A Declarative Debugger for Sequential Erlang Programs*

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

[1] Universidad Complutense de Madrid, Madrid, Spain
rafa@sip.ucm.es, {emartinm,ariesco}@fdi.ucm.es
[2] Universitat Politècnica de València, València, Spain
stamarit@dsic.upv.es

Abstract. Declarative debuggers are semi-automatic debugging tools that abstract the execution details to focus on the program semantics. Erroneous computations are represented by suitable trees, which are traversed by asking questions to the user until a bug is found. This paper applies declarative debugging to the sequential subset of the language Erlang. The debugger takes the intermediate representation generated by Erlang systems, known as Core Erlang, and an initial error detected by the user, and locates an erroneous program function responsible for the error. In order to represent the erroneous computation, a semantic calculus for sequential Core Erlang programs is proposed. The debugger uses an abbreviation of the proof trees of this calculus as debugging trees, which allows us to prove the soundness of the approach. The technique has been implemented in a debugger tool publicly available.

Keywords: Declarative debugging, Erlang, Semantics.

1 Introduction

Erlang 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 this kind of systems is a complicated process where tools such as discrepancy analyzers [21], test-case generators [27], or debuggers play an important rôle. In the case of debuggers, Erlang includes 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,

* Research supported by MICINN Spanish projects *StrongSoft* (TIN2012-39391-C04-04), *FAST-STAMP* (TIN2008-06622-C03-01) and *LEVITY* (TIN2008-06622-C03-02), Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465), and Generalitat Valenciana program *PROMETEO* (2011/052). Salvador Tamarit was partially supported by MICINN Spanish *FPI* grant (BES-2009-015019).

M. Veanes and L. Viganò (Eds.): TAP 2013, LNCS 7942, pp. 96–114, 2013.

and for this reason we think that alternative or complementary debugging tools are convenient. In this paper we take advantage of the declarative nature of the sequential subset of Erlang in order to propose a new debugger based on the general technique known as *declarative debugging* [32]. 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 results. This approach has been widely employed in the logic [23,36], functional [25,29], multi-paradigm [5,22], and object-oriented [6,20] programming languages.

Declarative debugging is a two-step scheme: it first computes a debugging tree representing a wrong computation, usually using a formal calculus that allows to prove the soundness and completeness of the approach, and then traverses this tree by asking questions to the user until the bug is identified.

The first contribution of the paper is the formalization of a semantic calculus for sequential Core Erlang programs, the intermediate language that Erlang uses to codify all the programs in a uniform representation. An interesting feature of this calculus is that it handles exceptions as usual values, which allows the user to debug expressions giving rise to exceptions in a natural way. The second and main contribution is the development of a framework for the declarative debugging of sequential Erlang programs based on an abbreviation of the proof trees obtained in this calculus as debugging trees. The process starts when the user observes that the evaluation of some expression produces an unexpected result. Therefore, the technique is restricted to terminating computations, a standard constraint in the declarative debugging approach. First, a debugging tree representing the computation is internally built. Each node in the tree corresponds to a function call occurred during the computation, and it is considered *valid* if the function call produced the expected result, and *invalid* otherwise. Then, the debugger asks questions to the user about the validity of some nodes until a buggy node—an invalid node with only valid children—is found and its associated function is pointed out as the cause of the error. The relation between the debugging trees and the proof trees in the semantic calculus allows us to prove the soundness and completeness of the technique. The formal ideas have been put into practice in the development of an Erlang system supporting the declarative debugging of sequential Erlang programs. The tool provides features such as different navigation strategies [33,34], trusting, higher-order functions, support for built-ins, and "don't know" answers.

The rest of the paper is organized as follows: Section 2 describes the related work and the similarities with our approach. Section 3 introduces Erlang and presents an example used throughout the rest of the paper. Section 4 presents the calculus we have tailored for sequential Core Erlang programs, while Section 5 shows how to use the proof trees obtained from this calculus to use declarative debugging. Section 6 outlines the main features of our tool, including a debugging session. Finally, Section 7 concludes and presents the future work.

2 Related Work

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 [18], Huch proposes an operational *small step* semantics for a subset of Erlang 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 PhD thesis [16]. This semantics is similar to [18] 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, [13] 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 [35]. Besides standard operational semantics, other approaches have been proposed to formalize Erlang semantics like the one based on congruence in [12], which works with partial evaluation of Erlang programs.

Regarding Core Erlang [8,9],[1] there is no formalized semantics in the literature. However, the language specification [9] contains a detailed but informal explanation of the expected behavior for the evaluation of expressions. The semantics proposed for Core Erlang in this paper—inspired in the semantics previously presented for Erlang, mainly [18,16]—formalizes the behavior explained in [9] and covers all the Core Erlang syntax except concurrency-related expressions (message sending and reception) and *bit string* notation.

Tools for testing and debugging programs are very popular in the Erlang community since long ago. The OTP/Erlang system comes with a classical trace-debugger with both graphical and command line interfaces. This official debugger supports the whole Erlang language—including concurrency—, allowing programmers to establish conditional breakpoints in their code, watch the stack trace of function calls, and inspect variables and other processes, among other features. Another tool included in the OTP/Erlang system is the *DIscrepancy AnaLYZer for ERlang programs* (Dialyzer) [21], a completely automatic tool that performs static analysis to identify software discrepancies and bugs such as definite type errors [31], race conditions [11], unreachable code, redundant tests, unsatisfiable conditions, and more. Model checking tools have also received much attention in the Erlang community [18,3], being *McErlang* [4] one of the most powerful nowadays due to its support for a very substantial part of the Erlang language. Regarding testing tools, the most important ones

[1] The official Core Erlang [8,9] should not be confused with the subsets of Erlang that the previous papers covered, although they usually refer them as *some* core Erlang.

are *EUnit* [10] and *Quviq QuickCheck* [19]. The EUnit tool is included in the OTP/Erlang system, and allows users to write their own unit tests to check that functions return the expected values. On the other hand, Quiviq QuickCheck is a commercial software that automatically generates and checks thousands of unit tests from properties stated by programmers.

Declarative debugging is a well-known debugging technique. In [30] a comparison between different debuggers is presented. From this comparison 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. An interesting contribution of our debugger 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 [22], for Haskell in its declarative debugger Buddha [28], and for Java programs [20]. 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 nodes in the debugging tree; Buddha uses a program transformation to build the debugging tree 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 [15,14], have been proposed for functional languages. In this paper we present, for the best of our knowledge, the first tool that uses a calculus to perform declarative debugging, allowing us to reason about exceptions as standard values.

