\mathbf{KSMap}_{AP}$ *son categorías (epi, mono regular).*

Demostración. Sea $f : \mathcal{A} \longrightarrow \mathcal{B}$ una flecha en alguna de estas categorías y denotemos por $f(\mathcal{A})$ la estructura de Kripke $(f(A), \to_{\mathcal{B}}|_{f(A)}, L_{\mathcal{B}}|_{f(A)})$. Definimos entonces $e : \mathcal{A} \longrightarrow f(\mathcal{A})$ a partir de f simplemente tomando el conjunto $f(A)$ como su imagen, y $m : f(\mathcal{A}) \longrightarrow \mathcal{B}$ como la inclusión obvia: (e, m) es la única factorización (epi, mono regular) de f (salvo isomorfismo).

Por las proposiciones 6.13 y 6.15, e y m son ciertamente epi y mono regular, respectivamente. Supongamos ahora que $e' : \mathcal{A} \longrightarrow C$, $m' : C \longrightarrow \mathcal{B}$ es otra factorización (epi, mono regular) de f. Definimos $g : f(\mathcal{A}) \longrightarrow C$ como $g(b) = e'(a)$ donde $e(a) = b$ (recordemos que e es sobreyectiva) y $h : C \longrightarrow f(\mathcal{A})$ mediante $h(c) = e(a)$ donde $e'(a) = c$. Comprobemos que están bien definidos. Si $e(a) = e(a')$ entonces $m(e(a)) = m(e(a'))$ y por lo tanto $m'(e'(a)) = m'(e'(a'))$; ahora, como m' es mono regular, $e'(a) = e'(a')$ y g está bien definido, y de manera análoga tenemos lo mismo para h. También es claro que cada uno es inverso del otro y que $g \circ e = e'$ y $m' \circ g$, por lo que tan solo nos queda por comprobar que son simulaciones; aquí damos los argumentos para g: los necesarios para h son simétricos. Si $b \to_{f(\mathcal{A})} b'$, donde $e(a) = b$ y $e(a') = b'$, entonces $m(e(a)) \to_{\mathcal{B}} m(e(a'))$ o, de manera equivalente, $m'(e'(a)) \to_{\mathcal{B}} m'(e'(a'))$ y por lo tanto, como m' es mono regular, $e'(a) \to_{\mathcal{B}} e'(a')$ y se tiene que $g(a) \longrightarrow_{f(\mathcal{A})} g(a')$. En el caso de las simulaciones tartamudas, un camino π en $f(\mathcal{A})$ se convierte en un camino $m(\pi)$ en $\mathcal{B}$ que m'-encaja con ρ en C; este mismo ρ también h-encaja con π. Finalmente,

$$\begin{aligned} L_{f(\mathcal{A})}(b) &= L_{\mathcal{B}}(e(a)) \\ &= L_{\mathcal{B}}(m(e(a))) \\ &= L_{\mathcal{B}}(m'(e'(a))) \\ &= L_{C}(e'(a)) \\ &= L_{C}(g(b)) \end{aligned}$$

La primera igualdad asume que $b = e(a)$ y la segunda y la cuarta se satisfacen porque m y m' son monos regulares. □

Aunque no tenemos ningún contraejemplo, creemos que en general no existen factorizaciones (epi regular, mono). Sin embargo cuando las hay son únicas: para probarlo basta con aplicar un argumento similar al de las factorizaciones (epi, mono regular).

Capítulo 7

Un prototipo para la abstracción de predicados

A lo largo de buena parte de esta tesis, el hilo conductor que ha venido guiando todo su desarrollo podría resumirse en tres puntos fundamentales:

- que la lógica de reescritura ofrece un marco muy flexible para la especificacioń de sistemas concurrentes;
- que el diseño de tales sistemas es complicado y tendente a errores; y
- que para abordar la inevitable tarea de comprobar que el sistema satisface las propiedades deseadas una posible alternativa consiste en el uso de abstracciones.

En los capítulos 4 y 5, y de forma algo más abstracta en el capítulo 6, se ha desarrollado toda una teoría sobre la especificación de sistemas en lógica de reescritura, su relación con las estructuras de Kripke y la representación de simulaciones. Sin embargo todo este desarrollo exige todavía bastante esfuerzo por parte del usuario a la hora de llevarlo a la práctica. Aunque este es menor en el caso de las abstracciones ecuacionales, donde las abstracciones son correctas por construcción y solo hay que comprobar que los requisitos de ejecutabilidad se satisfacen, en el caso de las simulaciones generales del capítulo 5 es el usuario quien tiene que hacer todo el trabajo de definir la simulación y comprobar que es correcta. Ciertamente este enfoque es el más flexible y potente posible, pero en muchas ocasiones uno cedería alegremente parte de esa flexibilidad y potencia a cambio de un mayor automatismo que signifique menos trabajo por nuestra parte.

El objetivo de este capítulo es el dar al menos los primeros pasos para tratar de satisfacer tales deseos. Para ello se va a considerar la técnica de abstracción más popular, la *abstracción de predicados*, y se va a describir un prototipo que la implementa en el lenguaje Maude.

7.1 ¿Qué es la abstracción de predicados?

Recordemos que en la sección 5.7 se presentó la abstracción de predicados como ejemplo para ilustrar el uso de morfismos teoroidales en la descripción de simulaciones. Ya allí se dijo que la especificación resultante no sería ejecutable en general y que, para conseguir una que sí lo fuera, el enfoque seguido por muchos investigadores era construir una aproximación más sencilla de computar. En las siguientes secciones describiremos cómo el cómputo de una de esas aproximaciones ha sido implementado en Maude; en la presente nos limitaremos a repasar y echar un vistazo algo más detallado a la técnica en cuestión.

La abstracción de predicados es una técnica para computar automáticamente abstracciones finitas de sistemas complejos. Dado un sistema de transiciones con estados $a, b, c, \ldots$, pertenecientes a un conjunto S y con transiciones dadas por una relación de transición $\rightarrow \subseteq S \times S$, una abstracción de predicados queda definida por un conjunto de predicados $\phi_1, \ldots, \phi_n$ sobre los estados, de la siguiente forma:

- El conjunto de estados del sistema abstracto es el conjunto de n-tuplas de valores booleanos, donde n es el número de predicados.
- Cada estado concreto a es aplicado en la tupla $\alpha(a) = \langle \phi_1(a), \ldots, \phi_n(a) \rangle$.

La relación de transición en el sistema abstracto queda entonces determinada en la manera estándar (definición 4.5):

- Existe una transición desde el estado abstracto $\langle b_1, \ldots, b_n \rangle$ a $\langle b'_1, \ldots, b'_n \rangle$ si y solo si hay estados concretos a y b tales que $\alpha(a) = \langle b_1, \ldots, b_n \rangle$, $\alpha(b) = \langle b'_1, \ldots, b'_n \rangle$ y $a \rightarrow b$.

Computar el estado abstracto asociado a uno concreto a, aunque puede depender de la complejidad de los predicados, suele ser inmediato. Es en la computación de la relación de transición abstracta donde el reside la dificultad, puesto que saber si podemos dar un único paso requiere potencialmente inspeccionar un número infinito de estados concretos.

En la práctica existen dos enfoques para la construcción de la relación de transición del sistema abstracto. En la primera de ellas, que es la que nosotros seguiremos, cada predicado o regla (dependiendo de la lógica que se haya usado para especificar el sistema) que interviene en la definición de la relación de transición es directamente transformado en un predicado o regla diferente que a su vez formará parte de la definición de la relación de transición en el sistema abstracto. Esta relación normalmente no coincide con la relación de transición abstracta tal y como se ha definido más arriba, pero se trata de una aproximación suficientemente buena. En la segunda aproximación, llamada *abstracción de predicados implicita* en Das (2003), el sistema de transiciones abstracto no es computado explícitamente; en su lugar, se abstrae el estado inicial y todos los estados alcanzables desde él son computados.

7.2 ¿Cómo se abstraen las reglas?

Para ilustrar el funcionamiento del método, y el de la abstracción de predicados en general, vamos a utilizar el siguiente ejemplo adaptado de la tesis de Das (2003):

```
mod SD-EXAMPLE is
  protecting NAT .

  sort Config .
  subsort Nat < Config .

  op init : -> Config .
  eq init = 0 .

  var N : Nat .

  crl N => s(N) if (N < 10) = true .
  rl N => N * 2 .
  rl s(s(N)) => N .
endm
```

Este módulo especifica un sistema con los números naturales como conjunto de estados y con un número infinito de estados alcanzables a partir del inicial. (El tipo `Config` y el operador `init` son necesarios porque la herramienta que describimos en este capítulo asume que estos son los nombres del tipo de los estados y del estado inicial, respectivamente.) Precisamente esta infinitud es la que impide utilizar la comprobación de modelos para demostrar que, digamos `151`, no es alcanzable desde `0`. La alternativa es computar una abstracción de predicados del sistema a la que sí se pueda aplicar un comprobador de modelos.

Aunque no existen unas normas fijas que seguir a la hora de decidir qué predicados utilizar para crear el sistema abstracto, sí que es habitual incluir al menos todos los que se han utilizado en la especificación (o al menos aquellos que parezcan más relevantes entre ellos). Del mismo modo, habrá que incluir predicados relacionados con la propiedad que se quiera demostrar. Ciertamente, esta "metodología" no es muy precisa y cada caso concreto debe tratarse de manera particular: nosotros vamos a intentar iluminar su aplicación a través de los ejemplos de este capítulo.

Los predicados que se van a usar para abstraer el módulo `SD-EXAMPLE` son:

- $\phi_1(n) \iff n \leq 10$.
- $\phi_2(n) \iff n$ es par.

ϕ_1 es una ligera modificación de la condición de la primera regla, que preferimos porque da lugar a un sistema abstracto más preciso. Si un estado satisface ϕ_1 lo seguirá satisfaciendo después de aplicarle esta regla; en cambio, si hubiésemos utilizado directamente la condición nos encontraríamos con que sería posible tanto que el estado resultante lo satisfaciera como que no, lo que introduciría más indeterminismo en el sistema abstracto. Al predicado ϕ_2 llegamos a partir de la "condición" (en forma de patrón) de la tercera regla.

Utilizado en conjunción con ϕ_1, nos permitiría probar la propiedad deseada si fuéramos capaces de demostrar que al menos uno de ellos es cierto en todo momento. De esta manera, el conjunto de estados abstractos está formado por los cuatro pares ordenados de valores booleanos y el estado inicial `0` se convierte en $\langle$`true`, `true`$\rangle$.

Para construir la relación de transición del sistema abstracto consideremos la primera regla del módulo `SD-EXAMPLE`. Vamos a describir su abstracción por medio de un predicado $\lambda(b_1, b_2, b'_1, b'_2)$ que determine los pasos de transición

$$\langle b_1, b_2 \rangle \to \langle b'_1, b'_2 \rangle$$

si y solo si $\lambda(b_1, b_2, b'_1, b'_2)$ es cierto.

Por su condición, si la primera regla se puede aplicar a un número n entonces $\phi_1(n)$ debe ser cierto. Esto significa que el predicado abstracto λ tiene que ser de la forma

$$\lambda(b_1, b_2, b'_1, b'_2) = b_1 \wedge \lambda'(b_1, b_2, b'_1, b'_2)\,.$$

Igualmente, el número $n + 1$ resultante también satisface ϕ_1 (recuérdese la discusión anterior para elegir $n \leq 10$ en vez de $n < 10$) y con esta información λ puede ser refinado en

$$\lambda(b_1, b_2, b'_1, b'_2) = b_1 \wedge b'_1 \wedge \lambda''(b_1, b_2, b'_1, b'_2)\,.$$

El mismo argumento no funciona con ϕ_2. Sin embargo, nos podemos dar cuenta de que si la regla se aplica a un número n que satisface ϕ_2 entonces el número $n + 1$ resultante siempre satisface la negación de ϕ_2; análogamente, si n satisface la negación de ϕ_2 se tiene que $n + 1$ satisface ϕ_2. Con esto tenemos un predicado

$$\lambda(b_1, b_2, b'_1, b'_2) = b_1 \wedge b'_1 \wedge (b_2 \to \neg b'_2) \wedge (\neg b_2 \to b'_2) \wedge \lambda'''(b_1, b_2, b'_1, b'_2)\,.$$

Aparte de estas, y las que se derivan lógicamente de ellas como $\neg b_1 \to b'_2$, no existen más relaciones entre las variables booleanas que definen λ; su valor final es por lo tanto

$$\begin{aligned}\lambda(b_1, b_2, b'_1, b'_2) \;=\; & b_1 \wedge b'_1 \wedge (b_2 \to \neg b'_2) \wedge (\neg b_2 \to b'_2) \wedge \\ & (\neg b_1 \to b'_2) \wedge (\neg b_1 \to \neg b'_2) \wedge (b_2 \to b'_1) \wedge (\neg b_2 \to b'_1)\,.\end{aligned}$$

Obviamente, las cuatro últimas conjunciones son innecesarias y se podrían eliminar para simplificar λ.

