Towards Automatic Verification of Erlang Programs by π–Calculus Translation

Chanchal Roy[1] Thomas Noll[2] Banani Roy[1] James R Cordy[1]

[1]School of Computing
Queen's University, Canada
{croy, broy, cordy}@cs.queensu.ca
[2]Lehrstuhl für Informatik 2
RWTH Aachen University, Germany
noll@cs.rwth-aachen.de

Fifth ACM SIGPLAN Erlang Workshop, 2006

Motivation

- High quality demands for telecommunication software (availability, robustness, correctness, ...)
- Testing not sufficient to guarantee properties
- Solution: formal verification

> Use of formal methods to prove that (a model of) a system has certain properties specified in a suitable logic.

Motivation

- High quality demands for telecommunication software (availability, robustness, correctness, ...)
- Testing not sufficient to guarantee properties
- Solution: formal verification

> Use of formal methods to prove that (a model of) a system has certain properties specified in a suitable logic.

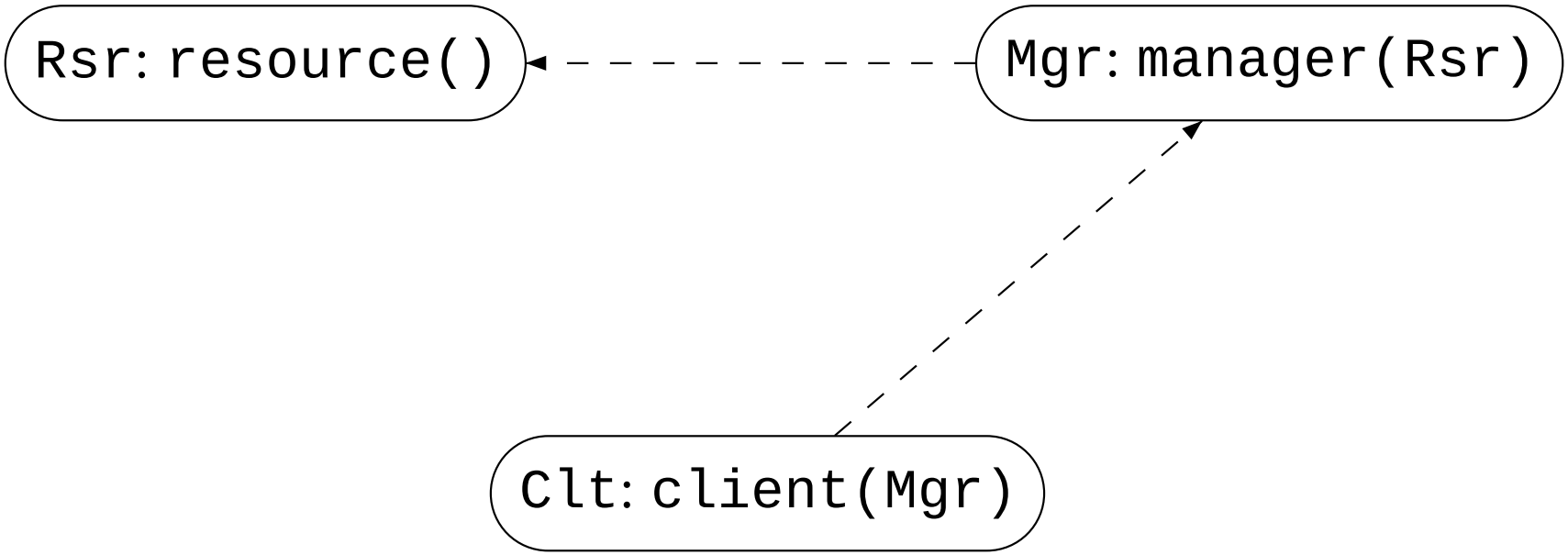
Here:

- Concentrate on first step: model construction
- Put emphasis on mobility
- Refinement of translation presented at last year's Erlang WS

