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 [26, 41], 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 [38, 39], 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. A presents additional information about *Bit String*, a particular feature of Erlang which is considered in a separated way to simplify the presentation. Finally, 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 [40]. 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],[1] 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 [16]. 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.[2]

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[3] 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 [36], race conditions [13], unreachable code, redundant tests, unsatisfiable conditions, and more. Model checking tools have also received much attention in the Erlang community [21, 3], being *McErlang* [4] one of the most powerful nowadays due to its support for a very substantial part of the Erlang language. Regarding testing tools, the most important ones 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, Quiviq 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,

[1]The official Core Erlang [9, 10] should not be confused with the subsets of Erlang that the previous papers covered, although they usually refer them as *some* core Erlang.

[2]Although a very expressive feature, we have omitted bit syntax for two reasons: First, to keep the presentation as clear as possible. Second, because it is not completely supported by the `cerl_clauses` library we use to evaluate Core Erlang `case` expressions in the implementation of the debugger. However, in A we explain all the modifications and extensions needed to incorporate bit syntax in the theoretical framework of our debugger.

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

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 `if`/`case`/`try-catch` expressions or inner function calls in the code of a suspicious function are correct. The zoom declarative debugger fulfills a similar task, asking only about the correctness of bindings and `if`/`case`/`try-catch` branch selections since it abstracts from inner function calls—they have been checked in the previous phase. Moreover, the navigation strategy automatically guides the debugging session over the *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 `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 *hot swapping*.

Example 3.1. *Figure 1 presents a* Vigenère cipher *written in Erlang, extracted from the programming chrestomathy* Rosetta Code.[4] *The program exports functions* `encrypt/2` *and* `decrypt/2` *that, given a text and a key, encipher or decipher it, respectively. The Vigenère cipher is a simple and well-known method for encrypting alphabetic text by adding* modulo 26 *(or subtracting, in the case of decrypting) the letters of the text and key in the same position. For example, to cypher the text* `Attack tomorrow dawn` *using the key* `lemon` *we have:*

$$\begin{array}{rll} \textit{Text} & \texttt{ATTACKTOMORROWDAWN} & \\ \textit{Key} & \texttt{LEMONLEMONLEMONLEM} & +_{26} \\ \hline \textit{Cryptogram} & \texttt{LXFOPVXAABCVAKQLAZ} & \end{array}$$

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

[4]`http://rosettacode.org/wiki/Vigen%C3%A8re_cipher#Erlang`, accessed May 9th, 2013.

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

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

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

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

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

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

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

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` *passed as an argument to the letters in the text and the extended key. Finally, functions* `encrypt/2` *and* `decrypt/2` *encrypt and decrypt a text using a key by simply calling* `crypt/3` *with the functions* `encipher/2` *and* `decipher/2`*. The complete program also contains other simple functions like* `isalpha/1` *to check whether the argument is a lowercase or uppercase letter, or* `toupper/1` *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>,[],"N")`*. 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.

fname	::=	Atom / Integer
lit	::=	Atom | Integer | Float | Char | String | `[ ]`
fun	::=	`fun(`var$_1$ `,` ... `,` var$_n$`)` `->` exprs
clause	::=	pats `when` exprs$_1$ `->` exprs$_2$
pat	::=	var | lit | `[` pats$_1$ `|` pats$_2$ `]` | `{` pats$_1$`,` ... `,` pats$_n$ `}` | var `=` pats
pats	::=	pat | `<` pat$_1$`,` ... `,` pat$_n$ `>`
exprs	::=	expr | `<` expr$_1$`,` ... `,` expr$_n$ `>`
expr	::=	var | lit | ξ | fname | fun | `[` exprs$_1$ `|` exprs$_2$ `]` | `{` exprs$_1$`,` ... `,` exprs$_n$ `}`
		| `let` vars `=` exprs$_1$ `in` exprs$_2$
		| `letrec` fname$_1$ `=` fun$_1$ ... fname$_n$ `=` fun$_n$ `in` exprs
		| `apply` exprs$_{n+1}$ `(` exprs$_1$ `,` ... `,` exprs$_n$ `)`
		| `call` exprs$_{n+1}$`:`exprs$_{n+2}$ `(` exprs$_1$ `,` ... `,` exprs$_n$ `)`
		| `primop` Atom `(` exprs$_1$ `,` ... `,` exprs$_n$ `)`
		| `try` exprs$_1$ `of` `<`var$_1$ `,` ... `,` var$_n$`>` `->` exprs$_2$
		`catch` `<`var'$_1$ `,` ... `,` var'$_m$`>` `->` exprs$_3$
		| `case` exprs `of` clause$_1$ ... clause$_n$ `end`
		| `do` exprs$_1$ exprs$_2$ | `catch` exprs
ξ	::=	Exception(val$_1$, ..., val$_m$)
val	::=	lit | fname | fun | `[` vals$_1$ `|` vals$_2$ `]` | `{`vals$_1$`,` ... `,` vals$_n$`}`
eval	::=	lit | fname | fun | `[` evals$_1$ `|` evals$_2$ `]` | `{`evals$_1$`,` ... `,` evals$_n$`}` | ξ
vals	::=	val | `<` val$_1$`,` ... `,` val$_n$ `>`
evals	::=	eval | `<` eval$_1$`,` ... `,` eval$_n$ `>`
vars	::=	var | `<` var$_1$`,` ... `,` var$_n$ `>`

Figure 2: Core Erlang's Syntax

- `call`: similar to the previous expression but the function applied is the one defined by exprs$_{n+2}$ in the module defined by exprs$_{n+1}$. Both expressions should be evaluated to an atom. For instance, the expression `vigenere:encrypt("Attack tomorrow dawn", "lemon")` in Example 1 will be written in Core Erlang as a `call` expression.

- `primop`: application of built-in functions mainly used to report errors. A typical example is the report of a matching failure in a `case` expression: `primop 'match fail' ('case clause', ...)`.

- `try-catch`: the expression exprs$_1$ is evaluated. If the evaluation does not report any error, then exprs$_2$ is evaluated. Otherwise, the evaluated expression is exprs$_3$. In both cases the appropriate variables are bound to the value of exprs$_1$. Note that m (in the `catch` branch) is the system-dependent number of arguments that exceptions contain, usually the kind of exception and information about the reason.

- `case`: a pattern-matching expression.

The `case` statement plays an important rôle in our presentation and its syntax deserves a more detailed presentation:

Definition 3.1. *The general form of a `case` expression is:*

$$\begin{array}{ll} \texttt{case}\ \mathit{exprs}\ \texttt{of} & \mathit{pats}_1\ \texttt{when}\ \mathit{exprs}'_1\ \texttt{->}^{r_1}\ \mathit{exprs}''_1 \\ & \dots \\ & \mathit{pats}_n\ \texttt{when}\ \mathit{exprs}'_n\ \texttt{->}^{r_n}\ \mathit{exprs}''_n\ \texttt{end} \end{array}$$

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 r_i is not part of the code, it is added in our presentation as a label indicating a reference to the ith 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 ξ 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* $< x_1, \dots, x_n >$ 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):

```
'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 >= $A, L =< $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:

