

# Programmation impérative

Luc.Maranget@inria.fr

<http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/>

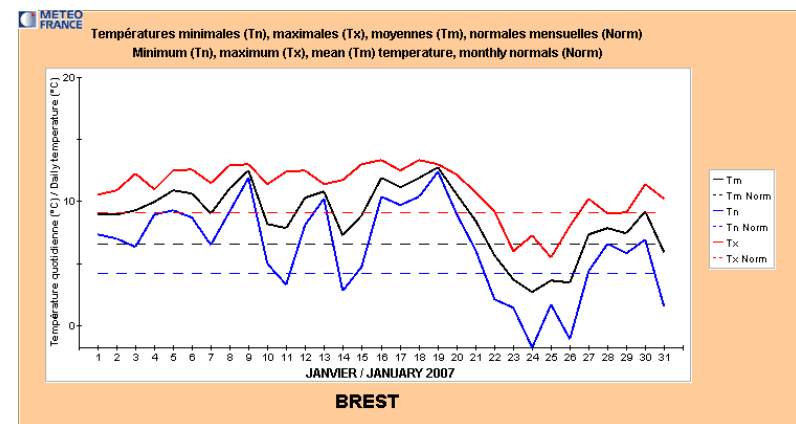
## Des nombres qui changent

- ▶ La capacité d'une salle de concert, vs. le nombre de places restantes.
- ▶ La température de fusion de la glace, vs. la température à Palaiseau.
- ▶ Une variable de PCF, vs. une variable de Java.

- A Des nombres qui changent.
- B Formalisation des construction impératives.
- C Quelques trucs.

## Une fonction du temps

La température à Palaiseau (ou à Brest) est une fonction du temps.



Mais vu de Palaiseau (ou de Brest) on se contente de mesurer (ressentir) la température.

### Le nombre de places disponibles

Change avec le temps.

Veut-on/Peut-on-le modéliser comme une fct `places: Nat -> Nat` ?

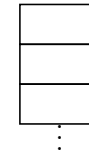
Si oui qqs questions.

- ▶ Quelle heure est-il ?
- ▶ A-t-on réellement besoin de `places(t-10)` ?
- ▶ Peut-on connaître `places(t+10)` ?

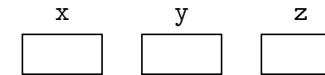
En fait l'ordinateur possède une mémoire à l'état changeant, analogue d'un système physique qui ressent la température à Palaiseau.

### La référence, une adresse en mémoire civilisée

La mémoire est un grand tableau de cases.



On donne des noms aux cases (cellules).



**Let** `x = Ref ... In ...`

La valeur exacte des adresses est sans importance.

Ce sont des *références*, des constantes  $r_1, r_2$  etc.

### Opérations sur les références

- ▶ Création

**Let** `x = Ref 1`

Création d'une nouvelle case. **Noter** : valeur initiale obligatoire (typage, contrôle de la programmation).

- ▶ Regarder dans la case (valeur actuelle).

`!x`

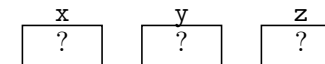
- ▶ Changer le contenu de la case (contribution à la construction de la fonction du temps).

`x := 2`

### Dans un langage impératif

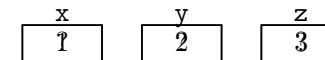
Une variable est toujours une référence, pour créer, suffit de déclarer.

**int** `x, y, z ;`



Changer le contenu d'une case, c'est contribuer à la construction de la fonction du temps.

`x = 1 ; y = 2 ; z = 3 ;`



## Dans un langage impératif

Toutes les variables sont des références.

```
y = x + 1 ; // Affecter y, lire x
```

À gauche de “=”, une variable  $\sim$  une case à affecter, vs. à droite de “=”, une case à lire.

## En PCF

Les opérations « affecter » et « lire » sont plus explicites.

```
t1 := !t2 + 1
```

Le terme  $t_1$  s'évalue en une référence, le terme  $t_2$  s'évalue en une référence. Évidemment,  $t_1$  et  $t_2$  peuvent être des variables.

```
y := !x + 1
```

Dans ce cas, l'environnement doit contenir des liaisons de  $x$  et  $y$  à des références.

## Le monde

Un modèle de la mémoire, ou de fonctions du temps vu de l'intérieur du système.