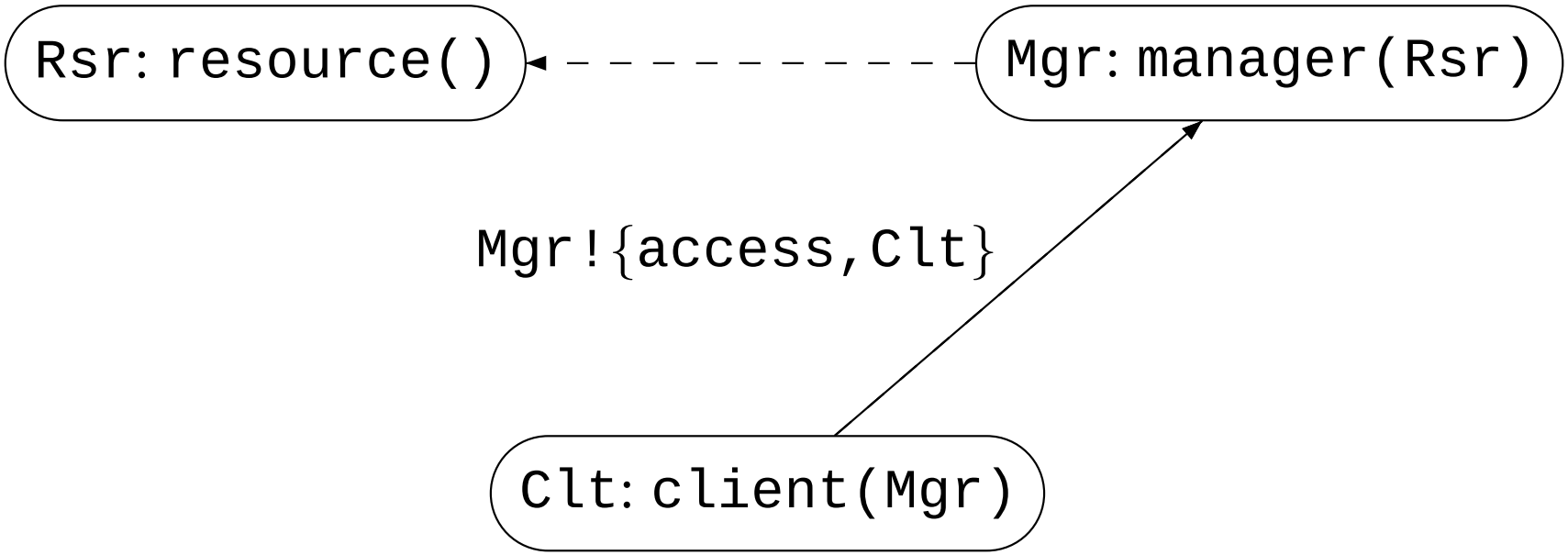
PIErlang Syntax: Subset of Erlang

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Fdef}_1 \ldots \mathit{Fdef}_n \texttt{ ; } n{>}0 \\
\mathit{Fdef} & ::= & \texttt{f(}X_1\texttt{, } \ldots \texttt{,}X_n\texttt{) -> }E\texttt{ ; } n{>}{=}0 \\
E & ::= & n \mid a \mid X \\
 & \mid & X\texttt{=}E \mid E_1\texttt{, }E_2 \\
 & \mid & \mathit{self}\texttt{ ( ) } \mid \texttt{f(}A_1\texttt{, } \ldots \texttt{,}A_n\texttt{) ; } n{>}{=}0 \\
 & \mid & \texttt{spawn(}f\texttt{, [}A_1\texttt{, } \ldots \texttt{,}A_n\texttt{]) ; } n{>}{=}0 \\
 & \mid & \{A_1\texttt{, } \ldots \texttt{,}A_n\}\texttt{ ; } n{>}0 \\
 & \mid & A_1\texttt{!}A_2 \mid A\texttt{!}\{A_1\texttt{, } \ldots \texttt{,}A_n\}\texttt{ ; } n{>}0 \\
 & \mid & \texttt{receive}\, M_1\texttt{; } \ldots \texttt{;}M_n\, \texttt{end ; } n{>}0 \\
 & \mid & \texttt{case}\, E\, \texttt{of}\, M_1\texttt{; } \ldots \texttt{;}M_n\, \texttt{end ; } n{>}0 \\
M & ::= & P\texttt{->}E \mid \{P_1\texttt{, } \ldots \texttt{,}P_n\}\texttt{->}E\texttt{ ; } n{>}0 \\
P & ::= & n \mid a \mid X \\
A & ::= & n \mid a \mid X \mid \mathit{self}\texttt{ ( )}
\end{array}
$$

