The Join calculus

A calculus of mobile agents

Martin Mosegaard Jensen

Mobile Computing seminar 2004, DAIMI

Plan

- √ Motivation
- √ The reflexive CHAM
- √ Distribution: locality, migration, failure detection
- √ Observational congruence
- √ Comparison to π
- √ The JoCaml system

Motivation

Match concurrency and distribution:

- √ π has a simple and precise abstract foundation.
- √ Distributed setting: location, migration, failure?

Solution:
Take π, add reflection and notion of locality.
Formal model:
Join and The distributed reflexive CHAM
Implementation:
The JoCaml system

Overview

In the taxomony of the survey, Join has:

- √ Labile processes (names as values, like in π)
- √ Motile processes (notion of location)

Syntax - Join

Terms of the calculus are processes, definitions and join-patterns:

$P \stackrel{def}{=} x\langle\tilde{v}\rangle$ | **def** D **in** P | $P|P$ | **0**

$D \stackrel{def}{=} J \triangleright P$ | $D \wedge D$ | **T**

$J \stackrel{def}{=} x\langle\tilde{v}\rangle$ | $J|J$

Notice the difference from π:
Restriction, reception and replication is combined in join pattern.

The chemical abstract machine

- Higher-order solutions $\mathcal{R} \vdash \mathcal{M}$ of reactions and molecules.
- Structural rules ($\rightleftharpoons$) :
  Reversible (syntactical rearrangements)
  Reduction rules ($\longrightarrow$) :
  Consume terms in the solution (computation step)

Reaction rules - Reflexive CHAM

struc-join	$\vdash P_1 \| P_2$	$\rightleftharpoons$	$\vdash P_1, P_2$
struc-null	$\vdash \mathbf{0}$	$\rightleftharpoons$	$\vdash$
struc-and	$D_1 \wedge D_2 \vdash$	$\rightleftharpoons$	$D_1, D_2 \vdash$
struc-nodef	$\mathbf{T} \vdash$	$\rightleftharpoons$	$\vdash$
struc-def	$\vdash \textbf{def } D \textbf{ in } P$	$\rightleftharpoons$	$D\sigma_{dv} \vdash P\sigma_{dv}$
reduction	$J \triangleright P \vdash J\sigma_{rv}$	$\longrightarrow$	$J \triangleright P \vdash P\sigma_{rv}$

Operational semantics

reduction $J \triangleright P \vdash J\sigma_{rv} \longrightarrow J \triangleright P \vdash P\sigma_{rv}$

In one computation step, reductions:

- √ consume any molecule with a given port pattern
- √ make a fresh copy of their guarded process
- √ substitute its received parameters for the sent names
- √ release the process

Example

$$\vdash\ \textbf{def}\ fruit\langle f\rangle \mid cake\langle c\rangle \triangleright P$$
$$\textbf{in}\ fruit\langle apple\rangle \mid fruit\langle pear\rangle \mid cake\langle pie\rangle$$
$$\rightleftharpoons$$
$$fruit\langle f\rangle \mid cake\langle c\rangle\ \triangleright\ P\ \vdash$$
$$fruit\langle apple\rangle \mid cake\langle pie\rangle \mid fruit\langle pear\rangle$$
$$\longrightarrow$$
$$fruit\langle f\rangle \mid cake\langle c\rangle\ \triangleright\ P\ \vdash$$
$$fruit\langle apple\rangle \mid P_{\{pear/f,pie/c\}}$$

Distribution

Issues:

- Location
- Migration
- Failure detection

Distributed RCHAM

The *distributed* RCHAM is a multiset of CHAMS:
$\| \mathcal{R}_i \vdash \mathcal{M}_i$

with a notion of *local solutions*

Interaction of solutions (**comm**):

$$\vdash_\varphi \; x\langle\tilde{v}\rangle \parallel J \triangleright P \vdash \longrightarrow \vdash_\varphi \; \parallel J \triangleright P \vdash \; x\langle\tilde{v}\rangle \quad (x \in dv[J])$$

(2-step: message transport, may be followed by message treatment (**reduction**))

Location

√ Attach *location names* to local solutions

Location names: $a, b, \ldots \in \mathcal{L}$
Location paths: $\varphi, \psi, \ldots \in \mathcal{L}^*$
Solutions are now labelled: $\mathcal{R} \vdash_\varphi \mathcal{M}$
Define:
$\vdash_\varphi$ is a *sub location* of $\vdash_\psi$ when
ψ is a prefix of φ.

Example: $\vdash_{abc}$ is a sub location of $\vdash_a$

Thus ordered solutions form a tree.

