INF 544 Luc Maranget

Enregistrements, puis objets

Luc.Maranget@inria.fr
http://www.enseignement.polytechnique.fr/profs/informatique/Luc.Maranget/TLP/

Faire des paquets

$$\frac{dx}{dt} = kx \qquad \frac{dy}{dt} = ky \qquad \frac{dz}{dt} = kz$$

Noté

$$\frac{d\vec{u}}{dt} = k\vec{u}, \quad \text{avec } \vec{u} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

En mathématiques deux outils,

- Les couples (prod. cartésien) et les n-uplets, les couples etc..
- Les suites finies. les tableaux.

Et en informatique...

Enregistrement

Une position sur la terre est une paire `latitude` × `longitude`.

Une erreur idiote : **let** `x = (2.208, 47.800)` au lieu de **let** `x = (47.800, 2.208)`.

Un enregistrement est d'abord un n-uplet dont les composantes sont nommées,

```
{ latitude=47.800; longitude=2.208; }
```

Le programmeur n'a plus besoin de se souvenir que la latitude vient en premier, il peut écrire :

```
{ longitude=2.208; latitude=47.800; }
```

Cela évite bien des erreurs idiotes et conduit à des programmes plus clairs : `x.latitude` au lieu de `fst x`.

D'abord une commodité

Dans les languages usuels, les enregistrements expriment le produit (cartésien) de façon commode.

- Déclarer un type enregistrement :

  type `pos = { latitude:float ; longitude:float; }`

  Ce type déclare un produit cartésien, le compilateur sachant en outre que les composantes sont nommées par des étiquettes. (Par ex. le *champ* `latitude` est la première composante etc.)

- L'utilisateur écrit

  let `x = { longitude=2.208; latitude=47.800; }`

  Le compilateur sait qu'il s'agit de la paire `(47.800,2.208)` .

- L'utilisateur écrit `x.latitude`, le compilateur comprend `fst x`.

Une commodité supplémentaire

Ajoutons les altitudes aux positions sur la terre.

type `pos2 = { latitude:float ; longitude:float; z:float; }`

Monter d'un mètre :

```
let up pos =
  { latitude = pos.latitude ;
    longitude = pos.longitude ;
    z = pos.z +. 1.0 ; }
```

C'est assez lourd, on peut abréger en :

let `up pos = { pos` **with** `z = pos.z +. 1.0 }`

Ajouts à la syntaxe abstraite de PCF

Une sorte supplémentaire : les étiquettes (comme des variables, mais ne sont jamais liées).

Des symboles supplémentaires qui fabriquent des termes.

- **Rec** qui prend $2n$ arguments de sortes (étiquettes, terme, ..., étiquettes, termes) : `{` ℓ_1`=`t_1 `;` $\cdots$ ℓ_n`=`t_n`; }`
- **Access** prend un terme et une étiquette : $t.\ell$.
- **CopyWith** (simplifié) qui prend un terme, une étiquette et un terme : `{` t_1 **With** ℓ`=`t_2 `}`.

Définition de l'AST de PCF en Caml.

```
type t = ...
| Rec of (label * t) list
| Access of t * label
| CopyWith of t *  (label * t) list (* Généralisé *)
```


Fabriquer un enregistrement

Syntaxe concrète :

`{ x=1 ; y=2 ; }`

Syntaxe abstraite :

type `t = ... | Rec` **of** `(label * t) list | ...`

Et une nouvelle règle d'interprétation.

$$\frac{E \vdash t_1 \hookrightarrow v_1 \qquad \cdots \qquad E \vdash t_n \hookrightarrow v_n}{E \vdash \{\ell_1 = t_1\,;\, \cdots \ell_n = t_n\,;\} \hookrightarrow \{\ell_1 = v_1;\, \cdots \ell_n = v_n;\}}$$

Donc une nouvelle valeur (notée $\{\ell_1 = v_1; \cdots \ell_n = v_n;\}$).

