A Declarative Debugger for Concurrent Erlang Programs (extended version)*

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

Technical Report 15/13

Departamento de Sistemas Informáticos y Computación,
Universidad Complutense de Madrid

December, 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 a calculus for Core Erlang programs. It provides a *medium-sized normal form*, that is, expressions are evaluated to an intermediate normal form, which might be either a value or a `let` expression containing an unevaluated `receive`. These medium-sized normal forms are then evaluated by consuming messages from an outbox.

Keywords: Erlang, Semantics, Concurrency.

1 Sintax

We present here the syntax for Core Erlang.

fname	::=	Atom `/` Integer
lit	::=	Atom | Integer | Float | Char | String | BitString | `[ ]`
fun	::=	`fun(`var_1 `, ... ,` var_n`) ->` expr
clause	::=	pat `when` expr_1 `->` expr_2
pat	::=	var | lit | `[` pat_1 | pat_2 `]` | `{` pat_1`, ... ,` pat_n `}` | `#{` bitpat_1, ..., bitpat_n `}#` | var `=` pat
bitpat	::=	`#<` pat_b `>(` *opts* `)`
pat_b	::=	var | Integer | Float
expr	::=	var | lit | ξ | fname | fun | `[` expr_1 | expr_2 `]` | `{` expr_1`, ... ,` expr_n `}` | `#{` bitexpr_1, ..., bitexpr_n `}#` | `let` vars `=` expr_1 `in` expr_2 | `letrec` fname_1 `=` fun_1 ... fname_n `=` fun_n `in` expr | `apply` expr `(` expr_1 `, ... ,` expr_n `)` | `call` expr_{n+1}`:`expr_{n+2} `(` expr_1 `, ... ,` expr_n `)` | `primop` Atom `(` expr_1 `, ... ,` expr_n `)` | `try` expr_1 `of` var `->` expr_2 `catch` `{`var'_1 `, ... ,` var'_m`}` `->` expr_3 | `case` expr `of` clause_1 ... clause_n `end` | `do` expr_1 expr_2 | `catch` expr | `receive` clause_1 ... clause_n `end` | `receive` clause_1 ... clause_n `after` expr_t `->` expr `end`
bitexpr	::=	`#<` expr `>(` *opts* `)`
ξ	::=	Exception($\overline{val_m}$)
val	::=	lit | fname | fun | `[` val_1 | val_2 `]` | {val_1`, ... ,` val_n} | Time
Time	::=	nat | `infinity`
eval	::=	lit | fname | fun | `[` eval_1 | eval_2 `]` | {eval_1`, ... ,` eval_n} | ξ
lock	::=	`receive` clause_1 ... clause_n `end` | `receive` clause_1 ... clause_n `after` Time `->` expr `end` | `let` var `=` lock `in` expr `end`
mnf	::=	eval | lock

The *opts* argument in bit patterns and expressions is a list/tuple/sequence of encoding options that is system dependent. It usually covers the size in bits, the number of blocks, the interpretation (`integer`, `float`, `bytes`—also `binary`—or `bits`—also `bitstring`), the sign (`signed` or `unsigned`) and the endian (`big` or `little`). It is important to notice that *opts* can contain variables to be bound during evaluation (for example, to parse a field in a TCP packet we may need the value `length` of a previous field).

To abstract from the system dependent options of bit strings, we will assume that there are two functions:

- to_bits(val, *opts*), which given a value *val* ∈ Integer ∪ Float and some enconding options returns the bit string that represents *val*. For example, to_bits(127, {8,1,integer,unsigned,big}) = "011111111".
- from_bits(bits, *opts*), which given a bit string and some encoding options *opts*, returns a pair (*val*,*bits'*) where *val* is the value represented in the first *n* bit of *bits* (according to *opts*) and *bits'* is the rest of bits after the n^{th} bit. For example, from_bits("01111111100000000", {8,1,integer,unsigned,big}) = (127, "00000000").

2 Semantics

We present in this section our calculus for concurrent Erlang programs, called *CEC* (Concurrent Erlang Calculus). Processes in *CEC* are represented as tuples of the form $\ll pid, \langle expr, \theta \rangle, l \gg$, with *pid* the process identifier, *expr* the expression being evaluated in the process, θ a substitution mapping variables to values and standing for the *context* where *expr* appears, and l the outbox. This substitution is not necessary when the expression is a value, and we use the notation $E[expr]$ to denote evaluation contexts, that is, the expression *expr* is a subexpression of E. A *configuration* Π is a set of processes. Moreover, the calculus introduces the idea of *medium-sized normal form* (*mnf*), which stands for expressions that cannot be further reduced by themselves. More specifically, an *mnf* can be either a value, represented by the word *val* in *CEC*, or an expression $\langle expr, \theta \rangle$ with *expr* an expression whose leftmost, innermost subexpression is a `receive` expression, and θ is a substitution with the *context* for *expr*. Due to the Core Erlang transformation described in Section **??**, `receive` expressions can only be evaluated in `let` bindings, thus $expr \equiv$ `let` x_1 $=$ ... `let` x_n $=$ `receive` ... `end in` e_n ... `in` e_1. In *CEC* the pair $\langle expr, \theta \rangle$, with *expr* a "chain" of `let` expressions of this form, is often represented by the word *lock*, and the associated `receive` by $rec(expr)$. Finally, we use *references* in some rules. References are unique identifiers that point to specific parts of the code, and they can be understood as a tuple containing the module name, the line, and the row in the source code; we use them to distinguish between the actual code and the behavior the user had in mind when writing the code, as we will explain in the next section. The calculus proves two kinds of statements:

- $\Pi \Rightarrow \Pi'$, which indicates that the configuration Π evolves into Π' by evaluating the processes in Π and (possibly) creating new processes.
- $expr \rightarrow (mnf, l, \Pi)$, which indicates that the expression *expr* has reached the medium-sized normal form *mnf*, sending the messages in l, and creating the processes in Π.

