A Zoom-Declarative Debugger for Sequential Erlang Programs
(extended version)*

Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit

Technical Report 01/14
Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

June, 2014
(updated on September 9, 2014)

Abstract

We present a declarative debugger for sequential Erlang programs. The tool is started by the user when a program produces some unexpected result. It automatically builds a computation tree representing the structure of the erroneous computation. The tree is then traversed asking questions to the user about the validity of some nodes of the tree, which represent the results of intermediate subcomputations, until a bug in the program is found. The navigation process first concentrates in locating an erroneous function in the program. Then, the user can refine the granularity by zooming out the function, checking the values bound to variables and the if/case/try-catch branches taken during the execution. 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 in this calculus as debugging trees, which allows us to prove the soundness of the approach. The technique has been implemented in a tool publicly available. An exhaustive analysis of the usability of the tool is also presented.

Keywords: Declarative debugging, Erlang, Semantics, Zoom

1 Introduction

Erlang [12] is a programming language that combines the elegance and expressiveness of functional languages (higher-order functions, lambda abstractions, single assignments), with features required in the development of scalable commercial applications (garbage collection, built-in concurrency, and even hot-swapping). The language is used as the base of many fault-tolerant, reliable software systems. The development of this kind of systems is a complicated process where tools such as discrepancy analyzers [24], test-case generators [32], 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, and for this reason we think that alternative or complementary debugging tools are convenient.

Taking advantage of the declarative nature of the sequential subset of Erlang, we propose a new debugger based on the general technique known as declarative debugging [37]. Also known as declarative diagnosis or algorithmic debugging, this technique abstracts the execution details, which may be difficult to follow in declarative languages, to focus on the validity of the results. This approach has been widely employed in the logic [20] [11], functional [30] [33], multi-paradigm [6] [25], and object-oriented [23] [5] 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 us 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. Note that, since the process starts when an unexpected result is found, the technique is usually restricted to terminating computations.

*Research supported by MICINN Spanish project StrongSoft (TIN2012-39391-C04-04) and Comunidad de Madrid program PROMETIDOS (S2009/TIC-1465).
In order to obtain the debugging trees we present a semantic calculus for sequential Core Erlang programs, the intermediate language that Erlang uses to codify all the programs in a uniform representation. We selected this intermediate language because it is simpler than Erlang (i.e. with less and simpler syntactical constructs), which is very convenient to simplify our analysis. The debugging trees are defined as abbreviations of the proof trees obtained in this calculus. Each node in this 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 navigates the tree by asking questions to the user about the validity of some nodes until a buggy node—an invalid node with only valid children—is found, being its associated function the source of the error. Once a buggy function has been found, the user can proceed further by zoom debugging the function. One can picture this zoom as if each node of the tree was itself another debugging tree representing the computation of the code defining the buggy function. This second level of debugging is a new contribution to the general scheme of declarative debugging, since it allows to vary the granularity of the detected error depending on the necessities and the knowledge of the user. Note that the debugging trees for this stage are obtained from the same semantic calculus but are completely different from the ones in the previous phase, hence requiring a separate approach. In this phase the user is asked questions such as the particular function rule employed to obtain the erroneous result, the local variable bindings, or the branch selected in if/case/try-catch statements. The result is a particular fragment of code in the function previously pointed out as buggy by the tool.

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. Using these ideas we developed an Erlang tool supporting the declarative debugging of sequential Erlang programs. The tool provides features such as different navigation strategies, trusting, higher-order functions, support for built-ins, and “don’t know” answers. The present work extends and completes these results, that were presented in [8], by:

- Introducing zoom debugging, which allows the user to detect more specific errors in a function previously detected as buggy.
- Improving the debugging trees to support erroneous lambda abstractions. This extension requires a modification of the abbreviation function to include this kind of computation in the debugging tree, as well as new proofs for taking it into account.
- Updating our debugger to support the features listed above. Moreover, the navigation of the debugging tree has been eased by providing new answers that direct the current navigation strategy, reducing the number of questions.
- Compiling a study of the applicability of the system to real programs. We have applied our debugger to a wide range of applications developed by others, being able to detect non-documented errors. This gives us confidence on the usability of the tool and on its implementation.
- Giving a detailed description of the calculus. Besides the extra explanations, this calculus has been modified from the one in [8] by adding additional rules dealing with bound variables and with the path taken during the execution, which are required by the zoom.

The rest of the report is organized as follows: Section 2 describes the related work and the similarities with our approach. Section 3 introduces Erlang and the transformation to Core Erlang, illustrating it with an example. Section 4 presents the calculus we have tailored for sequential Core Erlang programs. Section 5 describes the different errors detected by our approach. Section 6 presents the transformations applied to the semantic calculus proof trees to apply declarative debugging, as well as the associated soundness and completeness results. Section 7 outlines the main features of our tool and describes debugging sessions for both standard and zoom debugging. Section 8 presents the experimental results obtained when using our tool, while Section 9 concludes and presents the future work. Appendix A presents additional information about Bit String, a particular feature of Erlang which is considered in a separated way to simplify the presentation. Finally, Appendix B includes the proofs of the theoretical results presented in Section 6.

2 Related work

First, we need a calculus to be able to build our debugging trees. 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 [21], Huch proposes, for a subset of Erlang, an operational small step semantics
based on evaluation contexts to only perform reductions in certain points of the expression. It covers single-node concurrency (spawning and communication by messages between processes in the same node) and reductions that can yield runtime errors. However, it does not cover other sequential features of the language like lambda abstractions or higher-order functions. Another important small-step semantics for Erlang is proposed in Fredlund’s Ph.D. thesis [19]. This semantics is similar to [21] and also uses evaluation contexts but covers a broader subset of the language including single-node concurrency, runtime errors, and lambda abstractions. However, it also lacks support for higher-order features. To overcome the limitations of Fredlund’s single-node semantics when dealing with distributed systems, [15] proposes a semantics based on Fredlund’s but adding another top-level layer describing nodes. This distributed multi-node semantics for Erlang was further refined and corrected in [10]. Besides standard operational semantics, other approaches have been proposed to formalize Erlang semantics like the one based on congruence in [14], which works with partial evaluation of Erlang programs.

Regarding Core Erlang [9, 10], there is no official semantics either. The only official document is the language specification [10], which contains a detailed but informal explanation of the expected behavior for the evaluation of expressions. Unlike Erlang, which has several formalizations by different authors, the only formalization of Core Erlang appears in [29, 28], created mainly for model checking Core Erlang programs using Maude [15]. That small-step operational semantics of Core Erlang follows the informal explanation in [10] and covers many aspects of the language like concurrency (message passing, spawning processes) and higher-order features. Though the semantics in [29, 28] fits nicely with the Rewriting Logic used by Maude, a small-step operational semantic is difficult to use for declarative debugging, since the conclusions of the inference rules contain values which are not fully evaluated, hence making the associated questions difficult to answer. Therefore, we propose here a natural (big-step) semantics for Core Erlang to complement the small-step semantics in [29, 28] and to base our declarative debugging approach to Core Erlang programs. The semantics we propose for Core Erlang—inspired in the natural semantics previously presented for Erlang, mainly [21, 19]—formalizes the behavior explained in [10] 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 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) [24], a completely automatic tool that performs static analysis to identify software discrepancies and bugs such as definite type errors [30], race conditions [13], unreachable code, redundant tests, unsatisfiable conditions, and more. Model checking tools have also received much attention in the Erlang community [21, 13], 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 are EUnit [11] and Quviq QuickCheck [22]. 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, Quviq QuickCheck is a commercial software that automatically generates and checks thousands of unit tests from properties stated by programmers.

It is interesting to see the main advantages of the declarative debugging approach w.r.t. the trace-debugger. We distinguish between the standard approach for debugging functions and the novel one performing zoom on them. In the first case, our declarative debugger provides more clarity and simplicity of usage for sequential programs. In the trace-debugger, programmers must compile their code with the debug_info option, and set some breakpoints where they want to stop the execution. From those points they can proceed step by step, checking whether the results of the functions or the arguments and bindings are the expected ones. If they skip the evaluation of a function but they discover it returns a wrong value, they have to restart the session to enter and debug its code. This is a burden even with conditional breakpoints, since conditions in the trace-debugger must be coded as boolean functions in a module. The advantage of the declarative debugger is that, starting only from an expression returning a wrong value, it finds a buggy function by simply asking about the results of the functions in the computation.
avoiding low-level details. It focuses on the intended meaning of functions, which is something very clear to programmers (or debuggers), and the built-in navigation strategies saves them from choosing what functions check and in what order as with breakpoints in the trace-debugger.

Comparing the zoom declarative debugger and the trace-debugger, both provide tools to detect a bug hidden in the code of a concrete sequential function. With a trace-debugger, programmers proceed instruction by instruction checking whether the bindings, the branches selected in \texttt{if/case/try-catch} expressions or inner function calls in the code of a suspicious function are correct. The zoom declarative debugger fulfills a similar task, asking only about the correctness of bindings and \texttt{if/case/try-catch} branch selections since it abstracts from inner function calls—they have been checked in the previous phase. Moreover, the navigation strategy automatically guides the debugging session over the \textit{zoom} computation tree 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 than the trace-debugger, although the complexity of the elements involved (meaning of variables, selected \texttt{if/case/try-catch} branches, ...) is similar. Since our debugging setting does not support concurrency (spawning processes, sending/receiving messages between processes) the trace-debugger is the only possibility for those programs. It has several features like step by step execution of processes in different nodes or the inspection of the message queues of processes that makes it a very valuable tool for debugging concurrent programs. In the near future we plan to extend our debugging setting so it can deal with concurrency, as it is a key feature of Erlang.

Declarative debugging is a well-known debugging technique. In [35] 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 [25], for Haskell in its declarative debugger Buddha [34], and for Java programs [23]. 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 [18, 17], have been proposed for functional languages. 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 [20], but these techniques are not closely related to declarative debugging, so we do not provide a detailed comparison.

3 Erlang and Core 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 \textit{hot swapping}.

Example 3.1. Figure [4] presents a Vigenère cipher written in Erlang, extracted from the programming chrestomathy Rosetta Code [4]. The program exports functions \texttt{encrypt/2} and \texttt{decrypt/2} that, given a text and a key, encipher or decipher it, respectively. The Vigenère cipher is a simple and well-known method for encrypting alphabetic text by adding modulo 26 (or subtracting, in the case of decrypting) the letters of the text and key in the same position. For example, to cypher the text \texttt{Attack tomorrow dawn} using the key \texttt{lennon} we have:

<table>
<thead>
<tr>
<th>Text</th>
<th>ATTACKTOMORROWDAWN</th>
</tr>
</thead>
<tbody>
<tr>
<td>Key</td>
<td>LEMONLEMONLEMONLEM</td>
</tr>
<tr>
<td>Cryptogram</td>
<td>LXFOPVXABCVKQLAZ</td>
</tr>
</tbody>
</table>

For instance, the second letter (T, letter \#19) is cyphered with \texttt{E} (letter \#4), so the resulting cyphered letter is \texttt{X}: \((19 + 4) \mod 26 = 23\). The third letter is also T but now it is cyphered with \texttt{M} (letter \#12), so

\[http://rosettacode.org/wiki/Vigen%C3%A8re_cipher#Erlang\] accessed May 9th, 2013.
\[ \text{mod}(X, Y) \rightarrow (X \text{ rem } Y + Y) \text{ rem } Y. \]

\[ \text{to_pos}(L) \text{ when } L \geq $A, L <= $Z \rightarrow L - $A. \]

\[ \text{from_pos}(N) \rightarrow \text{mod}(N, 26) + $A. \]

\[ \text{encipher}(P, K) \rightarrow \text{from_pos}(\text{to_pos}(P) + \text{to_pos}(K)). \]

\[ \text{decipher}(C, K) \rightarrow \text{from_pos}(\text{to_pos}(C) - \text{to_pos}(K)). \]

\[ \text{cycle_to}(N, \text{List}) \text{ when } \text{length}(\text{List}) \geq N \rightarrow \text{List}; \]

\[ \text{cycle_to}(N, \text{List}) \rightarrow \text{lists:append}(\text{List}, \text{cycle_to}(N - \text{length}(\text{List}), \text{List})). \]

\[ \text{normalize}(\text{Str}) \rightarrow \text{toupper}(\text{filter}(\text{fun} \text{isalpha/1}, \text{Str})). \]

\[ \text{crypt}(\text{RawText}, \text{RawKey}, \text{Func}) \rightarrow \]
\[ \text{PlainText} = \text{normalize}(\text{RawText}), \]
\[ \text{lists:zipwith}(\text{Func, PlainText}, \]
\[ \text{cycle_to}(\text{length}(\text{PlainText}), \text{normalize}(\text{RawKey}))). \]

\[ \text{encrypt}(\text{Text}, \text{Key}) \rightarrow \text{crypt}(\text{Text}, \text{Key}, \text{fun encipher/2}). \]

\[ \text{decrypt}(\text{Text}, \text{Key}) \rightarrow \text{crypt}(\text{Text}, \text{Key}, \text{fun decipher/2}). \]

Figure 1: Erlang code implementing Vigenère cipher

the result is the letter F: \((19 + 12) \mod 26 = 5\). As keys are usually shorter than the text to cypher, we need to repeat the key until it matches the length of the text.

Besides the exported functions encrypt/2 and decrypt/2, the program in Figure 1 contains some other functions that are used only internally. Function mod/2 computes \(X \mod Y\) guaranteeing that the result is positive even when \(X\) is negative, which may happen when deciphering. Functions from_pos/1 and to_pos/1 convert between positions in the alphabet and letters to perform the calculations. Functions encipher/2 and decipher/2 encipher and decipher a letter of the text using a letter of the key, respectively. cycle_to/2 repeats a list until it has a certain length. Function normalize/1 converts a string into uppercase and removes all the characters that are not letters. Function crypt/3 repeats the key until it has the same length as the text and applies a function Func as an argument to the letters in the text and the extended key. Finally, functions encrypt/2 and decrypt/2 encrypt and decrypt a text using a key by simply calling crypt/3 with the functions encipher/2 and decipher/2. The complete program also contains other simple functions like isalpha/1 to check whether the argument is a lowercase or uppercase letter, or toupper/1 that returns a letter or string in uppercase, but we have omitted them because they do not play any relevant rôle.

Observe that the code contains calls to built-ins and standard library functions, e.g. length/1 (a built-in) in line 6 or append/1 in line 7 and zipwith/1 in line 11, both of module lists. Users can encrypt the previous message using the following Erlang expression: vigenere:encrypt("Attack tomorrow dawn", "lemon"). The system evaluates this expression but instead of the expected cryptogram it returns an exception of type exception error: no function clause matching indicating that the function responsible for the error is lists:zipwith(#Fun<vigenere.2.122144413>,[],"#"). This is an unexpected result, an initial symptom indicating that there is some erroneous function in the program. However, the message does not provide useful information to fix the bug, since the function zipwith is correct. In the next sections we show how the debugger helps in the task of finding the bug.

The intermediate language Core Erlang [9 10] 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 to a number of arguments. exprs must be evaluated to a lambda-abstraction or a function name in the current module.
The general form of a case expression is:

\[
\text{case exprs of } \quad \begin{array}{c}
pats_1 \text{ when } exprs_1' \rightarrow exprs_1'' \\
\vdots \\
pats_n \text{ when } exprs_n' \rightarrow exprs_n'' 
\end{array} \quad \text{end}
\]

The expression exprs is called the argument. Each clause, also called branch, consists of a pattern pats\(_i\), an optional guard exprs\(_i'\), and a result exprs\(_i''\). The value \(v_i\) is not part of the code, it is added in our presentation as a label indicating a reference to the \(i\)th clause. The evaluation of a case expression is the evaluation of the result exprs\(_i''\) corresponding to the first clause whose pattern pats\(_i\) matches the value of the argument exprs and whose guard exprs\(_i'\) evaluates to true. There is always at least one clause fulfilling these conditions, as we explain below.

