Programmer en JoC

Louis Mandel & Luc Maranget Inria Rocq

Plan

A Introduction à JoCaml(9).

B Lancer de rayons (2).

C Fonctionnel et concurrent (14).

D Distribué (8).

E Quelques détails du monde réel (8).

L'ancien JoCaml (crédit F. Le Fessa

- Basé sur une copie des sources de OCaml 1.07.
- Nombreuses modifications.
- Impossible de suivre le développement de OCaml.

Le nouveau JoCaml

- Le source est une branche des sources de OCaml.
- Le système d'exécution est construit sur `Thread`.

+ Compatibilité : suivi des versions, outils `jocaml`, `jo` `jocamlopt`, threads Posix, compatibilité binaire.

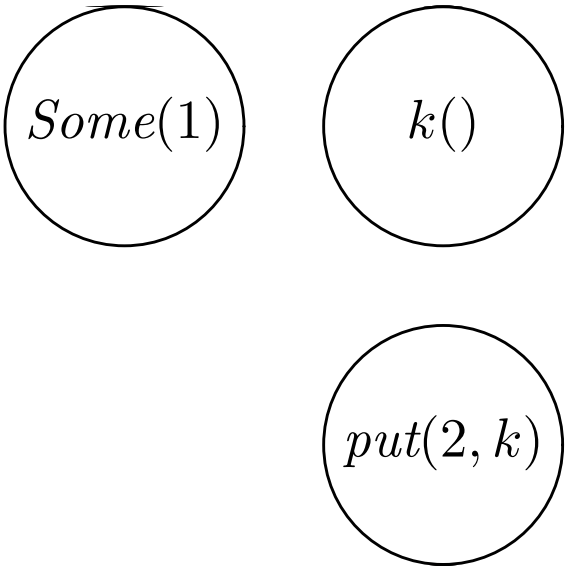
- Fonctionnalités : pas de mobilité de code.

Le buffer à une place, en join-calcu

La définition de règles de réactions :

$$\mathcal{D} = \begin{cases} Some(v) \,\&\, get(k) & \rhd & None() \,\&\, k(v) \\ None() \,\&\, put(v,k) & \rhd & Some(v) \,\&\, k() \end{cases}$$

Les règles s'appliquent à une soupe de molécules.



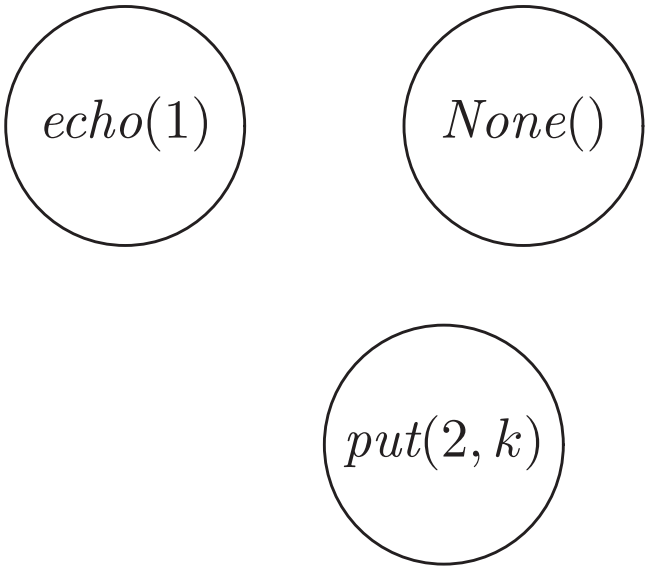
Poursuite du calcul

Supposons la règle :

$$k() \quad \triangleright \quad get(echo)$$

Où `echo` est l'écho sur la console.