Finally, other non-conventional approaches to debugging have been studied in the literature, like abstract diagnosis [1] or symbolic execution [17], but these techniques are not closely related to declarative debugging, so we do not provide a detailed comparison.

3 Erlang

Erlang [2] is a concurrent language with a sequential subset that is a functional language with dynamic typing and strict evaluation. It allows to model programs where different processes communicate through asynchronous messages and gives support to fault-tolerant and soft real-time applications, and also to non-stop applications thanks to the so-called *hot swapping*.

Example 1. Figure 1 presents an example of an Erlang program (for the time being ignore the boxes). The program exports function `mergesort/2` that, given a list `L` and a comparison function `Comp`, orders the elements of `L` according to `Comp`. Function `comp/2` is also exported in order to provide an atom comparison function. The rest of functions are used internally. Function `merge/3` merges two lists according to a given comparison function. Finally, `take/2` and `last/2` extract the `N` first (respectively last) elements of a list. Observe that the code contains calls to built-ins and standard library functions, e.g. `is_atom/1` in line 16 or `reverse/1` of module `lists` in line 21. Now the user can write the next expression to sort the list `[b,a]` using the comparison function `comp/2`:

```
 1 mergesort([], _Comp) -> [];
 2 mergesort([X], _Comp) -> [X];
 3 mergesort(L, Comp) ->
 4    Half = length(L) div 2,
 5    LOrd1 = mergesort(take(Half, L), Comp),
 6    LOrd2 = mergesort(last(length(L) - Half, L), Comp),
 7    merge(LOrd1, LOrd2, Comp).

 8 merge([], [], _Comp) -> [];
 9 merge([], S2, _Comp) -> S2;
10 merge(S1, [], _Comp) -> S1;
11 merge([H1 | T1], [H2 | T2], Comp)  ->
12    case Comp(H1,H2) of
13       false -> [ H1 | merge([ H2 | T1], T2, Comp)];
14       true ->  [H1 | merge(T1, [H2 | T2], Comp)]
15    end.

16 comp(X,Y) when is_atom(X) and is_atom(Y) -> X < Y.

17 take(0, _List) -> [];
18 take(1, [H|_T]) -> [H];
19 take(_, []) -> [];
20 take(N, [H|T]) -> [ N | take(N-1, T)].

21 last(N, List) -> lists:reverse(take(N, lists:reverse(List))).
```

Fig. 1. Erlang code implementing mergesort algorithm

`merge:mergesort([b,a], fun merge:comp/2)`. The system evaluates this expression and displays `[b,a]`. This is an unexpected result, an initial symptom indicating that there is some erroneous function in the program. In the next sections we show how the debugger helps in the task of finding the bug.

The intermediate language Core Erlang [8,9] can be considered as a simplified version of Erlang, where the syntactic constructs have been reduced by removing syntactic sugar. It is used by the compiler to create the final bytecode and it is very useful in our context, because it simplifies the analysis required by the tool. Figure 2 presents its syntax after removing the parts corresponding to concurrent operations, i.e. `receive`, and also the bit syntax support. The most significant element in the syntax is the expression (*expr*). Besides variables, function names, lambda abstractions, lists, and tuples, expressions can be:

- `let`: its value is the one resulting from evaluating $exprs_2$ where *vars* are bound to the value of $exprs_1$.
- `letrec`: similar to the previous expression but a sequence of function declarations (*fname* = *fun*) is defined.
- `apply`: applies *exprs* (defined in the current module) to a number of arguments.
- `call`: similar to the previous expression but the function applied is the one defined by $exprs_{n+2}$ in the module defined by $exprs_{n+1}$. Both expressions should be evaluated to an atom.
- `primop`: application of built-in functions mainly used to report errors.

```
fname ::= Atom / Integer
lit   ::= Atom | Integer | Float | Char | String | [ ]
fun   ::= fun(var_1 , ... , var_n) -> exprs
clause ::= pats when exprs_1 -> exprs_2
pat   ::= var | lit | [ pats | pats ] | { pats_1 , ... , pats_n } | var = pats
pats  ::= pat | < pat , ... , pat >
exprs ::= expr | < expr , ... , expr >
expr  ::= var | fname | fun | [ exprs | exprs ] | { exprs_1 , ... , exprs_n }
        | let vars = exprs_1 in exprs_2
        | letrec fname_1 = fun_1 ... fname_n = fun_n in exprs
        | apply exprs ( exprs_1 , ... , exprs_n )
        | call exprs_{n+1}:exprs_{n+2} ( exprs_1 , ... , exprs_n )
        | primop Atom ( exprs_1 , ... , exprs_n )
        | try exprs_1 of <var_1 , ... , var_n> -> exprs_2 catch <var'_1 , ... , var'_m> -> exprs_3
        | case exprs of clause_1 ... clause_n end | do exprs_1 exprs_2 | catch exprs
ξ     ::= Exception(val_m)
val   ::= lit | fname | fun | [ vals | vals ] | {vals_1 , ... , vals_n} | ξ
vals  ::= val | < val , ... , val >
vars  ::= var | < var , ... , var >
```

Fig. 2. Core Erlang's Syntax

- `try-catch`: the expression $exprs_1$ is evaluated. If the evaluation does not report any error, then $exprs_2$ is evaluated. Otherwise, the evaluated expression is $exprs_3$. In both cases the appropriate variables are bound to the value of $exprs_1$.
- `case`: a pattern-matching expression. Its value corresponds to the one in the body of the first clause whose pattern matches the value of *exprs* and whose guard evaluates to `true`. There is always at least one clause fulfilling these conditions, as we explain below.

