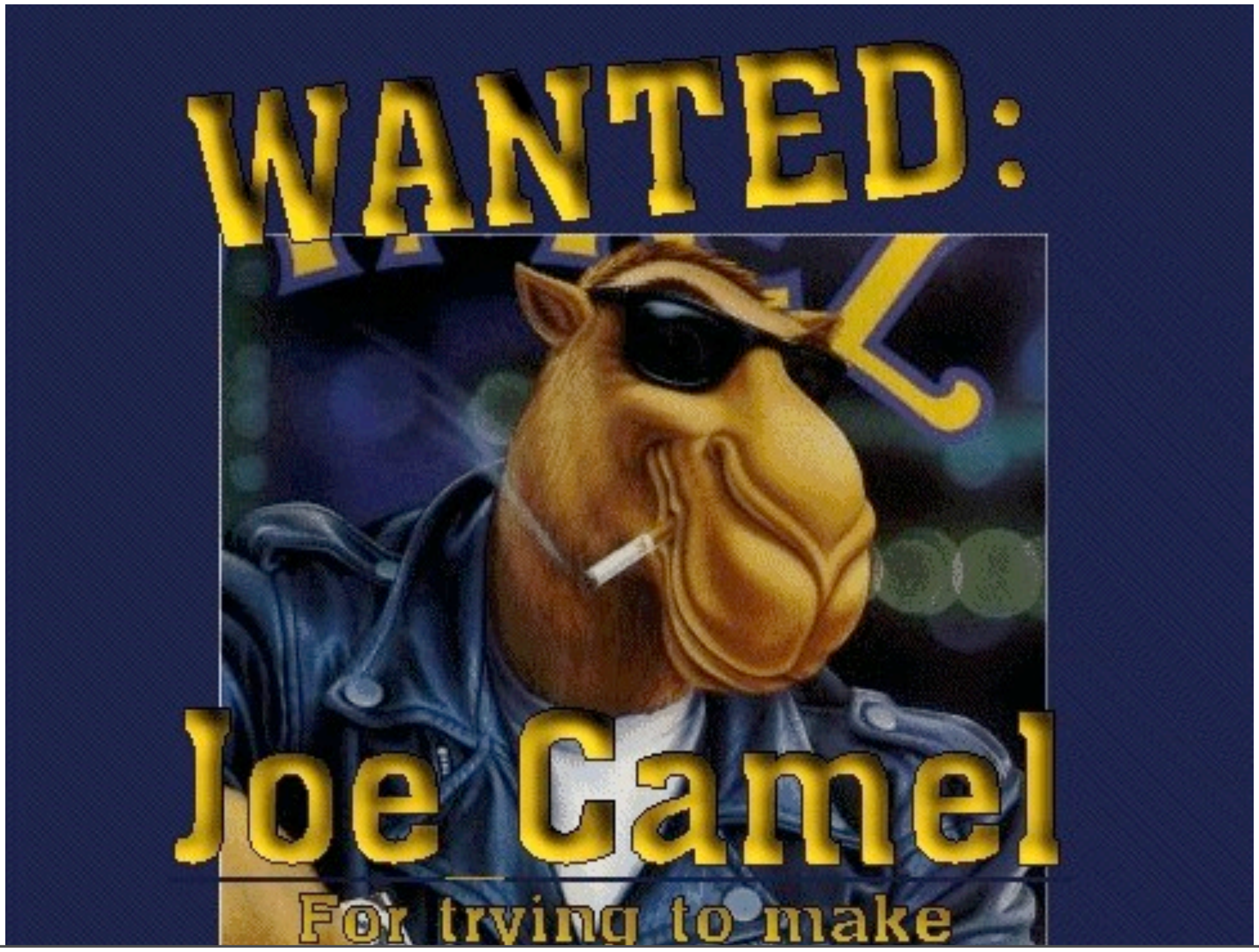
J-O-Caml (6)

jean-jacques.levy@inria.fr

pauillac.inria.fr/~levy/qinghua/j-o-caml

Qinghua, December 4



Plan of this class

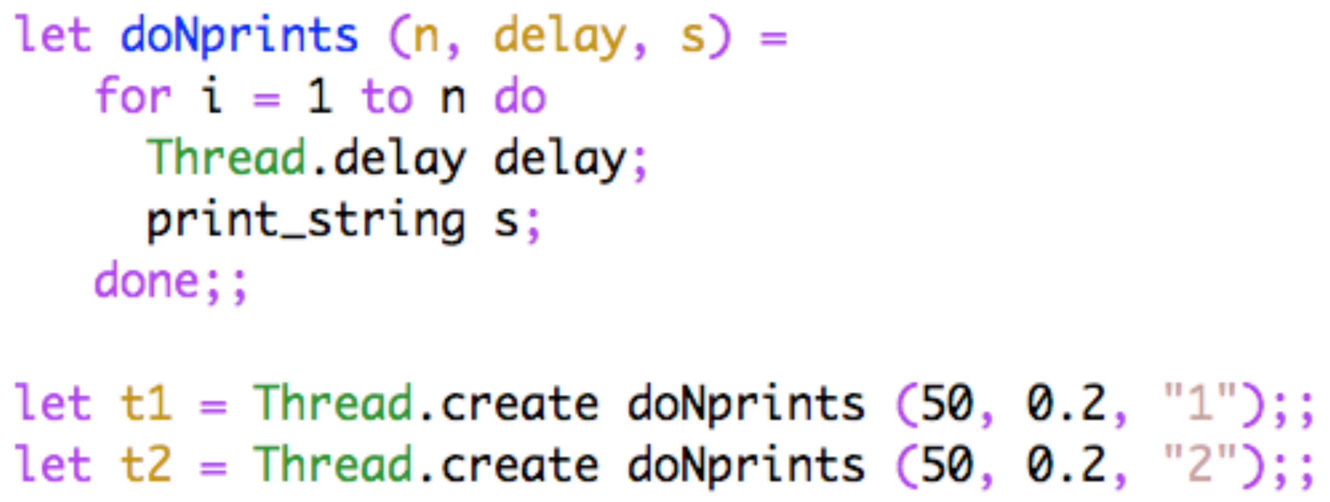
- concurrency
- threads, locks, conditions, shared memory
- communication channels, CML
- distributed setting
- principles of Jocaml
- a concurrent labeling program ?!!!