On peut finir par obtenir



Le buffer à une place, en JoCaml

```
type 'a buffer = { get : unit -> 'a ; put : 'a

let create_buff () =
  def some(v) & get() = none() & reply v to get
  or none() & put(v) = some(v) & reply () to pu
  spawn none() ;
  { get = get ; put = put ; }
```

- Syntaxe ascii.
- Types de données et constructions de OCaml.
- Processus comme une catégorie syntaxique nouvelle
  « P_1 `&` P_2 », « $c(E)$ », etc. mais `spawn` P est une
- Canaux synchrones — un peu plus que des fonction

Un verrou

```
type lock = { lock : unit -> unit ; unlock : un
(* NB: lock synchrone, unlock asynchrone *)

let create_lock () =
  def lock() & free() = reply to lock
  spawn free() ;
  { lock=lock ; unlock=free ; }
val create_lock : unit -> lock
```

Ne fonctionne que selon la discipline `lock() ; ... ; u`

```
let l = create_lock () in
begin l.lock() ; print_char '(' ; print_char ')
&
begin l.lock() ; print_char '<' ; print_char '>
```

Affiche (nécessairement) « `()<>` » ou « `<>()` ».

Un buffer bien plus utile

État codé comme une paire de listes : `state(xs,ys)`.

```
def state(xs,ys) & put(x) =
      state(x::xs,ys) & reply () to put
 or state(xs,y::ys) & get() =
      state(xs,ys) & reply y to get
 or state(_::_ as xs,[]) & get() =
      state([], List.rev xs) & reply get() to g
val put : '_a -> unit
val state : ('_a list * '_a list) Join.chan
val get : unit -> '_a
```

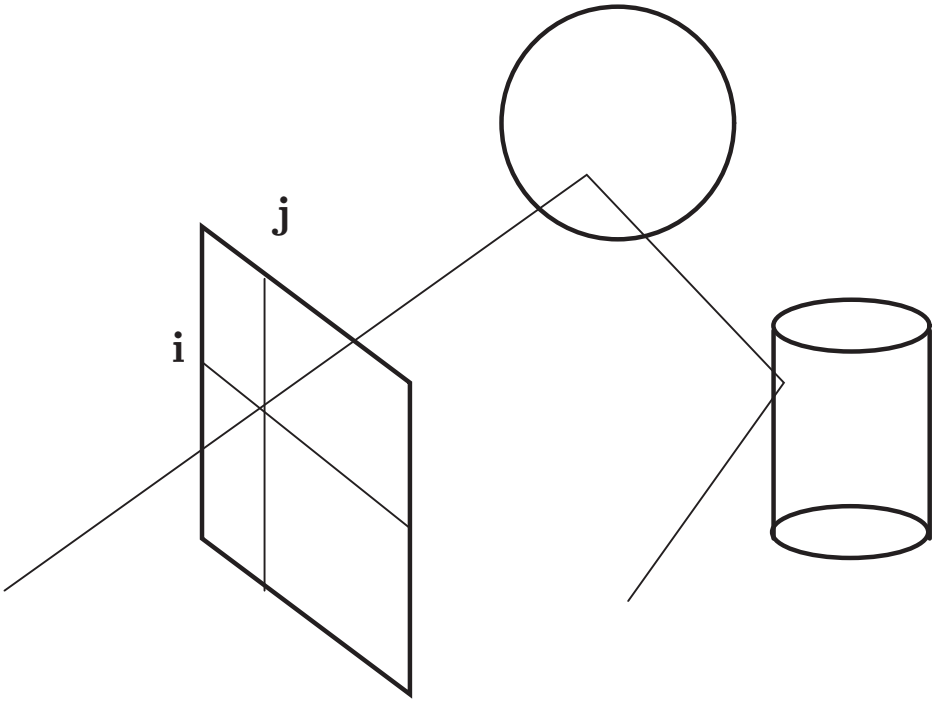
Ou plutôt

```
let create_buffer () =
  def ... in
  spawn state([],[]) ;
  { get=get ; put=put ; }
val create_buffer : unit -> 'a buffer
```

Lancer de rayons

Une technique pour produire des images (bitmap) $h \times w$ à partir de la description de scènes en 3D.

- Scène décrite par un langage ad-hoc (CSG).
- Rendu en suivant le regard $h \times w$ fois.



