JOCAML

A JOIN-CALCULUS IMPLEMENTATION IN OCAML

Locations  --  agents and sites

Channels  --  communication links

Locations organized in hierarchical tree:

. Sites are top-level locations or root

. Agents are nested locations.
Channels

Created by definitions: channels associate messages to processes, in an asymmetric way:

Messages sent on a channel are all received by the same channel definition.

Communication links are kept during migrations, i.e. messages sent on channels are transparently forwarded to the channel definition, from any age.
Channel Types

Asynchronous channels

Synchronous channels (a.k.a. RPC)

- Go primitive triggers migration.
- Fail guard only returns when location is halted or crashed.
Starting Communications

Ns. register to register name values in the name server.

Ns. lookup to request values from their names.

Typechecking

OCAML does static type-checking.

In open systems, dynamic types are used at application interfaces.

var-type provides type info checked at name server.
(SERVER)

let host = Unix.gethostname () ;;
Ns. register "server host" host varType ;;

val host : string = "here.cs.rpi.edu"

Warning: VARTYPE replaced by type string metatype

Using multicast
(Querying name server here.cs.rpi.edu: 20001
-: unit = ()

EXAMPLE
EXAMPLE (continued)

(* CLIENT *)

# let host : string = No.lookup "server host"

   vartype;;

   printf ("The server is on =" ^ host);

   string

   Warning: VARTYPE replaced by type

   string metatype

   Using multicast

   (Querying name server here.cs.rpi.edu:20001

   val host : string = "here.cs.rpi.edu"

   The server is on here.cs.rpi.edu :: unit = ()
**Processes**

- OCaml uses expressions, i.e. terms computed to a value.

- OCaml process execution yields no value: used for side-effects, mainly communication & synchronisation.

\[
\{ \ldots \} \quad \text{Syntax}
\]

\[
\uparrow
\]

expressions and sub-processes.

\[ P \mid Q \] for parallel process composition.

\[ \text{spawn } \{ P \} \] to create new thread executing process \( P \).
CHANNELS

- Asymmetric links between one receiver and multiple senders.
- Channel names can be exchanged.
- Receiver agent can migrate.
  \[ \Rightarrow \text{Semantics remain the same} \]

SYNCHRONOUS CHANNEL DEFINITION

let def name(args) = P(args)

A new thread gets created per msg receive reply statement to return a value.

ASYNCHRONOUS CHANNEL DEFINITION

out def name!(args) = P(args)
Printer Example

(* SERVER *)

let def print s = {
    print_string s; reply };;;
N.s. register "Printer" print var_type;;

(* CLIENT *)

let printer = N.s. lookup "Printer" var_type;;
printer "Hello ";;
printer "I'm the client ";;
Join Patterns

\[
\begin{align*}
\text{let def } & \quad u(a) \mid v(b) = P(a, b) \\
\text{or } & \quad u(a) \mid w(b) = Q(a, b)
\end{align*}
\]

If only one message is present on \( u \), \( P \) and \( Q \) will never happen in parallel.

\( \pi \)-Calculus Channels

\[
\begin{align*}
\text{let def } & \quad \text{new.pi.channel } () = \\
\text{let def } & \quad \text{send } x \mid \text{receive } () = \\
\text{let def } & \quad \text{reply } x \text{ to receive} \\
\text{let def } & \quad \text{reply to send} \\
\text{in reply send, receive } \}
\end{align*}
\]

\[
\text{val new.pi.channel : unit } \to (\ 'a \to \text{unit}) \times (\text{unit } \to \ 'a)
\]
Printer Example with Rendez-vous

(* SERVER *)

let def print! (str, cont) | lock! () =
    {
        print.string str;
        {
            cont() | lock () }
    };;

spawn {
    lock () }
;;

Ns.register "Printer" print var:type;;
( #CLIENT # )

let printer = Ns.lookup "Printer" vartype;
let def cont1!() | cont2!() | cont3!() =
  { exit 0; }

