A calculus for sequential Erlang programs*

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

Technical Report 03/13

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

April, 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. We first introduce the syntax of the programs we want to evaluate and then present the calculus in two steps. Once the syntax has been presented, we describe the rules for computing correct values, and then we present the rules dealing with errors and throwing exceptions.

Keywords: Sequential Erlang, semantics.

1 Syntax

We present in this section the syntax of Sequential Erlang. Besides the standard syntactical categories we have added eval, which stands for a *correct* value, and hence rules out exceptions, and evals, which stands for a sequence of eval.

fname	::=	Atom / Integer
lit	::=	Atom | Integer | Float | Char | String | `[ ]`
fun	::=	`fun`(var_1 , . . . , var_n) `->` exprs
clause	::=	pats `when` exprs_1 `->` exprs_2
pat	::=	var | lit | `[` pats | pats `]` | { pats_1 , . . . , pats_n } | var `=` pats
pats	::=	pat | < pat, . . . , pat >
exprs	::=	expr | < expr, . . . , expr >
expr	::=	var | fname | fun | `[` exprs | exprs `]` | { exprs_1 , . . . , exprs_n } | `let` vars `=` exprs_1 `in` exprs_2 | `letrec` fname_1 `=` fun_1 . . . fname_n `=` fun_n `in` exprs | `apply` exprs (exprs_1 , . . . , exprs_n) | `call` exprs_{n+1}`:`exprs_{n+2} (exprs_1 , . . . , exprs_n) | `primop` Atom (exprs_1 , . . . , exprs_n) | `try` exprs_1 `of` < var_1 , . . . , var_n > `->` exprs_2 `catch` < var'_1 , . . . , var'_m > `->` exprs_3 | `case` exprs `of` clause_1 . . . clause_n `end` | `do` exprs_1 exprs_2 | `catch` exprs
ξ	::=	Exception($\overline{val_m}$)
val	::=	lit | fname | fun | `[` vals | vals `]` | {vals_1 , . . . , vals_n}
eval	::=	lit | fname | fun | `[` evals | evals `]` | {evals_1 , . . . , evals_n} | ξ
vals	::=	val | < val, . . . , val >
evals	::=	eval | < eval, . . . , eval >
vars	::=	var | < var, . . . , var >

2 Calculus for values

We present in this section the inference rules used to obtain values in Erlang. The basic rule in our calculus is (VAL), which states that values can be evaluated to themselves:

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

The rule (SEQ) is in charge of evaluating the expressions inside a sequence:

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

Similarly, the rule (TUP) evaluates the expressions inside a tuple:

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

The rule (LIST) evaluates the first expression in a list and, if it reaches a value, then evaluates the second expression, returning the second value:

$$(\mathsf{LIST})\,\frac{\langle exprs_1, \theta\rangle \rightarrow vals_1 \qquad \langle exprs_2, \theta\rangle \rightarrow vals_2}{\langle [exprs_1 | exprs_2], \theta\rangle \rightarrow [vals_1 | vals_2]}$$

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

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

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

The rule (LETREC) extends the environment ρ to add the functions. We assume that all the function names are different.

$$(\mathsf{LETREC})\,\frac{\langle exprs, \theta\rangle \rightarrow vals}{\langle \texttt{letrec}\ fname_1\texttt{=}fun_1 \ldots fname_n\texttt{=}fun_n\ \texttt{in}\ exprs, \theta\rangle \rightarrow vals}$$

where ρ has been extended with $[\overline{fname_n \mapsto fun_n}]$

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:

$$(\mathsf{APPLY}_1)\,\frac{\begin{array}{c}\langle exprs, \theta\rangle \rightarrow r_\lambda \\ \langle exprs_1, \theta\rangle \rightarrow val_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \rightarrow val_n \\ \langle r_\lambda, \theta'\rangle \rightarrow vals\end{array}}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \rightarrow vals}$$

where r_λ references a lambda abstraction, $r_\lambda \equiv \texttt{fun(}var_1\texttt{,} \ldots \texttt{,}var_n\texttt{) ->}\ exprs'$, and $\theta' \equiv \theta[\overline{var_n \mapsto val_n}]$

