Debugging Meets Testing in Erlang

Salvador Tamarit[1], Adrián Riesco[2], Enrique Martin-Martin[2], and Rafael Caballero[2(✉)]

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

Abstract. We propose a bidirectional collaboration between declarative debugging and testing for detecting errors in the sequential subset of the programming language Erlang. In our proposal, the information obtained from the user during a debugging session is stored in form of unit tests. These test cases can be employed afterwards to check, through testing, if the bug has been actually corrected. Moreover, the debugger employs already existing tests to determine the correctness of some subcomputations, helping the user to locate the error readily. The process, contrarily to usual debugger frameworks is cumulative: if later we find a new bug we have more information from the previous debugging and testing iterations that can contribute to find the error readily.

1 Introduction

One of the most important underlying ideas of the software development life cycle [1] is that the assets from one phase can be employed both in the next phases and in successive iterations of the cycle. For instance, the testing phase produces test cases that allow checking whether the system satisfies the initial requirements. If later the system is modified, for instance to improve its efficiency, the initial tests (or at least part of them) can be employed again to check whether the initial requirements are still verified.

However, there is a task in the software development cycle that often constitutes the exception to this rule: debugging. In spite of the introduction of tools that try to automatize the location of errors, debugging is still a manual and very time-consuming non-trivial task that requires a careful comparison between the actual and the expected results of some subcomputations. Unfortunately, the very useful information gathered during a debugging session is usually thrown away once the debugging session is finished.

Research supported by the Comunidad de Madrid project N-Greens Software-CM (S2013/ICE-2731), by the MINECO Spanish projects *StrongSoft* (TIN2012-39391-C04-04), *VIVAC* (TIN2012-38137), *CAVI-(ROSE/ART)* (TIN2013-44742-C4-(1/3)-R), *LOBASS* (TIN2015-69175-C4-2-R), and *TRACES* (TIN2015-67522-C3-3-R), and by the European Union project POLCA (STREP FP7-ICT-2013.3.4 610686).

B.K. Aichernig and C.A. Furia (Eds.): TAP 2016, LNCS 9762, pp. 171–180, 2016.
DOI: 10.1007/978-3-319-41135-4_10

We propose a modification of the general framework followed in *declarative debugging*, also known as *algorithmic debugging* [9], a debugging technique that asks questions to the user until a bug is found. In our proposal the answers given by the user are stored in the form of test cases that make persistent the valuable information obtained during a debugging session. In order to prove the applicability of the new debugging schema, we have implemented the new schema in the Erlang Declarative Debugger EDD [3]. The same ideas can enhance any declarative debugger implemented for a system allowing unit tests.

Furthermore, the relation between debugging and testing can be seen as a bidirectional collaboration. One of the major complaints about declarative debugging is the large number of questions asked to the user in order to find the bug. In our proposals, each question is compared in advance with the existing test cases, in order to determine if the answer can be entailed without further assistance from the user. We have observed that this feature is very helpful in practice, especially if there are more than one bug in the system, since the user can focus directly on the code affected by the error, disregarding the fragments of code that have been checked and found correct in previous debugging sessions.

The rest of the paper is structured as follows: Sect. 2 presents our proposal as a new general debugging framework. Section 3 describes how our tool takes advantage of test cases to improve declarative debugging, while Sect. 4 shows how test cases are generated by our declarative debugger. Finally, Sect. 5 concludes and discusses some lines of future work. The tool EDD, modified to take into account the generation and use of test cases, is publicly available at https://github.com/tamarit/edd.

