A calculus for zoom debugging sequential Erlang programs*

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

Technical Report 07/13

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

July, 2013

*Research supported by MICINN Spanish project *StrongSoft* (TIN2012-39391-C04-04) and Comunidad de Madrid program *PROMETIDOS* (S2009/TIC-1465).

Abstract

We present here the evaluation semantics for sequential Erlang programs specially developed to be used for “zoom debugging.” We first introduce the syntax of the programs we want to evaluate and then present the different evaluations that take place in the calculus. The rest of the sections describes the calculus for references, values, and exceptions.

Keywords: Sequential Erlang, semantics.

1 Syntax

$$
\begin{array}{lll}
\text{fname} & ::= & \text{Atom} \texttt{ / } \text{Integer} \\
\text{lit} & ::= & \text{Atom} \mid \text{Integer} \mid \text{Float} \mid \text{Char} \mid \text{String} \mid \text{BitString} \mid \texttt{[ ]} \\
\text{fun} & ::= & \texttt{fun(}\text{var}_1 \texttt{ , } \ldots \texttt{ , } \text{var}_n\texttt{) ->} \text{ exprs} \\
\text{clause} & ::= & \text{pats } \texttt{when} \text{ exprs}_1 \texttt{ -> } \text{exprs}_2 \\
\text{pat} & ::= & \text{var} \mid \text{lit} \mid \texttt{[} \text{ pats} \mid \text{pats } \texttt{]} \mid \{ \text{ pats}_1 \texttt{ , } \ldots \texttt{ , } \text{pats}_n \ \} \mid \text{var} \texttt{ = } \text{pats} \\
 & & \mid \texttt{\#\{} \text{ bitpat}_1, \ldots, \text{bitpat}_n \ \texttt{\}\#} \mid \text{var} \texttt{ = } \text{pats} \\
\text{bitpat} & ::= & \texttt{\#<} \text{ pat\_b } \texttt{>(} \ \mathit{opts} \ \texttt{)} \\
\text{pat\_b} & ::= & \text{var} \mid \text{Integer} \mid \text{Float} \\
\text{pats} & ::= & \text{pat} \mid \texttt{<} \text{ pat} \texttt{, } \ldots \texttt{ , } \text{pat } \texttt{>} \\
\text{exprs} & ::= & \text{expr} \mid \texttt{<} \text{ expr} \texttt{, } \ldots \texttt{ , } \text{expr } \texttt{>} \\
\text{expr} & ::= & \text{var} \mid \text{fname} \mid \text{fun} \mid \texttt{[} \text{ exprs} \mid \text{exprs } \texttt{]} \mid \{ \text{ exprs}_1 \texttt{ , } \ldots \texttt{ , } \text{exprs}_n \ \} \\
 & & \mid \texttt{\#\{} \text{ bitexpr}_1, \ldots, \text{bitexpr}_n \ \texttt{\}\#} \\
 & & \mid \texttt{let} \text{ vars} \texttt{ = } \text{exprs}_1 \texttt{ in } \text{exprs}_2 \\
 & & \mid \texttt{letrec} \text{ fname}_1 \texttt{ = } \text{fun}_1 \ldots \text{fname}_n \texttt{ = } \text{fun}_n \texttt{ in } \text{exprs} \\
 & & \mid \texttt{apply} \text{ exprs } \texttt{(} \text{ exprs}_1 \texttt{ , } \ldots \texttt{ , } \text{exprs}_n \texttt{)} \\
 & & \mid \texttt{call} \text{ exprs}_{n+1}\texttt{:}\text{exprs}_{n+2} \texttt{ (} \text{ exprs}_1 \texttt{ , } \ldots \texttt{ , } \text{exprs}_n \texttt{)} \\
 & & \mid \texttt{primop} \text{ Atom } \texttt{(} \text{ exprs}_1 \texttt{ , } \ldots \texttt{ , } \text{exprs}_n \texttt{)} \\
 & & \mid \texttt{try} \text{ exprs}_1 \texttt{ of <}\text{var}_1 \texttt{ , } \ldots \texttt{ , } \text{var}_n\texttt{> -> } \text{exprs}_2 \\
 & & \quad \texttt{catch <}\text{var'}_1 \texttt{ , } \ldots \texttt{ , } \text{var'}_m\texttt{> -> } \text{exprs}_3 \\
 & & \mid \texttt{case} \text{ exprs } \texttt{of} \text{ clause}_1 \ldots \text{clause}_n \texttt{ end} \\
 & & \mid \texttt{do} \text{ exprs}_1 \text{ exprs}_2 \mid \texttt{catch} \text{ exprs} \\
\text{bitexpr} & ::= & \texttt{\#<} \text{ expr } \texttt{>(} \ \mathit{opts} \ \texttt{)} \\
\xi & ::= & \text{Exception}(\overline{\mathit{val}_m}) \\
\text{val} & ::= & \text{lit} \mid \text{fname} \mid \text{fun} \mid \texttt{[} \text{ vals} \mid \text{vals } \texttt{]} \mid \{\text{vals}_1 \texttt{ , } \ldots \texttt{ , } \text{vals}_n\} \\
\text{eval} & ::= & \text{lit} \mid \text{fname} \mid \text{fun} \mid \texttt{[} \text{ evals} \mid \text{evals } \texttt{]} \mid \{\text{evals}_1 \texttt{ , } \ldots \texttt{ , } \text{evals}_n\} \mid \xi \\
\text{vals} & ::= & \text{val} \mid \texttt{<} \text{ val} \texttt{, } \ldots \texttt{ , } \text{val } \texttt{>} \\
\text{evals} & ::= & \text{eval} \mid \texttt{<} \text{ eval} \texttt{, } \ldots \texttt{ , } \text{eval } \texttt{>} \\
\text{vars} & ::= & \text{var} \mid \texttt{<} \text{ var} \texttt{, } \ldots \texttt{ , } \text{var } \texttt{>}
\end{array}
$$