We will use the following function for syntactic matching:

Definition (Matching function)

$match(var, val) = [var \mapsto val]$
$match(lit_1, lit_2) = id$, if $lit_1 \equiv lit_2$
$match([pat_1|pat_2], [val_1|val_2]) = \theta_1 \uplus \theta_2$, where $\theta_i \equiv match(pat_i, val_i)$
$match(\{pat_1, \ldots, pat_n\}, \{val_1, \ldots, val_n\}) = \theta_1 \uplus \ldots \uplus \theta_n$, where $\theta_i \equiv match(pat_i, val_i)$
$match(var = pat, val) = \theta[var \mapsto val]$, where $\theta \equiv match(pat, val)$
$match(\#\{bitpat_1, \ldots, bitpat_n\}\#, BitString) = \theta_1 \uplus \ldots \uplus \theta_n$, where
$(\theta_1, BitString_1) \equiv match_b(bitpat_1, BitString)$
$(\theta_2, BitString_2) \equiv match_b(bitpat_2\theta_1, BitString_1)$
...
$(\theta_n, \epsilon) \equiv match_b(bitpat_n\theta_1 \ldots \theta_{n-1}, BitString_{n-1})$
$match(pat, val) = \perp$ otherwise

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

2.1 Rules for values

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

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

$$(\mathsf{GUARD})\ \frac{\langle expr\theta, \theta \rangle \rightarrow (mnf, [], \emptyset)}{\langle guard(r_b), \theta \rangle \rightarrow mnf}$$

where r_b is a reference to *pat* `when` *expr* `->` $expr'$. Note that *expr* cannot contain a *spawn* or a *bang* expression.

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

where r_b is a reference to *pat* `when` *expr* `->` *expr′* and $\hat{\theta} \equiv match(pat\theta, val)$

$$\text{(FAIL}_1\text{)}\ \frac{\langle patbind(r_b, val), \theta\rangle \to \bot}{\langle fails(val, r_b), \theta\rangle}$$

where r_b is a reference to *pat* `when` *expr* `->` *expr′*

$$\text{(FAIL}_2\text{)}\ \frac{\langle patbind(r_b, val), \theta\rangle \to \theta' \quad \langle guard(r_b), \theta''\rangle \to \texttt{'false'}}{\langle fails(val, r_b), \theta\rangle}$$

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

$$\text{(SUCC)}\ \frac{\langle patbind(r_b, val), \theta\rangle \to \theta' \quad \langle guard(r_b), \theta''\rangle \to \texttt{'true'}}{\langle succeeds(val, r_b), \theta\rangle \to \theta'}$$

with $\theta'' \equiv \theta \uplus \theta'$, and where r_b is a reference to *pat* `when` *expr* `->` *expr′*

$$\text{(BIND)}\ \frac{\langle expr, \theta\rangle \to (val, l, \Pi)}{\langle r, exprs, \theta\rangle \to (\{var \mapsto val\}, l, \Pi)}$$

with r a reference to the variable *var*.

$$\text{(MNF)}\ \frac{}{mnf \to (mnf, [], \emptyset)}$$

$$\text{(TUP)}\ \frac{\langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n)}{\langle\{expr_1, \ldots, expr_n\}, \theta\rangle \to (\{val_1, \ldots, val_n\}, l_1 + \ldots + l_n, (\Pi_1, \ldots, \Pi_n))}$$

$$\text{(LIST)}\ \frac{\langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \langle expr_2, \theta\rangle \to (val_2, l_2, \Pi_2)}{\langle[expr_1|expr_2], \theta\rangle \to ([val_1|val_2], l_1 + l_2, (\Pi_1, \Pi_2))}$$

$$\text{(BIT)}\ \frac{\langle expr_1, \theta\rangle \to BitString_1 \quad \ldots \quad \langle expr_n, \theta\rangle \to BitString_n}{\langle\#\{expr_1, \ldots, expr_n\}\#, \theta\rangle \to BitString_1 \ldots BitString_n}$$

$$\text{(BIT}_\text{e}\text{)}\ \frac{\langle expr, \theta\rangle \to val \quad to\_bits(val, opts) = BitString}{\langle\# <expr>(opts), \theta\rangle \to BitString}$$

$$\text{(LET}_1\text{)}\ \frac{\langle r, expr_1, \theta\rangle \to (\theta', l, \Pi) \quad \langle expr_2\theta'', \theta''\rangle \to (mnf, l', \Pi')}{\langle\texttt{let}\ var^r\ \texttt{=}\ expr_1\ \texttt{in}\ expr_2, \theta\rangle \to (mnf, l + l', (\Pi, \Pi'))}$$

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

$$\text{(LET}_2\text{)}\ \frac{\langle expr_1, \theta\rangle \to (lock, l, \Pi)}{\langle\texttt{let}\ var\ \texttt{=}\ expr_1\ \texttt{in}\ expr_2, \theta\rangle \to (lock', l, \Pi)}$$

with $lock \equiv \langle expr', \theta'\rangle$ and $lock' \equiv \langle\texttt{let}\ var\ \texttt{=}\ lock\ \texttt{in}\ expr_2, \theta'\rangle$.