Se sigue el mismo procedimiento para construir los predicados abstractos correspondientes a las demás reglas. Para la tercera, por ejemplo, resulta que ni ϕ_1 ni ϕ_2 son en general ciertos ni para $s(s(n))$ ni para el número n al que se reescribe. Sí se cumple, sin embargo, que si $s(s(n))$ es menor o igual que 10 entonces n también lo es, con lo que un primer refinamiento de la abstracción λ sería

$$\lambda(b_1, b_2, b'_1, b'_2) = b_1 \to b'_1 \wedge \lambda'(b_1, b_2, b'_1, b'_2)\,.$$

Se tiene también que si ϕ_2 se cumple para $s(s(n))$, es decir, si $s(s(n))$ es par, entonces ϕ_2 también es cierto en n, y viceversa. Como no existen más relaciones entre las variables, finalmente se tiene:

$$\lambda(b_1, b_2, b'_1, b'_2) = b_1 \to b'_1 \wedge b_2 \to b'_2 \wedge (\neg b_2 \to \neg b'_2)\,.$$

La especificación completa en Maude del sistema abstracto aparece en la página 186.

En realidad, las únicas expresiones booleanas que se comprueban al realizar una abstracción con n predicados son las incluidas en B, B' y $B \to B'$, donde $B = \{b_i, \neg b_i\}_{1 \leq i \leq n}$, $B' = \{b'_i, \neg b'_i\}_{1 \leq i \leq n}$ y $B \to B' = \{x \to x' \mid x \in B, x' \in B'\}$. El algoritmo concreto tal y como lo implementa la herramienta que proponemos es el descrito en Colón y Uribe (1998), donde estas expresiones se llaman *puntos de prueba* (test points). No vamos a dar aquí los detalles de su funcionamiento porque no son necesarios para lo que sigue y esperamos que la idea fundamental haya quedado clara con el ejemplo; una discusión detallada, además de en la referencia anterior, también se puede encontrar en la tesis de Uribe (1998).

7.3 Uso de la herramienta

El propósito de esta sección es presentar, tanto a nivel de uso como de implementación, una herramienta que computa abstracciones de predicados en Maude. Una descripción a alto nivel de su funcionamiento sería:

1. Escribir el módulo con la especificación del sistema en el que se está interesado.
2. Extender el módulo con los predicados que se vayan a utilizar para la abstracción y cargarlo en la base de datos de Maude.
3. Cargar el fichero `pa-prototype.maude` en el que está escrita la herramienta. Este módulo importa a su vez los ficheros `model-checher.maude` y `xitp-tool.maude`, que deben estar por tanto presentes en el directorio actual.
4. Ejecutar alguna de las funciones `computeAbsModule`, `abstractionGround` o bien `abstractionGen`, para obtener una de las tres posibles representaciones del módulo abstracto.

7.3.1 Abstracción de predicados en Maude

En su condición actual, la herramienta asume una serie de requisitos sobre el módulo del que se va a calcular su abstracción:

1. El tipo de los estados es `Config`.
2. Hay un estado inicial que se llama `init`.
3. Los predicados que definen la abstracción están incluidos en el mismo módulo.
4. El módulo es `[Config]`-encapsulado y las condiciones de sus reglas solo contienen ecuaciones.

Es previsible que, exceptuando la última, estas restricciones se eliminen en un futuro cercano.

Continuando con nuestro ejemplo, tenemos que extender el módulo `SD-EXAMPLE` con la especificación de los dos predicados, que a su vez hacen uso de un operador auxiliar que comprueba si un número natural es par o impar.

```
--- Predicados utilizados en la abstraccion y sus funciones auxiliares.

ops phi1 phi2 : Config -> Bool .
op isEven? : Nat -> Bool .

eq phi1(N) = (N <= 10) .
eq phi2(N) = isEven?(N) .

eq isEven?(0) = true .
eq isEven?(s(N)) = not(isEven?(N)) .
```

Para demostrar las implicaciones necesarias para construir el predicado abstracto, la herramienta invoca al ITP. Como con todos los demostradores de teoremas, podría ocurrir que una determinada fórmula sea válida pero que sin embargo el ITP no llegue a demostrarla. En realidad esta situación es frecuente y produce como resultado predicados abstractos menos precisos, al no ser descubiertas algunas de las relaciones entre las variables booleanas. Pero en general el ITP realiza un trabajo aceptable y en ocasiones cuando falla se le puede ayudar introduciendo ecuaciones adicionales. De hecho esta situación se produce en nuestro ejemplo, donde resulta necesario añadir la siguiente ecuación en la especificación del predicado `isEven?` para que el sistema abstracto calculado sea lo suficientemente preciso como para permitir probar la propiedad deseada:

```
eq isEven?(N * 2) = true .   --- redundante, pero necesaria para el ITP
```

Una vez que el módulo conteniendo la especificación del sistema y las de los predicados es cargado en la base de datos de Maude, podemos ejecutar la herramienta y obtener el módulo abstracto con ayuda de la función

```
op computeAbsModule : Qid QidList -> Module .
```

Esta función toma como argumentos el nombre del módulo precedido de un apóstrofo y una lista de identificadores, también precedidos de apóstrofos, con los nombres de los predicados que se van a usar en la abstracción, y devuelve la *metarrepresentación* del módulo que contiene la especificación de los predicados que definen la relación de transición abstracta.

```
Maude> red computeAbsModule('SD-EXAMPLE, 'phi1 'phi2) .

result FModule: fmod 'SD-EXAMPLE-ABS is
  including 'BOOL .
  sorts none .
  none
  op 'absInit : 'Bool 'Bool -> 'Bool [none] .
  op 'lambda1 : 'Bool 'Bool 'Bool 'Bool -> 'Bool [none] .
  op 'lambda2 : 'Bool 'Bool 'Bool 'Bool -> 'Bool [none] .
  op 'lambda3 : 'Bool 'Bool 'Bool 'Bool -> 'Bool [none] .
  none
  eq 'absInit['B1:Bool,'B2:Bool] = '_and_['B1:Bool,'B2:Bool] [none] .
```

```
  eq 'lambda1['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = '_and_['B*2:Bool,'_and_[
    '_implies_['B1:Bool,'B*2:Bool],'_and_['_implies_['B2:Bool,'B*2:Bool],
    '_and_['_implies_['not_['B1:Bool],'B*2:Bool],'_and_['_implies_['not_[
    'B1:Bool],'not_['B*1:Bool]],'_implies_['not_['B2:Bool],'B*2:Bool]]]]]] [
    none] .
  eq 'lambda2['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = '_and_['_implies_[
    'B1:Bool,'B*1:Bool],'_and_['_implies_['B2:Bool,'B*2:Bool],'_implies_['not_[
    'B2:Bool],'not_['B*2:Bool]]]] [none] .
  eq 'lambda3['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = '_and_['B1:Bool,'_and_[
    'B*1:Bool,'_and_['_implies_['B1:Bool,'B*1:Bool],'_and_['_implies_['B2:Bool,
    'B*1:Bool],'_and_['_implies_['B2:Bool,'not_['B*2:Bool]],'_and_['_implies_[
    'not_['B1:Bool],'B*1:Bool],'_and_['_implies_['not_['B1:Bool],'B*2:Bool],
    '_and_['_implies_['not_['B1:Bool],'not_['B*1:Bool]],'_and_['_implies_[
    'not_['B1:Bool],'not_['B*2:Bool]],'_and_['_implies_['not_['B2:Bool],
    'B*1:Bool],'_implies_['not_['B2:Bool],'B*2:Bool]]]]]]]]]] [none] .
endfm
```

Para facilitar su comprensión, a continuación damos el módulo objeto que corresponde a la metarrepresentación anterior.

```
fmod SD-EXAMPLE-ABS is
  including BOOL .
  op absInit : Bool Bool -> Bool .
  op lambda1 : Bool Bool Bool Bool -> Bool .
  op lambda2 : Bool Bool Bool Bool -> Bool .
  op lambda3 : Bool Bool Bool Bool -> Bool .
  vars B1 B2 B*1 B*2 : Bool .

  eq absInit(B1,B2) = B1 and B2 .
  eq lambda1(B1,B2,B*1,B*2) = B*2 and (B1 implies B*2) and (B2 implies B*2) and
                              (not(B1) implies B*2) and
                              (not(B1) implies not(B*1)) and
                              (not(B2) implies B*2) .
  eq lambda2(B1,B2,B*1,B*2) = (B1 implies B*1) and (B2 implies B*2) and
                              (not(B2) implies not(B*2:Bool)) .
  eq lambda3(B1,B2,B*1,B*2) = B1 and B*1 and (B1 implies B*1) and
                              (B2 implies B*1) and (B2 implies not(B*2)) and
                              (not(B1) implies B*1) and (not(B1) implies B*2) and
                              (not(B1) implies not(B*1)) and
                              (not(B1) implies not(B*2)) and
                              (not(B2) implies B*1) and
                              (not(B2) implies B*2) .
endfm
```

Nótese que el estado inicial abstracto también es expresado por medio de un predicado. Además, debido al modo en que Maude procesa internamente la metarrepresentación de módulos, el orden de los predicados en el sistema abstracto no se atiene al de las reglas en el módulo original; aquí, por ejemplo, `lambda3` corresponde a la primera regla.

7.3.2 Dos funciones más

La función `computeAbsModule` resuelve el problema de computar el sistema abstracto pero presenta el importante inconveniente de no ser directamente ejecutable como teoría de reescritura: las transiciones no están expresadas por medio de reglas. Para remediar esta situación la herramienta incluye las siguientes dos funciones:

```
op abstractionGround : Qid QidList -> Module .
op abstractionGen : Qid QidList -> Module .
```

Ambas reciben los mismos argumentos que la función `computeAbsModule`, a la que invocan como primer paso. A continuación, la función `abstractionGround` enumera todas las posibles tuplas

$$\langle b_1, \ldots, b_n, b'_1, \ldots, b'_n \rangle$$

de valores booleanos y comprueba, para cada una de ellas, si satisface alguno de los predicados que definen la relación de transición abstracta. Devuelve como resultado la metarrepresentación de un nuevo módulo en el que cada tupla que satisface algún predicado ha sido transformada en una regla que solo utiliza términos cerrados:

$$\langle b_1, \ldots, b_n \rangle \longrightarrow \langle b'_1, \ldots, b'_n \rangle .$$

Además, este módulo también incluye una regla que define el valor de la constante `initial`, que representa el estado inicial.

Para nuestro ejemplo, obtenemos:

```
Maude> red abstractionGround('SD-EXAMPLE, 'phi1 'phi2) .

result Module: mod 'SD-EXAMPLE-ABS-RULES is
  including 'BOOL .
  sorts 'AbsState .
  none
  op 'initial : nil -> 'AbsState [none] .
  op 'st : 'Bool 'Bool -> 'AbsState [none] .
  none
  none
  rl 'initial.AbsState => 'st['true.Bool,'true.Bool] [none] .
  rl 'st['false.Bool,'false.Bool] => 'st['false.Bool,'false.Bool] [none] .
  rl 'st['false.Bool,'false.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['false.Bool,'false.Bool] => 'st['true.Bool,'false.Bool] [none] .
  rl 'st['false.Bool,'true.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['false.Bool,'true.Bool] => 'st['true.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'false.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'false.Bool] => 'st['true.Bool,'false.Bool] [none] .
  rl 'st['true.Bool,'false.Bool] => 'st['true.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'true.Bool] => 'st['false.Bool,'true.Bool] [none] .
  rl 'st['true.Bool,'true.Bool] => 'st['true.Bool,'false.Bool] [none] .
  rl 'st['true.Bool,'true.Bool] => 'st['true.Bool,'true.Bool] [none] .
endm
```

Por lo tanto, el resultado devuelto por `abstractionGround` es una (meta)especificación que ya sí es ejecutable en Maude como teoría de reescritura. Nótese que sin embargo este procedimiento es muy costoso puesto que se comprueban 2^{2n} combinaciones booleanas, donde n es el número de predicados utilizados en la abstracción.

La alternativa más económica que ofrece `abstractionGen` consiste en transformar el módulo funcional devuelto por `computeAbsModule` en un módulo de sistema, añadiendo para cada predicado λ una única regla de la forma

$$\langle b_1, \ldots, b_n \rangle \longrightarrow \langle b'_1, \ldots, b'_n \rangle \textbf{ if } cBool \longrightarrow b'_1 \wedge \ldots \wedge cBool \longrightarrow b'_n \wedge \lambda(b_1, \ldots, b_n, b'_1, \ldots, b'_n)$$

donde *cBool* es una constante que puede ser reescrita a `true` y `false`. De esta forma, la generación real de las transiciones es pospuesta hasta la fase de exploración del grafo de estados y la eficiencia global mejorará en general porque muchas de las reglas que se producían antes solo se aplicaban a estados no alcanzables y eran por lo tanto innecesarias.