It is important to know some basis of how the translation from Erlang to Core Erlang is done. One of the most relevant is that the body of a Core Erlang function is always a `case`-expression representing the different clauses of the original Erlang function. `case`-expressions as shown below always contain an extra clause whose pattern matches with every value of the `case` argument and whose body evaluates to an error (reported by a `primop`). This clause is introduced by the compiler and placed last. Consider, for instance, the function `take/2` in Figure 1. The translation to Core Erlang produces the following code:

```
'take'/2 = fun (_cor1,_cor0) ->
       case <_cor1,_cor0> of
           <0,_cor6> when 'true' -> []
           <1,[H|_cor7]> when 'true' -> [H|[]]
           <_cor8,[]> when 'true' -> []
           <N,[H|T]> when 'true' -> ...
           <_cor5,_cor4> when 'true' ->
                   primop 'match_fail' ('function_clause',_cor5,_cor4)
       end
```

Note that the compiler has introduced new variables in the translation with the form `_cor` followed by an integer. Another important point is the introduction

of `let`-expressions. These expressions are not present in Erlang, and in Core are introduced for different reasons. One usage of `let`-expressions in the translation is to ensure that function applications always receive simple expressions (values or variables) as arguments. For instance, the call to `take/2` at line 20 in Figure 1 (`take(N-1, T)`) is translated to Core as:

```
let <_cor2> = call 'erlang':'-'(N, 1) in  apply 'take'/2(_cor2, T)
```

`let`-expressions are also used to translate sequences of Erlang expressions. For instance, the sequence of expressions going from line 4 to line 7 in Figure 1 is translated to the following Core Erlang code:

```
let <_cor2> = call 'erlang':'length'(L)
in let <Half> = call 'erlang':'div'(_cor2, 2)
   in let <L1> = apply 'take'/2(Half, L)
      in let <_cor5> =  call 'erlang':'length'(L)
         in let <_cor6> = call 'erlang':'-'(_cor5, Half)
            in let <L2> = apply 'last'/2(_cor6, L)
               in let <LOrd1> = apply 'mergesort'/2(L1, Comp)
                  in let <LOrd2> = apply 'mergesort'/2(L2, Comp)
                     in  apply 'merge'/3(LOrd1, LOrd2, Comp)
```

It can be observed that the translation from plain Erlang to Core enforces the function applications to receive values or variables as arguments. In principle, we could simplify our language and semantics taking into account these particularities, but our tool allows the more general grammar of Core Erlang for two reasons. First, Core Erlang could be used as intermediate language produced by other languages. Second, the Core Erlang structure could be modified by optimization tools, and thus a particular structure cannot be assumed.

4 A Calculus for Sequential Erlang

This section presents the main rules of our calculus for Core Erlang Sequential Programs (*CESC* in the following). The complete set of rules is presented in [7]. The calculus uses evaluations of the form $\langle exprs, \theta \rangle \to vals$, where *exprs* is the expression being evaluated, θ is a substitution, and *vals* is the value obtained for the expression. Moreover, we use the notation $\langle exprs, \theta \rangle \to^i vals$ in some cases to indicate that the ith clause of a function was used to obtain a value. We assume that all the variables for *exprs* are in the domain of θ, and the existence of a global environment ρ which is initially empty and is extended by adding the functions defined by the `letrec` operator. *References to functions*, denoted as r_f, are unique identifiers pointing to the function in the source code (this can be described in general with the tuple *(mod, line, column)* with *line* and *column* the starting position of the function in *mod*). We extend the idea of reference to reserved words, denoted by r, to obtain the module where they are defined, which is denoted by $r.mod$. These references are used to unify the handling of function calls with the inference rule (BFUN), which is explained below. The notation $CESC \models_{(P,T)} \mathcal{E}$, where $\mathcal{E}$ is an evaluation, is employed to indicate that $\mathcal{E}$ can be proven w.r.t. the program P with the proof tree T in *CESC*, while

$CESC \not\vdash_P \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in $CESC$ with respect to the program P.

The basic rule in our calculus is (VAL), which states that values are evaluated to themselves:

$$(\mathsf{VAL})\ \frac{}{\langle vals, \theta\rangle \rightarrow vals}$$

The (CASE) rule is in charge of evaluating `case`-expressions. It first evaluates the expression used to select the branch. Then, it checks that the values thus obtained match the pattern on the ith branch and verify the `when` guard, while the side condition indicates that this is the first branch where this happens. The evaluation continues by applying the substitution to the body of the ith branch.

$$(\mathsf{CASE})\ \frac{\langle exprs'', \theta\rangle \rightarrow vals'' \quad \langle exprs'_i\theta', \theta'\rangle \rightarrow \texttt{'true'} \quad \langle exprs_i\theta', \theta'\rangle \rightarrow vals}{\langle \texttt{case}\ exprs''\ \texttt{of}\ \overline{pats_n\ \texttt{when}\ exprs'_n\ \texttt{->}\ exprs_n}\ \texttt{end}, \theta\rangle \rightarrow^i vals}$$

where $\theta' \equiv \theta \uplus matchs(pats_i, vals'')$; $\forall j < i.\nexists\theta_j.matchs(pats_j, vals'') = \theta_j \wedge \langle exprs'_j\theta_j, \theta_j\rangle \rightarrow \texttt{'true'}$; and $matchs$ a function that computes the substitution binding the variables to the corresponding values using syntactic matching as follows:

$matchs(\texttt{<}\ pat_1, \ldots, pat_n\ \texttt{>}, \texttt{<}\ val_1, \ldots, val_n\ \texttt{>}) = \theta_1 \uplus \ldots \uplus \theta_n$,
with $\theta_i = match(pat_i, val_i)$ where $match$ is an auxiliary function defined as:

$match(var, val) = [var \mapsto val]$
$match(lit_1, lit_2) = id$, if $lit_1 \equiv lit_2$
$match([pat_1|pat_2], [val_1|val_2]) = \theta_1 \uplus \theta_2$, where $\theta_i = match(pat_i, val_i)$
$match(\{pat_1, \ldots, pat_n\}, \{val_1, \ldots, val_n\}) = \theta_1 \uplus \ldots \uplus \theta_n$,
where $\theta_i = match(pat_i, val_i)$
$match(var = pat, val) = \theta[var \mapsto val]$, where $\theta = match(pat, val)$

Similarly, the (LET) rule evaluates $exprs_1$ and binds it to the variables. The computation continues by applying the substitution to the body:

$$(\mathsf{LET})\ \frac{\langle exprs_1, \theta\rangle \rightarrow vals_1 \quad \langle exprs_2\theta', \theta'\rangle \rightarrow vals}{\langle \texttt{let}\ vars\ \texttt{=}\ exprs_1\ \texttt{in}\ exprs_2, \theta\rangle \rightarrow vals}$$

where $\theta' \equiv \theta \uplus matchs(vars, vals_1)$.