2 A New General Debugging Schema

Declarative debugging is a semi-automatic debugging technique that abstracts the execution details to focus on results. It can be presented as a general schema with the following structure:

```
declarative_debugger(initialSymptom) ->
  T = execution_tree(initialSymptom)
  while (|T| ≠ 1)
      pick up a node N in T with N ≠ root(T)
      ask the oracle whether N is valid/invalid
      if N is valid:
        remove N and its subtree from T
      else
        T = subtree rooted by N
  return root(T)
```

The debugger starts when the user detects a computation returning an unexpected result, the initialSymptom. Then, it builds an *execution tree* representing the initial symptom. The nodes of the tree can be depicted with the form $C = V$, with C a computation (function evaluation) and V its computed result. A node is considered *valid* if its result is the expected for the associated computation,

and *invalid* otherwise. In particular, the root of the tree represents the computation of the initial symptom and thus it is invalid. The children of each node correspond to the subcomputations needed to obtain the result at the parent. The final goal is to locate a *buggy node*, an invalid node with valid children. The fragment of code represented by this node is then considered as the source of the error, because it has produced an erroneous output from valid inputs (the children results). Each iteration of the main loop chooses an *unknown* node N, possibly following some strategy [10], and asks to the user about its validity. If N is valid, the subtree rooted by N is removed from the tree. If it is invalid then the subtree rooted by N becomes the new debugging tree. Observe that after each iteration the size of the tree decreases, and that in every iteration its root is invalid. Then, it is possible to ensure that in a finite number of iterations we will get a tree with only one node ($|\mathrm{T}| = 1$), and that this node is a buggy node. Both operations, removing subtrees rooted by valid nodes, and replacing the tree by a subtree rooted by an invalid node, are safe, in the sense that the tree obtained after the operation contains at least one buggy node, and every buggy node in the new tree is also buggy in the original tree.

In this paper, we consider EDD [3], a declarative debugger for the sequential subset of the programming language Erlang [6] that follows this schema. The nodes of the execution trees in EDD have the form $m : f(t_1, \ldots, t_n) = r$, with m the name of an Erlang module, f a function defined in $m, t_1, \ldots, t_n$ the arguments of a call to f occurred during the computation, and r the computed result. EDD also debugs anonymous functions, and allows the user inspecting the body of functions looking for more particular errors [4] but these features are not used in this paper.

Our proposal extends the initial framework by taking into account existing test cases and also by generating new test cases following the information gathered from the user. We distinguish two kinds of test cases:[1]

- *Positive* test cases, depicted as **?assertEqual**(C,V), indicating that V is the expected result for computation C.
- *Negative* test cases **?assertNotEqual**(C,V'), indicate that V' is *not* the expected result for C.

The extended framework that we propose is presented in Fig. 1. It takes as additional input parameter a set of test cases inputTCs, used to decide initially that some of the nodes of the execution tree are valid or invalid. In particular, the lines 5 and 6 look for positive test cases that occur in the tree. These nodes are valid, and their subtrees can be safely removed. Lines 7–10 initialize outputTCs, the list that stores the new test cases obtained after each question answered by the user. It takes the initial symptom with its value as first negative test case if there is not a positive test case for the same call in the initial set of

[1] Erlang also supports test cases ?assert(...) with predicates. Our system can handle these test cases when their predicates involve equality or inequality operators, but we will focus only on **?assertEqual** and **?assertNotEqual** for simplicity in the presentation.

```
 1 declarative_debugger(initialSymptom, inputTCs) ->
 2
 3    T = execution_tree(initialSymptom)
 4
 5    for  ?assertEqual(C,V) ∈ inputTCs
 6         remove from T every subtree rooted by C=V
 7    if ∄ ?assertEqual(initialSymptom, V) ∈ inputTCs
 8         outputTCs = [?assertNotEqual(initialSymptom, value(initialSymptom)]
 9    else
10        outputTCs = []
11
12    let T' be the smallest subtree of T verifying
13         invalidTC(root(T'),inputTCs ∪ outputTCs)
14    T=T'
15
16    while (|T| ≠ 1)
17         pick up a node N ≡ (C=V) from T with N ≠ root(T)
18         ask the oracle whether N is valid/invalid
19
20         if N is valid
21           outputTCs = [?assertEqual(C,V) | outputTCs]
22           remove from T the subtree rooted by N
23         else
24           outputTCs = [?assertNotEqual(C,V) | outputTCs]
25           T = subtree rooted by N
26
27    return (root(T), outputTCs \ inputTCs)
28
29 invalidTC(C=V,inputTCs ) ->
30   return (?assertEqual(C,V') ∈ inputTCs and V'≠V) or
31           (?assertNotEqual(C,V) ∈ inputTCs)
```