Exercices

- ???

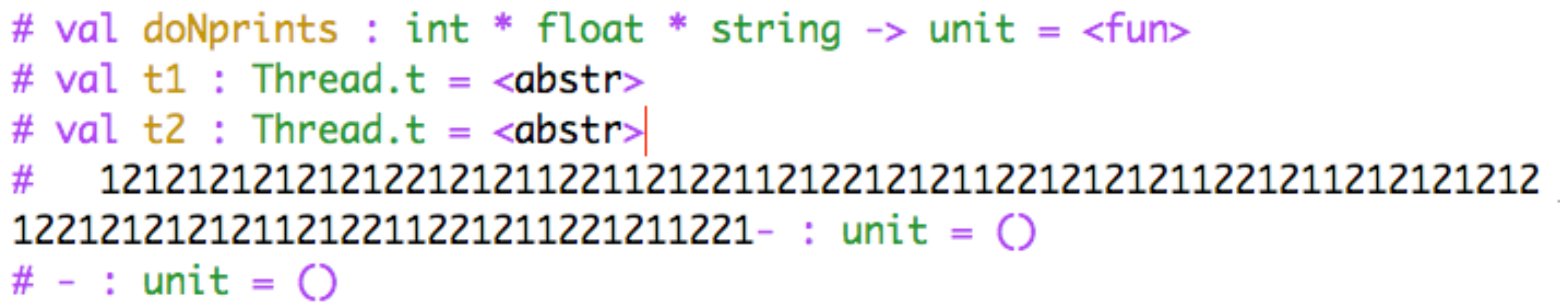
Threads

- light weight processes are implemented in Thread module
- both VM threads (byte code) and Linux threads (binary if available)



```
let doNprints (n, delay, s) =
   for i = 1 to n do
     Thread.delay delay;
     print_string s;
   done;;

let t1 = Thread.create doNprints (50, 0.2, "1");;
let t2 = Thread.create doNprints (50, 0.2, "2");;
```

produces



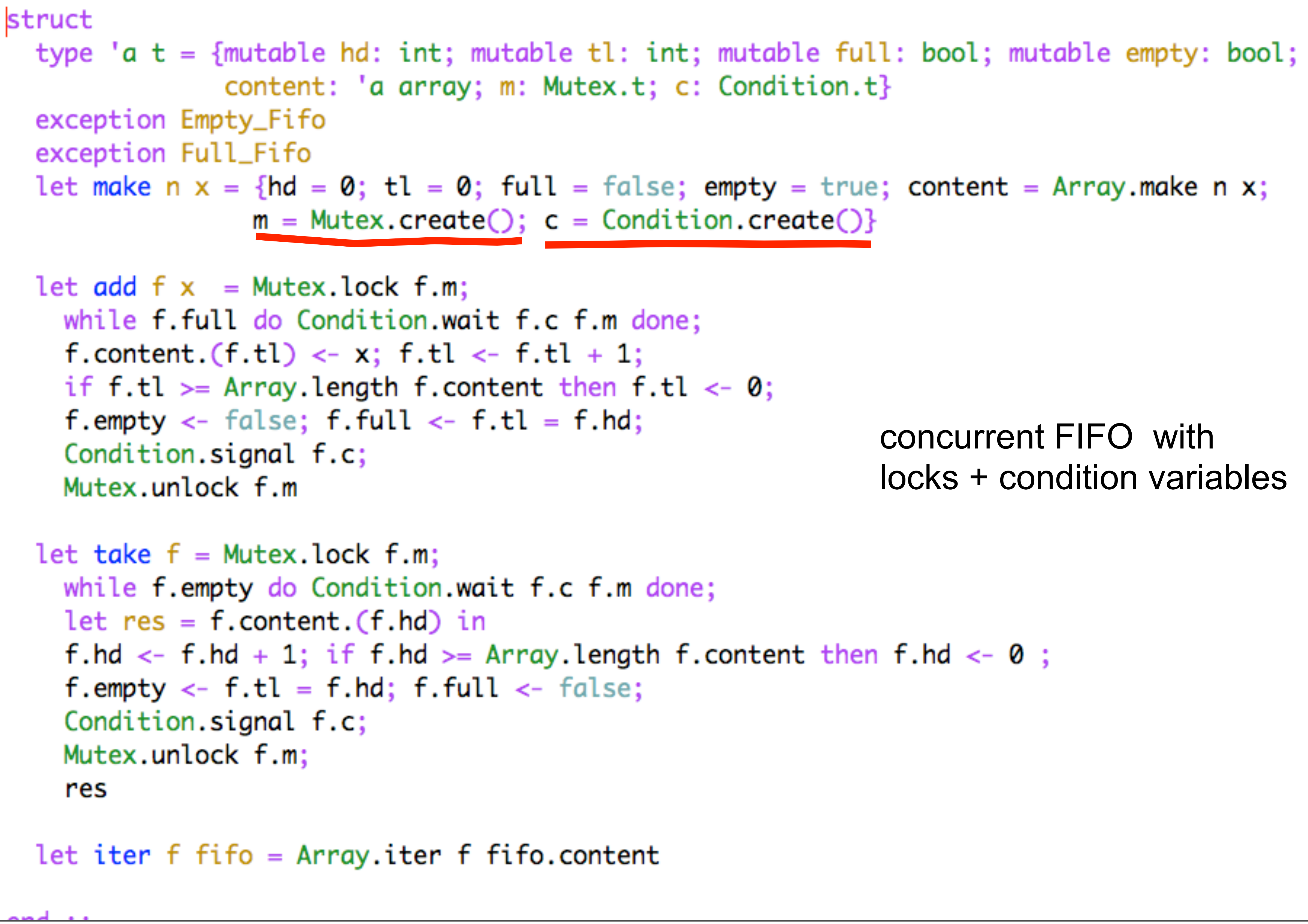
```
# val doNprints : int * float * string -> unit = <fun>
# val t1 : Thread.t = <abstr>
# val t2 : Thread.t = <abstr>
#   1212121212121221212112211212211212221212112212121211221211212121212
1221212121211212211221211221211221- : unit = ()
# - : unit = ()
```

