Towards Modelling Actor-Based Concurrency in Term Rewriting

Adrián Palacios

(joint work with Germán Vidal)

Technical University of Valencia

2nd Int'l Workshop on Rewriting Techniques for Program Transformations and Evaluation

July 2, 2015

Warsaw, Poland

Actor model

Program: a pool of processes which interact by exchanging messages.

Each process has a local mailbox (not shared).

A process can:

- Send a message to another process.
- Receive a message.
- Update its local state.
- Create new processes.

This is the model underlying Erlang or Scala.

Actor model in sequential TR

Our goal: model an Erlang-like language within sequential TR.

To achieve this, we will describe:

- A process as a term.

$$process(\Box)$$

- A system as a term composed of processes.

$$process(\Box) \odot process(\Box) \odot process(\Box) \odot \ldots$$

Modelling actor-based concurrency in sequential TR

The traditional approach is based on implementing an interpreter (an operational semantics):

- A complex implementation is required.
- A significant overhead is introduced.
- One can end up analyzing the interpreter rather than the model (the model becomes data).

In this paper, instead, we aim at the following:

- We keep the sequential part untouched.
- We introduce only a few rules to deal with concurrent actions (some restrictions will be needed).

Problems with sequential TR

When modelling an Erlang-like language,

- Functional part: Straightforward.
- Concurrent part:
  - Difficult.
  - These actions have side effects.

Language properties

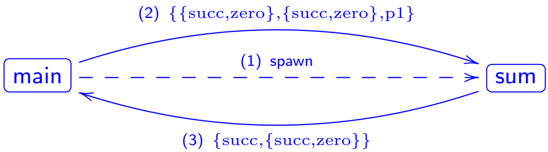
We consider an Erlang-like language with...

- Functional features:
  - Pattern matching.
  - Eager evaluation ($\approx$ innermost rewriting).
  - Evaluation of the first matching clause only.

- Concurrent actions for processes:
  - **self**: Returns the pid of the process.
  - **spawn**: Creates a new process.
  - **send**: Sends a message to a process.
  - **receive**: Find a message from the mailbox that matches the given patterns. Suspend execution if there is no match.

Example of an actor-based program

$$
\begin{array}{rcl}
\mathsf{main}(X,Y) & \to & P = \mathsf{spawn}(\mathsf{sum},[\,]),\\
 & & P\ !\ \{X,Y,\mathsf{self}()\},\\
 & & \mathsf{receive}\\
 & & \quad Z \to Z\\
 & & \mathsf{end}.\\
\mathsf{sum}() & \to & \mathsf{receive}\\
 & & \quad \{N,M,P\} \to P\ !\ \mathsf{add}(N,M)\\
 & & \mathsf{end},\\
 & & \mathsf{sum}().\\
\mathsf{add}(N,M) & \to & \mathsf{case}\ N\ \mathsf{of}\\
 & & \quad \mathrm{zero} \to M\\
 & & \quad \{\mathrm{succ},X\} \to \{\mathrm{succ},\mathsf{add}(X,M)\}\\
 & & \mathsf{end}.
\end{array}
$$



Modelling Concurrency

System specification structure

Definition (System specification structure)

An actor system is specified as a constructor TRS $\mathcal{R} = \mathcal{E} \cup \mathcal{A} \cup \mathcal{S}$ where:

- $\mathcal{E}$ is the functional component.
- $\mathcal{A}$ specifies the evaluation of concurrent actions.
- $\mathcal{S}$ defines a scheduling policy.

Process definition

Definition (Process)

A process is denoted by a term $\mathrm{p}(pid, t, m)$ where:

- p/3 is a constructor symbol.
- *pid* is the process identifier (a constructor constant).
- *t* is the process' term.
- *m* is the mailbox (a list of constructor terms).

$$\mathrm{p}(0, \mathrm{main}(t_1, t_2), [\,])$$

System definition

Definition (System)

A system is denoted by a term $\mathsf{s}(k, m, procs)$ where:

- s/3 is a defined symbol.
- k is a natural number (used to produce fresh pids).
- m is a global mailbox of the system.
- $procs$ is a pool of processes.

$$\mathsf{s}(2, [\,], \mathsf{p}(0, \mathsf{self}(\ldots), [\,]) \odot \mathsf{p}(1, \mathsf{sum}, [\,]))$$

where $\odot$ is an AC constructor symbol.

Definition of concurrent actions

In our specification language, concurrent actions have the form

- self($p[, cont]$)
- spawn($p, expr[, cont]$)
- send($p, t[, cont]$)
- rec($clauses[, cont]$)

where *clauses* is a list of the form $[(pat_1, expr_1), \ldots, (pat_n, expr_n)]$

Unfortunately, reducing these actions could be **problematic**.

