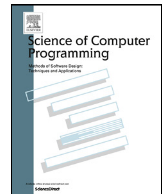
Science of Computer Programming 110 (2015) 104–118

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico



A zoom-declarative debugger for sequential Erlang programs

Rafael Caballero [a,*], Enrique Martin-Martin [a], Adrián Riesco [a], Salvador Tamarit [b]

[a] *Dpto. de Sistemas Informáticos y Computación, Fac. Informática, Universidad Complutense de Madrid, C/ Profesor José García Santesmases, 9, 28040 Madrid, Spain*
[b] *Babel Research Group, Fac. Informática, Universidad Politécnica de Madrid, Campus de Montegancedo, s/n. 28660 Boadilla del Monte, Spain*

A R T I C L E I N F O

Article history:
Received 18 March 2015
Received in revised form 24 June 2015
Accepted 30 June 2015
Available online 8 July 2015

Keywords:
Erlang
Zoom debugging
Declarative debugging

A B S T R A C T

We present a declarative debugger for sequential Erlang programs. The tool is started when a program produces some unexpected result, and proceeds asking questions to the user about the correctness of some subcomputations until an erroneous program function is found. Then, the user can refine the granularity by zooming in the function, checking the values bound to variables and the `if/case/try-catch` branches taken during the execution. We show by means of an extensive benchmark that the result is a usable, scalable tool that complements already existing debugging tools such as the Erlang tracer and Dialyzer. Since the technique is based on a formal calculus, we are able to prove the soundness and completeness of the approach.

© 2015 Elsevier B.V. All rights reserved.

1. Introduction

Erlang [12] is a programming language that combines the elegance and expressiveness of functional languages (higher-order functions, lambda abstractions, single assignments), with features required in the development of scalable commercial applications (garbage collection, built-in concurrency, and even hot-swapping). The language is used as the base of many fault-tolerant, reliable software systems. The development of these systems is a complicated process where tools such as discrepancy analyzers [23], test-case generators [27], or debuggers play an important rôle. In the case of debuggers, Erlang provides a useful trace debugger including different types of breakpoints, stack tracing, and other features. However, debugging a program is still a difficult, time-consuming task—according to a National Institute of Standards and Technology (NIST) study, software engineers spend 70–80% of their time testing and debugging [31]. Therefore alternative or complementary debugging tools are still needed.

Taking advantage of the declarative nature of the sequential subset of Erlang, in this paper we present a new debugger based on the general technique known as *declarative debugging* [33]. Also known as *declarative diagnosis* or *algorithmic debugging*, this technique abstracts the execution details, which may be difficult to follow in declarative languages, to focus on the validity of the results. This approach was first proposed in the logic paradigm [25,37], where debugging programs including features like backtracking using a traditional trace debugger can be really difficult. Soon, declarative debugging was applied to functional [26,29] and multi-paradigm [6,24] languages. Declarative debuggers of non-strict functional languages display the terms evaluated to the point required by the actual computation, something very useful in these languages which include the possibility of representing infinite data structures. More recently, the same principles have been applied

* Corresponding author.
E-mail addresses: rafacr@ucm.es (R. Caballero), emartinm@ucm.es (E. Martin-Martin), ariesco@fdi.ucm.es (A. Riesco), stamarit@babel.ls.fi.upm.es (S. Tamarit).

http://dx.doi.org/10.1016/j.scico.2015.06.011
0167-6423/© 2015 Elsevier B.V. All rights reserved.

```
1 mod(X,Y)-> (X  rem Y + Y) rem Y.

2 to_pos(L) when L >= $ A, L =< $Z -> L - $A.
3 from_pos(N)-> mod(N, 26)+ $A.

4 encipher(P, K)->from_pos(to_pos(P)+ to_pos(K)).
5 decipher(C, K)->from_pos(to_pos(C)- to_pos(K)).

6 cycle_to(N, List) when length(List)>= N ->List;
7 cycle_to(N, List)->lists:append(List,cycle_to(N - length(List),List)).

8 normalize(Str)->toupper(filter(fun isalpha/1, Str)).

9 crypt(RawText, RawKey, Func)->
10    PlainText = normalize(RawText),
11    lists:zipwith(Func, PlainText,
12                   cycle_to(length(PlainText), normalize(RawKey))).

13 encrypt(Text, Key)->crypt(Text, Key, fun encipher/2).
14 decrypt(Text, Key)->crypt(Text, Key, fun decipher/2).
```

Fig. 1. Erlang code implementing Vigenère cipher.

to object-oriented [7,22] programming languages. The basic idea of this scheme consists of asking questions to the user about the validity of the subcomputations associated to a computation that has produced an error until an erroneous fragment of code is located.

Our tool provides features such as trusting, support for higher-order functions and built-ins, and "don't know" answers. In fact, checking the comparison from [30], we can see that our debugger has most of the features in state-of-the-art tools, although we still lack some elements such as a graphical user interface, or more navigation strategies. Typical declarative debuggers in functional programming usually look for incorrect functions. However, in real-world programs, functions can be quite intricate. Thus, detecting that a function is erroneous, although helpful, still leaves to the user the problem of finding *where* is the error inside the body function. For this reason we allow the user to employ declarative debugging also *inside* an erroneous function in order to detect the precise piece of code that caused the bug. We call this feature *zoom debugging* and constitutes, to the best of our knowledge, the first work where such a feature is described. The present work extends and completes the results in [8], where we introduced the foundations of the debugger, and in [9], where a short explanation of the tool was presented, by:

- Comparing in detail the different debugging techniques that can be used in Erlang, focusing on their strengths and flaws.
- Giving a detailed description of *zoom debugging*, which was just succinctly presented in [9]. This feature is not present in any other declarative debugger.
- Compiling a study of the applicability of the system to real programs. We have applied our debugger to a wide range of small-medium applications *developed by others* and real-world projects in the Erlang community obtained from *GitHub*. In all these cases our tool has been able to find the errors using a small number of questions. This gives us confidence on the usability and potential of the tool.

The rest of the paper is organized as follows: Section 2 introduces Erlang by means of a motivating example. Section 3 describes different techniques to debug Erlang and the similarities with our approach. Section 4 describes the main features of our tool, while Section 5 focuses on zoom debugging. Section 6 recounts the main theoretical results. Section 7 presents the experimental results obtained when using our tool, while Section 8 concludes and presents the future and ongoing work.

2. Motivating example

We start introducing a running example taken from the Erlang community. It is important to remark that the bug was already in the program and has not been introduced by the authors.

Fig. 1 presents a *Vigenère cipher* [5] written in Erlang, extracted from the programming chrestomathy *Rosetta Code*.[1] The program exports the functions `encrypt/2` and `decrypt/2` that, given a text and a key, encipher or decipher it, respectively. The Vigenère cipher is a simple and well-known method for encrypting alphabetic text by adding *modulo 26*

[1] http://rosettacode.org/mw/index.php?title=Vigen%C3%A8re_cipher.

(or subtracting, in the case of decrypting) the letters of the text and key in the same position. For example, to cypher the text `Attack tomorrow dawn` using the key `lemon` we have:

Text	ATTACKTOMORROWDAWN	
Key	LEMONLEMONLEMONLEM	$+_{26}$
Cryptogram	LXFOPVXAABCVAKQLAZ	

For instance, the `T` (19th letter of the alphabet A–Z) in second position is cyphered using `E` (4th letter of the alphabet), so the resulting cyphered letter is `X`, the 23rd letter of the vocabulary: $(19+4)$ *mod* $26 = 23$. The letter in third position is also `T` but now it is cyphered using `M` (12th letter), so the result is the letter `F`, the 5th letter of the vocabulary: $(19+12)$ *mod* $26 = 5$. As keys are usually shorter than the text to cypher, we need to repeat the key until it exactly matches the length of the text.