Concurrency - Shared memory

- critical sections -- locks

```
type aircraft = {name : string; seats: bool array } ;;

let m = Mutex.create () ;;

let reserve a =
  Mutex.lock m;
  try
  for i = 0 to (Array.length a.seats) - 1 do
    if not a.seats.(i) then begin
      a.seats.(i) <- true;
      raise Exit;
    end;
  done with
  Exit -> ();
  Mutex.unlock m;;
```

Concurrency - Shared memory



```
struct
  type 'a t = {mutable hd: int; mutable tl: int; mutable full: bool; mutable empty: bool;
               content: 'a array; m: Mutex.t; c: Condition.t}
  exception Empty_Fifo
  exception Full_Fifo
  let make n x = {hd = 0; tl = 0; full = false; empty = true; content = Array.make n x;
                  m = Mutex.create(); c = Condition.create()}

  let add f x  = Mutex.lock f.m;
    while f.full do Condition.wait f.c f.m done;
    f.content.(f.tl) <- x; f.tl <- f.tl + 1;
    if f.tl >= Array.length f.content then f.tl <- 0;
    f.empty <- false; f.full <- f.tl = f.hd;
    Condition.signal f.c;
    Mutex.unlock f.m

  let take f = Mutex.lock f.m;
    while f.empty do Condition.wait f.c f.m done;
    let res = f.content.(f.hd) in
    f.hd <- f.hd + 1; if f.hd >= Array.length f.content then f.hd <- 0 ;
    f.empty <- f.tl = f.hd; f.full <- false;
    Condition.signal f.c;
    Mutex.unlock f.m;
    res

  let iter f fifo = Array.iter f fifo.content

end ;;
```