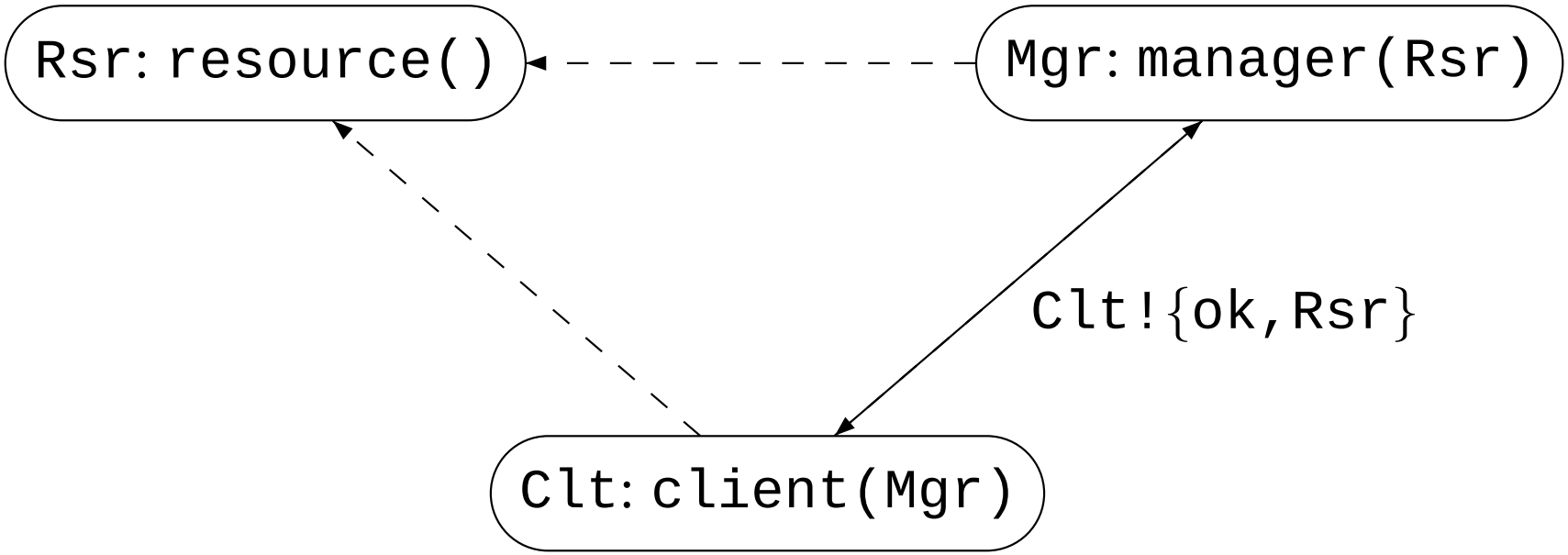
Mobility in Erlang II
Behaviour of resource manager