Core Erlang values represent the possible results of an expression evaluation. To make the semantic rules dealing with exceptions clearer, we have considered two categories: val, representing values that cannot contain an exception \(\xi\) at any position; and eval, representing values possibly with exceptions at some positions. These exceptions must contain the same system-dependent number of values \(m\) as the catch branch of the try expression. In contrast to Erlang, the evaluation of an expression in Core Erlang...
returns an ordered sequence \(<x_1, \ldots, x_n>\) of zero or more values. Sequences, which were added in Core Erlang to simplify the generation of efficient code and to allow certain optimizations to be performed at the core level [9], are used intensively in the translation from Erlang to Core Erlang (for example introducing case expressions that match several arguments at once, instead of nested chains of case expressions matching the arguments in order). We use evals and vals to differentiate between sequences of values possibly containing exceptions and sequences of values without exceptions, respectively. For simplicity, in single-value sequences we usually omit the angular brackets, so \(<x>\) and \(x\) are considered equal.

It is important to know some basis of how the translation from Erlang to Core Erlang is done. One of the most relevant details is that the body of a Core Erlang function is always a case expression representing the different clauses of the original Erlang function. Core Erlang case expressions, as shown below, always contain an extra clause whose pattern matches with any 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 to_pos/1 in Figure 1. The translation to Core Erlang produces the following code (we have omitted some code unrelated to the translation for the sake of readability):

```erlang
'to_pos'/1 = fun (_cor0) ->
  case _cor0 of
  <L> when (... call 'erlang':'>='(L, 65) ...) -> call 'erlang':'-'(L, 65)
  <cor3> when 'true' -> primop 'match_fail'('function_clause',_cor3)
  end

This generated clause will handle the case expression when the guard \(L \geq $A, L \leq $Z\) fails. 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 from_pos(to_pos(P) + to_pos(K)) at line 4 in Figure 1 is translated to Core as:

```erlang
let <cor3> = apply 'to_pos'/1(P)
in let <cor2> = apply 'to_pos'/1(K)
in let <cor4> = call 'erlang':'+'(cor3, cor2)
in apply 'from_pos'/1(cor4)
```

Let expressions are also used to translate sequences of Erlang expressions. For instance, the sequence of expressions going from line 10 to line 12 in Figure 1 is translated to the following Core Erlang code:

```erlang
let <PlainText> = apply 'normalize'/1(RawText)
...
in call 'lists':zipwith'(Func, PlainText, _cor6)
```

There are different Erlang expressions that can be translated to a Core Erlang case expression. Table 1 summarizes all the instances. For sake of simplicity, the translation of the inner expressions and patterns has been assumed. Erlang case expressions translation is straightforward. The only addition is the special clause to report a matching failure. This clause is present in all the Core Erlang case expressions with a different behavior (i.e. body) for each Erlang expression. Erlang if expressions are translated to Core Erlang case expressions without argument and patterns. An Erlang try expression introduces two Core Erlang case expressions: one for the try cases and another one for the error handler (i.e. catch section). While the last case is present in all the translated try expressions, the first one is not always needed. Note that Erlang allows to write try expressions of the form try expr catch ... end which would not introduce any Core Erlang case expression. Finally, the Erlang pattern matching operator uses the Core Erlang case expression to check whether the matching between the pattern and the expression is feasible. An exception to this translation rule is when the pattern is an unbound variable. In this case, a let expression is used.

Note that the translation from plain Erlang to Core Erlang 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.
These references, which are denoted by \( r \), and usually have a subindex establishing its origin, are unique identifiers pointing to the source code (this can be described in general with the tuple \((\text{mod}, \text{line}, \text{column})\) with \text{line} and \text{column} the starting position in the code of \text{mod}). This idea is used to refer to functions, variables, \text{case} expressions, their argument and their branches, and reserved words.

The set of variables occurring in an expression \( \text{expr} \) is denoted by \( \text{var}(\text{expr}) \). The notation \( \text{locvar}(r) \), with \( r \) a reference to either a function clause or to a lambda-expression, indicates the set of local variables defined in the body of the function/lambda-expression. The notation \( \text{ctx}(r_i) \) with \( r_i \) a reference to a lambda expression \( \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr} \) represents the context variables of \( r_i \), that is \( \text{ctx}(r_i) = \text{var}(\text{expr}) \setminus (\{\text{var}_1, \ldots, \text{var}_n\} \cup \text{locvar}(r_i)) \). Observe that the set of context variables for a named function is always empty, but the body of lambda-expressions defined inside named function can include local variables/arguments of the function in their bodies.

Second, the Core Erlang structure could be modified by optimization tools, and thus a particular structure cannot be assumed.

### 4 A Calculus for Sequential Core Erlang

This section first introduces the basic notions used in the calculus and then presents the main rules of our calculus for Sequential Core Erlang Programs (CESC in the following). The complete set of rules can be found in [7].

#### 4.1 Preliminaries

Throughout the rest of the paper we will use \text{references} to univocally identify some parts of the code. These references, which are denoted by \( r \) and usually have a subindex establishing its origin, are unique identifiers pointing to the source code (this can be described in general with the tuple \((\text{mod}, \text{line}, \text{column})\) with \text{line} and \text{column} the starting position in the code of \text{mod}). This idea is used to refer to functions, variables, \text{case} expressions, their argument and their branches, and reserved words.

The set of variables occurring in an expression \( \text{expr} \) is denoted by \( \text{var}(\text{expr}) \). The notation \( \text{locvar}(r) \), with \( r \) a reference to either a function clause or to a lambda-expression, indicates the set of local variables defined in the body of the function/lambda-expression. The notation \( \text{ctx}(r_i) \) with \( r_i \) a reference to a lambda expression \( \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr} \) represents the context variables of \( r_i \), that is \( \text{ctx}(r_i) = \text{var}(\text{expr}) \setminus (\{\text{var}_1, \ldots, \text{var}_n\} \cup \text{locvar}(r_i)) \). Observe that the set of context variables for a named function is always empty, but the body of lambda-expressions defined inside named function can include local variables/arguments of the function in their bodies.

Finally, when generating substitutions we will use in some cases the notation \( \hat{\theta} \), which stands for \( \theta \cup \bot \), being \( \bot \) the error substitution. In this way we indicate that the computation of the substitution
may fail, hence leading to $\bot$. Using these notions we can present the evaluations used in the calculus:

\[
\langle \text{guard}(r_b), \theta \rangle \rightarrow \text{val}, \text{which indicates that the guard of the branch referenced by } r_b \text{ has been evaluated to } \text{val}, \text{given the context in } \theta.
\]

\[
\langle \text{pathbind}(r_b, \text{vals}), \theta \rangle \rightarrow \hat{\theta}, \text{which indicates that, given the context } \theta, \text{the matching between the pattern in the branch referenced by } r_b \text{ and the value } \text{vals} \text{ is } \hat{\theta}.
\]

\[
\langle \text{fails}(\text{vals}, r_b), \theta \rangle, \text{which indicates that the branch referenced by } r_b \text{ is not taken when the } \text{case} \text{ argument is evaluated to } \text{vals} \text{ and the context is denoted by the substitution } \theta.
\]

\[
\langle \text{succeeds}(\text{vals}, r_b), \theta \rangle \rightarrow \theta', \text{which indicates that the branch referenced by } r_b \text{ is taken when the } \text{case} \text{ argument is evaluated to } \text{vals} \text{ and the context is denoted by the substitution } \theta, \text{ returning a substitution } \theta' \text{ binding the new variables in the pattern}.
\]

\[
\langle \text{vars, exprs}, \theta \rangle \rightarrow \theta', \text{which indicates that the variables in } \text{vars} \text{ are bound to the values obtained when evaluating the expression } \text{exprs}, \text{ giving rise to the substitution } \theta'.
\]

\[
\langle \text{exprs}, \theta \rangle \rightarrow \text{vals}, \text{which indicates that the value } \text{vals} \text{ is obtained when evaluating the expression } \text{exprs} \text{ with the context } \theta.
\]

\[
\langle r, \theta \rangle \rightarrow \text{vals}, \text{which indicates that the expression referenced by } r \text{ is evaluated to the value } \text{vals} \text{ when the context } \theta \text{ is used.}
\]

\[
\langle \text{c_arg}(r_c), \theta \rangle \rightarrow \text{evals}, \text{which computes the value } \text{evals} \text{ obtained when evaluating the argument of a } \text{case} \text{ expression, when } r_c \text{ is the reference to the } \text{case} \text{ expression and } \theta \text{ is the context.}
\]

\[
\langle \text{c_result}(r_b), \theta \rangle \rightarrow \text{evals}, \text{which computes the value } \text{evals} \text{ obtained when a branch of a } \text{case} \text{ expression is evaluated, where } r_b \text{ is the reference to the branch and } \theta \text{ is the current context.}
\]

We assume in all cases that all the variables appearing in the expressions or in the references are in the domain of $\theta$, and the existence of a global environment $\rho$ which is initially empty and is extended by adding the functions defined by the $\text{letrec}$ expressions. The notation $\text{CESC} \models_{(P, T)} \mathcal{E}$, 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 $\text{CESC} \not\models_{\rho} \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in CESC with respect to the program $P$. In our context, a program is an Erlang module. Nodes $N$ in a proof tree $T$ which are conclusions of an inference rule (R) are called (R)-nodes in the rest of the paper. Finally, we introduce the $\text{match}$ function to reproduce the syntactic matching performed by Erlang.

**Definition 4.1.** The $\text{match}$ function is defined as follows:

\[
\text{match}(\langle \text{pat}_1, \ldots, \text{pat}_n \rangle, \langle \text{val}_1, \ldots, \text{val}_n \rangle) = \theta_1 \sqcup \ldots \sqcup \theta_n
\]

where $\theta_i = \text{synMatch}(\text{pat}_i, \text{val}_i)$

\[
\text{match}(\text{pat}, \text{val}) = \text{synMatch}(\text{pat}, \text{val})
\]

\[
\text{synMatch}(\text{var}, \text{val}) = [\text{var} \mapsto \text{val}]
\]

\[
\text{synMatch}(\text{lit}_1, \text{lit}_2) = \text{id}, \text{if } \text{lit}_1 \equiv \text{lit}_2
\]

\[
\text{synMatch}(\langle \text{pat}_1, \text{pat}_2 \rangle, \langle \text{val}_1, \text{val}_2 \rangle) = \theta_1 \sqcup \theta_2, \text{where } \theta_i \equiv \text{synMatch}(\text{pat}_i, \text{val}_i)
\]

\[
\text{synMatch}(\{ \text{pat}_1, \ldots, \text{pat}_n \}, \{ \text{val}_1, \ldots, \text{val}_n \}) = \theta_1 \sqcup \ldots \sqcup \theta_n,
\]

where $\theta_i \equiv \text{synMatch}(\text{pat}_i, \text{val}_i)$

\[
\text{synMatch}(\text{var} = \text{pat}, \text{val}) = \theta[\text{var} \mapsto \text{val}], \text{where } \theta \equiv \text{synMatch}(\text{pat}, \text{val})
\]

\[
\text{synMatch}(\text{pat}, \text{val}) = \bot \text{ otherwise}
\]

We will present in the following the inference rules for the calculus, distinguishing between the rules for references, the rules generating values, and the rules generating and propagating exceptions.

### 4.2 Rules for references

We present here the rules dealing with references. As explained above, references are just a way to point to specific fragments of the code. However, we will shown in Section 5 that the distinction between references and the actual code is important, because it allows us to ask to the user whether the results obtained by executing the program are the ones expected.

The (PATBIND) axiom binds the variables in the pattern of the branch referenced by $r_b$ to the values $\text{vals}$. This matching generates the substitution $\theta$, which will be $\bot$ when the matching is not possible:
\( \text{(PATBIND)} \)

\[
\langle \text{patbind}(r_b, \text{vals}), \theta \rangle \rightarrow \hat{\theta}
\]

with \( r_b \) a reference to \( \text{pats} \) when \( \text{exprs}_1 \rightarrow \text{exprs}_2 \) and \( \hat{\theta} \equiv \text{match}(\text{pats}\theta, \text{vals}) \).

The \( \text{(GUARD)} \) rule evaluates the guard of the branch referenced by \( r_b \):

\( \text{(GUARD)} \)

\[
\langle \text{exprs}\theta, \theta \rangle \rightarrow \text{val} \quad \langle \text{guard}(r_b), \theta \rangle \rightarrow \text{val}
\]

where \( r_b \) is a reference to \( \text{pats} \) when \( \text{exprs}_1 \rightarrow \text{exprs}_2 \).

The \( \text{(FAIL}_1 \text{)} \) rule indicates that a branch cannot be executed when the pattern fails:

\( \text{(FAIL}_1 \text{)} \)

\[
\langle \text{patbind}(r_b, \text{vals}), \theta \rangle \rightarrow \bot \\
\langle \text{fails}(\text{vals}, r_b), \theta \rangle
\]

where \( r_b \) is a reference to \( \text{pats} \) when \( \text{exprs}_1 \rightarrow \text{exprs}_2 \).

The \( \text{(FAIL}_2 \text{)} \) rule is used when the matching succeeds but the \( \text{when} \) condition evaluated with the new substitution fails:

\( \text{(FAIL}_2 \text{)} \)

\[
\langle \text{patbind}(r_b, \text{vals}), \theta \rangle \rightarrow \theta' \\
\langle \text{guard}(r_b), \theta'' \rangle \rightarrow '\text{false}' \\
\langle \text{fails}(\text{vals}, r_b), \theta \rangle
\]

with \( \theta'' \equiv \theta \uplus \theta' \) and \( r_b \) a reference to \( \text{pats} \) when \( \text{exprs}_1 \rightarrow \text{exprs}_2 \).

The \( \text{(SUCC)} \) rule computes a new substitution when a branch is taken. This substitution consists of the new variables obtained from the matching with the pattern:

\( \text{(SUCC)} \)

\[
\langle \text{patbind}(r_b, \text{vals}), \theta \rangle \rightarrow \theta' \\
\langle \text{guard}(r_b), \theta'' \rangle \rightarrow '\text{true}' \\
\langle \text{succeeds}(\text{vals}, r_b), \theta \rangle \rightarrow \theta''
\]

with \( \theta'' \equiv \theta \uplus \theta' \), and where \( r_b \) is a reference to \( \text{pats} \) when \( \text{exprs}_1 \rightarrow \text{exprs}_2 \).

The \( \text{(BIND)} \) rule evaluates the given expression and binds the variables in the sequence to the values thus obtained:

\( \text{(BIND)} \)

\[
\langle \text{exprs}, \theta \rangle \rightarrow <\text{val}_1, \ldots, \text{val}_n > \\
\langle \text{vars}, \theta \rangle \rightarrow \{ \text{var}_1 \mapsto \text{val}_1, \ldots, \text{var}_n \mapsto \text{val}_n \}
\]

with \( r_1, \ldots, r_n \) references to variables \( \text{var}_1, \ldots, \text{var}_n \).

The \( \text{(BFUN)} \) rule evaluates a reference to a lambda-expression or a function, given a substitution binding all its parameters. This is accomplished by applying the substitution to the body (with notation \( \text{exprs}\theta \)) and then evaluating it:

\( \text{(BFUN)} \)

\[
\langle \text{expr}\theta, \theta \rangle \rightarrow \text{evals} \\
\langle \text{rf}, \theta \rangle \rightarrow \text{evals}
\]

where \( r_f \) references either to a function \( f/n = \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr} \), or to a lambda expression defined as \( \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr} \).

4.3 Rules for expressions

We present here the rules for evaluating expressions to values. The basic rule is \( \text{(VAL)} \), which states that values are evaluated to themselves:

\( \text{(VAL)} \)

\[
\langle \text{vals}, \theta \rangle \rightarrow \text{vals}
\]

The rule \( \text{(SEQ)} \) is in charge of evaluating a sequence of expressions, obtaining the final value for each expression:

\( \text{(SEQ)} \)

\[
\langle \text{expr}_1, \theta \rangle \rightarrow \text{val}_1 \\
\ldots \\
\langle \text{expr}_n, \theta \rangle \rightarrow \text{val}_n \\
\langle <\text{expr}_1, \ldots, \text{expr}_n>, \theta \rangle \rightarrow <\text{val}_1, \ldots, \text{val}_n>
\]

\[10\]
Similarly, the rules (TUP) and (LIST) evaluate tuples and lists, respectively:

\[
\text{(TUP): } \langle \text{exprs}_1, \theta \rangle \rightarrow \text{vals}_1 \quad \ldots \quad \langle \text{exprs}_n, \theta \rangle \rightarrow \text{vals}_n \\
\langle \{\text{exprs}_1, \ldots, \text{exprs}_n\}, \theta \rangle \rightarrow \{\text{vals}_1, \ldots, \text{vals}_n\}
\]

\[
\text{(LIST): } \langle \text{exprs}_1, \theta \rangle \rightarrow \text{vals}_1 \\
\langle \text{exprs}_2, \theta \rangle \rightarrow \text{vals}_2 \\
\langle \{\text{exprs}_1, \text{exprs}_2\}, \theta \rangle \rightarrow \{\text{vals}_1, \text{vals}_2\}
\]

The (CASE) rule is in charge of evaluating case expressions. It first evaluates the expression used to select the branch. Once this evaluation has been performed, it checks that the values thus obtained match the pattern on the \(i\)th branch and verify the guard, being this the first branch where this happens. Finally, the evaluation continues to compute the final result:

\[
\langle \text{fails}(<\text{vals}, r_1>), \theta \rangle \\
\langle \text{fails}(<\text{vals}, r_{i-1}>, \theta) \rangle \\
\ldots \\
\langle \text{succeeds}(<\text{vals}, r_i>, \theta) \rightarrow \theta' \rangle \\
\langle \text{c_result}(<r_i>, \theta'') \rightarrow \text{vals} \rangle \\
\langle \text{case}^c \text{exprs of clause}_1 \ldots \text{clause}_n \text{end}, \theta \rangle \rightarrow \text{vals}
\]

where \(\theta'' \equiv \theta \uplus \theta'\) and \(r_c\) is a reference to a case statement where each clause \text{clause}_i is defined as \text{pats}_i \text{ when } \text{exprs}'_i \rightarrow \text{exprs}''_i\) and the labels \(r_1, \ldots, r_n\) are references to the different branches that can be selected by the statement.

The (C_ARG) rule evaluates the argument of a case expression, represented by its reference, given a context:

\[
\langle \text{exprs}, \theta \rangle \rightarrow \text{vals} \\
\langle \text{c_arg}(<r_i>, \theta) \rightarrow \text{vals} \rangle
\]

with exprs the argument expression of the case referenced by \(r_c\).

The (C_RESULT) rule evaluates the body of the branch referenced by \(r_i\) with the context \(\theta\):

\[
\langle \text{exprs}, \theta, \theta \rangle \rightarrow \text{vals} \\
\langle \text{c_result}(<r_i>, \theta) \rightarrow \text{vals} \rangle
\]

with exprs the result expression of the case branch referenced by \(r_i\).

The (LET) rule first binds the variables and then the computation continues by applying the substitution thus obtained to the body:

\[
\langle \text{let } <\text{var}_1'', \ldots, \text{var}_n''> = \text{exprs}_1 \text{ in } \text{exprs}_2, \theta \rangle \rightarrow \text{vals} \\
\langle <\text{r}_1, \ldots, \text{r}_n>, \text{exprs}_1, \theta \rangle \rightarrow \theta' \\
\langle \text{exprs}, \theta'' \theta'' \rangle \rightarrow \text{vals} \\
\langle \langle \text{let } <\text{var}_1'', \ldots, \text{var}_n''> = \text{exprs}_1 \text{ in } \text{exprs}_2, \theta \rangle \rightarrow \text{vals} \rangle
\]

with \(\theta'' \equiv \theta \uplus \theta'\).

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

\[
\langle \text{exprs}_{n+1}, \theta \rangle \rightarrow \text{Atom}_1 \\
\langle \text{exprs}_{n+2}, \theta \rangle \rightarrow \text{Atom}_2 \\
\langle \text{exprs}_1, \theta \rangle \rightarrow \text{vals}_1 \\
\ldots \\
\langle \text{exprs}_n, \theta \rangle \rightarrow \text{vals}_n \\
\langle \text{call } \text{exprs}_{n+2}(\text{exprs}_1, \ldots, \text{exprs}_n), \theta \rangle \rightarrow \text{vals}
\]

where \(\text{Atom}_2/n\) is a function defined as \(\text{Atom}_2/n = \text{fun } \langle \text{var}_1, \ldots, \text{var}_n \rangle \rightarrow \text{expr} \) in the \text{Atom}_1 module (\text{Atom}_1) must be different from the built-in module \text{erlang}, \(\text{f} \) its reference, and \(\theta' \equiv \{\text{var}_1 \mapsto \text{vals}_1, \ldots, \text{var}_n \mapsto \text{vals}_n\}\).

Analogously, the (CALL_EXEC) rule is in charge of evaluating built-in functions in the \text{erlang} module.

To abstract the concrete system, we use an auxiliary function \text{exec} that returns the value the Erlang system would compute:

\[
\langle \text{exprs}_{n+1}, \theta \rangle \rightarrow \text{'erlang}' \\
\langle \text{exprs}_{n+2}, \theta \rangle \rightarrow \text{Atom}_2 \\
\langle \text{exprs}_1, \theta \rangle \rightarrow \text{val}_1 \\
\ldots \\
\langle \text{exprs}_n, \theta \rangle \rightarrow \text{val}_n \\
\langle \text{exec}(\text{Atom}_2, \text{val}_1, \ldots, \text{val}_n) = \text{vals} \rangle \\
\langle \text{call } \text{exprs}_{n+2}(\text{exprs}_1, \ldots, \text{exprs}_n), \theta \rangle \rightarrow \text{vals}
\]
where $\text{Atom}_2/n$ is a built-in function included in the erlang module.

The rule (APPLY$_1$) evaluates a function defined be means of a lambda-expression. It evaluates the function and the arguments and then uses them to obtain the value:

$\langle \text{exprs}, \theta \rangle \rightarrow r_\lambda$

$\langle \text{exprs}_1, \theta \rangle \rightarrow \text{vals}_1 \ldots \langle \text{exprs}_n, \theta \rangle \rightarrow \text{vals}_n$

$\langle r_\lambda, \theta' \rangle \rightarrow \text{eval}$

(APPLY$_1$)

and $\text{eval}$ with $r_\lambda$ a reference to $\text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr}'$, and $\theta' \equiv \theta \cup \{ \text{var}_1 \mapsto \text{vals}_1, \ldots, \text{var}_n \mapsto \text{vals}_n \}$.

Analogously, the rule (APPLY$_2$) evaluates a function defined in a letrec expression, thus contained in $\rho$. The rule first evaluates the arguments and then uses the definition of the function to reach the final result:

$\langle \text{exprs}, \theta \rangle \rightarrow \text{Atom}/n$

$\langle \text{exprs}_1, \theta \rangle \rightarrow \text{vals}_1 \ldots \langle \text{exprs}_n, \theta \rangle \rightarrow \text{vals}_n$

$\langle \text{exprs}'/\theta', \theta' \rangle \rightarrow \text{evals}'$

(APPLY$_2$)

if $\rho(\text{Atom}/n) = \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr}'$ and $\theta' \equiv \theta \cup \{ \text{var}_1 \mapsto \text{vals}_1, \ldots, \text{var}_n \mapsto \text{vals}_n \}$.

The rule (APPLY$_3$) 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:

$\langle \text{exprs}, \theta \rangle \rightarrow \text{Atom}/n$

$\langle \text{exprs}_1, \theta \rangle \rightarrow \text{vals}_1 \ldots \langle \text{exprs}_n, \theta \rangle \rightarrow \text{vals}_n$

$\langle r_f, \theta' \rangle \rightarrow \text{evals}$

(APPLY$_3$)

$\langle \text{apply}' \text{exprs}_1, \ldots, \text{exprs}_n, \theta \rangle \rightarrow \text{evals}$

where $\text{Atom}/n$ is a function defined in the module $r\_mod$ as $\text{Atom}/n = \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{expr}$, $r_f$ its reference, and $\theta' \equiv \{ \text{var}_1 \mapsto \text{vals}_1, \ldots, \text{var}_n \mapsto \text{vals}_n \}$.

The rule (PRIMOP) evaluates Erlang predefined functions by using the auxiliary function $\text{eval}$:

$\langle \text{exprs}_1, \theta \rangle \rightarrow \text{val}_1 \ldots \langle \text{exprs}_n, \theta \rangle \rightarrow \text{val}_n$

$\text{eval}(\text{Atom}, \text{val}_1, \ldots, \text{val}_n) = \text{val}'$

(PRIMOP)

$\langle \text{primop}\ \text{Atom}(\text{exprs}_1, \ldots, \text{exprs}_n), \theta \rangle \rightarrow \text{val}'$

The rule (TRY$_1$) evaluates a try expression when no exceptions are thrown. It just evaluates the argument expressions and continues with the expression in the body:

$\langle \text{exprs}, \theta \rangle \rightarrow \text{val}' \langle \text{exprs}', \theta'' \rangle \rightarrow \text{evals}$

(TRY$_1$)

with $\theta'' \equiv \text{match}<\text{var}_1, \ldots, \text{var}_n >, \text{val}', \theta'>$.