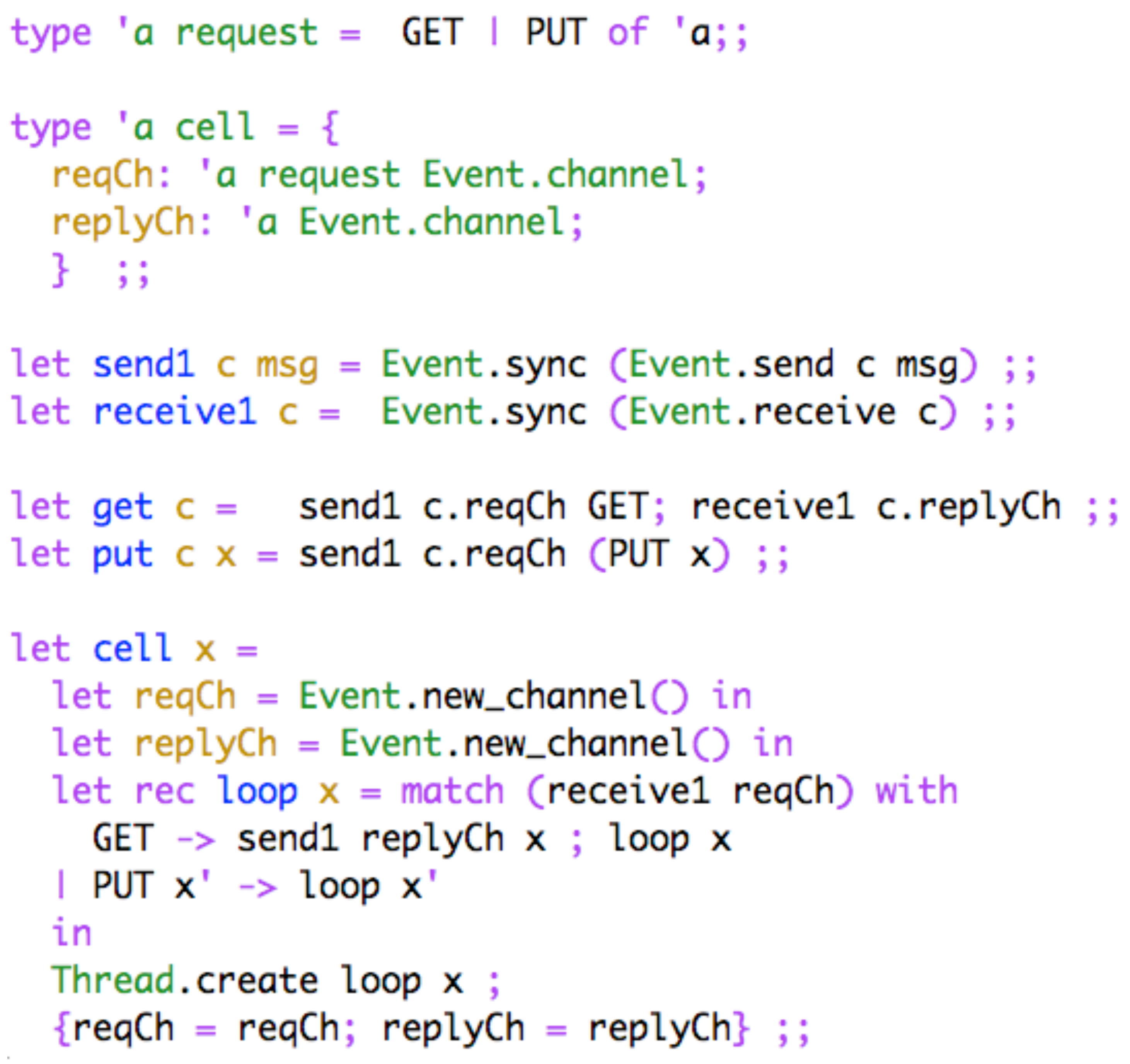
concurrent FIFO with
locks + condition variables

Concurrency - Shared memory

- locks and condition variables are light implementations of old Hoare/Hansen/Dijkstra monitors
- implemented in many languages with shared memory [Modula3, C#]
- curiously missing in Java [locks but no conditions !]
- impossible to write correct version of readers/writers without them

Communication channels

- John Reppy’s `Event` module for CML (SML/NJ) in Ocaml



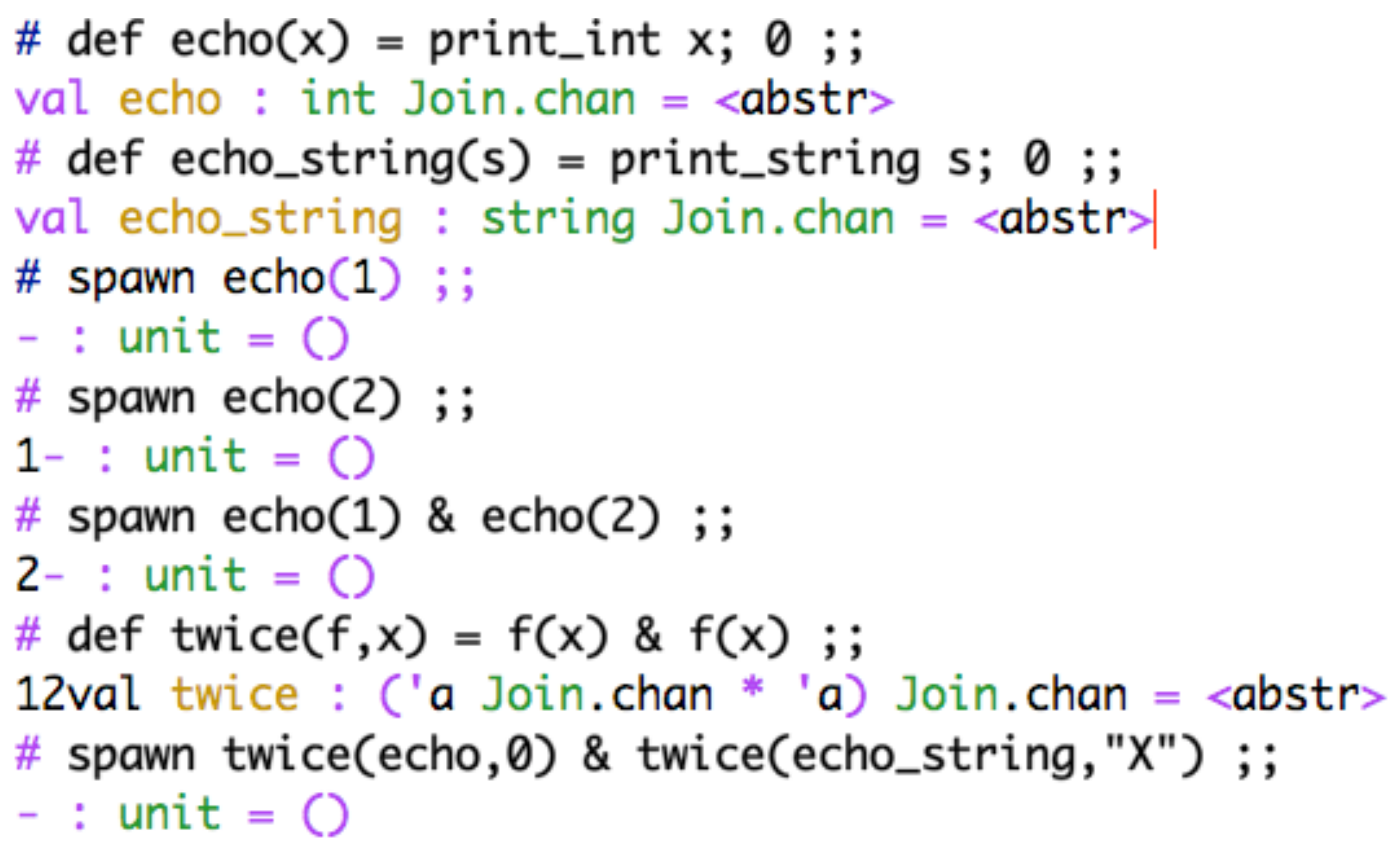
```ocaml
type 'a request =  GET | PUT of 'a;;

type 'a cell = {
  reqCh: 'a request Event.channel;
  replyCh: 'a Event.channel;
  }  ;;

let send1 c msg = Event.sync (Event.send c msg) ;;
let receive1 c =  Event.sync (Event.receive c) ;;

let get c =   send1 c.reqCh GET; receive1 c.replyCh ;;
let put c x = send1 c.reqCh (PUT x) ;;

let cell x =
  let reqCh = Event.new_channel() in
  let replyCh = Event.new_channel() in
  let rec loop x = match (receive1 reqCh) with
    GET -> send1 replyCh x ; loop x
  | PUT x' -> loop x'
  in
  Thread.create loop x ;
  {reqCh = reqCh; replyCh = replyCh} ;;
```