Para nuestro ejemplo particular, `abstractionGen` extiende la especificación de la página 186 con las reglas:

```
rl 'cBool.Bool => 'false.Bool [none] .
rl 'cBool.Bool => 'true.Bool [none] .
crl 'initial.AbsState => 'st['B*1:Bool,'B*2:Bool] if
    'cBool.Bool => 'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\
    'absInit['B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
crl 'st['B1:Bool,'B2:Bool] => 'st['B*1:Bool,'B*2:Bool] if
    'cBool.Bool => 'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\
    'lambda1['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
crl 'st['B1:Bool,'B2:Bool] => 'st['B*1:Bool,'B*2:Bool] if
    'cBool.Bool => 'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\
    'lambda2['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
crl 'st['B1:Bool,'B2:Bool] => 'st['B*1:Bool,'B*2:Bool] if
    'cBool.Bool => 'B*1:Bool /\ 'cBool.Bool => 'B*2:Bool /\
    'lambda3['B1:Bool,'B2:Bool,'B*1:Bool,'B*2:Bool] = 'true.Bool [none] .
```

7.3.3 Demostración de propiedades

Ahora es el momento de volver a pensar en la pregunta que ha motivado toda la discusión de las pasadas secciones, esto es, si `151` es un estado alcanzable a partir del inicial `0`. Para demostrar que este no es el caso es suficiente con demostrar que para todos los estados $\langle b_1, b_2 \rangle$ alcanzables desde el estado inicial en el sistema abstracto, o bien b_1 o bien b_2 son ciertos. Y esto se puede hacer aplicando la orden `metaSearch` al resultado devuelto por `abstractionGround` o `abstractionGen`, para buscar un estado $\langle b_1, b_2 \rangle$ tal que $b_1 \vee b_2 =$ `false`. Por la forma como se definió la relación de transición abstracta (ver la página 182) y el hecho de que la aproximación que nosotros computamos contiene a aquella, esto demuestra que todos los números alcanzables en el sistema original son o bien menores o iguales que `10`, o pares; en particular, `151` no sería alcanzable.

```
Maude> red metaSearch(abstractionGround('SD-EXAMPLE, 'phi1 'phi2),
```

```
                    'initial.AbsState,
                    'st['B1:Bool,'B2:Bool],
                    '_or_['B1:Bool,'B2:Bool] = 'false.Bool,'*,unbounded,0) .

result ResultTriple?: (failure).ResultTriple?
```

Lo que hace la función `metaSearch` es buscar si a partir de `initial`, el estado inicial en el sistema abstracto devuelto por `abstractionGround`, se puede alcanzar un estado `st[B1, B2]` en el que la condición `B1 or B2` sea falsa. El resultado devuelto, `failure`, implica la negación de la condición en todos los estados alcanzables, estableciendo que siempre se tiene b_1 o b_2.

7.3.4 Algunos ejemplos

Para ilustrar más el funcionamiento de la herramienta vamos a explicar cómo se pueden tratar algunos de los sistemas que han aparecido como ejemplos a lo largo de la tesis.

Lectores y escritores. Empezamos con el sistema de lectores y escritores de la sección 5.8.2. La propiedad que hay que comprobar es que nunca hay simultáneamente lectores y escritores en el sistema y que, en ningún caso, hay más de un escritor. Utilizando esta información como guía, tras tantear un poco llegamos a los siguientes predicados:

```
ops phi1 phi2 phi3 : Config -> Bool .

eq phi1(< R, W >) = (R <= 0) .
eq phi2(< R, W >) = (W <= 0) .
eq phi3(< R, W >) = (W < 2) .
```

El objetivo de los dos primeros es distinguir si los números naturales `R` y `W` son iguales a `0`, y para ello utilizamos el operador `_<=_` (que puede ser manejado por los procedimientos de decisión implementados en el ITP) en lugar de la doble igualdad `_==_`. Esto es necesario porque este último no es un operador lógico sino que está implementado de manera interna en Maude y devuelve resultados incorrectos cuando se aplica a términos no cerrados. Por lo tanto, su uso con el ITP para probar la validez de fórmulas cualesquiera puede dar lugar a resultados falsos.

Una vez que tenemos especificado el sistema junto con los predicados para la abstracción, la propiedad se comprueba mediante la siguiente orden:

```
Maude> red metaSearch(abstractionGen('R&W, 'phi1 'phi2 'phi3),
                    'initial.AbsState,
                    'st['B1:Bool,'B2:Bool, 'B3:Bool],
                    '_and_['_or_['B1:Bool,'B2:Bool],
                           'B3:Bool] = 'false.Bool, '+, unbounded, 0) .
```

El estado inicial en el sistema abstracto devuelto por `abstractionGen` también se llama `initial`, y `metaSearch` comprueba si se puede alcanzar un estado `st[B1, B2, B3]` en el

que la condición `(B1 or B2) and B3` sea falsa. Esto es, `metaSearch` intenta encontrar un estado en el que ninguno de los dos primeros predicados sea cierto (con lo que habría tanto lectores como escritores en el sistema) o en el que el tercero sea falso (con lo que habría al menos dos escritores). El resultado

```
result ResultTriple?: (failure).ResultTriple?
```

muestra que ello no es posible.

Matemáticos pensadores. Consideremos ahora el protocolo de la sección 5.8 en el que los dos procesos se interpretan como si fueran "matemáticos pensadores". De lo que se trata ahora es de comprobar que los dos procesos no están nunca simultáneamente en la región crítica; esto, junto con el hecho de que el predicado `odd` desempeña un papel importante en las reglas nos lleva a los predicados:

```
op equal : Status Status -> Bool .
eq equal(S:Status, S:Status) = true .
eq equal(think,eat) = false .
eq equal(eat,think) = false .

eq phi1(st(L0, L1, N)) = equal(L0, think) .
eq phi2(st(L0, L1, N)) = equal(L1, think) .
eq phi3(st(L0, L1, N)) = not(odd(N)) .
```

Nótese el uso del predicado de igualdad `equal` en lugar de la doble igualdad _==_. Demostrar que el protocolo consigue exclusión mutua se consigue comprobando que no se puede alcanzar un estado en el que ninguno de los matemáticos esté pensando (`think`), primero cuando el contador inicialmente es un número impar

```
Maude> red metaSearch(abstractionGround('MATHEMATICIANS, 'phi1 'phi2 'phi3),
                      'st['true.Bool,'true.Bool,'true.Bool],
                      'st['false.Bool,'false.Bool,'B3:Bool],
                      nil, '+, unbounded, 0) .
result ResultTriple?: (failure).ResultTriple?
```

y de forma análoga cuando el contador comienza siendo par (la tercera componente del estado abstracto es ahora `false`).

```
Maude> red metaSearch(abstractionGround('MATHEMATICIANS, 'phi1 'phi2 'phi3),
                      'st['true.Bool,'true.Bool,false.Bool],
                      'st['false.Bool,'false.Bool,'B3:Bool],
                      nil, '+, unbounded, 0) .
result ResultTriple?: (failure).ResultTriple?
```

Protocolo de la panadería. Por último, consideremos el protocolo de la panadería, que se resolvió mediante una abstracción ecuacional en el capítulo 4. Este sistema ya se estudió desde el punto de vista de la abstracción de predicados en la sección 5.7.2, pero el sistema resultante no era ejecutable. Nuestra herramienta, como ya se ha explicado con anterioridad, no devuelve la misma abstracción sino una aproximación que a cambio presenta la ventaja de que se puede ejecutar. Los predicados que se usan son los mismos.

```
eq phi1(< P, X, Q, Y >) = equal(P, wait) .
eq phi2(< P, X, Q, Y >) = equal(P, crit) .
eq phi3(< P, X, Q, Y >) = equal(Q, wait) .
eq phi4(< P, X, Q, Y >) = equal(Q, crit) .
eq phi5(< P, X, Q, Y >) = (X <= 0) and (0 <= X) .
eq phi6(< P, X, Q, Y >) = (Y <= 0) and (0 <= Y) .
eq phi7(< P, X, Q, Y >) = (Y < X) .
```

Que no puede ocurrir que dos procesos se encuentren al mismo tiempo en la región crítica se demuestra comprobando que no se puede alcanzar un estado en el que las componentes segunda y cuarta sean ambas `true`.

```
Maude> red metaSearch(
               abstractionGen('BAKERY,'phi1 'phi2 'phi3 'phi4 'phi5 'phi6 'phi7),
               'initial.AbsState,
               'st['B1:Bool,'true.Bool,'B3:Bool,'true.Bool,'B5:Bool,'B6:Bool,
                   'B7:Bool],
               nil, '+, unbounded, 0) .
result ResultTriple?: (failure).ResultTriple?
```

Todas las propiedades que se han demostrado en los ejemplos de esta sección son propiedades de seguridad que afirman que no se puede alcanzar un cierto estado indeseado. En ocasiones, sin embargo, interesa comprobar propiedades más generales, como las propiedades de vivacidad que aseguran que si un determinado estado es alcanzable entonces eventualmente también se alcanzará cierto otro. En particular, este es el caso para la segunda propiedad sobre el protocolo de la panadería estudiada en el capítulo 4: un proceso que se encuentre en modo de espera terminará entrando en su región crítica.

El sistema abstracto devuelto por el prototipo también se puede utilizar para demostrar si estas propiedades se cumplen en el sistema concreto, pero ello requiere utilizar el comprobador de modelos de Maude en lugar de simplemente la función `metaSearch`. Esto ocasiona un pequeño problema puesto que el comprobador de modelos trabaja con módulos al nivel objeto mientras que el prototipo devuelve tan solo la metarrepresentación del sistema abstracto. Actualmente el prototipo no ofrece soporte para manejar esta situación, con lo que la solución más sencilla consiste en editar manualmente el módulo abstracto y modificar la representación (básicamente eliminando los apóstrofos y sustituyendo los corchetes por paréntesis) hasta obtener un módulo al nivel objeto.

En el caso concreto del protocolo de la panadería, la representación al nivel objeto del módulo devuelto por `abstractGround` queda de la forma

```
mod BAKERY-ABS-RULES is
  including BOOL .
  sorts AbsState .

  op initial : -> AbsState  .
  op st : Bool Bool Bool Bool Bool Bool Bool -> AbsState  .

  rl initial => st(false,false,false,false,true,true,false)  .
  rl st(false,false,false,false,false,false,false) =>
     st(false,false,true,false,false,false,false)  .
  rl st(false,false,false,false,false,false,false) =>
     st(true,false,false,false,false,false,true)  .
  ...
endm
```

y sobre él se definen los predicados de estado necesarios para expresar la propiedad de vivacidad, en la forma explicada en los capítulos 2 y 4:

```
mod BAKERY-ABS-RULES-CHECK is
  inc BAKERY-ABS-RULES .
  inc MODEL-CHECKER .
  subsort AbsState < State .

  ops 1wait 1crit 2wait 2crit : -> Prop .

  vars B1 B2 B3 B4 B5 B6 B7 : Bool .

  eq st(B1,B2,B3,B4,B5,B6,B7) |= 1wait = B1 .
  eq st(B1,B2,B3,B4,B5,B6,B7) |= 1crit = B2 .
  eq st(B1,B2,B3,B4,B5,B6,B7) |= 2wait = B3 .
  eq st(B1,B2,B3,B4,B5,B6,B7) |= 2crit = B4 .
endm
```

Finalmente, la propiedad se demuestra mediante la siguiente orden:

```
Maude> red modelCheck(initial, (1wait |-> 1crit) /\ (2wait |-> 2crit)) .
result Bool: true
```

7.4 Implementación de la herramienta y detalles técnicos

Aunque no vamos a entrar a describir los detalles de la implementación, pues no los consideramos demasiado interesantes, sí creemos oportuno dar una visión somera del diseño de la herramienta y de sus funciones principales. Para empezar, el prototipo trabaja fundamentalmente con metaobjetos y hace un uso intensivo de la reflexión a través del módulo `META-LEVEL` (sección 2). También, como se explica a continuación, se apoya en el demostrador de teoremas ITP y en el comprobador de modelos.

La función principal que realiza el trabajo de abstracción es `computeAbsModule`, sobre la que después operan `abstractionGround` y `abstractionGen`.

```
op computeAbsModule : Qid QidList -> Module .
eq computeAbsModule(QMod, QIL) =
         (fmod qid(string(QMod) + "-ABS") is
            including 'BOOL .
            sorts none .
            none
            computeOps(cardRS(getRls(upModule(QMod, false))),
                       lengthQidList(QIL))
            none
            computeInit(QMod,QIL)
            computeRules(QMod,QIL)
          endfm) .
```

Dados como argumentos los nombres del módulo a abstraer y los predicados a usar para hacerlo, esta función construye la metarrepresentación del módulo abstracto haciendo uso de tres funciones auxiliares que se encargan, respectivamente, de declarar los operadores del módulo, calcular el estado abstracto inicial y abstraer las reglas. De esta última tarea se encarga la función `computeRules` que, tras realizar una serie de transformaciones y cómputos previos, termina llamando, para cada regla `Rl` en el módulo `QMod` que se está abstrayendo, a la función `abstractRule`.

```
op abstractRule : Qid QidList Rule AbsSpaceList Nat -> Equation .
ceq abstractRule(QMod, QIL, Rl, ASL, N) =
    (eq qid("lambda" + string(N,10))
           [generateBoolList("B",M), generateBoolList("B*",M)] =
        abs2MetaAbsVar(abstractRuleAux(QMod, QIL, computePreFormula(Rl),
                                       computePreVariables(Rl),ASL)) [none] .)
    if M := lengthQidList(QIL) .
```

De nuevo, los detalles concretos no son importantes. Nótese que esta función construye el armazón de la regla abstracta, dándole el nombre `lambda` correspondiente (échese un vistazo al resultado devuelto en la página 186) y generando sus argumentos antes de llamar a la función `abstractRuleAux` que, junto con `tryRulePredicate`, es la que realmente soporta todo el peso del proceso.

```
op abstractRuleAux : Qid QidList Formula VarList AbsSpaceList -> AbsSpace .
eq abstractRuleAux(QMod, QIL, F, VL, AS) =
   tryRulePredicate(QMod, QIL, F, VL, AS) .
eq abstractRuleAux(QMod, QIL, F, VL, (AS, ASL)) =
   aAnd(tryRulePredicate(QMod, QIL, F, VL, AS),
        abstractRuleAux(QMod, QIL, F, VL, ASL)) .
```

Finalmente la función `abstractRuleAux` es la que va estudiando las posibles relaciones entre las variables booleanas del predicado que representa la regla abstracta. Para cada posible relación, por ejemplo $b_1 \rightarrow b_2'$, almacenada en el último argumento, llama a `tryRulePredicate` que es quien finalmente invoca al ITP para comprobar si la relación es cierta en el módulo original: si lo es, se añade al predicado que se está construyendo; en caso contrario devuelve `true`, de modo que no se añade información nueva.