- En pratique pour nous.

```
let render sc =
  let img = Matrix.create sc.height sc.widt
  for i = 0 to sc.height-1 do
    for j = 0 to sc.width-1 do
       img.(i).(j) <- render_pxl sc i j
    done
  done ;
  save sc.filename img
```

- Ou plutôt :

```
let render_line sc i = ...

let render sc =
  let img = Array.create sc.height [| |] in
  for i = 0 to sc.height-1 do
    img.(i) <- render_line sc i
  done ;
  save sc.filename img
```

Plus général que la boucle `for`

« Énumerateurs » (crédit J.-C. Filliâtre), pour parcouri
éléments `elt` d'une collection `t`.

```
type enum
val start : t -> enum
val step : enum -> (elt * enum) option
```

Par ex. une scène se découpe en lignes :

```
type t = scene
type elt = scene * int (* numéro de ligne *)
type enum = scene * int (* prochaine ligne *)
let start sc = sc * sc.height-1
and step (sc,i) =
  if i < 0 then None
  else Some ((sc,i),(sc,i-1))
```

D'une part, définition d'un « itérateur » sur une collecti

```
let fold worker add y0 t =
  let rec fold_rec y enum = match step enum wit
  | None -> y
  | Some (elt,enum) -> fold_rec (add (worker el
  fold_rec y0 (start c)
val fold : (elt -> 'a) -> ('a -> 'b -> 'b) ->
```

Calcule donc :

$$add(w(x_{n-1}), add(w(x_{n-2}), \ldots add(w(x_0), y_0))$$

Calculer l'image, fonctionnellemen

Le « `fold` » calcule :

$$add(w(x_{n-1}), add(w(x_{n-2}), \ldots add(w(x_0), y_0))$$

D'autre part, calculer l'image.

```
let render sc =
  let img = Array.create sc.height [||] in
  fold
    (fun (sc,i) -> i,render_line sc i) (* worke
    (fun (i,line) () -> img.(i) <- line) (* ado
    () (* y0 *)
    sc ;
  save sc.filename img
```

Programmation concurrente

Écrire un `fold` concurrent, avec

- Les exécutions de `worker` se font concurremment (e
- Plus précisément, plusieurs « *agents* » offrent leur `w`
- `fold` combine les résultats, comme la version séquen

Évidemment, `fold` séquentiel renvoie :

$$add(w(x_{n-1}), add(w(x_{n-2}), \ldots add(w(x_0), y_0))$$

Mais `fold` concurrent ne suit par forcément l'ordre de l'énumérateur.

$$add(w(x_{i_{n-1}}), add(w(x_{i_{n-2}}), \ldots add(w(x_{i_0}), y_0))$$

Dans le cas de `render`, aucune importance.

Exécuter concurremment (&)

Exécuter les $w(x_i)$ en parallèle (sans tenir compte du ré

```
let iter worker t =
  def loop(enum) = match step enum with
    | None -> 0 (* Processus inactif ∅ *)
    | Some (elt, enum) ->
        loop(enum) & begin worker(elt) ; 0 end
  in
  spawn loop(start t)
val iter : (elt -> unit) -> t -> unit
```

Énumérateur simple sur un intervalle entier $[1 \ldots 3]$, et u
simple:

```
let print x = print_char '(' ; print_int x ; pr
```

On obtient `(1)(3)(2)`, ou `(3)(2)(1)`, ou même `((1)2)`

(1) Concurrence sauvage ; (2) impossible de savoir quan

Plusieurs « agents »...

Pour le moment, un travailleur est une fonction particul

```
let print_bis x = print_char '<' ; print_int x
```

Nous voulons que `iter` appelle `printer` et `printer_bis`

Soit le « *pool* », une espèce d'agence de main d'œuvre.

- Coté travailleur : s'enregistrer dans le pool.
- Coté exploitant : demander des travailleurs au pool

```
type pool =
  { register : (elt -> unit) Join.chan ;
    worker : elt -> unit ; }

let pool = create_pool ()

let () = spawn pool.register(print) & pool.regi
```

Contrôle des agents

Le code est assez simple : organiser la rencontre entre `w`
son argument.

```
let create_pool () =
  def compute(elt) & agent(worker) =
    worker x ;
    agent(worker) & reply () to compute in
  { register=agent ; worker=compute ; }
```

Et pour itérer :

```
let () = iter pool.worker (Interval (1,5))
```

Un résultat possible est `<(1>2)<(5>4)<3>`, mais pas `<1`

Collecter les résultats

Dans l'exemple précédent le résultat des `worker` est jeté
voulions en fait :

```
type ('a) pool =
  { register : (elt -> 'a) Join.chan ; worker :

val fold : (elt -> 'a) -> ('a -> 'b -> 'b) ->
```