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:

$$(\mathsf{APPLY}_2)\,\frac{\begin{array}{c}\langle exprs, \theta\rangle \rightarrow Atom/n \\ \langle exprs_1, \theta\rangle \rightarrow val_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \rightarrow val_n \\ \langle exprs'\theta', \theta'\rangle \rightarrow vals'\end{array}}{\langle \texttt{apply}\ exprs(exprs_1, \ldots, exprs_n), \theta\rangle \rightarrow vals'}$$

if $\rho(Atom/n) = \texttt{fun(}\ var_1\ \texttt{, ... ,}\ var_n\ \texttt{) ->}\ exprs$' and $\theta' \equiv \theta[\overline{var_n \mapsto val_n}]$

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:

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

where $1 \le i \le m$, $Atom/n \notin dom(\rho)$, $Atom/n$ is a function defined in the module $r.mod$, and r_f its reference, which must be of the form:

$Atom/n$ `= fun (`var_1 `, ... ,` var_n`) -> case` $exprs$ `of` $clause_1 \ldots clause_m$ `end`

and $\theta' \equiv [\overline{var_n \mapsto val_n}]$

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

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

where $1 \le i \le m$, $Atom_2/n$ is a function defined in the $Atom_1$ module ($Atom_1$ must be different from the built-in module `erlang`), r_f its reference, which must be of the form:

`fun (` var_1 `, ... ,` var_n `) -> case` $exprs$ `of` $clause_1 \ldots clause_m$ `end`

and $\theta' \equiv [\overline{var_n \mapsto val_n}]$

In the same way, the (CALL_EVAL) rule is in charge of evaluating built-in functions:

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

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

The (BFUN) rule evaluates a reference to a function, given a substitution binding all its arguments. This is accomplished by applying the substitution to the body of the function (with notation $exprs\theta$) and then evaluating it. This rule takes advantage of the fact that all Erlang functions are translated to Core Erlang as a `case`-expression distinguishing the different clauses. Since the evaluation of this `case`-expression provides the branch used to obtain the final value (i.e. the i labeling the evaluation), we are able to keep the clause used to evaluate the function:

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

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

The rule (λ) follows the ideas shown for (BFUN) to evaluate a lambda-expression. It uses the body of the referenced function to obtain the final value:

$$(\lambda)\ \frac{\langle expr\theta, \theta\rangle \rightarrow vals}{\langle r_\lambda, \theta\rangle \rightarrow vals}$$

where r_λ references to `fun`$(var_1, \ldots, var_n)$ `->` $expr$

The rule (PRIMOP) evaluates Erlang predefined functions by using an auxiliary function `eval`, which returns the value Erlang would compute:

$$\text{(PRIMOP)}\ \frac{\langle exprs_1, \theta\rangle \rightarrow val_1 \quad \ldots \quad \langle exprs_n, \theta\rangle \rightarrow val_n \quad eval(Atom, val_1, \ldots, val_n) = vals'}{\langle \texttt{primop}\ Atom(exprs_1, \ldots, exprs_n), \theta\rangle \rightarrow vals'}$$

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:

$$(\text{TRY}_1)\ \frac{\langle exprs_1, \theta\rangle \rightarrow vals' \quad \langle exprs_2\theta', \theta'\rangle \rightarrow vals}{\langle \texttt{try}\ exprs_1\ \texttt{of <}\overline{var_n}\texttt{> ->}\ exprs_2\ \texttt{catch <}\overline{var'_m}\texttt{> ->}\ exprs_3, \theta\rangle \rightarrow vals}$$

with $\theta' \equiv \theta \uplus matchs(\texttt{<}\overline{var_n}\texttt{>}, vals')$ and $vals'$ is not an exception