Besides the exported functions `encrypt/2` and `decrypt/2`, the program in Fig. 1 contains some other functions that are used only internally. Function `mod/2` computes X *mod* Y guaranteeing that the result is positive even when X is negative, which may happen when deciphering. Functions `from_pos/1` and `to_pos/1` convert between positions in the alphabet and letters in ASCII encoding to perform the calculations. Functions `encipher/2` and `decipher/2` encipher and decipher a letter of the text using a letter of the key, respectively. `cycle_to/2` is used to repeat a list until it reaches exactly the given length. Function `normalize/1` converts a string into uppercase and removes all the characters that are not letters. Function `crypt/3` repeats the key until it reaches the length of the text and then applies a function `Func` passed as an argument to the letters in the text and the extended key. Finally, functions `encrypt/2` and `decrypt/2` encrypt and decrypt a text using a key by simply calling `crypt/3` with the functions `encipher/2` and `decipher/2`. The complete program also contains other simple functions like `isalpha/1` to check whether the argument is a lowercase or uppercase letter, or `toupper/1`, which returns a letter or string in uppercase, but we have omitted them because they do not play any relevant rôle. Observe that the code contains calls to built-ins and standard library functions, e.g. `length/1` (a built-in) in line 6 or `append/2` in line 7 and `zipwith/3` in line 11, both from module `lists`.

A user encrypts the text using the expression `vigenere:encrypt("Attack tomorrow dawn", "lemon")`. The system evaluates this expression but instead of the expected cryptogram it returns an exception indicating that the function responsible for the error is `lists:zipwith/3`:

```
> vigenere:encrypt("Attack tomorrow dawn", "lemon").
** exception error: no function clause matching
lists:zipwith(#Fun<vigenere.2.122144413>,[],"ON") (lists.erl,line 436)
     in function  lists:zipwith/3 (lists.erl, line 436)
     in call from lists:zipwith/3 (lists.erl, line 436)
```

This is an unexpected result, an initial symptom indicating that there is an erroneous function in the program. However, the message does not provide useful information to fix the bug, since the function `lists:zipwith/3` is correct. Additionally, no line from the user's code is shown in the message. In the next sections we show how different debuggers help in the task of finding the bug.

3. Debugging Erlang

We present in this section the different methodologies to debug Erlang programs, including our own debugger. Then, we present the related work and its relation with our approach.

3.1. Different debugging approaches

We show here how to debug the error in the code from Fig. 1 in several different ways:

Trace debugger. The trace debugger is the standard approach for many programming languages. The OTP/Erlang system comes with a classical trace debugger[2] with both graphical and command line interfaces. This debugger supports the whole Erlang language—including concurrency—allowing programmers to establish (possibly conditional) breakpoints in their code to stop the execution and then inspect the trace step by step, watch the stack trace of function calls, and inspect variables and other processes, among other features.

A typical execution of the trace debugger starts with a breakpoint in the first function called, in our case `vigenere:encrypt/2`; a snapshot of this debugging session is shown in Fig. 2. Then, the user runs the program and realizes that the second and the third argument of the call to `lists:zipwith/3` differ in their sizes. Therefore the next step would be to establish two additional breakpoints, one in each rule of function `cycle_to/2`. After running again

[2] http://www.erlang.org/doc/apps/debugger/debugger_chapter.html.

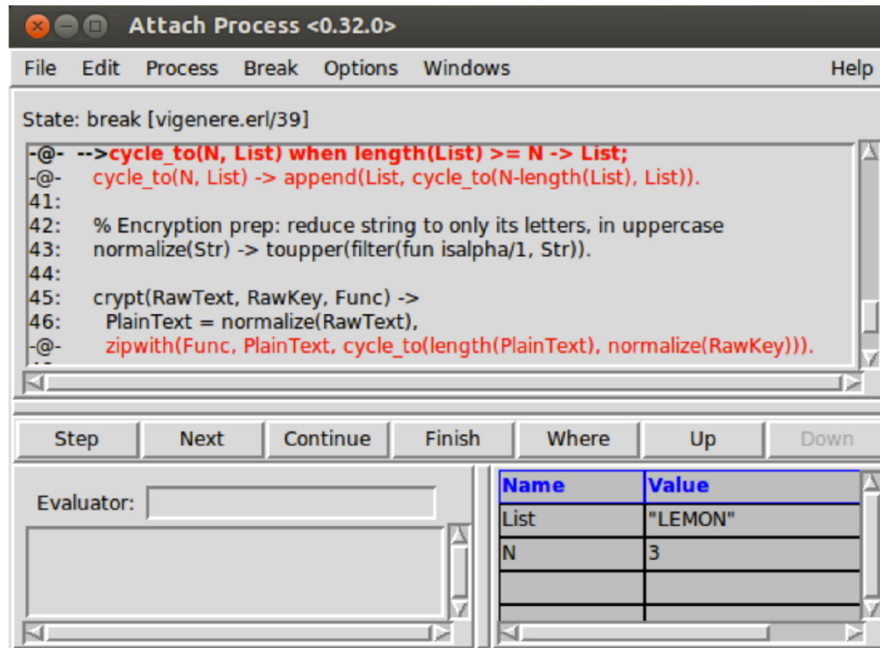


Fig. 2. Trace debugging session with the code of Fig. 1.

the program, the user would observe that the first clause is not returning what it was expected. In this case, two runs are required and three breakpoints (one first and two in the second run) are provided by the user. Of course, the trace debugger allows other possibilities instead of simply proceeding top-down in the execution. For instance the user could decide that the first step should be to put a breakpoint in the call to `lists:zipwith/3` since the reported error is pointing to this function. As before the user would discover that the actual problem is that `lists:zipwith/3` is invoked with lists of different lengths, but he would still need to set new breakpoints in `cycle_to/2` in order to find out the bug.

There is an alternative to the graphical user interface, that is to build the whole trace. For this example, the whole trace, with calls and returns, has 572 items, which makes the debugging very complicated. An alternative would be to define the functions of interest, that in our case could be `crypt/2` and `cycle_to/2`. This reduces considerably the size of the trace (from 572 to 10). However the user has to think which are the functions of interest and declare them to be traced, which is not always easy and quick.

***Print* debugger.** Although placing `print` statements is not a formal debugging technique, it is a very common practice, and hence it is worth considering it here. It consists in placing a `print` command in the "suspicious" functions, so the user can inspect the values of some variables while following the control flow.

This case is similar to the previous technique, but the user has to build his own trace using calls to the output printer function `io:format/2`. The user would need to run his code twice and insert in his code three calls to `io:format/2`. First, one previous to the call to function `lists:zipwith/3` and then, two inside `cycle_to/2`, which requires some restructuring of the code. Once the bug is found and fixed, the calls to the printing function must be removed and the code restructured again.

Dialyzer. Another tool included in the OTP/Erlang system is the *DIscrepancy AnaLYZer for ERlang programs* (Dialyzer) [23], a completely automatic tool that performs static analysis to identify software discrepancies and bugs such as definite type errors [32], race conditions [13], unreachable code, redundant tests, unsatisfiable conditions, and more. Although useful in many cases, in this example it cannot find any error in the code:

```
$ dialyzer vigenere.erl
  Checking whether the PLT .dialyzer_plt is up-to-date... yes
  Proceeding with analysis... done in 0m0.92s
done (passed successfully)
```

Our proposal, `edd`. Our debugger `edd` (**E**rlang **D**eclarative **D**ebugger) gets many ideas from the field of declarative debugging [33]. This kind of debugger asks questions to the user about the results returned by function calls, although in our case it also includes the possibility of zooming inside the code of the functions. First, let us consider a debugging session

in our tool which does not use the zoom feature. The answer `y` (*yes*) indicates that the result is correct, `n` (*no*) indicates that the result is wrong, while `t` (*trust*) can be understood as "consider that all the calls to this function are correct." More information about possible answers will be presented in Section 4.

```
> edd:dd("vigenere:encrypt(\"Attack tomorrow dawn\",\"lemon\")").
...
vigenere:normalize("Attack tomorrow dawn")= "ATTACKTOMORROWDAWN"? y
vigenere:normalize("lemon")= "LEMON"? [y/n/t/d/i/s/u/a]: y
vigenere:encipher(82, 69)= 86? t
vigenere:cycle_to(13, "LEMON")= "LEMONLEMONLEMON"? n
vigenere:cycle_to(3, "LEMON")= "LEMON"? n

Call to a function that contains an error:
vigenere:cycle_to(3, "LEMON")= "LEMON"
Please, revise the first clause:
cycle_to(N, List) when length(List)>= N ->List.
```