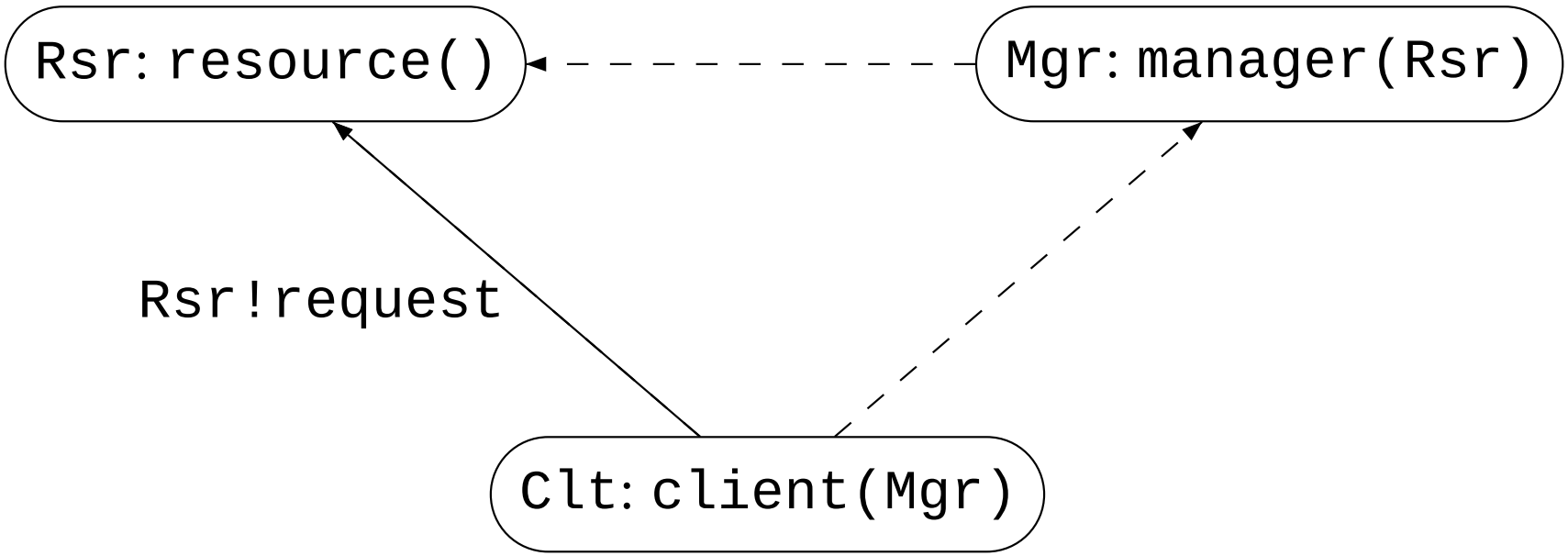
Mobility in Erlang II
Behaviour of resource manager



Mobility in Erlang II
Behaviour of resource manager



Mobility in Erlang II
Behaviour of resource manager



Mobility in Erlang I
A simplistic resource manager

```
start() ->
  Rsr = spawn(resource, []),
  Mgr = spawn(manager, [Rsr]),
  client(Mgr).

resource() ->
  receive
    Req->
      action
  end.

client(Mgr) ->
  Mgr!{access, self()},
  Receive
    {ok, R} ->
      R!request
  end.
```

```
manager(Rsr) ->
   receive
      {access, C} ->
        C!{ok, Rsr}
   end
```

The (Polyadic and Asynchronous) π–Calculus

Sys ::= $Pdef_1 \ldots Pdef_n$ % system

$Pdef$ ::= $i(x_1, \ldots, x_n) = Proc$ % process definition

$Proc$ ::= `nil` % inactive process

| $x_0(x_1, \ldots, x_n).Proc$ % input

| $\overline{x_0}\langle x_1, \ldots, x_n \rangle.\texttt{nil}$ % asynchronous output

| $Proc_1 \parallel Proc_2$ % parallel composition

| $Proc_1 + Proc_2$ % non–deterministic choice

| $(\nu\, x)Proc$ % new name

| $[x_1 = x_2]Proc$ % match

| $[x_1 \langle\rangle x_2]Proc$ % mismatch

| $i(x_1, \ldots, x_n)$ % process instantiation

Communication in the π–Calculus

Reaction rule:

$$\begin{array}{rl} & \overline{x_0}\texttt{<}y_1\texttt{, ..., }y_n\texttt{>.nil} \parallel x_0\texttt{(}x_1\texttt{, ..., }x_n\texttt{).}P \\ \rightarrow & \texttt{nil} \parallel P[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n] \end{array}$$

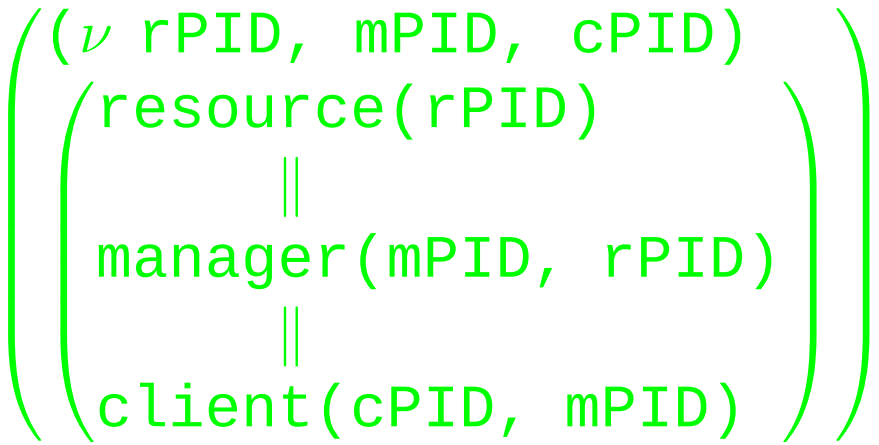
- actually synchronous
- however, special form of output is “non–blocking”