```
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:

```
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.

Erlang	**Core Erlang**
case $expr$ of pat_1 when $expr'_1$ -> $expr''_1$ ... pat_n when $expr'_n$ -> $expr''_n$ end	case <$expr$> of <pat_1> when <$expr'_1$> -> <$expr''_1$> ... <pat_n> when <$expr'_n$> -> <$expr''_n$> <cor_0> when 'true' -> primop 'match_fail' ({'case_clause', cor_0}) end
if $expr'_1$ -> $expr''_1$ ... $expr'_n$ -> $expr''_n$ end	case <> of <> when <$expr'_1$> -> <$expr''_1$> ... <> when <$expr'_n$> -> <$expr''_n$> <> when 'true' -> primop 'match_fail' ('if_clause') end
try $expr$ of pat_1 when $expr'_1$ -> $expr''_1$ ... pat_i when $expr'_i$ -> $expr''_i$ catch $class_{i+1}$: pat_{i+1} when $expr'_{i+1}$ -> $expr''{}_{i+1}$ ... $class_n$: pat_n when $expr'_n$ -> $expr''_n$ end	try <$expr$> of <cor_0> -> case <cor_0> of <pat_1> when <$expr'_1$> -> <$expr''_1$> ... <pat_i> when <$expr'_i$> -> <$expr''_i$> <cor_1> when 'true' -> primop 'match_fail' ({'try_clause', cor_1}) end catch <cor_2, cor_3, cor_4> -> case {cor_2, cor_3, cor_4} of <{$class_{i+1}, pat_{i+1}, cor_5$}> when <$expr'_{i+1}$> -> <$expr''_{i+1}$> ... <{$class_n, pat_n, cor_6$}> when <$expr'_n$> -> <$expr''_n$> <{$cor_7, cor_8, cor_9$}> when 'true' -> primop 'raise' (cor_8, cor_9) end end
pat = $expr$	case <$expr$> of <pat> when 'true' -> <$expr$> <cor_0> when 'true' -> primop 'match_fail' ({'badmatch', cor_0}) end

Table 1: Erlang expressions and their corresponding Core Erlang `case` expressions

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 *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 *(mod, line, column)* with *line* and *column* the starting position in the code of *mod*). This idea is used to refer to functions, variables, `case` expressions, their argument and their branches, and reserved words.

The set of variables occurring in an expression *expr* is denoted by $var(expr)$. The notation $locvar(r)$, 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 $ctx(r_\lambda)$ with r_λ a reference to a lambda expression `fun`$(var_1, \ldots, var_n)$ `->` $expr$ represents the context variables of r_λ, that is $ctx(r_\lambda) = var(expr) \setminus (\{var_1, \ldots, var_n\} \cup locvar(r_\lambda))$. 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 this notions we can present the evaluations used in the calculus:

$\langle guard(r_b), \theta\rangle \to val$**,** which indicates that the guard of the branch referenced by r_b has been evaluated to val, given the context in θ.

$\langle pathbind(r_b, vals), \theta\rangle \to \hat{\theta}$**,** which indicates that, given the context θ, the matching between the pattern in the branch referenced by r_b and the value $vals$ is $\hat{\theta}$.

$\langle fails(vals, r_b), \theta\rangle$**,** which indicates that the branch referenced by r_b is not taken when the `case` argument is evaluated to $vals$ and the context is denoted by the substitution θ.

$\langle succeeds(vals, r_b), \theta\rangle \to \theta'$**,** which indicates that the branch referenced by r_b is taken when the `case` argument is evaluated to $vals$ and the context is denoted by the substitution θ, returning a substitution θ' binding the new variables in the pattern.

$\langle vars, exprs, \theta\rangle \to \theta'$**,** which indicates that the variables in $vars$ are bound to the values obtained when evaluating the expression $exprs$, giving rise to the substitution θ'.

$\langle exprs, \theta\rangle \to vals$**,** which indicates that the value $vals$ is obtained when evaluating the expression $exprs$ with the context θ.

$\langle r, \theta\rangle \to vals$**,** which indicates that the expression referenced by r is evaluated to the value $vals$ when the context θ is used.

$\langle c\_arg(r_c), \theta\rangle \to evals$**,** which computes the value $evals$ obtained when evaluating the argument of a `case` expression, where r_c is the reference to the `case` expression and θ is the context.

$\langle c\_result(r_b), \theta\rangle \to evals$**,** which computes the value $evals$ obtained when a branch of a `case` expression is evaluated, where r_b is the reference to the branch and θ 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 θ, and the existence of a global environment ρ which is initially empty and is extended by adding the functions defined by the `letrec` expressions. The notation $CESC \models_{(P,T)} \mathcal{E}$, where $\mathcal{E}$ is an evaluation, is employed to indicate that $\mathcal{E}$ can be proven w.r.t. the program P with the proof tree T in $CESC$, while $CESC \not\models_P \mathcal{E}$ 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 a inference rule (R) are called (R)-nodes in the rest of the paper. Finally, we introduce the *match* function to reproduce the syntactic matching performed by Erlang.

Definition 4.1. *The* match *function is defined as follows:*
$match(<\ pat_1, \ldots, pat_n\ >, <\ val_1, \ldots, val_n\ >) = \theta_1 \uplus \ldots \uplus \theta_n$
  where $\theta_i = synMatch(pat_i, val_i)$
$match(pat, val) = synMatch(pat, val)$

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

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 6 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 $vals$. This matching generates the substitution $\hat{\theta}$, which will be $\bot$ when the matching is not possible:

$$(\mathsf{PATBIND})\frac{}{\langle \mathit{patbind}(r_b, \mathit{vals}), \theta\rangle \to \hat{\theta}}$$

with r_b a reference to *pats* `when` exprs_1 `->` exprs_2 and $\hat{\theta} \equiv \mathit{match}(\mathit{pats}\theta, \mathit{vals})$.

The (GUARD) rule evaluates the guard of the branch referenced by r_b:

$$(\mathsf{GUARD})\frac{\langle \mathit{exprs}\theta, \theta\rangle \to \mathit{val}}{\langle \mathit{guard}(r_b), \theta\rangle \to \mathit{val}}$$

where r_b is a reference to *pats* `when` exprs_1 `->` exprs_2.

The (FAIL_1) rule indicates that a branch cannot be executed when the pattern fails:

$$(\mathsf{FAIL}_1)\frac{\langle \mathit{patbind}(r_b, \mathit{vals}), \theta\rangle \to \bot}{\langle \mathit{fails}(\mathit{vals}, r_b), \theta\rangle}$$

where r_b is a reference to *pats* `when` exprs_1 `->` exprs_2.

The (FAIL_2) rule is used when the matching succeeds but the `when` condition evaluated with the new substitution fails:

$$(\mathsf{FAIL}_2)\frac{\langle \mathit{patbind}(r_b, \mathit{vals}), \theta\rangle \to \theta' \qquad \langle \mathit{guard}(r_b), \theta''\rangle \to \texttt{'false'}}{\langle \mathit{fails}(\mathit{vals}, r_b), \theta\rangle}$$

with $\theta'' \equiv \theta \uplus \theta'$ and r_b a reference to *pats* `when` exprs_1 `->` exprs_2.

The (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:

$$(\mathsf{SUCC})\frac{\langle \mathit{patbind}(r_b, \mathit{vals}), \theta\rangle \to \theta' \qquad \langle \mathit{guard}(r_b), \theta''\rangle \to \texttt{'true'}}{\langle \mathit{succeeds}(\mathit{vals}, r_b), \theta\rangle \to \theta'}$$

with $\theta'' \equiv \theta \uplus \theta'$, and where r_b is a reference to *pats* `when` exprs_1 `->` exprs_2.

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

$$(\mathsf{BIND})\frac{\langle \mathit{exprs}, \theta\rangle \to \texttt{<}\ \mathit{val}_1, \ldots, \mathit{val}_n\ \texttt{>}}{\langle \texttt{<}\ r_1, \ldots, r_n\ \texttt{>}, \mathit{exprs}, \theta\rangle \to \{\mathit{var}_1 \mapsto \mathit{val}_1, \ldots, \mathit{var}_n \mapsto \mathit{val}_n\}}$$

with $r_1, \ldots, r_n$ references to variables $\mathit{var}_1, \ldots, \mathit{var}_n$.

The (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 $\mathit{exprs}\theta$) and then evaluating it:

$$(\mathsf{BFUN})\frac{\langle \mathit{expr}\theta, \theta\rangle \to \mathit{evals}}{\langle r_f, \theta\rangle \to \mathit{evals}}$$

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

4.3 Rules for expressions

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

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

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

$$(\mathsf{SEQ})\frac{\langle \mathit{expr}_1, \theta\rangle \to \mathit{val}_1 \quad \ldots \quad \langle \mathit{expr}_n, \theta\rangle \to \mathit{val}_n}{\langle \texttt{<}\ \mathit{expr}_1, \ldots, \mathit{expr}_n\ \texttt{>}, \theta\rangle \to \texttt{<}\ \mathit{val}_1, \ldots, \mathit{val}_n\ \texttt{>}}$$

Similarly, the rules (TUP) and (LIST) evaluate tuples and lists, respectively:

$$\text{(TUP)}\ \frac{\langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n}{\langle \{\mathit{exprs}_1, \ldots, \mathit{exprs}_n\}, \theta\rangle \to \{\mathit{vals}_1, \ldots, \mathit{vals}_n\}}$$

$$\text{(LIST)}\ \frac{\langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \langle \mathit{exprs}_2, \theta\rangle \to \mathit{vals}_2}{\langle [\mathit{exprs}_1 | \mathit{exprs}_2], \theta\rangle \to [\mathit{vals}_1 | \mathit{vals}_2]}$$

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 ith branch and verify the guard, being this the first branch where this happens. Finally, the evaluation continues to compute the final result:

$$\text{(CASE)}\ \frac{\langle c\_arg(r_c), \theta\rangle \to \mathit{vals} \quad \begin{array}{l}\langle \mathit{fails}(\mathit{vals}, r_1), \theta\rangle \\ \langle \mathit{fails}(\mathit{vals}, r_{i-1}), \theta\rangle \\ \ldots \\ \langle \mathit{succeeds}(\mathit{vals}, r_i), \theta\rangle \to \theta'\end{array} \quad \langle c\_result(r_i), \theta''\rangle \to \mathit{evals}}{\langle \texttt{case}^{r_c}\ \mathit{exprs}\ \texttt{of}\ \mathit{clause}_1\ \ldots\ \mathit{clause}_n\ \texttt{end}, \theta\rangle \to \mathit{evals}}$$

where $\theta'' \equiv \theta \uplus \theta'$ and r_c is a reference to a `case` statement where each clause clause_i is defined as pats_i `when` exprs'_i `->`^{r_i} exprs''_i and the labels r_1, …, 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:

$$\text{(C\_ARG)}\ \frac{\langle \mathit{exprs}, \theta\rangle \to \mathit{evals}}{\langle c\_arg(r_c), \theta\rangle \to \mathit{evals}}$$

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 θ:

$$\text{(C\_RESULT)}\ \frac{\langle \mathit{exprs}_i\theta, \theta\rangle \to \mathit{evals}}{\langle c\_result(r_i), \theta\rangle \to \mathit{evals}}$$

with exprs_i 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:

$$\text{(LET)}\ \frac{\langle \texttt{<}r_1, \ldots, r_n\texttt{>}, \mathit{exprs}_1, \theta\rangle \to \theta' \quad \langle \mathit{exprs}_2\theta'', \theta''\rangle \to \mathit{evals}}{\langle \texttt{let <}\mathit{var}_1^{r_1}, \ldots, \mathit{var}_n^{r_n}\texttt{> = }\mathit{exprs}_1\ \texttt{in}\ \mathit{exprs}_2, \theta\rangle \to \mathit{evals}}$$

with $\theta'' \equiv \theta \uplus \theta'$.

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

$$\text{(CALL)}\ \frac{\begin{array}{c}\langle \mathit{exprs}_{n+1}, \theta\rangle \to \mathit{Atom}_1 \quad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \mathit{Atom}_2 \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n \\ \langle r_f, \theta'\rangle \to \mathit{evals}\end{array}}{\langle \texttt{call}\ \mathit{exprs}_{n+1}\texttt{:}\mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{evals}}$$

where Atom_2/n is a function defined as Atom_2/n `= fun (`var_1`, ... ,` var_n`) ->` *expr* in the Atom_1 module (Atom_1 must be different from the built-in module `erlang`), r_f its reference, and $\theta' \equiv \{\mathit{var}_1 \mapsto \mathit{vals}_1, \ldots, \mathit{var}_n \mapsto \mathit{vals}_n\}$.

Analogously, the (CALL_EXEC) rule is in charge of evaluating built-in functions in the `erlang` module. To abstract the concrete system, we use an auxiliary function *exec* that returns the value the Erlang system would compute:

$$\text{(CALL\_EXEC)}\ \frac{\begin{array}{c}\langle \mathit{exprs}_{n+1}, \theta\rangle \to \texttt{'erlang'} \quad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \mathit{Atom}_2 \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{val}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{val}_n \\ \mathit{exec}(\mathit{Atom}_2, \mathit{val}_1, \ldots, \mathit{val}_n) = \mathit{vals}\end{array}}{\langle \texttt{call}\ \mathit{exprs}_{n+1}\texttt{:}\mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{vals}}$$

where $Atom_2/n$ 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:

$$(\mathsf{APPLY}_1)\frac{\begin{array}{c}\langle exprs, \theta\rangle \to r_\lambda \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n \\ \langle r_\lambda, \theta'\rangle \to eval\end{array}}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to eval}$$

with r_λ a reference to $\texttt{fun(}var_1, \ldots, var_n\texttt{) ->}\ exprs'$, and $\theta' \equiv \theta \uplus \{var_1 \mapsto vals_1, \ldots, var_n \mapsto vals_n\}$.

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

$$(\mathsf{APPLY}_2)\frac{\begin{array}{c}\langle exprs, \theta\rangle \to Atom/n \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n \\ \langle exprs'\theta', \theta'\rangle \to evals'\end{array}}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to evals'}$$

if $\rho(Atom/n) = \texttt{fun(}\ var_1\ \texttt{,}\ \ldots\ \texttt{,}\ var_n\ \texttt{)}\ \texttt{->}\ exprs$' and $\theta' \equiv \theta \uplus \{var_1 \mapsto vals_1, \ldots, var_n \mapsto vals_n\}$.

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:

$$(\mathsf{APPLY}_3)\frac{\begin{array}{c}\langle exprs, \theta\rangle \to Atom/n \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n \\ \langle r_f, \theta'\rangle \to evals\end{array}}{\langle \texttt{apply}^r\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to evals}$$

where $Atom/n$ is a function defined in the module $r.mod$ as $Atom/n$ `=` $\texttt{fun (}var_1\ \texttt{,}\ \ldots\texttt{,}\ var\ _n\texttt{) ->}\ expr$, r_f its reference, and $\theta' \equiv \{var_1 \mapsto vals_1, \ldots, var_n \mapsto vals_n\}$.

The rule (PRIMOP) evaluates Erlang predefined functions by using the auxiliary function *eval*:

$$(\mathsf{PRIMOP})\frac{\begin{array}{c}\langle exprs_1, \theta\rangle \to val_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to val_n \\ eval(Atom, val_1, \ldots, val_n) = vals'\end{array}}{\langle \texttt{primop}\ Atom(exprs_1, \ldots, exprs_n), \theta\rangle \to vals'}$$

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:

$$(\mathsf{TRY}_1)\frac{\langle exprs_1, \theta\rangle \to vals' \quad \langle exprs_2\theta'', \theta''\rangle \to evals}{\langle \texttt{try}\ exprs_1\ \texttt{of <}var_1, \ldots, var_n\texttt{> ->}\ exprs_2\ \texttt{catch <}var'_1, \ldots, var'_m\texttt{> ->}\ exprs_3, \theta\rangle \to evals}$$

with $\theta' \equiv match(<\ var_1, \ldots, var_n\ >, vals')$ and $\theta'' \equiv \theta\ \uplus\ \theta'$.

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:

$$(\mathsf{TRY}_2)\frac{\langle exprs_1, \theta\rangle \to Except(val_1, \ldots, val_m) \quad \langle expr_3\theta'', \theta''\rangle \to evals}{\langle \texttt{try}\ exprs_1\ \texttt{of <}var_1, \ldots, var_n\texttt{> ->}\ exprs_2\ \texttt{catch <}var'_1, \ldots, var'_m\texttt{> ->}\ exprs_3, \theta\rangle \to evals}$$

with $\theta' \equiv match(\texttt{<}var'_1, \ldots, var'_m\texttt{>}, \texttt{<}val_1, \ldots, val_m\texttt{>})$ and $\theta'' \equiv \theta \uplus \theta'$

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:

$$(\mathsf{SEQ\_E})\frac{\langle expr_1, \theta\rangle \to val_1 \quad \ldots \quad \langle expr_i, \theta\rangle \to val_i \quad \langle expr_{i+1}, \theta\rangle \to \xi}{\langle \texttt{<}\ expr_1, \ldots, expr_n\ \texttt{>}, \theta\rangle \to \xi}$$

The rule ($\mathsf{CALL\_E_4}$) throws a `bad_argument` exception when the module is not an atom:

$$(\mathsf{CALL\_E_4})\frac{\begin{array}{c}\langle \mathit{exprs}'_1, \theta\rangle \to \mathit{vals}'_1 \qquad \langle \mathit{exprs}'_2, \theta\rangle \to \mathit{vals}'_2\\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n\end{array}}{\langle \texttt{call}\ \mathit{exprs}'_1 : \mathit{exprs}'_2(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{Exception}(\texttt{error}, \texttt{bad\_argument}, \ldots)}$$

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, $\theta_0 \equiv \{\texttt{\_cor1} \mapsto \mathit{txt}\ ;\ \texttt{\_cor0} \mapsto \mathit{key}\}$, $\theta_1 \equiv \theta_0 \uplus \{\texttt{Text} \mapsto \mathit{txt}\ ;\ \texttt{Key} \mapsto \mathit{key}\}$, $\theta_2 \equiv \{\texttt{\_cor2} \mapsto \mathit{txt}\ ;\ \texttt{\_cor1} \mapsto \mathit{txt}\ ;\ \texttt{\_cor0} \mapsto \mathit{fun}\}$ and $\theta_3 \equiv \theta_2 \uplus \{\texttt{RawText} \mapsto \mathit{txt}\ ;\ \texttt{RawKey} \mapsto \mathit{key}\ ;\ \texttt{fun} \mapsto \mathit{fun}\}$. 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 $\bigtriangledown$'s in the premises of the root stand for the reflexive evaluation of the arguments of the function to themselves, while the leftmost subtree indicates that we must evaluate the reference to the function being evaluated, `encrypt/2`, using the substitution θ_0, 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, θ_1. This latter evaluation is proven in $\bigtriangledown_1$.

Figure 3(middle) describes $\bigtriangledown_1$. It shows that the function `crypt/3` is evaluated by reducing the function name and the arguments to themselves (in the $\bigtriangledown$'s) and then the reference to the function is evaluated with the substitution θ_2, obtained by matching the parameters of the function. The proof for $\bigtriangledown_3$ 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 $\bigtriangledown_3$, 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 $\mathcal{I}$, 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 *ICESC*$_f$ (during the first phase when the debugging concentrates only on user functions) and *ICESC*$_Z$ (inside a function code). The discrepancies between the results produced by the program, represented by *CESC* and the results expected by the user, represented by *ICESC*$_f$, *ICESC*$_Z$ 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:

$$\mathcal{I} = \mathcal{I}_{\mathit{fun}} \ \cup\ \mathcal{I}_{\lambda} \ \cup\ \mathcal{I}_{\mathit{var}} \ \cup\ \mathcal{I}_{\mathit{case}}$$

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

- $\mathcal{I}_{\mathit{fun}} = \{\ldots,\ \langle r_f, \theta\rangle \to \mathit{evals},\ \ldots\}$ where
  - r_f is a reference to a function `f/n` defined as
    `f/n = fun (` var_1 `, ... ,` var_n `) ->` *exprs*
  - The domain of the substitution θ must be $\{\mathit{var}_1, \ldots, \mathit{var}_n\}$.
- $\mathcal{I}_{\lambda} = \{\ldots, \langle r_\lambda, \theta\rangle \to \mathit{val}, \ldots\}$ where
  - r_λ is a reference to a lambda abstraction defined as
    `fun(`$\mathit{var}_1, \ldots, \mathit{var}_n$`) ->` *exprs*
  - θ is a substitution such that $\mathit{ctx}(r_\lambda) \cup \{\mathit{var}_1, \ldots, \mathit{var}_n\} \subseteq \mathit{dom}(\theta)$.

$$
\textsf{(CALL)}\dfrac{\triangledown\ \cdots\ \triangledown\quad \textsf{(BFUN)}\dfrac{\textsf{(CASE)}\dfrac{\dfrac{\langle \mathit{vals},\theta_0\rangle \to \mathit{vals}}{\langle c\_\mathit{arg}(r_c),\theta_0\rangle \to \mathit{vals}}\quad \textsf{(SUCC)}\dfrac{\triangledown\quad\triangledown}{\langle \mathit{succeeds}(\mathit{vals},r_1),\theta_0\rangle \to \theta_1}\quad \dfrac{\triangledown_1}{\langle c\_\mathit{result}(r_{b_1}),\theta_1\rangle \to \xi}}{\langle \texttt{case } \mathit{vals} \texttt{ of <Text,Key> -> apply 'crypt'(Text,Key,}\mathit{fun}\texttt{)},\theta_0\rangle \to \xi}}{\langle r_{\texttt{encrypt}},\mathit{id}\rangle \to \xi}}{\langle \texttt{vigenere:encrypt("Attack tomorrow dawn", "lemon")},\mathit{id}\rangle \to \xi}
$$

where proof tree $\triangledown_1$ is defined as:

$$
\textsf{(APPLY}_3\textsf{)}\dfrac{\triangledown\quad\triangledown\quad\triangledown\quad\triangledown\quad \textsf{(BFUN)}\dfrac{\triangledown_2}{\langle r_{\texttt{crypt}},\theta_2\rangle \to \xi}}{\langle \texttt{apply 'crypt'(}\mathit{txt}\texttt{,}\mathit{key}\texttt{,}\mathit{fun}\texttt{)},\theta_1\rangle \to \xi}
$$

where proof tree $\triangledown_2$ is defined as:

$$
\textsf{(CASE)}\dfrac{\dfrac{\langle \mathit{vals}',\theta_2\rangle \to \mathit{vals}'}{\langle c\_\mathit{arg}(r_c'),\theta_2\rangle \to \mathit{vals}'}\quad \textsf{(SUCC)}\dfrac{\triangledown\quad\triangledown}{\langle \mathit{succeeds}(\mathit{vals}',r_1'),\theta_2\rangle \to \theta_3}\quad \dfrac{\triangledown_3}{\langle c\_\mathit{result}(r_{b_1'}),\theta_3\rangle \to \xi}}{\langle \texttt{case } \mathit{vals}' \texttt{ of <RawText,RawKey,Fun> -> let <PlainText> = apply 'normalize'(RawText)}\ldots,\theta_2\rangle \to \xi}
$$

Figure 3: Proof tree for $\langle \texttt{encrypt("Attack tomorrow dawn", "lemon")},\mathit{id}\rangle \to \texttt{Exception(no function clause matching)}$

Thus, $\mathcal{I}_{fun}$ 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, $\mathcal{I}_{\lambda}$ contains the results for any parameters and possible contexts.