The (BFUN) rule evaluates a reference to a function, given a substitution binding all its arguments. This is accomplished by applying the substitution to the body of the function (with notation $exprs\theta$) and then evaluating it. This rule takes advantage of the fact that, as explained in Section 3, all Erlang functions are translated to Core Erlang as a `case`-expression distinguishing the different clauses. The particular branch of the `case`-expression employed during the computation t (i.e. the i labeling the evaluation) corresponds to the clause used to evaluate the function:

$$(\mathsf{BFUN})\ \frac{\langle \texttt{case}\ exprs\theta\ \texttt{of}\ clause_1\theta \ldots clause_m\theta\ \texttt{end}, \theta\rangle \rightarrow^i vals}{\langle r_f, \theta\rangle \rightarrow^i vals}$$

where $1 \leq i \leq m$ and r_f references to a function f defined as $f\texttt{/}n\ \texttt{= fun (}\ var_1\ \texttt{, ... ,}\ var_n\ \texttt{) ->}\ \texttt{case}\ exprs\ \texttt{of}\ clause_1 \ldots clause_m\ \texttt{end}$.

The rule (CALL) evaluates a function defined in another module:

$$(\mathsf{CALL})\ \frac{\begin{array}{c}\langle exprs_{n+1},\theta\rangle \to Atom_1 \quad \langle exprs_{n+2},\theta\rangle \to Atom_2 \\ \langle exprs_1,\theta\rangle \to val_1 \quad \ldots \quad \langle exprs_n,\theta\rangle \to val_n \\ \langle r_f,\theta'\rangle \to^i vals\end{array}}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1,\ldots,exprs_n),\theta\rangle \to vals}$$

where $Atom_2/n$ is a function defined in the $Atom_1$ module as $Atom_2/n$ `= fun (` var_1 `, ... ,` var_n `) -> case` $exprs$ `of` $clause_1 \ldots clause_m$ `end`; r_f its reference; $1 \leq i \leq m$; and $\theta' \equiv \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$.

Analogously, the (CALL_EVAL) rule is in charge of evaluating built-in functions:

$$(\mathsf{CALL\_EVAL})\ \frac{\begin{array}{c}\langle exprs_{n+1},\theta\rangle \to \texttt{'erlang'} \quad \langle exprs_{n+2},\theta\rangle \to Atom_2 \\ \langle exprs_1,\theta\rangle \to val_1 \quad \ldots \quad \langle exprs_n,\theta\rangle \to val_n \\ eval(Atom_2, val_1, \ldots, val_n) = vals\end{array}}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1,\ldots,exprs_n),\theta\rangle \to vals}$$

where $Atom_2/n$ is a built-in function included in the `erlang` module.

Finally, the rule (APPLY_3)[2] indicates that first we need to obtain the name of the function, which must be defined in the current module (extracted from the reference to the reserved word `apply`) and then compute the arguments of the function. Finally the function, described by its reference, is evaluated using the substitution obtained by binding the variables in the function definition to the values for the arguments:

$$(\mathsf{APPLY}_3)\ \frac{\begin{array}{c}\langle exprs,\theta\rangle \to Atom/n \\ \langle exprs_1,\theta\rangle \to val_1 \quad \ldots \quad \langle exprs_n,\theta\rangle \to val_n \\ \langle r_f,\theta'\rangle \to^i vals\end{array}}{\langle \texttt{apply}^r\ exprs(exprs_1,\ldots,exprs_n),\theta\rangle \to vals}$$

where $Atom/n \notin dom(\rho)$ (i.e., it was not defined in a `letrec` expression); r is the reference to `apply`; $Atom/n$ is a function defined in the current module $r.mod$ as $Atom/n$ `= fun (` var_1 `, ... ,` var_n `) -> case` $exprs$ `of` $clause_1 \ldots clause_m$ `end`; r_f its reference; $1 \leq i \leq m$; and $\theta' \equiv [\overline{var_n \mapsto val_n}]$.

We can use this calculus to build the proof tree for any evaluation. For example, Figure 3(top) shows the proof tree for $\langle$`mergesort/2([b,a], comp/2)`$, id\rangle \to$ `[b,a]`, where `ms` stands for `mergesort/2` and `c` for the comparison function `comp/2`. Note that the root of the tree contains the identity substitution and that all the arguments are evaluated (in this case by using the rule (VAL)) before really reducing the function in $\bigtriangledown_1$.

The proof tree $\bigtriangledown_1$ is partially shown in Figure 3(middle), where $r_{\texttt{ms}}$ is the reference to `mergesort` and θ_1 is $\{$`_cor1` $\mapsto$ `[b,a]`; `_cor0` $\mapsto$ `c`$\}$. The root of this tree presents some of the features described in Section 3: the function has been translated as a `case`-expression and Core Erlang variables (`_cor1` and `_cor0`) have been introduced to check whether they match any clause. This matching is

[2] The rule (APPLY_1) executes a function previously defined in a `letrec`, while (APPLY_2) executes functions defined as lambda abstractions. See [7] for details.

accomplished by evaluating the values to themselves in the premises of (CASE), and then checking that the condition holds trivially by evaluating `'true'` to itself. The computation continues by evaluating the body of this branch, which is represented in $\bigtriangledown_2$.

The tree in Figure 3(bottom) shows $\bigtriangledown_2$, where θ_2 extends θ_1 with $\{$`Comp` $\mapsto$ `c;L` $\mapsto$ `[b,a]`$\}$, `l` stands for the built-in function `length/2`, and `m` for `merge/3`. It first computes the length of the list, that is kept in a new variable `_cor2`, and then used to split the list, as explained in the translation to Core Erlang at the end of Section 3. Note that `length/2` is evaluated with (CALL_EVAL), an inference rule for applying built-in functions. Finally, the subtree $\bigtriangledown_3$ stands for several inferences using the (LET) rule that bind all the variables until we reach the function `merge/3([b], [a], comp)`, which is executed using the (APPLY_3) as shown for the proof tree on the top of the figure. Note that this node indicates that the 4th clause of the function `merge/3` has been used. The rest of the $\bigtriangledown$ and the substitutions θ, θ', and θ'' are not relevant and are not described.

5 Debugging with Abbreviated Proof Trees

Our debugger detects functions incorrectly defined by asking questions to the user about the *intended interpretation* of the program, which is represented as a set of the form $\mathcal{I} = \{\ldots,\ m.f(val_1,\ \ldots, val_n) \rightarrow vals,\ \ldots\}$ where m is a module name (omitted for simplicity in the rest of the section), f is a user-defined function in module m, and $val_1, \ldots, val_n, vals$ are values such that the user expects $f(val_1, \ldots, val_n) \rightarrow vals$ to hold. The validity of the nodes in a proof tree is obtained defining an *intended interpretation calculus*, *ICESC*, which contains the same inference rules as *CESC*, except by the (BFUN) rule, which is replaced by the following:

$$(\mathsf{BFUN}_{\mathcal{I}})\ \frac{}{\langle r_f, \theta\rangle \rightarrow vals}$$

where r_f references a program function f`/`n `= fun(` var_1 `, ..., ` var_n `) ->` B, (B represents the body of the function), and $f(var_1\theta, \ldots, var_n\theta)$ `->` $vals) \in \mathcal{I}$.