La inferencia lógica está especificada en el ITP por medio de reglas, por lo que no está disponible de manera inmediata para ser utilizada en definiciones ecuacionales (la condición de una ecuación no puede contener reescrituras). Una manera de solventar este problema sin incurrir en el elevado coste (en términos de eficiencia) que supondría utilizar el metanivel y `metaSearch` consiste en llamar a la función `modelCheck` del comprobador de modelos de Maude. Esta es una función definida de manera interna que, como ya se ha visto en capítulos anteriores, utiliza para sus cómputos las reglas del módulo y que se puede utilizar en la especificación ecuacional de operaciones, permitiendo integrar definiciones basadas en reglas. Este es el motivo por el que el prototipo carga el fichero `model-checker.maude` además de `xitp-tool.maude`; en el futuro, esta dependencia podría ser eliminada.

La interacción entre el ITP y el comprobador de modelos, y de hecho el único lugar en el código en el que se utiliza alguno de los dos (además de en la función `tryInitPredicate` que computa el estado abstracto inicial), queda reflejada en la primera de las dos ecuaciones que definen `tryRulePredicate`.

```
op tryRulePredicate : Qid QidList Formula VarList AbsSpace -> AbsSpace .
ceq tryRulePredicate(QMod, QIL, PreF, VL, AS) = AS
    if F := AQuantification('C@0:Config : 'C@1:Config : VL,
                implication(PreF,
                            abs2XitpPredicate(AS,QIL)))
       /\
       modelCheck(state(attrs(
               db : createNewGoalModule(QMod, 'abs$0),
               input : ('auto*‘..Input),
               output : nil,
               proofState : < prove("abs$0", 0, 0, 'abs$0, F, nilTermList) ;
                             nil ; lemma('abs, F, QMod, "abs$0") >,
               defaultGoal : "abs$0")),
           <> isProved?) .
eq tryRulePredicate(QMod, QIL, F, VL, AS) = aTrue [owise] .
```

El ITP trabaja con estados que almacenan, entre otros atributos, una base de datos de módulos (`db`), la orden que se está ejecutando (`input`) e información sobre el objetivo que se intenta demostrar (`proofState`), que en particular contiene la fórmula `F` que se está probando. Sobre estos estados se aplican reglas de reescritura que implementan las reglas de derivación de la lógica y los procedimientos de decisión; una fórmula queda demostrada cuando se alcanza un estado en el que el valor del campo `input` es `nilTermList`, que es la propiedad que captura el predicado de estado `isProved?`.

```
eq state(attrs(db : DB, input : nilTermList, output  : QIL,
               proofState : < emptyGoalSet ; PT ; L >,
               defaultGoal : ST, Atts))
   |= isProved? = true .
```

Lo que hace el prototipo es inicializar el estado del ITP con el nombre `QMod` del módulo sobre el que se va a razonar, con la fórmula `F` que se quiere demostrar (en un

formato adecuado) y con la estrategia, `auto*`, que queremos que se aplique en la prueba; a continuación, llama al comprobador de modelos para verificar si se puede alcanzar un estado en el que `input` sea `nilTermList`, es decir, un estado en el que la fórmula se haya demostrado.

Por último, echemos un vistazo al aspecto que tienen las fórmulas que el prototipo suministra al ITP para que demuestre. Como se explicó en la sección 7.2, para cada regla lo que se intenta es descubrir relaciones de la forma $b_i \to b'_j$ que significan que si el predicado ϕ_i se cumple en el estado actual entonces ϕ_j también es cierto en el estado que se alcanza tras aplicar dicha regla. Supongamos que la regla tiene la forma $(\forall X)\, t_1 \longrightarrow t_2$ **if** C; la fórmula que se le pasa al ITP para ver si se tiene $b_i \to b'_j$ es la cuantificación universal de la implicación

$$x = t_1 \wedge y = t_2 \wedge C \to (\phi_i(x) \to \phi_j(y))\,.$$

En concreto, la conjunción $x = t_1 \wedge y = t_2 \wedge C$ se pasa ya precalculada a `tryRulePredicate` a través de la variable `PreF`, mientras que de la sustitución de las variables booleanas por los predicados en la expresión $b_i \to b'_j$ se encarga la función `abs2XitpPredicate`.

El código completo del prototipo, con comentarios en inglés, se encuentra en el apéndice al final de esta tesis.

7.5 Conclusiones

La herramienta hace un uso intensivo del ITP y tanto su precisión como su eficiencia dependen críticamente de él. Obviamente, cuantas más fórmulas sea capaz de probar el ITP, más preciso será el sistema abstracto computado. De la misma manera, cada llamada a `computeAbsModule` invoca a su vez $4mn(n+1)$ veces el ITP, donde m es el número de reglas en el módulo original y n es el número de predicados utilizados en la abstracción. Resulta claro entonces que el que el ITP se ejecute rápidamente es crucial para que la herramienta resulte útil. Actualmente, calcular la abstracción del protocolo de la panadería, un sistema con tan solo 8 reglas y 7 predicados, le costó a la herramienta (en un 1.25Ghz G4) casi 12 minutos; es obvio que aunque manejable, esta duración por el momento resulta demasiado elevada. Este tiempo no es resultado de la inherente complejidad abstracta del problema, sino de unos pequeños problemas en la implementación del ITP que provocan que se realicen muchas más reescrituras de las que se deberían hacer al intentar demostrar un objetivo.

Hemos escrito también una herramienta que implementa la técnica de abstracción de predicados implícita; su uso y diseño son muy parecidos a los del prototipo descrito en este capítulo, y de hecho gran parte del código es común. Sin embargo, las fórmulas que hay que demostrar en este caso son más problemáticas para el ITP, con lo que tanto su eficiencia como la precisión del sistema resultante son menores.

Cuando se compara con otras herramientas similares (por ejemplo, las descritas en Colón y Uribe, 1998; Das, 2003), nuestro prototipo resulta sumamente modesto. A pesar de ello, creemos importante destacar que su diseño reflexivo ha permitido desarrollarlo muy rápidamente y que en cualquier caso, como el término "prototipo" indica, el objetivo no era el de conseguir la mayor eficiencia posible sino el de experimentar un poco con

algunas de las ideas desarrolladas en la tesis. Sin embargo, esperamos poder utilizar la experiencia del presente prototipo para introducir mejoras en el ITP y en una versión más moderna de la presente herramienta que pueda competir en eficiencia con, o incluso superar a, otras herramientas de investigación avanzadas.

Consideraciones finales

En esta tesis se ha avanzado en el estudio de la lógica de reescritura a lo largo de dos direcciones principales. La primera de ellas, de carácter quizás más teórico, ha sido desarrollada fundamentalmente en el capítulo 3, en el que se han presentado demostraciones detalladas de la reflexividad de la lógica ecuacional de pertenencia y de la lógica de reescritura sobre aquella. Estas pruebas extienden de manera considerable diversos resultados previos y constituyen una fundamentación teórica sólida para el uso del metanivel en el lenguaje Maude. Además, como se ilustra al final del mismo capítulo, las descripciones de las teorías universales de estas lógicas dadas en las demostraciones tienen también una componente práctica, al permitir metarrazonar formalmente sobre conjuntos de teorías y abrir la puerta a la posibilidad de reutilizar herramientas ya existentes para realizar mecánicamente este razonamiento.

El segundo gran tema de la tesis ha sido la verificación semiautomática de propiedades de sistemas concurrentes, especialmente infinitos, mediante su reducción a sistemas más sencillos. Aquí el concepto fundamental es el de simulación, que permite transferir propiedades entre sistemas, y hemos trabajado en él a dos niveles: el de la lógica de reescritura en el que se especifican los sistemas, y a un nivel matemático más abstracto en el que dichos sistemas se representan mediante estructuras de Kripke. Para permitir la mayor flexibilidad posible a la hora de relacionar estructuras de Kripke, hemos extendido la noción de simulación hasta dar con una definición muy general que extiende la habitual en tres direcciones distintas. A continuación, hemos trasladado progresivamente esta noción al nivel de la lógica de reescritura. Existen diversas formas de expresar estas simulaciones entre teorías de reescritura, desde las más simples correspondientes a las abstracciones ecuacionales a las más complejas y expresivas simulaciones tartamudas algebraicas. Para todas ellas hemos dado obligaciones de prueba que han de comprobarse antes de poder afirmar que se tiene una simulación y hemos ilustrado su uso con ejemplos. En su conjunto, las diversas nociones de simulación y las técnicas presentadas para su comprobación complementan al comprobador de modelos de Maude y contribuyen a paliar (aunque sea parcialmente) la carencia de técnicas y herramientas para la demostración semiautomática de propiedades de sistemas especificados en la lógica de reescritura.

Finalmente, el capítulo 7 concluye unificando los dos grandes temas de la tesis desarrollando una herramienta de diseño reflexivo para computar abstracciones.

Aunque consideramos que el trabajo presentado en esta tesis constituye un todo coherente y autocontenido, existen numerosas líneas de investigación que se podrían seguir

para extender los resultados presentados. En particular, en el ámbito de las capacidades reflexivas de las lógicas de pertenencia y de reescritura, algunos temas que quedan pendientes son:

- La extensión de las demostraciones en el caso de la lógica de reescritura para tener en cuenta los recientemente introducidos atributos *congelados*, bajo los cuales la reescritura no está permitida.
- El estudio de la reflexión en otras lógicas más restrictivas pero usadas frecuentemente, como la lógica de Horn sin igualdad.
- El desarrollo de nuevos esquemas de inducción metalógicos y de herramientas que permitan utilizar las teorías universales para razonar formalmente sobre conjuntos de teorías.
- Un estudio sistemático de las relaciones entre las teorías universales de lógicas reflexivas y, en especial, de las correspondientes a lógicas relacionadas. Este estudio debería llevarse a cabo en el marco de la teoría de las lógicas generales para obtener una respuesta precisa e independiente del formalismo. Un primer enfoque a este problema, sugerido por el profesor Mario Rodríguez Artalejo, consistiría en el uso de los sistemas formales elementales (EFS) de Smullyan (1961). La idea sería construir el sistema de derivación de todos los EFS y utilizarlo como "intermediario" entre los sistemas de derivación de lógicas relacionadas, aprovechando el hecho de que todos los conjuntos r.e. pueden ser reconocidos por EFS y que el conjunto de sentencias derivables en muchas lógicas de interés es r.e.

En el tema de las simulaciones y su representación en la lógica de reescritura, consideramos que los fundamentos teóricos han quedado bien establecidos por lo que la principal área de trabajo debería ir encaminada a la mejora de las herramientas necesarias para la demostración de las obligaciones de prueba, como pueden ser el ITP o el comprobador Church-Rosser. Sería muy conveniente, en particular, que el ITP fuera extendido para poder razonar directamente sobre la relación de reescritura, mientras que algunas de las limitaciones del comprobador Church-Rosser (como el permitir tan solo el atributo de conmutatividad) deberían ser eliminadas; asimismo, para poder utilizar la herramienta para la abstracción de predicados en situaciones más complejas es imprescindible un aumento dramático en la eficiencia del ITP. En la actualidad, la mayor parte de estas cuestiones se encuentran en desarrollo dentro de la comunidad de Maude.

Por último, y tal como se señaló en el capítulo 6, desde un punto de vista categórico existen todavía muchas cuestiones sin responder, como la existencia de límites en las categorías de Grothendieck, además de quedar pendiente un estudio que construya una institución adecuada para la lógica de reescritura y la relacione con la correspondiente a las estructuras de Kripke.