Figure 1: Core Erlang's Syntax

We present in this section the syntax of Sequential Core Erlang [1, 2]. The intermediate language Core Erlang 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 1 presents its syntax after removing the parts corresponding to concurrent operations, i.e. `receive`. The most significant element in the syntax is the expression (*expr*). Besides variables, function names, lambda abstractions, lists, and tuples, expressions can be:

- `let`: its value is the one resulting from evaluating $exprs_2$ where *vars* are bound to the value of $exprs_1$.
- `letrec`: similar to the previous expression but a sequence of function declarations (*fname* = *fun*) is defined.
- `apply`: applies *exprs* (defined in the current module) to a number of arguments.
- `call`: similar to the previous expression but the function applied is the one defined by $exprs_{n+2}$ in the module defined by $exprs_{n+1}$. Both expressions should be evaluated to an atom. For example, the expression `call mergesort:comp('a','b')` considering the previous program.
- `primop`: application of built-in functions mainly used to report errors. A typical example is the report of a matching failure in a `case` expresion: `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 expections contain, usually the kind of exception and information about the reason.

- `case`: a pattern-matching expression. Its value corresponds to the one in the body of the first clause whose pattern matches the value of *exprs* and whose guard evaluates to `true`. There is always at least one clause fulfilling these conditions, as we explain below.

Moreover, 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.

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. 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. 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.

Finally, values represent the possible results of an expressions 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* `<` $x_1, \ldots, x_n$ `>` of zero or more values. Sequences, which were added in Core Erlang to simplify the generation of efficient code and to allow certain optimizations to be performed at the core level [1], 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 posibly containing exceptions and sequences of values without expections, respectively.

2 Preliminaries

The set of variables occurring in an expression e is denoted by $var(e)$. The notation $locvar(r)$, with r a reference to either a function clause or to a lambda-expression to indicate 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 $\texttt{fun}(var_1, \ldots, var_n)$ `->` *expr* represents the context variables of r_λ, that is $ctx(r_\lambda) = var(expr) - (\{var_1, \ldots, var_n\} \cup locvar(r_\lambda))$. Observe that the set of context variables for a function clause is always empty, but the body of lambda-expressions defined inside function clauses can include local variables/arguments of the function in their bodies.