Analogously to the case of *CESC*, the notation $ICESC \models_{(P,\mathcal{I},T)} \mathcal{E}$ indicates that the evaluation $\mathcal{E}$ can be proven w.r.t. the program P and the intended interpretation $\mathcal{I}$ with proof tree T in *ICESC*, while $ICESC \not\models_{(P,\mathcal{I})} \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in *ICESC*. The tree T, the program P, and the intended interpretation $\mathcal{I}$ are omitted when they are not needed. The rôle of the two calculi is further clarified by the next two assumptions:

1. If an evaluation $e\theta \rightarrow vals$ is computed by some Erlang system with respect to P then $CESC \models_P \langle |e|, \theta\rangle \rightarrow |vals|$, with $|\cdot|$ the transformation that converts an Erlang expression into a Core expression.
2. If a reduction $e\theta \rightarrow vals$ computed by some Erlang system is considered unexpected by the user then $ICESC \not\models_{P,\mathcal{I}} \langle |e|, \theta\rangle \rightarrow |vals|$.

Thus, *CESC* represents the actual computations, while *ICESC* represents the 'ideal' computations expected by the user. Next we define some key concepts:

Definition 1. *Let P be a program with intended interpretation $\mathcal{I}$. Let $\mathcal{E} \equiv \langle e, \theta \rangle \to vals$ be an evaluation such that CESC $\models_P \mathcal{E}$. Then we say that:*

1. *$\mathcal{E}$ is* valid *when ICESC $\models_{(P,\mathcal{I})} \mathcal{E}$, and* invalid *when ICESC $\not\models_{(P,\mathcal{I})} \mathcal{E}$.*
2. *A node is buggy if it is invalid with valid premises.*
3. *Let r_f be a reference to a function f/n* `= fun(` var_1`, ..., ` var_n `) ->` B*, and θ a substitution. Then $\langle r_f, \theta \rangle$ is a* wrong function instance *iff there exists a value v such that ICESC $\not\models_{(P,\mathcal{I})} \langle r_f, \theta \rangle \to v$ and ICESC $\models_{(P,\mathcal{I})} \langle B\theta, \theta \rangle \to v$.*
4. *$P_{\langle e,\theta \rangle}$ denotes the (Core Erlang) program $P \cup \{$* `main/0 = fun() -> case <> of <> when 'true' ->` $|e\theta|$ `end` *$\}$, with $|\cdot|$ the operator converting Erlang code into Core Erlang code, and* `main` *a new identifier in P.*

The first two items do not need further explanations. In order to understand the third point observe that we cannot say that a function is wrong simply because it computes an unexpected result, since this can be due to the presence of wrong functions in its body. Instead, a wrong function instance corresponds to a function that produces an erroneous result assuming that all the functions in its body are correct. Finally, the purpose of definition of $P_{\langle e,\theta \rangle}$ is to introduce the expression producing the initial symptom as part of the program. The two following auxiliary results are a straightforward consequence of the coincidence of *CESC* and *ICESC* in all the rules excepting (BFUN):

Lemma 1. *Let T be a CESC-proof tree and let N be a node in T conclusion of an inference rule different from* (BFUN) *with all the premises valid. Then N is valid.*

Lemma 2. *Let T be a CESC-proof tree which does not use the* (BFUN) *inference. Then all the nodes in T are valid.*

The next theorem uses the two lemmas to establish that *CESC* proof trees are suitable for debugging.

Theorem 1. *Let P be a Core Erlang program, $\mathcal{I}$ its intended interpretation, and $e\theta \to vals$ an unexpected evaluation computed by an Erlang system.*
Then:

1. *There is a proof tree T such that CESC $\models_{(P_{\langle e,\theta \rangle},T)} \langle r_{\texttt{main}}, id \rangle \to |vals|$.*
2. *T contains at least one buggy node N verifying:*
   (a) *N is the conclusion of a* (BFUN) *inference rule of the form $\langle r_f, \theta \rangle \to vals$ with $f \neq$* `main`*.*
   (b) *$\langle r_f, \theta \rangle$ is a wrong function instance.*

$$\text{(APPLY}_3\text{)}\ \frac{\text{(VAL)}\ \overline{\langle \texttt{ms}, id\rangle \to \texttt{ms}} \quad \text{(VAL)}\ \overline{\langle \texttt{[b,a]}, id\rangle \to \texttt{[b,a]}} \quad \text{(VAL)}\ \overline{\langle \texttt{c}, id\rangle \to \texttt{c}} \quad \bigtriangledown_1}{\langle \texttt{apply ms([b,a],c)}, id\rangle \to \texttt{[b,a]}}$$

where Proof tree $\bigtriangledown_1$ is defined as:

$$\text{(BFUN)}\ \frac{\text{(CASE)}\ \dfrac{\text{(VAL)}\ \overline{\texttt{<[b,a],c>} \to \texttt{<[b,a],c>}} \quad \text{(VAL)}\ \overline{\texttt{'true'} \to^3 \texttt{'true'}} \quad \bigtriangledown_2}{\langle \texttt{case <[b,a],c> of ... <L,Comp> when 'true' -> } exp, \theta_1\rangle \to^3 \texttt{[b,a]}}}{\langle r_{\texttt{ms}}, \theta_1\rangle \to^3 \texttt{[b,a]}}$$

and Proof tree $\bigtriangledown_2$ is defined as:

$$\text{(LET)}\ \frac{\text{(CALL\_EVAL)}\ \dfrac{\bigtriangledown \quad \bigtriangledown \quad \bigtriangledown}{\langle \texttt{call 'erlang':'l'([b,a])}, \theta_2\rangle \to 2} \quad \text{(LET)}\ \dfrac{\bigtriangledown \quad \text{(APPLY}_3\text{)}\ \dfrac{\bigtriangledown \quad \bigtriangledown \quad \text{(BFUN)}\ \dfrac{\bigtriangledown}{\langle r_{\texttt{m}}, \theta''\rangle \to^4 \texttt{[b,a]}}}{\langle \texttt{apply 'm'([b],[a],c)}, \theta'\rangle \to \texttt{[b,a]}}}{\bigtriangledown_3}}{\langle \texttt{let \_cor2 = call 'erlang':'l'([b,a]) in ... in apply 'm'(LOrd1,LOrd2,c)}, \theta_2\rangle \to \texttt{[b,a]}}$$