Distributed environments

- asynchronous communications
- rendez-vous on unlocalized channels are impossible because of distributed consensus
- need for join patterns

Join-Patterns

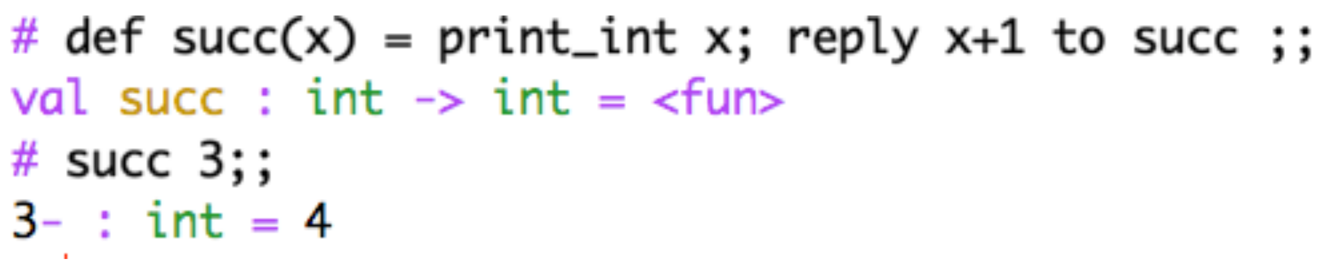
- asynchronous channels



```
# def echo(x) = print_int x; 0 ;;
val echo : int Join.chan = <abstr>
# def echo_string(s) = print_string s; 0 ;;
val echo_string : string Join.chan = <abstr>
# spawn echo(1) ;;
- : unit = ()
# spawn echo(2) ;;
1- : unit = ()
# spawn echo(1) & echo(2) ;;
2- : unit = ()
# def twice(f,x) = f(x) & f(x) ;;
12val twice : ('a Join.chan * 'a) Join.chan = <abstr>
# spawn twice(echo,0) & twice(echo_string,"X") ;;
- : unit = ()
```

