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, π1-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 $\pi$-calculus to Join calculus (2/3)

Extra problems

- if $x$-channel manager dies, where to send a message for $x$?
  $\Rightarrow$ channel managers are always alive $\equiv$ permanent receivers

- in CCS/$\pi$-calculus, synchronization achieved by consumption of receivers, E.g. a lock is a channel without receiver during the critical section.

- permanent receivers $\Rightarrow$ synchronization achieved by waiting for several messages on several channels.

$\Rightarrow$ receivers are guards joining several messages (as for Petri nets)

From $\pi$-calculus to Join calculus (3/3)

Caveat

- remote procedure calls are nearly transparent [B. Nelson]

- RPCs $\rightarrow$ big success for programming

- remote synchronization should also be quasi transparent [Magic Cap]

$\Rightarrow$ 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 : (a) -> (a)
# do print(id(1)); print_string (id("hello"))

hello
# 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 <= i 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 : ((a) -> (a)) -> ((a) -> (a))
```
Concurrency

```ocaml
# 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

```ocaml
# 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

```ocaml
# 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

```ocaml
# let join1 () | join2 () = reply to join1
    | reply to join2

# spawn ... join1 (); player1 (); ...
| ... join2 (); player2 (); ...

Full-duplex channels

```ocaml
# 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

```OCaml
# 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)
```

```OCaml
# 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)

```OCaml
# 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
  none() | reply get, put
  in
  go(a); none()
end

# 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());
}
end
```

log keeps on server side.

Distribution and mobility (2/2)

```OCaml
# let new_cell_mlog (a) = Cell server
  let log (s) = print_string ("cell" ^ s ^ 
  and 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

# 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
```

The join-calculus

```
P, Q ::= processes
| x(\tilde{v}) sending \tilde{v} on x
| def D in P (rec) definition of D in P
| P | Q parallel composition
| 0 empty process

D, E ::= definitions
| J :: P elementary clause
| D ∧ E simultaneous definitions
| T empty definition

J, J' ::= join-patterns
| x(\tilde{v}) receiving \tilde{v} on x
| J | J' composed patterns

x, v_1, v_2, \ldots defined and receiving variables
```

Defined variables are bound in def D in P
Receiving variables are bound in J :: P

a, applet, user are locations. Subjective moves.
Free and bound variables

<table>
<thead>
<tr>
<th>Defined var</th>
<th>Free var</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dv(T)) = (\emptyset)</td>
<td>(fv(0)) = (\emptyset)</td>
</tr>
<tr>
<td>(dv(D \land D')) = (dv(D) \cup dv(D'))</td>
<td>(fv(P</td>
</tr>
<tr>
<td>(dv(J \triangleright P)) = (dv(J))</td>
<td>(fv(x(v)) = {x} \cup {u \in \tilde{v}})</td>
</tr>
<tr>
<td>(dv(J</td>
<td>J')) = (dv(J) \cup dv(J'))</td>
</tr>
<tr>
<td>(dv(x(v))) = ({x})</td>
<td>(fv(a[D : P]) = {a} \cup fv(D) \cup fv(P))</td>
</tr>
<tr>
<td>(dv(a[D : P])) = ({a} \cup dv(D))</td>
<td>(fv(go(a, \kappa)) = {a, \kappa})</td>
</tr>
<tr>
<td>Receiving var</td>
<td></td>
</tr>
<tr>
<td>(rv(J</td>
<td>J')) = (rv(J) \cup rv(J'))</td>
</tr>
<tr>
<td>(rv(x(\tilde{v}))) = ({u \in \tilde{v}})</td>
<td>(fv(D \land D') = fv(D) \cup fv(D'))</td>
</tr>
<tr>
<td></td>
<td>(fv(J \triangleright P) = dv(J) \cup (fv(P) - rv(J)))</td>
</tr>
</tbody>
</table>

Processes

<table>
<thead>
<tr>
<th>Defined var</th>
<th>Free var</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dv(D \land D')) = (dv(D) \cup dv(D'))</td>
<td>(fv(\text{def } D \text{ in } P) = (fv(P) \cup fv(D)) - dv(D))</td>
</tr>
<tr>
<td>(dv(x(v))) = ({x} \cup {u \in \tilde{v}})</td>
<td>(fv(\text{def } D \text{ in } P) = (fv(P) \cup fv(D)) - dv(D))</td>
</tr>
<tr>
<td>(dv(a[D : P])) = ({a} \cup dv(D))</td>
<td>(fv(go(a, \kappa)) = {a, \kappa})</td>
</tr>
</tbody>
</table>

Structural equivalence and calculus (1/2)

Monoidal rules

- \(P | Q \equiv Q | P\)
- \((P | Q) | R \equiv P | (Q | R)\)
- \(P | \emptyset \equiv P\)
- \(D \land D' \equiv D' \land D\)
- \((D \land D') \land D'' \equiv D \land (D' \land D'')\)
- \(D \land T \equiv D\)

Binding rules

- \(P | \text{def } D \text{ in } Q \equiv \text{def } D \text{ in } P | Q\)
- \(\text{def } D \text{ in } D' \text{ in } P \equiv \text{def } D \land D' \text{ in } P\)
- \(\text{def } T \text{ in } P \equiv P\)

Free var

- \(fv(0)\) = \(\emptyset\)
- \(fv(P | P') = fv(P) \cup fv(P')\)
- \(fv(x(v)) = \{x\} \cup \{u \in \tilde{v}\}\)
- \(fv(a[D : P]) = \{a\} \cup fv(D) \cup fv(P)\)
- \(fv(go(a, \kappa)) = \{a, \kappa\}\)

Structural equivalence and calculus (2/2)

Monotony

- \(P =_\alpha Q \implies P \equiv Q\)
- \(P \equiv Q \implies P | R \equiv Q | R\)
- \(P \equiv Q \implies J \triangleright P \equiv J \triangleright Q\)
- \(D \equiv D', P \equiv Q \implies \text{def } D \text{ in } P \equiv \text{def } D' \text{ in } Q\)

Reduction rules

- \(\text{def } D \land J \triangleright P \text{ in } J_0 | Q \implies \text{def } D \land J \triangleright P \text{ in } P_0 | Q\)
- \(P \equiv R \implies S \equiv Q \implies P \rightarrow Q\)

Join-Calculus wrt other calculi (1/2)

wrt the \(\pi\)-calculus [Milner, Parrow, Walker]

- one-way channels
- fixed static set of receptors per channel
- permanent definitions
- JC is a subset of the \(\pi\)-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

\[ D, E ::= \ldots | a[D : P] \]

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
  \[
  \text{def } a[D : P] \land a[E : Q] \triangleright R \text{ in } S
  \]
- a defined name appears in the join-patterns of a unique location, e.g. the following definition is illegal
  \[
  \text{def } a[x(u) \triangleright P : Q] \land b[x(v) \triangleright R : S] \text{ in } T
  \]

Join-Calculus with migrations

\[ P, Q ::= \ldots | \text{go}(a, \kappa) \]

current location becomes a sublocation of \( a \), then send a trigger on channel \( \kappa \)

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 \( \neq \) failures of sites

Failures are a big and large problem \( \leftrightarrow \) Distributed algorithms?
\( \leftrightarrow \) distributed operating systems ?
Failures should be part of semantics of languages.
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 =
mismatch args with
| s :: l -> s
| [ ] -> 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)

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 resources [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 [Mersley]
- typed marshalling [Leifer, Peskine, Dewall, 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