Fig. 1. Declarative debugging with test cases

test cases.[2] Line 13 looks for nodes that can be detected as invalid from the information contained in the test cases. This is the task of the Boolean function invalidTC defined in lines 29–31, which receives a node and a set of input test cases and return **true** if it is possible to determine that the node is invalid from the information contained in the test cases. As the function indicates, a node in the tree can be pointed out as invalid in two situations:

1. If there is a positive test case for the same computation but with a different associated value (then the value contained in the node is not the correct result).
2. If there is a negative test case for the same computation and for the same value, that is, the test case indicates directly that the result is unexpected.

[2] The positive test case, if exists, indicates that the initial symptom is wrong indicating also the expected value. This makes the addition of the negative test case redundant.

It is safe to replace the tree T by the subtree T' rooted by a node verifying any of these two conditions. Line 12 of Fig. 1 takes the smallest tree T' with these characteristics; although this tree might not be unique, the completeness property for debugging trees with an invalid root [4] ensures that any of these trees will reveal an error. This is important, because a smaller tree means, generally, less questions to the user. The rest of the code is similar to the original schema. Finally, both the buggy node and the new test cases are returned.

This general idea has been put into practice extending EDD using EUnit tests [5]. EUnit belongs to the general testing framework family known as *unit testing*, a well-established testing methodology that allows users to indicate the expected values obtained when executing a function with some specific arguments. Thus, the tests generated by EDD can be executed using EUnit, which allows the user to check that the problem has been actually solved after the error has been corrected, simply running the generated tests. Notice that the tests will check not only the main result, that could be checked readily by the user, but also all the intermediate results obtained during the debugging process. This is important because correcting an error sometimes introduces inadvertently a new one. The exhaustive checking of the computation helps to check that this is not the case. Note also that the tests generated are not affected by code changes since they are only expressing the intended interpretation of one particular function.

```
1  -module(quicksort).
2
3  qs(_, []) -> [];
4  qs(F, [E|R]) ->
5    {A, B} = partition(F, E, R),
6    qs(F, B) ++ [E] ++ qs(F, A).
7
8  leq(A, B) -> A =< B.
9
10 partition(_, _, []) -> {[], []};
11 partition(F, E, [H|T]) ->
12   {A, B} = partition(F, E, T),
13   case F(H, E) of
14     true  -> {[H|A], B};
15     false -> {A, B}
16   end.
17
18 quicksort_test() ->
19   ?assertEqual(qs(fun leq/2, []), []),
20   ?assertEqual(qs(fun leq/2, [1]), [1]),
21   ?assertEqual(qs(fun leq/2, [7,1]), [1,7]),
22   ?assertEqual(qs(fun leq/2, [7,8,1]),
         [1,7,8]).
```

Fig. 2. Code for the quicksort function and its corresponding tests

3 When Declarative Debugging Met Testing

We illustrate these ideas with the quicksort module presented in Fig. 2, which is an adaptation of the code in [7]. The module contains 3 functions: qs, leq and partition. The function qs takes as arguments a binary predicate F representing the notion of *order* and a list, and returns the list ordered using the QuickSort algorithm. Lists in Erlang are represented as sequences of elements [E_1,..., E_n] or [H|T] where H is the first element of the list (called *head*) and T is the rest of the list (called *tail*). Erlang also allows the use of *tuples*, represented as sequences of