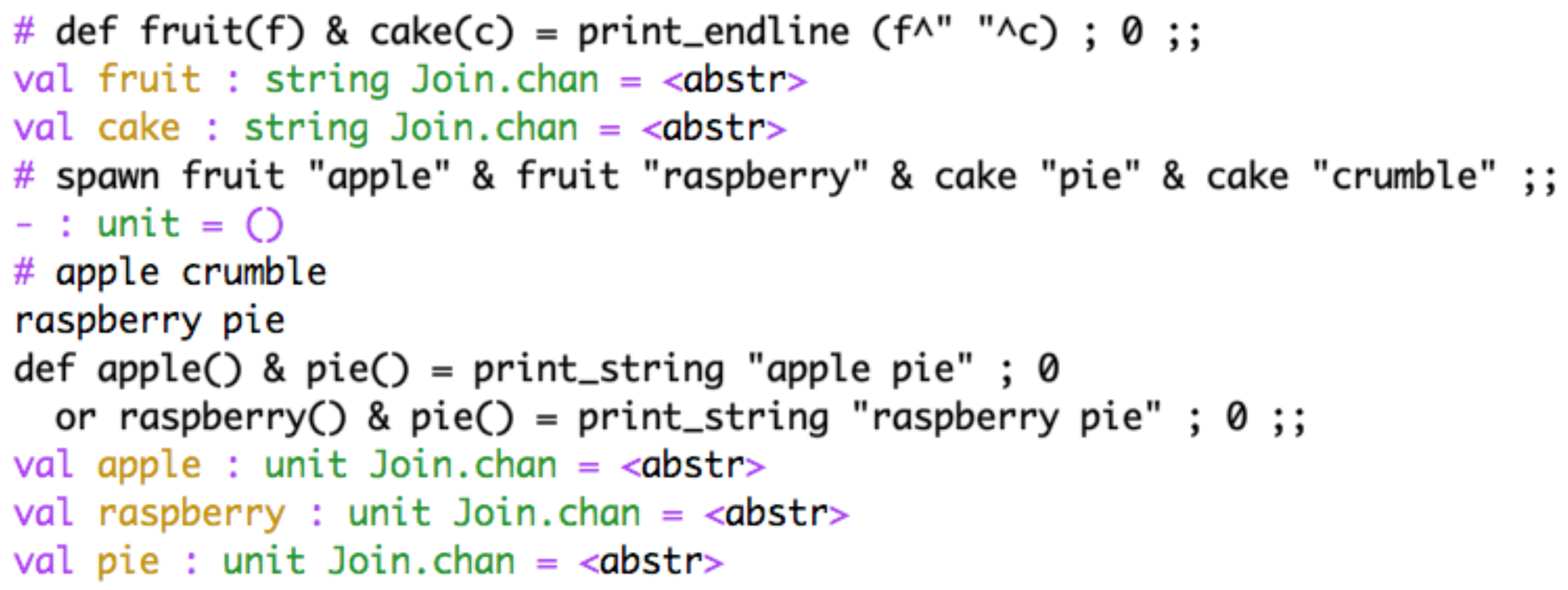
Join-Patterns

- synchronous channels



```
# def succ(x) = print_int x; reply x+1 to succ ;;
val succ : int -> int = <fun>
# succ 3;;
3- : int = 4
```

- join patterns [waiting for presence of several messages]

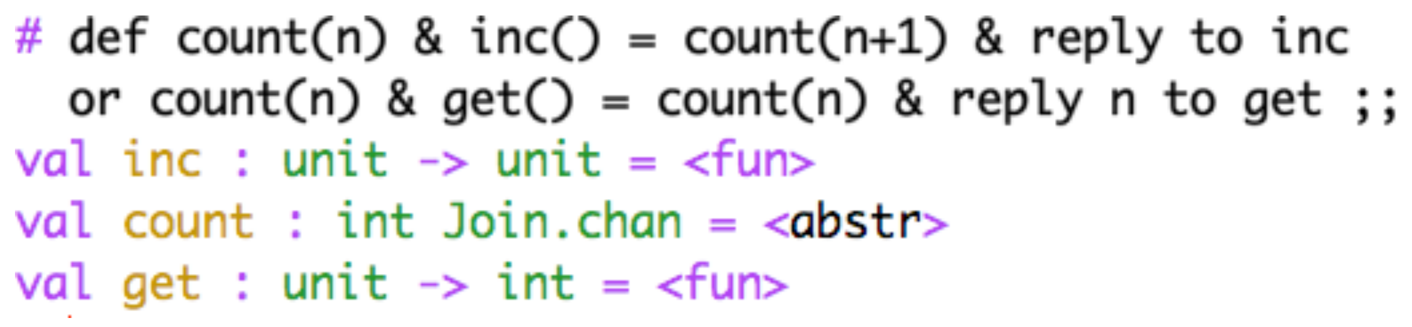


```
# def fruit(f) & cake(c) = print_endline (f^" "^c) ; 0 ;;
val fruit : string Join.chan = <abstr>
val cake : string Join.chan = <abstr>
# spawn fruit "apple" & fruit "raspberry" & cake "pie" & cake "crumble" ;;
- : unit = ()
# apple crumble
raspberry pie
def apple() & pie() = print_string "apple pie" ; 0
  or raspberry() & pie() = print_string "raspberry pie" ; 0 ;;
val apple : unit Join.chan = <abstr>
val raspberry : unit Join.chan = <abstr>
val pie : unit Join.chan = <abstr>
```

Join-Patterns

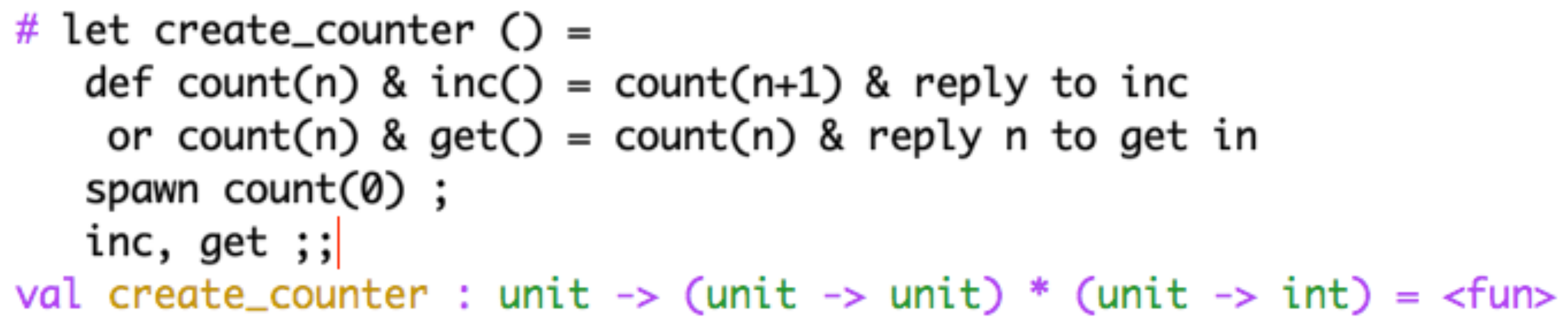
- actors without memory state !