Note : Dans un contexte typé (simple), la valeur d'un enregistrement serait un n-uplet.

La taille n et la position des champs sont définis à partir d'une déclaration de type enregistrement.

Accéder à un champ

Syntaxe concrète

Let `pos = ... In`
`v.x + v.y`

Syntaxe abstraite :

type `t = ... | Access of t * label | ...`

$$\frac{E \vdash t \hookrightarrow \{\ldots \ell = v; \ldots\}}{E \vdash t.\ell \hookrightarrow v}$$

La copie (avec changement)

Syntaxe concrète

Let `pos = ... In`
`{ pos With x = 3 }`

Syntaxe abstraite :

type `t = ... | CopyWith t * label * t | ...`

$$\frac{E \vdash t_1 \hookrightarrow \{\ldots \ell = v_1; \ldots\} \qquad E \vdash t_2 \hookrightarrow v_2}{E \vdash \{t_1 \textbf{ With } \ell = t_2\} \hookrightarrow \{\ldots \ell = v_2; \ldots\}}$$

- t_1 *doit* s'évaluer en un enregistrement qui possède un champ ℓ.
- On peut généraliser la construction :

$$\{t \textbf{ With } \ell_1 = t_1\,;\ \cdots \ell_n = t_n\,;\ \} \stackrel{\text{def}}{=} \{\cdots\{t \textbf{ With } \ell_1 = t_1\}\cdots \textbf{ With } \ell_n = t_n\}$$

Définition par « sucre syntaxique »

Sur le transparent précédent nous avons « défini »

$$\{t \textbf{ With } \ell_1 = t_1\,;\ \cdots \ell_n = t_n\,;\ \} \stackrel{\text{def}}{=} \{\cdots\{t \textbf{ With } \ell_1 = t_1\}\cdots \textbf{ With } \ell_n = t_n\}$$

« Sucre syntaxique » veut dire qu'il n'y pas de nœud correspondant à la construction définie dans l'AST de PCF.

La nouvelle construction existe slt. dans la syntaxe concrète.

Par abus de langage, on peut considérer un AST généralisé.

type `t = ... | CopyWith t * (label * t) list | ...`

Et que la sémantique de la nouvelle construction se déduit facilement (ici la nouvelle règle est lourde).

Une restriction injustifiée ?

En PCF le code suivant est pour le moment interdit :

`{ { x=1 } With y=2 }`

Pourquoi ? À cause de la sémantique de **CopyWith**, (justifiée par l'implémentation efficace dans le cas typé.)

Mais, pour PCF non-typé, on ajoute facilement la règle :

$$\frac{E \vdash t_1 \hookrightarrow r \qquad r = \{\cdots\} \qquad \ell \notin \mathcal{L}(r) \qquad E \vdash t_2 \hookrightarrow v_2}{E \vdash \{t_1 \textbf{ With } \ell = t_2\} \hookrightarrow \{\cdots \ell = v_2\}}$$

Avec la notation $\mathcal{L}(\{\ell_1 = v_1; \cdots \ell_n = v_n;\}) = \{\ell_1, \ldots, \ell_n\}$.

Encore un peu plus loin

Dans le cadre non-typé, l'accès $t.\ell$ réussit dès que t s'évalue en un enregistrement qui possède le champ ℓ.

$$\frac{E \vdash t \hookrightarrow \{\ldots \ell = v; \ldots\}}{E \vdash t.\ell \hookrightarrow v}$$

Comment transporter cette bonne propriétée à PCF typé :

```
type pos2 = { x:float; y:float; }
  ...
type pos3 = { x:float; y:float; x:float; }
  ...
let getx p = p.x (* getx s'applique à pos2 et pos3 ? *)
```

Il faut une forme de *sous-typage*.

Genre `getx` prend un `p` qui comprend au moins le champ `x` (type `pos` = `{x:float;...}`) et `pos2` et `pos3` sont des sous-type `pos`.