The first two questions about `normalize` are correct because this function is supposed to convert to uppercase and remove all non-letter characters. The next question about `encipher` is presented in terms of ASCII codes. The call is correct because encrypting `R` (ASCII 82, 17th letter of the alphabet A–Z) using `E` (ASCII 69, 4th letter) must return `V` (ASCII 86, 21st letter), i.e., $(17+4)$ *mod* $26=21$. The definition of this function is quite straightforward and the user indicates that it can be trusted for the rest of the session. The next question is a call to `cycle_to` that returns an unexpected result: it should produce a string of length 13 and not 15 as it happens. Analogously, in the next question the result should be `LEM` (length 3) instead of `LEMON` (length 5), and thus the user marks this call as incorrect.

Only five questions have been asked in order to find the bug. Note that no additional configuration or information is provided by the user, only the initial call with the erroneous behavior.

It is interesting to see the main advantages of the declarative debugging approach w.r.t. the trace debugger. In the first case, our declarative debugger provides more clarity and simplicity of usage for sequential programs. In the trace debugger, programmers must compile their code with the `debug_info` option, and set some breakpoints where they want to stop the execution. From those points they can proceed step by step, checking whether the results of the functions or the arguments and bindings are the expected ones. If they skip the evaluation of a function but they discover it returns a wrong value, they have to restart the session to enter and debug the function body. This is a burden even with conditional breakpoints, since conditions in the trace debugger must be coded as boolean functions in a module. The advantage of the declarative debugger is that, starting only from an expression returning a wrong value, it finds a buggy function by simply asking about the results of the functions in the computation, avoiding low-level details. It focuses on the intended meaning of functions, which is something very clear to programmers (or debuggers). Observe that the same or very similar questions are also implicitly considered by the user using trace debugging, when the result after each method call is examined. However, there are some points that favor the questions asked by our debugger:

1. Questions are simplified because every nested call has been replaced by its result in advance. Observe for instance that the questions for `cycle_to` contain no reference to `append` or `length`, which are used in its definition. More important, the questions also avoid references to the nested recursive call, which simplifies the task of the user.
2. The built-in navigation strategies abstracts away the execution order from the debugging phase, and the number of questions are reduced in order to find the error with the minimum number of questions.
3. If one question is still too complicated the user can answer `d`, meaning *don't know*. In this case the debugger tries to look for new questions that can lead to the bug. Only if the question is really needed will it be asked again. In the trace debugger skipping a difficult method call means possibly missing the error, with no possibility of coming back without restarting the session.

Another important advantage of declarative debugging is that it is *stateless*: each debugging question can be answered independently, without any reference to the previous ones. On the other hand, when debugging using breakpoints every choice is affected by the previous ones, which can make them hard to use.

Moreover, we present in Section 5 an improved declarative debugger that allows the user to inspect inside functions to find a more specific source of error. Comparing this zoom declarative debugger and the trace debugger, both provide tools to detect a bug hidden in the code of a concrete sequential function. With a trace debugger, programmers proceed instruction by instruction checking whether the bindings, the branches selected in `if`/`case`/`try-catch` expressions or inner function calls in the code of a suspicious function are correct. The zoom declarative debugger fulfills a similar task, asking only about the correctness of bindings and `if`/`case`/`try-catch` branch selections since it abstracts from inner function calls—they have been checked in the previous phase. Furthermore, the navigation strategy automatically guides the debugging session over the *zoom* computation without the participation of the user to choose breakpoints and steps, finally pointing out a very concrete expression causing the error. Therefore, it provides a simpler and clearer way of finding bugs

in concrete functions than the trace debugger, although the complexity of the elements involved (meaning of variables, selected `if`/`case`/`try-catch` branches, ...) is similar.

The main limitation of our proposal is that it cannot provide help for finding errors due to concurrency issues. In these cases, the trace debugger, which includes several features in this context, is the only possibility. In the near future we plan to extend our debugging setting so it can deal with concurrency, as it is a key feature of Erlang. However, we consider that `edd` is useful even for concurrent programs, since often the concurrent features are described in a specific module, while the basic behavior of the application is defined in a sequential module that can be debugged with `edd`. Concretely, in Section 7 we successfully apply our `edd` debugger to real Erlang projects from the *GitHub* repository, some of them using concurrency. Therefore we are confident about the applicability of our tool in concurrent programs.

3.2. Related work

Although not directly related to our proposal, it is worth mentioning the existence of other tools that can be useful for testing and detecting errors in Erlang programs. In particular, model checking tools have also received much attention in the Erlang community [20,3]. *McErlang* [4] is one of the most powerful model checkers nowadays due to its support for a very substantial part of the Erlang language. In this tool, a bug corresponds to a property that is not fulfilled by the implementation. McErlang includes its own trace debugger which can be used after a bug is reported to find its origin in the source code. Therefore, using McErlang, a user can easily find some unexpected behavior. However, the user has not direct information of where it should go in the source code in order to fix the bug. Regarding testing tools, the most important ones are *EUnit* [11] and *Quviq QuickCheck* [21]. The EUnit tool is included in the OTP/Erlang system, and allows users to write their own unit tests to check whether functions return the expected values. On the other hand, Quviq QuickCheck is a commercial software that automatically generates and checks thousands of unit tests from properties stated by programmers. Thus, testing tools allow users to detect function calls leading to an unexpected result. These buggy function calls can be a starting point for our declarative debugger, which will find the bug origin in the source code.

Finally, other non-conventional approaches to debugging have been studied in the literature, like abstract diagnosis [1] or symbolic execution [19], but these techniques are not closely related to declarative debugging, so we do not provide a detailed comparison.

4. Introducing EDD

The technique described in the previous section has been implemented in Erlang. The tool, called `edd` (**E**rlang **D**eclarative **D**ebugger), is publicly available at https://github.com/tamarit/edd.

The user starts the debugger when an unexpected result is obtained during an expression evaluation. Then, `edd` proceeds by asking questions about the correctness of some calls occurred during the evaluation. These questions are simply of the form `call = value?`, asking whether the function or lambda abstraction application `call` should evaluate to `value`. As mentioned in the previous section, the arguments of `call` are always values since nested calls are previously evaluated in order to simplify the questions.

Internally, the debugger structures the calls in the form of a computation tree, with the root corresponding to the initial expression, and the children of each call representing those calls occurring in the body of the corresponding code (function or lambda abstraction) that have been evaluated during the computation. Building the computation tree is the first step before starting with the debugging process. The goal of the debugging process is to locate a call pointed out as incorrect but such that all its children correspond to correct calls.[3] This is called a buggy call, and corresponds to a function that is causing an error [33].

In order to choose the next question, that is, in order to select the next call from the computation tree, the system can internally employ two different navigation strategies [34,35] to present the questions, *Top Down Heaviest First* and *Divide & Query*. In both cases each selection step starts with a computation tree with incorrect root (initially this root corresponds to the initial symptom detected by the user):

Top Down Heaviest First selects among the root children the call that corresponds to the biggest subtree (hence *heaviest*).

Divide & Query selects the node rooting a subtree that contains half the number of the total nodes in the computation tree. If such a node does not exist it selects the best approximation.