$$\text{(LETREC)}\ \frac{\langle expr, \theta\rangle \to (mnf, l, \Pi)}{\langle\texttt{letrec}\ fname_1\texttt{=}fun_1 \ldots fname_n\texttt{=}fun_n\ \texttt{in}\ expr, \theta\rangle \to (mnf, l, \Pi)}$$

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

$$(\textsf{APPLY}_1)\ \frac{\begin{array}{c}\langle expr,\theta\rangle \to (r_\lambda, l, \Pi) \\ \langle expr_1,\theta\rangle \to (val_1, l_1, \Pi_n) \quad \ldots \quad \langle expr_n,\theta\rangle \to (val_n, l_n, \Pi_n) \\ \langle r_\lambda,\theta'\rangle \to (mnf, l', \Pi')\end{array}}{\langle \texttt{apply}\ expr(expr_1,\ldots,expr_n),\theta\rangle \to (mnf, l+l_1+\ldots+l_n+l', n\Pi)}$$

where r_λ references a lambda abstraction, $r_\lambda \equiv \texttt{fun(}var_1\texttt{,}\ldots\texttt{,}var_n\texttt{) ->}\ expr'$, $\theta' \equiv \theta \uplus \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$, $nl \equiv l + l_1 + \ldots + l_n + l'$, and $n\Pi \equiv (\Pi, \Pi_1, \ldots, \Pi_n, \Pi')$.

$$(\textsf{APPLY}_2)\ \frac{\begin{array}{c}\langle expr,\theta\rangle \to (Atom/n, l, \Pi) \\ \langle expr_1,\theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n,\theta\rangle \to (val_n, l_n, \Pi_n) \\ \langle expr'\theta',\theta'\rangle \to (mnf, l', \Pi')\end{array}}{\langle \texttt{apply}\ expr(expr_1,\ldots,expr_n),\theta\rangle \to (mnf, nl, n\Pi)}$$

if $\rho(Atom/n) = \texttt{fun(}\ var_1\ \texttt{,}\ \ldots\texttt{,}\ var_n\ \texttt{)}\ \texttt{->}\ expr'$, $\theta' \equiv \theta \uplus \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$, $nl \equiv l + l_1 + \ldots + l_n + l'$, and $n\Pi \equiv (\Pi, \Pi_1, \ldots, \Pi_n, \Pi')$.

$$(\textsf{APPLY}_3)\ \frac{\begin{array}{c}\langle expr,\theta\rangle \to (Atom/n, l, \Pi) \\ \langle expr_1,\theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n,\theta\rangle \to (val_n, l_n, \Pi_n) \\ \langle r_f,\theta'\rangle \to (mnf, l', \Pi')\end{array}}{\langle \texttt{apply}^r\ expr(expr_1,\ldots,expr_n),\theta\rangle \to (mnf, nl, n\Pi)}$$

where $Atom/n$ is a function defined in the module $r.mod$ as $Atom/n$ `= fun (`var_1`, ... ,` var_n`) ->` $expr$, r_f its reference, $\theta' \equiv \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$, $nl \equiv l+l_1+\ldots+l_n+l'$, and $n\Pi \equiv (\Pi, \Pi_1, \ldots, \Pi_n, \Pi')$.

$$(\textsf{CALL})\ \frac{\begin{array}{c}\langle expr_{n+1},\theta\rangle \to (Atom_1, l, \Pi) \quad \langle expr_{n+2},\theta\rangle \to (Atom_2, l', \Pi') \\ \langle expr_1,\theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n,\theta\rangle \to (val_n, l_n, \Pi_n) \\ \langle r_f,\theta'\rangle \to (mnf, l'', \Pi'')\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1,\ldots,expr_n),\theta\rangle \to (mnf, nl, n\Pi)}$$

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, $\theta' \equiv \{var_1 \mapsto val_1, \ldots, var_n \mapsto val_n\}$, $nl \equiv l + l' + l_1 + \ldots + l_n + l''$, and $n\Pi \equiv (\Pi, \Pi', \Pi_1, \ldots, \Pi_n, \Pi'')$.

$$(\textsf{CALL\_EXEC})\ \frac{\begin{array}{c}\langle expr_{n+1},\theta\rangle \to (\texttt{'erlang'}, l, \Pi) \quad \langle expr_{n+2},\theta\rangle \to (Atom_2, l', \Pi') \\ \langle expr_1,\theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n,\theta\rangle \to (val_n, l_n, \Pi_n) \\ exec(Atom_2, val_1, \ldots, val_n) = eval\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1,\ldots,expr_n),\theta\rangle \to (eval, nl, n\Pi)}$$

where $Atom_2/n$ is a built-in function included in the `erlang` module, $nl \equiv l + l' + l_1 + \ldots + l_n$, and $n\Pi \equiv (\Pi, \Pi', \Pi_1, \ldots, \Pi_n)$.

$$(\textsf{PRIMOP})\ \frac{\begin{array}{c}\langle expr_1,\theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n,\theta\rangle \to (val_n, l_n, \Pi_n) \\ eval(Atom, val_1, \ldots, val_n) = eval'\end{array}}{\langle \texttt{primop}\ Atom(expr_1,\ldots,expr_n),\theta\rangle \to evals'}$$

where $nl \equiv l_1 + \ldots + l_n$, and $n\Pi \equiv (\Pi_1, \ldots, \Pi_n)$.