The two other sets conforming $\mathcal{I}$ are used for zoom debugging. They are described as follows:

- $\mathcal{I}_{var} = \{\ldots, \langle < r_1, \ldots, r_n >, \theta\rangle \rightarrow < val_1, \ldots, val_n >, \ldots\}$, with r_1, …, r_n references to variables $var_1, \ldots, var_n$.
- $\mathcal{I}_{case} = \mathcal{I}_{r_{c_1}} \cup \ldots \cup \mathcal{I}_{r_{c_n}}$ with $r_{c_1} \ldots r_{c_n}$ references to all the `case` expressions occurring in the program and $\mathcal{I}_{r_{c_1}} \ldots \mathcal{I}_{r_{c_n}}$ their intended interpretations (defined below).

The intended representation of variables $\mathcal{I}_{var}$ 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:

$$\mathcal{I}_{var} = \{\ldots, \langle r_v, \theta\rangle \rightarrow val, \ldots, \}$$

with r_v a reference to a variable v.

The associated value 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 (BIND) inference rule. Observe that the same variable with the same reference r can occur more than once in $\mathcal{I}_{var}$ 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

$$\mathcal{I}_{var} = \{\ldots, \langle r_x, \{y \mapsto 2\}\rangle \rightarrow 4, \langle r_x, \{y \mapsto 3\}\rangle \rightarrow 6, \ldots\}$$

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

The values associated to the `case` expression depend on the context θ. Therefore each set $\mathcal{I}_{r_c}$, with r_c a reference to a `case` expression, is defined as the union of the intended interpretations for the different possible contexts:

$$\mathcal{I}_{r_c} = \langle \mathcal{I}_{r_c}, \theta_1\rangle \cup \langle \mathcal{I}_{r_c}, \theta_2\rangle \cup \ldots$$

Assuming that r_c is a reference to a `case` expression following the syntax of Definition 3.1, with the labels r_1, …, 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 `case` expression in the context θ. Then $\langle \mathcal{I}_{r_c}, \theta\rangle$ is a set with the form:

$$\begin{aligned}\langle \mathcal{I}_{r_c}, \theta\rangle = \{ & \langle c\_arg(r_c), \theta\rangle \rightarrow evals, \\ & \langle patbind(r_1, evals), \theta\rangle \rightarrow \hat{\theta}_1, \langle guard(r_1), \theta'_1\rangle \rightarrow \texttt{'false'}, \langle fails(r_1), \theta\rangle \\ & \langle patbind(r_2, evals), \theta\rangle \rightarrow \hat{\theta}_2, \langle guard(r_2), \theta'_2\rangle \rightarrow \texttt{'false'}, \langle fails(r_2), \theta\rangle \\ & \ldots, \\ & \langle patbind(r_i, evals), \theta\rangle \rightarrow \theta', \langle guard(r_i), \theta''\rangle \rightarrow \texttt{'true'}, \\ & \langle succeeds(r_i), \theta\rangle, \langle c\_result(r_i), \theta''\rangle \rightarrow evals' \ \}\end{aligned}$$

where:

- $\langle c\_arg(r_c), \theta\rangle \rightarrow vals$ represents the expected value ($vals$) obtained when evaluating the `case` argument in the context θ. Each set $\langle \mathcal{I}_{r_c}, \theta\rangle$ contains only one evaluation containing $c\_arg(r_c)$.
- For every failing branch j, $j < i$, $\langle \mathcal{I}_{r_c}, \theta\rangle$ includes $\langle patbind(r_j, evals), \theta\rangle \rightarrow \hat{\theta}_j$, $\langle guard(r_j), \theta\rangle \rightarrow val_j$, and $\langle fails(r_j), \theta\rangle$, with:
  - $\langle patbind(r_j, evals), \theta\rangle \rightarrow \hat{\theta}_j$ represents the matching of $evals$, the value obtained after evaluating the `case` argument, and the pattern of the clause j, $pats_j$, in the context θ. The expected substitution $\hat{\theta}_j$ can be $\perp$ if the matching is not feasible.
  - $\langle guard(r_j), \theta'_j\rangle \rightarrow val_j$ only occurs when $\hat{\theta}_j$ is not $\perp$, and in this case corresponds to the expected evaluation of the guard $exprs'_j$ in the context θ'_j which must be of the form $\theta'_j \equiv \theta \uplus \hat{\theta}_j$.

  - $\langle \mathit{fails}(r_j), \theta\rangle$ indicates that the jth branch is expected to fail in this context.

  Since $j < i$ the clause j has not been selected and therefore we require that either $\hat{\theta_j} \equiv \perp$ or $val_j \equiv$ `'false'`.

- The evaluations $\langle \mathit{patbind}(r_i, \mathit{evals}), \theta\rangle \to \theta'$, $\langle \mathit{guard}(r_i), \theta''\rangle$ represent respectively the pattern matching and the evaluation of the guard of the successful clause, with $\theta'' \equiv \theta \uplus \theta'$.
- $\langle \mathit{succeeds}(r_i), \theta\rangle$ indicates that the ith branch is the successful one in this context.
- $\langle c\_result(r_i), \theta''\rangle \to \mathit{evals}'$ represents the evaluation of the expression associated to the ith 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, $\langle r_f, \theta\rangle \to \mathit{evals}' \in \mathcal{I}_{fun}$ (respectively $\langle r_\lambda, \theta\rangle \to \mathit{evals}' \in \mathcal{I}_\lambda$) iff $\langle c\_result(r_i), \theta'\rangle \to \mathit{evals}' \in \mathcal{I}_{r_c}$, with r_i a reference the success branch of the `case` expression referenced by r_c and θ' a substitution extending θ.

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

$$(\mathsf{BFUN}_\mathcal{I})\frac{}{\langle r_f, \theta\rangle \to \mathit{evals}}$$

where either r_f references to a function and $\langle r_f, \theta\rangle \to \mathit{vals} \in \mathcal{I}_{fun}$, or r_f references to a lambda expression and $\langle r_f, \theta\rangle \to \mathit{vals} \in \mathcal{I}_\lambda$.

The second calculus is called ICESC_Z (intended *CESC* calculus with zoom). This calculus, in addition to replacing (BFUN) by ($\mathsf{BFUN}_\mathcal{I}$), replaces the inferences (BIND), (FAIL_1), (FAIL_2), (SUCC), (C_ARG), (C_RESULT), (PATBIND) and (GUARD) by new rules with the same name subscripted by $\mathcal{I}$. In the following r_b is a reference to a branch of a `case` expression with reference r_c.

$$(\mathsf{BIND}_\mathcal{I})\frac{}{\langle < r_1, \ldots, r_n >, \mathit{exprs}, \theta\rangle \to \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}}$$
with $r_1, \ldots, r_n$ references to variables $var_1, \ldots, var_n$, and $(\langle < r_1, \ldots, r_n >, \theta\rangle \to < val_1, \ldots, val_n >) \in \mathcal{I}_{var}$

$$(\mathsf{FAIL}_\mathcal{I})\frac{}{\langle \mathit{fails}(\mathit{vals}, r_b), \theta\rangle} \quad \text{with } (\langle \mathit{fail}(r_b), \theta\rangle) \in \langle \mathcal{I}_{r_c}, \theta\rangle$$

$$(\mathsf{SUCC}_\mathcal{I})\frac{}{\langle \mathit{succeeds}(\mathit{vals}, r_b), \theta\rangle \to \theta'} \quad \text{with } (\langle \mathit{succeeds}(r_b), \theta\rangle) \in \langle \mathcal{I}_{r_c}, \theta\rangle$$

$$(\mathsf{C\_ARG}_\mathcal{I})\frac{}{\langle c\_arg(r_c), \theta\rangle \to \mathit{vals}} \quad \text{with } (\langle c\_arg(r_c), \theta\rangle \to \mathit{vals}) \in \langle \mathcal{I}_{r_c}, \theta\rangle$$

$$(\mathsf{C\_RESULT}_\mathcal{I})\frac{}{\langle c\_result(r_c), \theta'\rangle \to \mathit{vals}} \quad \text{with } (\langle c\_result(r_c), \theta'\rangle \to \mathit{vals}) \in \langle \mathcal{I}_{r_c}, \theta\rangle$$

$$(\mathsf{PATBIND}_\mathcal{I})\frac{}{\langle \mathit{patbind}(r_b, \mathit{vals}), \theta\rangle \to \hat{\theta}} \quad \text{with } (\langle \mathit{patbind}(r_b, \mathit{vals}), \theta\rangle \to \hat{\theta}) \in \langle \mathcal{I}_{r_c}, \theta\rangle$$

$$(\mathsf{GUARD}_\mathcal{I})\frac{}{\langle \mathit{guard}(r_b), \theta'\rangle \to \mathit{val}} \quad \text{with } (\langle \mathit{guard}(r_b), \theta'\rangle \to \mathit{val}) \in \langle \mathcal{I}_{r_c}, \theta\rangle$$

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 calculus we write $\mathit{ICESC}_\diamond$. Analogously to the case of *CESC*, the notation $\mathit{ICESC}_\diamond \models_{(P,\mathcal{I},T)} \mathcal{E}$ indicates that the evaluation $\mathcal{E}$ can be proven w.r.t. the program P and the intended interpretation $\mathcal{I}$ with proof tree T in $\mathit{ICESC}_\diamond$, while $\mathit{ICESC}_\diamond \not\models_{(P,\mathcal{I})} \mathcal{E}$ indicates that $\mathcal{E}$ cannot be proven in $\mathit{ICESC}_\diamond$. The tree T, the program P, and the intended interpretation $\mathcal{I}$ are omitted when they are not needed.

CESC is used by our tool in order to represent actual Erlang computations, while *ICESC*$_\diamond$ 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$_\diamond$ when ICESC$_\diamond \models_{(P,\mathcal{I})} \mathcal{E}$, and* invalid *when ICESC$_\diamond \not\models_{(P,\mathcal{I})} \mathcal{E}$.*
2. *N is called* buggy *with respect to ICESC$_\diamond$ if $\mathcal{E}$ is invalid with all its children valid (in both cases w.r.t. ICESC$_\diamond$).*

Instead of valid/buggy w.r.t. *ICESC*$_\diamond$ we often use in the rest of the paper the notation *ICESC*$_\diamond$-valid/*ICESC*$_\diamond$-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 $|\cdot|$ the transformation that converts an Erlang expression into a Core expression. Then:*

1. *An evaluation $e\theta \to eval$ is computed by some Erlang system with respect to P iff CESC $\models_P \langle |e|, \theta\rangle \to |eval|$.*
2. *A reduction $e\theta \to eval$ computed by some Erlang system is considered unexpected with respect to ICESC$_\diamond$ by the user iff ICESC$_\diamond \not\models_{P,\mathcal{I}} \langle |e|, \theta\rangle \to |eval|$.*

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*$_\diamond$. 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 $\mathcal{I}$. Let r_c a reference to a* `case` *expression in P and r_i a reference to the ith branch in r_c, which must be of the form: $pats_i$* `when` *$exprs'_i$* `->` *$exprs''_i$. We say that the ith branch of the case r_c is a* wrong branch *when there exists a value evals and a context θ such that some of the followings items hold:*

1. *Suppose that*

$$ICESC_Z \models_{(P,\mathcal{I})} \langle succeeds(evals, r_i), \theta\rangle \to \theta'$$
$$ICESC_Z \models_{(P,\mathcal{I})} \langle patbind(r_i, evals), \theta\rangle \to \theta''$$

$\theta' = \theta \uplus \theta''$ and at least one of the following items hold:

   (a) *$match(pats_i\theta, evals) \neq \theta''$. In this case we say also that $\langle r_i, \theta\rangle$ contains a* wrong fail pattern instance.

(b)

$$\begin{array}{l} match(pats_i\theta, evals) = \theta'' \\ ICESC_Z \models_{(P,\mathcal{I})} \langle guard(r_i), \theta' \rangle \to \texttt{'true'} \\ ICESC_Z \models_{(P,\mathcal{I})} \langle exprs'_i, \theta' \rangle \to \texttt{'false'} \end{array}$$