If the chosen call is incorrect then its associated subtree (which already has an incorrect root) is used to recursively continue the process. Otherwise, the selected node and its subtree are removed from the computation tree, and another call is selected following the same selection criterium. Top Down Heaviest First sessions usually present more questions to the user, but they are presented in a logical order, while Divide & Query leads to shorter sessions of unrelated questions. The user can choose their preferred strategy when the debugger is started.

[3] A correct call is either a call returning the expected value for some expected arguments or a call receiving unexpected arguments. The latter is called *inadmissible*, as we explain below.

In response to the debugger questions, the possible answers are:

- *yes (y)*: the evaluation is correct.
- *no (n)*: the evaluation is not correct.
- *trusted (t)*: the user knows that the function or lambda abstraction used in the evaluation is correct, so further questions about it are not necessary. Note that a list of trusted functions can be indicated before starting the debugging process.
- *inadmissible (i)*: the question does not apply because the arguments should not take these values. The behavior of the tool is the same as the one for *yes*.
- *don't know (d)*: the answer is unknown. The question is skipped, but might be asked again if it is required for finding the error.
- *switch strategy (s)*: changes the navigation strategy.
- *undo (u)*: reverts to the previous question.
- *abort (a)*: finishes the debugging session.

The tool includes a memoization feature that stores the answers *yes*, *no*, *trusted*, and *inadmissible*, preventing the system from asking the same question twice. It is worth noting that although *don't know* is used to ease the interaction with the debugger it may introduce incompleteness, so we have not considered it in our proofs. If there are questions marked as unknown at the end of a debugging session and the bug has not been identified, these questions will be asked again to the user. If they still refuse to answer, the debugger will show a list of candidates to be erroneous based on the provided answers.

The result of a complete debugging session (without *don't know* answers) is either an incorrect user-defined function or an incorrect lambda abstraction. Although in both cases the tool points out the clause that produced the bug, the error might be in another clause. We will show in the following section how to find more specific errors.

5. Zoom debugging

This section first presents zoom debugging by means of an example. Then, we list all the possible errors detected by the tool in this mode.

5.1. An example using zoom

When debugging the Vigenère cypher from Fig. 1, `edd` found that the buggy function was `cycle_to`. This function is very simple because it has only two clauses consisting of one statement each. Therefore, in this case it is easy to find the bug in the code and fix it. However, in many cases the exact location inside a buggy function that is causing the error may still be difficult to find. The existence of different paths that the execution can follow inside the function body due to the existence of nested `case` expressions, or the possible values taken by the variables, that depend in general on the values of several previous variables, complicate the situation. For example, consider the following program for managing stocks:

Example 5.1. Fig. 3 presents an Erlang program defining a module `stock`. It only exports the function `check_orders/2` which, given a list of orders and a stock of items, returns the list of items the company has to buy to supply all the orders. Both orders and stocks are represented as lists of tuples `{item, Name, Quantity}`, where `Name` is the name of the item and `Quantity` the number of items of that kind. We assume that a valid order cannot contain two tuples for the same `Name`, and analogous for any valid stock. For handling a list of orders, we simply unify the orders (add all the entries with the same `Name` in different orders into one `item` tuple), and call `check_order/2`, which behaves the same as `check_orders/2` but considering only one order. `check_order/2` uses the predefined `lists:zf/2` function, that performs a filter and a mapping at the same time.[4] `lists:zf/2` takes a unary function `F` and a list `L`, and for each element `E` of `L`:

- If `F(E) = true`, the element remains in the list.
- If `F(E) = false`, the element is discarded.
- If `F(E) = {true, Value}`, the element is replaced by `Value`.

The program uses `check_item/2` to filter and map the order into the list of items to buy. Given an `item` tuple and a stock, `check_item/2` looks for a tuple with the same name in the stock using `lists:keyfind/3` function. If there is not such a tuple in the stock, all the elements must be bought. If the tuple exists in the stock but its quantity is less than the required quantity, we must buy the difference. If the tuple appears in the stock and there are enough items, the `item` tuple can be discarded because we do not need to buy anything. Finally, `test/0`

[4] `zf/2` is currently an undocumented and obscure function of the `lists` module, but there is a proposal for renaming and documenting it http://erlang.org/pipermail/erlang-patches/2013-April/003907.html.

```
1 test() -> stock:check_orders(
2   [[{item, water, 3},{item, rice, 3}], [{item, rice, 4}]],
3   [{item, rice, 5}, {item, bajoqueta, 8}]
4 )

5 check_order( Order, Stock )->
6   lists:zf( fun(X)->check_item(X,Stock) end, Order ).

7 check_item( Needed = {item, Name, Q1}, Stock )->
8   ItemStock = lists:keyfind(Name, 1, Stock),
9   case ItemStock of
10     {item, Name, Q2 }->
11        if Q1 > Q2 ->{true, {item, Name, Q1 - Q2}};
12           true ->false
13        end;
14     false ->{true, Needed}
15  end.

16 check_orders( List_of_orders, Stock )->
17   Flat_orders = lists:flatten( List_of_orders ),
18   Unique_orders = unify_orders( Flat_orders ),
19   check_order( Unique_orders, Stock ).

20 unify_orders( [])-> [];
21 unify_orders( [{item,Name,Quantity}|R])->
22  (...)
```

Fig. 3. Erlang program for managing stocks.

is a function to check that the program is working. It calls `check_orders/2` with two orders of ingredients for a *paella* dish (`[[{item, water, 3},{item, rice, 3}]` and `[{item, rice, 4}]]`), considering that the current available stock consists of `[{item, rice, 5},{item, bajoqueta, 8}]`.[5] Observe that the orders need a total of 7 packs of rice and 3 bottles of water, while the stock only contains 5 packs of rice and no water. Thus, the expected result should be `[{item, water, 3},{item, rice, 2}]`. However, the program returns the erroneous value `[{item,water,3},{item,rice,7}]`.

The `edd` tool will find that `check_item` is the buggy function, but as it has 9 lines, one `case` expression and a nested `if` statement it is still not easy to fix the bug. In situations like this one, `edd` can help programmers by zooming in the buggy function and asking more questions in order to locate the concrete statement causing the bug. For *zoom* debugging the tool uses four kinds of questions:

- Questions about `case` expressions. It shows several options and the user must select the first wrong item (otherwise, the last option indicates that everything is correct):
  1. **The context of the expression**, that is, the values of the variables used to evaluate the argument and all the branches until the successful one.
  2. **The argument value**, that is, the value used to select the suitable case branch and to continue the evaluation.
  3. **The successful branch**, that is, the actual branch selected to continue with the evaluation. When this value is wrong the tool asks to the user for the index of the expected branch. It also asks about the correctness of some clause guards as explained below.
  4. **The generated bindings**, that is, the bindings generated by matching the argument value with the pattern in the successful branch. They are only asked when they exist.
  5. **The final value**, that is, the value reached by evaluating the whole case expression.
- Questions about `if` and `try/catch` statements and about function clauses. They are treated as specific cases of `case` expressions.
- Questions about the correctness of guards in the function clauses. In this case the possible answers are the usual ones explained in Section 4.
- Question about variable bindings. The tool shows a list of variable bindings performed in a pattern-matching expression, and the user answers using the standard options (Section 4).