$$(\textsf{TRY}_1)\ \frac{\langle expr_1,\theta\rangle \to (val', l, \Pi) \quad \langle expr_2\theta'',\theta''\rangle \to (mnf, l', \Pi')}{\langle \texttt{try}\ expr_1\ \texttt{of}\ var\ \texttt{->}\ expr_2\ \texttt{catch}\ \{var_1',\ldots,var_m'\}\ \texttt{->}\ expr_3,\theta\rangle \to (mnf, nl, n\Pi)}$$

with $\theta' \equiv match(var, val')$, $\theta'' \equiv \theta\ \uplus\ \theta'$, $nl \equiv l + l'$, and $n\Pi \equiv (\Pi, \Pi')$.

$$(\textsf{TRY}_2)\ \frac{\langle expr_1,\theta\rangle \to (Except(val_1,\ldots,val_m), l, \Pi) \quad \langle expr_3\theta'',\theta''\rangle \to (mnf, l', \Pi')}{\langle \texttt{try}\ expr_1\ \texttt{of}\ var\ \texttt{->}\ expr_2\ \texttt{catch}\ \{var_1',\ldots,var_m'\}\ \texttt{->}\ expr_3,\theta\rangle \to (mnf, nl, n\Pi)}$$

with $\theta' \equiv match(\texttt{\{}var_1',\ldots,var_m'\texttt{\}}, \texttt{\{}val_1,\ldots,val_m\texttt{\}})$, $\theta'' \equiv \theta \uplus \theta'$, $nl \equiv l + l'$, and $n\Pi \equiv (\Pi, \Pi')$.

$$(\textsf{CASE})\ \frac{\langle c\_arg(r_c),\theta\rangle \to (vals, l, \Pi) \quad \begin{array}{l}\langle fails(val, r_1),\theta\rangle \\ \ldots \\ \langle fails(val, r_{i-1}),\theta\rangle \\ \langle succeeds(val, r_i),\theta\rangle \to \theta'\end{array} \quad \langle c\_result(r_i),\theta''\rangle \to (mnf, l, \Pi')}{\langle \texttt{case}^{r_c}\ expr\ \texttt{of}\ clause_1\ \ldots\ clause_n\ \texttt{end},\theta\rangle \to (mnf, nl, n\Pi)}$$

where $nl \equiv l + l'$, $n\Pi \equiv (\Pi, \Pi')$ $\theta'' \equiv \theta \uplus \theta'$ and r_c is a reference to a `case` statement defined as

$$\begin{array}{ll}\texttt{case}\ expr\ \texttt{of} & pat_1\ \texttt{when}\ expr'_1\ \texttt{->}^{r_1}\ expr''_1 \\ & \ldots \\ & pat_n\ \texttt{when}\ exprs'_n\ \texttt{->}^{r_n}\ expr''_n\ \texttt{end}\end{array}$$

and the labels $r_1, \ldots, r_n$ are references to the different branches that can be selected by the statement.

$$(\mathsf{SPAWN})\frac{\langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n)}{\langle spawn(m, f, [expr_1, \ldots, expr_n]), \theta\rangle \to (pid, l, \Pi)}$$

with $l \equiv l_1 + \ldots + l_n$, $\Pi \equiv (\Pi_1, \ldots, \Pi_n, \ll pid, \langle f(val_1, \ldots, val_n), id\rangle, [] \gg)$, and pid a new process identifier.

$$(\mathsf{BANG})\frac{\langle expr_1, \theta\rangle \to (pid, l_1, \Pi_1) \quad \langle expr_2, \theta\rangle \to (val, l_2, \Pi_2)}{\langle expr_1 \,!\, expr_2, \theta\rangle \to (val, l_1 + l_2 + [pid \,!\, val], (\Pi_1, \Pi_2))}$$

$$(\mathsf{RCV})\frac{\langle expr_t, \theta\rangle \to (val, l, \Pi)}{\langle E[expr], \theta\rangle \to (\langle E[expr'], \theta\rangle, l, \Pi)}$$

where $expr$ is a `receive` expression defined as:

$$\begin{array}{ll}\texttt{receive} & pat_1\ \texttt{when}\ expr'_1\ \texttt{->}^{r_1}\ expr''_1 \\ & \ldots \\ & pat_n\ \texttt{when}\ expr'_n\ \texttt{->}^{r_n}\ expr''_n \\ & \texttt{after}\ expr_t\ \texttt{->}\ expr_f\ \texttt{end}\end{array}$$

and $expr'$ is defined as

$$\begin{array}{ll}\texttt{receive} & pat_1\ \texttt{when}\ expr'_1\ \texttt{->}^{r_1}\ expr''_1 \\ & \ldots \\ & pat_n\ \texttt{when}\ expr'_n\ \texttt{->}^{r_n}\ expr''_n \\ & \texttt{after}\ val\ \texttt{->}\ expr_f\ \texttt{end}\end{array}$$

$$(\mathsf{C\_ARG})\frac{\langle expr, \theta\rangle \to (val, l, \Pi)}{\langle c\_arg(r_c), \theta\rangle \to (val, l, \Pi)}$$

with $expr$ the argument expression of the `case` referenced by r_c.

$$(\mathsf{C\_RESULT_1})\frac{\langle expr_i\theta, \theta\rangle \to (mnf, l, \Pi)}{\langle c\_result(r_i), \theta\rangle \to (mnf, l, \Pi)}$$

with $expr_i$ the result expression of a `case` branch referenced by r_i.