Apéndice A

Código del prototipo para la abstracción de predicados en Maude

```
---
--- pa-prototype.maude
---
--- Tool to compute abstract systems using the transformation in
---
--- M. Colon, T. Uribe. "Generating Finite-State Abstractions of Reactive
--- Systems Using Decision Procedures", LNCS 1427
---
---
--- Assumptions:
---
--- 1. The original module is topmost and its rules have only equational
---    conditions.
---
--- 2. The sort of the states is "Config".
---
--- 3. The initial state is called "init".
---
--- 4. The predicates that define the abstraction are defined in that same
---    original module.
---
---
--- How to use it:
---
--- 1. Load the module into Maude's databade.
---
--- 2. Load this file.
---
--- 3. To get the abstract system, run either the command
---    "abstractionGround(QMod, QIL)", or "abstractionGen(QMod, QIL)",
---    with "QMod" the module's name and "QIL" the list of predicates used
---    for the abstraction.
```

```
in model-checker .

---
--- The prototype uses the Xitp command "auto*" to check the validity of a
--- number of formulas. Unfortunately, this command is implemented by means of
--- rules which forbids its use when equationally defining an operation. In
--- order to get around this problem we use the built-in operation "modelCheck",
--- to see if the given formula can be "rew"ritten to the formula "true", and
--- that's why we need to import the model checker.
---

in xitp-tool .


---
--- Auxiliary functions for QidLists.
---

fmod QID-LIST-AUX is
  pr QID-LIST .

  var Q : Qid .
  var QIL : QidList .
  var N : Nat .

  --- "lengthQidList"

  op lengthQidList : QidList -> Nat .
  eq lengthQidList(nil) = 0 .
  eq lengthQidList(Q QIL) = s(lengthQidList(QIL)) .

  --- "getNQid"
  --- It returns the N-th element in a list.

  op getNQid : QidList NzNat -> Qid .
  eq getNQid(Q QIL, 1) = Q .
  eq getNQid(Q QIL, s(s(N))) = getNQid(QIL, s(N)) .

endfm


---
--- This module defines auxiliary functions to translate abstract variables back
--- and forth to their associated predicates.
---

fmod ABS-VARIABLES is
  pr META-LEVEL .
  pr QID-LIST-AUX .
  pr CONVERSION .   --- strings <-> numbers
```

```
sorts AbsVariable AbsPoint AbsSpace AbsSpaceList .
subsort AbsVariable < AbsPoint < AbsSpace < AbsSpaceList .

--- Constructors for the abstract space.
--- b(N) corresponds to the N-th predicate applied to the current state, while
--- b*(N) corresponds to the N-th predicate applied to the next state.
---
--- These operators are only used during the computation of the abstraction,
--- to ease its construction. In the final abstract module, states are given
--- by tuples of Boolean values.

ops b b* : Nat -> AbsVariable .
op aTrue : -> AbsSpace .
op aNot : AbsVariable -> AbsPoint .
op aImplication : AbsPoint AbsPoint -> AbsPoint .
op aAnd : AbsSpace AbsSpace -> AbsSpace [assoc comm id: aTrue] .
op _,_ : AbsSpaceList AbsSpaceList -> AbsSpaceList [assoc] .

var N : Nat .
var AV : AbsVariable .
vars AP AP1 AP2 : AbsPoint .
var AS : AbsSpace .

--- "abs2MetaAbsVar"
--- Given an abstract state, it returns its metarepresentation as a
--- combination of Boolean variables in the abstract module.

op abs2MetaAbsVar : AbsSpace -> Term .
eq abs2MetaAbsVar(aTrue) = 'true.Bool .
eq abs2MetaAbsVar(b(N)) = qid("B" + string(N,10) + ":Bool") .
eq abs2MetaAbsVar(b*(N)) = qid("B*" + string(N,10) + ":Bool") .
eq abs2MetaAbsVar(aNot(AV)) = 'not_[abs2MetaAbsVar(AV)] .
eq abs2MetaAbsVar(aImplication(AP1,AP2)) =
              '_implies_[abs2MetaAbsVar(AP1),abs2MetaAbsVar(AP2)] .
eq abs2MetaAbsVar(aAnd(AP,AS)) =
              '_and_[abs2MetaAbsVar(AP),abs2MetaAbsVar(AS)] .

--- "initialTestSet"
--- It returns the set of test points (abstract states) for the initial
--- state, assuming N predicates.

op initialTestSet : Nat -> AbsSpaceList .
eq initialTestSet(1) = (b(1), aNot(b(1))) .
eq initialTestSet(s(s(N))) = (initialTestSet(s(N)),
                              b(s(s(N))), aNot(b(s(s(N))))) .


--- "ruleTestSet"
--- It returns the set of test points (abstract states) for the rules,
--- assuming N predicates.
--- "ruleTestSet1" computes P_U and P'_U in Colon & Uribe, and
```

```
  --- "ruleTestSet2" computes P_U -> P'_U with implications expressed using
  --- negations and disjunctions.

  op ruleTestSet : Nat -> AbsSpaceList .
  eq ruleTestSet(N) = (ruleTestSet1(N), ruleTestSet2(N,N)) .

  op ruleTestSet1 : Nat -> AbsSpaceList .
  eq ruleTestSet1(1) = (b(1), aNot(b(1)), b*(1), aNot(b*(1))) .
  eq ruleTestSet1(s(s(N))) = (ruleTestSet1(s(N)),
                              b(s(s(N))), aNot(b(s(s(N)))),
                              b*(s(s(N))), aNot(b*(s(s(N))))) .

  var M : Nat .

  op ruleTestSet2 : Nat Nat -> AbsSpaceList .
  eq ruleTestSet2(1, N) = ruleTestSet3(1, N) .
  eq ruleTestSet2(s(s(M)), N) = (ruleTestSet2(s(M), N),
                                 ruleTestSet3(s(s(M)), N)) .

  op ruleTestSet3 : Nat Nat -> AbsSpaceList .
  eq ruleTestSet3(N, 1) = (aImplication(b(N),b*(1)),
                           aImplication(aNot(b(N)),b*(1)),
                           aImplication(b(N),aNot(b*(1))),
                           aImplication(aNot(b(N)),aNot(b*(1)))) .
  eq ruleTestSet3(N, s(s(M))) = (ruleTestSet3(N,s(M)),
                                 aImplication(b(N),b*(s(s(M)))),
                                 aImplication(aNot(b(N)),b*(s(s(M)))),
                                 aImplication(b(N),aNot(b*(s(s(M))))),
                                 aImplication(aNot(b(N)),aNot(b*(s(s(M)))))) .

endfm

---
--- This is the main module, defining the abstraction function. The abstract
--- system is computed as described in Colon & Uribe by "computeAbsModule". It
--- returns, however, a module in which the transitions are expressed by means
--- of equationally defined predicates instead of rules.
---

mod PROTOTYPE is
  inc ITP-INTERFACE .
  inc MODEL-CHECKER * (sort Formula to LTLFormula, sort State to LTLState) .
  pr ABS-VARIABLES .
  pr QID-LIST-AUX .

  ------------------------------------------------------------------------------

  --- This part establishes the interface with the model checker by specifying
  --- an atomic proposition "isProved?" that holds in those Xitp-states that
  --- result from discharging a goal.
```

```
subsort State < LTLState .

op isProved? : -> Prop .

var DB : ITPDatabase .
var QIL : QidList .
var PT : ProofTrace .
var L : LemmaSet .
var ST : String .
var Atts : ITPAttrSet .

eq state(attrs(db : DB, input : nilTermList, output  : QIL,
               proofState : < emptyGoalSet ; PT ; L >,
               defaultGoal : ST, Atts))
   |= isProved? = true .

------------------------------------------------------------------------------

var QMod : Qid .

--- "computeAbsModule"
--- It takes the module's name and the predicates' names, and returns the
--- abstract module.
--- It assumes that the sort of concrete states is "Config" with initial
--- state "init"; they become "absState" and "absInit" in the abstract
--- system.

op computeAbsModule : Qid QidList -> Module .
eq computeAbsModule(QMod, QIL) =
         (fmod qid(string(QMod) + "-ABS") is
            including 'BOOL .
            sorts none .
            none
            computeOps(cardRS(getRls(upModule(QMod, false))),
                       lengthQidList(QIL))
            none
            computeInit(QMod,QIL)
            computeRules(QMod,QIL)
          endfm) .

var Rl : Rule .
var RS : RuleSet .

--- "cardRS"
--- Auxilary function thatcomputes the cardinality of a set of rules.

op cardRS : RuleSet -> Nat .
eq cardRS(none) = 0 .
eq cardRS(Rl RS) = 1 + cardRS(RS) .

vars M N : Nat .
```

```
--- "computeOps"
--- Declares the predicates that correspond to the initial state and
--- the abstract rules. It receives the number of rules in the concrete
--- module and the number of predicates used for the abstraction.

op computeOps : Nat Nat -> OpDeclSet .
eq computeOps(N,M) = (op 'absInit : generateNBools(M) -> 'Bool [none]. )
                     generateNLambdas(N, M + M) .

--- "generateNBools"
--- It returns a list with N copies of "Bool".

op generateNBools : Nat -> TypeList .
eq generateNBools(1) = 'Bool .
eq generateNBools(s(s(N))) = 'Bool generateNBools(s(N)) .

--- "generateNLambdas"
--- generateNLambdas(N, M) declares N operators with name "lambdaN",
--- each of them taking M Boolean arguments.

op generateNLambdas : Nat Nat -> OpDeclSet .
eq generateNLambdas(1, M) =
   (op 'lambda1 : generateNBools(M) -> 'Bool [none].) .
eq generateNLambdas(s(s(N)), M) =
   generateNLambdas(s(N), M)
   (op qid("lambda" + string(s(s(N)),10)) : generateNBools(M) ->
                                            'Bool [none].) .

--- "computeInit"
--- Builds the initial abstract state. It receives the module's name and
--- the list of predicates.

op computeInit : Qid QidList -> EquationSet .
ceq computeInit(QMod, QIL) =
    (eq 'absInit[generateBoolList("B",N)] =
        abs2MetaAbsVar(computeInitAux(QMod,initialTestSet(N),QIL))
        [none] .)
    if N := lengthQidList(QIL) .

--- ST : String .

--- "generateBoolList"
--- Given a string S and number N, it returns a list of metarepresented
--- Boolean variables: 'S1:Bool, ..., 'SN:Bool

op generateBoolList : String Nat -> VarList .
eq generateBoolList(ST, 1) = qid(ST + "1:Bool") .
eq generateBoolList(ST, s(s(N))) = (generateBoolList(ST, s(N)),
                                    qid(ST + string(s(s(N)),10) + ":Bool")) .
```

```
var AS : AbsSpace .
var ASL : AbsSpaceList .

--- "computeInitAux"
--- It is called by "computeInit": it simply applies the algorithm in
--- Colon & Uribe with the help of "tryInitPredicate" and returns the
--- abstract initial state. It'll then be converted into the
--- metarepresentation a Boolean expression in "computeInit".

op computeInitAux : Qid AbsSpaceList QidList -> AbsSpace .
eq computeInitAux(QMod, AS, QIL) = tryInitPredicate(QMod, AS, QIL) .
eq computeInitAux(QMod, (AS, ASL), QIL) =
       aAnd(tryInitPredicate(QMod, AS, QIL), computeInitAux(QMod, ASL, QIL)) .

vars AP AP1 AP2 : AbsPoint .
--- N : Nat .
--- QIL : QidList .

--- "abs2XitpPredicate"
--- Given an abstract state and a list of predicate names, it returns
--- the representation of the predicate that corresponds to it in Xitp
--- internal signature.
--- In the representation, the current state is written C@0:Config while
--- the succesor, used for b*(N), is written C@1:Config.

op abs2XitpPredicate : AbsSpace QidList -> Formula .
eq abs2XitpPredicate(aTrue, QIL) = equality('true.Bool,'true.Bool) .
   --- not used when translating test points
eq abs2XitpPredicate(b(N), QIL) = equality(getNQid(QIL,N)['C@0:Config],
                                           'true.Bool) .
eq abs2XitpPredicate(b*(N), QIL) = equality(getNQid(QIL,N)['C@1:Config],
                                            'true.Bool) .
eq abs2XitpPredicate(aNot(b(N)), QIL) = equality(getNQid(QIL,N)['C@0:Config],
                                                 'false.Bool) .
eq abs2XitpPredicate(aNot(b*(N)), QIL) = equality(getNQid(QIL,N)['C@1:Config],
                                                  'false.Bool) .
eq abs2XitpPredicate(aImplication(AP1, AP2), QIL) =
        implication(abs2XitpPredicate(AP1,QIL), abs2XitpPredicate(AP2,QIL)) .
eq abs2XitpPredicate(aAnd(AP,AS),QIL) =
        conjunction(abs2XitpPredicate(AP,QIL), abs2XitpPredicate(AS,QIL)) .
   --- not used when translating test points

var F : Formula .
--- AS : AbsSpace .

--- "tryInitPredicate"
--- It takes the module's name, a test point (abstract state), and the name of
--- the predicates, and calls the Xitp to see if the concrete initial state
--- satisfies the predicate that corresponds to it; if so, it is returned and
--- added to the initial abstract state.
```

```
op tryInitPredicate : Qid AbsSpace QidList -> AbsSpace .
ceq tryInitPredicate(QMod, AS, QIL) = AS
    if F := AQuantification('C@0:Config,
                implication(equality('C@0:Config, 'init.Config),
                            abs2XitpPredicate(AS,QIL)))
       /\
       modelCheck(state(attrs(
               db : createNewGoalModule(QMod, 'abs$0),
               input : ('auto*`..Input),
               output : nil,
               proofState : < prove("abs$0", 0, 0, 'abs$0, F, nilTermList) ;
                              nil ; lemma('abs, F, QMod, "abs$0") >,
               defaultGoal : "abs$0")),
           <> isProved?) .
eq tryInitPredicate(QMod, AS, QIL) = aTrue [owise] .

--- "computeRules"
--- It returns a set of equations for the predicates that define the
--- transitions in the abstract system.

op computeRules : Qid QidList -> EquationSet .