Using `edd` to debug the `stock` program from Fig. 3, the user finds that the buggy function is `check_item` (this part of the session is omitted for simplicity). Then the user zooms in looking for a more precise bug:

[5] *Bajoqueta* is the word used in the Spanish region of Valencia for a kind of green bean essential in the *paella valenciana*.

```
Call to a function that contains an error:
stock:check_item({item, rice, 7}, [{item, rice, 5},
{item, bajoqueta, 8}])= {true, {item, rice, 7}}

Please, revise the first clause:
(... omitted)
Do you want to continue the debugging session inside this function?
  [y/n]: y

(1) For the case expression:
case ItemStock of
  {item, Name, Q2}->
      if Q1 > Q2 ->{true, {item, Name, Q1 - Q2}};
         true ->false
      end;
  false ->{true, Needed}
end
Is there anything incorrect?
1.- The context:
        ItemStock = false
        Needed = {item,rice,7}
2.- The argument value: false.
3.- Enter in the second clause.
4.- The final value: {true,{item,rice,7}}.
5.- Nothing.
[1/2/3/4/5/d/s/u/a]? 5

(2) Given the context:
        Name = rice
        Stock = [{item,rice,5},{item,bajoqueta,8}],
the following variable is assigned:
ItemStock = false.
Is this correct? [y/n/d/i/s/u/a]: n

This is the reason for the error:
Variable ItemStock is badly assigned false in the expression:
ItemStock = lists:keyfind(Name, 1, Stock)(Line 8).
```

Note that we have drawn squares to distinguish user answers, and circled numbers are not part of the debugging session but they have been inserted because they will be used in Section 7. In the first question, when `Needed` is `{item,rice,7}` and `ItemStock` is `false` the `case` expression must enter in the second branch and return `{true,{item,rice,7}}`. Therefore, we select the option `5` to say that everything is correct. In the second question, when `Name` is `rice` and the `Stock` is `[{item,rice,5}, {item, bajoqueta,8}]` we expect the value of `ItemStock` to be `{item,rice,5}`. In the evaluation `ItemStock` is bound to `false`, so we mark the step as incorrect pressing the key `n`. After 2 answers,[6] the debugger marks the assignment of the variable `ItemStock` as the source of the error, showing the whole instruction and the line in the source program. With this information, it is easy to discover where the error really is: in the index `1` we pass to the function `lists:keyfind/3`. We want `ItemStock` to be the item in the `Stock` with `Name` in the second element of the tuple. As usual in programming, we thought that the first element had index 0, so `lists:keyfind/3` should look for `Name` in the element index 1. However, Erlang does not follow that convention but starts indices from 1. Therefore, to fix the problem we have to write the assignment as `ItemStock = lists:keyfind(Name, 2, Stock)`.

5.2. *Errors detected by* `edd`

Once the zoom features of the tool have been described, we can list all the errors detected by `edd`. When the first step of the debugging session finishes, a wrong function (either a defined function or a lambda abstraction) is detected. When using zoom debugging the following errors can be found:

- A variable is not correctly assigned.
- The argument expression of a `case` or `try` expression is not correctly defined.
- The expression in charge of returning the final value does not compute the appropriate value.
- The pattern or the guard, only one of them, of a clause in a `case`, `if`, or `try` expression or in a function or lambda abstraction definition is not correctly defined.

[6] Note that the first question involved 4 smaller questions; we will consider, in our measurements in the next section, that we answered 5 questions.

6. Theoretical basis

We start this section explaining in more detail the basis of declarative debugging and their relation to the language semantics. Then, we outline the ideas behind the proofs for soundness and completeness of our tool, which can be found in the extended version of the article [10].

6.1. Semantic basis of declarative debugging

As mentioned above, declarative debugging is a well-known debugging technique. It consists of two steps: first, a tree representing the computation is built; then, this tree is traversed by asking questions to the user. As we have shown in the previous sections, the tree construction phase remains hidden to the user, who is only concerned with the questions. However, in our theoretical setting the tree allows us to prove the completeness and soundness of our tool, because it is built following a formal calculus.

More specifically, it follows a calculus we have defined for Core Erlang, an intermediate language that represents Erlang programs internally without syntactic sugar. Although we do not present the details here (see [10] for more information), we can deal with simplified Erlang programs while keeping track of all the errors in the original Erlang program as described in Section 5.2. The semantics of Erlang is informally described in [2], but there is no *official* formalized semantics. However, several authors have proposed and used different formalizations in their works, most of them aiming to cover the concurrent behavior of the language. In [20], Huch proposes, for a subset of Erlang, an operational *small step* semantics based on *evaluation contexts* to only perform reductions in certain points of the expression. It covers single-node concurrency (spawning and communication by messages between processes in the same node) and reductions that can yield runtime errors. However, it does not cover other sequential features of the language like lambda abstractions or higher-order functions. Another important small-step semantics for Erlang is proposed in Fredlund's Ph.D. thesis [18]. This semantics is similar to [20] and also uses evaluation contexts but covers a broader subset of the language including single-node concurrency, runtime errors, and lambda abstractions. However, it also lacks support for higher-order features. To overcome the limitations of Fredlund's single-node semantics when dealing with distributed systems, [15] proposes a semantics based on Fredlund's but adding another top-level layer describing nodes. This distributed multi-node semantics for Erlang was further refined and corrected in [36]. Besides standard operational semantics, other approaches have been proposed to formalize Erlang semantics like the one based on congruence in [14], which works with partial evaluation of Erlang programs.

Hence, the existing semantic calculi for Erlang are usually small-step semantics that do not cover all the sequential features but focus on concurrency. If we use these small-step semantics for declarative debugging, the questions shown to the user would be slight transitions between expressions like "`V = f(3*2,2+1) -> V = f(6,2+1)?`." In order to provide more meaningful questions, it is convenient to use a big-step semantics leading to questions that contain complete evaluations from expressions to values like "`f(6,3) = 43?`," which are simpler and closer to the expected behavior of the program that users have in mind. This is the main reason why we have devised our own Erlang big-step semantic calculus, as well as including important sequential features like lambda abstractions and higher-order functions. Moreover, an interesting contribution of our calculus is the debugging of exceptions. Declarative debugging of programs throwing exceptions has already been studied from an operational point of view for the Mercury debugger [24], for Haskell in its declarative debugger Buddha [28], and for Java programs [22]. However, these approaches are operational and do not provide a calculus to reason about exceptions: in Mercury, exceptions are considered another potential value and thus functions throwing exceptions are included as standard calls; Buddha uses a program transformation to build the appropriate data structures while executing the program; finally, the approaches for Java return and propagate exceptions without defining the inference rules. Similarly, several calculi handling exceptions, like the ones in [17,16], have been proposed for functional languages. `edd` is, for the best of our knowledge, the first tool that uses a calculus to perform declarative debugging of exceptions, allowing us to reason about exceptions as standard values.

6.2. Soundness and completeness

Formally, a debugging session can be seen as the process of comparing the expected behavior of the program, also known as *intended interpretation* ($\mathcal{I}$ in the rest of the section), and the actual computation represented by a suitable computation tree T. An oracle, the user in our case, must determine whether a subcomputation result is correct (that is, it is valid with respect to $\mathcal{I}$).

In the first phase of our debugger the set $\mathcal{I}$ can be seen as the union of two parts $\mathcal{I}_{fun} \cup \mathcal{I}_{\lambda}$, containing respectively the expected results produced by function calls and lambda abstractions occurring in the program for each different tuple of input values. If the *zoom* feature is employed then $\mathcal{I}$ must be enriched adding new sets describing the expected variable bindings inside the selected function, the `if`/`case`/`try-catch` branches that should occur in a given context, as well as the selected function rule for the given arguments.

In order to obtain the computation tree our debugger proceeds as follows:

1. The initial expression e that produced the initial erroneous result e' is (implicitly) introduced as part of the program as body of a function `main/0`, with `main` a new function name.

2. Then, the debugger builds a tree T representing the computation `main/0` $\rightarrow e'$. This tree is obtained from the proof calculus.

The nodes in the tree T are called *correct* if they are entailed from the information in $\mathcal{I}$, and *incorrect* otherwise. A *buggy node* (which generalizes the idea of buggy call presented before) is thus an incorrect node with only correct children. The goal of the debugger is to find a buggy node; the fragment of code associated to the buggy node is then pointed out as the cause of the error. The overall idea of our theoretical foundations is common to many declarative debuggers:

- The root of T is assumed incorrect (it represents the initial symptom).
- It is straightforward to ensure that any tree with incorrect root contains a buggy node, which corresponds to the completeness result.
- Moreover, the relation with the proof calculus ensures that a buggy node corresponds to a proof inference with correct premises and incorrect conclusion, which can only happen when an incorrect fragment of code has been applied. This establishes the soundness of the proposal.