Program specification

We expect the user to specify a program like this:

$$
\begin{array}{rcl}
\mathrm{main}(x, y) & \to & \mathrm{spawn}(p, \mathrm{sum}, \\
 & & \quad \mathrm{self}(s, \\
 & & \quad\quad \mathrm{send}(p, d(x, y, s), \\
 & & \quad\quad\quad \mathrm{rec}([\mathrm{clause}(z, z)])))) \\
\mathrm{sum}() & \to & \mathrm{rec}([\mathrm{clause}(d(n, m, p), \mathrm{send}(p, \mathrm{add}(n, m), \\
 & & \quad\quad\quad\quad \mathrm{sum}))]) \\
\mathrm{add}(0, m) & \to & m \\
\mathrm{add}(\mathrm{succ}(n), m) & \to & \mathrm{succ}(\mathrm{add}(n, m))
\end{array}
$$

Problems with original specification

In the previous example, the pid p is passed as an argument to the spawn function.

But p is also used in the send function, where the instantation of p is required.

Two ways of solving this:

- Using narrowing (an extension of rewriting that allows the instantiation of variables in the reduced term).
- Apply some preprocessing to avoid this situation (our approach).

Preprocess of concurrent actions

Basically, for each concurrent action with a continuation, we introduce an auxiliary function to handle this continuation.

E.g., given the following rule:

$$\ell \quad \rightarrow \quad \mathsf{self}(p, cont)$$

the preprocessing will produce the following rules:

$$\ell \rightarrow \mathsf{self}(id, vars)$$

$$\mathsf{fself}(id, vars, p) \rightarrow cont$$

and the system rules take care of calling fself with the appropriate p

Compiled program

$$
\begin{array}{rcl}
\mathsf{main}(x, y) & \rightarrow & \mathsf{spawn}(\mathsf{main1}, [x, y], \mathsf{sum}) \\
\mathsf{fspawn}(\mathsf{main1}, [x, y], p) & \rightarrow & \mathsf{self}(\mathsf{main1}, [x, y, p]) \\
\mathsf{fself}(\mathsf{main1}, [x, y, p], s) & \rightarrow & \mathsf{send}(p, \mathsf{d}(x, y, s), \\
 & & \quad \mathsf{rec}(\mathsf{main1}, [x, y]) \\
 & &) \\
\mathsf{frec}(\mathsf{main1}, [x, y], z) & \rightarrow & z \\
 & & \\
\mathsf{sum} & \rightarrow & \mathsf{rec}(\mathsf{sum1}, [\,]) \\
\mathsf{frec}(\mathsf{sum1}, [\,], \mathsf{d}(n, m, p)) & \rightarrow & \mathsf{send}(p, \mathsf{add}(n, m), \\
 & & \quad \mathsf{sum} \\
 & &) \\
 & & \\
\mathsf{frec}(h, vs, t) & \rightarrow & \mathsf{no\_match}(h, vs) \\
 & & \\
\mathsf{add}(0, m) & \rightarrow & m \\
\mathsf{add}(\mathsf{succ}(n), m) & \rightarrow & \mathsf{succ}(\mathsf{add}(n, m))
\end{array}
$$

System rules

$$\mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{self}(h, vs), ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{fself}(h, vs, pid), ms') \odot ps)$$

$$\mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{spawn}(h, vs, t), ms') \odot ps) \to \mathsf{s}(\mathsf{succ}(k), ms, \mathsf{p}(pid, \mathsf{fspawn}(h, vs, k), ms') \odot \mathsf{p}(k, t, [\,]) \odot ps)$$

$$\mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{send}(pid', t, t'), ms') \odot ps) \to \mathsf{s}(k, ms{+}{+}[\mathsf{m}(pid, pid', t)], \mathsf{p}(pid, t', ms') \odot ps)$$

$$\mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{rec}(h, vs), m : ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{frec}(h, vs, m), ms') \odot ps)$$
$$\mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{no\_match}(h, vs), m : ms') \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{frec}(h, vs, m), ms') \odot ps)$$
$$\mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{no\_match}(h, vs), [\,]) \odot ps) \to \mathsf{s}(k, ms, \mathsf{p}(pid, \mathsf{rec}(h, vs), [\,]) \odot ps)$$

Note that concurrent actions are constructor symbols and not functions.

Conclusion and future work

Conclusion and future work

We have introduced a simple concurrent language that follows the actor model and can be specified within term rewriting

This is an ongoing work. Currently, we are working on the following extensions:

- Formally define the specification language and its properties.
- Prove the correctness of the preprocessing stage and implement it.
- Prove the semantic equivalence between the original concurrent language and its specification in term rewriting.

  (in order to keep the usual semantics, innermost rewriting and priority rules are required)
- . . .

Thanks for your attention!