Spawn &
  printer ("Hello", cont1) |
  printer ("beautiful ", cont2) |
  printer ("world ", cont3) }
let def create_ref v∅ =
    let def state! v | get () =
        state v | reply v
    or state! v | set new.v =
        state new.v | reply
    in state v∅ | reply get, set ::

val create_ref : 'a→(unit→'a)×(a→unit) =<ψm>
### Reference Cell Usage in ocaml

```ocaml
# let g0, s0 = create_ref 0
and g1, s1 = create_ref "" ;;

val g0 : unit -> int = <fun>
val s0 : int -> unit = <fun>
val g1 : unit -> string = <fun>
val s1 : string -> unit = <fun>

# print_int (g0 ()) ;;
0-: unit = ()

# s0 5 ;;
-: unit = ()

# print_int (g0 ()) ;;
5-: unit = ()
```

13B
Agents

Sites, agents, and groups of agents in OCaml share a single abstraction: the location.

let loc agent do process
creates a new agent, named agent, executing the thread process.
Agent Example

(* SERVER *)

let def print s = print_string s ; reply ;
let target = Ns.lookup "client" var_type ;

let loc agent do {
  print ("I’m on computer " ^
    Unix. gethostname () ) ;
  go target ;
  print ("I’m on computer " ^
    Unix. gethostname () ) ;
  print_string "Hello" ;
}

AGENT EXAMPLE (continued)

(* CLIENT *)

let loc target do 23 ;;
N.s. register "client" target vartype ;;
let loc agent

def channels
do 
{ process }

Channels are unidirectional. These channel definitions allow agents to receive messages.

Channel bindings never change: communication is independent of the location of sending or receiving agents.
(* SERVER *)

let def create_speaker (l, name) =

    let loc agent
        def print! (s, send) = {
            print_string s ; speak!(send)
        }

        and speak! send = {
            print_string (name ^ " >");
            let s = read_line () in
            send (s, send)
        }

    do { go l; }

    in { reply print, speak } ;
let def talk! (e1, n1) \mid no\_client! () =
    one\_client (e1, n1)
or talk! (e2, n2) \mid one\_client(e2, n2) 
{ let print1, speak1 =
    create\_speaker (e1, n1) in
    let print2, speak2 =
    create\_speaker (e2, n2) in
    speak1 (print2) \mid no\_client () 
    speak2 (print1) ?
}

spawn {no\_client ()} ;

Ns. register "talk" talk vartype ;
Communicating Agent Example (3)

(* CLIENT *)

let username = getenv "USER";;
let talk = Ns. lookup "talk" vartype;;
let loc client do {
    talk (client, username) } ;;
FAILURE MODEL IN TOCAML

- **halt ()** primitive process, atomically halts every process inside of this location.

- **fail there:** primitive detects that location "there" halted and runs P.

- Fail is detected (in current TVCMC implementation) only in same run-time. It is also triggered if run-time containing halting location becomes unreachable or terminates.
Failing Agent Example

let loc_agent
  def say s = print_string s; reply
  do { halt(); } fi
spawn { say "it may work before.\n"; } fi
spawn { fail_agent;
  print_string "location stopped";
  say "it never works after \n"; } fi

val agent: Join.location
val say : string → unit
⇒ the location stopped
LOCATIONS AND BINDING EXAMPLE (1)

# let def cell There:

let def log s =
    print_string (s) ; reply in

let loc applet
    def get() | some! x
        = log ("empty") ; none(
        | reply x
    and put x | none() |
        = log ("contains" | x)
        some x | reply
    do { go there ; none() } in
        reply get , put ;;
LOCATIONS AND BINDING EXAMPLE (2)

Ns.register "cell" cell vartypes
Tom.server()

# let cell = Ns.lookup "cell" vartype
let loc user
  do {
    let get, (put: string->unit) =
      cell user in
      put "world";
      got ("hello " ^ get());
      print.string (get());
      exit 0;
  }