Un ensemble de paires  $(r, v)$ , tel que  $(r, v) \in m$  et  $(r, v') \in m$  entraîne  $v = v' - r$  référence, peu importe ce que c'est précisément.

L'évaluation est donnée par une relation à cinq places.

$$E, m \vdash t \hookrightarrow v, m'$$

*L'évaluation du terme  $t$  dans l'env.  $E$  et le monde  $m$  produit la valeur  $v$  et change le monde en  $m'$ .*

- $E$  donne la valeur des variables libres de  $t$ .
- Le monde  $m$  donne les valeurs associées aux références.

*Le changement de  $m$  en  $m'$  vient en plus de la valeur  $v$  : rend compte des « effets de bord » (cf. « side effect »).*

## Syntaxe étendue

Cinq nouvelles constructions.

```
type t =
| Num of int | Var of string | Op of op * t * t
| App of t * t | Let of t * string * t
| Fun of string * t | Fix of string * t
(* Syntaxe de PCF usuel *)
| Ref of t (* Nouvelle référence, Ref t *)
| Get of t (* Lecture, !t *)
| Set of t * t (* Affectation, t1 := t2 *)
| Void (* Une nouvelle constante, notée () *)
| Seq of t * t (* Séquence t1; t2 *)
```

## Valeurs étendues

Aux valeurs de PCF usuel (en appel par valeur), entiers  $n$  et fermetures  $\langle x \bullet t \bullet E \rangle$ , on ajoute.

- Des références notées  $r_1, r_2$  etc, qui donnent accès aux paires du monde.
- void, noté encore  $()$ .

```
type env = (string * value) list
and value =
| Num_v of int | Clo_v of string * Ast.t * env
(* Valeurs de PCF usuel *)
| Void_v
| Ref_v of value World.ref
```

## Le module World

Pour le moment, le monde est une valeur (Caml) de type `World.t`

```
type 'a t (* Le type du monde *)
```

```
val create : unit -> 'a t (* Nouveau monde *)
```

```
type 'a ref (* Le type des références *)
```

```
(* Création d'une nouvelle case, renvoie une ref vers icelle *)
```

```
val alloc : 'a t -> 'a -> 'a ref * 'a t
```

```
(* Affectation *)
```

```
val set : 'a t -> 'a ref -> 'a -> 'a t
```

```
(* Lecture *)
```

```
val get : 'a t -> 'a ref -> 'a
```

## Règles des constructions impératives

### Séquence

La constante “void” `()` et la séquence  $t_1 ; t_2$ .

$$E, m \vdash () \hookrightarrow (), m \quad \frac{E, m \vdash t_1 \hookrightarrow v_1, m_1 \quad E, m_1 \vdash t_2 \hookrightarrow v_2, m_2}{E, m \vdash (t_1 ; t_2) \hookrightarrow v_2, m_2}$$

L'enchaînement en séquence de  $t_1$  puis  $t_2$  est exprimé par la production puis l'usage de  $m_1$  (une espèce de relation de Chasles).

Comparer avec :

$$\frac{E \vdash t_1 \hookrightarrow v_1 \quad E \vdash t_2 \hookrightarrow v_2}{E \vdash (t_1 ; t_2) \hookrightarrow v_2}$$

### Variante de la séquence

Si on souhaite ne pas oublier des valeurs  $v_1$  potentiellement significatives.

$$\frac{E, m \vdash t_1 \hookrightarrow (), m_1 \quad E, m_1 \vdash t_2 \hookrightarrow v_2, m_2}{E, m \vdash (t_1 ; t_2) \hookrightarrow v_2, m_2}$$

Ainsi, on exige que  $t_1$  s'évalue en “void”.

Dans ce cas  $(1 ; 2)$  est une erreur de type choisie.

Le contrôle statique des types reste possible (“void” seule valeur du type `unit`).

## Références

Créer une référence, lire une référence, affecter une référence.

$$\frac{E, m \vdash t \hookrightarrow v, m'}{E, m \vdash (\mathbf{Ref} \ t) \hookrightarrow r, (m' \uplus (r, v))}$$

$$\frac{E, m \vdash t \hookrightarrow r, m' \quad (r, v) \in m'}{E, m \vdash !t \hookrightarrow v, m'}$$

$$\frac{E, m \vdash t_1 \hookrightarrow r, m' \uplus (r, -) \quad E, m' \uplus (r, -) \vdash t_2 \hookrightarrow v, m'' \uplus (r, -)}{E, m \vdash t_1 := t_2 \hookrightarrow (), m'' \uplus (r, v)}$$

**Note :** dans la première règle,  $r$  est nouveau, (exprimé par  $\uplus +$  pas de paires avec la même référence), et on exige une valeur initiale.