The Resource Manager in π–Calculus

```
                Main  =  (ν self)(start(self))
          start(self) =  (ν rPID, mPID, cPID, p, q)
                         (p̄<rPID>.nil ‖ resource(rPID) ‖
                          p(Rsr).(q̄<mPID>.nil ‖
                          manager(mPID,Rsr) ‖
                          q(Mgr).client(cPID,Mgr)))
       resource(self) =  self(Req).res̄<action>.nil
manager(self,Rsr)     =  self(input,C).
                         [input=access]C̄<ok,Rsr>.nil
client(self, Mgr)     =  Mgr̄<access,self>.nil ‖
                         self(input,R).
                         [input=ok]R̄<request>.nil
```

Observing Behavior of Resource Manager in π–Calculus

Instantiation of `start` process $\Longrightarrow$ react on `p` and `q` $\Longrightarrow$ omit `nil` process



Observing Behavior of Resource Manager in π–Calculus

Instantiation of `start` process $\Longrightarrow$ react on `p` and `q` $\Longrightarrow$ omit `nil` process

$$\left(\begin{array}{l}(\nu\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\quad\Vert\\ \texttt{manager(mPID, rPID)}\\ \quad\quad\Vert\\ \texttt{client(cPID, mPID)}\end{array}\right)\end{array}\right)$$

Upon instantiation of `manager` and `client` process, we get

$$\left(\begin{array}{l}(\nu\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\quad\Vert\\ \texttt{mPID(input,C).[input=access]}\overline{\texttt{C}}\texttt{<ok,rPID>.nil}\\ \quad\quad\Vert\\ \overline{\texttt{mPID}}\texttt{<access,cPID>.nil}\ \Vert\\ \texttt{cPID(input,R).[input=ok]}\overline{\texttt{R}}\texttt{<request>.nil}\end{array}\right)\end{array}\right)$$

Observing Behavior of Resource Manager in π–Calculus

`client` asks `manager` for handle to `resource`: react on `mPID`

$$\left(\begin{array}{l}(\nu\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\quad\|\\ \texttt{[access=access]}\overline{\texttt{cPID}}\texttt{<ok,rPID>.nil}\\ \quad\quad\|\\ \texttt{nil} \| \texttt{cPID(input,R).[input=ok]}\overline{\texttt{R}}\texttt{<request>.nil}\end{array}\right)\end{array}\right)$$

Observing Behavior of Resource Manager in π–Calculus

`client` asks `manager` for handle to `resource`: react on `mPID`

$$\left(\begin{array}{l}(\nu\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\quad\Vert\\ \texttt{[access=access]}\overline{\texttt{cPID}}\texttt{<ok,rPID>.nil}\\ \quad\quad\Vert\\ \texttt{nil}\Vert\ \texttt{cPID(input,R).[input=ok]}\overline{\texttt{R}}\texttt{<request>.nil}\end{array}\right)\end{array}\right)$$

Matching `access=access`, react on `cPID`

$$\left(\begin{array}{l}(\nu\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{resource(rPID)}\\ \quad\quad\Vert\\ \quad\texttt{nil}\\ \quad\quad\Vert\\ \texttt{nil}\Vert\ \texttt{[ok=ok]}\overline{\texttt{rPID}}\texttt{<request>.nil}\end{array}\right)\end{array}\right)$$

Observing Behavior of Resource Manager in π–Calculus

Invoking the `resource` process, we get

$$\left(\begin{array}{l}(\nu\ \texttt{rPID, mPID, cPID})\\ \left(\begin{array}{l}\texttt{rPID(Req).}\overline{\texttt{res}}\texttt{<action>.nil}\\ \quad\Vert\\ \quad\texttt{nil}\\ \quad\Vert\\ \texttt{nil}\ \Vert\ \texttt{[ok=ok]}\overline{\texttt{rPID}}\texttt{<request>.nil}\end{array}\right)\end{array}\right)$$