Le point sur le typage des enregistrements

- Dans tous les cas, la présence des enregistrements introduit de nouvelles erreurs de type à l'exécution. Par ex : `4.x`.
- Mais un typage simple des enregistrements (celui de C et de Caml par ex.) refuse de laisser passer des termes qui n'échouent pas.
  - ▷ `{ {x=1} With y=2 }`.
  - ▷ Et aussi

```
Let p1 = { x = 1 ; y = 2 ; } In
Let p2 = { x = 1 ; y = 2 ; z = 3 ; } In
Let getx = Fun p -> p.x In
getx p1 + getx p2
```

- Le sous-typage (des types d'enregistrements) offre une solution.

  t_1 sous-type de t_2 veut dire essentiellement que t_1 a plus de champs que t_2.

Détour — Appel par nom

Le principe général est d'évaluer aussi tard que possible.

Les règles sont donc logiquement.

$$E \vdash \{\ell_1 = t_1; \cdots \ell_n = t_n;\} \hookrightarrow \{\ell_1 = \langle t_1 \bullet E\rangle; \cdots \ell_n = \langle t_n \bullet E\rangle;\}$$

$$\frac{E \vdash t \hookrightarrow \{\ldots \ell = \langle t_1 \bullet E_1\rangle; \ldots\} \qquad E_1 \vdash t_1 \hookrightarrow v}{E \vdash t.\ell \hookrightarrow v}$$

Il y a peu à dire en fait.

```
Let x = { a = fact 10 } In
x.a + x.a
```

(`fact 10` évalué deux fois en appel par nom.)

Programmation objet

Une petite entreprise `X`.

D'une part un salarié à payer.

```
let travailleur = { nom = "luc" ; salaire = ... }
```

D'autre part des commandes.

```
let commande = { quoi = "feutre" ; combien=1 ; prix=... }
```

Éditer des fiches de paie/des bons de commande

- Écrire des fonctions `imprimer_fiche`, `imprimer_bon`, etc.
- Ou, solution « objet », les enregistrements comprennent un champ nommé « `imprimer` » qui sait imprimer une repectivement une fiche de paie et un bon de commande.

En « PCF»

Rien n'empêche d'ajouter un champ « `imprimer` », à la fois aux salariés et aux commandes :

```
let travailleur = {
  ... ;
  imprimer = fun w ->
    Printf.printf "%s: %i\n" w.nom w.salaire ;
}

let commande = {
  ... ;
  imprimer =
    fun o ->
      Printf.printf "%s: %i X %i\n" o.quoi o.combien o.prix ;
}
```

Et pour imprimer que x soit un salarié ou une commande :

x`.imprimer` x

La même chose en pur PCF

```
Let tortue = { x=0 ; step=Fun t -> { t With x = t.x+1 } }
Let lapin = { x=0 ; step=Fun l -> { l With x = l.x+5 } }
```

Faire avancer un animal :

```
Let go = Fun a -> a.step a
```

Et on a (avec un toplevel approprié).

```
PCF> go lapin;;
 - = { x=5 ; step=<fun> }
PCF> go tortue;;
 - = { x=1 ; step=<fun> }
PCF> go (go lapin) ;;
 - = { x=10 ; step=<fun> }
```


Systématiser

Objet : enregistrement dont tous les champs sont des *méthodes*.

Méthode :

- Une fonction qui prend un objet comme premier argument.
- Toujours appelée sous la forme

$$o.m\ o\ a_1 \cdots a_n$$

Notre tortue n'est pas encore tout à fait un objet :

```
Let tortue = { x=0 ; step=Fun t -> { t With x = t.x+1 } }
```

Mais elle possède bien une méthode `step`.