## Affectation

La règle est particulièrement précise.

$$\frac{E, m \vdash t_1 \hookrightarrow r, m' \uplus (r, -) \quad E, m' \uplus (r, -) \vdash t_2 \hookrightarrow v, m'' \uplus (r, -)}{E, m \vdash t_1 := t_2 \hookrightarrow (), m'' \uplus (r, v)}$$

- $t_1$  est évalué avant  $t_2$ .
- L'affectation change le dernier monde  $m'' \uplus (r, -)$ .

Ainsi, nous connaissons la sémantique de par ex.

```
Let x = Ref 0 In Let y = Ref 1 In
(Ifz !x Then x Else y) := !x + 1
```

Le monde final est  $\{(r_x, 1), (r_y, 1)\}$

## Variante de l'affectation

Pratiquée par Java (et C).

$$\frac{E, m \vdash t_1 \hookrightarrow r, m' \uplus (r, -) \quad E, m' \uplus (r, -) \vdash t_2 \hookrightarrow v, m'' \uplus (r, -)}{E, m \vdash t_1 := t_2 \hookrightarrow v, m'' \uplus (r, v)}$$

Permet ce genre de truc :

```
x = (y = 1) ;
```

À comprendre avec des parenthèses.

## Encore une variation

En C, il est possible de récupérer la référence associée à une case  
Avec l'opérateur « & ».

On peut se hasarder à lui donner une sémantique (au moins dans le cas où on l'applique à une variable.

$$\frac{E(x) = r}{E, m \vdash \&x \hookrightarrow r, m}$$

Le type d'une référence est « pointeur vers le type du contenu de la case », noté  $t^*$ .

```
int x = 1 ;
int * p = &x ; /* p est une référence vers la case de nom x */
*p = 2 ;      /* !p := 2, en quelque sorte */
```

## Le reste des règles

Extension des règles de l'amphi 03. Transporter le monde.

$$E, m \vdash \mathbf{n} \hookrightarrow n, m \quad E, m \vdash (\mathbf{Fun} \ x \rightarrow t) \hookrightarrow \langle x \bullet t \bullet E \rangle, m$$

$$\frac{E, m \vdash t_1 \hookrightarrow n_1, m_1 \quad E, m_1 \vdash t_2 \hookrightarrow n_2, m_2}{E, m \vdash (t_1 \ \mathbf{op} \ t_2) \hookrightarrow (n_1 \ \mathbf{op} \ n_2), m_2}$$

$$\frac{E, m \vdash t_1 \hookrightarrow \langle x \bullet t \bullet E_1 \rangle, m_1 \quad E, m_1 \vdash t_2 \hookrightarrow v_2, m_2 \quad E_1 \oplus [x = v_2], m_2 \vdash t \hookrightarrow v, m_3}{E, m \vdash (t_1 \ t_2) \hookrightarrow v, m_3}$$

$$\frac{E, m \vdash t_1 \hookrightarrow v_1, m_1 \quad E \oplus [x = v_1], m_1 \vdash t_2 \hookrightarrow v_2, m_2}{E, m \vdash (\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2) \hookrightarrow v_2, m_2}$$

$$E, m \vdash (\mathbf{Fix} \ f \rightarrow \mathbf{Fun} \ x \rightarrow t) \hookrightarrow (C = \langle x \bullet t \bullet E \oplus [f = C] \rangle), m$$

## Variante sans la séquence

La séquence  $t_1 ; t_2$  est exprimable comme **Let**  $x = t_1$  **In**  $t_2$ , où  $x \notin \mathcal{F}(t_2)$ .

$$\frac{E, m \vdash t_1 \hookrightarrow v_1, m_1 \quad E \oplus [x = v_1], m_1 \vdash t_2 \hookrightarrow v_2, m_2}{E, m \vdash (\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2) \hookrightarrow v_2, m_2}$$

Car la liaison impose d'évaluer  $t_1$  avant  $t_2$ .

Une construction spécifique de la séquence reste plus parlante.