Locations (cont'd)

Location constructor:

$$D \overset{def}{=} \ldots \mid a[D : P]$$

Creation of a sub location (**struc-loc**):

$$a[D : P] \vdash_{\varphi}$$
$$\rightleftharpoons$$
$$\vdash_{\varphi} \parallel \{D\} \vdash_{\varphi a} \{P\}$$

Migration

Concerns the movement of a *location*

Syntax extension:

$$P \overset{def}{=} \ \ldots \mid go\langle b, \kappa\rangle$$

plus a new reduction rule (**move**):

$$a[D : P \mid go\langle b, \kappa\rangle] \vdash_\varphi \ \| \ \vdash_{\psi b}$$
$$\longrightarrow$$
$$\vdash_\varphi \ \| \ a[D : P \mid \kappa\langle\rangle] \vdash_{\psi b}$$

Failure detection

- √ In a realistic setting we need to consider failures

Failure detection

- √ In a realistic setting we need to consider failures
- √ Simple failure model for the π-calculus?

Failure detection

- In a realistic setting we need to consider failures
- Simple failure model for the π-calculus?
- Join model:
  Prohibit reactions inside a failed location

Failure detection

- ✓ In a realistic setting we need to consider failures
- ✓ Simple failure model for the π-calculus?
- ✓ Join model:
  Prohibit reactions inside a failed location
- ✓ So we need to distinguish a failed location...

Representing failures

- Tag failed locations: $\Omega \notin \mathcal{L}$
- Location φ is *dead* if it contains Ω
- The position of Ω in φ denotes the origin of the failure

Failure extensions

New primitives: $halt\langle\rangle$ and $fail\langle\cdot,\cdot\rangle$

Rule for halting (**halt**):

$$a[D : P \mid halt\langle\rangle] \vdash_\varphi \longrightarrow \Omega a[D : P] \vdash_\varphi$$

And for failure detection (**detect**):

$$\vdash_\varphi fail\langle a, \kappa\rangle \parallel \vdash_{\psi\varepsilon a} \longrightarrow \vdash_\varphi \kappa\langle\rangle \parallel \vdash_{\psi\varepsilon a}$$

(if $\psi\varepsilon a$ is dead)

Plan

- Motivation
- The reflexive CHAM
- Distribution: locality, migration, failure detection
- **Observational congruence**
- Comparison to π
- The JoCaml system

Observational congruence

- What is observable?

Observational congruence

- What is observable?
- Capability of a process to emit on free channel names

Observational congruence

- What is observable?
- Capability of a process to emit on free channel names
- Define a reduction relation between processes:

  $$P \longrightarrow P' \stackrel{def}{=} \emptyset \vdash \{P\} \, (\rightleftharpoons^* \longrightarrow \rightleftharpoons^*) \, \emptyset \vdash \{P'\}$$

  and associate an *output barb* $\Downarrow_x$ to free channel names x:

  $$P \Downarrow_x \stackrel{def}{=} x \in fv(P) \ \wedge \ \exists \tilde{v}, \mathcal{R}, \mathcal{M}, \emptyset \vdash P \longrightarrow^* \mathcal{R} \vdash \mathcal{M}, x\langle \tilde{v} \rangle$$

Observational congruence (cont'd)

We can now define the *observational congruence* to be the largest equivalence relation $\approx$ satisfying $\forall P, Q, P \approx Q$:

$$\begin{array}{rcl}
\forall x \in \mathcal{N},\ P \Downarrow_x & \Rightarrow & Q \Downarrow_x \\
P \longrightarrow^* P' & \Rightarrow & \exists Q',\ Q \longrightarrow^* Q' \textit{ and } P' \approx Q' \\
\forall D,\ \mathbf{def}\ D\ \mathbf{in}\ P & \approx & \mathbf{def}\ D\ \mathbf{in}\ Q \\
\forall R,\ R \mid P & \approx & R \mid Q
\end{array}$$

Comparison with the π-calculus

- $\surd$ π is a well-studied reference calculus

Using:

- $\surd$ The observational congruence, and
- $\surd$ A core join-calculus:

$$P \stackrel{def}{=} x\langle u\rangle \mid P_1|P_2 \mid \textbf{def}\ x\langle u\rangle|y\langle v\rangle \ \triangleright\ P_1 \textbf{ in } P_2$$

Join is shown to be as expressive as the asynchronous π-calculus (up to their weak barbed congruences)

Full abstraction

Let $\mathcal{P}_1, \mathcal{P}_2$ be two process calculi, with resp. equivalences $\approx_1 \subset \mathcal{P}_1 \times \mathcal{P}_1, \approx_2 \subset \mathcal{P}_2 \times \mathcal{P}_2$.