elements enclosed in curly braces: {E$_1$,..., E$_n$}. As usual in functional languages, the function qs is defined by two clauses that are tried in top-down order by applying *pattern matching*. The first clause (line 3) returns the empty list if the argument is an empty list. The second clause (lines 4–6) accepts a non-empty list [E|R], splits the tail R using E as pivot and recursively sort the partitions A and B. The function leq is simply a wrapper of the predefined operator =<. Finally, the function partition takes as input parameters an order function F, a pivot element E and a list and divides the latter into two according to the pivot and the order function. Notice the usage of a **case** expression to decide in which partition the head H must be inserted. The module quicksort includes also a testing function (quicksort_test) defining four simple positive unit tests obtained from a previous debugging session. The third test in this function (line 21) fails, hence revealing that there is at least one error. We start the debugging process by introducing the failing test case in EDD:

```
> edd:dd("quicksort:qs( fun quicksort:leq/2, [7,1] )").
```

Following the schema of Fig. 1, EDD builds the execution tree corresponding to this computation, which can be examined in Fig. 3(A),[3] and uses the same test cases to prune the tree, obtaining that nodes 3, 5, 6, and 7 are entailed as valid (marked with diagonals in the corners) and can be safely removed together with their subtrees. The pruning of the associated subtrees removes 5 of the 8 initial possible questions, and leaves only the shaded nodes in the debugging tree. Then, the following question about node 2 is asked to the user:

```
quicksort:partition(fun quicksort:leq/2, 7, [1]) = {[1], []}? y
```

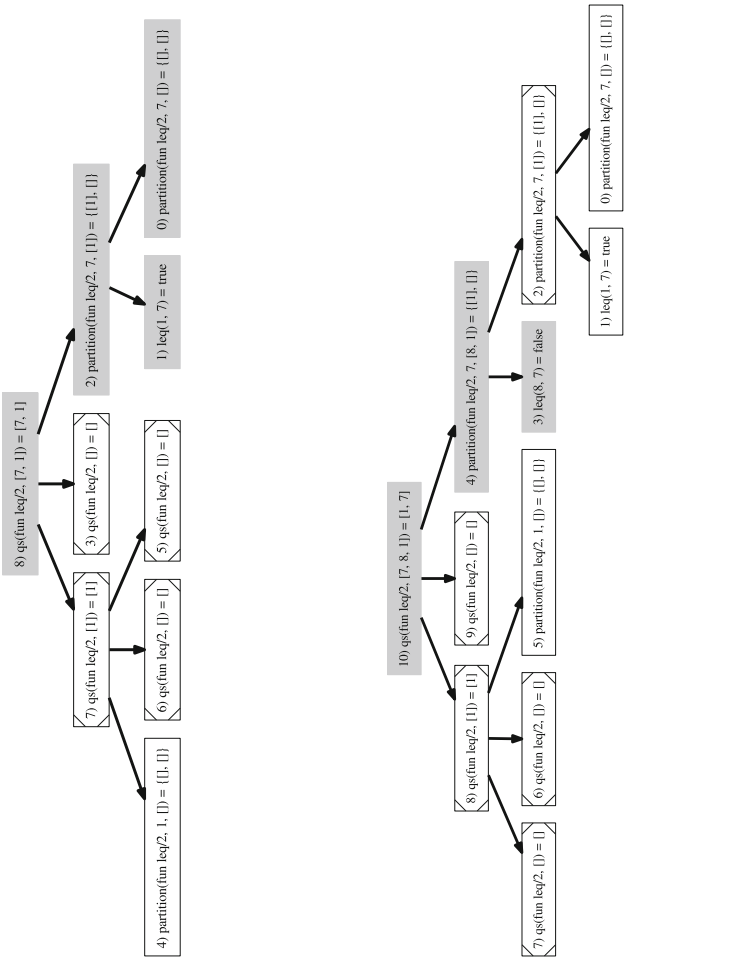
The intended meaning of partition is to split the input list ([1]) into two lists, one containing the elements less than or equal to 7, and another one with the elements greater than 7. The result of the call is valid so the user answers y (*yes*). At this point, after just one user answer, the debugger identifies node 8 as buggy:

```
Call to a function that contains an error:
quicksort:qs(fun quicksort:leq/2, [7, 1]) = [7, 1]
Please, revise the second clause:
qs(F, [E | R]) -> {A, B} = partition(F, E, R),
                  qs(F, B) ++ [E] ++ qs(F, A).
```

The error is in line 6, which should be qs(F, A) ++ [E] ++ qs(F, B). Before starting the next section we assume that the error has been corrected.