then, we also say that $\langle r_i, \theta\rangle$ contains a wrong fail guard instance.

2. *Suppose that*

$$\begin{array}{l} ICESC_Z \models_{(P,\mathcal{I})} \langle fail(evals, r_i), \theta\rangle \\ ICESC_Z \models_{(P,\mathcal{I})} \langle patbind(r_i, evals), \theta\rangle \to \hat{\theta}'' \end{array}$$

and at least one of the following items hold:

(a) $match(pats_i\theta, evals) \neq \hat{\theta}''$ *In this case we say also that $\langle r_i, \theta\rangle$ contains a* wrong success pattern instance.

(b)

$$\begin{array}{l} match(pats_i\theta, evals) \equiv \hat{\theta}, \hat{\theta} \neq \perp, \theta' \equiv \theta \uplus \hat{\theta}'' \\ ICESC_Z \models_{(P,\mathcal{I})} \langle guard(r_i), \theta' \rangle \to \texttt{'false'} \\ ICESC_Z \models_{(P,\mathcal{I})} \langle exprs'_i, \theta' \rangle \to \texttt{'true'} \end{array}$$

Then, we also say that $\langle r_i, \theta\rangle$ contains a wrong success guard instance.

For instance if we expect the branch i to succeed and the patter matching to give some substitution as result, but we obtain a different substitution after performing the matching we have found an error in the pattern definition. Analogously, if we expect the branch i to succeed, the pattern matching is expected to provide some binding that it actually obtains, but the guard expression returns `'false'` instead of `'true'` we have found an error in the guard definition. In particular we are interested in errors associated to the first erroneous branch:

Definition 5.4. *We say that the ith branch of the* `case` *expression r_c is the first wrong branch of r_c if:*

1. *r_c contains no wrong* `case` *argument instance (see next definition).*
2. *for $j = 1 \ldots i-1$, r_j is not a wrong branch for r_c.*

Now we are ready to define wrong program:

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

1. *Let r_f be a reference to a either a function f/n =* **fun***(var_1, ..., var_n) -> B, or to a lambda function* **fun***($var_1, \ldots, var_n$) -> B and θ a substitution. Then $\langle r_f, \theta\rangle$ is a* wrong function instance *iff there exists a value evals such that:*

   (a) $ICESC_f \models_{(P,\mathcal{I})} \langle B\theta, \theta\rangle \to evals$.

   (b) $ICESC_f \not\models_{(P,\mathcal{I})} \langle r_f, \theta\rangle \to evals$.

2. *Let $\langle$< $r_1, \ldots, r_n$ >, $exprs, \theta\rangle$ be such that $r_1, \ldots, r_n$ are references to the variables $var_1, \ldots, var_n$ and θ is a context.*

   Then, $\langle$< $r_1, \ldots, r_n$ >, $exprs, \theta\rangle$ is a wrong binding instance *iff there exists a substitution θ' defined as $var_1 \mapsto val_1 \uplus \ldots \uplus var_n \mapsto val_n$ verifying:*

   (a) $ICESC_Z \models_{(P,\mathcal{I})} \langle expr, \theta\rangle \to \theta'$ *and*

   (b) $ICESC_Z \not\models_{(P,\mathcal{I})} \langle$< $r_1, \ldots, r_n$ >, $exprs, \theta\rangle \to \theta'$.

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:*

   (a) *$\langle c\_arg(r_c), \theta\rangle$ is a* wrong `case` argument instance *iff there exists a value evals such that*

      i. $ICESC_Z \models_{(P,\mathcal{I})} \langle exprs, \theta\rangle \to evals$.

      ii. $ICESC_Z \not\models_{(P,\mathcal{I})} \langle c\_arg(r_c), \theta\rangle \to evals$.

   (b) *r_c contains a first wrong branch (Definition 5.4)*

   (c) *$\langle c\_result(r_c), \theta'\rangle$ is a* wrong `case` result instance *iff exists values evals, evals$'$ such that:*

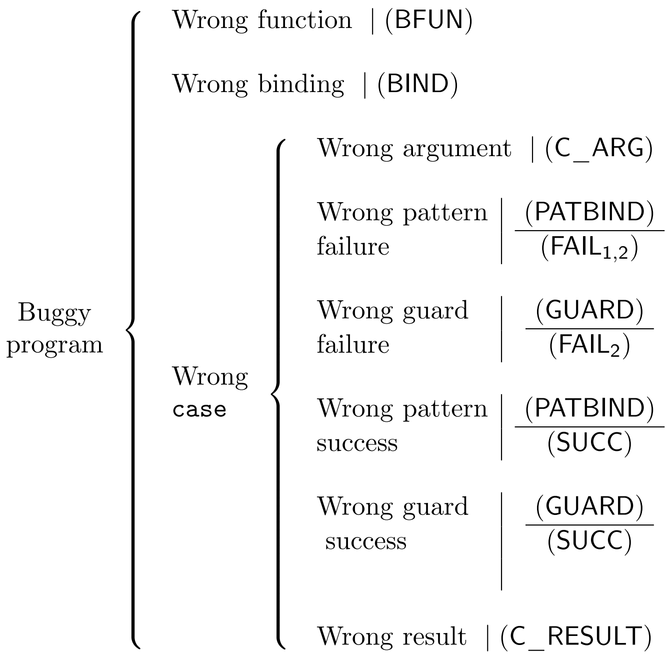

Figure 4: Errors in Core Erlang programs and their corresponding buggy nodes

i. The `case` *expression does not contain a first wrong branch.*
ii. $ICESC_Z \models_{(P,\mathcal{I})} \langle succeeds(evals, r_i), \theta\rangle \rightarrow \theta'$
iii. $ICESC_Z \not\models_{(P,\mathcal{I})} \langle c\_result(r_i), \theta'\rangle \rightarrow evals'$ *and* $ICESC_Z \models_{(P,\mathcal{I})} \langle exprs''_i, \theta\rangle \rightarrow evals'$

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 θ' 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, $P_{\langle e,\theta\rangle}$ denotes the (Core Erlang) program $P \cup \{$* `main/0` *=* `fun() -> case <> of <> when 'true' ->` *$|e\theta|$* `end` *$\}$, with $|\cdot|$ the operator converting Erlang code into Core Erlang code, and* `main` *a new identifier in P.*

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 $\mathit{fib}(n)$ is intended to represent the nth Fibonacci number, and we get a question about the validity of $\mathit{fib}(\texttt{'a'}) \rightarrow 4$. 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 $\mathcal{I}_{fun}$ or $\mathcal{I}_{\lambda}$? 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 $\mathit{fib}(\texttt{'a'}) \rightarrow 4$ 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 enquiries 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 $\langle e, \theta \rangle \rightarrow eval$ an evaluation. A tree T is an Abbreviated Proof Tree (APT in short) for the evaluation with respect to P, iff there is a proof tree T' such that $CESC \models_{(P_{\langle e,\theta\rangle},T')} \langle r_{\mathtt{main}}, id\rangle \rightarrow eval$, and $T = APT(T')$ with the transformation APT defined as follows:*

$$APT\left(\textsf{(BFUN)}\,\frac{T}{\mathcal{E}}\right) \quad = \frac{APT(T)}{\mathcal{E}}$$

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

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, $\mathcal{I}$ its intended interpretation, and $e\theta \rightarrow eval$ an evaluation unexpected with respect to $ICESC_f$. Then, exists at least one proof tree T' such that $CESC \models_{(P_{\langle e,\theta\rangle},T')} \langle r_{\mathtt{main}}, id\rangle \rightarrow |eval|$, and every tree T' in these conditions verifies that $T \equiv APT(T')$ contains at least*

$$\text{(BFUN)}\,\dfrac{\text{(BFUN)}\,\dfrac{\text{(BFUN)}\,\dfrac{\nabla \quad \cdots \quad \nabla}{\langle r_{\texttt{crypt/3}}, \theta_1\rangle \to \xi}}{\langle r_{\texttt{encrypt/2}}, id\rangle \to \xi}}{\langle r_{\texttt{main}}, id\rangle \to \xi}$$

Figure 5: APT for `encrypt("Attack tomorrow dawn", "lemon")`

one buggy node N, and every buggy node in T corresponds to a wrong instance of a user function different from `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 ∇'s on top of the tree correspond to applications of the functions used directly by `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, that depend in general on the values of several previous 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 $\langle e, \theta\rangle \to eval$ an unexpected evaluation w.r.t. P. Let T' be a proof tree such that $CESC \models_{(P_{\langle e,\theta\rangle}, T')} \langle r_{\mathtt{main}}, id\rangle \to eval$, 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, $APTZ(T)$ is defined as follows:*

$$(APTZ_1) \quad APTZ\left(\text{(R)}\,\frac{T_1 \ldots T_n}{\mathcal{E}}\right) \quad = \text{(R)}\,\frac{APTZ(T_1)\ldots APTZ(T_n)}{\mathcal{E}}$$

with (R) *either any zoom inference or a* (CASE) *inference.*

$$(APTZ_2) \quad APTZ\left(\text{(BFUN)}\,\frac{T_1 \ldots T_n}{\mathcal{E}}\right) \quad = \emptyset,$$

if $\mathcal{E}$ is not the root of T.

$$(APTZ_3) \quad APTZ\left(\text{(R)}\,\frac{T_1 \ldots T_n}{\mathcal{E}}\right) \quad = APTZ(T_1)\ldots APTZ(T_n),$$

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) *$ICESC_Z$-buggy node in some APTZ T. Then we say that N is a* wrong `case` branch witness *for the ith branch of a* `case` *expression r_c iff:*

1. *N is the premise a $ICESC_Z$-invalid* (SUCC), (FAIL$_1$), *or* (FAIL$_2$) *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.*

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, $\mathcal{I}$ its intended interpretation, and $e\theta \to eval$ an unexpected evaluation. Let T' be a proof tree such that $CESC \models_{(P_{\langle e,\theta\rangle},T')} \langle r_{\mathtt{main}}, id\rangle \to |eval|$. Let T'' be defined as $T'' \equiv APT(T')$, M be an $ICESC_f$ buggy node in T'', and T'_M the subtree of T' rooted by M. Finally, define a new tree T as $T \equiv APTZ(T'_M)$. Then T contains at least one $ICESC_Z$-buggy node N that verifies one of the following items:*

1. *N is a* (BIND) *node of the form*

$$\langle < r_1, \ldots, r_n >, exprs, \theta\rangle \to \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$$

   Then, $\langle < r_1, \ldots, r_n >, exprs, \theta\rangle$ is a wrong binding instance.

2. *N a* (C_ARG) *node, $N \equiv \langle c\_arg(r_c), \theta\rangle \to evals$. Then, $\langle c\_argc(r_c), \theta\rangle$ is a wrong* `case` *argument instance.*

3. *N is a* (PATBIND)/(GUARD) *premise of a* (FAIL$_1$)/(FAIL$_2$) *node P, $P \equiv \langle fails(vals, r_i), \theta\rangle$, with r_i a reference to the ith branch of a* `case` *expression r_c, and such that N is a wrong* `case` *branch witness. Then $\langle r_i, \theta\rangle$ is an unexpected* `case` *failure instance, and in particular:*

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

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

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

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

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

5. *N is a* (C_RESULT) *node of the form $\langle c\_result(r_i), \theta\rangle \to evals$, and there is neither a* (C_ARG) *buggy node nor a wrong* `case` *branch witness for the same* `case` *expression. Then, $\langle c\_result(r_c), \theta\rangle \to evals$ 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 4 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 (FAIL$_1$) or (FAIL$_2$) 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` (**E**rlang **D**eclarative **D**ebugger), 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 `call = value?`, asking whether the function or λ-abstraction application `call` should evaluate to `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 $c\_arg(r_b)$) 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, $c\_result(r_b)$) 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.

[5]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 presents more questions to the user, but they are presented in asked 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 could contain an error" is used instead of "Call to a function that contains 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 asigned:
     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.
```

