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.
• 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

Concurrency, Locality and Mobility

• \( \pi \)-calculus is a calculus for reconfigurable (extendible) communicating systems, named “mobile processes”.
• its variants make localization more explicit: distributed Join calculus, distributed \( \pi \)-calculus, \( \pi_1 \)-calculus, etc
• the calculus of Mobile Ambients has all its synchronization based on localization.

From \( \pi \)-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)
\( \equiv \) point-to-point one-way communications from senders to channel managers

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
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.

- 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)

```ml
# let x = 1 ;; Type inference
val x : int
# let y = x+1 ;;
val y : int
# do print(x); print(y) Synchronous expr.
do
  print(x); print(y)
# do print(id(1)); print_string (id("hello"))
  lhello
# let id(x) = reply x ;; Polymorphism
val id : (\alpha \rightarrow \alpha)
# do print(id(1)); print_string (id("hello"))
  lhello
# let succ(x) = reply x+1 ;;
val succ : (\alpha \rightarrow \alpha)
# let s = id (succ) ;;
val s : (\alpha \rightarrow \alpha)
# spawn echo(1) Asynchronous expr.
# let e = id (echo)
val e : (\alpha)
```

ML style (2/2)

```ml
# let f(x,y) = reply x+y, x-y ;; Tuples
val f : (\alpha \times \alpha) \rightarrow (\alpha \times \alpha)
# let fib(n) = Recursive let
  if x <= 1 then { reply 1 }
  else { reply fib (n-1) + fib (n-2)}
val fib : (\alpha) \rightarrow (\alpha)
# let twice f = High-order
  let r(x) = reply f(f(x)) in
  reply r
val twice : ((\alpha) \rightarrow (\alpha)) \rightarrow ((\alpha) \rightarrow (\alpha))
```
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
    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();
end

Distribution and mobility (1/2)

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.

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", 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

applet, user are locations. Subjective moves.

The join-calculus

\[
P, Q ::= \begin{array}{l}
    \text{processes} \\
    \mid x(\bar{v}) \\
    \mid \text{def } D \text{ in } P \\
    \mid P \mid Q \\
    \mid 0 \\
    \text{definitions} \\
    \mid J \triangleright P \\
    \mid D \land E \\
    \mid T \\
    \text{join-patterns} \\
    \mid x(\bar{v}) \\
    \mid J \triangleright J' \\
\end{array}
\]

\begin{array}{l}
    \text{Defined variables are bound in def } D \text{ in } P \\
    \text{Receiving variables are bound in } J \triangleright P
\end{array}

\begin{array}{l}
    x, v_1, v_2, \ldots \text{ defined and receiving variables}
\end{array}

\text{defined and receiving variables}

\begin{array}{l}
    \text{composed patterns}
\end{array}

\text{defined and receiving variables}
Free and bound variables

- **Def var**
  - $\emptyset$ = $\emptyset$
  - $\text{fv}(D) = \text{dv}(D)$
  - $\text{fv}(P') = \emptyset$
  - $\text{fv}(x) = \{x\} \cup \{u \in \bar{v}\}$

- **Free var**
  - $\text{fv}(0) = \emptyset$
  - $\text{fv}(P|P') = \text{fv}(P) \cup \text{fv}(P')$
  - $\text{fv}(x(v)) = \{x\} \cup \{u \in \bar{v}\}$
  - $\text{fv}(\text{def } D \text{ in } P) = \left(\text{fv}(P) \cup \text{fv}(D)\right) - \text{dv}(D)$

- **Living var**
  - $\text{rv}(J') = \text{rv}(J) \cup \text{rv}(J')$
  - $\text{fv}(\top) = \emptyset$
  - $\text{fv}(D \land D') = \text{fv}(D) \cup \text{fv}(D')$
  - $\text{fv}(J \triangleright P) = \text{dv}(J) \cup \left(\text{fv}(P) - \text{rv}(J)\right)$

Structural equivalence and calculus (2/2)

- **Monotony**
  - $P =_{\neq} Q \Rightarrow P \equiv Q$
  - $P \equiv Q \Rightarrow P | R \equiv Q | R$
  - $P \equiv Q \Rightarrow J \triangleright P \equiv J \triangleright Q$

- $D \equiv D', P \equiv Q \Rightarrow \text{def } D \text{ in } P \equiv \text{def } D' \text{ in } Q$

- **Reduction rules**
  - $\text{def } D \land J \triangleright P \text{ in } J\sigma | Q \rightarrow \text{def } D \land J \triangleright P \text{ in } P\sigma | Q$
  - $P \equiv R \rightarrow S \equiv Q \Rightarrow P \rightarrow Q$

Join-Calculus wrt other calculi (1/2)

- **Monoidal rules**
  - $P | Q \equiv Q | P$
  - $(P | Q) | R \equiv P | (Q | R)$
  - $P | \emptyset \equiv P$
  - $D \land D' \equiv D' \land D$
  - $(D \land D') \land D'' \equiv D \land (D' \land D'')$
  - $D \land \top \equiv D$

- **Binding rules**
  - $P | \text{def } D \text{ in } Q \equiv \text{def } D \text{ in } P | Q$
  - $\text{fv}(P) \cap \text{dv}(D) = \emptyset$
  - $\text{def } D \text{ in } D' \text{ in } P \equiv \text{def } D \land D' \text{ in } P$
  - $\text{def } T \text{ in } P \equiv P$
  - similar

- **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)

-Ambients \([\text{Cardelli, Gordon}]\)
-lexically scoped
-communication and migration are orthogonal
-JC = communication, Ambients = administration
-Ambients good for security

\(\pi_1\)-calculus \([\text{Amadio}]\)
-pi-one relies on a condition on types
-JC based on its syntax
-quasi identical

Join-Calculus with locations

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

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

Join-Calculus with migrations

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

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 \([\text{Fisher, Lynch, Paterson}], \text{[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 ↔ 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 = 0 to w - 1 do
    for j = 0 to h - 1 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;
    reply r to square
and kill! () = Join.kill Join.here;
do { Join.go there } in
reply (square, kill)

Jocaml (3/3)

let def \$!(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! () = \$!(n,m) | reply
    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, Corthier]
• labeled transition systems (open JC) [Boreale, Fournet, Laneve]
• semantics of security [Abadi, Fournet, Corthier]
• 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, Qis, 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