Fig. 3. Proof tree for `mergesort([b,a],c)` → `[b,a]`

Proof Sketch

1. By assumption 1, there is a proof tree T' such that $CESC \models_{P,T'} \langle |e|, \theta\rangle \to |vals|$. From T' it is easy to construct a proof tree T'' for $\langle |e|\theta, id\rangle \to |vals|$. Then, the tree T such that $CESC \models_{(P_{\langle e,\theta\rangle},T)} \langle r_{\texttt{main}}, id\rangle \to |vals|$ starting by an application of the (BFUN) inference rule for `main/0` at the root, followed by a (CASE) inference with the body of `main/0` (see Definition 1, item 4) and then followed by T''.
2. Every proof tree with an invalid root contains a buggy node [24].
   (a) $\langle e, \theta\rangle \to vals$ is unexpected, by assumption 2, $ICESC \not\models_{P,\mathcal{I}} \langle |e|, \theta\rangle \to |vals|$, and it is easy to check that then $ICESC \not\models \langle |e|\theta, id\rangle \to |vals|$. Hence, the root of T'' is invalid and T'' contains a buggy node N. Since T'' is a subtree of T which does not contain `main/0` then the node is not related to this function. Moreover, since all the inference rules except for (BFUN) are both in *CESC* and in *ICESC*, and the buggy node has all its premises valid, by Lemma 1 N is the conclusion of a (BFUN) inference. Thus, N is of the form $\langle r_f, \theta\rangle \to vals$, $f \neq \texttt{main}$.
   (b) By Definition 1.2 the buggy node N in T'' (and therefore in T) verifies:
      i. N is invalid, and thus by Definition 1.1: $ICESC \not\models_{(P,\mathcal{I})} \langle r_f, \theta\rangle \to vals$.
      ii. The premises of N in T are valid. N is the conclusion of a (BFUN) rule, and then the only premise corresponds to the proof in *CESC* of the function body B. Since it is valid, by Definition 1.1 the proof in *CESC* implies the proof in *ICESC*.

   By 2(b)i and 2(b)ii and according to Definition 1.3, $\langle r_f, \theta\rangle$ is a wrong function instance. □

Although the previous result shows that the proof trees obtained by using *CESC* might be used as debugging trees, they contain many questions difficult to answer for the user, including the results of nested applications, or even new statements introduced in the Core representation of the program. In order to overcome these difficulties we abbreviate the proof tree keeping only the relevant information for detecting buggy nodes.

Definition 2. *Let P be a program and $\langle e, \theta\rangle \to vals$ an evaluation. A tree T is an Abbreviated Proof Tree (APT in short) for the evaluation with respect to P, iff there is a proof tree T' such that $CESC \models_{(P_{\langle e,\theta\rangle},T')} \langle r_{\texttt{main}}, id\rangle \to vals$, and $T = APT(T')$ with the transformation APT defined as follows:*

$$APT\left(\text{(BFUN)}\ \frac{T}{\mathcal{E}}\right) = \frac{APT(T)}{\mathcal{E}}$$

$$APT\left(\text{(R)}\ \frac{T_1 \ldots T_n}{\mathcal{E}}\right) = APT(T_1)\ldots APT(T_n),\ \textit{with}\ \text{(R)} \neq \text{(BFUN)}$$

The transformation keeps the conclusion of (BFUN) inferences, removing the rest of the nodes. Observe that the result is a single tree because the root of T' always corresponds to the application of (BFUN) to an evaluation over `main/0`.

$$\dfrac{\dfrac{\bigtriangledown_1 \quad \texttt{t(1,[b,a])} \to \texttt{[b]} \quad \texttt{ms([a],c)} \to \texttt{[a]} \quad \bigtriangledown_2 \quad \texttt{ms([b],c)} \to \texttt{[b]}}{(\circ)\ \texttt{ms([b,a],c)} \to \texttt{[b,a]}}}{\texttt{main()} \to \texttt{[b,a]}}$$

where:

$$\bigtriangledown_1 = \dfrac{\texttt{m([a],[],c)} \to \texttt{[a]} \quad \texttt{c(b,a)} \to \texttt{false}}{(\circ, \bullet)\ \texttt{m([b],[a],c)} \to \texttt{[b,a]}} \qquad \bigtriangledown_2 = \dfrac{\texttt{t(1,[a,b)]} \to \texttt{[a]}}{\texttt{l(1,[b,a])} \to \texttt{[a]}}$$

Fig. 4. Abbreviated Proof tree for `mergesort([b,a],comp)` → `[b,a]`

The structure of the APTs is similar to the *evaluation dependence tree* employed in functional languages [26]. The main difference, apart from the particularities of each language, is that in our case the APTs are obtained from a semantic calculus, which allows us to prove their adequacy as debugging trees:

Theorem 2. *Let P be a Core Erlang program, $\mathcal{I}$ its intended interpretation, and $e\theta \to vals$ an unexpected evaluation. Let T' be such that CESC $\models_{(P_{\langle e,\theta\rangle}, T')}$ $\langle r_{\texttt{main}}, id\rangle \to vals$, and $T = APT(T')$. Then T contains at least one buggy node N including a wrong instance of a user function different from `main/0`.*

Proof Sketch
As observed in the sketch of Theorem 1, there is a tree T'', subtree of T', which contains a buggy node N not associated to `main`. By construction (Definition 2) N is in T, and thus N is the invalid root of a subtree of T which does not contain `main/0`. Hence T contains a buggy node N not associated to `main/0`.

Let $N_1, \ldots, N_k$ be the direct descendants of N in T' which correspond to conclusions of (BFUN) inferences. By Definition 2 these nodes are the premises of N in T, and since N is buggy then $N_1, \ldots, N_k$ must be valid. Applying Lemma 1 it is easy to check that all the intermediate nodes between N and $N_1, \ldots, N_k$ are valid. This proves that all the premises of N in T' which are ancestors of (BFUN) inferences are valid. The rest of the premises of N in T' are roots of proof trees without (BFUN) inferences, and by Lemma 2 are valid as well. Thus N is buggy in T', and by Theorem 1 it contains a wrong function instance. □

Therefore, during the debugging process the user only needs to answer questions about the validity of nodes in the APT, which corresponds exactly to the evaluations in the intended interpretation. Figure 4 shows the APT for our running example. The calls to `reverse/1` have been removed because predefined functions are automatically trusted. The evaluations $\langle r_f, \theta\rangle \to vals$ are shown with the more user-friendly form $f(var_1\theta, \ldots, var_n\theta) \to vals$, which is also the form employed by the debugger. For the sake of space we use `c` for `comp/2`, `t` for `take/2`, `l` for `last/2`, `ms` for `mergesort/2`, and `m` for `merge/3`. The invalid nodes are preceded by ∘ and the only buggy node is preceded also by •.

