Languages and Systems
for Global Computing

MPRI, 4/1/2006

Goals

- global computing can be used to access and synchronize large data, to access large computing resources, to customize groupware environments.
- global computing ⇒ scalability and decentralized systems.
- global computing is a very (too?) ambitious project
- basic theory: concurrent and localized objects, extendible languages and systems, security, etc
- engineering: compiling for several run-times, inter-pointer analysis, distributed garbage collection, etc
- reality and vaporware: Java, .Net, peer-to-peer, etc

Already existing

- agents in AI
- distributed systems
- theory of concurrency: CSP, CCS, π-calculus

Concurrency theory

- concurrent programs are always difficult to understand
- concurrency theory (1978 → 1992) is an elegant theory, mainly interested by non-distributed systems
- distributed systems are asynchronous (no output guards, no broadcasts)
- routing is important in distributed systems
- failure detection has to be handled

Concurrency, Locality and Mobility

- π-calculus is a calculus for reconfigurable (extendible) communicating systems, named "mobile processes".
- its variants make localization more explicit: distributed Join calculus, distributed π-calculus, $\pi 1$-calculus, etc
- the calculus of Mobile Ambients has all its synchronization based on localization.

From π-calculus to Join calculus (1/3)

Suppose we have:

- one sender on location s communicates on channel x,
- several receivers on locations a and b wait for data on channel x,

Then which routing strategy?

- sending one of them, but fairness?
- sending both ⇒ distributed consensus between sender s and receivers a and b.
- protocol for atomic broadcast?

⇒ receivers are uniquely located (per channel name)

≡ point-to-point one-way communications from senders to channel managers

From π-calculus to Join calculus (2/3)

Extra problems

- if x-channel manager dies, where to send a message for x ?
  ⇒ channel managers are always alive ≡ permanent receivers
- in CCS/π-calculus, synchronization acheived by consumption of receivers, E.g. a lock is a channel without receiver during the critical section.
- permanent receivers ⇒ synchronization acheived by waiting for several messages on several channels.

⇒ receivers are guards joining several messages
(as for Petri nets)

From π-calculus to Join calculus (3/3)

Caveat

- remote procedure calls are nearly transparent `[B. Nelson]`
- RPCs → big success for programming
- remote synchronization should also be quasi transparent `[Magic Cap]`
- ⇒ local and remote communication follow the same schemes.

The Join-Calculus Language, release 1.05

See `[Fournet, Gonthier, Maranget]`

ML style (1/2)

```
# let x = 1 ;; Type inference
val x:  int
# let y = x+1 ;;
val y:  int
# do print(x); print(y) Synchronous expr.
12
# let id(x) = reply x ;; Polymorphism
val id:  ⟨α⟩ → ⟨α⟩
# do print(id(1)); print_string (id("hello"))
1hello
# let succ(x) = reply x+1 ;;
val succ:  ⟨int⟩ → ⟨int⟩
# let s = id (succ) ;;
val s:  ⟨int⟩ → ⟨int⟩
# spawn echo(1) Asynchronous expr.
# let e = id (echo)
val e:  ⟨int⟩
```

ML style (2/2)

```
# let f(x,y) = reply x+y, x-y ;; Tuples
val f:  ⟨int × int⟩ → ⟨int × int⟩

# let fib(n) = Recursive let
     if x <= 1 then { reply 1 }
     else { reply fib (n-1) + fib (n-2)}
val fib:  ⟨int⟩ → ⟨int⟩

# let twice (f) = High-order
     let r(x) = reply f(f(x)) in
     reply r
val twice:  ⟨⟨α⟩ → ⟨α⟩⟩ → ⟨⟨α⟩ → ⟨α⟩⟩
```

Concurrency

```
# spawn echo (1) | echo (2) Non determinism

# let fruit (f) | cake (c) = Synchronization
     {print_string(f ^ "_" ^ c ^ "\n");}
val fruit:  ⟨string⟩
val cake:  ⟨string⟩

# spawn fruit ("apple") | fruit ("blueberry") |
       cake ("pie") | cake ("crumble")
apple pie
blueberry crumble or
blueberry pie
apple crumble or ...
```

Local definitions

```
# let count(n) | inc() = count(n+1) | reply to inc
  and count(n) | get() = count(n) | reply n to get
val count:  ⟨int⟩
val inc:  ⟨ ⟩ → ⟨ ⟩
val get:  ⟨ ⟩ → ⟨int⟩

# let new_counter () = Scope extrusion
    let count(n) | inc() = count(n+1) | reply to inc
    and count(n) | get() = count(n) | reply n to get
  in count (0) | reply inc,get
val new_counter:  ⟨ ⟩ → ⟨⟨ ⟩ → ⟨ ⟩ ∗ ⟨ ⟩ → ⟨⟨int⟩⟩⟩
```