Avec, par exemple :

```
(* Travailleurs *)
let agent1 x = Printf.sprintf "(%i)" x
and agent2 x = Printf.sprintf "<%i>" x
let () = spawn pool.register(agent1) & pool.reg
(* Exploitant *)
let xs =
  fold pool.worker (fun x xs -> x::xs) [] (Inte
val xs : string list = ["<4>"; "(5)"; "(3)"; "
```

Détour : attendre n_0 événements

La fonction `fold` termine quand tous les travailleurs on
total) n_0 résultats.

Examinons le problème plus simple d'attendre n_0 événe

```
type countdown =
  { tick : unit Join.chan ; wait : unit -> unit

let create_countdown n0 =
  def rem(n) & tick() = rem(n-1)
    (* Compter un événement  *)
  or rem(0) & wait() = reply () to wait in
    (* C'est fini  *)
  spawn rem(n0) ;
    (* État initial *)
  { tick=tick; wait=wait; }
```

Usage du compte à rebours

Afficher les entiers de 1 à 5 (concurremment) et attendr

```
let c = create_countdown 5

def echo(x) = print x; c.tick()

let () =
  print_char '<' ; (* Avant *)
  for k=1 to 5 do spawn echo(k) done ; (* Pendd
  c.wait() ;
  print_char '>' (* Après  *)
```

Affiche par ex. `<(1)(5)(4)(2)(3)>`.

Peu pratique, par ex. inapplicable à `enum` (on ne connaî
nombre d'éléments à l'avance).

Décompter un certain nombre d'événe

On veut un compte à rebours qui ne connaît pas le nom
d'événements à décompter à l'avance.

```
let print_concur xs = (* Afficher la liste xs *
  let c = create_countdown () in (* NB: « () »
  def loop(x::xs) =
    c.enter() ; (* signaler le début d'une tâch
    loop(xs) & (* itération aynchrone *)
    begin (* traitement asynchrone de x *)
      print(x) ;
      c.leave() (* signaler la fin d'une tâche
    end
  or loop([]) = c.finished() (* signaler la fin
  in
  print_char '<' ; (* Avant *)
  spawn loop(xs) ; c.wait() ; (* Pendant *)
  print_char '>' (* Après *)
```

Compte à rebours dynamique

Note : Dans l'usage du compte à rebours.

1. Un `c.enter()` précède un `print(x)` qui est suivi d'
   `c.leave()`.
2. Tous les `c.enter()` précèdent `c.finished()`.

Sous ces conditions, le code suivant est correct.

```
def state(n) & enter() = state(n+1) & reply to
or state(n) & leave() = state(n-1)
or state(0) & finished() & wait() = reply to wa
spawn state(0)
```

Que représente le message `n` dans `state(n)` ? n tâches
(actives).

Du décompteur au *moniteur*

On veut en outre combiner par `add` de type `'a -> 'b -` renvoyer le `'b` final.

```
type ('a, 'b) monitor =
  { enter : unit -> unit ;
    leave : 'a Join.chan ;
    finished : unit Join.chan ;
    wait : unit -> 'b ; }

let create_monitor add y0 =
  def state(n,y) & enter() = state(n+1,y) & rep
  or state(n,y) & leave(v) = state(n-1,add v y)
  or state(0,y) & finished() & wait() = reply y
  spawn state(0,y0) ;
  { enter=enter; ... ; wait=wait; }
```

Au passage

Le décompteur est un moniteur particulier.

```
let create_countdown () =
   create_monitor (fun () () -> ()) ()
```

Ou encore mieux, un décompteur qui compte :

```
let create_countdown () =
  create_monitor (fun () n -> n+1) 0
```

Usage du moniteur

```
let tostring_concur xs = (* transformer en list
  let m = create_monitor (fun x -> x::xs) [] in
  def loop(x::xs) =
    m.enter() ; (* signaler le début d'une tâch
     loop(xs) (* itération aynchrone *)
    & (* traitement asynchrone de x *)
      m.leave(string_of_integer x) (* et signal
  or loop([]) = c.finished() (* signaler la fin
  in
  spawn loop(xs) ; (* intialisation *)
  m.wait() (* récupérer le résultat *)
```