6 System Description

The technique described in the previous sections has been implemented in Erlang. The tool is called `edd` and it has approximately 1000 lines of code. It is publicly available at `https://github.com/tamarit/edd`.

When a user detects an unexpected result, `edd` asks questions of the form `call = value`, where the user has the option to answer either *yes* (`y`) or *no* (`n`). In addition to these two responses, the user can answer *don't know* (`d`), when the answer is unknown; *inadmissible* (`i`), when the question does not apply; or *trusted* (`t`), when the user knows that the function is correct and further questions are not necessary. Before starting the debugger, the user can also define a number of functions to be trusted. The user can also *undo* (`u`), which reverts to the previous question, and *abort* (`a`), which closes the debugging session. Moreover, the tool includes a memoization feature that stores the answers *yes*, *no*, *trusted*, and *inadmissible*, preventing the system from asking the same question twice. Finally, it is worth noting that the answer *don't know* is used to ease the interaction with the debugger but it may introduce incompleteness, so we did not consider it in our proofs; if the debugger reaches a deadlock due to these answers it will inform the user about this fact.

The system can internally utilize two different navigation strategies [33,34], *Divide & Query* and *Heaviest First*, in order to choose the next node and therefore the next question presented to the user.

Both strategies help the system to minimize the number of questions and they can be switched during the session with option `s`. The rest of the section shows how the tool is used through a debugging session. Assume the user has decided to use `edd` to debug the error shown in Section 3. The debugging session with the default *Divide & Query* navigation strategy is:

```
> edd:dd("merge:mergesort([b,a], fun merge:comp/2)").
Please, insert a list of trusted functions [m1:f1/a1, m2:f2/a2 ...]:
merge:merge([b], [a], fun merge:comp/2) = [b, a]?: n
```

The strategy selects the subtree marked with (○, •) in Figure 4, because the whole tree (without the dummy node for `main` and the node for the initial call, that we know is wrong because the user is using the debugger) has 8 nodes and this subtree, with 3 nodes, is the closest one to half the size of the whole tree. The question associated to the node, shown above, asks the user whether she expects to obtain the list `[b,a]` when evaluating `merge([b], [a], fun merge:comp/2)`. She answers `n`o because this is obviously an unexpected result. Thus, the subtree rooted by this node becomes the current debugging tree.

```
merge:comp(b, a) = false?: t
```

The next question corresponds to one of the children of the previous node. In this case the tool asks to the user whether the result for `comp(b, a)` is `false`. The answer is `t`, which removes all the nodes related to the function `comp` (in this case the answer only affects this node, but trusting a function may remove several nodes in general). The next question is:

```
merge:merge([a], [], fun merge3:comp/2) = [a]?: y
```

In this case the function `merge` produces the expected result, so the user says `y`es. Since the first answer marked a node as invalid node and the next two answers

indicated that all its children are correct, the tool infers that the node (○, ●) in Figure 4 is buggy:[3]

```
Call to a function that contains an error:
merge:merge([b], [a], fun merge:comp/2) = [b, a]
Please, revise the fourth clause:
merge([H1 | T1], [H2 | T2], Comp) ->  case Comp(H1, H2) of
      false -> [H1 | merge([H2 | T1], T2, Comp)];
      true -> [H1 | merge(T1, [H2 | T2], Comp)]
    end.
```

This information is enough to find the error, marked by boxes in the definition of `merge` in Section 3. It is corrected by using `[H2 | merge([H1 | T1], T2, Comp)]`.

Although this change fixes the error for the list `[b,a]`, when `mergesort` is executed with a larger list, like `[o,h,i,o]`, a new problem arises:

```
> merge:mergesort([o,h,i,o], fun merge:comp/2).
** exception error: no function clause matching
merge:comp(2,h) (merge.erl, line 16)
     in function  merge:merge/3 (merge.erl, line 12)
     in call from merge:mergesort/2 (merge.erl, line 7)
```

Note that the information given by the system is not useful to find the error, since `merge` and `mergesort` seem correct. Moreover, the user is sure that the error is not in `comp/2`, even though it receives a erroneous argument, and hence she has marked it as trusted. The user decides to follow a *Top Down* strategy in this session. The debugging session runs as follows:

```
> edd:dd("merge:mergesort([o,h,i,o], fun merge:comp/2)",top_down).
Please, insert a list of trusted functions [m1:f1/a1, m2:f2/a2 ...]:
merge:comp/2
merge:mergesort([2, h], fun merge:comp/2) = {error, match_fail}?: i
merge:last(2, [o, h, i, o]) = [i, 2]?: n
merge:take(2, [o, i, h, o]) = [2, i]?: n
merge:take(1, [i, h, o]) = [i]?: y
Call to a function that contains an error:
merge:take(2, [o, i, h, o]) = [2, i]
Please, revise the fourth clause:
take(N, [H | T]) -> [N | take(N - 1, T)].
```

The answer to the first question is *inadmissible*, because the user considers that the function `mergesort/2` is intended to sort lists of atoms, and the list contains an integer. In this case the debugger shows that the problem is in the function `take/2`. In fact, if we inspect the code in Section 3 (the box in the definition of `take`) we realize that it contains an error. It should be `[H | take(N - 1, T)]`.

After these changes all the errors have been solved. Note that only seven questions were required to locate two errors, one of them involving exceptions.

[3] Note that we have used here the debugging tree to explain the main ideas of the technique, although it is not needed during a standard debugging session.

7 Concluding Remarks and Ongoing Work

Debugging is usually the most time-consuming phase of the software life cycle, yet it remains a basically unformalized task. This is unfortunate, because formalizing a task introduces the possibility of improving it. With this idea in mind we propose a formal debugging technique that analyzes proof trees of erroneous computations in order to find bugs in sequential Erlang programs. A straightforward benefit is that it allows to prove the soundness and completeness of the scheme. Another benefit is that, since the debugger only requires knowing the intended meaning of the program functions, the team in charge of the debugging phase do 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 user functions, and thus abstracts away the implementation details. It can be viewed as complementary to the trace debugger already included in Erlang: first the declarative debugger is used for singling out an erroneous program function and then the standard tracer is employed for inspecting the function body. The declarative debugger is particularly useful for detecting wrong functions that produce unexpected exceptions, as shown in the running example included in the paper. 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.