or
```
   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(Pars) -> 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:

```
> 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:

```
> 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:

```
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:

```
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 `"LEMONLEMONLEM"`, 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

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

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

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.

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

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

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 items the company has to buy to supply all the orders. Both orders and stocks are represented as lists of tuples* `{item, Name, Quantity}`*, where* `Name` *is the name of the item and* `Quantity` *the number of items of that kind (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.*[6] `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`*.*

[6] `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`.

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}]`*.*[7] *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/t/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/t/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/t/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/t/d/i/s/u/a]: n
stock:check_item({item,rice,7}, [{item,rice,5},
{item,bajoqueta,8}]) = {true, {item,rice,7}}?
  [y/n/t/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`, *uo* for `unify_orders`, *ci* for `check_item`, *res* stands for `[{item, water, 3}, {item, rice, 7}]`, and *res'* for `{true, {item, water, 3}}`, the substitutions θ_0 and θ_1 basically transmit the initial requirements explained above, θ_2 takes the result of the unification and the initial stock, and θ_3 and θ_4 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
```

[7] *Bajoqueta* is the Valencian word for a kind of green bean essential in the *paella valenciana*.

$$\dfrac{\dfrac{\dfrac{\dfrac{\triangledown \quad \triangledown \quad \triangledown}{(2)\ \langle r_{uo}, \theta_1\rangle \to \mathit{res}} \qquad \dfrac{\dfrac{\dfrac{}{(5)\ \langle r_{ci}, \theta_4\rangle \to \mathit{res}'}}{(4)\ \langle r_{\lambda}, \theta_3\rangle \to \mathit{res}'} \qquad \triangledown}{(3)\ \langle r_{co}, \theta_2\rangle \to \mathit{res}}}{(1)\ \langle r_{cos}, \theta_0\rangle \to \mathit{res}}}{\langle r_{\texttt{test}}, id\rangle \to \mathit{res}}}{\langle r_{\texttt{main}}, id\rangle \to \mathit{res}}$$