Observing Behavior of Resource Manager in π–Calculus

Invoking the `resource` process, we get

$$\left(\begin{array}{l}(\nu\ \mathtt{rPID},\ \mathtt{mPID},\ \mathtt{cPID}) \\ \left(\begin{array}{l}\mathtt{rPID(Req).}\overline{\mathtt{res}}\mathtt{<action>.nil} \\ \quad \| \\ \quad \mathtt{nil} \\ \quad \| \\ \mathtt{nil} \| \mathtt{[ok=ok]}\overline{\mathtt{rPID}}\mathtt{<request>.nil}\end{array}\right)\end{array}\right)$$

`client` can send actual request to `resource`

$$\left(\begin{array}{l}(\nu\ \mathtt{rPID},\ \mathtt{mPID},\ \mathtt{cPID}) \\ \left(\begin{array}{l}\overline{\mathtt{res}}\mathtt{<action>.nil} \\ \quad \| \\ \quad \mathtt{nil} \\ \quad \| \\ \mathtt{nil} \| \mathtt{nil}\end{array}\right)\end{array}\right)$$

The Translation Mapping

Goal: define

$$\mathit{TrPI} : \text{Erlang} \longrightarrow \pi\text{–Calculus}$$

such that the “essential behaviour” of programs is represented

The Translation Mapping

Goal: define

$$\mathit{TrPI} : \text{Erlang} \longrightarrow \pi\text{–Calculus}$$

such that the "essential behaviour" of programs is represented **Important**

issues:

- Data structures
- Process creation
- Asynchronous communication via mailboxes
- Polyadic (i.e., tuple) communication
- Deterministic matching (`case`/`receive`)

Translation of Programs and Function Definitions

Translation of programs:

$$TrPI_{prog} : Program \longrightarrow Sys$$

$$TrPI_{prog}(Fdef_1 \ldots Fdef_n) := \begin{pmatrix} \texttt{Main = (}\nu\ \texttt{self,...)}\ TrPI_{exp}(\texttt{self}, f_0) \\ TrPI_{fundef}(Fdef_1) \ldots TrPI_{fundef}(Fdef_n) \end{pmatrix}$$

Translation of Programs and Function Definitions

Translation of programs:

$$TrPI_{prog} : Program \longrightarrow Sys$$

$$TrPI_{prog}(Fdef_1 \ldots Fdef_n) := \begin{pmatrix} \texttt{Main = (}\nu\ \texttt{self,...)}\ TrPI_{exp}(\texttt{self}, f_0) \\ TrPI_{fundef}(Fdef_1) \ldots TrPI_{fundef}(Fdef_n) \end{pmatrix}$$

Translation of function definitions:

$$TrPI_{fundef} : Fdef \longrightarrow Pdef$$

$$TrPI_{fundef}(f\ \texttt{(}X_1\texttt{, ..., }X_n\texttt{) -> }E) := f\texttt{(self,}X_1\texttt{, ..., }X_n\texttt{)=}\ TrPI_{exp}(\texttt{self}, E)$$

`self`: pid of evaluating process (= `self()`)

Translation of Expressions I

$$TrPI_{exp} : Name \times Expression \rightarrow Proc$$

- yields a process which evaluates the given expression...
- ... and returns the value along the `res` channel
- abstracts from (most) data structures (numbers, lists, ...)
- atoms and pids are faithfully represented

Translation of Expressions II
Simple expressions

Using the auxiliary function

$$TrPI_{arg} : Argument \rightarrow Name$$

with

$$\begin{aligned} TrPI_{arg}(n) &:= \texttt{unknown} \\ TrPI_{arg}(a) &:= a \\ TrPI_{arg}(X) &:= X \\ TrPI_{arg}(\texttt{self()}) &:= \texttt{self} \end{aligned}$$

simple expressions can be translated as follows:

$$\begin{aligned} TrPI_{exp}(\texttt{self}, A) &:= \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A)\texttt{>.nil} \\ TrPI_{exp}(\texttt{self}, \{A_1, \ldots, A_n\}) &:= \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil} \end{aligned}$$

