EDD: A Declarative Debugger for Sequential Erlang Programs*

Rafael Caballero[1], Enrique Martin-Martin[1], Adrian Riesco[1], and Salvador Tamarit[2]

[1] Universidad Complutense de Madrid, Madrid, Spain
rafa@sip.ucm.es, emartinm@fdi.ucm.es, ariesco@fdi.ucm.es
[2] Babel Research Group, Universitat Politécnica de Madrid, Madrid, Spain
stamarit@fi.upm.es

Abstract. Declarative debuggers are semi-automatic debugging tools that abstract the execution details to focus on the program semantics. This paper presents a tool implementing this approach for the sequential subset of Erlang, a functional language with dynamic typing and strict evaluation. Given an erroneous computation, it first detects an erroneous function (either a "named" function or a lambda-abstraction), and then continues the process to identify the fragment of the function responsible for the error. Among its features it includes support for exceptions, predefined and built-in functions, higher-order functions, and trusting and undo commands.

1 Introduction

Declarative debugging, also known as algorithmic debugging, is a well-known technique that only requires from the user knowledge about the *intended behavior* of the program, that is, the expected results of the program computations, abstracting the execution details and hence presenting a *declarative* approach. It has been successfully applied in logic [5], functional [6], and object-oriented [4] programming languages. In [3,2] we presented a declarative debugger for sequential Erlang. These works gave rise to EDD, the Erlang Declarative Debugger presented in this paper. EDD has been developed in Erlang. EDD, its documentation, and several examples are available at `https://github.com/tamarit/edd` (check the `README.md` file for installing the tool).

As usual in declarative debugging the tool is started by the user when an unexpected result, called the *error symptom*, is found. The debugger then builds internally the so-called *debugging tree*, whose nodes correspond to the auxiliary computations needed to obtain the error symptom. Then the user is questioned

* Research supported by EU project FP7-ICT-610582 ENVISAGE, Spanish projects *StrongSoft* (TIN2012-39391-C04-04), *DOVES* (TIN2008-05624), and *VIVAC* (TIN2012-38137), and Comunidad de Madrid *PROMETIDOS* (S2009/TIC-1465). Salvador Tamarit was partially supported by research project POLCA, Programming Large Scale Heterogeneous Infrastructures (610686), funded by the European Union, STREP FP7.

E. Ábrahám and K. Havelund (Eds.): TACAS 2014, LNCS 8413, pp. 581–586, 2014.

about the validity of some tree nodes until the error is found. In our proposal, the debugger first concentrates in the function calls occurred during the computation. The goal is to find a function call that returned an *invalid* result, but such that all the function calls occurring in the function body returned a *valid* result. The associated node is called a *buggy node*. We prove in [3] that such function is a *wrong function*, and that every program producing an error symptom[1] contains at least one wrong function. An important novelty of our debugger w.r.t. similar tools is that it allows using *zoom debugging* to detect an erroneous fragment of code inside the wrong function. At this stage the user is required to answer questions about the validity of certain variable matchings, or about the branch that should be selected in a `case/if` statement for a given context. The theoretical results in [2] ensure that this code is indeed erroneous, and that a wrong function always contains an erroneous statement.

The rest of the paper is organized as follows: Section 2 introduces Erlang and EDD. Section 3 describes the questions that can be asked by the tool and the errors that can be detected. Section 4 concludes and presents the future work.

2 Erlang and EDD

In this section we introduce some pieces of Erlang [1] which are relevant for our presentation. At the same time we introduce the basic features of our tool. Erlang is a concurrent language with a sequential subset that is a functional language with dynamic typing and strict evaluation. Programs are structured using modules, which contain functions defined by collections of clauses.

Example 1. Obtain the square of a number X without using products. This is possible defining $Y = X - 1$ and considering $X^2 = (Y+1)^2 = Y^2 + Y + Y + 1$.

```
-module(mathops).
-export([square/1]).
square(0) -> 0;
square(X) when X>0 -> Y=X-1, DoubleY=X+X, square(Y)+DoubleY+1.
```