Fold générique, (enfin presque)

On combine la rencontre travailleurs/exploitant et itéra
(internalisation du recrutement en quelque sorte).

```
type ('a,'b) pool =
  { register : (elt -> 'a) Join.chan ;
    fold : ('a -> 'b -> 'b) -> 'b -> t -> 'b }
```

De sorte que par exemple

```
(* Travailleurs *)
let agent1 x = Printf.sprintf "(%i)" x
and agent2 x = Printf.sprintf "<%i>" x
let () = spawn pool.register(agent1) & pool.reg
(* Exploitant *)
let xs = pool.fold (fun x xs -> x::xs) [] (Inte
val xs : string list = ["<4>"; "(5)"; "(3)"; "<
```

Fold, enfin

De `loop(enum)` (ltération asynchrone) à `loop(monitor` (itération contrôlée).

```
let create_pool () =
  def loop(m,enum) & agent(worker) = match step
    | None -> m.finished() & agent(worker)
    | Some (elt,enum) ->
       m.enter() ; call_worker(m,elt,worker) &

  and call_worker(m,elt,worker) =
     let v = worker elt in
     m.leave(v) & agent(worker) in
  { register = agent ;
    fold = (fun add y0 t ->
      let m = create_monitor add y0 in
      spawn loop(m,start t) ;
      m.wait()) ; }
```

Et le lancer de rayons ?

Deux sortes de programmes — de processus Unix, évent
sur des machines différentes.

- Le maître.
  - Crée un pool.
  - Calcule la scène `sc`.
  - Confie le calcul de l'image au pool.
  - Sauve l'image.
  - Termine.
- Les esclaves.
  - S'enregistrent dans le pool.
  - Terminent quand le maître n'a plus besoin d'eux

Certes...

Maître et esclaves doivent avoir un canal en commun.

- Maître : recevoir les offres d'enregistrement. (`regis`
  pool).
- Esclave : s'enregistrer.

C'est tout simplement un envoi de message distant.

« **Sémantique** » **:** Lors d'un envoi à distance, le messa

- Est copié, si c'est une valeur Caml (même mutable)
- N'est pas copié, si c'est un canal (fabrication d'un p

Partager un canal en pratique

- Le maître définit `pool.register` (un canal asynchr
- L'esclave envoie un message sur `pool.register`.

L'esclave connaît `pool.register`, par le truchement du *service* » `Join.Ns`.

Le serveur de noms

Un serveur de noms (de canaux).

```
type t (* Type du serveur de nom *)

val here : t (* Service local *)

val there : Unix.sockaddr -> t (* Service dista

val register : t -> string -> 'a -> unit (* enr

val lookup : t -> string -> 'a (* récupérer un
```

Tout cela est très peu sûr, évidemment.

Code du maître

```
let pool = pool.create ()

let () =
  Join.Ns.register Join.Ns.here "register"
    (pool.register : (enum -> (int * Color.t ar

let render sc =
  let img = Array.create sc.height [||] in
  pool.fold
    (fun (i,line) () -> img.(i) <- line)
    () sc ;
  save sc.filename img
```

Code d'un esclave

```
let master_ns = Join.Ns.there ...

let register : (enum -> (int * Color.t array))
  Join.Ns.lookup master_ns "register"

def worker(sc,i) = reply i,render_line sc i to
(* défini sur l'esclave -> RPC *)

let () = spawn register(worker)
```

Et le code continue en séquence... jusqu'où ?

Et la terminaison ?

Par le truchement du module `Join.Site` qui abstrait le
programmes).

```
type t
val here : t
val there : Unix.sockaddr -> t
(* Du connu *)

val at_fail : t -> unit Join.chan -> unit
(* « at_fail s c » enregistre une garde sur l'e
```

Suite de la séquence d'un esclave

```
let master_site = Join.Site.there ...

let () =
  def wait() & release() = reply to wait in
  Join.Site.at_fail master_site release ;
  wait() ;
  exit 0
```

De petits détails passés sous silenc

Le véritable énumerateur est :

```
type t = scene
type elt = string * int (* nom de la scène, num
type enum = string * int

let start sc = sc.filename, sc.height-1
...
```

- Le maître lit la description de la scène et range l'AS `Join.Ns.here`. Puis il calcule la scène.
- Un esclave récupère la description de la scène et la
- En fait il peut y avoir plusieurs scènes, que maître e désignent par leurs noms (`sc.filename`).
- En fait, un `elt` peut désigner plusieurs lignes.