$$(\mathsf{C\_RESULT_2})\frac{\langle expr_i\theta, \theta\rangle \to (mnf, l, \Pi)}{\langle c\_result(r_i), \theta\rangle \to (mnf, l, \Pi)}$$

with $expr_i$ the result expression of a `receive` branch referenced by r_i.

$$(\mathsf{DO})\frac{\langle \texttt{let}\ \_ = expr_1\ \texttt{in}\ expr_2, \theta\rangle \to (mnf, l, \Pi)}{\langle \texttt{do}\ expr_1\ expr_2, \theta\rangle \to (mnf, l, \Pi)}$$

$$(\mathsf{CATCH})\frac{\langle expr', \theta\rangle \to (mnf, l, \Pi)}{\langle \texttt{catch}\ expr, \theta\rangle \to (mnf, l, \Pi)}$$

with $expr' \equiv$

```
try expr of var -> var
catch {var1, var2, var3} ->
  case var1 of
    'throw' when 'true' ->
      var2
    'exit' when 'true' ->
      {'EXIT', var2}
    'error' when 'true' ->
      {'EXIT', {var2, primop exc_trace(var3)}}
  end
```

2.2 Rules for processes

$$(\mathsf{PROC})\ \frac{\langle expr, \theta\rangle \to (mnf, l', \Pi')}{\Pi, \ll pid, \langle expr, \theta\rangle, l \gg \Rightarrow \Pi, \ll pid, mnf, l + l' \gg, \Pi'}$$

$$(\mathsf{RCV}_1)\ \frac{rcv(\langle expr_1, \theta\rangle, l_1, j) \to (val, l, \Pi) \qquad \langle expr_2\theta', \theta'\rangle \to (mnf, l', \Pi')}{rcv(\langle \texttt{let}\ \ var\ \texttt{=}\ expr_1\ \ \texttt{in}\ \ expr_2, \theta\rangle, l_1, j) \to (mnf, l + l', (\Pi, \Pi'))}$$

where $\theta' \equiv \theta \uplus \{var \mapsto val\}$.

$$(\mathsf{RCV}_2)\ \frac{rcv(\langle expr_1, \theta\rangle, l_1, j) \to (\langle mnf, \theta'\rangle, l, \Pi)}{rcv(\langle \texttt{let}\ \ var\ \texttt{=}\ expr_1\ \ \texttt{in}\ \ expr_2, \theta\rangle, l_1, j) \to (expr', l, \Pi)}$$

where mnf is not a value and $expr' \equiv \langle \texttt{let}\ var\ \texttt{=}\ mnf\ \texttt{in}\ expr_2, \theta'\rangle$.

$$(\mathsf{RCV}_3)\ \frac{\begin{array}{c}\langle fails(msg_1, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_1, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_{j-1}, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_{j-1}, r_n, pid), \theta\rangle \\ \langle fails(msg_j, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_j, r_{i-1}, pid), \theta\rangle,\ \langle succeeds(msg_j, r_i, pid), \theta\rangle \to \theta' \\ \langle c\_result(r_i), \theta''\rangle \to (mnf, l', \Pi)\end{array}}{rcv(\langle expr, \theta\rangle, l, j) \to (mnf, l', \Pi)}$$

where $l \equiv [msg_1, \ldots, msg_m]$, $l'' \equiv [msg_1, \ldots, msg_{j-1}, msg_{j+1}, \ldots msg_m]$, $\theta'' \equiv \theta \uplus \theta'$, and $expr$ is a `receive` expression defined as:

```
receive   pat1 when expr'1 ->^r1 expr''1
          ...
          patn when expr'n ->^rn expr''n end
```

or

```
receive   pat1 when expr'1 ->^r1 expr''1
          ...
          patn when expr'n ->^rn expr''n
          after N -> exprf end
```

and the labels r_1, ..., r_n are references to the different branches that can be selected by the statement.

$$(\mathsf{RCV}_4)\ \frac{\begin{array}{c}\langle fails(msg_1^1, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_1^1, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_{l_1}^1, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_{l_1}^1, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_1^m, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_1^m, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_{l_m}^m, r_1, pid), \theta\rangle\ \ldots\ \langle fails(msg_{l_m}^m, r_n, pid), \theta\rangle \\ \langle c\_result(r_a), \theta\rangle \to (mnf, l, \Pi)\end{array}}{rcv(\langle expr, \theta\rangle, \{l_1, \ldots, l_m\}) \to (mnf, l, \Pi)}$$

where $l_i \equiv [msg_1^i, \ldots, msg_{l_i}^i]$, $expr$ is a `receive` expression defined as:

```
receive   pat1 when expr'1 ->^r1 expr''1
          ...
          patn when expr'n ->^rn expr''n
          after val ->^ra exprf end
```

$val \neq$ `infinity`, and the labels r_1, ..., r_n, r_a are references to the different branches that can be selected by the statement.

$$(\mathsf{CONSUME}_1)\ \frac{rcv(mnf_1, l_2', j) \to (mnf', l', \Pi') \qquad 1 \leq j \leq m}{\Pi, \ll pid_1, mnf_1, l_1 \gg, \ll pid_2, mnf_2, l_2 \gg \Rightarrow \Pi, \Pi' \ll pid_1, mnf', l_3 \gg, \ll pid_2, mnf_2, l_2'' \gg}$$

where $l_2 \equiv [msg_1, \ldots, msg_m]$, l_2' is l_2 restricted to messages addressed to pid, i the position of the jth message of l_2' in l_2, l_2'' is l_2 after removing the ith message, and $l_3 \equiv l_1 + l'$.