The rule (TRY$_2$) is in charge of evaluating try expressions throwing exceptions. It finds the pattern matching the exception and the evaluates the expression in the catch branch:

$\langle \text{exprs}, \theta \rangle \rightarrow \text{Excep}(\text{val}_1, \ldots, \text{val}_n) \langle \text{exprs}', \theta'' \rangle \rightarrow \text{evals}$

(TRY$_2$)

with $\theta'' \equiv \text{match}<\text{var}_1, \ldots, \text{var}_n >, \text{val}_1, \ldots, \text{val}_n >, \theta'>$.

4.4 Rules for exceptions

The rules for exceptions are in charge of generating exceptions in specific cases and propagating them. Since all of them are very similar, we only present here one rule for generating exceptions and another one for propagating them, while the rest of them are presented in [7]. The rule (SEQ_E) propagates the first exception thrown inside a sequence:

$\langle \text{expr}_1, \theta \rangle \rightarrow \text{val}_1 \ldots \langle \text{expr}_i, \theta \rangle \rightarrow \text{val}_i \langle \text{expr}_{i+1}, \theta \rangle \rightarrow \xi$

(SEQ_E)

$\langle \text{expr}_1, \ldots, \text{expr}_n >, \theta \rangle \rightarrow \xi$
The rule (CALL_E₄) throws a `bad_argument` exception when the module is not an atom:

\[
\begin{align*}
\text{(CALL_E₄)} : & \langle \text{exprs}', \theta \rangle \rightarrow \text{vals}'_1 & \langle \text{exprs}'_2, \theta \rangle \rightarrow \text{vals}'_2 \\
& \langle \text{exprs}, \theta \rangle \rightarrow \text{vals}_1 & \ldots & \langle \text{exprs}_n, \theta \rangle \rightarrow \text{vals}_n \\
\end{align*}
\]

if `vals'_1 is not an atom.

We can use this calculus to build the proof tree for any evaluation leading to a value. For example, we can build the tree for the buggy computation shown in Section 3. Figure 3 depicts part of this tree, where `txt` stands for "Attack tomorrow dawn", `key` for "lemon", `fun` for `fun 'encipher'/2`, `vals` for `<txt, key, vals'>` for `txt, key, fun 'encipher'/2`, the conditions when `true` and the arity of the functions have been removed, `θ₀ ≡ {_{\text{cor}1} \mapsto \text{txt} ; _{\text{cor}0} \mapsto \text{key}}, θ₁ ≡ θ₀ ∪ \{\text{Text} \mapsto \text{txt} ; \text{Key} \mapsto \text{key}}, θ₂ ≡ {_{\text{cor}2} \mapsto \text{txt} ; _{\text{cor}1} \mapsto \text{txt} ; _{\text{cor}0} \mapsto \text{fun}}. The root of the tree is shown on the top of the figure; it uses the (CALL) rule to evaluate the initial expression to an error message. The ∇'s in the premises of the root stand for the reflexive evaluation of the arguments of the program to themselves, while the leftmost subtree indicates that we must evaluate the reference to the function being evaluated, `encrypt/2`, using the substitution `θ₀`, which has bound the values to the dummy variables introduced by the transformation to Core Erlang. This is proven by evaluating the argument of the `case` expression to itself, checking that the first branch succeeds, and then evaluating the body of this branch with the new substitution, `θ₁`. This latter evaluation is proven in ∇₁.

Figure 3 (middle) describes ∇₁. It shows that the function `encrypt/3` is evaluated by reducing the function name and the arguments to themselves (in the ∇'s) and then the reference to the function is evaluated with the substitution `θ₂`, obtained by matching the parameters of the function. The proof for ∇₂ is shown in Figure 3 (bottom). As shown above, a `case` expression is computed by evaluating its argument, checking that the first branch succeeds, and using the extended substitution to continue with the evaluation of the body of the selected branch. The evaluation continues with ∇₃, which has a similar form to the ones shown here and will not be further explained.

5 Buggy Erlang Programs

In this section we define the errors that our proposal can detect. This is done by defining first the concept of intended interpretation, represented as \( I \), which corresponds to the behavior expected by the user for the basic pieces of the program. This behavior is extrapolated to complete programs by suitable calculi `ICECₐ` (during the first phase when the debugging concentrates only on user functions) and `ICECₐ` (inside a function code). The discrepancies between the results produced by the program, represented by `CESₐ` and the results expected by the user, represented by `ICECₐ`, `ICECₐ` define the buggy parts of the program.

5.1 Intended Interpretations

The intended interpretation of a program \( P \) is defined as the union of four sets:

\[
I = I_{\text{fun}} \cup I_{\lambda} \cup I_{\text{var}} \cup I_{\text{case}}
\]

The first two sets are used by the phase without zoom, and contain the intended interpretation of functions and lambda expressions, respectively:

- \( I_{\text{fun}} = \{\ldots, (r_f, \theta) \rightarrow \text{vals}, \ldots\} \) where
  - \( r_f \) is a reference to a function \( f/n \) defined as
    \[
    f/n = \text{fun}(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{exprs}
    \]
  - The domain of the substitution \( \theta \) must be \( \{\text{var}_1, \ldots, \text{var}_n\} \).

- \( I_{\lambda} = \{\ldots, (r_λ, \theta) \rightarrow \text{val}, \ldots\} \) where
  - \( r_λ \) is a reference to a lambda abstraction defined as
    \[
    \text{fun(\text{var}_1, \ldots, \text{var}_n) \rightarrow \text{exprs}}
    \]
  - \( \theta \) is a substitution such that \( \text{ctx}(r_λ) \cup \{\text{var}_1, \ldots, \text{var}_n\} \subseteq \text{dom(θ)} \).
Figure 3: Proof tree for encrypt(\texttt{"Attack tomorrow dawn"}, \texttt{"lemon"}), \textit{id}) → Exception(no function clause matching)
Thus, \( I_{\text{var}} \) contains the results expected for any possible function call to program functions. In the case of lambda we must take into account the context, that is the variables defined outside but employed in the body of the lambda. Thus, \( I_\lambda \) contains the results for any parameters and possible contexts.

The two other sets conforming \( I \) are used for zoom debugging. They are described as follows:

- \( I_{\text{var}} = \{\ldots, \langle r_1, \ldots, r_n \rangle \to \langle \text{val}_1, \ldots, \text{val}_n \rangle, \ldots\} \), with \( r_1, \ldots, r_n \) references to variables \( \text{var}_1, \ldots, \text{var}_n \).
- \( I_{\text{case}} = I_{r_1} \cup \ldots \cup I_{r_n} \) with \( r_{c_1}, \ldots, r_{c_n} \) references to all the \texttt{case} expressions occurring in the program and \( I_{r_1} \ldots I_{r_n} \) their intended interpretations (defined below).

The intended representation of variables \( I_{\text{var}} \) use the form of tuples because this is the format allowed in Core Erlang, although in most of the cases it corresponds to a single variable. For simplicity we can consider the (usual) case of only one variable, representing it by:

\[
I_{\text{var}} = \{\ldots, \langle r_v, \theta \rangle \to \text{val}, \ldots\}
\]

with \( r_v \) a reference to a variable \( v \).

The associated value \text{val} is the value expected by the user for this variable, introduced by assignment statements in the code and represented in the semantic calculus by the \texttt{(BIND)} inference rule. Observe that the same variable with the same reference \( r \) can occur more than once in \( I_{\text{var}} \) but associated to different contexts. For instance if we have defined in the code a variable \( x \) with the intended meaning of containing the double of another variable \( y \), we will have

\[
I_{\text{var}} = \{\ldots, \langle r_x, \{y \mapsto 2\} \rangle \to 4, \langle r_x, \{y \mapsto 3\} \rangle \to 6, \ldots\}
\]

The \( I_{\text{case}} \) set contains the intended interpretations of all the \texttt{case} expressions occurring in the code. They are particularly important because Core Erlang represents by \texttt{case} expressions not only the \texttt{case} expressions in the source code, but also the \texttt{if} expressions and the choice of the clause by a function call (see Section 3).

The values associated to the \texttt{case} expression depend on the context \( \theta \). Therefore each set \( I_{r_c} \), with \( r_c \) a reference to a \texttt{case} expression, is defined as the union of the intended interpretations for the different possible contexts:

\[
I_{r_c} = (I_{r_c}, \theta_1) \cup (I_{r_c}, \theta_2) \cup \ldots
\]

Assuming that \( r_c \) is a reference to a \texttt{case} expression following the syntax of Definition 3.1, with the labels \( r_1, \ldots, r_n \), referencing the different clauses (branches) that can be selected by the statement. Let \( i \) be the textual position of the clause that the user expects that will be selected for the \texttt{case} expression in the context \( \theta \). Then \((I_{r_c}, \theta)\) is a set with the form:

\[
(I_{r_c}, \theta) = \{c\_\text{arg}(r_c), \theta \to \text{evals}, \langle \text{pathbind}(r_1, \text{evals}), \theta \rangle \to \hat{\theta}_1, \langle \text{guard}(r_1), \theta'_1 \rangle \to 'false', \langle \text{fails}(r_1), \theta \rangle \langle \text{pathbind}(r_2, \text{evals}), \theta \rangle \to \hat{\theta}_2, \langle \text{guard}(r_2), \theta'_2 \rangle \to 'false', \langle \text{fails}(r_2), \theta \rangle \langle \text{pathbind}(r_1, \text{evals}), \theta \rangle \to \theta'_1, \langle \text{guard}(r_1), \theta'' \rangle \to 'true', \langle \text{arg}(r_i), \theta \rangle, \langle c\_\text{result}(r_i), \theta'' \rangle \to \text{evals} \}
\]

where:

- \( c\_\text{arg}(r_c), \theta \to \text{evals} \) represents the expected value (\text{evals}) obtained when evaluating the \texttt{case} argument in the context \( \theta \). Each set \((I_{r_c}, \theta)\) contains only one evaluation containing \( c\_\text{arg}(r_c) \).
- For every failing branch \( j, j < i, (I_{r_c}, \theta) \) includes \( \langle \text{pathbind}(r_j, \text{evals}), \theta \rangle \to \hat{\theta}_j, \langle \text{guard}(r_j), \theta \rangle \to \text{val}_j \), and \( \langle \text{fails}(r_j), \theta \rangle \), with:
  - \( \langle \text{pathbind}(r_j, \text{evals}), \theta \rangle \to \hat{\theta}_j \) represents the matching of \text{evals}, the value obtained after evaluating the \texttt{case} argument, and the pattern of the clause \( j \), \text{pats}_j, in the context \( \theta \). The expected substitution \( \theta_j \) can be \( \perp \) if the matching is not feasible.
  - \( \langle \text{guard}(r_j), \theta'_j \rangle \to \text{val}_j \) only occurs when \( \hat{\theta}_j \) is not \( \perp \), and in this case corresponds to the expected evaluation of the guard \text{exprs}'\( j \) in the context \( \theta'_j \) which must be of the form \( \theta'_j \equiv \theta \lor \hat{\theta}_j \).
\[
\langle \text{fail}(r_j), \theta \rangle \text{ indicates that the } j \text{th branch is expected to fail in this context.}
\]

Since \( j < i \), the clause \( j \) has not been selected and therefore we require that either \( \hat{\theta}_j \equiv \bot \) or \( \text{val}_j \equiv \text{'false'}. \)

- The evaluations \( \langle \text{pathbind}(r_i, \text{evals}), \theta \rangle \rightarrow \theta', \langle \text{guard}(r_i), \theta'' \rangle \rangle \text{ represent respectively the pattern matching and the evaluation of the guard of the successful clause, with } \theta'' \equiv \theta \oplus \theta'.

- \( \langle \text{succeeds}(r_i), \theta \rangle \) indicates that the \( i \)th branch is the successful one in this context.

- \( \langle c\_\text{result}(r_i), \theta'' \rangle \rightarrow \text{evals}' \) represents the evaluation of the expression associated to the \( i \)th clause, that is the expected value returned by the case statement.

Intended interpretations must fulfill the following property, which relates the results obtained when evaluating case expressions defining the body of functions.

**Property 5.1.** Let \( r_c \) be a reference to a case expression constituting the body of a user function \( f \) (respectively, of a lambda abstraction).

Then, \( \langle r_f, \theta \rangle \rightarrow \text{evals}' \in I_{\text{fun}} \) (respectively \( r_f, \theta \rightarrow \text{evals}' \in I_\lambda \)) iff \( \langle c\_\text{result}(r_i), \theta'' \rangle \rightarrow \text{evals}' \in I_{r_c} \), with \( r_f \) a reference to the success branch of the case expression referenced by \( r_c \) and \( \theta' \) a substitution extending \( \theta \).

### 5.2 Intended Semantic Calculus

The validity of the nodes in a CESC-proof tree is obtained defining two intended interpretation calculus. The first one, called ICESC\(_f\), contains the same inference rules as CESC, except by the replacement of (BFUN) by a new rule

\[
\text{(BFUN\(_z\))} : \langle r_f, \theta \rangle \rightarrow \text{evals} \quad \text{where either } r_f \text{ references to a function and } \langle r_f, \theta \rangle \rightarrow \text{vals} \in I_{\text{fun}}, \text{ or } r_f \text{ references to a lambda expression and } \langle r_f, \theta \rangle \rightarrow \text{vals} \in I_\lambda.
\]

The second calculus is called ICESC\(_z\) (intended CESC calculus with zoom). This calculus, in addition to replacing (BFUN) by (BFUN\(_z\)), replaces the inferences (BIND), (FAIL\(_1\)), (FAIL\(_2\)), (SUCCE\(_\lambda\)), (C\_ARG), (C\_RESULT), (PATBIND) and (GUARD) by new rules with the same name subscripted by \( z \). In the following \( r_c \) is a reference to a branch of a case expression with reference \( r_c \).

\[
\text{(BIND\(_z\))} : \langle < r_1, \ldots, r_n >, \text{exprs}, \theta \rangle \rightarrow \{ \text{var}_1 \mapsto \text{val}_1, \ldots, \text{var}_n \mapsto \text{val}_n \} \quad \text{with } \langle < r_1, \ldots, r_n >, \theta \rangle \rightarrow < \text{val}_1, \ldots, \text{val}_n > \in I_{\text{var}}
\]

\[
\text{(FAIL\(_z\))} : \langle \text{fail}(\text{vals}, r_c), \theta \rangle \quad \text{with } (\langle \text{fail}(r_c), \theta \rangle) \in (I_{r_c}, \theta)
\]

\[
\text{(SUCCE\(_z\))} : \langle \text{succeeds}(\text{vals}, r_c), \theta \rangle \rightarrow \theta' \quad \text{with } (\langle \text{succeeds}(r_c), \theta \rangle) \in (I_{r_c}, \theta)
\]

\[
\text{(C\_ARG\(_z\))} : \langle \text{c\_arg}(r_c), \theta \rangle \rightarrow \text{vals} \quad \text{with } (\langle \text{c\_arg}(r_c), \theta \rangle) \in (I_{r_c}, \theta)
\]

\[
\text{(C\_RESULT\(_z\))} : \langle \text{c\_result}(r_c), \theta' \rangle \rightarrow \text{vals} \quad \text{with } (\langle \text{c\_result}(r_c), \theta' \rangle) \in (I_{r_c}, \theta)
\]

\[
\text{(PATBIND\(_z\))} : \langle \text{patbind}(r_c, \text{vals}, \theta) \rangle \rightarrow \theta \quad \text{with } (\langle \text{patbind}(r_c, \text{vals}, \theta) \rangle) \in (I_{r_c}, \theta)
\]

\[
\text{(GUARD\(_z\))} : \langle \text{guard}(r_c), \theta' \rangle \rightarrow \text{val} \quad \text{with } (\langle \text{guard}(r_c), \theta' \rangle) \in (I_{r_c}, \theta)
\]

That is, ICESC\(_f\) and ICESC\(_z\) replace the computed results by their intended values, in the first case concentrating in functions, and in the second case considering the code inside functions. In the following when it is interesting to establish some statement that can be applied to any of the two calculi we write ICESC\(_z\). Analogously to the case of CESC, the notation ICESC\(_z\) \( \models_{(P, I, T)} E \) indicates that the evaluation \( E \) can be proven w.r.t. the program \( P \) and the intended interpretation \( I \) with proof tree \( T \) in ICESC\(_z\), while ICESC\(_z\) \( \not\models_{(P, I, T)} E \) indicates that \( E \) cannot be proven in ICESC\(_z\). The tree \( T \), the program \( P \), and the intended interpretation \( I \) are omitted when they are not needed.
CESC is used by our tool in order to represent actual Erlang computations, while ICESC⁰ represents the knowledge that the tool obtains from the user and is used as an oracle to determine the validity of the evaluations:

Definition 5.1. Let \( P \) be a Core Erlang Program, let \( T \) be a CESC proof tree with respect to \( P \), and let \( N \) be a node in \( T \) containing an evaluation \( \mathcal{E} \).

1. \( N \) is valid with respect to \( ICESC_0 \) when \( ICESC_0 \models_{(P,T)} \mathcal{E} \), and invalid when \( ICESC_0 \not\models_{(P,T)} \mathcal{E} \).
2. \( N \) is called buggy with respect to \( ICESC_0 \) if \( \mathcal{E} \) is invalid with all its children valid (in both cases w.r.t. \( ICESC_0 \)).

Instead of valid/buggy w.r.t. \( ICESC_0 \) we often use in the rest of the paper the notation \( ICESC_0 \)-valid/\( ICESC_0 \)- buggy. The idea behind these definitions is that a buggy node represents an erroneous computation based on correct subcomputations. Thus, the piece of code associated to a buggy node represents an error in the program. This informal idea will be formalized in the rest of the section. The rôle of the two calculi is further clarified by the next two assumptions:

Assumption 5.1. Let \( P \) be an Erlang program and \( \cdot \mid \cdot \) the transformation that converts an Erlang expression into a Core expression. Then:

1. An evaluation \( e\theta \rightarrow eval \) is computed by some Erlang system with respect to \( P \) iff \( CESC \models_P (\langle |e|, \theta \rangle \rightarrow |eval|) \).
2. A reduction \( e\theta \rightarrow eval \) computed by some Erlang system is considered unexpected with respect to \( ICESC_0 \) by the user iff \( ICESC_0 \not\models_{(P,I)} (\langle |e|, \theta \rangle \rightarrow |eval|) \).

The next definition introduce terminology useful for the rest of the paper:

Definition 5.2.

1. The term zoom inference refers to either the \((C\_ARG), (SUCC), (BIND), (PATBIND), (GUARD), (FAIL_1), (FAIL_2), \) or \((C\_RESULT) \) CESC inference rule.
2. Let \( T \) be a tree with nodes labeled by inference rules. We say that a node \( M \) is an immediate \((R)\)-descendant of another node \( N \) if:

   (a) Either \( M \) is a descendant of \( N \) or \( M \equiv N \).
   (b) \( M \) is the conclusion of a inference labeled by \((R)\).
   (c) In the path from \( N \) to \( M \) (excluding both nodes) there are no nodes labeled by \((R)\).

Now we are ready to define the errors detected by our tool.

5.3 Errors in Core Erlang Programs

The errors are detected as discrepancies between the actual computations represented by \( CESC \) and the ‘ideal’ computations expected by the user, represented by \( ICESC_0 \). The two following mutually recursive definitions detail the different errors found in Core Erlang programs. We start defining the errors due to the choice of the erroneous branch in a case expression.

Definition 5.3. Let \( P \) be a Core Erlang program with intended interpretation \( I \). Let \( r_c \) a reference to a case expression in \( P \) and \( r_i \) a reference to the \( i \)th branch in \( r_c \), which must be of the form: \( pats_i \) when \( exprs_i' \rightarrow exprs_i'' \). We say that the \( i \)th branch of the case \( r_c \) is a wrong branch when there exists a value \( evals \) and a context \( \theta \) such that some of the followings items hold:

1. Suppose that
   \[
   \begin{align*}
   ICESC_Z \models_{(P,I)} \text{succeeds}(evals, r_i, \theta) \rightarrow \theta' \\
   ICESC_Z \models_{(P,I)} \text{pathbind}(r_i, evals, \theta) \rightarrow \theta''
   \end{align*}
   \]
   \( \theta' = \theta \uplus \theta'' \) and at least one of the following items hold:

   (a) \( \text{match}(pats_i, \theta, evals) \neq \theta'' \). In this case we say also that \( \langle r_i, \theta \rangle \) contains a wrong fail pattern instance.
Definition 5.4. We say that the first erroneous branch: 'true' we have found an error in the guard definition. In particular we are interested in errors associated to wrong program: pattern definition. Analogously, if we expect the branch result, but we obtain a different substitution after performing the matching we have found an error in the pattern definition. Then, we also say that \( r_i, \theta \) contains a wrong success guard instance.

2. Suppose that
   \[ \text{ICESC}_Z \models_{(P,I)} \langle \text{guard}(r_i), \theta \rangle \rightarrow \text{true} \]
   \[ \text{ICESC}_Z \not\models_{(P,I)} \langle \text{exprs}', \theta \rangle \rightarrow \text{false} \]

then, we also say that \( r_i, \theta \) contains a wrong fail guard instance.

Definition 5.5. Let \( P \) be a Core Erlang program with intended interpretation \( I \). Then we say that \( P \) is a wrong program in any of the following cases:

1. Let \( r_j \) be a reference to a either a function \( \text{fun} = \text{fun}(\var_1, \ldots, \var_n) \rightarrow B \), or to a lambda function \( \text{fun}(\var_1, \ldots, \var_n) \rightarrow B \) and \( \theta \) a substitution. Then \( \langle r_j, \theta \rangle \) is a wrong function instance if
   \[ \text{ICESC}_I \models_{(P,I)} \langle B\theta, \theta \rangle \rightarrow \text{evals} \]
   \[ \text{ICESC}_I \not\models_{(P,I)} \langle r_j, \theta \rangle \rightarrow \text{evals} \]

2. Let \( \langle \var_1, \ldots, \var_n, \text{exprs}, \theta \rangle \) be such that \( \var_1, \ldots, \var_n \) are references to the variables \( \var_1, \ldots, \var_n \) and \( \theta \) is a context.

   \[ \langle \var_1, \ldots, \var_n, \text{exprs}, \theta \rangle \] is a wrong binding instance iff there exists a substitution \( \theta' \) defined as \( \var_1 \mapsto \text{val}_1 \cup \ldots \cup \var_n \mapsto \text{val}_n \) verifying:
   \[ \text{ICESC}_Z \models_{(P,I)} \langle \text{expr}, \theta \rangle \rightarrow \theta' \]
   \[ \text{ICESC}_Z \not\models_{(P,I)} \langle \var_1, \ldots, \var_n, \text{exprs}, \theta \rangle \rightarrow \theta' \]

3. Let \( r_c \) be a reference to a case expression (with the structure of Definition 3.1). Then \( \langle r_c, \theta \rangle \) is a wrong case expression instance if it contains any of the following errors:
   \[ \langle \text{c_arg}(r_c), \theta \rangle \] is a wrong case argument instance if there exists a value \( \text{evals} \) such that
   \[ \text{ICESC}_Z \models_{(P,I)} \langle \text{exprs}, \theta \rangle \rightarrow \text{evals} \]