Locks

```
# let new_lock () =
    let free() | lock() = reply to lock
    and unlock() = free() | reply to unlock in
    free() | reply lock, unlock
val new_lock:  ⟨ ⟩ → ⟨⟨ ⟩ → ⟨ ⟩ × ⟨ ⟩ → ⟨ ⟩⟩
# spawn ... lock(); ... ; unlock(); ...
```

Barriers

```
# let join1 () | join2 () = reply to join1
                           | reply to join2
# spawn ... join1 (); player1 (); ...
      | ... join2 (); player2 (); ...
```

Full-duplex channels

```
# let new_channel () = Asynchronous ch.
   let send(x) | receive() = reply x to receive in
   reply send, receive
val new_channel:  ⟨ ⟩ → ⟨⟨α⟩ × ⟨ ⟩ → ⟨α⟩⟩

# let new_schannel () = Synchronous ch.
   let send(x) | receive() = reply x to receive
                           | reply to send in
   reply send, receive
val new_schannel:  ⟨ ⟩ → ⟨⟨α⟩ → ⟨ ⟩ × ⟨ ⟩ → ⟨α⟩⟩
```

Distribution

```
# let new_cell_d () = Cell server
    let get() | some(x) = none() | reply x to get
    and put(x) | none() = some(x) | reply to put in
    none() | reply get, put

# do ns.register ("cell_d", new_cell_d)
```

```
# let new_cell_d = ns.lookup ("cell_d") ;; Cell client

# let read, write = new_cell_d() do (
    write ("world");
    write ("hello," ^ read());
    print_string (read());
    print_newline()
  ) ;;
```

Checking types in name service ? ↔ typed marshalling ?

Distribution and mobility (1/2)

```
# let new_cell_m (a) = Cell server
    loc applet
    with get() | some(x) = none() | reply x to get
    and put(x) | none() = some(x) | reply to put in
    init go(a); none()
    end in
    reply get, put

# do ns.register ("cell_m", new_cell_m)
```

```
# let new_cell_m = ns.lookup ("cell") Cell client

# loc user
    init
    let read, write = new_cell_m(user) in {
       write ("world");
       write ("hello," ^ read());
       print_string (read());
       print_newline();
    }
    end
```

`a`, `applet`, `user` **are locations. Subjective moves.**

Distribution and mobility (2/2)

```
# let new_cell_mlog (a) = Cell server
    let log (s) = print_string ("cell" ^ s ^ "\n"); reply to log in
    loc applet
    with get() | some(x) = log ("is empty");
                           none() | reply x to get
    and put(x) | none() = log ("contains" ^ x);
                           some(x) | reply to put in
    init go(a); none()
    end in
    reply get, put

# do ns.register ("cell", new_cell)
```

```
# let new_cell_mlog = ns.lookup ("cell") ;; Cell client

# loc user
    init
    let read, write = new_cell_mlog(user) in {
       write ("world");
       write ("hello," ^ read());
       print_string (read());
    }
    end
```

`log` **keeps on server side.**

The join-calculus

P, Q	$::=$		**processes**
	$\mid$	$x\langle\tilde{v}\rangle$	sending $\tilde{v}$ on x
	$\mid$	$\texttt{def}\ D\ \texttt{in}\ P$	(rec) definition of D in P
	$\mid$	$P \mid Q$	parallel composition
	$\mid$	$\mathbf{0}$	empty process
D, E	$::=$		**definitions**
	$\mid$	$J \triangleright P$	elementary clause
	$\mid$	$D \wedge E$	simultaneous definitions
	$\mid$	$\mathbf{T}$	empty definition
J, J'	$::=$		**join-patterns**
	$\mid$	$x\langle\tilde{v}\rangle$	receiving $\tilde{v}$ on x
	$\mid$	$J \mid J'$	composed patterns

x, v_1, v_2, ... **defined and receiving variables**

Defined variables are bound in $\texttt{def}\ D\ \texttt{in}\ P$
Receiving variables are bound in $J \triangleright P$

Free and bound variables

defined var

$$\begin{aligned}
\mathbf{dv}(\mathbf{T}) &= \emptyset\\
\mathbf{dv}(D \wedge D') &= \mathbf{dv}(D) \cup \mathbf{dv}(D')\\
\mathbf{dv}(J \triangleright P) &= \mathbf{dv}(J)\\
\mathbf{dv}(J|J') &= \mathbf{dv}(J) \cup \mathbf{dv}(J')\\
\mathbf{dv}(x\langle\tilde{v}\rangle) &= \{x\}\\
\mathbf{dv}(a[D:P]) &= \{a\} \uplus \mathbf{dv}(D)
\end{aligned}$$