$$(\mathsf{CONSUME}_2)\ \frac{rcv(mnf, l', j) \to (mnf', l_1, \Pi') \qquad 1 \leq j \leq m}{\Pi, \ll pid, mnf, l \gg \Rightarrow \Pi, \Pi' \ll pid_1, mnf', l_2 \gg}$$

where l' is l restricted to messages addressed to pid, i the position of the jth message of l' in l, l'' is l after removing the ith message, and $l_2 \equiv l'' + l_1$.

$$(\mathsf{CONSUME_3})\ \frac{rcv(mnf, \{l'_1, \ldots, l'_n\}) \rightarrow (mnf', l', \Pi')}{\Pi, \ll pid, mnf, l \gg \Rightarrow \Pi, \Pi' \ll pid_1, mnf', l + l' \gg}$$

where $\Pi \equiv \ll pid_1, mnf_1, l_1 \gg, \ldots, \ll pid_n, mnf_n, l_n \gg$ and l'_i is l_1 restricted to the messages addressed to pid.

$$(\mathsf{LOCK})\ \frac{\begin{array}{c}\langle fails(msg_1^1, r_1, pid), \theta\rangle \ \ldots\ \langle fails(msg_1^1, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_{l_1}^1, r_1, pid), \theta\rangle \ \ldots\ \langle fails(msg_{l_1}^1, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_1^m, r_1, pid), \theta\rangle \ \ldots\ \langle fails(msg_1^m, r_n, pid), \theta\rangle \\ \ldots \\ \langle fails(msg_{l_m}^m, r_1, pid), \theta\rangle \ \ldots\ \langle fails(msg_{l_m}^m, r_n, pid), \theta\rangle\end{array}}{lock(pid, (\Pi, \ll pid, \langle E[expr], \theta\rangle, l \gg))}$$

where $\Pi \equiv \ll pid_1, expr'_1, [msg_1^1, \ldots, msg_{l_1}^1 \gg, \ldots, \ll pid_m, expr'_m, [msg_1^m, \ldots, msg_{l_m}^m \gg$, $\Pi' \equiv \Pi, \Pi_1, \Pi_2$, $l' \equiv l + l_1 + l_2$, $expr$ is a `receive` expression defined as:

```
receive   pat_1 when expr'_1 ->^{r_1} expr''_1
          ...
          pat_n when expr'_n ->^{r_n} expr''_n end
```

or:

```
receive   pat_1 when expr'_1 ->^{r_1} expr''_1
          ...
          pat_n when expr'_n ->^{r_n} expr''_n
          after infinity ->^{r_a} expr_f end
```

and the labels r_1, ..., r_n, r_a are references to the different branches that can be selected by the statement.

$$(\mathsf{ENDLOCK})\ \frac{lock(pid_1, \Pi) \quad \ldots \quad lock(pid_i, \Pi)}{\Pi \Rightarrow endlock}$$

$\Pi \equiv \ll pid_1, mnf_1, l_1 \gg, \ldots, \ll pid_i, mnf_i, l_i \gg, \ll pid_{i+1}, val_{i+1}, l_{i+1} \gg, \ldots, \ll pid_n, val_n, l_n \gg$, $i > 0$, and mnf_j, with $1 \leq j \leq i$, are not values.

$$(\mathsf{Tr})\ \frac{\Pi \Rightarrow \Pi_1 \quad \Pi_1 \Rightarrow \Pi'}{\Pi \Rightarrow \Pi'}$$

$$(\mathsf{FAIL_1^{PID}})\ \frac{\langle patbind(r_b, val), \theta\rangle \rightarrow \bot}{\langle fails(pid\,!\,val, r_b), \theta\rangle}$$

where r_b is a reference to $pats$ `when` $expr$ `->` $expr'$

$$(\mathsf{FAIL_2^{PID}})\ \frac{\langle patbind(r_b, val), \theta\rangle \rightarrow \theta' \quad \langle guard(r_b), \theta''\rangle \rightarrow \texttt{'false'}}{\langle fails(pid\,!\,val, r_b), \theta\rangle}$$

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

$$(\mathsf{SUCC^{PID}})\ \frac{\langle patbind(r_b, val), \theta\rangle \rightarrow \theta' \quad \langle guard(r_b), \theta''\rangle \rightarrow \texttt{'true'}}{\langle succeeds(pid\,!\,val, r_b, pid), \theta\rangle \rightarrow \theta'}$$

with $\theta'' \equiv \theta \uplus \theta'$, and where r_b is a reference to pat `when` $expr$ `->` $expr'$

2.3 Exception rules

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

$$(\mathsf{SEQ\_E})\ \frac{\begin{array}{c}\langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \ldots \quad \langle \mathit{expr}_i, \theta\rangle \to (\mathit{val}_i, l_i, \Pi_i) \\ \langle \mathit{expr}_{i+1}, \theta\rangle \to (\xi, l_{i+1}, \Pi_{i+1})\end{array}}{\langle \texttt{<}\ \mathit{expr}_1, \ldots, \mathit{expr}_n\ \texttt{>}, \theta\rangle \to (\xi, l, \Pi)}$$

where $l \equiv l_1 + \ldots + l_{i+1}$ and $\Pi \equiv \Pi_1, \ldots, \Pi_{i+1}$.