Un gros détail

Revenons sur l'appel (distant) d'un `worker` :

```
def loop(...) & agent(...) = ...

and call_worker(m,elt,worker) =
  let v = worker elt in
  m.leave(v) & agent(worker)
```

Or, un site distant peut échouer.

Si l'échec est détecté, l'appel `worker elt` lève l'exceptio
`Join.Exit`.

Et... l'image ne sera jamais terminée.

Échec détecté d'un esclave

```
def loop(...) & agent(...) = ...
or compute(m,elt) & agent(worker) = call_worker

and call_worker(m,elt,worker) =
  let v = try Some (worker elt) with _ -> None
  match v with
  | Some v -> m.leave(v) & agent(worker)
  | None -> compute(m,elt)
```

La tâche qui a échoué est simplement ré-émise.

Échec non détecté

L'environnement d'exécution JoCaml fait ce qu'il peut.

Mais il ne peut pas détecter tous les échecs.

Or, un échec non détecté empêche la complétion de l'im

Idée : pas exactement un timeout.

Plutôt, le maître dit :

> *Mieux vaut n esclaves qui font le même travail qu'u*
> *esclave qui travaille et $n-1$ qui regardent.*

Pas très social, mais pas si bête dans un réseau hétérogè

Un moniteur plus bavard

- `enter(elt)` renvoie un identifiant (de tâche) `id`.
- `leave` prend l'argument `(id,v)`.
- Un nouveau canal `get_active()` renvoie les tâches

Un nouveau pool.

```
def loop(m,enum) & agent(worker) = match step e
  | Some (elt,enum) ->
     let id = m.enter elt in
     loop(m,enum) & call_worker(m,id,elt,worker
  | None ->
     m.finished() & agent(worker) & do_again(m)
or do_again(m) = ...

and call_worker(m,id,elt,worker) =
  let v = worker elt in m.leave(id,v) & agent(w
```

Occuper les inoccupés

Le nouveau moniteur possède aussi

- un nouveau canal `is_active(id)`.

```
or do_again(m) = match m.get_active () with
| [] -> 0 (* vraiment plus rien à faire *)
| xs -> again(m,xs) (* refaire xs *)

or again(m,(id,elt)::xs) & agent(worker) =
  again(m,xs) & (* itération concurrente *)
  if m.is_active id then
    call_worker(m,id,elt,worker) (* id pas enco
  else
    agent(worker) (* id fini *)

or again(m,[]) = do_again(m)
```

Le nouveau moniteur

Un état précisé :

- `active` liste de tâches actives (remplace le compte r
- `next_id` le prochain identificateur.
- y comme avant.

Commençons par les nouveaux canaux.

```
def state(next_id,active,y) & is_active(id) =
  state(next_id,active,y) &
  reply List.mem_assoc id active to is_active
or state(next_id,active,y) & get_active() =
  state(next_id,active,y) & reply active to get
```

Et poursuivons par les canaux modifiés.

```
or state(next_id,active,y) & enter(elt) =
    state(next_id+1,(next_id,elt)::active,y) &
    reply next_id to enter

or state(next_id,active,y) & leave(id,v) =
    if List.mem_assoc id active then
      let active = List.remove_assoc id active
      state(next_id,active,add v y)
    else
      state(next_id,active,y)

or state(next_id,[],y) & wait() & finished() =
    reply y to wait & state(next_id,[],y)
```

Ouf

Travaux futurs...

- Programmer en JoCaml.
  - Vous (`http://jocaml.inria.fr/`),
  - Nous.
- Développement de JoCaml.
  - Rattraper l'ancien JoCaml, GC distribué, mobilit
  - Réseau moins naïf (routage, échecs), `camlp3l`, X services, ...