Those interested in the proof calculi supporting the theory or in the detailed explanation of the theoretical results and their proofs can consult [10].

7. Experimental results

In this section we measure the effectiveness of the Erlang Declarative Debugger. We have used, first, a set of small and medium-size sequential Erlang programs that solve classical problems, listed in Table 1: Miller–Rabin primality test, Sieve of Eratosthenes, Caesar cipher, Run-length encoding, etc. These programs have been extracted from the *Rosetta Code* website (http://rosettacode.org) and modified to have a bug.[7] The inserted bugs are small mistakes that developers can easily make and are usually hard to locate: use the wrong arithmetic operation (e.g. $+/-$), use an incorrect guard (e.g. $\leq/<$), use incorrect constant values, select the incorrect field of a register, mix the values returned by consecutive branches in a `case` expression, etc. We have completed this first set of examples with our `stock` program from Section 5.

Additionally, we have looked for real-world errors by (i) asking the Erlang community to provide bugs found in their projects, and (ii) traversing the *GitHub* repository looking for fixed bugs in purely sequential but also concurrent projects—see Table 2. For each error found in the *GitHub* repository (actually, only the *io_lib_format* has been provided from our “call for bugs” to the Erlang community) we have written an initial term that reveals the error,[8] making sure that the function call was exported in the module. Note that when measuring the lines of codes in Table 2 we have considered only the minimum number of files involved in the problem, although the projects are in general much bigger (some of them more than 30,000 lines of code).

Then, we have used `edd` to find the wrong function in these problems, and also the zoom feature to spot the bug more precisely. In some examples of the first set (*ackermann*, *dutch*, *rfib*, and *vigenère*) the zoom debugger was not applicable because the wrong function was very simple, so the bug was easily found by inspecting the code of the buggy function once our tool had pointed it out. For all the tests, `edd` has spotted the bug added to the programs, and the time consumed to generate the computation trees has been reasonable—no more than 10 seconds in a computer with an Intel® Core™ 2 Quad CPU at 2.83 GHz and 4 GB of physical memory under Erlang/OTP version 17. The source code of all the programs (or a link to the GitHub repository where the example can be found), as well as the debugging sessions can be found in the folder *examples* at https://github.com/tamarit/edd.

Tables 1 and 2 show the results of the debugger for the programs of the two sets. For each one, the table shows the lines of source code (LOC), the number of nodes of the computation tree (Nodes), the complexity of the debugging session using the Divide & Query (D&Q) and Top Down Heaviest First (TD) strategies for each step of the debugger (wrong functions and zoom), the memory consumption in kilobytes (Size), and whether Dialyzer found an error. The complexity of the session is illustrated by the number of questions (Q), the value of the most complex question in the session (X), and the total complexity (T), obtained by adding up the single complexity of all the questions. In order to measure the complexity of a question, we sum up the size of the data structures and the number of bindings, clauses, and lambda abstractions involved in the question (plus a base complexity of 1). Note that questions related to `case` expressions contain several questions that must be answered in order (the last option just indicates that the answer to all of them is correct); from the complexity point of view, we consider each one of these questions as a different one. That is, if the user indicates that the ith question is wrong we consider i answers were required; if the user answers that everything is correct we consider that all the questions were answered.

Example 7.1. We present the complexity of the zoom session in Section 5.1. The first question, preceded by ①, forced the user to answer 4 questions. The first one involved (i) a data structure with 3 elements, so the complexity due to data

[7] Note that the Vigenère program from Section 2 was not modified because the bug was already present in the original code.

[8] In fact, in most of the cases this initial term has been obtained from a test suite in the same project or from the issue messages that pointed out the symptom of an erroneous function.

Table 1
Results of `edd` and Dialyzer with small/medium programs.

Program	LOC	Wrong functions								Zoom								Dialyzer
		Nodes	D&Q			TD			Size (KB)	Nodes	D&Q			TD			Size (KB)	
			Q	X	T	Q	X	T			Q	X	T	Q	X	T		
24_game	73	19	4	6	15	3	9	16	30.0	9	9	17	59	9	17	59	32.6	✓
ackermann	13	91	7	1	7	8	1	8	74.7	–		–			–		–	
align_cols	40	32	3	40	108	3	40	108	340.4	4	3	67	112	3	67	112	25.9	
caesar	38	147	2	2	3	3	4	7	152.4	5	3	4	9	3	4	9	14.2	
complex	60	7	2	9	18	2	9	18	18.6	6	2	11	18	3	8	22	13.9	
dutch	41	123	12	46	522	11	46	476	272.2	–		–			–		–	
miller_rabin	80	152	6	6	12	10	6	24	174.4	7	4	4	9	4	4	9	27.1	
rfib	6	288	11	1	11	12	1	12	190.3	–		–			–		–	
rle	44	69	5	2	6	19	2	22	225.0	8	3	6	9	3	6	9	29.9	
roman	24	9	3	1	3	6	1	6	17.1	9	8	14	70	8	14	75	44.9	✓
sieve	37	31	6	46	161	13	46	303	63.1	7	4	15	19	4	15	19	28.8	
stock	61	14	5	28	113	5	34	127	33.3	7	5	13	31	5	13	31	20.9	
sum_digits	16	7	2	1	2	6	1	6	13.8	4	2	3	5	2	3	5	13.1	
ternary	91	64	7	18	106	10	18	127	76.7	7	5	8	30	4	8	23	25.4	
turing	72	49	7	12	58	9	13	83	77.2	10	5	14	37	7	14	56	29.8	
vigenère	55	197	5	1	5	8	5	12	181.3	–		–			–		–	

Table 2
Results of `edd` and Dialyzer with real-world programs.

Program	LOC	Wrong functions								Zoom								Dialyzer
		Nodes	D&Q			TD			Size (KB)	Nodes	D&Q			TD			Size (KB)	
			Q	X	T	Q	X	T			Q	X	T	Q	X	T		
dns	1710	1		–			–		8.1	2	1	40	40	1	40	40	13.9	
erlson$_1$	1158	8	3	21	34	3	17	30	25.2	8	5	8	21	3	8	17	28.5	
erlson$_2$	377	2	1	11	11	1	11	11	10.4	4	2	4	2	2	4	2	15.9	
etorrent	1160	4	2	9	13	2	9	13	12.4	4	2	6	12	2	6	12	14.1	✓
io_lib_format	720	25	5	23	78	5	22	69	49.0	15	6	4	15	3	4	15	39.3	
luerl$_1$	1335	3	1	16	16	1	16	16	11.4	5	5	8	28	5	8	28	16.4	
luerl$_2$	682	6	3	6	17	1	5	5	14.6	5	5	5	17	5	5	17	17.4	
luerl$_3$	1880	20	5	10	33	1	6	6	33.6	5	5	6	18	5	6	18	17.7	
queue	487	3	2	38	65	2	38	65	12.8	5	4	1	4	4	1	4	18.9	∼
random	115	2	2	5	6	2	5	6	9.8	11	8	9	42	2	5	9	20.9	
redbug_msc	368	42	5	49	135	5	43	194	283.0	6	4	10	33	2	9	17	34.5	∼

structures is 4 (the 3 elements plus the tuple constructor); (ii) we have 2 variables to check, but (iii) we have no clauses involved and (iv) no lambda abstractions; hence, the total complexity is 7. The next question is much simpler: it involves no data structures, no matching, no clauses, and no lambda abstractions, so the total complexity is 1. The third one just involves 2 clauses, so its final complexity is 3. Finally, the fourth question just involves two tuples with 4 elements, so the complexity due to data structures is 6, while the total complexity is 7.

The next question, preceded by ②, involved 1 list, 2 tuples, and 6 elements, so the complexity due to constructors is 9, and 3 bindings are involved. Hence, the total complexity is 13. Wrapping up this zoom-debugging session, we had 5 questions, a complexity of 31, and the maximum complexity was 13.