receiving var

$$\begin{aligned}
\mathbf{rv}(J|J') &= \mathbf{rv}(J) \uplus \mathbf{rv}(J')\\
\mathbf{rv}(x\langle\tilde{v}\rangle) &= \{u \in \tilde{v}\}
\end{aligned}$$

free var

$$\begin{aligned}
\mathbf{fv}(\mathbf{0}) &= \emptyset \qquad \textbf{Processes}\\
\mathbf{fv}(P|P') &= \mathbf{fv}(P) \cup \mathbf{fv}(P')\\
\mathbf{fv}(x\langle v\rangle) &= \{x\} \cup \{u \in \tilde{v}\}\\
\mathbf{fv}(\texttt{def}\ D\ \texttt{in}\ P) &= (\mathbf{fv}(P) \cup \mathbf{fv}(D)) - \mathbf{dv}(D)\\
\mathbf{fv}(a[D:P]) &= \{a\} \cup \mathbf{fv}(D) \cup \mathbf{fv}(P)\\
\mathbf{fv}(go\langle a, \kappa\rangle) &= \{a, \kappa\}
\end{aligned}$$

$$\begin{aligned}
\mathbf{fv}(\mathbf{T}) &= \emptyset \qquad \textbf{Defs}\\
\mathbf{fv}(D \wedge D') &= \mathbf{fv}(D) \cup \mathbf{fv}(D')\\
\mathbf{fv}(J \triangleright P) &= \mathbf{dv}(J) \cup (\mathbf{fv}(P) - \mathbf{rv}(J))
\end{aligned}$$

Structural equivalence and calculus (1/2)

Monoidal rules

$$\begin{aligned}
P \mid Q &\equiv Q \mid P\\
(P \mid Q) \mid R &\equiv P \mid (Q \mid R)\\
P \mid \mathbf{0} &\equiv P\\
D \wedge D' &\equiv D' \wedge D\\
(D \wedge D') \wedge D'' &\equiv D \wedge (D' \wedge D'')\\
D \wedge \mathbf{T} &\equiv D
\end{aligned}$$

Binding rules

$$\begin{aligned}
P \mid \texttt{def}\ D\ \texttt{in}\ Q &\equiv \texttt{def}\ D\ \texttt{in}\ P \mid Q \qquad \mathbf{fv}(P) \cap \mathbf{dv}(D) = \emptyset\\
\texttt{def}\ D\ \texttt{in}\ \texttt{def}\ D'\ \texttt{in}\ P &\equiv \texttt{def}\ D \wedge D'\ \texttt{in}\ P \qquad \textbf{similar}\\
\texttt{def}\ \mathbf{T}\ \texttt{in}\ P &\equiv P
\end{aligned}$$

Structural equivalence and calculus (2/2)

Mononoty

$$\begin{aligned}
P =_\alpha Q &\implies P \equiv Q\\
P \equiv Q &\implies P \mid R \equiv Q \mid R\\
P \equiv Q &\implies J \triangleright P \equiv J \triangleright Q\\
D \equiv D', P \equiv Q &\implies \texttt{def}\ D\ \texttt{in}\ P \equiv \texttt{def}\ D'\ \texttt{in}\ Q
\end{aligned}$$

Reduction rules

$$\texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ J\sigma \mid Q \quad \rightarrow \quad \texttt{def}\ D \wedge J \triangleright P\ \texttt{in}\ P\sigma \mid Q$$

$$P \equiv R \rightarrow S \equiv Q \implies P \rightarrow Q$$

Join-Calculus wrt other calculi (1/2)

wrt the π-calculus `[Milner, Parrow, Walker]`

- one-way channels
- fixed static set of receptors per channel
- permanent definitions
- JC is a subset of the π-calculus easily implementable in a standard distributed environment (Unix/WinXXX). No need for distributed-consensus protocols (Isis-like).
- Simple failures. Channel and receptors fail at same time (permanent failure model)

Join-Calculus wrt other calculi (2/2)

wrt Ambients [Cardelli, Gordon]

- lexically scoped
- communication and migration are orthogonal
- JC = communication, Ambients = administration
- Ambients good for security

wrt π1-calculus [Amadio]

- pi-one relies on a condition on types
- JC based on its syntax
- quasi identical

Join-Calculus with locations

$$D, E ::= \quad \ldots \mid a[D : P]$$

a is a location

Caution: scopes and linearity

- the scope of a in $a[D : P]$ delimited by the enclosing `def` statement
- a location only defined once, e.g. the following definition is illegal

  `def` $a[D : P] \wedge a[E : Q] \triangleright R$ `in` S
- a defined name appears in the join-patterns of a unique location, e.g. the following definition is illegal

  `def` $a[x\langle u\rangle \triangleright P : Q] \wedge b[x\langle v\rangle \triangleright R : S]$ `in` T

Join-Calculus with migrations

$$P, Q ::== \quad \ldots \mid go\langle a, \kappa\rangle$$

current location becomes a sublocation of a, then send a trigger on channel κ

Remarks: **hierarchy**

- a location moves with its sublocations

- if a goes to b, then b must not be a sublocation of a. Syntactic check at compile time (**move lock** freeness).