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:

$$(\text{TRY}_2)\ \frac{\langle exprs_1, \theta\rangle \rightarrow Except(\overline{val_m}) \quad \langle expr_3\theta', \theta'\rangle \rightarrow vals}{\langle \texttt{try}\ exprs_1\ \texttt{of <}\overline{var_n}\texttt{> ->}\ exprs_2\ \texttt{catch <}\overline{var'_m}\texttt{> ->}\ exprs_3, \theta\rangle \rightarrow vals}$$

with $\theta' \equiv \theta \uplus [\overline{var'_m \mapsto val_m}]$

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

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

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

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

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

Finally, the rules (DO) and (CATCH) expressions, simply reuse previous constructions, since they are syntactic sugar [1]:

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

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

with $expr' \equiv \{$

```
try exprs of < var_1, ..., var_n > ->
  < var_1, ..., var_n >
catch < var_{n+1}, var_{n+2}, var_{n+3} > ->
  case var_{n+1} of
    'throw' when 'true' ->
      var_{n+2}
    'exit' when 'true' ->
      {'EXIT', var_{n+2}}
    'error' when 'true' ->
      {'EXIT', {var_{n+2}, primop exc_trace(var_{n+3})}}
  end
```

3 Calculus for exceptions

We present in this section the inference rules to generate and propagate exceptions.

The rule (VAR_E) indicates that a variable cannot be evaluated:

$$(\mathsf{VAR\_E})\ \frac{}{\langle var, \theta\rangle \to Exception(\texttt{error}, \texttt{unbound\_var}, \ldots)}$$

The rule (SEQ_E) propagates an exception thrown inside a sequence:

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

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

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

We use the rules ($\mathsf{LIST\_E_1}$) and ($\mathsf{LIST\_E_2}$) to propagate an exception thrown on the first or second component of a list, respectively:

$$(\mathsf{LIST\_E_1})\ \frac{\langle exprs_1, \theta\rangle \to \xi}{\langle [exprs_1|exprs_2], \theta\rangle \to \xi}$$

$$(\mathsf{LIST\_E_2})\ \frac{\langle exprs_1, \theta\rangle \to vals_1 \quad \langle exprs_2, \theta\rangle \to \xi}{\langle [exprs_1|exprs_2], \theta\rangle \to \xi}$$

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

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

The rules ($\mathsf{APPLY\_E_1}$) and ($\mathsf{APPLY\_E_2}$) indicate that an exception is thrown if either the function or the arguments throw an exception:

$$(\mathsf{APPLY\_E_1})\ \frac{\langle \mathit{exprs}, \theta\rangle \to \xi}{\langle \mathtt{apply}\ \mathit{exprs}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \xi}$$

$$(\mathsf{APPLY\_E_2})\ \frac{\begin{array}{c}\langle \mathit{exprs}, \theta\rangle \to \mathit{vals} \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_i, \theta\rangle \to \mathit{vals}_i \\ \langle \mathit{exprs}_{i+1}, \theta\rangle \to \xi\end{array}}{\langle \mathtt{apply}\ \mathit{exprs}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \xi}$$

The rule ($\mathsf{APPLY\_E_3}$) throws a `bad_function` exception when the function being applied has not been defined:

$$(\mathsf{APPLY\_E_3})\ \frac{\begin{array}{c}\langle \mathit{exprs}, \theta\rangle \to \mathit{vals} \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n\end{array}}{\langle \mathtt{apply}^r\ \mathit{exprs}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{Except}(\mathtt{error}, \mathtt{bad\_function}, \ldots)}$$

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

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:

$$(\mathsf{APPLY\_E_4})\ \frac{\begin{array}{c}\langle \mathit{exprs}, \theta\rangle \to \mathtt{fun(}\mathit{var}_1\mathtt{,}\ldots\mathtt{,}\mathit{var}_m\mathtt{)}\ \mathtt{->}\ \ \mathit{exprs}' \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n\end{array}}{\langle \mathtt{apply}\ \mathit{exprs}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{Except}(\mathtt{error}, \mathtt{anon\ called\ with\ m\ args}, \ldots)}$$

if $m \neq n$

$$(\mathsf{APPLY\_E_5})\ \frac{\langle \mathit{exprs}, \theta\rangle \to \mathit{Atom}/m \quad \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n}{\langle \mathtt{apply}\ \mathit{exprs}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{Except}(\mathtt{error}, \mathtt{called\ with\ n\ args}, \ldots)}$$