Note that the debugger finds the wrong functions using a very small number of questions compared to the number of nodes in the computation tree, i.e., the number of function and lambda abstraction applications in the computation. Considering the total complexity of the debugging sessions, we observe that D&Q is usually a better option for debugging wrong functions than TD: in 14 programs out of 27 D&Q produces smaller complexities, whereas TD obtains smaller complexities than D&Q in only 5 cases. The most extreme examples of both situations are *sieve* in (161 vs. 303), *redbug_msc* (135 vs. 194), *luerl*$_2$ (17 vs. 5) and *luerl*$_3$ (33 vs. 6).

Regarding bug detection using zoom inside functions, the performance of both strategies is almost equal: D&Q obtains debugging sessions with a smaller total complexity in 3 programs, whereas TD is better in 4 programs. However, when TD performs better the difference is larger, namely in *random* (42 vs. 9) and *redbug_msc* (33 vs. 17). In the majority of the programs (16 out of 23 in which zoom debugging is applicable) both D&Q and TD performs the same from the point of view of total complexity. In these cases the number is mainly the same, with TD using slightly less questions (the most extreme case is *random*, where D&Q asks 8 questions and TD only 2). Finally, observe that the maximum complexity is 49 (in *redbug_msc*) in the wrong functions phase and 67 (*align_cols*) in the zoom phase. In both cases the complexity is due to

the size of the data structures, involving several lists. We could answer to these questions in approximately 1 minute even though we did not know how they were implemented, just understanding what the functions were supposed to do, so we consider that the complexity of the questions should not prevent the users from using declarative debugging.

In order to have a better comparison, the last column of Tables 1 and 2 contains the results of applying the automatic tool Dialyzer to the buggy programs. In the programs *24_game*, *roman*, and *etorrent* Dialyzer finds type inconsistencies in the code that are related to the bugs, and that information helps to point out the erroneous code (marked as ✓). For the programs *queue* and *redbug_msc* Dialyzer shows warning messages about incorrect type specifications and improper lists, but they are not related to the bug and cannot be used to find the erroneous code (marked as ∼). However in the rest of programs Dialyzer cannot find any inconsistency and returns a *"done (passed successfully)"* message.

An important limitation of declarative debugging is the growth of the computation tree, which can need a great amount of memory for long running programs or programs involving big data structures. In order to evaluate the memory consumption of our tool we include a column *Size* for *Wrong functions* and *Zoom* in Tables 1 and 2, which contains the size in KB of the computation trees. The first conclusion we extract is that `edd` uses a small amount of memory, less than 350 KB in the tested programs. The explanation is that the programs do not run for long time, and therefore their debugging trees do not have a large number of nodes (the maximum is 288 nodes). However, for some bugs and programs the tree could contain millions of nodes, and if the involved data structures are big then some nodes could require megabytes of memory. For these situations where the memory consumed by the tree is greater than the available RAM in the computer, our tool could integrate some of the techniques proposed in [22]. Namely, the computation tree could be stored in an external database, therefore supporting trees of arbitrary size, or the debugging session could use a partially constructed tree generated *on-the-fly*. Another conclusion we extract from the *Size* columns is that there is no relation between the total debugging session complexity and the size of the computation tree. The reason is that the total complexity of a debugging session takes into account only those nodes visited by the strategy, whereas the size considers all the nodes in the tree, even those not involved in any question. Finally, it becomes clear that the computation trees for zoom debugging have more complex nodes than the computation trees for finding wrong functions: for a similar number of nodes, computation trees for zoom debugging need more memory (see for example in Table 1 the *Wrong functions* column for *complex* and *sum_digits* and the *Zoom* column for *miller_rabin*, *sieve*, *stock* and *ternary*; all of them with 7 nodes). This fact was expected, as nodes in computation trees for zoom debugging need to store more information than their *wrong function* counterparts.

Considering the good results of `edd` for the two sets of programs, we think it can be an interesting tool for programmers when debugging the sequential part of real Erlang programs, supplementing the standard trace debugger of the OTP/Erlang system and tools like Dialyzer. The results in Tables 1 and 2 also sustain our choice of standard strategies: D&Q for finding wrong functions, as it usually asks less questions than TD, and TD for finding bugs inside functions, since it is slightly better than D&Q and moreover it shows the questions in an order closer to the evaluation one, which users might find clearer.

8. Concluding remarks and ongoing work

Debugging is usually the most time-consuming phase of the software life cycle [31], yet it remains a basically unformalized task. This is unfortunate, because formalizing a task introduces the possibility of improving it, since it provides information about the details that usually are left as trivial and allows us to prove properties of our tools. With this idea in mind we propose a formal debugging technique that structures Erlang executions and asks questions to the user in order to find bugs in sequential Erlang programs. Since the debugger only requires knowing the intended meaning of the program functions, the team in charge of the debugging phase does not need to include the actual programmers of the code. This separation of rôles is an advantage during the development of any software project.

Although most of the applications based on Erlang employ the concurrent features of the language, the concurrency is usually located in specific modules, and thus specific tools for debugging the sequential part are also interesting. Our debugger locates the error based only on the intended meaning of the program code, and thus abstracts away the implementation details. The main limitation of the proposal is that an initial unexpected result must be detected by the user, which implies in particular that it cannot be used to debug non-terminating computations unless a timeout is defined, losing in this way the completeness of the result.

We have used these ideas to implement a tool that supports different navigation strategies, trusting, built-in and library functions, and zoom debugging among other features. Zoom debugging is specially interesting since, to the best of our knowledge, this is the first declarative debugger that finds bugs with different levels of granularity. It has been used to debug several buggy medium-size and real-world programs, presenting an encouraging performance. Moreover, we have introduced it in our Master's degree course to collect feedback on usability. More information can be found at https://github.com/tamarit/edd.

Currently we are extending the current framework to debug concurrent Erlang programs. This extension will require new rules in the calculus to deal with functions for creating new processes and sending and receiving messages, as well as the identification of new kinds of errors that the debugger can detect. However, it is worth noticing that the debugger presented in this paper can be used as well in the sequential part of concurrent programs. Thus, we think that the debugger presented here will still be useful when the concurrent debugger has been completed.

A different line of future work is to take advantage of the information provided by the user during the debugging session in order to derive tests or properties. A first and straightforward approach could be to generate *EUnit* test-cases [11] from the user answers to questions regarding the validity of methods calls:

1. If the user answers that the result r is valid, then EDD can generate automatically an EUnit test-case indicating that for the given parameters the result must be equal to r.
2. Otherwise, if the user indicates result r is unexpected, we can generate a *EUnit* test-case that indicates that for the given parameters the result must be different from r.

In this way, after the debugging session the user obtains a set of test-cases that can be applied for checking the application. Moreover, this idea can also be used the other way round: the debugger could take a set of *EUnit* test-cases as input, and employ the information contained in the tests for answering automatically some user questions. Apart from tests generated by the user, the set of test-cases could include as well the tests generated by the debugger itself in previous debugging sessions, thus employing *EUnit* as a kind of answer database that can partially substitute the user as oracle during the debugging process.

A natural generalization of this idea would be to allow the user to introduce properties that can determine the validity of a method call as part of the debugging process. These properties could be saved and used both in the current and in future debugging sessions. Moreover, if such properties are introduced in the format required by some testing tool like *Quviq QuickCheck* [21], then the tester could be used afterwards for trying to locate new bugs. Once more, an interesting future line of research could be to reverse this process, allowing EDD to employ properties defined in a Quviq QuickCheck file for answering automatically some questions during the debugging process.

Acknowledgements

We thank Shayan Pooya and Anthony Ramine for their collaboration providing helpful examples of real bugs through the Erlang community mailing list. We also thank the anonymous reviewers for their suggestions and comments, which we believe that have greatly improved the quality of the paper.