  half of J.-P. Banatre multi-functions paradigm



```
# def count(n) & inc() = count(n+1) & reply to inc
  or count(n) & get() = count(n) & reply n to get ;;
val inc : unit -> unit = <fun>
val count : int Join.chan = <abstr>
val get : unit -> int = <fun>
```

- hiding internal channels



```
# let create_counter () =
   def count(n) & inc() = count(n+1) & reply to inc
    or count(n) & get() = count(n) & reply n to get in
   spawn count(0) ;
   inc, get ;;
val create_counter : unit -> (unit -> unit) * (unit -> int) = <fun>
```

Join-Patterns

- collectors are frequent programming scheme

```
let create_collector f y0 n =
   def count(y,n) & collect(x) = count(f x y,n-1)
   or  count(y,0) & wait() = reply y to wait in
   spawn count(y0,n) ;
   collect, wait ;;

let collect_as_sum n =
   let add, wait = create_collector (+) 0 n in
   add,wait ;;
```

- example of a compute server

```
let create_sum n =
   let add,wait = collect_as_sum n in
   def add_square(x) & register(square) = add(square(x)) & register(square) in
   for i=0 to n-1 do spawn add_square(i) done ;
   register, wait
 ;;

let register, wait = create_sum 32 ;;
spawn register(square2) & register(square1) ;;
print_int (wait ());  ;;
```

Distribution

- server side (with help of nameserver) on `here.inria.fr`

```
def f (x) =
   reply x*x to f
 in Join.Ns.register Join.Ns.here "square" f ;;

let wait =
   def x () & y () = reply to x
   in x ;;

let main =
   Join.Site.listen (Unix.ADDR_INET (Join.Site.get_local_addr(), 12345));
   wait() ;;
```

- client side (with help of nameserver)

```
let server =
   let server_addr = Unix.gethostbyname "here.inria.fr" in
   Join.Site.there (Unix.ADDR_INET(server_addr.Unix.h_addr_list.(0),12345)) ;;

let ns = Join.Ns.of_site server ;;

let sqr = (Join.Ns.lookup ns "square": int -> int) ;;

let rec sum s n = if n = 0 then s else sum (s+sqr(n)) (n-1) ;;
```

Real examples

- ray tracers (given in standard jocaml release)
- one of which won the ICFP programming contest
- slight modifications to get the concurrent or distributed version
- join patterns are polyphonic C# (Benton), VB 9.0 (Russo)
- Jocaml [without mobility] now follows Ocaml releases
- and mobility ?

General philosophy

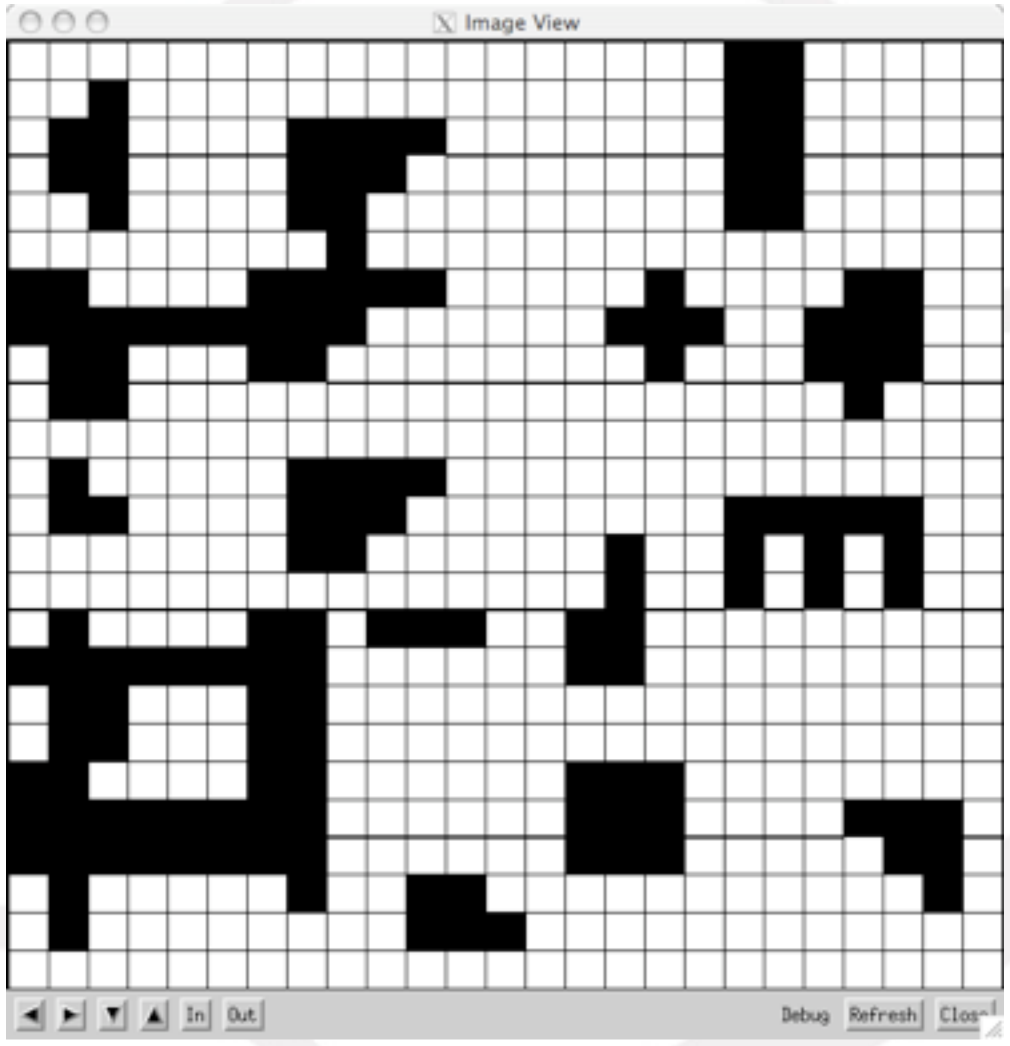
- concurrent programming is very hard
- distribution is impossible to debug
- local and remote behaviors should be identical