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:

$$(\mathsf{CALL\_E_1})\ \frac{\langle \mathit{exprs}_{n+1}, \theta\rangle \to \xi}{\langle \mathtt{call}\ \mathit{exprs}_{n+1} \mathtt{:} \mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \xi}$$

$$(\mathsf{CALL\_E_2})\ \frac{\langle \mathit{exprs}_{n+1}, \theta\rangle \to \mathit{vals}_1 \quad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \xi}{\langle \mathtt{call}\ \mathit{exprs}_{n+1} \mathtt{:} \mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \xi}$$

$$(\mathsf{CALL\_E_3})\ \frac{\begin{array}{c}\langle \mathit{exprs}_{n+1}, \theta\rangle \to \mathit{vals}'_1 \qquad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \mathit{vals}'_2 \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_i, \theta\rangle \to \mathit{vals}_i \quad \langle \mathit{exprs}_{i+1}, \theta\rangle \to \xi\end{array}}{\langle \mathtt{call}\ \mathit{exprs}_{n+1} \mathtt{:} \mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \xi}$$

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:

$$(\mathsf{CALL\_E_4})\ \frac{\begin{array}{c}\langle \mathit{exprs}_{n+1}, \theta\rangle \to \mathit{vals}'_1 \quad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \mathit{vals}'_2 \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n\end{array}}{\langle \mathtt{call}\ \mathit{exprs}_{n+1} \mathtt{:} \mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{Exception}(\mathtt{error}, \mathtt{bad\_argument}, \ldots)}$$

if vals'_1 is not an atom

$$(\mathsf{CALL\_E_5})\ \frac{\begin{array}{c}\langle \mathit{exprs}_{n+1}, \theta\rangle \to \mathit{Atom}_1 \quad \langle \mathit{exprs}_{n+2}, \theta\rangle \to \mathit{vals}'_2 \\ \langle \mathit{exprs}_1, \theta\rangle \to \mathit{vals}_1 \quad \ldots \quad \langle \mathit{exprs}_n, \theta\rangle \to \mathit{vals}_n\end{array}}{\langle \mathtt{call}\ \mathit{exprs}_{n+1} \mathtt{:} \mathit{exprs}_{n+2}(\mathit{exprs}_1, \ldots, \mathit{exprs}_n), \theta\rangle \to \mathit{Exception}(\mathtt{error}, \mathtt{bad\_argument}, \ldots)}$$

if vals'_2 is not an atom

The rule (CALL_E$_6$) throws an `undefined_function` exception when the function is not defined in the specified module:

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

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

The rule (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 (PATH_E) indicates that none of the paths can be taken by proving that all of them fail:

$$(\mathsf{PATH\_E})\ \frac{fails(\theta, vals, pats_1, exprs_1) \quad \ldots \quad fails(\theta, vals, pats_n, exprs_n)}{path(\theta, vals, \overline{pats_n}, \overline{exprs_n}) \to \bot}$$

The rule (CASE_E$_1$) propagates an exception thrown while evaluating the expression:

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

The rule (CASE_E$_2$) throws an exception when the value obtained from the expression does not allow to take any of the branches in the case expression:

$$(\mathsf{CASE\_E_2})\ \frac{\langle exprs_1,\theta\rangle \to vals_1 \quad path(\theta, vals, \overline{pats_n}, \overline{exprs_n}) \to \bot}{\langle \texttt{case}\ exprs_1\ \texttt{of}\ \overline{pats_n\ \texttt{when}\ exprs'_n\ \texttt{->}\ exprs_n}\ \texttt{end},\theta\rangle \to Exception(\texttt{error},\texttt{case\_match\_fail},\ldots)}$$

References

[1] 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`.