It is important to note that, since EDD is based in a formal semantics, it is possible to prove the soundness and completeness of the technique. The key point for proving these properties is the existence of an *intended interpretation*, that corresponds with the semantics that the programmer had in mind when implementing the system; by comparing this intended interpretation with the actual execution we are able to discover the buggy node. In our work, the soundness and completeness results are easily extended by ensuring that the test cases are

[3] Nodes' modules are not shown for the sake of clarity.



(A) Tree of first session (B) Tree of second session

Fig. 3. Debugging trees of the EDD sessions

a subset of the intended interpretation, that can be used to appropriately prune the tree before asking the user to answer the rest of the questions. More details on the proofs are available in [4].

4 When Testing Met Declarative Debugging

The previous debugging session not only finds a bug but it also generates one new positive test case[4] **?assertEqual**(partition(fun leq/2, 7, [1]), {[1], [ ]}). As outlined

[4] Notice that, as explained in Sect. 2, a negative test case **?assertNotEqual** for the root qs(fun leq/2, [7,1]) is not generated as the test suite already contains a positive test for it (see line 21 in Fig. 2).

in the introduction, this test case can be used later in the software development cycle, as well as by EDD in later debugging sessions. In fact, if we execute the test cases again after fixing the bug detected in the previous section, we find out that the fourth unit test of quicksort_test (line 22) still fails, indicating that another bug is hidden in the program:

```
> quicksort:qs( fun quicksort:leq/2, [ 7, 8, 1 ] ).
[1,7]
```

Again we start the debugger, using this unit test as initial symptom. In this case, the execution tree contains 11 nodes—Fig. 3(B)—with 10 potential questions to be asked. Thanks to the original test suite together with the unit test case generated in the previous debugging session, the debugger prunes the tree, keeping only the 3 grey nodes, i.e., 2 potential questions. As explained before, nodes with diagonals in the corners correspond to valid results w.r.t. the test cases.

Hence, the debugger presents the following debugging session:

```
> edd:dd( "quicksort:qs( fun quicksort:leq/2 , [7,8,1 ] )" ).
quicksort:partition(fun quicksort:leq/2, 7, [8, 1]) = {[1], []}? v
What is the value you expected? {[1],[8]}
quicksort:leq(8, 7) = false?  t

 Call to a function that contains an error:
 quicksort:partition(fun quicksort:leq/2, 7, [8, 1]) = {[1], []}
 Please, revise the second clause
partition(F, E, [H | T]) ->
    {A, B} = partition(F, E, T),
    case F(H, E) of
      true -> {[H | A], B};
      false -> {A, B}
    end.
```

The first question is about the validity of the partition of `[8,1]` using 7 as pivot. The result `{[1],[]}` is wrong, so the user could simply answer `n` (*no*). However, our debugger introduces a refinement on the schema of Fig. 1. Since positive test cases are more informative than negative ones and they allow EDD to prune more nodes, we have introduced in EDD an option (letter `v` from *value*) that allows the user to type the correct value in addition to answering `no`. This command produces an **?assertEqual** test case instead of the negative one. In this session the user decides to use this option and indicates that the correct value should be `{[1],[8]}`. The second question is answered by the user with `t`, meaning *trusted*. In EDD this answer indicates that the user considers that function `leq` is correct so all the nodes containing a call to `leq` must be marked as valid, therefore generating as many positive **?assertEqual** unit tests as distinct calls to `leq` are found in the tree—in this case there are two calls. Finally, EDD points out to the second error (line 15 in Fig. 2). We realize that the **false** branch inside `partition` is incorrect: the first element `H` of the list, which is greater than the

pivot E, must be appended to B. Fixing this second bug will result in replacing line 15 by **false** $->${A, [H|B]}.