Translation of Expressions III

Sequencing expressions

$$TrPI_{exp}(\texttt{self}, X = E_1\texttt{,}\, E_2)$$
$$:= \quad (\nu\, \texttt{res}')(TrPI_{exp}(\texttt{self}, E_1) \parallel \texttt{res}'(X).\, TrPI_{exp}(\texttt{self}, E_2))$$

$$TrPI_{exp}(\texttt{self}, E_1\texttt{,}\, E_2)$$
$$:= \quad (\nu\, \texttt{res}')(TrPI_{exp}(\texttt{self}, E_1) \parallel \texttt{res}'(\texttt{dummy}).\, TrPI_{exp}(\texttt{self}, E_2))$$

Translation of Expressions III

Sequencing expressions

$$TrPI_{exp}(\texttt{self}, X = E_1, E_2)$$
$$:= \quad (\nu\, \texttt{res}')(TrPI_{exp}(\texttt{self}, E_1) \parallel \texttt{res}'(X).TrPI_{exp}(\texttt{self}, E_2))$$

$$TrPI_{exp}(\texttt{self}, E_1, E_2)$$
$$:= \quad (\nu\, \texttt{res}')(TrPI_{exp}(\texttt{self}, E_1) \parallel \texttt{res}'(\texttt{dummy}).TrPI_{exp}(\texttt{self}, E_2))$$

Example:

$$TrPI_{exp}(\texttt{self}, \texttt{X = a, \{X,b\}})$$
$$= \quad (\nu\, \texttt{res}')(\overline{\texttt{res}'}\texttt{<a>.nil} \parallel \texttt{res}'(\texttt{X}).\overline{\texttt{res}}\texttt{<X,b>.nil})$$

Translation of Expressions IV

Function calls and process creation

$$TrPI_{exp}(\texttt{self}, f(A_1, \ldots, A_n)) := f(\texttt{self}, TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n))$$

$$TrPI_{exp}(\texttt{self}, \texttt{X = spawn}(f, [A_1, \ldots, A_n]), E) := (\nu\, \texttt{self}')(\nu\, \texttt{res}') \left(\begin{array}{l} f(\texttt{self}', TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)) \parallel \\ \texttt{res}'(\texttt{dummy}).\texttt{nil} \parallel \\ \overline{\texttt{res}}\texttt{<self}'\texttt{>}.\texttt{nil} \parallel \\ \texttt{res}(X).TrPI_{exp}(\texttt{self}, E) \end{array} \right)$$

Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(\texttt{self}, A_1\,\texttt{!}\,A_2)$$
$$:= \quad \overline{TrPI_{arg}(A_1)}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil}$$

$$TrPI_{exp}(\texttt{self}, A\,\texttt{!}\,\{A_1\texttt{,}\ldots\texttt{,}A_n\})$$
$$:= \quad \overline{TrPI_{arg}(A)}\texttt{<}TrPI_{arg}(A_1)\texttt{,}\ldots\texttt{,}TrPI_{arg}(A_n)\texttt{>.nil} \parallel$$
$$\overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1)\texttt{,}\ldots\texttt{,}TrPI_{arg}(A_n)\texttt{>.nil}$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(\texttt{self}, A_1\,\texttt{!}\,A_2)$$
$$:= \quad \overline{TrPI_{arg}(A_1)}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil}$$

$$TrPI_{exp}(\texttt{self}, A\,\texttt{!}\,\{A_1, \ldots, A_n\})$$
$$:= \quad \overline{TrPI_{arg}(A)}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil} \parallel$$
$$\overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil}$$

- Implicit representation of Erlang mailbox using concurrent “output particles”
- Abstracts from arrival order of messages

Example:

```
P!a,P!b
```

Translation of Expressions V

Message passing (polyadic)

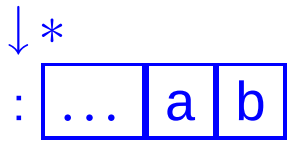
$$TrPI_{exp}(\texttt{self}, A_1\,\texttt{!}\,A_2)$$
$$:= \overline{TrPI_{arg}(A_1)}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil}$$

$$TrPI_{exp}(\texttt{self}, A\,\texttt{!}\,\{A_1, \ldots, A_n\})$$
$$:= \overline{TrPI_{arg}(A)}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil} \parallel$$
$$\overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil}$$

- Implicit representation of Erlang mailbox using concurrent "output particles"
- Abstracts from arrival order of messages