eq computeRules(QMod, QIL) =
   computeRulesAux(QMod, QIL, getRls(upModule(QMod, false)),
                   ruleTestSet(lengthQidList(QIL)), 1) .

--- AS : AbsSpace .
--- ASL : AbsSpaceList .
--- Rl : Rule .
--- RS : RuleSet .
--- N : Nat .

--- "computeRulesAux"
--- It is called by "computeRules". The fourth argument stores the set of
--- test points: it is not needed yet, not even in "abstractRule", but it
--- is kept for efficiency reasons so as not to compute it more than once.
--- The last argument counts how many rules have been already dealt with,
--- and is used to name the corresponding predicate.

op computeRulesAux : Qid QidList RuleSet AbsSpaceList Nat -> EquationSet .
eq computeRulesAux(QMod, QIL, none, ASL, N) = none .
eq computeRulesAux(QMod, QIL, Rl RS, ASL, N) =
   abstractRule(QMod,QIL,Rl,ASL,N) computeRulesAux(QMod,QIL,RS,ASL,N + 1) .

--- "abstractRule"
--- Before calling "abstracRuleAux", it computes that part of the formula
--- that are common for all the test poins using "computePreFormula" and
--- "computePreVariables".

op abstractRule : Qid QidList Rule AbsSpaceList Nat -> Equation .
ceq abstractRule(QMod, QIL, Rl, ASL, N) =
    (eq qid("lambda" + string(N,10))
```

```
           [generateBoolList("B",M), generateBoolList("B*",M)] =
        abs2MetaAbsVar(abstractRuleAux(QMod, QIL, computePreFormula(Rl),
                                       computePreVariables(Rl),ASL)) [none] .)
    if M := lengthQidList(QIL) .

vars T1 T2 : Term .
var ATS : AttrSet .
var EqC : Condition .

--- "computePreFormula"
--- For a rule T1 => T2 if Cond, it returns T1 = C@0 /\ T2 = C@1 /\ Cond
--- in Xitp signature.
--- Only equational conditions are allowed.

op computePreFormula : Rule -> Formula .
eq computePreFormula(rl T1 => T2 [ATS].) =
   conjunction(equality('C@0:Config, T1),equality('C@1:Config, T2)) .
eq computePreFormula(crl T1 => T2 if EqC [ATS].) =
   conjunction(equality('C@0:Config, T1),equality('C@1:Config, T2),
               condition2ITPFormula(EqC)) .

var T : Term .
vars EqC1 EqC2 : Condition .
var S : Sort .

op condition2ITPFormula : Condition -> Formula .
eq condition2ITPFormula(nil) = trueFormula .
eq condition2ITPFormula((T1 = T2)) = equality(T1,T2) .
eq condition2ITPFormula((T : S)) = sortP(T,S) .
eq condition2ITPFormula(EqC1 /\ EqC2) =
   conjunction(condition2ITPFormula(EqC1), condition2ITPFormula(EqC2)) .

--- "computePreVariables"
--- The functions "getVarList" and "getVarListEqCond" are defined in
--- xitp-tool.maude.

op computePreVariables : Rule -> VarList .
eq computePreVariables(rl T1 => T2 [ATS].) = getVarList((T1, T2)) .
eq computePreVariables(crl T1 => T2 if EqC [ATS].) =
   getVarList((T1, T2)) : getVarListEqCond(EqC) .

var VL : VarList .

--- "abstractRuleAux"
--- It traverses the list of test points, building the abstract transition.

op abstractRuleAux : Qid QidList Formula VarList AbsSpaceList -> AbsSpace .
eq abstractRuleAux(QMod, QIL, F, VL, AS) =
   tryRulePredicate(QMod, QIL, F, VL, AS) .
eq abstractRuleAux(QMod, QIL, F, VL, (AS, ASL)) =
   aAnd(tryRulePredicate(QMod, QIL, F, VL, AS),
```

```
          abstractRuleAux(QMod, QIL, F, VL, ASL)) .

  var PreF : Formula .

  --- "tryRulePredicate"
  --- Analogous to "tryInitPredicate", but now part of the formula to prove
  --- is given as a parameter.

  op tryRulePredicate : Qid QidList Formula VarList AbsSpace -> AbsSpace .
  ceq tryRulePredicate(QMod, QIL, PreF, VL, AS) = AS
      if F := AQuantification('C@0:Config : 'C@1:Config : VL,
                  implication(PreF,
                              abs2XitpPredicate(AS,QIL)))
         /\
         modelCheck(state(attrs(
                 db : createNewGoalModule(QMod, 'abs$0),
                 input : ('auto*‘..Input),
                 output : nil,
                 proofState : < prove("abs$0", 0, 0, 'abs$0, F, nilTermList) ;
                               nil ; lemma('abs, F, QMod, "abs$0") >,
                 defaultGoal : "abs$0")),
             <> isProved?) .
  eq tryRulePredicate(QMod, QIL, F, VL, AS) = aTrue [owise] .

endm

---
--- This modules defines two main functions:
---
--- 1. "abstractionGround" returns an executable abstraction of a module,
---    in which all the transitions are expressed by means of ground rules.
--- 2. "abstractionGen" in which the module returned by "computeAbsModule" above
---    is extended with a rule of the form "B => B* if lambda(B,B*) = true" for
---    each predicate "lambda" defining the transition relation.
---

mod PROTOTYPE-EXT is
  inc PROTOTYPE .

  var QMod : Qid .
  var QIL : QidList .

  --- "abstractionGround"
  --- One of the two main functions.

  op abstractionGround : Qid QidList -> Module .
  eq abstractionGround(QMod,QIL) =
         rulifyGround(QMod,computeAbsModule(QMod,QIL),lengthQidList(QIL)) .

  var Mod : Module .
  var N : Nat .
```

```
--- "rulifyGround"
--- It takes a module's name, its abstraction, and the list of predicates,
--- and returns a module in which the equationally defined predicates have
--- been transformed into rules.

op rulifyGround : Qid Module Nat -> Module .
eq rulifyGround(QMod, Mod, N) =
                    (mod qid(string(QMod) + "-ABS-RULES") is
                       including 'BOOL .
                       sorts 'AbsState .
                       none
                       (op 'st : generateNBools(N) -> 'AbsState [none]. )
                       (op 'initial : nil -> 'AbsState [none]. )
                       none
                       none
                       computeConstInit(Mod,N)
                       computeConstRules(Mod,
                                         cardRS(getRls(upModule(QMod, false))),
                                         N)
                     endm) .

--- Predicates are transformed into rules by brute force: they are
--- instantiated in all possible manners and, for those tuples that
--- satisfy them the corresponding rule is added. To generate the
--- instantiations, a list of metaterms 'false.Bool is generated and is
--- succesively incremented until it reaches a list of metaterms 'true.Bool.

var T : Term .

--- "generateNTemList"
--- It creates a list with N copies of the term.

op generateNTermList : Nat Term -> TermList .
eq generateNTermList(1, T) = T .
eq generateNTermList(s(s(N)), T) = (T, generateNTermList(s(N),T)) .

var TL : TermList .

--- "advanceList"
--- It computes the successor of a list of Boolean values, in a binary base
--- fashion.

op advanceList : TermList -> TermList .
eq advanceList('false.Bool) = 'true.Bool .
eq advanceList((TL,'false.Bool)) = (TL, 'true.Bool) .
eq advanceList((TL,'true.Bool)) = (advanceList(TL), 'false.Bool) .

--- "computeConstInit"
--- Rules for the initial state.

op computeConstInit : Module Nat -> RuleSet .
```

```
eq computeConstInit(Mod,N) =
   computeConstInitAux(Mod,generateNTermList(N,'false.Bool),
                           generateNTermList(N,'true.Bool)) .

vars TL1 TL2 : TermList .

--- "computeConsInitAux"
--- Tries all possible instantiations of the predicate defining the initial
--- state in turn.
--- The second argument is the current instantiation. The last one is the
--- list of 'true.Bool, used to test when to stop; it is precomputed for
--- efficiency reasons.

op computeConstInitAux : Module TermList TermList -> RuleSet .
eq computeConstInitAux(Mod, TL1, TL1) = computeConstInitAux1(Mod, TL1) .
eq computeConstInitAux(Mod, TL1, TL2) =
   computeConstInitAux1(Mod, TL1)
   computeConstInitAux(Mod, advanceList(TL1), TL2) [owise] .

op computeConstInitAux1 : Module TermList -> RuleSet .
eq computeConstInitAux1(Mod, TL) =
   if getTerm(metaReduce(Mod, 'absInit[TL])) == 'true.Bool
      then (rl 'initial.AbsState => 'st[TL] [none].)
      else none
   fi .

var M : Nat .

--- "computeConstRules"
--- The second argument is the number of rules, and the third one is the
--- number of predicates.
op computeConstRules : Module Nat Nat -> RuleSet .
eq computeConstRules(Mod, N, M) =
   computeConstRulesAux(Mod, N, generateNTermList(M + M, 'false.Bool),
                                generateNTermList(M + M, 'true.Bool)) .

--- "computeConsRulesAux"
--- It traverses the predicates that specify the transitions.

op computeConstRulesAux : Module Nat TermList TermList -> RuleSet .
eq computeConstRulesAux(Mod, 0, TL1, TL2) = none .
eq computeConstRulesAux(Mod, s(N), TL1, TL2) =
   computeConstRulesAux1(Mod, s(N), TL1, TL2)
   computeConstRulesAux(Mod, N, TL1, TL2) .

--- "computeConstRulesAux1"
--- Tries all possible instantiations of the predicate "lambdaN" in turn.
--- The second argument is the current instantiation. The last one is the
--- list of 'true.Bool, used to test when to stop; it is precomputed for
--- efficiency reasons.
```

```
op computeConstRulesAux1 : Module Nat TermList TermList -> RuleSet .
eq computeConstRulesAux1(Mod, N, TL1, TL1) =
   computeConstRulesAux2(Mod, N, TL1) .
eq computeConstRulesAux1(Mod, N, TL1, TL2) =
   computeConstRulesAux2(Mod, N, TL1)
   computeConstRulesAux1(Mod, N, advanceList(TL1), TL2) [owise] .

var Q : Qid .

--- The label is added to the rule to help while debugging. It will be
--- removed in a final version.

op computeConstRulesAux2 : Module Nat TermList -> RuleSet .
ceq computeConstRulesAux2(Mod, N, TL) =
    if getTerm(metaReduce(Mod, Q[TL])) == 'true.Bool
       then (rl 'st[TL1] => 'st[TL2] [none].)  --- label(Q)
       else none
    fi
    if Q := qid("lambda" + string(N,10)) /\
       pairTermList(TL1, TL2) := splitTermList(TL) .

--- Auxiliary declarations to split a list of terms in two halves.

sort PairTermList .
op pairTermList : TermList TermList -> PairTermList .

vars T1 T2 : Term .

op splitTermList : TermList -> PairTermList .
eq splitTermList((T1, T2)) = pairTermList(T1, T2) .
ceq splitTermList((T1, TL, T2)) = pairTermList((T1, TL1), (TL2, T2))
    if pairTermList(TL1, TL2) := splitTermList(TL) .

------------------------------------------------------------------------------

--- The other main function.

op abstractionGen : Qid QidList -> Module .
eq abstractionGen(QMod,QIL) =
       rulifyGen(computeAbsModule(QMod,QIL),
                 cardRS(getRls(upModule(QMod, false))),
                 lengthQidList(QIL)) .

--- "rulifyGen"
--- It takes a module's abstraction as computed by "computeAbsModule" together
--- with the number of rules and predicates, and extends it with a rule of
--- the form "B => B* if lambda(B,B*) = true" for each predicate "lambda"
--- defining the transition relation.

op rulifyGen : Module Nat Nat -> Module .
eq rulifyGen(Mod,M,N) =
```

```
             (mod getName(Mod) is
                including 'BOOL .
                sorts 'AbsState .
                none
                op 'cBool : nil -> 'Bool [none] .
                op 'initial : nil -> 'AbsState [none] .
                op 'st : generateNBools(N) -> 'AbsState [none] .
                getOps(Mod)
                getMbs(Mod)
                getEqs(Mod)
                rl 'cBool.Bool => 'true.Bool [none] .
                rl 'cBool.Bool => 'false.Bool [none] .
                makeRules(M,N)
              endm) .

  --- "makeRules"
  --- Generates the rule corresponding to each predicate and the one for the
  --- initial state.

  op makeRules : Nat Nat -> RuleSet .
  eq makeRules(M, N) =
     (crl 'initial.AbsState =>
          'st[generateBoolList("B*",N)]
          if makecBool(N) /\
            'absInit[generateBoolList("B*",N)] = 'true.Bool [none] .)
     makeRulesAux(M,N) .

  op makeRulesAux : Nat Nat -> RuleSet .
  eq makeRulesAux(0, N) = none .
  eq makeRulesAux(s(M), N) =
     makeRulesAux(M,N)
     (crl 'st[generateBoolList("B",N)] => 'st[generateBoolList("B*",N)]
          if makecBool(N) /\
             qid("lambda" + string(s(M),10))[generateBoolList("B",N),
                                             generateBoolList("B*",N)] =
             'true.Bool [none] .) .

  --- "makecBool"
  --- Builds the condition "cBool => B1* /\ ... /\ cBool => BN*".

  op makecBool : Nat -> Condition .
  eq makecBool(1) = 'cBool.Bool => 'B*1:Bool .
  eq makecBool(s(s(N))) =
     makecBool(s(N)) /\ 'cBool.Bool => qid("B*" + string(s(s(N)),10) + ":Bool") .

endm
```

Bibliografía

Abdulla, P., Annichini, A. y Bouajjani, A. Symbolic verification of lossy channel systems: Application to the bounded retransmission protocol. En W. R. Cleaveland, editor, *Tools and Algorithms for the Construction of Analysis of Systems, 5th International Conference, TACAS'99, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volumen 1579 de *Lecture Notes in Computer Science*, páginas 208–222. Springer-Verlag (1999).