Join-Calculus and Failures

- permanent failures
- a location fails with its sublocations
- emission or moves from dead sites are impossible
- sending to or moves to dead sites are possible
- failure detection impossible in an asynchronous world [Fisher, Lynch, Paterson], [Chandra, Toueg]
- a trace-semantics equivalent implementation is feasible
- positive information about failures in practice.
- only suicides presently implemented (next version with asynchronous failures ?)
- failures of channels ≠ failures of sites

Failures are a big and large problem ↔ Distributed algorithms? ↔ distributed operating systems ?

Failures should be part of semantics of languages.

Jocaml (1/3)

Interface with the outside world

```
let agent = ref 0 ;;

let def register_me (loc, name, (args:string list)) =
  reply () |
  let name = incr agent; Printf.sprintf
    "%s %d" (match args with [name] -> name | _ -> "Agent") !agent in
  let name =
    match args with
    | s :: l -> s
    | [ ] -> name in
  let name = if String.length(name) > 8 then String.sub name 0 8
    else name in
  let job, kill = make_comp (loc) in
    next (name, job, kill) ;;

let _ =
  Ns.register !ns_name register_me (vartype:
    (Join.location * string * string list -> unit) metatype);
  Join.server () ;;
;;
```

Jocaml (2/3)

```
let _ =
  spawn { counter 0 };
  for i = ww - 1 downto 0 do
    for j = hh - 1 downto 0 do
      spawn { s(i*w,j*w) }
    done
  done ;;

let def make_comp (there) =
  let loc mandel [Quad;Calc]
  def square (i0,j0,w,h) =
    let r = Quad.empty w h limit in
    for i = 0 to w - 1 do
      for j = 0 to h - 1 do
        ...
        Quad.set r i j m
      done
    done;
    reply r to square
  and kill! () = Join.kill Join.here;
  do { Join.go there } in
  reply (square, kill)
```

Jocaml (3/3)

```
let ww = 6 and hh = 6 and let w = size_x () / ww and h = size_y () / hh

let def s!(n,m) | next!(name,job,kill) =
  let w = min w (sx-n) and h = min h (sy-m) in
  print_name (n,m,w,h,name,black) ;
  let def finished r | mutex! () =
    draw_square (name,n,m,w,h,r); job_done ();
    next(name,job,kill) | reply
    or restart () | mutex! () = s(n,m) | reply
  in
  mutex () |
  loc boss do {
    { Join.fail job; restart (); Join.halt (); } |
    { Thread.delay 15.0; restart (); Join.halt (); } |
      let r = job (n/pixel,m/pixel,w/pixel,h/pixel) in
      print_string "job done"; print_newline ();
      finished r;
      Join.halt ();
    }
or killAll! () | next! (name,job,kill) = killAll() | kill()
and counter! n | job_done () =
  { if ww*hh = n+1 then killAll () else counter (n+1) } | reply ()
```

Then go!

Join Research (1/2)

- semantics of equivalence [Fournet, Gonthier]
- labeled transition systems (open JC) [Boreale, Fournet, Laneve]
- semantics of security [Abadi, Fournet, Gonthier]
- types and interference [Conchon, Pottier]
- dynamic ressources [Schmitt]
- implementation JC 1.05 [Fournet, Maranget]
- implementation Jocaml [Fournet, le Fessant, Schmitt]
- compiling join patterns [le Fessant, Maranget]
- distributed runtime (GC) [Fournet, le Fessant]
- control of communication and migration, the M-calculus [Schmitt, Stefani]
- coding of pi-calculus and Ambients [Fournet, Lévy, Schmitt]
- distributed objects [Fournet, Laneve, Maranget, Qin, Rémy]

Join Research (2/2)

- functional nets [Odersky]
- typed marshalling [Leifer, Peskine, Sewell, Wansbrough]
- Petri nets and JC [Bruni, Montanari, Sassone]
- Distributed patterns [Bruni, Montanari]
- Symmetric run-times (P2P) To be done! . . . ML-Donkey [le Fessant]

see `http://join.inria.fr`

Conclusion and Future work

- usefulness of mobility
  Missing the Global Computing Fibonacci
  - worldwide computing
  - customization of groupware applications
  - extendible systems, hot restart
  - distributed games
- in Jocaml: games, mobile editor, hevea
- reconsidering compilation problems
- locality and interference analysis
- connection with security
- correct handling of failures

- mastering Jocaml releases