Observe that variables start with an uppercase letter or underscore. In order to evaluate a function call Erlang scans sequentially the function clauses until a match is found. Then, the variables occurring in the head are bound. In our example the second clause of the function `square` is erroneous: the underlined subterm `X+X` should be `Y+Y`. Using this program we check that `mathops:square(3)` is unexpectedly evaluated to `15`. Then, we can start EDD, obtaining the debugging session in Fig. 1, where the user answers are boxed. Section 3.1 explains all the possible answers to the debugger questions. Here we only use 'n' (standing for 'no'), indicating that the result is invalid, and 'y' (standing for 'yes'), indicating that it is valid. After two questions the debugger detects that the tree node containing the call `mathops:square(1)` is buggy (it produces and invalid result while its only child `mathops:square(0)` returns a valid result). Consequently,

[1] Note that, if the module has multiple errors that compensate each other, there is no error symptom and hence declarative debugging cannot be applied.

```
> edd:dd("mathops:square(3)").
mathops:square(1) = 3?  n
mathops:square(0) = 0?  y
Call to a function that contains an error:
mathops:square(1) = 3
Please, revise the second clause:
square(X) when X > 0 -> Y=X-1, DoubleY=X+X, square(Y)+DoubleY+1.
Continue the debugging session inside this function?  y
In the function square(1) matching with second clause succeed.
Is this correct?  y
Given the context:  X = 1
the following variable is assigned: DoubleY = 2?  n
This is the reason for the error:
Variable DoubleY is badly assigned 2 in the expression:
DoubleY = X + X (line 4).
```

Fig. 1. EDD session corresponding to the call `mathops:square(3)`

the tool points out the second clause of `square` as wrong. Next, the user is asked if zoom debugging must be used. The user agrees with inspecting the code associated to the buggy node function call. The debugger proceeds asking about the validity of the chosen function clause, which is right (the second one), and about the validity of the value for `DoubleY`, which is incorrect (it should be 0 since X=1 implies Y=0). The session finishes pointing to this incorrect matching as the source of the error. Observe that an incorrect matching is not always associated to wrong code, because it could depend on a previous value that contains an incorrect result. However, the correctness results in [3] ensure that only matchings with real errors are displayed as errors by our tool. Note in this session the improvement with respect to the trace, the standard debugging facility for Erlang programs. While the trace shows every computation step, our tool focuses first on function calls, simplifying and shortening the debugging process.

The next example shows that Erlang allows more sophisticated expressions in the function bodies, including `case` or `if` expressions.

Example 2. Select the appropriate food taking into account different preferences.

```
-module(meal).
-export([food/1]).
food(Preferences) ->
    case Preferences of
        {vegetarian,ovo_vegetarian} -> omelette;
        {vegetarian,_lacto_vegetarian} -> yogurt;
        {vegetarian,vegan} -> salad;
        _Else -> fish
    end.
```

Now we can evaluate the expression `meal:food({vegetarian,vegan})` and we obtain the unexpected result `yogurt`. This time the first phase of the debugger is not helpful: it points readily to the only clause of `food`. In order to obtain more precise information we use zoom debugging, and the debugger asks:

```
For the case expression:
(... omitted for the sake of space ... )
Is there anything incorrect?
1.- The context:  Preferences = {vegetarian,vegan}
2.- The argument value: {vegetarian,vegan}.
3.- Enter in the second clause.
4.- The bindings:  _lacto_vegetarian = vegan
5.- The final value: yogurt.
6.- Nothing.
[1/2/3/4/5/6]? 3
```

This question asks for anything invalid in the evaluation of the `case` expressions with respect to its intended meaning. It is important to mention that in the case of something wrong the answer must be the first wrong item. In our case the context and the `case` argument are correct, but we did not expect to use the second branch/clause but the third. Therefore we answer `3` indicating that this is the first error in the list. The next question is:

```
Which clause did you expect to be selected [1/2/3/4]? 3
```

As explained above, we expected to use the third clause for this context. Then the debugger stops indicating the error:

```
This is the reason for the error:
The pattern of the second clause of case expression:
case Preferences of {vegetarian, _lacto_vegetarian} -> yogurt end
```

Indeed there is an erroneous underscore in `_lacto_vegetarian`. It converts the constant into an anonymous variable, and thus the branch was incorrectly selected. The debugger has found the error, indicating that the second branch is wrong and that in particular the pattern definition is incorrectly defined. Note that, with a trace-debugger, programmers proceed instruction by instruction checking whether the bindings, the branches selected in `case`/`if` expressions or inner function calls are correct. Our tool fulfills a similar task, although it discards inner function calls—they were checked in the previous phase. Moreover, the navigation strategy automatically guides the session without the participation of the user by choosing breakpoints and steps, finally pointing out the piece of code causing the bug. Therefore, it provides a simpler and clearer way of finding bugs in concrete functions, although the complexity is similar.