$\mathcal{P}_2$ is *more expressive* than $\mathcal{P}_1$ when $\exists$ fully abstract encoding $[\![\ ]\!]_{1\to 2}$ from $\mathcal{P}_1$ to $\mathcal{P}_2$ s.t. $\forall P, Q \in \mathcal{P}_1$:

$$P \approx_1 Q \iff [\![P]\!]_{1\to 2} \approx_2 [\![Q]\!]_{1\to 2}$$

$\mathcal{P}_1$ and $\mathcal{P}_2$ have the *same expressive power* when each one is more expressive than the other

Encoding: $\pi \longleftrightarrow Join$

Method: provide fully abstract encodings:

- $\surd$ $\pi \longrightarrow Join$
- $\surd$ $Join \longrightarrow CoreJoin$
- $\surd$ $CoreJoin \longrightarrow \pi$

Encoding: $\pi \longleftrightarrow Join$

From π to Join, naive encoding:

$$
\begin{aligned}
[\![P|Q]\!]_\pi &\stackrel{def}{=} [\![P]\!]_\pi | [\![Q]\!]_\pi \\
[\![\nu x.P]\!] &\stackrel{def}{=} \mathbf{def}\ x_o\langle v_o, v_i\rangle | x_i\langle\kappa\rangle \ \triangleright\ \kappa\langle v_o, v_i\rangle\ \mathbf{in}\ [\![P]\!]_\pi \\
[\![\bar{x}v]\!]_\pi &\stackrel{def}{=} x_o\langle v_o, v_i\rangle \\
[\![x(v).P]\!]_\pi &\stackrel{def}{=} \mathbf{def}\ \kappa\langle v_o, v_i\rangle \ \triangleright\ [\![P]\!]_\pi\ \mathbf{in}\ x_i\langle\kappa\rangle \\
[\![!x(v).P]\!]_\pi &\stackrel{def}{=} \mathbf{def}\ \kappa\langle v_o, v_i\rangle \ \triangleright\ x_i\langle\kappa\rangle | [\![P]\!]_\pi\ \mathbf{in}\ x_i\langle\kappa\rangle
\end{aligned}
$$

Problem with naive encoding

It should not be possible to observe reception of a message, but..

$$[\![x(u).\bar{x}u]\!]_\pi = \textbf{def}\ \kappa\langle v_o, v_i\rangle \ \triangleright\ x_o\langle v_o, v_i\rangle\ \textbf{in}\ x_i\langle\kappa\rangle$$

$$\not\approx_j \ \mathbf{0}$$

To ensure translation is secure *in all contexts* we need a “firewall” mechanism (in the paper)

Encoding: $\pi \longleftrightarrow Join$

Naive encoding from (core) Join to π:

$$\begin{aligned}
[\![Q|R]\!]_j &\stackrel{def}{=} [\![Q]\!]_j|[\![R]\!]_j \\
[\![x\langle v\rangle]\!]_j &\stackrel{def}{=} \bar{x}v \\
[\![\textbf{def } x\langle u\rangle|y\langle v\rangle \triangleright Q \textbf{ in } R]\!]_j &\stackrel{def}{=} \nu xy.(!x(u).y(v).[\![Q]\!]_j|[\![R]\!]_j)
\end{aligned}$$

This translation also needs a firewall

The JoCaml system

- Extension of Objective Caml
- Primitives for controlling locality, migration and failure detection
- Tight connection to the calculus
- Proposed as the next-generation Internet programming language

Example in JoCaml

$$\textbf{def } fruit\langle f\rangle \mid cake\langle c\rangle \triangleright P$$
$$\textbf{ in } fruit\langle apple\rangle \mid fruit\langle pear\rangle \mid cake\langle pie\rangle$$

is written:

```
let def fruit! f | cake! c =
  print_string (f ^ " " ^ c ^ "\n");
in
spawn {fruit "apple" | fruit "pear"
       | cake "pie"};;
```

A mobile agent - server side

```
let def f x =
  print_string ("["^string_of_int(x)^"] ");
  flush stdout;
  reply x*x in
Ns.register "square" f vartype
;;
let loc there do {}
;;

Ns.register "there" there vartype;
Join.server ()
```

A mobile agent - client side

```
let loc mobile
  do {
    let there = Ns.lookup "there" vartype in
    go there;
    let sqr = Ns.lookup "square" vartype in
    let def sum (s,n) =
      reply (if n = 0
               then s
               else sum (s+sqr n, n-1)) in
    let res = sum (0,5) in
    print_string ("sum 5= "^string_of_int res);
    flush stdout;
  }
```

Further reading

- √ The reflexive CHAM and the join-calculus
- √ A Calculus of Mobile Agents
- √ The JoCaml system

Questions?