The calculus uses evaluations of the form:

$\langle guard(r_b), \theta\rangle \to val$, 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 expression 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 expression is evaluated to *vals* and the context is denoted by the substitution θ.

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

$\langle exprs, \theta\rangle \to vals$**,** where *exprs* is the expression being evaluated, θ is a substitution, and *vals* is the value obtained for the expression.

$\langle r, \theta\rangle \to \theta'$**,** where r is a reference to a lambda-expression or a function, θ is a substitution, and θ' is the a new substitution obtained by extending θ. We also use the notation $\langle r, \theta\rangle \to \theta'$ when r references to a function and we want to indicate that the ith clause has been used.

$\langle r_c, vals, \theta\rangle \to vals'$**,** which computes the value $vals'$ obtained when evaluating a case expression, where r_c is the reference to the case expression, *vals* is the value obtained when computing the expression on the top of the case, and *vals* is the context where the case is evaluated.

We assume in all cases that all the variables appearing in the first element of the tuples are in the domain of θ, and the existence of a global environment ρ which is initially empty and is extended by adding the functions defined by the `letrec` operator. The notation $CESC \models_{(P,T)} \mathcal{R}$, where $\mathcal{R}$ is an evaluation, is employed to indicate that $\mathcal{R}$ can be proven w.r.t. the program P with the proof tree T in *CESC*, while $CESC \not\models_P \mathcal{R}$ indicates that $\mathcal{R}$ cannot be proven in *CESC* with respect to the program P.

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 propagating exceptions.

3 Calculus 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. 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 $\perp$ when the matching is not possible:

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

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

The function match is in charge of performing syntactic matching as follows:
$match(\texttt{<}\ pat_1, \ldots, pat_n\ \texttt{>}, \texttt{<}\ val_1, \ldots, val_n\ \texttt{>}) = \theta_1 \uplus \ldots \uplus \theta_n$ 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 $\theta \equiv synMatch(pat, val)$
$synMatch(\#\{bitpat_1, \ldots, bitpat_n\}\#, BitString) = \theta_1 \uplus \ldots \uplus \theta_n$, where
$(\theta_1, BitString_1) \equiv synMatch_b(bitpat_1, BitString)$
$(\theta_2, BitString_2) \equiv synMatch_b(bitpat_2\theta_1, BitString_1)$
...
$(\theta_n, \epsilon) \equiv synMatch_b(bitpat_n\theta_1 \ldots \theta_{n-1}, BitString_{n-1})$
$synMatch(pat, val) = \perp$ otherwise

$synMatch_b(\#<var>(opts), BitString) = ([var \mapsto val], BitString')$, if
$from\_bits(BitString, opts) = (val, BitString')$
$synMatch_b(\#<val>(opts), BitString) = (id, BitString')$, if
$from\_bits(BitString, opts) = (val, BitString')$ and $val \in Integer \cup Float$

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

$$(\mathsf{GUARD})\frac{\langle exprs\theta, \theta\rangle \rightarrow eval}{\langle guard(r_b), \theta\rangle \rightarrow eval}$$

where r_b is a reference to *pats* `when` *exprs* `->` *exprs'*

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

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

where r_b is a reference to *pats* `when` *exprs* `->` *exprs'*

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

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

with $\theta' \neq \bot$, $\theta'' \equiv \theta \uplus \theta'$ and r_b a reference to *pats* `when` *exprs* `->` *exprs'*.

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:

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

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

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

$$(\mathsf{BIND})\frac{\langle exprs, \theta\rangle \rightarrow \texttt{<}\ val_1, \ldots, val_n\ \texttt{>}}{\langle \texttt{<}\ r_1, \ldots, r_n\ \texttt{>}, exprs, \theta\rangle \rightarrow \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}}$$

with $r_1, \ldots, r_n$ references to variables $var_1, \ldots, var_n$.