Aider à pratiquer le style « objet »

Créer un objet :

Obj x `->` $\{\ell_1 = t_1\ ;\ \cdots \ell_n = t_n\ ;\}$ $\sim$ $\{\ell_1 =$ **Fun** x `->` $t_1\ ;\ \cdots \ell_n =$ **Fun** x `->` $t_n\ ;\}$

On peut voir **Obj** comme « sucre syntaxique » (*i.e.* existe slt dans la syntaxe concrète).

Ou comme nouvelle construction de la syntaxe abstraite, dont la sémantique est :

$$E \vdash \textbf{Obj}\ x \texttt{ -> } \{\ldots \ell_i = t_i;\ \ldots\} \hookrightarrow \{\ldots \ell_i = \langle x \bullet t_i \bullet E\rangle;\ \ldots\}$$

Invoquer une méthode :

$$t\#\ell \sim \textbf{Let}\ o = t\ \textbf{In}\ o.\ell\ o$$

Éventuellement, sémantique :

$$\frac{E \vdash t \hookrightarrow o \quad o = \{\ldots \ell = \langle x \bullet t_1 \bullet E_1 \rangle; \ldots\} \quad E_1 \oplus [x = o] \vdash t_1 \hookrightarrow v}{E \vdash t\#\ell \hookrightarrow v}$$

Redéfinition (Ajout) de méthode

$$\textbf{Obj}\ x\ \texttt{->}\ \{t_1\ \textbf{With}\ \ell\texttt{=}t_2\} \sim \{t_1\ \textbf{With}\ \ell\texttt{=}\textbf{Fun}\ x\ \texttt{->}\ t_2\}$$

Éventuellement, sémantique (assez évidente).

$$\frac{E \vdash t_1 \hookrightarrow \{\ldots \ell = ?; \ldots\}}{E \vdash \textbf{Obj}\ x\ \texttt{->}\ \{t_1\ \textbf{With}\ \ell\texttt{=}t_2\} \hookrightarrow \{\ldots \ell = \langle x \bullet t_2 \bullet E \rangle; \ldots\}}$$

$$\frac{E \vdash t_1 \hookrightarrow r \qquad r = \{\cdots\} \qquad \ell \notin \mathcal{L}(r)}{E \vdash \textbf{Obj}\ x\ \texttt{->}\ \{t_1\ \textbf{With}\ \ell\texttt{=}t_2\} \hookrightarrow \{\cdots \ell = \langle x \bullet t_2 \bullet E \rangle\}}$$

Obliger à l'objet

Supprimer les constructions des enregistrements de la syntaxe (mais conserver les enregistrements dans la sémantique).

Mais alors comment écrire la tortue « objet ». Facile ?

```
Let tortue = Obj self {
   x = 0 ;
   step = Obj s { self With x = self#x+1 } ;
}
```

Qui veut dire :

```
Let tortue = {
   x = Fun self -> 0 ;
   step = Fun self -> { self With x = Fun s -> self#x+1 } ;
}
```

On note que dans un language comme Java, `self` est implicite et s'appelle **this**.

Essai (de la tortue)

```
PCF> let t = tortue#step;;
 val t = { x=<fun> ; step=<fun> }
PCF> t#x;;
 - = 1
```

Qui s'explique, puisque `t` est

$$\{\texttt{x} = \langle \texttt{s} \bullet \texttt{self\#x+1} \bullet E \rangle ; \cdots\}$$

Où $E = [\texttt{self} = \{\texttt{x} = \langle \texttt{self} \bullet \texttt{0} \bullet E_0 \rangle ; \cdots\}; \ldots]$

Ou si on préfère : `t#x` est « **Fun** `s -> self#x+1` » pris dans un environnement où `self` est un objet dont la méthode `x` vaut **Fun** `_ -> 0`

Et, miracle, `tt` = `t#step` vaut bien 2.

Car `tt#x` est « **Fun** `s -> self#x+1` » pris dans un environnement où `self` vaut `t` (et donc `self#x` renvoie 1).