Figure 7: Abbreviated tree for the zoom debugging session

```
For the case expression:
case ItemStock of
  {item, Name, Q2} ->
      if Q1 > Q2 -> {true, {item, Name, Q1 - Q2}};
         true -> false
      end;
  false -> {true, Needed}
end
Is there anything incorrect?
1.- The context:
        ItemStock = false
        Needed = {item,rice,7}
2.- The argument value: false.
3.- Enter in the second clause.
4.- The final value: {true,{item,rice,7}}.
5.- Nothing.
[1/2/3/4/5/d/s/u/a]? 5
Given the context:
        Name = rice
        Stock = [{item,rice,5},{item,bajoqueta,8}],
the following variable is asigned:
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:
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, 2, Stock)`.

The tree for this debugging session is partially depicted in Figure 8. The root of the tree is the buggy node detected in the previous session, which has the substitution $\theta_4 \equiv \{$`Needed` $\mapsto$ `{item,rice,7}` ; `Stock` $\mapsto$ `[{item,rice,5},{item,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,

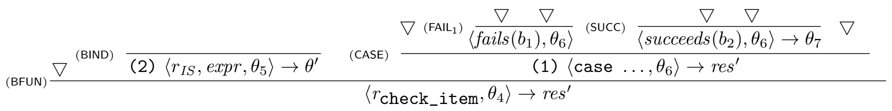
(FAIL$_1$)
$\langle fails(b_1), \theta_6\rangle$
(SUCC)
$\langle succeeds(b_2), \theta_6\rangle \to \theta_7$
(BIND)
(CASE)
(2) $\langle r_{IS}, expr, \theta_5\rangle \to \theta'$
(1) $\langle$case $\ldots, \theta_6\rangle \to res'$
(BFUN)
$\langle r_{\texttt{check\_item}}, \theta_4\rangle \to res'$

Figure 8: Abbreviated zoom tree for the debugging session

Program	**LOC**	**Wrong functions**			**Zoom**		
		Nodes	D&Q	TD	Nodes	D&Q	TD
24_game	73	19	4	3	9	2	2
ackermann	13	91	7	8	-	-	-
align cols	40	32	3	3	4	2	2
caesar	38	147	3	2	5	3	3
complex	60	7	2	2	6	2	3
dutch	41	81	12	11	-	-	-
miller_rabin	80	152	6	10	7	1	1
rfib	6	288	11	12	-	-	-
rle	44	69	5	19	8	1	1
roman	24	9	3	6	9	8	8
sieve	37	56	6	13	7	1	1
sum_digits	16	8	2	6	4	2	2
ternary	91	64	7	10	7	5	4
turing	72	49	7	9	10	2	4

Figure 9: Results of `edd` with different programs.

Sieve of Eratosthenes, Caesar cipher, Run-length encoding... These programs have been extracted from the *Rosetta Code* website (`http://rosettacode.org`) and modified to have a bug.[8] 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.

[8]The bug shown in Section 3 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 rôles is an advantage during the development of any software project.

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

[1] M. Alpuente, D. Ballis, F. Correa, and M. Falaschi. An integrated framework for the diagnosis and correction of rule-based programs. *Theoretical Computer Science*, 411(47):4055–4101, 2010.

[2] J. Armstrong, M. Williams, C. Wikstrom, and R. Virding. *Concurrent Programming in Erlang*. Prentice-Hall, Englewood Cliffs, New Jersey, USA, second edition, 1996.

[3] Thomas Arts, Clara Benac Earle, and Juan José Sánchez Penas. Translating Erlang to muCRL. In *Proceedings of the International Conference on Application of Concurrency to System Design, ACSD 2004*, pages 135–144. IEEE Computer Society Press, June 2004.

[4] Clara Benac Earle and Lars Ake Fredlund. Recent improvements to the McErlang model checker. In *Proceedings of the 8th ACM SIGPLAN workshop on ERLANG*, ERLANG 2009, pages 93–100, New York, NY, USA, 2009. ACM.

[5] R. Caballero, C. Hermanns, and H. Kuchen. Algorithmic debugging of Java programs. In Francisco López-Fraguas, editor, *Proceedings of the 15th Workshop on Functional and (Constraint) Logic Programming, WFLP 2006, Madrid, Spain*, volume 177 of *Electronic Notes in Theoretical Computer Science*, pages 75–89. Elsevier, 2007.

[6] Rafael Caballero. A declarative debugger of incorrect answers for constraint functional-logic programs. In Sergio Antoy and Michael Hanus, editors, *Proceedings of the 2005 ACM SIGPLAN Workshop on Curry and Functional Logic Programming, WCFLP 2005, Tallinn, Estonia*, pages 8–13. ACM Press, 2005.

[7] Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. A calculus for zoom debugging sequential Erlang programs. Technical Report 07/13, Departamento de Sistemas Informáticos y Computación, April 2013.

[8] Rafael Caballero, Enrique Martin-Martin, Adrián Riesco, and Salvador Tamarit. A declarative debugger for sequential Erlang programs. In M. Veanes and L. Vigan, editors, *Proceedings of the 7th International Conference on Tests and Proofs, TAP 2013*, volume 7942 of *Lecture Notes in Computer Science*, pages 96–114. Springer, 2013.

[9] Richard Carlsson. An introduction to Core Erlang. In *Proceedings of the Erlang Workshop 2001, in connection with PLI 2001*, pages 5–18, 2001.

[10] Richard Carlsson, Björn Gustavsson, Erik Johansson, Thomas Lindgren, Sven-Olof Nyström, Mikael Pettersson, and Robert Virding. *Core Erlang 1.0.3 language specification*, November 2004. Available at `http://www.it.uu.se/research/group/hipe/cerl/doc/core_erlang-1.0.3.pdf`.

[11] Richard Carlsson and Mickaël Rémond. EUnit: a lightweight unit testing framework for Erlang. In *Proceedings of the 2006 ACM SIGPLAN workshop on Erlang*, ERLANG 2006, pages 1–1, New York, NY, USA, 2006. ACM.

[12] Francesco Cesarini and Simon Thompson. *Erlang Programming: A Concurrent Approach to Software Development*. O'Reilly Media, Inc., 2009.

[13] Maria Christakis and Konstantinos Sagonas. Static detection of race conditions in Erlang. In Manuel Carro and Ricardo Pena, editors, *Practical Aspects of Declarative Languages, PADL 2010*, volume 5937 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2010.

[14] N. H. Christensen. *Domain-specific languages in software development - and the relation to partial evaluation*. PhD thesis, DIKU, Dept. of Computer Science, University of Copenhagen, Denmark, July 2003.

[15] Koen Claessen and Hans Svensson. A semantics for distributed Erlang. In *Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, ERLANG 2005, pages 78–87, New York, NY, USA, 2005. ACM.

[16] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude: A High-Performance Logical Framework*, volume 4350 of *Lecture Notes in Computer Science*. Springer, 2007.

[17] R. David and G. Mounier. An intuitionistic lambda-calculus with exceptions. *Journal of Functional Programming*, 15(1):33–52, January 2005.

[18] Philippe de Groote. A simple calculus of exception handling. In Mariangiola Dezani-Ciancaglini and Gordon D. Plotkin, editors, *Proceedings of the 2nd International Conference on Typed Lambda Calculi and Applications, TLCA 1995*, volume 902 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 1995.

[19] Lars-Åke Fredlund. *A Framework for Reasoning about Erlang Code*. PhD thesis, The Royal Institute of Technology, Sweden, August 2001.

[20] Reiner Hähnle, Marcus Baum, Richard Bubel, and Marcel Rothe. A visual interactive debugger based on symbolic execution. In Charles Pecheur, Jamie Andrews, and Elisabetta Di Nitto, editors, *25th IEEE/ACM International Conference on Automated Software Engineering, ASE 2010*, pages 143–146. ACM, 2010.

[21] Frank Huch. Verification of erlang programs using abstract interpretation and model checking. *SIGPLAN Not.*, 34(9):261–272, September 1999.

[22] John Hughes. QuickCheck testing for fun and profit. In Michael Hanus, editor, *Practical Aspects of Declarative Languages*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer, 2007.

[23] David Insa and Josep Silva. An algorithmic debugger for Java. In Michele Lanza and Andrian Marcus, editors, *Proceedings of the 26th IEEE International Conference on Software Maintenance, ICSM 2010*, pages 1–6. IEEE Computer Society, 2010.

[24] Tobias Lindahl and Konstantinos Sagonas. Detecting software defects in telecom applications through lightweight static analysis: A war story. In Wei-Ngan Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems, APLAS 2004*, volume 3302 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2004.

[25] Ian MacLarty. Practical declarative debugging of Mercury programs. Master's thesis, University of Melbourne, 2005.

[26] Lee Naish. Declarative diagnosis of missing answers. *New Generation Computing*, 10(3):255–286, 1992.

[27] Lee Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[28] Martin Neuhäußer. Abstraktion und model checking von Core Erlang-programmen in Maude, 2005. RWTH Aachen University, Diplomarbeit.

[29] Martin Neuhäußer and Thomas Noll. Abstraction and model checking of Core Erlang programs in Maude. *Electronic Notes in Theoretical Computer Science*, 176(4):147–163, July 2007.

[30] Henrik Nilsson. How to look busy while being as lazy as ever: the implementation of a lazy functional debugger. *Journal of Functional Programming*, 11(6):629–671, 2001.

[31] Henrik Nilsson and Jan Sparud. The evaluation dependence tree as a basis for lazy functional debugging. *Automated Software Engineering*, 4:121–150, 1997.

[32] Manolis Papadakis and Konstantinos Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 2011 ACM SIGPLAN Erlang Workshop*, pages 39–50. ACM Press, 2011.

[33] B. Pope. *A Declarative Debugger for Haskell.* PhD thesis, The University of Melbourne, Australia, 2006.

[34] Bernard Pope. Declarative debugging with Buddha. In Varmo Vene and Tarmo Uustalu, editors, *5th International School on Advanced Functional Programming, AFP 2004*, volume 3622 of *Lecture Notes in Computer Science*, pages 273–308. Springer, 2005.

[35] Adrián Riesco, Alberto Verdejo, Narciso Martí-Oliet, and Rafael Caballero. Declarative debugging of rewriting logic specifications. *Journal of Logic and Algebraic Programming*, 81(7-8):851–897, 2012.

[36] Konstantinos Sagonas, Josep Silva, and Salvador Tamarit. Precise explanation of success typing errors. In *Proceedings of the ACM SIGPLAN 2013 workshop on Partial evaluation and program manipulation*, PEPM 2013, pages 33–42, New York, NY, USA, 2013. ACM.

[37] Ehud Yehuda Shapiro. *Algorithmic Program Debugging.* ACM Distinguished Dissertation. MIT Press, 1983.

[38] Josep Silva. A comparative study of algorithmic debugging strategies. In Germán Puebla, editor, *Proceedings of the 16th International Symposium on Logic-Based Program Synthesis and Transformation, LOPSTR 2006*, volume 4407 of *Lecture Notes in Computer Science*, pages 143–159. Springer, 2007.

[39] Josep Silva. A survey on algorithmic debugging strategies. *Advances in Engineering Software*, 42(11):976–991, 2011.

[40] Hans Svensson and Lars-Åke Fredlund. A more accurate semantics for distributed Erlang. In *Proceedings of the 2007 SIGPLAN workshop on ERLANG Workshop*, ERLANG 2007, pages 43–54, New York, NY, USA, 2007. ACM.

[41] Alexandre Tessier and Gérard Ferrand. Declarative diagnosis in the CLP scheme. In Pierre Deransart, Manuel V. Hermenegildo, and Jan Maluszynski, editors, *Analysis and Visualization Tools for Constraint Programming, Constraint Debugging (DiSCiPl project)*, volume 1870 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2000.

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

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

lit	::=	... | BitString
pat	::=	... | #{ $\text{bitpat}_1, \ldots, \text{bitpat}_n$ }#
bitpat	::=	#< pat_b >(*opts*)
pat_b	::=	var | Integer | Float
expr	::=	... | #{ $\text{bitexpr}_1, \ldots, \text{bitexpr}_n$ }#
bitexpr	::=	#< expr >(*opts*)

Figure 11: Extension of Core Erlang's Syntax to consider bit syntax.

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 socked-based communication applications, where *segments*—a.k.a. *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, `<<Origin:8, Destination:8, Length:8, Message:Length>>` 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:

- `to_bits(val, opts)`, which given an integer or float value `val` and some encoding options `opts` returns the bit string that represents `val`. For example, `to_bits(127, {8,1,integer,unsigned,big})` will be evaluated to the bit string `"011111111"`, the binary value of the unsigned integer 127 using 8 bits and big-endian.

- `from_bits(bits, opts)`, which given a bit string `bits` and some encoding options `opts`, returns a pair `(val, bits')` where `val` is the value represented in the first bits of `bits` (according to the encoding options `opts`) and `bits'` is the rest of the bit string. For example, `from_bits("0111111100000011", {8,1,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 `"00000011"` 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 *synMatch* to match bit patterns and bit strings and also a new bit-matching function $synMatch_b$:

$synMatch(\#\{bitpat_1, \ldots, bitpat_n\}\#, BitString) = \theta_1 \uplus \ldots \uplus \theta_n$, where
$(\theta_1, BitString_1) \equiv synMatch_b(bitpat_1, BitString)$
$(\theta_2, BitString_2) \equiv synMatch_b(bitpat_2\theta_1, BitString_1)$
$\ldots$
$(\theta_n, \epsilon) \equiv synMatch_b(bitpat_n\theta_1 \ldots \theta_{n-1}, BitString_{n-1})$

$synMatch_b(\texttt{\#<}\ var\ \texttt{>(}\ opts\ \texttt{)}, BitString) = ([var \mapsto val], BitString')$, if
`from_bits(`$BitString, opts$`)` $= (val, BitString')$
$synMatch_b(\texttt{\#<}\ val\ \texttt{>(}\ opts\ \texttt{)}, BitString) = (id, BitString')$, if
`from_bits(`$BitString, opts$`)` $= (val, BitString')$ and
val is an integer or float value

The new rule for *synMatch* simply composes the matching substitutions computed for the *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 $bitpat_{i+1}$, as some encoding options in $bitpat_{i+1}$ can be bound to a value in the previous matchings. The last call to $synMatch_b$ returns the empty bit string ϵ to express that after the complete matching all the bit string must be “consumed”. On the other hand, $synMatch_b$ uses the `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 $synMatch_b$ 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 `to_bits`:

$$(\mathsf{VAL\_BITS})\ \frac{\langle expr_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle expr_n, \theta\rangle \to vals_n}{\langle \texttt{\#\{}bitexpr_1, \ldots, bitexpr_n\texttt{\}\#}, \theta\rangle \to B_1\texttt{++}B_2\texttt{++}\ldots\texttt{++}B_n}$$