Aggarwal, S., Courcoubetis, C. y Wolper, P. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339 (1990).

Agha, G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press (1986).

Arrais, M. y Fiadeiro, J. L. Unifying theories in different institutions. En M. Haveraaen, O. Owe y O.-J. Dahl, editores, *Recent Trends in Data Type Specification. 11th Workshop on Specification of Abstract Data Types, Joint with the 8th COMPASS Workshop, Oslo, Norway, September 19–23, 1995, Selected Papers*, volumen 1130 de *Lecture Notes in Computer Science*, páginas 81–101. Springer-Verlag (1996).

Astesiano, E., Kreowski, H.-J. y Krieg-Brückner, B., editores. *Algebraic Foundations of Systems Specification*. IFIP State-of-the-Art Reports. Springer-Verlag (1999).

Barr, M. y Wells, C. *Category Theory for Computing Science. Third Edition*. Centre de Recherches Mathématiques (1999).

Bartlett, K., Scantlebury, R. y Wilkinson, P. A note on reliable full-duplex transmission over half-duplex lines. *Communications of the ACM*, 12(5):260–261 (1969).

Basin, D., Clavel, M. y Meseguer, J. Rewriting logic as a metalogical framework. En S. Kapoor y S. Prasad, editores, *Proceedings Twentieth Conference on the Foundations of Software Technology and Theoretical Computer Science, New Delhi, India, December 13–15*, volumen 1974 de *Lecture Notes in Computer Science*, páginas 55–80. Springer-Verlag (2000).

Basin, D., Clavel, M. y Meseguer, J. Reflective metalogical frameworks. *ACM Transactions on Computational Logic*, 5(3):528–576 (2004).

BENSALEM, S., LAKHNECH, Y. Y OWRE, S. Computing abstractions of infinite state systems compositionally and automatically. En A. J. Hu y M. Y. Vardi, editores, *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*, volumen 1427 de *Lecture Notes in Computer Science*, páginas 319–331. Springer-Verlag (1998).

BERGSTRA, J. Y TUCKER, J. Characterization of computable data types by means of a finite equational specification method. En J. W. de Bakker y J. van Leeuwen, editores, *7th International Conference on Automata, Languages and Programming*, volumen 81 de *Lecture Notes in Computer Science*, páginas 76–90. Springer-Verlag (1980).

BERRY, G. Y BOUDOL, G. The chemical abstract machine. En *POPL'90. Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, páginas 81–94. ACM (1990).

BERTOT, Y. Y CASTÉRAN, P. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer-Verlag (2004).

BIDOIT, M. Y MOSSES, P. D. CASL *User Manual*, volumen 2900 de *Lecture Notes in Computer Science*. Springer-Verlag (2004).

BOROVANSKÝ, P., KIRCHNER, C., KIRCHNER, H. Y MOREAU, P.-E. ELAN from a rewriting logic point of view. *Theoretical Computer Science*, 285(2):155–185 (2002).

BOUHOULA, A., JOUANNAUD, J.-P. Y MESEGUER, J. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132 (2000).

BROWNE, M. C., CLARKE, E. M. Y GRÜMBERG, O. Characterizing finite Kripke structures in propositional temporal logic. *Theoretical Computer Science*, 59:115–131 (1988).

BRUNI, R. Y MESEGUER, J. Generalized rewrite theories. En J. C. M. Baeten, J. K. Lenstra, J. Parrow y G. J. Woeginger, editores, *Automata, Languages and Programming. 30th International Colloquium, ICALP 2003, Eindhoven, The Netherlands, June 30 - July 4, 2003. Proceedings*, volumen 2719 de *Lecture Notes in Computer Science*, páginas 252–266. Springer-Verlag (2003).

BURSTALL, R. Y GOGUEN, J. A. Putting theories together to make specifications. En R. Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, páginas 1045–1058. Department of Computer Science, Carnegie-Mellon University (1977).

BURSTALL, R. Y GOGUEN, J. A. Algebras, theories and freeness: An introduction for computer scientists. En M. Broy y G. Schmidt, editores, *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School*, Mathematical and Physical Sciences, páginas 329–348. Reidel Publishing Company (1982).

CLARKE, E. M. Y EMERSON, E. A. Design and synthesis of synchronization skeletons using branching time temporal logic. En D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs*, volumen 131 de *Lecture Notes in Computer Science*, páginas 52–71. Springer-Verlag (1981).

Clarke, E. M., Grumberg, O., Jha, S., Lu, Y. y Veith, H. Counterexample-guided abstraction refinement. En E. A. Emerson y A. P. Sistla, editores, *Computer Aided Verification. 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000 Proceedings*, volumen 1855 de *Lecture Notes in Computer Science*, páginas 154–169. Springer-Verlag (2000).

Clarke, E. M., Grumberg, O. y Long, D. E. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542 (1994).

Clarke, E. M., Grumberg, O. y Peled, D. A. *Model Checking*. MIT Press (1999).

Clavel, M. *Reflection in General Logics and in Rewriting Logic, with Applications to the Maude Language*. Tesis Doctoral, Universidad de Navarra, España (1998).

Clavel, M. *Reflection in Rewriting Logic: Metalogical Foundations and Metaprogramming Applications*. CSLI Publications (2000).

Clavel, M. The ITP tool. En A. Nepomuceno, J. F. Quesada y F. J. Salguero, editores, *Logic, Language, and Information. Proceedings of the First Workshop on Logic and Language*, páginas 55–62. Kronos (2001).

Clavel, M. The ITP home page (2004). http://geminis.sip.ucm.es/~clavel/.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N. y Meseguer, J. Metalevel computation in Maude. En C. Kirchner y H. Kirchner, editores, *Proceedings Second International Workshop on Rewriting Logic and its Applications, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998*, volumen 15 de *Electronic Notes in Theoretical Computer Science*, páginas 3–24. Elsevier (1998). http://www.elsevier.nl/locate/entcs/volume15.html.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Quesada, J. F. Maude: Specification and programming in rewriting logic (1999). http://maude.cs.uiuc.edu.

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Quesada, J. F. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 285(2):187–243 (2002a).

Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J. y Talcott, C. Maude manual (version 2.1) (2004a). http://maude.cs.uiuc.edu/manual/.

Clavel, M., Durán, F., Eker, S. y Meseguer, J. Building equational proving tools by reflection in rewriting logic. En K. Futatsugi, A. T. Nakagawa y T. Tamai, editores, *Cafe: An Industrial-Strength Algebraic Formal Method*, páginas 1–31. Elsevier (2000). http://maude.csl.sri.com/papers.

Clavel, M., Eker, S., Lincoln, P. y Meseguer, J. Principles of Maude. En J. Meseguer, editor, *Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996*, volumen 4 de *Electronic Notes in Theoretical Computer Science*, páginas 65–89. Elsevier (1996).

CLAVEL, M., MARTÍ-OLIET, N. Y PALOMINO, M. Formalizing and proving semantic relations between specifications by reflection. En C. Rattray, S. Maharaj y C. Shankland, editores, *Algebraic Methodology and Software Technology. 10th International Conference, AMAST 2004, Stirling, Scotland, UK, July 12-16, 2004, Proceedings*, volumen 3116 de *Lecture Notes in Computer Science*, páginas 72–86. Springer-Verlag (2004b).

CLAVEL, M. Y MESEGUER, J. Axiomatizing reflective logics and languages. En G. Kiczales, editor, *Proceedings of Reflection'96, San Francisco, California, April 1996*, páginas 263–288 (1996).

CLAVEL, M. Y MESEGUER, J. Reflection in conditional rewriting logic. *Theoretical Computer Science*, 285(2):245–288 (2002).

CLAVEL, M., MESEGUER, J. Y PALOMINO, M. Reflection in membership equational logic, many-sorted equational logic, horn-logic with equality, and rewriting logic. En F. Gadducci y U. Montanari, editores, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA'02, Pisa, Italy, September 19–21, 2002*, volumen 71 de *Electronic Notes in Theoretical Computer Science*. Elsevier (2002b).

CLAVEL, M., PALOMINO, M. Y SANTA-CRUZ, J. Integrating decision procedures in reflective rewriting-based theorem provers. En S. Antoy y Y. Toyama, editores, *Fourth International Workshop on Reduction Strategies in Rewriting and Programming*, páginas 15–24 (2004c). Informe técnico AIB-2004-06, Department of Computer Science, RWTH, Aachen.

COLÓN, M. A. Y URIBE, T. E. Generating finite-state abstractions of reactive systems using decision procedures. En A. J. Hu y M. Y. Vardi, editores, *Computer Aided Verification. 10th International Conference, CAV'98, Vancouver, BC, Canada, June 28-July 2, 1998, Proceedings*, volumen 1427 de *Lecture Notes in Computer Science*, páginas 293–304. Springer-Verlag (1998).

CONTEJEAN, E. Y MARCHÉ, C. C*i*ME: Completion modulo *E*. En H. Ganzinger, editor, *Rewriting Techniques and Applications. 7th International Conference, RTA-96, New Brunswick, NJ, USA July 27 - 30, 1996. Proceedings*, volumen 1103 de *Lecture Notes in Computer Science*, páginas 416–419. Springer-Verlag (1996).

CONTEJEAN, E., MARCHÉ, C., MONATE, B. Y URBAIN, X. The C*i*ME Rewrite tool (2004). http://cime.lri.fr/.

DAMS, D., GERTH, R. Y GRUMBERG, O. Abstract interpretation of reactive systems. *ACM Transactions on Programming Languages and Systems*, 19:253–291 (1997).

D'ARGENIO, P. R., KATOEN, J. P., RUYS, T. Y TRETMANS, G. T. The bounded retransmission protocol must be on time. En E. Brinksma, editor, *Tools and Algorithms for the Construction and Analysis of Systems Third International Workshop, TACAS'97, Enschede, The Netherlands, April 2-4, 1997, Proceedings*, volumen 1217 de *Lecture Notes in Computer Science*, páginas 416–432. Springer-Verlag (1997).

DAS, S. *Predicate Abstraction*. Tesis Doctoral, Department of Electrical Engineering, Stanford University (2003).

Das, S., Dill, D. L. y Park, S. Experience with predicate abstraction. En N. Halbwachs y D. Peled, editores, *Computer Aided Verification. 11th International Conference, CAV'99, Trento, Italy, July 6-10, 1999, Proceedings*, volumen 1633 de *Lecture Notes in Computer Science*, páginas 160–171. Springer-Verlag (1999).

Dershowitz, N. y Jouannaud, J.-P. Rewrite systems. En J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B*, páginas 243–320. North-Holland (1990).

Durán, F. *A Reflective Module Algebra with Applications to the Maude Language*. Tesis Doctoral, Universidad de Málaga, España (1999). http://maude.cs.uiuc.edu/papers.

Durán, F. Coherence checker and completion tools for Maude specifications (2000a). http://maude.cs.uiuc.edu/tools.

Durán, F. Termination checker and Knuth-Bendix completion tools for Maude equational specifications (2000b). Manuscrito, Computer Science Laboratory, SRI International, http://maude.cs.uiuc.edu/papers.

Durán, F. y Meseguer, J. A Church-Rosser checker tool for Maude equational specifications (2000). http://maude.cs.uiuc.edu/tools.

Ehrig, H. y Mahr, B. *Fundamentals of Algebraic Specification 1. Equations and Initial Semantics*, volumen 6 de *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag (1985).

Eker, S., Meseguer, J. y Sridharanarayanan, A. The Maude LTL model checker. En F. Gadducci y U. Montanari, editores, *Proceedings Fourth International Workshop on Rewriting Logic and its Applications, WRLA'02, Pisa, Italy, September 19–21, 2002*, volumen 71 de *Electronic Notes in Theoretical Computer Science*. Elsevier (2002).