3 Using the Tool

3.1 User Answers

The possible answers to the debugging questions during the first phase are:

- *yes (y)/ no (n)*: the statement is valid/invalid.
- *trusted (t)*: the user knows that the function or λ-abstraction used in the evaluation is correct, so further questions about it are not necessary. In this case all the calls to this function are marked as valid.

- *inadmissible (i)*: the question does not apply because the arguments should not take these values. The statement is marked as valid.
- *don't know (d)*: the answer is unknown. The statement is marked as unknown, and might be asked again if it is required for finding the buggy node.
- *switch strategy (s)*: changes the navigation strategy. The navigation strategies provided by the tool are explained below.
- *undo (u)*: reverts to the previous question.
- *abort (a)*: finishes the debugging session.

In the case of zoom debugging, the answer *trusted* has not any sense and it is never available, while the answers *yes*, *no*, and *inadmissible* cannot be used in some situations, for instance in compound questions about `case/if` expressions. The rest of answers are always available.

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 *don't know* is used to ease the interaction with the debugger but it may introduce incompleteness; if the debugger reaches a deadlock due to these answers it presents two alternatives to the user: either answering some of the discarded questions to find the buggy node or showing the possible buggy code, depending on the answers to the nodes marked as unknown.

3.2 Strategies

As indicated in the introduction, the statements are represented in suitable debugging trees, which represents the structure of the wrong computation. The system can internally utilize two different navigation strategies [7,8], *Divide & Query* and *Top Down Heaviest First*, in order to choose the next node and therefore the next question presented to the user. Top Down selects as next node the largest child of the current node, while Divide & Query selects the node whose subtree is closer to half the size of the whole tree. In this way, Top Down sessions usually presents more questions to the user, but they are presented in a logical order, while Divide & Query leads to shorter sessions of unrelated questions.

3.3 Detected Errors

Next we summarize the different types of errors detected by our debugger. As we have seen, the first phase always ends with a wrong function. The errors found during the zoom debugging phase are:

Wrong case argument, which indicates that the argument of a specific `case` statement has not been coded as the user expected.

Wrong pattern, which indicates that a pattern in the function arguments or in a `case/if` branch is wrong.

Wrong guard, which indicates that a guard in either a function clause or in a `case/if` branch is wrong.

Wrong binding, which indicates that a variable binding is incorrect.

4 Concluding Remarks and Ongoing Work

EDD is a declarative debugger for sequential Erlang. Program errors are found by asking questions about the intended behavior of some parts of the program being debugged, until the bug is found. Regarding usability, EDD provides several features that make it a useful tool for debugging real programs, such as support for built-in functions and external libraries, anonymous (lambda) functions, higher-order values, *don't know* and *undo* answers, memoization, and trusting mechanisms, among others. See [2,3] for details.

We have used this tool to debug several programs developed by others. This gives us confidence in its robustness, but also illustrates an important point of declarative debugging: it does not require the person in charge of debugging to know the details of the implementation; it only requires to know the intended behavior of the functions, which is much easier and more intuitive, hence allowing a simpler form of debugging than other approaches, like tracing or breakpoints.

As future work we plan to extend this proposal to include the concurrent features of Erlang. This extension requires first to extend our calculus with these features. Then, we must identify the errors that can be detected in this new framework, define the debugging tree, and adapt the tool to work with these modifications.

References

1. Armstrong, J., Williams, M., Wikstrom, C., Virding, R.: Concurrent Programming in Erlang, 2nd edn. Prentice-Hall (1996)
2. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: A zoom-declarative debugger for sequential Erlang programs. Submitted to the JLAP
3. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: A declarative debugger for sequential Erlang programs. In: Veanes, M., Viganò, L. (eds.) TAP 2013. LNCS, vol. 7942, pp. 96–114. Springer, Heidelberg (2013)
4. Insa, D., Silva, J.: An algorithmic debugger for Java. In: Lanza, M., Marcus, A. (eds.) Proc. of ICSM 2010, pp. 1–6. IEEE Computer Society (2010)
5. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
6. Nilsson, H.: How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. Journal of Functional Programming 11(6), 629–671 (2001)
7. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
8. Silva, J.: A survey on algorithmic debugging strategies. Advances in Engineering Software 42(11), 976–991 (2011)