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

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

where r_f references either to a function `f = fun`$(var_1, \ldots, var_n)$ `->` *expr*, or to a lambda expression defined as `fun`$(var_1, \ldots, var_n)$ `->` *expr*

4 Calculus for values

We present in this section the inference rules for obtaining values from expressions. The basic rule is (VAL), which states that values are evaluated to themselves:

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

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

$$(\mathsf{SEQ})\frac{\langle expr_1, \theta\rangle \rightarrow val_1 \quad \ldots \quad \langle expr_n, \theta\rangle \rightarrow val_n}{\langle \texttt{<}\ expr_1, \ldots, expr_n\ \texttt{>}, \theta\rangle \rightarrow \texttt{<}\ val_1, \ldots, val_n\ \texttt{>}}$$

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

$$(\mathsf{TUP})\frac{\langle exprs_1, \theta\rangle \rightarrow vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \rightarrow vals_n}{\langle \{exprs_1, \ldots, exprs_n\}, \theta\rangle \rightarrow \{vals_1, \ldots, vals_n\}}$$

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

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:

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

where $\theta'' \equiv \theta \uplus \theta'$ and r_c is a reference to a `case` statement defined as

$\mathtt{case}\ \mathit{exprs}\ \mathtt{of}\quad \mathit{pats}_1\ \mathtt{when}\ \mathit{exprs}'_1\ \mathtt{->}^{r_1}\ \mathit{exprs}''_1$
$\qquad\qquad\qquad\ \dots$
$\qquad\qquad\qquad\ \mathit{pats}_n\ \mathtt{when}\ \mathit{exprs}'_n\ \mathtt{->}^{r_n}\ \mathit{exprs}''_n\ \mathtt{end}$

and the labels $r_1, \dots, r_n$ 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:

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

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:

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

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:

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

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

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

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

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 $\theta' \equiv \{\mathit{var}_1 \mapsto \mathit{vals}_1, \dots, \mathit{var}_n \mapsto \mathit{vals}_n\}$.

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

$$(\mathsf{CALL\_EVAL})\ \frac{\begin{array}{c}\langle \mathit{exprs}_{n+1}, \theta\rangle \to \mathtt{'erlang'} \qquad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \mathit{Atom}_2 \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{val}_1 \qquad \dots \qquad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{val}_n \\ \mathit{eval}(\mathit{Atom}_2, \mathit{val}_1, \dots, \mathit{val}_n) = \mathit{vals}\end{array}}{\langle \mathtt{call}\ \mathit{exprs}_{n+1}\mathtt{:}\mathit{exprs}_{n+2}(\mathit{exprs}_1, \dots, \mathit{exprs}_n), \theta\rangle \to \mathit{vals}}$$

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 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 $\texttt{fun(}var_1\texttt{,}\ldots\texttt{,}var_n\texttt{) ->}\ exprs'$, 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 current module $r.mod$ as $Atom/n$ `=` `fun (`var_1 `,` $\ldots$ `,` var_n`)` `->` $expr$, 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 an auxiliary function *eval*, which returns the value Erlang would compute:

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

The rule (TRY_1) evaluates a `try` expression when no exceptions are thrown. It just evaluates the expressions and continues with the expression in the body:

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

with $\theta' \equiv match(\texttt{<}\ var_1, \ldots, var_n\ \texttt{>}, vals')$ 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:

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

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

The rule ($\mathsf{VAL\_BITS}$) is used to evaluate bit strings. It 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 $bitexpr_i$ = `#<` $expr_i$ `>(` $opts_i$ `)`, `to_bits(` $vals_i$ `,` $opts_i$ `)` $= B_i$ and $vals_i$ are integer or float values.