Example:

```
P!a,P!b
```



Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(\texttt{self}, A_1\,\texttt{!}\,A_2)$$
$$:= \overline{TrPI_{arg}(A_1)}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil}$$

$$TrPI_{exp}(\texttt{self}, A\,\texttt{!}\,\{A_1, \ldots, A_n\})$$
$$:= \overline{TrPI_{arg}(A)}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil} \parallel$$
$$\overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil}$$

- Implicit representation of Erlang mailbox using concurrent "output particles"
- Abstracts from arrival order of messages

Example:

$$\texttt{P!a,P!b} \xrightarrow{TrPI_{exp}} (\nu\, res')\left(\begin{array}{l}\overline{\texttt{P}}\texttt{<a>.nil} \parallel \overline{\texttt{res}'}\texttt{<a>.nil} \parallel \\ \texttt{res}'\texttt{(dummy).(}\overline{\texttt{P}}\texttt{<b>.nil} \parallel \overline{\texttt{res}}\texttt{<b>.nil)}\end{array}\right)$$

$\downarrow *$

$\text{Mailbox}_{\texttt{P}}$: | ... | a | b |

Translation of Expressions V

Message passing (polyadic)

$$TrPI_{exp}(\texttt{self}, A_1\,\texttt{!}\,A_2)$$
$$:= \quad \overline{TrPI_{arg}(A_1)}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil} \parallel \overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_2)\texttt{>.nil}$$

$$TrPI_{exp}(\texttt{self}, A\,\texttt{!}\,\{A_1, \ldots, A_n\})$$
$$:= \quad \overline{TrPI_{arg}(A)}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil} \parallel$$
$$\overline{\texttt{res}}\texttt{<}TrPI_{arg}(A_1), \ldots, TrPI_{arg}(A_n)\texttt{>.nil}$$

- Implicit representation of Erlang mailbox using concurrent "output particles"
- Abstracts from arrival order of messages

Example:

$$\texttt{P!a,P!b} \xrightarrow{TrPI_{exp}} (\nu\, res')\left(\begin{array}{l}\overline{\texttt{P}}\texttt{<a>.nil} \parallel \overline{\texttt{res}'}\texttt{<a>.nil} \parallel \\ \texttt{res}'\texttt{(dummy).(}\overline{\texttt{P}}\texttt{<b>.nil} \parallel \overline{\texttt{res}}\texttt{<b>.nil)}\end{array}\right)$$

$$\downarrow *$$

$$\text{Mailbox}_{\texttt{P}} : \boxed{\ldots}\boxed{\texttt{a}}\boxed{\texttt{b}} \qquad \overline{\texttt{P}}\texttt{<a>.nil} \parallel \overline{\texttt{P}}\texttt{<b>.nil} \parallel \overline{\texttt{res}}\texttt{<b>.nil}$$

Translation of Expressions VI
Branching expressions

Previous simple translation: nondeterministic and monadic

```
receive
  ok -> a;
  {req, P} -> b;
  X -> c
end
```

$\xrightarrow{TrPI_{exp}}$

```
self(in).[in=ok]res<a>.nil +
self(in).[in=unknown]res<b>.nil +
self(X).res<c>.nil
```

Translation of Expressions VI

Branching expressions

Previous simple translation: nondeterministic and monadic

```
receive
  ok -> a;
  {req, P} -> b;
  X -> c
end
```

$\xrightarrow{TrPI_{exp}}$

```
self(in).[in=ok]res<a>.nil +
self(in).[in=unknown]res<b>.nil +
self(X).res<c>.nil
```

Improved translation: deterministic and polyadic

```
self(in).[in=ok]res<a>.nil +
self(in,P).[in=req]res<b>.nil +
self(X).[X<>ok]res<c>.nil
```

(`case`: analogous)

Conclusion

Done:

- Developed a π–calculus model which reflects “essential” behaviour of an Erlang program
- Improvement of previous approach:
  - respects order of overlapping patterns (deterministic branching)
  - supports tuple communication

Conclusion

Done:

- Developed a π–calculus model which reflects “essential” behaviour of an Erlang program
- Improvement of previous approach:
  - respects order of overlapping patterns (deterministic branching)
  - supports tuple communication

To do:

- Larger case studies
- Representation of list data structures
- Respect order of messages

Conclusion

Questions please?