En Caml, on peut écrire.

**let**  $\_ = t_1$  **in**  $t_2$  (\* Variante qui oublie  $v_2$  \*)

**let**  $() = t_1$  **in**  $t_2$  (\* Variante qui force  $v_1$  vaut void \*)

(NB. Dans les deux cas l'environnement d'évaluation de  $t_2$  n'est pas étendu.)

## Quelques trucs

- ▶ Implémentation du monde, en Caml et en PCF.
- ▶ Coder le **Fix**
- ▶ Constructions impératives et appel par nom.

## Impact du transport du monde

Avant.

$$\frac{E \vdash t_1 \hookrightarrow n_1 \quad E \vdash t_2 \hookrightarrow n_2}{E \vdash (t_1 \ \mathbf{op} \ t_2) \hookrightarrow (n_1 \ \mathbf{op} \ n_2)}$$

Maintenant.

$$\frac{E, m \vdash t_1 \hookrightarrow n_1, m_1 \quad E, m_1 \vdash t_2 \hookrightarrow n_2, m_2}{E, m \vdash (t_1 \ \mathbf{op} \ t_2) \hookrightarrow (n_1 \ \mathbf{op} \ n_2), m_2}$$

L'usage du monde impose de spécifier l'ordre d'évaluation, ici c'est de gauche à droite. On aurait pu choisir l'autre sens, mais on doit choisir.

Et de même pour l'application etc.

## Implémentation du monde

Voici une réalisation où le monde est une liste de valeurs.

```
exception Error of string
type 'a t = 'a list
type 'a ref = int (* Référence = indice dans la liste *)

let alloc w v0 = w@[v0], List.length w

let get w i =
  try List.nth w i
  with Failure _ -> raise (Error "Get")

let rec set w i vi = match i,w with
| 0, _::w -> vi::w
| _, v::w -> v::set w (i-1) w
| _, _ -> raise (Error "Set")
```

## Un bout d'interpréteur

Une règle :

$$\frac{E, m \vdash t_1 \hookrightarrow v_1, m_1 \quad E, m_1 \vdash t_2 \hookrightarrow v_2, m_2}{E, m \vdash (t_1 ; t_2) \hookrightarrow v_2, m_2}$$

```
let rec inter e m t = match t with
...
| Seq (t1,t2) ->
  let _,m1 = inter e m t1 in
  let v2,m2 = inter e m1 t2 in
  v2,m2
| Op (Add,t1,t2) ->
  let n1,m1 = inter_int e m t1 in
  let n2,m2 = inter_int e m1 t2 in
  Num_v (n1+n2),m2
...
```

## Usage du monde

L'usage du monde suit une « relation de Chasles ». Le paramètre monde suit exactement le temps.

```
let rec inter e m t = match t with
...
| Seq (t1,t2) ->
  let _,m = inter e m t1 in
  let v2,m = inter e m t2 in
  v2,m
| Op (Add,t1,t2) ->
  let n1,m = inter_int e m t1 in
  let n2,m = inter_int e m t2 in
  Num_v (n1+n2),m
...
```

## Caml est impératif

On peut implémenter les références de PCF par celles de Caml.

```
type value = ... | Ref_v of value ref

let rec inter e t = match t with
...
| Seq (t1,t2) ->
  let _ = inter e t1 in (* Respecter l'ordre *)
  let v2 = inter e t2 in
  v2
| Op (Add,t1,t2) ->
  let n1 = inter_int e t1 in
  let n2 = inter_int e t2 in
  Num_v (n1+n2)
...
```

Mais un monde explicite reste indispensable pour implémenter PCF impératif dans PCF.

### Liaison dynamique

Le monde donne accès à des valeurs « à un instant donné »

```

Let x = Ref 0 In
Let addx = Fun y -> !x + y In
x := 3 ; addx 1
    
```

Le résultat est 4.

C'est à dire que « !x » est calculé dans un monde  $(r_x, 3)$  (i.e. valeur à l'appel et pas lors de la définition de addx.)