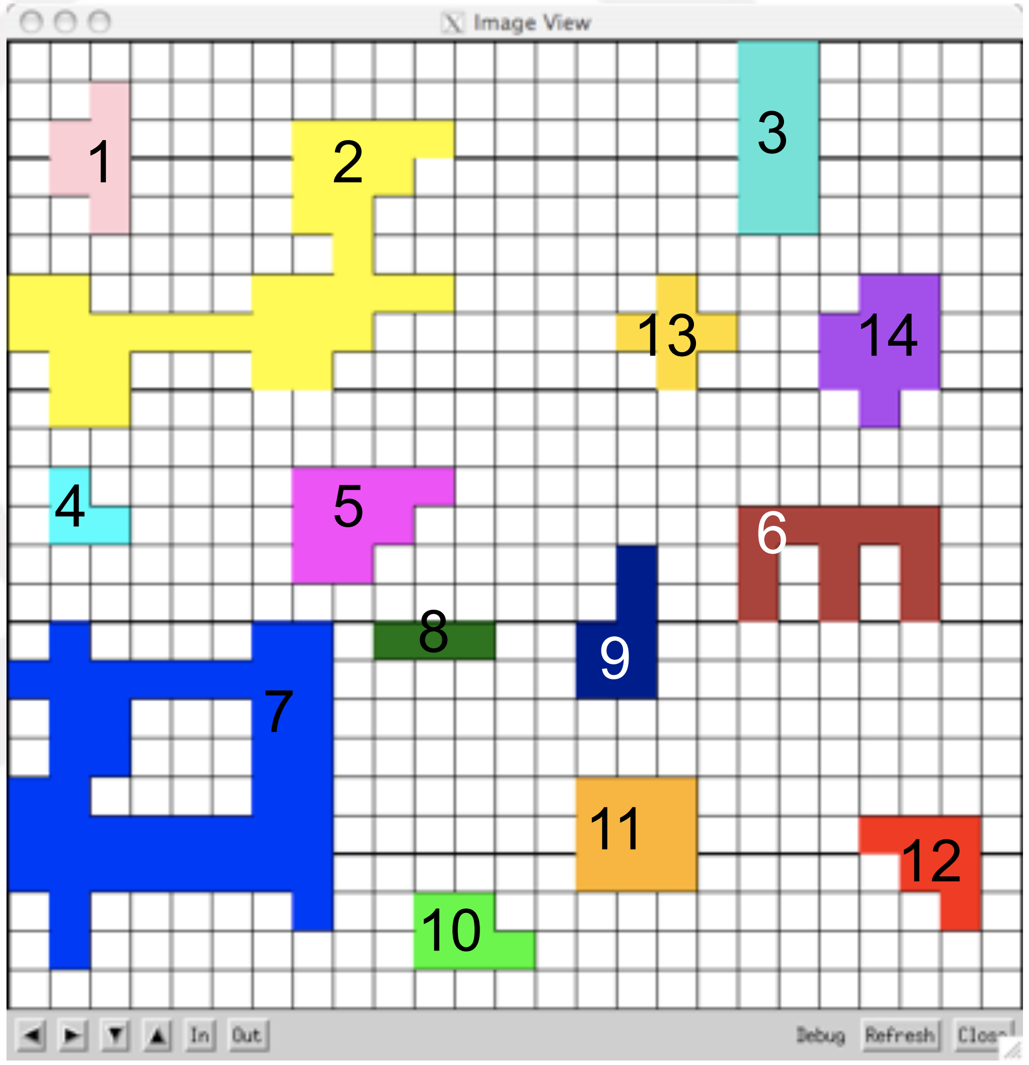
Combien d'objets dans une image?

Jean-Jacques Lévy
INRIA

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH

Labeling





16 objects in this picture

Concurrent algorithm

1) first pass

- scan pixels left-to-right, top-to-bottom giving a new object id each time a new object is met, can be done on dual-core with half image per core.

2) second pass

- generate equivalences between ids due to new adjacent relations met during scan of pixels, taking care of border (generating new equivalences)

3) third pass

- compute the number of equivalence classes and give result in parallel

4) think of the distributed setting with several machines on the net

CENTRE DE RECHERCHE COMMUN
INRIA MICROSOFT RESEARCH