$$(\mathsf{TUP\_E})\ \frac{\begin{array}{c}\langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \ldots \quad \langle \mathit{expr}_i, \theta\rangle \to (\mathit{val}_i, l_i, \Pi_i) \\ \langle \mathit{expr}_{i+1}, \theta\rangle \to (\xi, l_{i+1}, \Pi_{i+1})\end{array}}{\langle \{\mathit{expr}_1, \ldots, \mathit{expr}_n\}, \theta\rangle \to (\xi, l, \Pi)}$$

where $l \equiv l_1 + \ldots + l_{i+1}$ and $\Pi \equiv \Pi_1, \ldots, \Pi_{i+1}$.

$$(\mathsf{LIST\_E_1})\ \frac{\langle \mathit{expr}_1, \theta\rangle \to (\xi, l, \Pi)}{\langle [\mathit{expr}_1 | \mathit{expr}_2], \theta\rangle \to (\xi, l, \Pi)}$$

$$(\mathsf{LIST\_E_2})\ \frac{\langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \langle \mathit{expr}_2, \theta\rangle \to (\xi, l_2, \Pi_2)}{\langle [\mathit{expr}_1 | \mathit{expr}_2], \theta\rangle \to (\xi, l, \Pi)}$$

where $l \equiv l_1 + l_2$ and $\Pi \equiv \Pi_1, \Pi_2$.

$$(\mathsf{LET\_E_1})\ \frac{\langle \mathit{expr}_1, \theta\rangle \to (\xi, l, \Pi)}{\langle \texttt{let <}\ \mathit{var}_1, \ldots, \mathit{var}_n\ \texttt{> =}\ \mathit{expr}_1\ \texttt{in}\ \mathit{expr}, \theta\rangle \to (\xi, l, \Pi)}$$

$$(\mathsf{APPLY\_E_1})\ \frac{\langle \mathit{expr}, \theta\rangle \to (\xi, l, \Pi)}{\langle \texttt{apply}\ \mathit{expr}(\mathit{expr}_1, \ldots, \mathit{expr}_n), \theta\rangle \to (\xi, l, \Pi)}$$

$$(\mathsf{APPLY\_E_2})\ \frac{\begin{array}{c}\langle \mathit{expr}, \theta\rangle \to (\mathit{val}, l, \Pi) \\ \langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \ldots \quad \langle \mathit{expr}_i, \theta\rangle \to (\mathit{val}_i, l_i, \Pi_i) \\ \langle \mathit{expr}_{i+1}, \theta\rangle \to (\xi, l_{i+1}, \Pi_{i+1})\end{array}}{\langle \texttt{apply}\ \mathit{expr}(\mathit{expr}_1, \ldots, \mathit{expr}_n), \theta\rangle \to (\xi, l', \Pi')}$$

where $l' \equiv l + l_1 + \ldots + l_{i+1}$ and $\Pi' \equiv \Pi, \Pi_1, \ldots, \Pi_{i+1}$.

$$(\mathsf{APPLY\_E_3})\ \frac{\begin{array}{c}\langle \mathit{expr}, \theta\rangle \to (\mathit{val}, l, \Pi) \\ \langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \ldots \quad \langle \mathit{expr}_n, \theta\rangle \to (\mathit{val}_n, l_n, \Pi_n)\end{array}}{\langle \texttt{apply}^r\ \mathit{expr}(\mathit{expr}_1, \ldots, \mathit{expr}_n), \theta\rangle \to (\mathit{Except}(\texttt{error}, \texttt{bad\_function}, \ldots), l', \Pi')}$$

when *val* is not *fname* $\in dom(\rho)$ or in module $r.mod$ and it is not a *fun* expression, $l' \equiv l + l_1 + \ldots + l_n$, and $\Pi' \equiv \Pi, \Pi_1, \ldots, \Pi_n$.

$$(\mathsf{APPLY\_E_4})\ \frac{\begin{array}{c}\langle \mathit{expr}, \theta\rangle \to (\texttt{fun(}\mathit{var}_1\texttt{,} \ldots \texttt{,}\mathit{var}_m\texttt{) ->}\ \mathit{expr}', l, \Pi) \\ \langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \ldots \quad \langle \mathit{expr}_n, \theta\rangle \to (\mathit{val}_n, l_n, \Pi_n)\end{array}}{\langle \texttt{apply}\ \mathit{expr}(\mathit{expr}_1, \ldots, \mathit{expr}_n), \theta\rangle \to (\mathit{Except}(\texttt{error}, \texttt{anon called with m args}, \ldots), l', \Pi')}$$

if $m \neq n$, $l' \equiv l + l_1 + \ldots + l_n$, and $\Pi' \equiv \Pi, \Pi_1, \ldots, \Pi_n$.

$$(\mathsf{APPLY\_E_5})\ \frac{\begin{array}{c}\langle \mathit{expr}, \theta\rangle \to (\mathit{Atom}/m, l, \Pi) \\ \langle \mathit{expr}_1, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \ldots \quad \langle \mathit{expr}_n, \theta\rangle \to (\mathit{val}_n, l_n, \Pi_n)\end{array}}{\langle \texttt{apply}\ \mathit{expr}(\mathit{expr}_1, \ldots, \mathit{expr}_n), \theta\rangle \to (\mathit{Except}(\texttt{error}, \texttt{called with n args}, \ldots), l', \Pi')}$$

if $m \neq n$, $l' \equiv l + l_1 + \ldots + l_n$, and $\Pi' \equiv \Pi, \Pi_1, \ldots, \Pi_n$.