La liaison « dynamique » est une généralisation excessive de ce comportement : l'environnement est traité comme un monde.

```

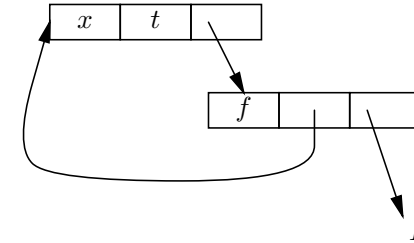
Let x = 0 In (* Env: (x,1) *)
Let addx = Fun y -> x + y In
Let x = 3 In (* Env: (x,3) *)
addx 1 (* x+y évalué dans (x,3);(y,1) *)
    
```

### Fermeture récursive

La valeur de **Fix**  $f \rightarrow$  **Fun**  $x \rightarrow t$  est la fermeture récursive.

$$C = \langle x \bullet t \bullet E \oplus [f = C] \rangle$$

Qui est une notation pour le graphe.



Nous pouvons fabriquer ce graphe avec une référence...

### Le Fix avec le Ref

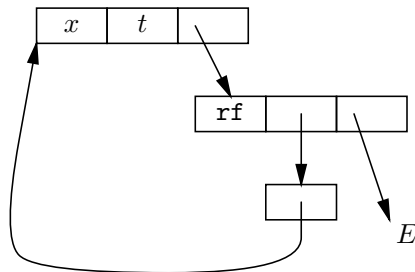
► Construire un arbre.

```

Let rf = Ref (Fun x -> x)
    
```

```

Let pow = Fun x -> Ifz x Then 1 Else 2 * !rf (x-1)
    
```



► Boucler rf := pow.

### Appel par nom

Les règles sont parfaitement définies.

**Rappel :** En appel par valeur.

$$\frac{E(x) = v}{E \vdash x \hookrightarrow_v v} \quad \frac{E \vdash t_1 \hookrightarrow_v v_1 \quad E \oplus [x = v_1] \vdash t_2 \hookrightarrow_v v}{E \vdash \mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \hookrightarrow_v v}$$

⋮

En appel par nom.

$$\frac{E(x) = \langle t \bullet E' \rangle \quad E' \vdash t \hookrightarrow_n v}{E \vdash x \hookrightarrow_n v} \quad \frac{E \oplus [x = \langle t_1 \bullet E \rangle] \vdash t_2 \hookrightarrow_n v}{E \vdash \mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \hookrightarrow_n v}$$

$$\frac{E \vdash t_1 \hookrightarrow_n \langle x \bullet b \bullet E_1 \rangle \quad E_1 \oplus [x, \langle t_2 \bullet E \rangle] \vdash b \hookrightarrow_n v}{E \vdash (t_1 \ t_2) \hookrightarrow_n v} \quad \dots$$



## Le monde en appel par nom

Formellement rien de neuf, on conserve les règles des références.

$$\frac{E, m \vdash t \hookrightarrow v, m'}{E, m \vdash (\mathbf{Ref} \ t) \hookrightarrow r, (m' \uplus (r, v))}$$

$$\frac{E, m \vdash t \hookrightarrow r, m' \quad (r, v) \in m'}{E, m \vdash !t \hookrightarrow v, m'}$$

⋮

Ou l'alternative logique :

$$E, m \vdash (\mathbf{Ref} \ t) \hookrightarrow r, (m \uplus (r, \langle t \bullet E \rangle)) \quad \dots$$

Mais passons.

Et on transporte le monde.

$$\frac{E(x) = \langle t \bullet E' \rangle \quad E', m \vdash t \hookrightarrow v, m'}{E, m \vdash x \hookrightarrow v, m'}$$

$$\frac{E \oplus [x = \langle t_1 \bullet E \rangle], m \vdash t_2 \hookrightarrow v, m'}{E, m \vdash (\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2) \hookrightarrow v, m'}$$

$$\frac{E, m \vdash t_1 \hookrightarrow \langle x \bullet b \bullet E_1 \rangle, m_1 \quad E_1 \oplus [x, \langle t_2 \bullet E \rangle], m_1 \vdash b \hookrightarrow v, m_2}{E, m \vdash (t_1 \ t_2) \hookrightarrow v, m_2}$$

⋯

## Un usage particulièrement délicat

```
Let x = Ref 0
Let id = Fun x -> x
Let kx = Fun y -> !x
```

Soit le terme  $t = (x := !x+1 ; !x)$ .

La valeur de `id t` est 1.

La valeur de `kx t` est 0.

Considérer maintenant :

```
Let f = Fun x -> x + x
```

Et la valeur de `f t`, qui est 3.

Les effets de bord sont retardés : en pratique c'est inutilisable car bien confus.