where $bitexpr_i = \texttt{\#<}\ expr_i\ \texttt{>(}\ opts_i\ \texttt{)}$, $\texttt{to\_bits(}\ vals_i\ ,\ opts_i\ \texttt{)} = B_i$ and $vals_i$ are integer or float values. Notice that the (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 $ICESC_\diamond$ 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 $\mathcal{E}$ an evaluation. Let T be a proof tree verifying $CESC \models_{P,T} \mathcal{E}$. Suppose also that the root of T (that is, $\mathcal{E}$) is $ICESC_f$-invalid.*

Then T contains an $ICESC_f$-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:

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

Suppose that every node $N \in S$ is $ICESC_f$-valid. Consider the tree T' obtained after replacing for each $N \in S$ the inference label (BFUN) by ($\mathsf{BFUN}_\mathcal{I}$) and removing the premises. Then $ICESC_f \models_{P,T'} \mathcal{E}$ and thus $\mathcal{E}$ is $ICESC_f$-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 $\mathcal{E}$ an evaluation. Let T be a proof tree verifying $CESC \models_{P,T} \mathcal{E}$. Suppose also that the root of T (that is, $\mathcal{E}$) is $ICESC_Z$-invalid. Suppose also that all the immediate* (BFUN)*-descendants of the root of T are $ICESC_Z$-valid.*[9]

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 $ICESC_f$. Then, exists at least one proof tree T' such that $CESC \models_{(P_{\langle e,\theta\rangle},T')} \langle r_{\mathtt{main}}, id\rangle \rightarrow |eval|$, 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 `main/0`.

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

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

Observing the `case` expression defining `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:

$$\frac{APT(T''')}{\langle r_{\mathtt{main}}, id\rangle \rightarrow |eval|}$$

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

[9]In the case of (BFUN) nodes $ICESC_Z$-validity corresponds to $ICESC_f$-validity

Lemma B.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, $\langle r_{\mathtt{main}}, id\rangle \to |eval|$, and thus the only reference to `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 `main`, $M \equiv \langle r_f, \theta\rangle \to eval$. To complete the theorem we must prove that M corresponds to a wrong function instance. This is done by considering M again in the *CESC* proof tree T'. There, the only child of M corresponds to $B\theta$ with B the `case` expression defining the body of function f. Then we have:

1. $ICESC_f \models_{(P,\mathcal{I})} \langle B\theta, \theta\rangle \to evals$ holds by Lemma B.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. $ICESC_f \not\models_{(P,\mathcal{I})} \langle r_f, \theta\rangle \to evals$ because $M \equiv \langle r_f, \theta\rangle \to evals$ is buggy.

Thus, according to Definition 5.5.1, $\langle r_f, \theta\rangle$ is a wrong function instance. $\square$

B.3 Correctness and Completeness of *APTZ*'s

The following Lemma shows the basic form of an APTZ.

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 $ICESC_Z$-invalid.*

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

Proof.

Observe that T must be of the form:

$$(\mathsf{BFUN})\,\dfrac{(\mathsf{CASE})\,\dfrac{\ldots \quad (\mathsf{C\_RESULT})\,\dfrac{\ldots}{\langle c\_result(r_i), \theta''\rangle \to evals}}{\langle \mathtt{case}^{r_c} \ldots, \theta\rangle \to evals}}{\langle r_f, \theta\rangle \to evals}$$

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 $APTZ$:

$$APTZ(T) =$$
$$APTZ\left((\mathsf{BFUN})\,\dfrac{(\mathsf{CASE})\,\dfrac{\ldots \quad (\mathsf{C\_RESULT})\,\dfrac{\ldots}{\langle c\_result(r_i), \theta''\rangle \to evals}}{\langle \mathtt{case}^{r_c} \ldots, \theta\rangle \to evals}}{\langle r_f, \theta\rangle \to evals} \right) =$$
$$\text{(using } APTZ_3\text{)}$$
$$APTZ\left((\mathsf{CASE})\,\dfrac{\ldots \quad (\mathsf{C\_RESULT})\,\dfrac{\ldots}{\langle c\_result(r_i), \theta''\rangle \to evals}}{\langle \mathtt{case}^{r_c} \ldots, \theta\rangle \to evals} \right) = \text{(using } APTZ_1\text{)}$$

$$(\mathsf{CASE})\,\dfrac{\ldots APTZ\left((\mathsf{C\_RESULT})\,\dfrac{\ldots}{\langle c\_result(r_i), \theta''\rangle \to evals} \right)}{\langle \mathtt{case}^{r_c} \ldots, \theta\rangle \to evals} = \text{(using } APTZ_1\text{)}$$

$$(\mathsf{CASE})\,\dfrac{\ldots \quad (\mathsf{C\_RESULT})\,\dfrac{\ldots}{\langle c\_result(r_i), \theta''\rangle \to evals)}}{\langle \mathtt{case}^{r_c} \ldots, \theta\rangle \to evals}$$

Then:

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 5.5.1

$$ICESC_f \nvDash_{(P,\mathcal{I})} \langle r_f, \theta\rangle \rightarrow v$$

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

$$ICESC_Z \nvDash_{(P,\mathcal{I})} \langle c\_result(r_i), \theta''\rangle \rightarrow v$$

   Thus, at least one of the premises of the ($\mathsf{CASE}_{\mathcal{I}}$) 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.

□

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. □

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 $N \equiv \langle succeeds(vals, r_i), \theta\rangle \rightarrow \theta'$. First observe that in the $APTZ$ each (SUCC)-node contains the same premises as in T', which must be of the form: $\langle patbind(r_i, vals), \theta\rangle \rightarrow \theta'$ (with $\theta' \neq \perp$), and $\langle guard(r_i), \theta''\rangle \rightarrow$ `'true'`. The validity of this premises depend on its membership to $\langle \mathcal{I}_{r_c}, \theta\rangle$ due to the definition of ($\mathsf{PATBIND}_{\mathcal{I}}$) and ($\mathsf{GUARD}_{\mathcal{I}}$).

N is $ICESC_Z$-invalid, that is $ICESC_Z \nvDash_{P,\mathcal{I}} \langle succeeds(vals, r_i), \theta\rangle \rightarrow \theta'$, and by the definition of ($\mathsf{SUCC}_{\mathcal{I}}$) in Section 5.2 it means $(\langle succeeds(r_i), \theta\rangle) \notin \langle \mathcal{I}_{r_c}, \theta\rangle$. The construction of $\langle \mathcal{I}_{r_c}, \theta\rangle$ indicates that the premises are not associated to the only *succeeds* evaluation in $\langle \mathcal{I}_{r_c}, \theta\rangle$, that is, i is not the expected successful branch of the `case` expression in the context θ. There are two possibilities:

1. The successful branch is some $j < i$. Then $\langle \mathcal{I}_{r_c}, \theta\rangle$ does not contain information about i (we do not require $\langle \mathcal{I}_{r_c}, \theta\rangle$ 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 $\langle \mathcal{I}_{r_c}, \theta\rangle$ includes evaluations $\langle patbind(r_i, vals), \theta\rangle \rightarrow \hat{\theta}'$, $\langle guard(r_i), \theta''\rangle \rightarrow val$, with $\theta'' \equiv \theta \uplus \hat{\theta}'$. But the conditions of failing branches require that $\hat{\theta_j} \equiv \perp$ or $val \equiv$ `'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, $\mathcal{I}$ its intended interpretation, and $e\theta \to eval$ an unexpected evaluation. Let T' be a proof tree such that $CESC \models_{(P_{\langle e,\theta\rangle},T')} \langle r_{\mathtt{main}}, id\rangle \to |eval|$. Let T'' be defined as $T'' \equiv APT(T')$, M be an $ICESC_f$ buggy node in T'', and T'_M the subtree of T' rooted by M. Finally, define a new tree T as $T \equiv APTZ(T'_M)$. Then T contains at least one $ICESC_Z$-buggy node N that verifies one of the following items:

1. *N is a* (BIND) *node of the form*

$$\langle < r_1,\ldots,r_n >, exprs, \theta\rangle \to \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$$

   Then, $\langle < r_1,\ldots,r_n >, exprs, \theta\rangle$ is a wrong binding instance.

2. *N a* (C_ARG) *node, $N \equiv \langle c\_arg(r_c), \theta\rangle \to evals$. Then, $\langle c\_argc(r_c), \theta\rangle$ is a wrong* `case` *argument instance.*

3. *N is a* (PATBIND)/(GUARD) *premise of a* (FAIL$_1$)/(FAIL$_2$) *node P, $P \equiv \langle fails(vals, r_i), \theta\rangle$, with r_i a reference to the ith branch of a* `case` *expression r_c, and such that N is a wrong* `case` *branch witness. Then $\langle r_i, \theta\rangle$ is an unexpected* `case` *failure instance, and in particular:*

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

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

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

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

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

5. *N is a* (C_RESULT) *node of the form $\langle c\_result(r_i), \theta\rangle \to evals$, and there is neither a* (C_ARG) *buggy node nor a wrong* `case` *branch witness for the same* `case` *expression. Then, $\langle c\_result(r_c), \theta\rangle \to evals$ is a wrong* `case` *result instance.*

Proof. In this proof the concepts of *valid*, *invalid* and *buggy* are assumed to be with respect $ICESC_Z$ unless the contrary is mentioned explicitly.

First observe that:

1. M, the root of T'_M, is $ICESC_f$ invalid (in fact it is $ICESC_f$ buggy by hypothesis).
2. M is (BFUN) node because $M \in T''$, T'' is an APT, and APTs only contain (BFUN) nodes.

Then by Lemma B.3 $T \equiv APTZ(T'_M)$ 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 B.4 and B.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, $N \equiv \langle < r_1,\ldots,r_n >, exprs, \theta\rangle \to \theta'$, with the substitution θ' defined as $\{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$. By the form of (BIND), in T' its only premise is of the form $M' \equiv \langle exprs, \theta\rangle \to < val_1,\ldots,val_n >$. 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 B.2 to deduce that M' is valid. Therefore we have:

(a) $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{expr}, \theta \rangle \to \theta'$ (M' valid)

(b) $\mathit{ICESC_Z} \not\models_{(P,\mathcal{I})} \langle \texttt{<}\ r_1, \ldots, r_n\ \texttt{>}, \mathit{exprs}, \theta \rangle \to \theta'$ (since M is buggy, it is invalid)

Then by Definition 5.5.2, $\langle \texttt{<}\ r_1, \ldots, r_n\ \texttt{>}, \mathit{exprs}, \theta \rangle$ is a *wrong binding instance*, which proves the item 1 of the theorem.

2. (C_ARG), that is $N \equiv \langle c\_arg(r_c), \theta \rangle \to \mathit{evals}$.

   N is buggy, that is invalid with valid children. N invalid means: $\mathit{ICESC_Z} \not\models_{(P,\mathcal{I})} \langle c\_arg(r_c), \theta \rangle \to \mathit{evals}$.

   The only child of (C_ARG) in *CESC* is $\langle \mathit{exprs}, \theta \rangle \to \mathit{evals}$ with *exprs* the expression defining the `case` expression argument. Since it is valid: $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{exprs}, \theta \rangle \to \mathit{evals}$.

   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 ($\mathsf{FAIL_1}$)/($\mathsf{FAIL_2}$) node P, with $P \equiv \langle \mathit{fails}(\mathit{vals}, r_i), \theta \rangle$ and r_i a reference to the ith branch of a `case` expression r_c. P is invalid (the path from N to the root contains only invalid nodes) and hence $\mathit{ICESC_Z} \not\models_{(P,\mathcal{I})} \langle \mathit{fails}(\mathit{vals}, r_i), \theta \rangle$. Since N is an wrong `case` branch witness, $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{suceeds}(\mathit{vals}, r_i), \theta \rangle \to \theta'$ for some θ'. Then:

   (a) If $N \equiv \langle \mathit{patbind}(r_i, \mathit{evals}), \theta \rangle \to \hat{\theta}''$ then

   - By the definition of (PATBIND) in *CESC*,

   $$\hat{\theta}'' \equiv \mathit{match}(\mathit{pats}\theta, \mathit{vals})$$

     with *pats* the pattern of the ith branch.
   - From the definition of $\mathit{ICESC_Z}$,

   $$\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{suceeds}(\mathit{vals}, r_i), \theta \rangle \to \theta'$$

     implies $(\langle \mathit{patbind}(r_i, \mathit{evals}), \theta \rangle \to \theta''') \in \langle \mathcal{I}_{r_c}, \theta \rangle$ with $\theta' \equiv \theta\ \uplus\ \theta'''$.
   - N is buggy, and hence it is invalid. This means that $\hat{\theta''} \neq \theta'''$ and $\mathit{match}(\mathit{pats}\theta, \mathit{vals}) \neq \theta'''$.

   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 5.4), as required by Definition 5.5.3b.

   (b) $N \equiv \langle \mathit{guard}(r_i), \theta \rangle \to \mathit{eval}$. In this case N must be a ($\mathsf{FAIL_2}$) node. Then:

   - $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{patbind}(r_i, \mathit{evals}), \theta \rangle \to \theta''$ and θ'' such that $\mathit{match}(\mathit{pats}\theta, \mathit{evals}) = \theta''$, *pats* the pattern of the ith 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 $\mathit{match}(\mathit{pats}\theta, \mathit{evals})$ (required by the side condition of (PATBIND)) is the same as the expected substitution. Moreover, $\theta' \equiv \theta\ \uplus\ \theta''$.
   - $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{guard}(r_i), \theta' \rangle \to \texttt{'true'}$ as a consequence of $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{suceeds}(\mathit{vals}, r_i), \theta \rangle \to \theta'$ (by the definition of $\langle \mathcal{I}_{r_c}, \theta \rangle$).
   - $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{guard}(r_i), \theta' \rangle \to \texttt{'false'}$ 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 5.3.1b) and analogously to the previous item is a first wrong branch (Definition 5.4), as required by Definition 5.5.3b.

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

5. (C_RESULT). Then $N \equiv \langle c\_result(r_i), \theta' \rangle \to \mathit{evals}$, 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: $\mathit{ICESC_Z} \models_{(P,\mathcal{I})} \langle \mathit{succeeds}(\mathit{evals}, r_i), \theta \rangle \to \theta'$.

Then, analogously to the (C_ARG) nodes: N buggy means

$$\begin{array}{ll} \mathit{ICESC}_Z \not\models_{(P,\mathcal{I})} \langle c\_\mathit{result}(r_i), \theta' \rangle \to \mathit{evals} & (N \text{ invalid}) \\ \mathit{ICESC}_Z \models_{(P,\mathcal{I})} \langle \mathit{exprs}_i \theta', \theta' \rangle \to \mathit{evals} & (\text{valid child}) \end{array}$$

which corresponds to the definition of $\langle c\_\mathit{result}(r_c), \theta \rangle \to \mathit{evals}$ is a wrong `case` result instance (Definition 5.5.3c).

$\square$