We have used these ideas to implement a tool that supports different navigation strategies, trusting, and built-in functions, among other features. It has been used to debug several buggy medium-size programs, presenting an encouraging performance. More information can be found at `https://github.com/tamarit/edd`.

As future work we plan to improve the location of the bug inside a wrong function, which implies an extended setting taking into account the code defining the functions. Another interesting line of future work consists on extending the current framework to debug concurrent Core 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.

References

1. Alpuente, M., Ballis, D., Correa, F., Falaschi, M.: An integrated framework for the diagnosis and correction of rule-based programs. Theoretical Computer Science 411(47), 4055–4101 (2010)
2. Armstrong, J., Williams, M., Wikstrom, C., Virding, R.: Concurrent Programming in Erlang, 2nd edn. Prentice-Hall, Englewood Cliffs (1996)

3. Arts, T., Earle, C.B., Penas, J.J.S.: Translating Erlang to muCRL. In: Proceedings of the International Conference on Application of Concurrency to System Design, ACSD 2004, pp. 135–144. IEEE Computer Society Press (June 2004)
4. Benac Earle, C., Fredlund, L.A.: Recent improvements to the McErlang model checker. In: Proceedings of the 8th ACM SIGPLAN Workshop on ERLANG, ERLANG 2009, pp. 93–100. ACM, New York (2009)
5. Caballero, R.: A declarative debugger of incorrect answers for constraint functional-logic programs. In: Antoy, S., Hanus, M. (eds.) Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia, pp. 8–13. ACM Press (2005)
6. Caballero, R., Hermanns, C., Kuchen, H.: Algorithmic debugging of Java programs. In: López-Fraguas, F. (ed.) Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain. Electronic Notes in Theoretical Computer Science, vol. 177, pp. 75–89. Elsevier (2007)
7. Caballero, R., Martin-Martin, E., Riesco, A., Tamarit, S.: A calculus for sequential erlang programs. Technical Report 03/13, Departamento de Sistemas Informáticos y Computación (April 2013)
8. Carlsson, R.: An introduction to Core Erlang. In: Proceedings of the Erlang Workshop 2001, In Connection with PLI (2001)
9. Carlsson, R., Gustavsson, B., Johansson, E., Lindgren, T., Nyström, S.-O., Pettersson, M., Virding, R.: Core Erlang 1.0.3 language specification (November 2004), `http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`
10. 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)
11. Christakis, M., Sagonas, K.: Static detection of race conditions in Erlang. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 119–133. Springer, Heidelberg (2010)
12. Christensen, N.H.: Domain-specific languages in software development - and the relation to partial evaluation. PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark (July 2003)
13. Claessen, K., Svensson, H.: A semantics for distributed Erlang. In: Proceedings of the 2005 ACM SIGPLAN Workshop on Erlang, ERLANG 2005, pp. 78–87. ACM, New York (2005)
14. David, R., Mounier, G.: An intuitionistic lambda-calculus with exceptions. Journal of Functional Programming 15(1), 33–52 (2005)
15. de Groote, P.: A simple calculus of exception handling. In: Dezani-Ciancaglini, M., Plotkin, G. (eds.) TLCA 1995. LNCS, vol. 902, pp. 201–215. Springer, Heidelberg (1995)
16. Fredlund, L.-A.: A Framework for Reasoning about Erlang Code. PhD thesis, The Royal Institute of Technology, Sweden (August. 2001)
17. Hähnle, R., Baum, M., Bubel, R., Rothe, M.: A visual interactive debugger based on symbolic execution. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) 25th IEEE/ACM International Conference on Automated Software Engineering, ASE 2010, pp. 143–146. ACM (2010)
18. Huch, F.: Verification of erlang programs using abstract interpretation and model checking. SIGPLAN Not. 34(9), 261–272 (1999)
19. Hughes, J.: QuickCheck testing for fun and profit. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 1–32. Springer, Heidelberg (2007)

20. Insa, D., Silva, J.: An algorithmic debugger for Java. In: Lanza, M., Marcus, A. (eds.) Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010, pp. 1–6. IEEE Computer Society (2010)
21. Lindahl, T., Sagonas, K.: Detecting software defects in telecom applications through lightweight static analysis: A war story. In: Chin, W.-N. (ed.) APLAS 2004. LNCS, vol. 3302, pp. 91–106. Springer, Heidelberg (2004)
22. MacLarty, I.: Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne (2005)
23. Naish, L.: Declarative diagnosis of missing answers. New Generation Computing 10(3), 255–286 (1992)
24. Naish, L.: A declarative debugging scheme. Journal of Functional and Logic Programming 1997(3) (1997)
25. 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)
26. Nilsson, H., Sparud, J.: The evaluation dependence tree as a basis for lazy functional debugging. Automated Software Engineering 4, 121–150 (1997)
27. 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)
28. Pope, B.: Declarative debugging with Buddha. In: Vene, V., Uustalu, T. (eds.) AFP 2004. LNCS, vol. 3622, pp. 273–308. Springer, Heidelberg (2005)
29. Pope, B.: A Declarative Debugger for Haskell. PhD thesis, The University of Melbourne, Australia (2006)
30. Riesco, A., Verdejo, A., Martí-Oliet, N., Caballero, R.: Declarative debugging of rewriting logic specifications. Journal of Logic and Algebraic Programming 81(7-8), 851–897 (2012)
31. Sagonas, K., Silva, J., Tamarit, S.: Precise explanation of success typing errors. In: Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013, pp. 33–42. ACM, New York (2013)
32. Shapiro, E.Y.: Algorithmic Program Debugging. ACM Distinguished Dissertation. MIT Press (1983)
33. Silva, J.: A comparative study of algorithmic debugging strategies. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 143–159. Springer, Heidelberg (2007)
34. Silva, J.: A survey on algorithmic debugging strategies. Advances in Engineering Software 42(11), 976–991 (2011)
35. Svensson, H., Fredlund, L.-A.: A more accurate semantics for distributed Erlang. In: Proceedings of the 2007 SIGPLAN Workshop on ERLANG Workshop, ERLANG 2007, pp. 43–54. ACM, New York (2007)
36. Tessier, A., Ferrand, G.: Declarative diagnosis in the CLP scheme. In: Deransart, P., Hermenegildo, M.V., Maluszynski, J. (eds.) DiSCiPl 1999. LNCS, vol. 1870, pp. 151–174. Springer, Heidelberg (2000)