Finally, the rules (DO) and (CATCH) expressions, simply reuse previous constructions, since they are syntactic sugar [2]:

$$(\mathsf{DO})\ \frac{\langle \texttt{let}\ \_ = exprs_1\ \texttt{in}\ exprs_2, \theta\rangle \to vals}{\langle \texttt{do}\ exprs_1\ exprs_2, \theta\rangle \to vals}$$

$$(\text{CATCH})\ \frac{\langle expr', \theta\rangle \to vals}{\langle \texttt{catch}\ \ exprs, \theta\rangle \to vals}$$

$$\text{with } expr' \equiv \left\{ \begin{array}{l} \texttt{try}\ exprs\ \texttt{of < } var_1, \ldots, var_n \texttt{ > ->} \\ \quad \texttt{< } var_1, \ldots, var_n \texttt{ >} \\ \texttt{catch < } var_{n+1}, var_{n+2}, var_{n+3} \texttt{ > ->} \\ \quad \texttt{case}\ var_{n+1}\ \texttt{of} \\ \qquad \texttt{`throw` when `true` ->} \\ \qquad\quad var_{n+2} \\ \qquad \texttt{`exit` when `true` ->} \\ \qquad\quad \{\texttt{`EXIT`}, var_{n+2}\} \\ \qquad \texttt{`error` when `true` ->} \\ \qquad\quad \{\texttt{`EXIT`}, \{var_{n+2}, \texttt{primop exc\_trace}(var_{\texttt{n+3}})\}\} \\ \quad \texttt{end} \end{array} \right.$$

5 Calculus for exceptions

We present in this section the inference rules to generate and propagate exceptions. The rule (SEQ_E) propagates an exception thrown inside a sequence:

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

Similarly, the rule (TUP_E) propagates an exception thrown inside a tuple:

$$(\text{TUP\_E})\ \frac{\langle expr_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle expr_i, \theta\rangle \to vals_i \qquad \langle expr_{i+1}, \theta\rangle \to \xi}{\langle \{exprs_1, \ldots, exprs_n\}, \theta\rangle \to \xi}$$

We use the rules (LIST_E_1) and (LIST_E_2) to propagate an exception thrown on the first or second component of a list, respectively:

$$(\text{LIST\_E}_1)\ \frac{\langle exprs_1, \theta\rangle \to \xi}{\langle [exprs_1|exprs_2], \theta\rangle \to \xi}$$

$$(\text{LIST\_E}_2)\ \frac{\langle exprs_1, \theta\rangle \to vals_1 \qquad \langle exprs_2, \theta\rangle \to \xi}{\langle [exprs_1|exprs_2], \theta\rangle \to \xi}$$

The rule (LET_E) propagates an exception thrown in the expression:

$$(\text{LET\_E})\ \frac{\langle exprs_1, \theta\rangle \to \xi}{\langle \texttt{let < } var_1, \ldots, var_n \texttt{ > = } exprs_1\ \texttt{in}\ exprs, \theta\rangle \to \xi}$$

The rules (APPLY_E_1) and (APPLY_E_2) indicate that an exception is thrown if either the function or the arguments throw an exception:

$$(\text{APPLY\_E}_1)\ \frac{\langle exprs, \theta\rangle \to \xi}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to \xi}$$

$$(\text{APPLY\_E}_2)\ \frac{\langle exprs, \theta\rangle \to vals \qquad \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_i, \theta\rangle \to vals_i \qquad \langle exprs_{i+1}, \theta\rangle \to \xi}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to \xi}$$

The rule (APPLY_E_3) throws a `bad_function` exception when the function being applied has not been defined:

$$(\text{APPLY\_E}_3)\ \frac{\langle exprs, \theta\rangle \to vals \qquad \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n}{\langle \texttt{apply}^r\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to Except(\texttt{error}, \texttt{bad\_function}, \ldots)}$$

if *vals* is neither a lambda abstraction nor an *fname* defined in ρ or in $r.mod$.

The rules ($\mathsf{APPLY\_E_4}$) and ($\mathsf{APPLY\_E_5}$) throw an exception indicating that the number of arguments is different from the number of parameters. The former is in charge of lambda abstractions while the latter is in charge of defined functions:

$$(\mathsf{APPLY\_E_4})\ \frac{\begin{array}{c}\langle exprs, \theta\rangle \to \texttt{fun(}var_1\texttt{,}\ldots\texttt{,}var_m\texttt{) ->}\ \ exprs' \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n\end{array}}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to Except(\texttt{error}, \texttt{anon called with m args}, \ldots)}$$