   \[ \text{ICESC}_Z \not\models_{(P,I)} \langle \text{c_arg}(r_c), \theta \rangle \rightarrow \text{evals} \]
   \[ \langle r_c, \theta \rangle \] contains a first wrong branch (Definition 5.4).
   \[ \langle \text{c_result}(r_c), \theta' \rangle \] is a wrong case result instance iff exists values \( \text{evals}, \text{evals}' \) such that:
The diagram in Figure 4 summarizes the different errors that can be found in a Core Erlang program (the labels after the symbol | are explained later). The point here is that for example 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 (item 1 in Definition 5.5) corresponds to a function whose body \( B \) produces an erroneous result even assuming that all the functions in its body are correct. Analogously a wrong binding instance is obtained when the expression that generates the value produces an unexpected result (the substitution \( \theta' \) in Definition 5.5.2). The definition of wrong case expression instance (item 3 in Definition 5.5) is more complicated, because a case expression can be erroneous due to four different bugs. Moreover, two of these bugs can be further divided in two (see Figure 4).

The first bug in a case expression is that the argument of the statement is erroneous. This happens when the result produced by its evaluation in ICESC\( Z \) is not an expected value in the context (expected values represented by c_arg facts in the intended interpretation).

The purpose of the following definition is to introduce the expression producing the initial symptom as part of the program.

**Definition 5.6.** Let \( P \) be a Core Erlang program and \( e \) an Erlang expression using functions defined in \( P \). Then, \( P_{(e,\theta)} \) denotes the (Core Erlang) program \( P \cup \{ \text{main/0 = fun() -> case <> of <> when 'true' -> [\theta] end } \} \), with \(|·|\) the operator converting Erlang code into Core Erlang code, and \( \text{main} \) a new identifier in \( P \).

This technical construction is useful for the theoretical results in the rest of the paper.

### 5.4 Errors in Erlang

So far we have described intended interpretations and errors in Core Erlang programs. But the user is writing Erlang programs, and the intended interpretations include also the expected results if, try or matching expressions, among other. Fortunately the relation between Erlang and Core Erlang is quite close and it is possible for the debugger to relate the original Erlang expressions to their translations and thus adapt the questions performed to the user to Erlang intended interpretations.
For instance, we can see in Table 1 that if expressions are converted into case expressions. Then the debugger will avoid questions about the case argument and about the clause patterns (which are trivially valid), and instead of asking about the ‘case branches’ will take about ‘if conditions’.

5.5 Inadmissible evaluations

Since we are dealing with buggy programs, sometimes the user must consider surprising unexpected evaluations. For instance imagine that fib($n$) is intended to represent the $n$th Fibonacci number, and we get a question about the validity of fib(’4’) → 4. Or suppose we have a local variable part of a context of a lambda-expression which is supposed to contain an integer but it is a list. Are the evaluations obtained in such cases part of $I_{fun}$ or $I_{\lambda}$? The answer is yes.

From the logic point of view both the parameters and the context in a function or the context in a lambda expression must be considered premises of an implication. That is, the user must wonder: ‘If I assume these values as parameters (or this context), then the expected result is...?’ And given false premises (like that ’a’ is a possible parameter for function fib) any conclusion is valid. Of course, answering that fib(’4’) → 4 is valid can be counterintuitive and, for this reason, the implementation includes the answer i standing for inadmissible in these cases, although internally inadmissible is simply another word representing yes. Indeed, inadmissible means that something is going wrong, but the error is elsewhere: in the function call with erroneous arguments, or in the binding of the variable that produces an inadmissible context. Therefore, in the case of inadmissible contexts the debugger enquires which variable has an unexpected value in order to direct the search for the buggy node.

6 Debugging with Abbreviated Proof Trees

In this section we present the debugging technique in two phases. During the first phase we should only consider function calls and their results, disregarding the code inside functions. The second phase considers a buggy function found in the first phase and locates a more detailed error in the function code. However, the CESC tree contains information about the two phases together in a single tree. In order to overcome this difficulty, we abbreviate the CESC proof trees keeping only the relevant information for detecting buggy nodes during each phase.

The following subsection shows how to find erroneous functions (both “named” defined functions and lambda-abstractions). Then, we show in Section 6.2 how to locate more detailed errors inside the function previously reported as buggy.

6.1 First Phase: debugging wrong functions

The first goal is to locate a wrong function asking questions only about the expected behavior of functions. This is done by defining the following abbreviated tree:

**Definition 6.1.** Let $P$ be a program and $(e, \theta) \rightarrow eval$ an evaluation. A tree $T$ is an Abbreviated Proof Tree (APT in short) for the evaluation with respect to $P$, if there is a proof tree $T'$ such that CESC $\vdash (P_{e, \theta}, T') \langle r_{main}.id \rangle \rightarrow eval$, and $T = APT(T')$ with the transformation $APT$ defined as follows:

\[
APT\left(\frac{T}{\theta}^{BFUN}\right) = \frac{APT(T)}{\theta}
\]

\[
APT\left(\frac{T_1 \ldots T_n}{\theta}^{R}\right) = APT(T_1) \ldots APT(T_n), \ for \ (R) \neq (BFUN)
\]

The transformation keeps only 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. The structure of the APTs is similar to the evaluation dependence tree employed in functional languages \[31\]. 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 6.1.** Let $P$ be a Core Erlang program, $I$ its intended interpretation, and $e\theta \rightarrow eval$ an evaluation unexpected with respect to ICESCf. Then, exists at least one proof tree $T'$ such that CESC $\vdash (P_{e, \theta}, T') \langle r_{main}.id \rangle \rightarrow |eval|$, and every tree $T'$ in these conditions verifies that $T \equiv APT(T')$ contains at least
one buggy node $N$, and every buggy node in $T$ corresponds to a wrong instance of a user function different from $\text{main/0}$.

Therefore, during the first step of the debugging process the user only needs to answer questions about the validity of nodes in the APT, which corresponds exactly to the evaluations produced by functions. For example, Figure 5 shows the APT corresponding to the proof tree in Figure 3. As explained above, it only keeps the inferences corresponding to (BFUN) rules. The $\nabla$’s on top of the tree correspond to applications of the functions used directly by $\text{crypt/3}$.

6.2 Second phase: zooming in for errors

Once a function has being pointed out as buggy, the exact location of the error may still be difficult to find. The existence of different paths that the execution can follow inside the body function due to the existence of nested case expressions, or the possible values taken by the variables, complicate the situation. Therefore, we introduce a new abbreviation of the proof trees presented in Section 4 that allows us to find the rest of errors included in Definition 5.5. These trees are rooted by the node corresponding to the buggy function applications in the first step, and hence we say that we “zoom in” this node to refine the bug location.

**Definition 6.2.** Let $P$ be a program and $(e, \theta) \rightarrow \text{eval}$ an unexpected evaluation w.r.t. $P$. Let $T'$ be a proof tree such that CESC $=_{(\nu_{e, \theta})} (\nu_{\text{main/id}}) \rightarrow \text{eval}$, and $T$ a subtree of $T'$ whose root is the conclusion of a (BFUN) inference rule. Then the Abbreviated Proof Tree with zoom of $T$, $\text{APTZ}(T)$ is defined as follows:

\[
\text{(APTZ}_1) \quad \text{APTZ}\left(\frac{T_1 \ldots T_n}{\varepsilon}\right) =_{(R)} \text{APTZ}(T_1) \ldots \text{APTZ}(T_n)
\]
with (R) either any zoom inference or a (CASE) inference.

\[
\text{(APTZ}_2) \quad \text{APTZ}\left(\frac{T_1 \ldots T_n}{\varepsilon}\right) = \emptyset,
\]
if $\varepsilon$ is not the root of $T$.

\[
\text{(APTZ}_3) \quad \text{APTZ}\left(\frac{T_1 \ldots T_n}{\varepsilon}\right) = \text{APTZ}(T_1) \ldots \text{APTZ}(T_n),
\]
if neither of the previous rules is applicable.

The occurrence of $\emptyset$ in the second rule indicates that (BFUN) inferences not at the root of $T$ are simply eliminated, including their subtrees. Also note that the (BFUN) function in the root is dealt with the third APTZ rule. The APTZ is used in our framework to find buggy nodes, which correspond to errors in the body of function definitions.

In the case of (PATBIND) and (GUARD) buggy nodes we are interested only in those that correspond to incorrect branches of case expressions.

**Definition 6.3.** Let $N$ be a (PATBIND)/(GUARD) ICESCZ-buggy node in some APTZ $T$. Then we say that $N$ is a wrong case branch witness for the $i$th branch of a case expression $r_c$ if:

1. $N$ is the premise a ICESCZ-invalid (SUCC), (FAIL1), or (FAIL2) node $M$, and $M$ contains the reference to some branch $i$ of a case expression referenced by $r_c$.
2. $T$ does not contain a (C_ARG) buggy node for the same case expression.
3. $T$ does not contain another node verifying the first item for some branch $j < i$ of the same case expression.
Let $P$ be a Core Erlang program, $\mathcal{I}$ its intended interpretation, and $\theta \mapsto \text{eval}$ an unexpected evaluation. Let $T'$ be a proof tree such that $\text{CESC} \models_{\langle P_{\text{core}}, T' \rangle} \{ r_{\text{main} \cdot \text{id}} \mapsto \text{eval} \}$. Let $T''$ be defined as $T'' = \text{APT}(T')$. $M$ be an ICESC bug node in $T''$, and $T_M$ the subtree of $T''$ rooted by $M$. Finally, define a new tree $T$ as $T \equiv \text{APTZ}(T_M'')$. Then $T$ contains at least one ICESC-buggy node $N$ that verifies one of the following items:

1. $N$ is a (BIND) node of the form $\langle r_1, \ldots, r_n \triangleright \text{exprs}, \theta \rangle \mapsto \{ v_1 \mapsto \text{val}_1, \ldots, v_n \mapsto \text{val}_n \}$

Then, $\langle r_1, \ldots, r_n \triangleright \text{exprs}, \theta \rangle$ is a wrong binding instance.

2. $N$ a (C_ARG) node, $N \equiv \langle \text{c_arg}(r_c), \theta \rangle \mapsto \text{evals}$. Then, $\langle \text{c_arg}(r_c), \theta \rangle$ is a wrong case argument instance.

3. $N$ is a (PATBIND)/(GUARD) premise of a (FAIL1)/(FAIL2) node $P$, $P \equiv \langle \text{fails}(\text{vals}, r_i), \theta \rangle$, with $r_i$ a reference to the $i$th branch of a case expression $r_c$, and such that $N$ is a wrong case branch witness. Then $(r_i, \theta)$ is an unexpected case failure instance, and in particular:

   (a) If $N$ is (PATBIND) node then $(r_i, \theta)$ is a wrong failure pattern instance.

   (b) If $N$ is a (GUARD) node, then $(r_i, \theta)$ is a wrong failure guard instance.

4. $N$ is a (PATBIND)/(GUARD) premise of a (SUCC) node $P$, with $P \equiv \langle \text{succeeds}(\text{vals}, r_i), \theta \rangle$, $r_i$ a reference to the $i$th branch of a case expression $r_c$, and such that $N$ is an wrong case branch witness. Then $(r_i, \theta)$ is an unexpected case success instance, and in particular:

   (a) If $N$ is a (PATBIND) node, then $(r_i, \theta)$ is a wrong success pattern instance.

   (b) If $N$ is a (GUARD) node, then $(r_i, \theta)$ is a wrong success guard instance.

5. $N$ is a (C_RESULT) node of the form $\langle \text{c_result}(r_c), \theta \rangle \mapsto \text{evals}$, and there is neither a (C_ARG) buggy node nor a wrong case branch witness for the same case expression. Then, $\langle \text{c_result}(r_c), \theta \rangle \mapsto \text{evals}$ is a wrong case result instance.

Theorem 6.2 indicates that the zooming phase concludes pointing out a more detailed error inside the buggy function. The inference rules labeling the different errors in Figure 2 summarize the relation between buggy nodes and bugs in the program. It is worth observing that the criterium used to select the buggy node $N$ is used explicitly by the tool debugger (asking first about the validity of the argument, then about the branches, finally about the result). Actually, the tool focuses on first wrong branches by asking explicitly to the user about the position of the expected branch $i$ (after ensuring that the argument evaluation is correct). If there is a (SUCC) node for $j < i$ then $j$ is chosen as first wrong branch. If on the contrary $i$ is a (FAIL1) or (FAIL2) then $i$ is chosen as first wrong branch. This correctly implements the ideas of Definition 5.4. The next section explains more details about the implementation.

7 System Description

The technique described in the previous sections—both the detection of wrong functions and the zoom version—has been implemented in Erlang. The tool, called edd (Erlang Declarative Debugger), is publicly available at https://github.com/tamarit/edd.

When the user detects an unexpected result in an expression evaluation, it calls the debugger passing that expression. The system edd proceeds by asking questions about the validity of some parts of the evaluation until it locates a buggy node, i.e., a function that is causing an error. From that point the user can choose to stop the debugging session and inspect manually the function code to locate the bug, or continue with a zoom debugging session that will help to find the concrete error inside the buggy function.

When debugging wrong functions (Section 6.1), the debugger questions are simply of the form $\text{call} = \text{value} \mapsto \theta$, asking whether the function or $\lambda$-abstraction application $\text{call}$ should evaluate to $\text{value}$. In response to the debugger questions, the possible answers are:

- **yes (y):** the evaluation is correct. In this case the node is marked as valid.
• no (n): the evaluation is not correct. In this case the node is marked as 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 nodes related to this function are marked as valid.

• inadmissible (i): the question does not apply because the arguments should not take these values. The node is marked as valid.

• don’t know (d): the answer is unknown. The node 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 the zoom approach (Section 6.2), the debugger asks more complicated questions, and hence the answers yes, no and inadmissible cannot be used in some cases (the rest of answers are always available). The answer trusted has not any sense in this approach, therefore it is never available. More specifically, the questions are of the form:

• When asking about a case node, the debugger distinguishes the different sources of error to ease the process. Each item presented in this catalog assumes that the previous ones are correct, so the user must select the first wrong item; otherwise, the last option indicates that everything is correct. The options, which are displayed after the case expression to allow the user to identify the code being debugged, are:

  The context of the expression, that is, the variables used to evaluate the argument and all the branches until the successful one. When this option is chosen as invalid the evaluation of the argument is marked as valid (since it is equivalent to indicating inadmissible in the previous case) and the tool presents the different variables in the context to allow the user to select the erroneous one. Once this has been pointed out, the tool selects as current node the one in charge of the evaluation of that variable.

  The argument value, that is, the value used to select the branch and continue the evaluation. If this value is erroneous then the case node is marked as valid and the node in charge of evaluating the argument (the evaluation of \( c_{\text{arg}}(r_b) \)) is set as current one.

  The successful branch, that is, the branch selected to continue with the evaluation. When this value is wrong the tool marks the current node as invalid and asks for the index of the expected branch. Once this index is given, we distinguish whether it is lower or greater than the current one\(^5\). If it is lower than the current one it means that the program should take a branch that was skipped, so the tool marks the ith node as erroneous and all the previous ones as valid. If it is greater than the current one it means that the successful branch should fail, and hence the debugger marks all the failure nodes as valid and the success one as invalid. In both cases, the debugging continues by asking about the correctness of the guard of the branch marked as invalid (if it has no guard then it is directly pointed out as buggy).

  The generated bindings, that is, the bindings generated by matching the argument value with the pattern in the successful branch. When this binding is wrong both the case node and the one standing for the successful branch are marked as invalid, and an error in the pattern is reported.

  The final value, that is, the value reached by evaluating the whole expression. When this value is wrong the case expression is marked as invalid, all the nodes related to branches are marked as valid, and the node in charge of continuing the evaluation (that is, \( c_{\text{result}}(r_b) \)) is marked as invalid.

• Since if and try/catch expressions are specific instances of case expressions a customized version of the questions above is presented.

Note that they cannot be equal, because in that case it would not be wrong.
• The debugger checks that the function in the root of the tree is being executed using the appropriate clause. Hence, it will present the context (which is just the values bound to the parameters in this case) and ask whether it is correct to discard the clauses previous to the one being used and to select the current one. All the answers available for wrong answers can be used here, presenting the same effects in the tree.

• The debugger can ask about the guards in the branches of case expressions or clause. All the answers are also available here.

• Finally, the tool presents the value assigned to a variable or a group of variables. In this case the context consists in the value of all the variables directly involved in the evaluation of the expression assigned to the variables. All the answers presented for functions are allowed for this question.

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, so we have not considered it in our proofs; 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.

The system can internally utilize two different navigation strategies [38, 39], 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 present more questions to the user, but they are presented in a logical order, while Divide & Query leads to shorter sessions of unrelated questions.

The tool can identify different kind of errors. When the first step of the debugging session finishes, the error is shown with the following scheme:

Call to a function that contains an error:

module_name:function_name(Args) = Val
or
fun Clause_1 ... Clause_n end(Args) = Val
fun location: (File, line Line)
and
Please, revise the ith clause:

function_name(Pars) -> Body. (ith clause definition)
or
fun Clause_i end
or
There is no clause matching.

If there are nodes marked as unknown at the end of the debugging session and the user chooses to not give an answer for them, the debugger will show the candidates to be erroneous in two different groups. First, those nodes whose children are correct and at least one of them is unknown. The error messages for this group follow the previous scheme, except that the sentence “Call to a function that contains an error” is used instead of “Call to a function that could contain an error.” The second group is formed by the unknown descendants of the previous group. In this case the first sentence is “This call has not been answered and could contain an error.” This last group of nodes is the one used for asking questions when the user selects to answer the missing information.

The errors shown after the second step of the debugging session, the zoom debugger, follow the scheme:

This is the reason for the error

Variable Var is badly assigned Val in the expression:

Expr (line Line).
or

The following variables are badly assigned:

Var_1 = Val_1
...
Var_n = Val_n
in the expression:

Expr (line Line).
or

Argument value Val of the case/if/catch of try/try expression:

Expr
is not correct.
or

Value Val for the final expression Expr (line Line) is not correct.
The pattern/guard of the ith clause of case/if/catch of try/try
expression:
Expr
or
The pattern/guard of the ith clause of function definition:
function_name(Param) -> Body. (ith clause definition)

In case that there is not enough information to determine the reason of the error, the debugger will
proceed in the same way as described above. The difference is that the message shown for the candidates
are “This could be a reasons for an error” for the first group, and “This could be a reason for an error if
an unanswered question were answered with no” for the second one. Now we will explain two different
debugging sessions using the tool.

7.1 Sample session debugging wrong functions

We first use the edd tool to find the bug in the Vigenère cipher program of Figure 1 (page 5). As have
we seen in Example 3.1 (page 4), the evaluation returns an unexpected exception:

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

To start a debugging session using the default Divide & Query navigation strategy we have to call
edd with the starting expression:

```erlang
> edd:dd("vigenere:encrypt(\"Attack tomorrow dawn\", \"lemon\")").
```

Please, insert a list of trusted functions [m1:f1/a1, m2:f2/a2 ...]:

- vigenere:normalize("Attack tomorrow dawn") = "ATTACKTOMORROWDAWN"?
  - [y/n/t/d/i/s/u/a]: y
- vigenere:normalize("lemon") = "LEMON"?
  - [y/n/t/d/i/s/u/a]: y

Notice that we have to escape the quotations marks that appear in the string representation of the
expression to debug. The first thing we can do is inserting a list of trusted functions for the debugger, but
we do not add anything and simply press Enter. Then the debugger asks the first two questions about
the computation, which are related to applications of vigenere:normalize. In both cases the results
of the normalization are the expected ones, so we answer yes. The next question is about the encipher
function:

```erlang
vigenere:encipher(82, 69) = 86?
```

- [y/n/t/d/i/s/u/a]: t

In this case the result is correct—encipher 82 (R) using 69 (E) results in 86 (V)—but, since we consider
the function simple enough, we also tell the debugger to trust the vigenere:encipher, so it will not ask
future questions about this function because it will treat them as correct automatically. Finally, the last
two questions of the debugger are related to the vigenere:cycle_to function:

```erlang
vigenere:cycle_to(13, "LEMON") = "LEMONLEMONLEMON"?
```

- [y/n/t/d/i/s/u/a]: n
- vigenere:cycle_to(3, "LEMON") = "LEMON"?
  - [y/n/t/d/i/s/u/a]: n

The call to vigenere:cycle_to(13, "LEMON") should return the string of length 13 that results from
the concatenation of the string "LEMON". In this case we expect "LEMONLEMONLEMON", so the result in
the first question is not correct and we answer no. For the same reason the result in the second question is
not correct (we expect "LEM"), so we answer again no. After this question the debugger finds a buggy
node and shows the following message:

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

The debugger marks vigenere:cycle_to as the buggy function, and in this case it is easy to see
where the problem is: it does not trim the list when N is less than the length of the list but returns
the whole list. Thanks to the simplicity of the cycle_to function we have spotted easily the bug from
the information of the wrong function, so we do not need to continue with a zoom debugging session to
1 test() -> stock:check_orders(
2 [[{item, water, 3}], [{item, rice, 3}]], [{item, rice, 4}]),
3 [{item, rice, 5}, {item, bajoqueta, 8}])
4).

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

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

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

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

Figure 6: Erlang program for managing stocks

locate more precisely the bug. To solve this erroneous behavior, we have to replace the first clause of the vigenere:cycle_to (line 6 of Figure 1, page 5) by this one, which simply returns the first N elements of the List:

cycle_to(N, List) when length(List) >= N -> lists:sublist(List,N);

Note that the abbreviated proof tree has 197 nodes, but after 5 questions the debugger is able to point the error. Notice that the program in Figure 1 is a program directly downloaded from Rosetta Code, and edd has helped to detect the bug involving unexpected exceptions with very few questions.

7.2 Sample debugging session with zoom

To show a zoom debugging session we will use an Erlang program for managing stocks, so that a company knows what items has to buy to supply a list of orders. The program, explained in Example 7.1, fits better with zoom debugging because functions have a more complex structure, so detecting the bug from the wrong function is not as easy as in the Vigenère program.

Example 7.1. Figure 6 presents an Erlang program defining a module stock to manage stocks of items. It only exports the function check_orders/2 which, given a list of orders and a stock of items, returns the list of orders the company has to buy to supply all the orders. Both orders and stocks are represented as lists of tuples \{item, Name, Quantity\}, where Name is the name of the item and Quantity the number of items of that kind (it is supposed that there are not two tuples for the same Name in the lists). For handling a list of orders, we simply unify the orders (add all the entries with the same name in one item tuple), and call check_order/2, which behaves the same as check_orders/2 but considering only one order. check_order/2 uses the predefined lists:zf/2 function, that performs a filter and a mapping at the same time. lists:zf/2 takes a unary function F and a list L, and for each element E of L:

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

zf/2 is currently an undocumented and obscure function of the lists module, but there is a proposal for renaming and documenting it [http://erlang.org/pipermail/erlang-patches/2013-April/003907.html](http://erlang.org/pipermail/erlang-patches/2013-April/003907.html)
The program uses check_item/2 to filter and map the order into the list of items to buy. Given an item tuple and a stock, check_item/2 looks for a tuple with the same name in the stock using lists:keyfind/3 function. If there is not such a tuple in the stock, all the elements must be bought. If the tuple exists in the stock but its quantity is less than the required quantity, we must buy the difference. If the tuple appears in the stock and there is enough items, the item tuple can be discarded because we do not need to buy anything.

Finally, test/0 is a function to check that the program is working. It calls check_orders/2 with some orders of ingredients for a paella ({{item, water, 3}, {item, rice, 3}} and {{item, rice, 4}}), considering that the stock is {{item, rice, 5}, {item, bajoqueta, 8}}. Since the orders need 7 packs of rice and 3 bottles of water but the stock only contains 5 packs of rice and no water, the expected result should be {{item, water, 3}, {item, rice, 2}}. However, the program returns the buggy value {{item,water,3},{item,rice,7}}.

To debug the program, we will first use edd—this time using the Top Down Heaviest First strategy—to locate the wrong function. The debugging session is:

> edd:dd( "stock:test()", top_down ).

Please, insert a list of trusted functions [m1:f1/a1, m2:f2/a2 ...]:

stock:check_orders([[item,water,3], {item,rice,3}], [[item,rice,4]]),
[[item,rice,5], {item,bajoqueta,8}]) = [[item,water,3], {item,rice,7}]?

[y/n/d/i/s/u/a]: n

stock:unify_orders([[item,water,3], {item,rice,3}, {item,rice,4}]) = [[item,water,3], {item,rice,7}]?

[y/n/d/i/s/u/a]: y

stock:check_order([[item,water,3], {item,rice,7}], [[item,rice,5], {item,bajoqueta,8}]) = [[item,water,3], {item,rice,7}]?

[y/n/d/i/s/u/a]: n

fun (X) -> check_item(X, [[item,rice,5], {item,bajoqueta,8}]) end
({item,rice,7}) = {true, {item,rice,7}}

fun location: (stock.erl, line 6)?

[y/n/d/i/s/u/a]: n

stock:check_item({item,rice,7}, [[item,rice,5], {item,bajoqueta,8}]) = {true, {item,rice,7}}?

[y/n/d/i/s/u/a]: n

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

Please, revise the first clause:
check_item(Needed = {item, Name, Q1}, Stock) ->
ItemStock = lists:keyfind(Name, 1, Stock),
case ItemStock of
  {item, Name, Q2} ->
    if Q1 > Q2 -> {true, {item, Name, Q1 - Q2}};
true -> false
end;
false -> {true, Needed}
end.

In this case the abbreviated proof tree has 14 nodes, and the debugger finds the buggy function with 5 questions using the Top Down Heaviest First (the Divide & Query strategy also needs 5 questions for this expression). More specifically, Figure 7 partially depicts the debugging tree for this evaluation, where all the labels are (BFUN). cos is the subindex for the function check_orders, co for check_order, wo for unify_orders, ci for check_item, res stands for [{{item, water, 3}, {item, rice, 3}}, and res’ for [true, {item, water, 3}], the substitutions θ₀ and θ₁ basically transmit the initial requirements explained above, θ₂ takes the result of the unification and the initial stock, and θ₃ and θ₄ carry the values for checking the required amounts of water. The numbers in the nodes indicate the order of the questions made by the debugger. Note that the strategy traverses the tree asking questions about the children of the current node, discarding the valid ones (2) until node (5) is pointed out as invalid. Since it is a leaf, it is the buggy node and the session finishes.

The spotted function check_item/2 is moderately complex (it has nested case expressions) so it is a good situation to use the help of the zoom version to find the origin of the bug. Therefore we press the key y to continue with the zoom debugging session inside the function:

Do you want to continue the debugging session inside this function?
[y/n]: y

Bajoqueta is the Valencian word for a kind of green bean essential in the paella valenciana.
For the case expression:

```erlang
case ItemStock of
    {item, Name, Q2} ->
        if Q1 > Q2 -> {true, {item, Name, Q1 - Q2}};
        true -> false
        end;
    false -> {true, Needed}
end
Is there anything incorrect?
1.- The context:
   ItemStock = false
   Needed = {item,rice,7}
2.- The argument value: false.
3.- Enter in the second clause.
4.- The final value: {true,{item,rice,7}}.
5.- Nothing.
```

Given the context:

```erlang
Name = rice
Stock = [{item,rice,5},{item,bajoqueta,8}],
the following variable is assigned:
ItemStock = false? [y/n/d/i/s/u/a]: n
```

This is the reason for the error:

Variable ItemStock is badly assigned false in the expression:

```erlang
ItemStock = lists:keyfind(Name, 1, Stock) (Line 8).
```

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

The tree for this debugging session is partially depicted in Figure 7. The root of the tree is the buggy node detected in the previous session, which has the substitution $\theta_4 \equiv \{\text{Needed} \mapsto \{\text{item}, \text{rice}, 7\}; \text{Stock} \mapsto \{\text{item}, \text{rice}, 5\}, \{\text{item}, \text{bajoqueta}, 8\}\}$. The first children of the root is in charge of proving that the selected clause is the correct one, the second one binds the variable ItemStock (identified by its reference, $r_{IS}$) to false when looking for rice in the stock, and the third one evaluates the case expression. As explained above, the first node selected by the tool is the one marked as (1) in the tree, since the Top Down strategy is implemented with the heaviest-first strategy, which selects first the biggest sibling. The answer yes discards this node, and then the strategy selects the node marked with (2). It is pointed out as buggy node because it is a leaf and the user indicated that it is invalid.

8 Experimental results

In this section we measure the effectiveness of the Erlang Declarative Debugger. We use a set of small and medium-size sequential Erlang programs that solve standard problems: Miller-Rabin primality test,
Then, we have used edd to find the wrong function and also the zoom debugger to spot the bug more precisely. In some cases (ackermann, dutch, and rfib) the zoom debugger was not applicable because the wrong function was very simple, so the bug was easily found by inspecting the code. For all the tests, edd has spotted the bug added to the programs, and the time consumed to generate the abbreviated proof trees has been reasonable—no more than 10 seconds in a standard desktop computer. The source code of the programs as well as the abbreviated proof trees can be found in the folder examples at https://github.com/tamarit/edd. Figure 9 shows the results of the debugger for the test programs. For each one it shows the lines of code (LOC), the number of nodes of the abbreviated proof tree (Nodes), and the number of questions using the Divide & Query (D&Q) and Top Down Heaviest First (TD) strategies for each step of the debugger (wrong functions and zoom). Regarding the detection of wrong functions, the debugger finds them using a very small number of questions compared to the number of nodes in the abbreviated proof tree, i.e., the number of function and lambda-abstractions applications in the computation. Although for some programs (24_game, caesar, dutch) TD performs slightly better than the D&Q strategy, the results show that D&Q is in general a better option for debugging wrong functions because it needs less questions for finding the bug, being the extreme cases the programs rle (5 vs. 19) and sieve (6 vs. 13). Regarding bug detection using zoom inside functions, the number of questions is also small compared to the number of nodes in the abbreviated proof tree with the exception of roman, where the debugger asks almost all the questions to find the bug. However, the number of asked questions is proportionally bigger than the number ask in the first step. This is due to that each question is actually several questions put together. For zoom debugging, the performance of both strategies is almost equal, with only a ±1 difference in complex and ternary and ±2 difference in turing.

Considering the good results of edd for the set of standard programs, we think it can be an interesting tool for programmers when debugging the sequential part of Erlang programs, supplementing the standard trace-debugger of the OTP/Erlang system. The results also support our choice of standard strategies: D&Q for finding wrong functions, as it usually asks less questions than TD, and TD for finding bugs inside functions, since it is as good as D&Q and it shows the questions in an order closer to the evaluation one, which users might find clearer.

---

*S*The bug shown in Section was already present in the original code.
9 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 using Core Erlang to perform the analysis. A straightforward benefit is that it allows us 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 roles is an advantage during the development of any software project.

Although most of the applications based on Erlang employ the concurrent features of the language the concurrency is usually located in specific modules, and thus specific tools for debugging the sequential part are also interesting. Our debugger locates the error based only on the intended meaning of the program code, and thus abstracts away the implementation details. The main limitation of the proposal is that an initial unexpected result must be detected by the user, which implies in particular that it cannot be used to debug non-terminating computations. From the point of view of declarative debugging, this paper presents a new idea allowing the user to vary the granularity of the bug detected by the tool by asking more specific questions. This is achieved by building a new tree from the buggy node discovered in the first phase, but using the same calculus.

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.

An natural line of future work consists in extending the current framework to debug concurrent Erlang programs. This extension will require new rules in the calculus to deal with functions for creating new processes and sending and receiving messages, as well as the identification of new kinds of errors that the debugger can detect.

References


decode(Segment) ->
case Segment of
  « SourcePort:16, DestinationPort:16,
  SequenceNumber:32, AckNumber:32,
  DataOffset:4, _Reserved:4, Flags:8, WindowSize:16,
  Checksum:16, UrgentPointer:16,
  Payload/binary » when DataOffset>4 ->
  OptSize = (DataOffset - 5)*32,
  « Options:OptSize, Message/binary » = Payload,
  «CWR:1, ECE:1, URG:1, ACK:1, PSH:1, RST:1,
  SYN:1, FIN:1 » = «Flags:8»,
  % Can now process the Message according to the
  % Options (if any) and the flags CWR, ..., FIN.
  _ -> {error, bad_segment}
end.

Figure 10: Bit Syntax Example: Decoding TCP Segments [12]

A Bit String syntax

As mentioned in Section 3, we have omitted the bit syntax from the presentation of the paper to keep clear the main ideas and because this feature is not supported in the metaprogramming library for Core Erlang we use in the implementation of the debugger. However, it can be included in the debugging setting by adding slight modification in the syntax level and the matching process. It is important to remark that the modifications at the semantic level (further than the matching definition), the debugging tree or the proofs are not necessary. In this appendix we will show the expressive power of bit syntax and the modifications needed to include it into our debugging setting.

Erlang supports a data type representing chunks of raw and untyped data called binaries. This data type is mainly used in socket-based communication applications, where segments—aka. packets or datagrams—are represented as binaries that are sent through the network. These chunks of bits are usually cumbersome to parse, but Erlang provides the bit syntax to easily parse the different fields by matching. Figure 10 shows an example (extracted from [12]) of decoding a TCP segment using bit syntax.

The decode/1 function matches the binary Segment against a binary pattern, where each field has a size option that controls the number of bits to take. In this case the binary patterns takes the first 16 bits as an integer in SourcePort, the second 16 bits in DestinationPort, the next 32 bits in SequenceNumber, and so on. In this way, the parsing is done at once and the subsequent code can use the integer variables that appear in the pattern. Although not used in this example, bit syntax supports also type options that specify the way that binary matching must be done: destination type (integer, float, byte), sign (signed or unsigned), endian value (big-endian or little-endian), etc.

To include bit syntax in our debugging setting, we need to first extend the syntax of Core Erlang considered. Figure 11 shows the modifications needed. First, we extend the literals lit with BitStrings, which are lists of bits to represent binaries. We also have to add the new categories to represent patterns over binaries (bitpat and pat_b) and expressions that will evaluate to binaries (bitexpr). Finally, we have to extend patterns and expressions to include these new categories.

The opts argument in bit patterns and expressions is a tuple of encoding options that is system dependent. It is important to notice that opts can contain variables to be bound during evaluation. This can be seen in the line 9 of Figure 10 where the size option OptSize is bound in the previous instruction. Unlike the rest of patterns, bit patterns are not linear, so this variable size options can also be bound previously.
in the same pattern. For example, \(<\text{Origin}:8, \text{Destination}:8, \text{Length}:8, \text{Message}:\text{Length}\rangle \) is a valid bit pattern. The non-linearity of bit patterns must be handled carefully when matching, as we will see shortly.

Since encoding options are system dependent, we will assume two functions to convert values to bit strings and vice versa that will be used when matching:

- \(\text{to\_bits}(\text{val}, \text{opts})\), which given an integer or float value \(\text{val}\) and some encoding options \(\text{opts}\) returns the bit string that represents \(\text{val}\). For example, \(\text{to\_bits}(127, \{8,1,\text{integer, unsigned, big}\})\) will be evaluated to the bit string "01111111", the binary value of the unsigned integer 127 using 8 bits and big-endian.

- \(\text{from\_bits}(\text{bits}, \text{opts})\), which given a bit string \(\text{bits}\) and some encoding options \(\text{opts}\), returns a pair \((\text{val}, \text{bits'})\) where \(\text{val}\) is the value represented in the first bits of \(\text{bits}\) (according to the encoding options \(\text{opts}\)) and \(\text{bits'}\) is the rest of the bit string. For example, \(\text{from\_bits}("0111111100000011", \{8,1,\text{integer, unsigned, big}\})\) is \((127, "00000011")\), where 127 is the result of interpreting the first 8 bits of the bit string as an unsigned integer with big-endian, and "0000011" is the rest of the input bit string.

Using the previously introduced converting function, we need to add a new rule to the syntactic matching function \(\text{synMatch}\) to match bit patterns and bit strings and also a new bit-matching function \(\text{synMatch}_b\):

\[
\text{synMatch}(\#\{\text{bitpat}_1, \ldots, \text{bitpat}_n\}#, \text{BitString}) = \theta_1 \sqcup \ldots \sqcup \theta_n, \quad \text{where}
\]
\[
(\theta_1, \text{BitString}_1) \equiv \text{synMatch}_b(\text{bitpat}_1, \text{BitString})
\]
\[
(\theta_2, \text{BitString}_2) \equiv \text{synMatch}_b(\text{bitpat}_2\theta_1, \text{BitString}_1)
\]
\[
\ldots
\]
\[
(\theta_n, \epsilon) \equiv \text{synMatch}_b(\text{bitpat}_n\theta_1, \ldots \theta_{n-1}, \text{BitString}_{n-1})
\]

\[
\text{synMatch}_b(\#\text{var }\times(\text{opts }), \text{BitString}) = ([\text{var }\mapsto \text{val}], \text{BitString'}), \quad \text{if}
\]
\[
\text{from\_bits}(\text{BitString}, \text{opts}) = (\text{val}, \text{BitString'})
\]

The new rule for \(\text{synMatch}\) simply composes the matching substitutions computed for the \(\text{bitpats}\). Since bit patterns are not linear, it is important to apply the matching substitution obtained so far \((\theta_1\theta_2\ldots\theta_i)\) to the next \(\text{bitpat}_{i+1}\), as some encoding options in \(\text{bitpat}_{i+1}\) can be bound to a value in the previous matchings. The last call to \(\text{synMatch}_b\) returns the empty bit string \(\epsilon\) to express that after the complete matching all the bit string must be “consumed”. On the other hand, \(\text{synMatch}_b\) uses the \(\text{from\_bits}\) function to find the bindings for variables in bit patterns (first rule) and to check that values in bit patterns are the same as the ones represented in the bit strings (second rule). In both cases \(\text{synMatch}_b\) returns a pair with the value parsed and the remaining bit string.

Finally, we need to add a new semantic rule in the calculus for sequential Erlang presented in Section 4 to evaluate bit expressions to values. This rule is pretty straightforward, since it only evaluates the inner expressions to values and then concatenates all their bit representations, obtained using the function \(\text{to\_bits}\):

\[
\frac{\langle \text{expr}_1, \theta \rangle \rightarrow \text{vals}_1 \ldots \langle \text{expr}_n, \theta \rangle \rightarrow \text{vals}_n}{\langle \#(\text{bitexpr}_1, \ldots, \text{bitexpr}_n), \theta \rangle \rightarrow \text{B}_1++\text{B}_2++\ldots++\text{B}_n}
\]

where \(\text{bitexpr}_i = \#< \text{expr}_i >(\text{opts}_i)\), \(\text{to\_bits}(\text{vals}_1, \text{opts}_1) = \text{B}_i\) and \(\text{vals}_i\) are integer or float values. Notice that the \(\text{(VAL\_BITS)}\) rule does not depend on the user code, so any proof tree obtained with the CESC calculus will be also obtained using the ICESC calculus. Therefore, this new rule does not affect the soundness results of the declarative debugger.
B Proofs of the main results

This Appendix includes the proofs of the theoretical results contained in the report.

B.1 Properties of $ICESC_\diamond$

The following lemmas are basic and straightforward properties of the calculi $ICESC_\diamond$ which are used in the main theorems.

**Lemma B.1.** Let $P$ be a Core Erlang program and $E$ an evaluation. Let $T$ be a proof tree verifying $CESC \models_{P,T} E$. Suppose also that the root of $T$ (that is, $E$) is $ICESC_I$-invalid.

Then $T$ contains an $ICESC_I$-invalid node $N \equiv \langle r_f, \theta \rangle \rightarrow$ evals immediate (BFUN)-descendant of the root of $T$.

**Proof Idea**

By reductio ad absurdum. Consider the set of nodes:

$$S = \{N \in T \mid N \text{ is consequence of a (BFUN) inference and there is no (BFUN) in the path from } N \text{ to the root} \}$$

Suppose that every node $N \in S$ is $ICESC_I$-valid. Consider the tree $T'$ obtained after replacing for each $N \in S$ the inference label (BFUN) by (BFUNZ) and removing the premises. Then $ICESC_I \models_{P,T'} E$ and thus $E$ is $ICESC_I$-valid, in contradiction with the hypotheses. □

An analogous Lemma can be applied to $ICESC_Z$:

**Lemma B.2.** Let $P$ be a Core Erlang program and $E$ an evaluation. Let $T$ be a proof tree verifying $CESC \models_{P,T} E$. Suppose also that the root of $T$ (that is, $E$) is $ICESC_Z$-invalid. Suppose also that all the immediate (BFUN)-descendants of the root of $T$ are $ICESC_Z$-valid.

Then, $T$ contains an immediate (R)-descendant $N$ of the root $T$, such that (R) is a zoom inference label and $N$ is $ICESC_Z$-invalid.

The proof idea is analogous to the one for Lemma B.1.

B.2 Correctness and Completeness of $APT$’s

This result is established in Theorem 6.1.

**Theorem 6.1**

Let $P$ be a Core Erlang program, $\mathcal{I}$ its intended interpretation, and $e\theta \rightarrow$ eval an evaluation unexpected with respect to $ICESC_I$. Then, exists at least one proof tree $T'$ such that $CESC \models_{(P,\mathcal{I},T')} \langle \text{main}-id \rangle \rightarrow$ [eval], and every tree $T''$ in these conditions verifies that $T \equiv APT(T')$ contains at least one buggy node $N$, and every buggy node in $T$ corresponds to a wrong instance of a user function different from $\text{main}/0$.

**Proof.**

In this proof all the references to valid, invalid, and buggy are assumed implicitly as with respect to $ICESC_I$.

By Assumption 5.1, there is a proof tree $T''$ such that $CESC \models_{P,T''} \langle |e|, \theta \rangle \rightarrow |\text{eval}|$. From $T''$ it is easy to construct a proof tree $T'''$ for $\langle |e|, \theta, \text{id} \rangle \rightarrow |\text{eval}|$. Then, a tree $T'$ such that $CESC \models_{(P,\mathcal{I},T')} \langle \text{main}-id \rangle \rightarrow |\text{eval}|$ can be constructed starting by an application of the (BFUN) inference rule for $\text{main}/0$ at the root, followed by a (CASE) inference with the body of $\text{main}/0$. The $\text{c_result}$ premise of the (CASE) has as premise the proof of $\langle |e|, \text{id} \rangle \rightarrow |\text{eval}|$, which is $T'''$. Moreover, any other proof will have the same structure.

Observing the case expression defining $\text{main}/0$ (Definition 5.6) and the form of the (CASE) inference we have that the rest of the premises of this inference cannot contain function calls. Therefore, $T \equiv APT(T')$ will be of the form:

$$APT(T''') \langle \text{main}-id \rangle \rightarrow |\text{eval}|$$

Again by Assumption 5.1 (now by item 2), since $e\theta \rightarrow$ eval is unexpected then $ICESC_I \not\models_{P,I} \langle |e|, \theta \rangle \rightarrow |\text{eval}|$, and consequently $ICESC_I \not\models_{P,I} \langle |e|, \text{id} \rangle \rightarrow |\text{eval}|$. Thus, the root of $T'''$ is invalid. Applying the

\[^9\text{In the case of (BFUN) nodes $ICESC_Z$-validity corresponds to $ICESC_I$-validity} \]
Lemma B.3.1 there is a node $N \in T''$ of the form $N \equiv \langle r_f, \theta \rangle \to evals$, consequence of a (BFUN) inference, such that $N$ invalid and that verifies that there is no other (BFUN) inference in the path from the root to $N$. This implies that $N$ will be the root of one of the subtrees returned by $APT(T'')$, that is that $N$ is an invalid child of the root of $T$, $\langle r_{main}, \text{id} \rangle \to \text{eval}$, and thus the only reference to $\text{main}$ cannot be buggy. Since every proof tree with an invalid root contains a buggy node $[27]$, this means that there is a buggy node $M \in T$ with a reference to some function different from $\text{main}$, $M \equiv \langle r_f, \theta \rangle \to evals$. To complete the theorem we must prove that $M$ corresponds to a wrong function instance. This is done by considering $M$ again in the $\text{CESC}$ proof tree $T'$. There, the only child of $M$ corresponds to $\theta$ with $B$ the case expression defining the body of function $f$. Then we have:

1. $\text{ICESC}_f \models_{(P, Z)} \langle \theta \rangle \to evals$ holds by Lemma B.3.1 because all its (BFUN) immediate descendants are valid, since they are children of $M$ in $T \equiv APT(T)$, and $M$ is buggy in $T$.

2. $\text{ICESC}_f \not\models_{(P, Z)} \langle r_f, \theta \rangle \to evals$ because $M \equiv \langle r_f, \theta \rangle \to evals$ is buggy.

Thus, according to Definition 5.5.4 $\langle r_f, \theta \rangle$ is a wrong function instance. 

\section{B.3 Correctness and Completeness of APTZ's}

The following Lemma shows the basic form of an APTZ.

\textbf{Lemma B.3.} Let $P, T, T'$ be as in Definition 6.2. Then the APTZ($T$) verifies:

1. It is a single tree rooted by the conclusion of a (CASE) inference.

2. If the root of $T$ corresponds to a wrong function instance then the root of APTZ($T$) is $\text{ICESC}_Z$-invalid.

3. It contains only nodes corresponding to Erlang expressions occurring in the body of the function found at the root of $T$.

\textbf{Proof.}

Observe that $T$ must be of the form:

\[
\begin{array}{c}
\text{(CASE)} \quad \ldots \quad \langle c_{\text{result}}(r_f), \theta'' \rangle \to evals \\
\text{(BFUN)} \quad \ldots \\
\langle \text{case}^{c_1}, \ldots, \theta \rangle \to evals \\
\langle r_f, \theta \rangle \to evals \\
\end{array}
\]

The root is the conclusion of (BFUN) inference by hypothesis. This inference only has one premise that corresponds to the conclusion of a (CASE) inference (Section 3 indicates that the body of a function is always a case expression). Among the premises of the (CASE) inference we make explicit the last one, which corresponds to the result returned by the successful branch $r_i$. Then, applying the rules of APTZ:

\[
\begin{array}{c}
\text{APTZ}(T) = \\
\text{APTZ} \left( \begin{array}{c}
\text{(CASE)} \\
\text{(BFUN)} \\
\langle \text{case}^{c_1}, \ldots, \theta \rangle \to evals \\
\langle r_f, \theta \rangle \to evals \\
\end{array} \right) = \text{APTZ}_{\theta}
\end{array}
\]

\[
\begin{array}{c}
\text{APTZ} \left( \begin{array}{c}
\text{(CASE)} \\
\langle \text{case}^{c_1}, \ldots, \theta \rangle \to evals \\
\langle r_f, \theta \rangle \to evals \\
\end{array} \right) = \text{APTZ}_{\theta}
\end{array}
\]

\[
\begin{array}{c}
\ldots \text{APTZ} \left( \begin{array}{c}
\text{(CASE)} \\
\langle \text{case}^{c_1}, \ldots, \theta \rangle \to evals \\
\langle r_f, \theta \rangle \to evals \\
\end{array} \right) = \text{APTZ}_{\theta}
\end{array}
\]

Then:

36
1. Obviously APTZ(T) is rooted by the conclusion of a (CASE) inference.

2. If root of T corresponds to a wrong function instance of a user function \( f \) (analogous for a lambda function), then by Definition 6.2

\[
ICESC_f \not\vdash_{(P,T)} (r_f, \theta) \rightarrow v
\]

In particular, since (BFUN\(_T\)) is the only ICESC\(_f\) inference that can be applied it means that \( \langle r_f, \theta \rangle \rightarrow v \notin T_{\text{fun}} \) (condition of the (BFUN\(_T\)) inference). Then, applying Property 5.1 there is not substitution \( \theta'' \) extending \( \theta \) such that \( \langle c\_\text{result}(r_i), \theta'' \rangle \rightarrow v \) with \( r_i \) a reference to some branch of the case expression defining the body of \( f \). By the (C\_RESULT\(_T\)) inference rule this means that

\[
ICESC_Z \not\vdash_{(P,T)} (c\_\text{result}(r_i), \theta'') \rightarrow v
\]

Thus, at least one of the premises of the (CASE\(_Z\)) inference cannot be proven in ICESC\(_Z\). Consequently, the conclusion of this inference, which corresponds to the root of APTZ(T), is invalid.

3. This is straightforward from the definition of the APTZ(T), since all the calls to other functions are removed including their subtrees using the second rule.

\[\square\]

The following two Lemmas indicate that the validity of certain nodes is related to the validity of their children:

**Lemma B.4.** Let \( T \) be an APTZ obtained from a CESC proof tree \( T' \) as explained in Definition 6.2. Then, the nodes (CASE)-nodes are ICESC\(_Z\)-valid iff all their children in \( T \) are ICESC\(_Z\)-valid.

**Proof.** The provability of a node with respect to some calculus depends on the validity of its premises in the same calculus, as well as in fulfilling the associated side-conditions. In the case of the nodes of this lemma, observe that:

1. By the definition of APTZ (CASE) nodes have in \( T \) the same children (premises) as in \( T' \).

2. The definition of the (CASE) inference rule is the same in CESC and in ICESC\(_Z\).

Thus the conclusions of (CASE) inferences are ICESC\(_Z\)-valid iff all their children in \( T \) are ICESC\(_Z\)-valid.

\[\square\]

**Lemma B.5.** Let \( T \) be an APTZ obtained from a CESC proof tree \( T' \) as explained in Definition 6.2. Let \( N \) be either a (FAIL1), (FAIL2), or (SUCC)-node, and suppose that \( N \) is ICESC\(_Z\)-invalid. Then \( N \) has an ICESC\(_Z\)-invalid child in \( T' \).

**Proof.**

We prove the result for (SUCC), it is analogous for (FAIL1), (FAIL2). Since \( N \) is a (SUCC)-node, it must be of the form \( N \equiv \langle \text{succeeds}(\text{vals}, r_i), \theta \rangle \rightarrow \theta' \). First observe that in the APTZ each (SUCC)-node contains the same premises as in \( T' \), which must be of the form: \( \langle \text{patbind}(r_i, \text{vals}), \theta \rangle \rightarrow \theta' \) (with \( \theta' \neq \bot \)), and \( \langle \text{guard}(r_i), \theta'' \rangle \rightarrow \text{'true'} \). The validity of this premises depend on its membership to \( \langle I_{r_i}, \theta \rangle \) due to the definition of (PATBIND\(_Z\)) and (GUARD\(_Z\)).

\( N \) is ICESC\(_Z\)-invalid, that is ICESC\(_Z\) \( \not\vdash_{(P,Z)} \langle \text{succeeds}(\text{vals}, r_i), \theta \rangle \rightarrow \theta' \), and by the definition of (SUCC\(_Z\)) in Section 5.2 it means \( \langle \text{succeeds}(r_i), \theta \rangle \notin \langle I_{r_i}, \theta \rangle \). The construction of \( \langle I_{r_i}, \theta \rangle \) indicates that the premises are not associated to the only succeeds evaluation in \( \langle I_{r_i}, \theta \rangle \), that is, \( i \) is not the expected successful branch of the case expression in the context \( \theta \). There are two possibilities:

1. The successful branch is some \( j < i \). Then \( \langle I_{r_i}, \theta \rangle \) does not contain information about \( i \) (we do not require \( \langle I_{r_i}, \theta \rangle \) to include information about the branches after the successful branch because this leads to very difficult questions to the user and can be avoided). Then, both premises are ICESC\(_Z\)-invalid.

2. The successful branch is some \( j > i \). Then \( i \) is a failing branch, and then \( \langle I_{r_i}, \theta \rangle \) includes evaluations \( \langle \text{patbind}(r_i, \text{vals}), \theta \rangle \rightarrow \theta', \langle \text{guard}(r_i), \theta'' \rangle \rightarrow \text{val} \), with \( \theta'' \equiv \theta \uplus \theta' \). But the conditions of failing branches require that \( \theta_j \equiv \bot \) or \( \text{val} \equiv \text{'false'} \), and therefore at least one of the premises cannot be proven in ICESC\(_Z\).
Now we can establish the correctness and completeness of APTZ trees as declarative debugging trees.

**Theorem 6.2**

Let $P'$ be a Core Erlang program, $I$ its intended interpretation, and $\theta' \mapsto \text{eval}$ an unexpected evaluation. Let $T'$ be a proof tree such that $\text{CESC} \models_{\langle P', \theta', T' \rangle} \langle r_{\text{main}}, \text{id} \rangle \mapsto \text{eval}$. Let $T''$ be defined as $T'' \equiv \text{APT}(T')$. $M$ is an $\text{ICESC}_I$ buggy node in $T''_I$, and $T''_M$ the subtree of $T'$ rooted by $M$. Finally, define a new tree $T$ as $T \equiv \text{APTZ}(T''_M)$. Then $T$ contains at least one $\text{ICESC}_Z$-buggy node $N$ that verifies one of the following items:

1. $N$ is a (BIND) node of the form
   \[
   \langle \langle r_1, \ldots, r_n \rangle, \text{exprs}, \theta \rangle \mapsto \{\text{var}_1 \mapsto \text{val}_1, \ldots, \text{var}_n \mapsto \text{val}_n\}
   \]
   Then, $\langle \langle r_1, \ldots, r_n \rangle, \text{exprs}, \theta \rangle$ is a wrong binding instance.

2. $N$ a (C_ARG) node, $N \equiv \langle \text{c_arg}(r_c), \theta \rangle \mapsto \text{evals}$. Then, $\langle \text{c_arg}(r_c), \theta \rangle$ is a wrong case argument instance.

3. $N$ is a (PATBIND)/(GUARD) premise of a (FAIL1)/(FAIL2) node $P$, $P \equiv \langle \text{fails}(\text{vals}, r_1), \theta \rangle$, with $r_1$ a reference to the $i$th branch of a case expression $r_c$, and such that $N$ is a wrong case branch witness. Then $\langle r_1, \theta \rangle$ is an unexpected case failure instance, and in particular:
   (a) If $N$ is (PATBIND) node then $\langle r_1, \theta \rangle$ is a wrong failure pattern instance.
   (b) If $N$ is a (GUARD) node, then $\langle r_1, \theta \rangle$ is a wrong failure guard instance.

4. $N$ is a (PATBIND)/(GUARD) premise of a (SUCC) node $P$, with $P \equiv \langle \text{succeeds}(\text{vals}, r_1), \theta \rangle$, $r_1$ a reference to the $i$th branch of a case expression $r_c$, and such that $N$ is an wrong case branch witness. Then $\langle r_1, \theta \rangle$ is an unexpected case success instance, and in particular:
   (a) If $N$ is a (PATBIND) node, then $\langle r_1, \theta \rangle$ is a wrong success pattern instance.
   (b) If $N$ is a (GUARD) node, then $\langle r_1, \theta \rangle$ is a wrong success guard instance.

5. $N$ is a (C_RESULT) node of the form the $\langle \text{c_result}(r_c), \theta \rangle \mapsto \text{evals}$, and there is neither a (C_ARG) buggy node nor a wrong case branch witness for the same case expression. Then, $\langle \text{c_result}(r_c), \theta \rangle \mapsto \text{evals}$ is a wrong case result instance.

**Proof.** In this proof the concepts of valid, invalid and buggy are assumed to be with respect $\text{ICESC}_Z$ unless the contrary is mentioned explicitly.

First observe that:

1. $M$, the root of $T''_M$, is $\text{ICESC}_I$ invalid (in fact it is $\text{ICESC}_I$ buggy by hypothesis).

2. $M$ is (BFUN) node because $M \in T''$, $T''$ is an APT, and APT's only contain (BFUN) nodes.

Then by Lemma 5.2 $T \equiv \text{APTZ}(T''_M)$ has an invalid root, which means that $T$ contains at least one buggy node such that the path from the root of $T$ to the buggy node only contains invalid nodes [27]. Let $S$ be the set of all buggy nodes, $S \neq \emptyset$.

In principle $N \in S$ could be either a conclusion of any zoom inference (Definition 5.2), or a (CASE) node. However, as a consequence of Lemmas 5.3 and 5.5, the (CASE), (FAIL1), (FAIL2), and (SUCC) nodes in $T$ cannot be buggy. Hence, $S$ only contains (BIND), (PATBIND), (GUARD), (C_ARG), or (C_RESULT) nodes. Now we can select $N$ from $S$ with the following criterium: choose any (BIND), (C_ARG), or (PATBIND)/(GUARD) node that it is an wrong case branch witness. If there are both a (PATBIND) and a (GUARD) buggy nodes in these conditions choose the (PATBIND) node as $N$. A (C_RESULT) node is only chosen if there is neither a (C_ARG) buggy node for the same case expression nor a (PATBIND)/(GUARD) node that it is a wrong case branch witness for the same case expression.

It is easy to check that $S$ always contains at least one node of this form. Then, we proceed distinguishing cases depending on the type of node of $N$:

1. (BIND). Then, $N \equiv \langle \langle r_1, \ldots, r_n \rangle, \text{exprs}, \theta \rangle \mapsto \theta'$, with the substitution $\theta'$ defined as $\{\text{var}_1 \mapsto \text{val}_1, \ldots, \text{var}_n \mapsto \text{val}_n\}$. By the form of (BIND), in $T'$ its only premise is of the form $M' \equiv \langle \text{exprs}, \theta \rangle \mapsto \langle r_1, \ldots, r_n \rangle$. Since $M$ is buggy, all its immediate descendants that are zoom inferences are valid. Considering this fact and the tree rooted by $M'$ in $T'$ we can apply Lemma 5.2 to deduce that $M'$ is valid. Therefore we have:
(a) \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{expr}, \theta \rangle \rightarrow \theta' \) (\( M' \) valid)

(b) \( \text{ICESC}_Z \not\models \langle P, I \rangle \langle r_1, \ldots, r_n, \text{exprs}, \theta \rangle \rightarrow \theta' \) (since \( M \) is buggy, it is invalid)

Then by Definition 5.5.3a \( \langle r_1, \ldots, r_n, \text{exprs}, \theta \rangle \) is a wrong binding instance, which proves the item 1 of the theorem.

2. (C_ARG), that is \( N \equiv \langle c_{\_arg}(r_c), \theta \rangle \rightarrow \text{evals} \).

\( N \) is buggy, that is invalid with valid children. \( N \) invalid means: \( \text{ICESC}_Z \not\models \langle P, I \rangle \langle c_{\_arg}(r_c), \theta \rangle \rightarrow \text{evals} \).

The only child of (C_ARG) in CESC is \( \langle \text{exprs}, \theta \rangle \rightarrow \text{evals} \) with \( \text{exprs} \) the expression defining the case expression argument. Since it is valid: \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{exprs}, \theta \rangle \rightarrow \text{evals} \).

Then, according to Definition 5.5.3a \( \langle c_{\_arg}(r_c), \theta \rangle \) is a wrong case argument instance.

3. (PATBIND)/(GUARD) premise of a (FAIL1)/(FAIL2) node \( P \), with \( P \equiv \langle \text{fails(vals}, r_i \rangle, \theta \rangle \) and \( r_i \) a reference to the \( i \)th branch of a case expression \( r_c \). \( P \) is invalid (the path from \( N \) to the root contains only invalid nodes) and hence \( \text{ICESC}_Z \not\models \langle P, I \rangle \langle \text{fails(vals}, r_i \rangle, \theta \rangle \). Since \( N \) is an wrong case branch witness, \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{succeeds(vals}, r_i \rangle, \theta \rangle \rightarrow \theta' \) for some \( \theta' \). Then:

(a) If \( N \equiv \langle \text{patbind}(r_i, \text{evals}), \theta \rangle \rightarrow \theta'' \) then

- By the definition of (PATBIND) in CESC,
  \[ \hat{\theta}'' \equiv \text{match}(\text{pats}\theta, \text{vals}) \]

  with \( \text{pats} \) the pattern of the \( i \)th branch.

- From the definition of ICESC,
  \[ \text{ICESC}_Z \models \langle P, I \rangle \langle \text{succeeds(vals}, r_i \rangle, \theta \rangle \rightarrow \theta' \]

  implies \( \langle \text{patbind}(r_i, \text{evals}), \theta \rangle \rightarrow \theta''' \) for some \( \theta''' \) such that \( \theta'' \equiv \theta \equiv \theta''' \).

- \( N \) is buggy, and hence it is invalid. This means that \( \theta'' \neq \theta''' \) and \( \text{match}(\text{pats}\theta, \text{vals}) \neq \theta''' \).

Then \( \langle r_i, \theta \rangle \) is a wrong fail pattern instance because it satisfies all the conditions required by Definition 5.3.1a. Moreover, as an easy consequence of the hypothesis that requires \( N \) to be a wrong case branch witness it is straightforward that \( r_i \) is the first wrong branch (Definition 5.3.3b), as required by Definition 5.3.3b.

(b) \( N \equiv \langle \text{guard}(r_i), \theta \rangle \rightarrow \text{eval} \). In this case \( N \) must be a (FAIL2) node. Then:

- \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{guard}(r_i), \theta \rangle \rightarrow \theta'' \) and \( \theta'' \) such that \( \text{match}(\text{pats}\theta, \text{evals}) \equiv \theta'' \), \( \text{pats} \) the pattern of the \( i \)th rule. This is a consequence of the criterium to choose \( N \) if the chosen node \( N \) is a (GUARD) node this means that its (PATBIND) sibling is not buggy. But the inference for (PATBIND) has no premises, and not buggy means in this case valid. Then the computed substitution \( \text{match}(\text{pats}\theta, \text{evals}) \) (required by the side condition of (PATBIND)) is the same as the expected substitution. Moreover, \( \theta'' \equiv \theta \equiv \theta''' \).

- \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{guard}(r_i), \theta \rangle \rightarrow '\text{true}' \) as a consequence of \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{succeeds(vals}, r_i \rangle, \theta \rangle \rightarrow \theta' \) (by the definition of (I_c, \theta)).

- \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{guard}(r_i), \theta \rangle \rightarrow '\text{false}' \) because \( N \) is invalid.

Then \( N \) satisfies all the conditions to prove that \( \langle r_i, \theta \rangle \) is a wrong fail guard instance (Definition 5.3.11) and analogously to the previous item is a first wrong branch (Definition 5.4, as required by Definition 5.3.3b).

4. The case of \( N \) a (PATBIND)/(GUARD) premise of a (SUCCE) node \( P \) is similar to the previous case and we skip the details.

5. (C_RESULT). Then \( N \equiv \langle c_{\_result}(r_i), \theta' \rangle \rightarrow \text{evals} \). By the criterium used to choose \( N \) there is neither a (C_ARG) buggy node nor a an wrong case branch witness for the same case expression. Observe that we only select this buggy node if there are no other buggy node for this case in the conditions of the hypothesis. This means in particular that the succeeds node previous to \( N \) is valid, that is: \( \text{ICESC}_Z \models \langle P, I \rangle \langle \text{succeeds(vals}, r_i \rangle, \theta \rangle \rightarrow \theta' \).
Then, analogously to the \((\text{C\_ARG})\) nodes: \(N\) buggy means

\[
\begin{align*}
\text{ICESC}_Z \not\models_{(P,I)} &\quad \langle c\_\text{result}(r_i), \theta' \rangle \rightarrow \text{evals} \quad \text{\(N\) invalid} \\
\text{ICESC}_Z \models_{(P,I)} &\quad \langle \text{exprs}, \theta', \theta' \rangle \rightarrow \text{evals} \quad \text{\(\text{valid child}\)}
\end{align*}
\]

which corresponds to the definition of \(\langle c\_\text{result}(r_i), \theta \rangle \rightarrow \text{evals}\) is a wrong \textit{case} result instance (Definition 5.5.3c).