As well as detecting the buggy function, EDD has extended the test suite with four unit tests:

```
?assertEqual(partition(fun leq/2, 7, [1]), {[1], []}),
?assertEqual(partition(fun leq/2, 7, [8, 1]), {[1],[8]}),
?assertEqual(leq(8, 7), false),
?assertEqual(leq(1, 7), true).
```

Of course, although very useful, employing/generating test can be disabled in EDD using options:

- *not_load_tests*: do not use existing EUnit tests to prune the execution tree.
- *not_save_tests*: do not generate EUnit tests from the user answers.

5 Conclusions and Ongoing Work

Debugging is usually a manual task that involves the comparison of the actual and the expected behavior of the debugged system. Unfortunately, this very valuable information, which requires a great amount of time and effort, is discarded once the error is found. In fact, debuggers are often considered auxiliary tools and are not properly integrated in the software development cycle. In this paper we have shown how to improve this situation by employing an algorithmic debugger that stores the information extracted from the user during the debugging sessions in the form of unit tests. These tests are especially useful to check whether the error has been effectively corrected, and can become part of the tests produced during the testing phase.

Moreover, the result of this fruitful collaboration between testing and algorithmic debugging is also beneficial for the debugger, since the unit tests can be employed for automatically detecting if some subcomputations are valid or not, thus reducing the number of questions that the user must consider. The unit tests employed for this purpose can be both those generated in a previous debugging session and those produced during the testing phase. We have applied the new general framework to the EDD, an already existing declarative debugger for the sequential subset of the programming language Erlang. The result is a debugger that generates for free and uses unit tests in the format required by the EUnit tool.

It is worth mentioning that although generated automatically, our test cases are different from those generated by usual automated test case generators, where the user needs to examine the generated test suites looking for unit tests producing erroneous results. This is known as the *oracle problem* [2]. In our case, the test case output is obtained directly from the user during the debugging, and thus this problem does not occur.

As future work, it would be interesting to use and generate not just specific tests but properties, as those defined by PropEr [8], a QuickCheck-inspired property-based testing tool for Erlang. In this way we could further prune the

debugging tree and store `trust` answers more accurately. Finally, it would also be interesting to apply the same framework to different languages, and performing an extensive experimental work to check the impact of the proposal.

Acknowledgments. We thank the anonymous reviewer of a previous work published in the journal *Science of Computer Programming for suggesting us this line of work.*

References

1. Alexander, I.F., Maiden, N.: Scenarios, Stories, Use Cases: Through the Systems Development Life-cycle. Wiley, New York (2005)
2. Barr, E., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: a survey. IEEE Trans. Softw. Eng. **41**(5), 507–525 (2015)
3. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: EDD: a declarative debugger for sequential erlang programs. In: Ábrahám, E., Havelund, K. (eds.) TACAS 2014 (ETAPS). LNCS, vol. 8413, pp. 581–586. Springer, Heidelberg (2014)
4. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: A zoom-declarative debugger for sequential Erlang programs. Sci. Comput. Program. **110**, 104–118 (2015)
5. Carlsson, R., Rémond, M.: EUnit: a lightweight unit testing framework for Erlang. In: Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, ERLANG 2006, p. 1. ACM, New York (2006)
6. Cesarini, F., Thompson, S.: Programming Erlang: A Concurrent Approach to Software Development. O'Reilly Media Inc., Beijing (2009)
7. Hebert, F.: Learn You Some Erlang for Great Good!: A Beginner's Guide. No Starch Press (2013). http://learnyousomeerlang.com
8. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Proceedings of the 2011 ACM SIGPLAN Erlang Workshop, pp. 39–50. ACM Press (2011)
9. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press, Cambridge (1983)
10. Silva, J.: A survey on algorithmic debugging strategies. Adv. Eng. Softw. **42**(11), 976–991 (2011)