Futatsugi, K. y Diaconescu, R. *CafeOBJ Report*. World Scientific, AMAST Series (1998).

Goguen, J. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. En E. Blum, M. Paul y S. Takasu, editores, *Mathematical Studies of Information Processing, Proceedings of the International Conference, Kyoto, Japan, August 23–26, 1978*, volumen 75 de *Lecture Notes in Computer Science*, páginas 425–473. Springer-Verlag (1979).

Goguen, J. y Burstall, R. Institutions: Abstract model theory for specification and programming. *Journal of the Association for Computing Machinery*, 39(1):95–146 (1992).

Goguen, J., Kirchner, C., Kirchner, H., Mégrelis, A., Meseguer, J. y Winkler, T. An introduction to OBJ3. En S. Kaplan y J.-P. Jouannaud, editores, *Conditional Term Rewriting Systems, 1st International Workshop Orsay, France, July 8–10, 1987, Proceedings*, volumen 308 de *Lecture Notes in Computer Science*, páginas 258–263. Springer-Verlag (1988).

Goguen, J. y Meseguer, J. Completeness of many-sorted equational logic. *Houston Journal of Mathematics*, 11(3):307–334 (1985).

GOGUEN, J. A., THATCHER, J., WAGNER, E. Y WRIGHT, J. Abstract data types as initial algebras and the correctness of data representations. En A. Klinger, editor, *Computer Graphics, Pattern Recognition and Data Structure*, páginas 89–93. IEEE Press, Beverley Hills CA (1975).

GOGUEN, J. A., WINKLER, T., MESEGUER, J., FUTATSUGI, K. Y JOUANNAUD, J.-P. Introducing OBJ. En J. A. Goguen y G. Malcolm, editores, *Software Engineering with OBJ: Algebraic Specification in Action*, Advances in Formal Methods, capítulo 1, páginas 3–167. Kluwer Academic Press (2000).

GRAF, S. Y SAÏDI, H. Construction of abstract state graphs with PVS. En O. Grumberg, editor, *Computer Aided Verification. 9th International Conference, CAV'97, Haifa, Israel, June 22-25, 1997, Proceedings*, volumen 1254 de *Lecture Notes in Computer Science*, páginas 72–83. Springer-Verlag (1997).

HAVELUND, K. Y SHANKAR, N. Experiments in theorem proving and model checking for protocol verification. En M.-C. Gaudel y J. Woodcock, editores, *FME '96: Industrial Benefit and Advances in Formal Methods. Third International Symposium of Formal Methods Europe Co-Sponsored by IFIP WG 14.3, Oxford, UK, March 18 - 22, 1996. Proceedings*, volumen 1051 de *Lecture Notes in Computer Science*, páginas 662–681. Springer-Verlag (1996).

HENDRIX, J. A completeness checker for Maude (2003). Manuscrito, University of Illinois at Urbana-Champaign.

HENNESSY, M. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. John Wiley & Sons (1990).

HERRLICH, H. Y STRECKER, G. E. *Category Theory: An Introduction*. Advanced Mathematics. Allyn and Bacon, Boston (1973).

HOARE, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580 (1969).

HOARE, C. A. R. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677 (1978).

HOLZMANN, G. J. *The SPIN Model Checker*. Addison-Wesley (2003).

JACOBS, B. *Categorical Logic and Type Theory*, volumen 141 de *Studies in Logic and the Foundations of Mathematics*. North-Holland (1999).

KAUFMANN, M., MANOLIOS, P. Y MOORE, J. S. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Press (2000).

KESTEN, Y. Y PNUELI, A. Control and data abstraction: The cornerstones of practical formal verification. *International Journal on Software Tools for Technology Transfer*, 4(2):328–342 (2000a).

Kesten, Y. y Pnueli, A. Verification by augmentary finitary abstraction. *Information and Computation*, 163:203–243 (2000b).

Lamport, L. A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455 (1974).

Lamport, L. What good is temporal logic? En R. E. A. Mason, editor, *Information Processing 83: Proceedings of the IFIP 9th World Congress*, páginas 657–668. North-Holland (1983).

Loeckx, J., Ehrich, H.-D. y Wolf, M. *Specification of Abstract Data Types*. J. Wiley & Sons and B. G. Teubner (1996).

Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A. y Bensalem, S. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–36 (1995).

Mac Lane, S. *Categories for the Working Mathematician. Second Edition*. Springer-Verlag (1998).

Manna, Z. y Pnueli, A. *The Temporal Logic of Reactive and Concurrent Systems. Specification*. Springer-Verlag (1992).

Manna, Z. y Pnueli, A. *Temporal Verification of Reactive Systems. Safety*. Springer-Verlag (1995).

Manolios, P. *Mechanical Verification of Reactive Systems*. Tesis Doctoral, University of Texas at Austin (2001).

Manolios, P. A compositional theory of refinement for branching time. En D. Geist y E. Tronci, editores, *Correct Hardware Design and Verification Methods. 12th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2003, L'Aquila, Italy, October 21-24, 2003, Proceedings*, volumen 2860 de *Lecture Notes in Computer Science*, páginas 304–318. Springer-Verlag (2003).

Martí-Oliet, N. y Meseguer, J. Rewriting logic as a logical and semantic framework. En D. Gabbay, editor, *Handbook of Philosophical Logic. Second Edition*, volumen 9, páginas 1–81. Kluwer Academic Press (2002a).

Martí-Oliet, N. y Meseguer, J. Rewriting logic: Roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154 (2002b).

Martí-Oliet, N., Meseguer, J. y Palomino, M. Theoroidal maps as algebraic simulations. En J. L. Fiadeiro, P. Mosses y F. Orejas, editores, *Recent Trends in Algebraic Development Techniques. 17th International Workshop, WADT 2004. Barcelona, Spain, March 27-30, 2004. Revised Selected Papers*, Lecture Notes in Computer Science. Springer-Verlag (2004). Se publicará en diciembre de 2004.

McMillan, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Press (1993).

MESEGUER, J. General logics. En H.-D. Ebbinghaus, J. Fernández-Prida, M. Garrido, D. Lascar y M. Rodríguez-Artalejo, editores, *Logic Colloquium'87*, páginas 275–329. North-Holland (1989).

MESEGUER, J. Rewriting as a unified model of concurrency. En J. C. M. Baeten y J. W. Klop, editores, *CONCUR'90, Theories of Concurrency: Unification and Extension, Amsterdam, The Netherlands, August 1990, Proceedings*, volumen 458 de *Lecture Notes in Computer Science*, páginas 384–400. Springer-Verlag (1990a).

MESEGUER, J. Rewriting as a unified model of concurrency. Informe técnico SRI-CSL-90-02, SRI International, Computer Science Laboratory (1990b). Revisión de junio de 1990.

MESEGUER, J. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155 (1992).

MESEGUER, J. A logical theory of concurrent objects and its realization in the Maude language. En G. Agha, P. Wegner y A. Yonezawa, editores, *Research Directions in Concurrent Object-Oriented Programming*, páginas 314–390. MIT Press (1993).

MESEGUER, J. Membership algebra as a logical framework for equational specification. En F. Parisi-Presicce, editor, *Recent Trends in Algebraic Development Techniques, 12th International Workshop, WADT'97, Tarquinia, Italy, June 3 - 7, 1997, Selected Papers*, volumen 1376 de *Lecture Notes in Computer Science*, páginas 18–61. Springer-Verlag (1998).

MESEGUER, J. Lecture notes for CS376 (2002). Computer Science Department, University of Illinois at Urbana-Champaign, http://www-courses.cs.uiuc.edu/~cs376/.

MESEGUER, J. Localized fairness: A rewriting semantics (2004). Artículo en preparación.

MESEGUER, J., PALOMINO, M. Y MARTÍ-OLIET, N. Equational abstractions. En F. Baader, editor, *Automated Deduction - CADE-19. 19th International Conference on Automated Deduction, Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings*, volumen 2741 de *Lecture Notes in Computer Science*, páginas 2–16. Springer-Verlag (2003).

MILNER, R. *A Calculus of Communicating Systems*. Springer-Verlag (1982).

MILNER, R., PARROW, J. Y WALKER, D. A calculus of mobile processes (Parts I and II). *Information and Computation*, 100:1–77 (1992).

MÜLLER, O. Y NIPKOW, T. Combining model checking and deduction for I/O-automata. En E. Brinksma, W. R. Cleaveland, K. G. Larsen, T. Margaria y B. Steffen, editores, *Tools and Algorithms for the Construction and Analysis of Systems. First International Workshop, TACAS '95, Aarhus, Denmark, May 19 - 20, 1995. Selected Papers*, volumen 1019 de *Lecture Notes in Computer Science*, páginas 1–16. Springer-Verlag (1995).

NAMJOSHI, K. S. A simple characterization of stuttering bisimulation. En S. Ramesh y G. Sivakumar, editores, *Foundations of Software Technology and Theoretical Computer Science. 17th Conference, Kharagpur, India, December 18 - 20, 1997. Proceedings*, volumen 1346 de *Lecture Notes in Computer Science*, páginas 284–296. Springer-Verlag (1997).

Owre, S., Rushby, J., Shankar, N. y Stringer-Calvert, D. PVS: An experience report. En D. Hutter, W. Stephan, P. Traverso y M. Ullmann, editores, *Applied Formal Methods - FM-Trends 98, International Workshop on Current Trends in Applied Formal Method, Boppard, Germany, October 7-9, 1998, Proceedings*, volumen 1641 de *Lecture Notes in Computer Science*, páginas 338–345. Springer-Verlag (1998).

Palomino, M. Comparing Meseguer's rewriting logic with the logic CRWL. En M. Hanus, editor, *International Workshop on Functional and (Constraint) Logic Programming (WFLP 2001). Selected Papers*, volumen 64 de *Electronic Notes in Theoretical Computer Science*. Elsevier (2001a).

Palomino, M. *Relating Meseguer's Rewriting Logic and the Constructor-Based Rewriting Logic*. Trabajo de Doctorado, Facultad de Matemáticas, Universidad Complutense de Madrid (2001b). http://maude.cs.uiuc.edu/papers.

Palomino, M., Martí-Oliet, N. y Verdejo, A. Playing with Maude. En S. Abdennadher y C. Ringeissen, editores, *Fifth International Workshop on Rule-Based Programming, RULE 2004, Aachen, Germany*, volumen por determinar de *Electronic Notes in Theoretical Computer Science*. Elsevier (2004).

Petri, C. A. Concepts of net theory. En *Mathematical Foundations of Computer Science*, páginas 137–146. Mathematical Institute of the Slovak Academy of Sciences (1973).

Pnueli, A. The temporal logic of programs. En *Proceedings of the* 18^{th} *IEEE Symposium on Foundations of Computer Science*, páginas 46–57. IEEE Computer Society Press (1977).

Quielle, J. P. y Sifakis, J. Specification and verification of concurrent systems in CESAR. En M. Dezani-Ciancaglini y U. Montanari, editores, *International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6–8, 1982, Proceedings*, volumen 137 de *Lecture Notes in Computer Science*, páginas 337–351. Springer-Verlag (1982).

Russinoff, D. M. A case study in fomal verification of register-transfer logic with acl2: The floating point adder of the amd athlon processor. En W. A. H. Jr. y S. D. Johnson, editores, *Formal Methods in Computer-Aided Design, Third International Conference, FMCAD 2000, Austin, Texas, USA, November 1-3, 2000, Proceedings*, volumen 1954 de *Lecture Notes in Computer Science*, páginas 3–36. Springer-Verlag (2000).

Saïdi, H. y Shankar, N. Abstract and model check while you prove. En N. Halbwachs y D. Peled, editores, *Computer Aided Verification. 11th International Conference, CAV'99, Trento, Italy, July 6-10, 1999, Proceedings*, volumen 1633 de *Lecture Notes in Computer Science*, páginas 443–454. Springer-Verlag (1999).

Scott, D. y Strachey, C. Towards a mathematical semantics for computer languages. *Computers and Automata*, páginas 19–46 (1971).

Shoenfield, J. R. *Degrees of Unsolvability*. North-Holland (1971).

Smullyan, R. M. *Theory of formal systems*, volumen 47 de *Annals of Mathematics Studies*. Princeton University Press (1961).

TARLECKI, A., BURSTALL, R. M. Y GOGUEN, J. A. Some fundamental algebraic tools for the semantics of computation. Part 3: Indexed categories. *Theoretical Computer Science*, 91(2):239–264 (1991).

URIBE RESTREPO, T. E. *Abstraction-Based Deductive-Algorithmic Verification of Reactive Systems*. Tesis Doctoral, Department of Computer Science, Stanford University (1998).

VERDEJO, A. Y MARTÍ-OLIET, N. Executable structural operational semantics in Maude. Informe técnico 134-03, Departamento de Sistemas Informáticos y Programación, Universidad Complutense de Madrid (2003).

VIRY, P. Equational rules for rewriting logic. *Theoretical Computer Science*, 285(2):487–517 (2002).

ZILLES, S. Algebraic specification of data types. Informe técnico 11, Laboratory for Computer Science, MIT (1974).