Research partially supported by European Union project POLCA (STREP FP7-ICT-2013.3.4 610686); MICINN Spanish projects TIN2013-44742-C4-1-R, TIN2013-44742-C4-2-R, TIN2013-44742-C4-3-R, StrongSoft (TIN2012-39391-C04-04) and VIVAC (TIN2012-38137-C02-02); Madrid regional projects N-GREENS Software-CM (S2013/ICE-2731), SICOMORo-CM (S2013/ICE-3006) and UCM grant GR3/14-910502.

References

[1] M. Alpuente, D. Ballis, F. Correa, M. Falaschi, An integrated framework for the diagnosis and correction of rule-based programs, Theor. Comput. Sci. 411 (47) (2010) 4055–4101.
[2] J. Armstrong, M. Williams, C. Wikstrom, R. Virding, Concurrent Programming in Erlang, 2nd edition, Prentice–Hall, Englewood Cliffs, New Jersey, USA, 1996.
[3] T. Arts, C.B. Earle, J.J.S. Penas, Translating Erlang to muCRL, in: Proceedings of the International Conference on Application of Concurrency to System Design, ACSD 2004, IEEE Computer Society Press, 2004, pp. 135–144.
[4] C. Benac Earle, L.A. Fredlund, Recent improvements to the McErlang model checker, in: Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG, ERLANG 2009, ACM, New York, NY, USA, 2009, pp. 93–100.
[5] A. Beutelspacher, Cryptology, MAA Spectrum, Mathematical Association of America, 1994.
[6] R. Caballero, A declarative debugger of incorrect answers for constraint functional-logic programs, in: S. Antoy, M. Hanus (Eds.), Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia, ACM Press, 2005, pp. 8–13.
[7] R. Caballero, C. Hermanns, H. Kuchen, Algorithmic debugging of Java programs, in: F. López-Fraguas (Ed.), Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain, Electron. Notes Theor. Comput. Sci. 177 (2007) 75–89.
[8] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, A declarative debugger for sequential Erlang programs, in: M. Veanes, L. Vigan (Eds.), Proceedings of the 7th International Conference on Tests and Proofs, TAP 2013, in: Lecture Notes in Computer Science, vol. 7942, Springer, 2013, pp. 96–114.
[9] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, Edd: a declarative debugger for sequential Erlang programs, in: E. Ábrahám, K. Havelund (Eds.), 20th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS 2014, in: Lecture Notes in Computer Science, vol. 8413, Springer, 2014, pp. 581–586.
[10] R. Caballero, E. Martin-Martin, A. Riesco, S. Tamarit, A zoom-declarative debugger for sequential Erlang programs (extended version), tech. rep. 02/14, Departamento de Sistemas Informáticos y Computación, June 2014.
[11] R. Carlsson, M. Rémond, EUnit: a lightweight unit testing framework for Erlang, in: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, ERLANG 2006, ACM, New York, NY, USA, 2006, pp. 1–1.
[12] F. Cesarini, S. Thompson, Erlang Programming: A Concurrent Approach to Software Development, O'Reilly Media, Inc., 2009.
[13] M. Christakis, K. Sagonas, Static detection of race conditions in Erlang, in: M. Carro, R. Pena (Eds.), Practical Aspects of Declarative Languages, PADL 2010, in: Lecture Notes in Computer Science, vol. 5937, Springer, 2010, pp. 119–133.
[14] N.H. Christensen, Domain-specific languages in software development—and the relation to partial evaluation, Ph.D. thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark, July 2003.
[15] K. Claessen, H. Svensson, A semantics for distributed Erlang, in: Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, ERLANG 2005, ACM, New York, NY, USA, 2005, pp. 78–87.
[16] R. David, G. Mounier, An intuitionistic lambda-calculus with exceptions, J. Funct. Program. 15 (1) (January 2005) 33–52.
[17] P. de Groote, A simple calculus of exception handling, in: M. Dezani-Ciancaglini, G.D. Plotkin (Eds.), Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications, TLCA 1995, in: Lecture Notes in Computer Science, vol. 902, Springer, 1995, pp. 201–215.
[18] L.-A. Fredlund, A framework for reasoning about Erlang code, Ph.D. thesis, The Royal Institute of Technology, Sweden, August 2001.

[19] R. Hähnle, M. Baum, R. Bubel, M. Rothe, A visual interactive debugger based on symbolic execution, in: C. Pecheur, J. Andrews, E.D. Nitto (Eds.), 25th IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, ACM, 2010, pp. 143–146.
[20] F. Huch, Verification of Erlang programs using abstract interpretation and model checking, SIGPLAN Not. 34 (9) (Sep. 1999) 261–272.
[21] J. Hughes, QuickCheck testing for fun and profit, in: M. Hanus (Ed.), Practical Aspects of Declarative Languages, in: Lecture Notes in Computer Science, vol. 4354, Springer, 2007, pp. 1–32.
[22] D. Insa, J. Silva, An algorithmic debugger for Java, in: M. Lanza, A. Marcus (Eds.), Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010, IEEE Computer Society, 2010, pp. 1–6.
[23] T. Lindahl, K. Sagonas, Detecting software defects in Telecom applications through lightweight static analysis: a war story, in: W.-N. Chin (Ed.), Proceedings of the 2nd Asian Symposium on Programming Languages and Systems, APLAS 2004, in: Lecture Notes in Computer Science, vol. 3302, Springer, 2004, pp. 91–106.
[24] I. MacLarty, Practical declarative debugging of Mercury programs, Master's thesis, University of Melbourne, 2005.
[25] L. Naish, Declarative diagnosis of missing answers, New Gener. Comput. 10 (3) (1992) 255–286.
[26] H. Nilsson, How to look busy while being as lazy as ever: the implementation of a lazy functional debugger, J. Funct. Program. 11 (6) (2001) 629–671.
[27] M. Papadakis, K. Sagonas, A PropEr integration of types and function specifications with property-based testing, in: Proceedings of the 2011 ACM SIGPLAN Erlang Workshop, ACM Press, 2011, pp. 39–50.
[28] B. Pope, Declarative debugging with Buddha, in: V. Vene, T. Uustalu (Eds.), 5th International School on Advanced Functional Programming, AFP 2004, in: Lecture Notes in Computer Science, vol. 3622, Springer, 2005, pp. 273–308.
[29] B. Pope, A declarative debugger for Haskell, Ph.D. thesis, The University of Melbourne, Australia, 2006.
[30] A. Riesco, A. Verdejo, N. Martí-Oliet, R. Caballero, Declarative debugging of rewriting logic specifications, J. Log. Algebr. Program. 81 (7–8) (2012) 851–897.
[31] RTI, The economic impacts of inadequate infrastructure for software testing, tech. rep. RTI project number 7007.011, National Institute of Standards and Technology, 2002.
[32] K. Sagonas, J. Silva, S. Tamarit, Precise explanation of success typing errors, in: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, ACM, New York, NY, USA, 2013, pp. 33–42.
[33] E.Y. Shapiro, Algorithmic Program Debugging, ACM Distinguished Dissertation, MIT Press, 1983.
[34] J. Silva, A comparative study of algorithmic debugging strategies, in: G. Puebla (Ed.), Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2006, in: Lecture Notes in Computer Science, vol. 4407, Springer, 2007, pp. 143–159.
[35] J. Silva, A survey on algorithmic debugging strategies, Adv. Eng. Softw. 42 (11) (2011) 976–991.
[36] H. Svensson, L.-A. Fredlund, A more accurate semantics for distributed Erlang, in: Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop, ERLANG 2007, ACM, New York, NY, USA, 2007, pp. 43–54.
[37] A. Tessier, G. Ferrand, Declarative diagnosis in the CLP scheme, in: P. Deransart, M.V. Hermenegildo, J. Maluszynski (Eds.), Analysis and Visualization Tools for Constraint Programming: Constraint Debugging (DiSCiPl Project), in: Lecture Notes in Computer Science, vol. 1870, Springer, 2000, pp. 151–174.