PCF purement objet

Est quand même un peu trop radical, il devient plus raisonable si on considère des constructions impératives :

```
Let tortue =
  Let x = Ref 0 In
  Obj self { x = !x ; step = x := !x+1 ; }
```

(Possible car la variable `x` et la méthode `x` habitent deux *espaces de noms* différents).

Mais en fait, sans champs mutable cela reste assez bizarre.

Nous en restons là, car PCF objet est quand même exemplaire de la programmation objet.

Liaison tardive

Comparer

```
Let a = 4 In
Let f = Fun y -> y+a In
Let a = 5 In
f 6
```

Résultat : `10` (liaison statique). Et

```
Let o = Obj self {
  a = 4 ;
  f = Fun y -> y + self#a ;
} In
Obj s { o With a = 5 }#f 6
```

Résultat : `11` (liason tardive de `self`).

Méthodes abstraites

En fait la méthode `a` n'a pas besoin d'exister quand on définit `f`.

```
Let o = Obj self {
  f = Fun y -> y + self#a ;
} In
Obj s { o With a = 5 }#f 6
```

Évidemment, dans un contexte typé, il faudra sans doute déclarer `a`.

Objets de Java

En Java les objets ne sont pas définis directement mais en instanciant des classes, par ex.

```
abstract class Animal {
  int x = 0 ;
  int getx() { return this.x ; }
  abstract void step() ;
  static void go(Animal a) { a.step() ; }
}
```

Et par exemple :

```
class Tortue extends Animal {
  void step () { this.x = this.x+1 ; }
}
class Lapin extends Animal {  void step () { x = x+5 ; } }
```

L'ajout/redéfinition de méthode opère donc plus sur les classes que sur les objets.

Java la suite

Un objet de la classe `Animal` est représenté en machine disons comme une paire :

- Un pointeur vers une table des champs (ici une case `x` qui contient un **int**).
- Un pointeur vers la table des méthodes (ici deux cases, la première contenant `getx` et la seconde `step`).

Il y a une table des champs par objet et une table des méthodes par classe.

`go` n'a pas besoin de savoir si `a` est une `Tortue` ou un `Lapin`, il lui suffit de savoir que `step` est en seconde position.

La methode `step` est une fonction qui prend un premier argument implicite (paramètre formel **this**), qui vaut `a` dans l'invocation `a.step()`.

Approcher l'héritage de Java

En java on peut écrire :

```
class TortueRapide extends Tortue {
  void step() { super.step() ; super.step() ; }
}
```

Et on obtient une tortue deux fois plus rapide.

En PCF objet c'est nettement plus tordu :

```
Let tortue_rapide =
  Let tortue_step = tortue.step In
  Obj this
    { tortue With step = tortue_step (tortue_step this) }
```

On peut donc conjecturer qu'en Java **super**`.step` appelle la méthode `step` de la classe parent sur... **this**.

On a vu...

- Une définition à peu près rigoureuse des termes, puis rigoureuse des propriétés sur les termes (pt. fixe).
- Trois sortes de sémantiques (petit pas, grand pas, dénotationnelle) rigoureusement définies.
- Une approche approfondie du typage.
- Diverses extensions du langage de base (un peu moins de rigueur, un peu plus de culture).
- Nous avons programmé (presque) tout cela, et c'est très important.

On a pas tout vu !

Nous avons laissé de côté bien des sujets :

- D'autre traits (types « algébriques » ?).
- Certaines techniques (analyse grammaticale, vraie compilation ?).
- La plupart des preuves de théorèmes (notablement correction du typage et approche complète de la sémantique dénotationelle).

⋮

- Où en est la recherche en langages de programmation?

La suite

- Cette année : compilation, analyse statique.
- L'année prochaine : MPRI ici, mais il y en a d'autres.

Et en attendant

- TP, interpréteur de PCF objet.
- La prochaine fois, examen...