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

$$(\mathsf{CALL\_E_2})\ \frac{\langle \mathit{expr}_{n+1}, \theta\rangle \to (\mathit{val}_1, l_1, \Pi_1) \quad \langle \mathit{expr}_{n+2}, \theta\rangle \to (\xi, l_2, \Pi_2)}{\langle \texttt{call}\ \mathit{expr}_{n+1}\texttt{:}\mathit{expr}_{n+2}(\mathit{expr}_1, \ldots, \mathit{expr}_n), \theta\rangle \to (\xi, l, \Pi)}$$

with $l \equiv l_1 + l_2$ and $\Pi \equiv \Pi_1, \Pi_2$.

$$(\mathsf{CALL\_E_3})\ \frac{\begin{array}{c}\langle expr_{n+1}, \theta\rangle \to (val'_1, l'_1, \Pi'_1) \quad \langle expr_{n+2}, \theta\rangle \to (val'_2, l'_2, \Pi'_2) \\ \langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_i, \theta\rangle \to (val_i, l_i, \Pi_i) \\ \langle expr_{i+1}, \theta\rangle \to (\xi, l_{i+1}, \Pi_{i+1})\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1, \ldots, expr_n), \theta\rangle \to (\xi, l, \Pi)}$$

with $l \equiv l'_1 + l'_2 + l_1 + \ldots + l_{i+1}$ and $\Pi \equiv \Pi'_1, \Pi'_2, \Pi_1, \ldots, \Pi_{i+1}$.

$$(\mathsf{CALL\_E_4})\ \frac{\begin{array}{c}\langle expr_{n+1}, \theta\rangle \to (val'_1, l'_1, \Pi'_1) \quad \langle expr_{n+2}, \theta\rangle \to (val'_2, l'_2, \Pi'_2) \\ \langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n)\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1, \ldots, expr_n), \theta\rangle \to (Exception(\texttt{error}, \texttt{bad\_argument}, \ldots), l, \Pi)}$$

if val'_1 is not an atom, $l \equiv l'_1 + l'_2 + l_1 + \ldots + l_n$, and $\Pi \equiv \Pi'_1, \Pi'_2, \Pi_1, \ldots, \Pi_n$.

$$(\mathsf{CALL\_E_5})\ \frac{\begin{array}{c}\langle expr_{n+1}, \theta\rangle \to (Atom_1, l'_1, \Pi'_1) \quad \langle expr_{n+2}, \theta\rangle \to (val'_2, l'_2, \Pi'_2) \\ \langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n)\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1, \ldots, expr_n), \theta\rangle \to (Exception(\texttt{error}, \texttt{bad\_argument}, \ldots), l, \Pi)}$$

if $vals'_2$ is not an atom, $l \equiv l'_1 + l'_2 + l_1 + \ldots + l_n$, and $\Pi \equiv \Pi'_1, \Pi'_2, \Pi_1, \ldots, \Pi_n$.

$$(\mathsf{CALL\_E_6})\ \frac{\begin{array}{c}\langle expr_{n+1}, \theta\rangle \to (Atom_1, l'_1, \Pi'_1) \quad \langle expr_{n+2}, \theta\rangle \to (Atom_2, l'_2, \Pi'_2) \\ \langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_n, \theta\rangle \to (val_n, l_n, \Pi_n)\end{array}}{\langle \texttt{call}\ expr_{n+1}\texttt{:}expr_{n+2}(expr_1, \ldots, expr_n), \theta\rangle \to (Exception(\texttt{error}, \texttt{undefined\_function}, \ldots), l, \Pi)}$$

if $\overline{vals'_n} \notin Exc$ and the function $Atom_2/n$ is not defined and exported in module $Atom_1$, $l \equiv l'_1 + l'_2 + l_1 + \ldots + l_n$, and $\Pi \equiv \Pi'_1, \Pi'_2, \Pi_1, \ldots, \Pi_n$.

$$(\mathsf{PRIMOP\_E})\ \frac{\begin{array}{c}\langle expr_1, \theta\rangle \to (val_1, l_1, \Pi_1) \quad \ldots \quad \langle expr_i, \theta\rangle \to (val_i, l_i, \Pi_i) \\ \langle expr_{i+1}, \theta\rangle \to (\xi, l_{i+1}, \Pi_{i+1})\end{array}}{\langle \texttt{primop}\ Atom(expr_1, \ldots, expr_n), \theta\rangle \to (\xi, l, \Pi)}$$

with $l \equiv l_1 + \ldots + l_{i+1}$ and $\Pi \equiv \Pi_1, \ldots, \Pi_{i+1}$.

$$(\mathsf{CASE\_E_1})\ \frac{\langle expr_1, \theta\rangle \to (\xi, l, \Pi)}{\langle \texttt{case}\ expr_1\ \texttt{of}\ \overline{pat_n\ \texttt{when}\ expr'_n\ \texttt{->}\ expr_n}\ \texttt{end}, \theta\rangle \to (\xi, l, \Pi)}$$

$$(\mathsf{PATH\_E})\ \frac{fails(\theta, val, pat_1, expr_1) \quad \ldots \quad fails(\theta, val, pat_n, expr_n)}{path(\theta, val, \overline{pat_n}, \overline{expr_n}) \to \bot}$$

$$(\mathsf{CASE\_E_2})\ \frac{\langle expr_1, \theta\rangle \to (val, l, \Pi) \quad path(\theta, val, \overline{pat_n}, \overline{expr_n}) \to \bot}{\langle \texttt{case}\ expr_1\ \texttt{of}\ \overline{pat_n\ \texttt{when}\ expr'_n\ \texttt{->}\ expr_n}\ \texttt{end}, \theta\rangle \to (Exception(\texttt{error}, \texttt{case\_match\_fail}, \ldots), l, \Pi)}$$