if $m \neq n$

$$(\mathsf{APPLY\_E_5})\ \frac{\langle exprs, \theta\rangle \to Atom/m \quad \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \to Except(\texttt{error}, \texttt{called with n args}, \ldots)}$$

if $m \neq n$

The rules ($\mathsf{CALL\_E_1}$), ($\mathsf{CALL\_E_2}$), and ($\mathsf{CALL\_E_3}$) throw an exception when either the module name, the function name, or any of the arguments are evaluated to an exception:

$$(\mathsf{CALL\_E_1})\ \frac{\langle exprs_{n+1}, \theta\rangle \to \xi}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1, \ldots, exprs_n), \theta\rangle \to \xi}$$

$$(\mathsf{CALL\_E_2})\ \frac{\langle exprs_{n+1}, \theta\rangle \to vals_1 \quad \langle exprs_{n+2}, \theta\rangle \to \xi}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1, \ldots, exprs_n), \theta\rangle \to \xi}$$

$$(\mathsf{CALL\_E_3})\ \frac{\begin{array}{c}\langle exprs_{n+1}, \theta\rangle \to vals'_1 \qquad\qquad \langle exprs_{n+2}, \theta\rangle \to vals'_2 \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_i, \theta\rangle \to vals_i \quad \langle exprs_{i+1}, \theta\rangle \to \xi\end{array}}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1, \ldots, exprs_n), \theta\rangle \to \xi}$$

The rules ($\mathsf{CALL\_E_4}$) and ($\mathsf{CALL\_E_5}$) throw a `bad_argument` exception when either the module or the function is not an atom:

$$(\mathsf{CALL\_E_4})\ \frac{\begin{array}{c}\langle exprs_{n+1}, \theta\rangle \to vals'_1 \quad \langle exprs_{n+2}, \theta\rangle \to vals'_2 \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n\end{array}}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1, \ldots, exprs_n), \theta\rangle \to Exception(\texttt{error}, \texttt{bad\_argument}, \ldots)}$$

if $vals'_1$ is not an atom

$$(\mathsf{CALL\_E_5})\ \frac{\begin{array}{c}\langle exprs_{n+1}, \theta\rangle \to Atom_1 \quad \langle exprs_{n+2}, \theta\rangle \to vals'_2 \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n\end{array}}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1, \ldots, exprs_n), \theta\rangle \to Exception(\texttt{error}, \texttt{bad\_argument}, \ldots)}$$

if $vals'_2$ is not an atom

The rule ($\mathsf{CALL\_E_6}$) throws an `undefined_function` exception when the function is not defined in the specified module:

$$(\mathsf{CALL\_E_6})\ \frac{\begin{array}{c}\langle exprs_{n+1}, \theta\rangle \to Atom_1 \quad \langle exprs_{n+2}, \theta\rangle \to Atom_2 \\ \langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \to vals_n\end{array}}{\langle \texttt{call}\ exprs_{n+1}\texttt{:}exprs_{n+2}(exprs_1, \ldots, exprs_n), \theta\rangle \to Exception(\texttt{error}, \texttt{undefined\_function}, \ldots)}$$

if the function $Atom_2/n$ is not defined and exported in module $Atom_1$

The rule ($\mathsf{PRIMOP\_E}$) propagates the exceptions thrown by its arguments:

$$(\mathsf{PRIMOP\_E})\ \frac{\langle exprs_1, \theta\rangle \to vals_1 \quad \ldots \quad \langle exprs_i, \theta\rangle \to vals_i \quad \langle exprs_{i+1}, \theta\rangle \to \xi}{\langle \texttt{primop}\ Atom(exprs_1, \ldots, exprs_n), \theta\rangle \to \xi}$$

The rule ($\mathsf{CASE\_E}$) propagates an exception thrown while evaluating the expression:

$$(\mathsf{CASE\_E_1})\ \frac{\langle exprs_1, \theta\rangle \to \xi}{\langle \texttt{case}\ exprs_1\ \texttt{of}\ \overline{pats_n\ \texttt{when}\ exprs'_n\ \texttt{->}\ exprs_n}\ \texttt{end}, \theta\rangle \to \xi}$$

References

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

[2] 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`.