

Le langage PCF

Luc.Maranget@inria.fr

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

A Quelques particularités

- ▶ Les fonctions.
- ▶ La récursion.

B Un peu de rigueur

- ▶ Syntaxe.
- ▶ Sémantique à petits pas.
- ▶ Sémantique à grands pas.

Un langage fonctionnel

- ▶ PCF est un *modèle* de langage de programmation.
 - ▷ Dans un modèle, on simplifie, au profit d'un traitement rigoureux et complet de la sémantique.
 - ▷ PCF dit ce que les langages fonctionnels (ML, Lisp, Haskell) ont en commun (leur noyau).
- ▶ PCF est d'abord un calcul des fonctions.
- ▶ Plus quelques constructions « pragmatiques » (genre: l'arithmétique et ses quatre opérations).

Les constructions pragmatiques

L'arithmétique: syntaxe

- ▶ Une infinité de constantes: 0, 1 etc.
- ▶ Quatre symboles binaires pour quatre opérations +, -, *, /.
- ▶ Une conditionnelle, symbole ternaire **Ifz** t_1 **Then** t_2 **Else** t_3 .

Nous avons défini un certain ensemble de termes.

L'arithmétique: sémantique

On définit une infinité d'axiomes qui donnent le sens des quatre opérations:

$$0+0 \longrightarrow 0 \quad 1+0 \longrightarrow 1 \quad 1+1 \longrightarrow 2 \quad \dots$$

Résumé en:

$$n_1 \text{ op } n_2 \longrightarrow n_1 \text{ op } n_2$$

Au passage: la soustraction est limitée aux entiers naturels.

$$x - y = 0 \text{ quand } x \leq y.$$

Cela revient à supposer que les entiers existent, ce qui est raisonnable

Pour la conditionnelle, on précise:

$$\text{Ifz } 0 \text{ Then } t_1 \text{ Else } t_2 \longrightarrow t_1$$

$$\text{Ifz } n \text{ Then } t_1 \text{ Else } t_2 \longrightarrow t_2 \quad (n \text{ constante entière différente de } 0)$$

Et si les entiers n'existent pas ?

Les définir selon l'arithmétique de Peano:

► Une constante 0 et un symbole unaire S (successeur) Deux se note alors S (S 0).

► Des axiomes en nombre fini du genre:

$$0 + t \longrightarrow t \quad S(t_1) + t_2 \longrightarrow S(t_1 + t_2)$$

$$\dots \text{ Ifz } 0 \text{ Then } t_1 \text{ Else } t_2 \longrightarrow t_1$$

$$\text{Ifz } S(t) \text{ Then } t_1 \text{ Else } t_2 \longrightarrow t_2$$

Mais bon, on va supposer que les entiers existent.

Une fonction ?

En maths : soit f définie de N dans N, qui à n associe n x n.

Parfois écrit f : N -> N, n ↦ n x n.

En PCF:

$$\text{Fun } n \text{ -> } n * n$$

La différence: la liaison de la variable n est explicite.

Ces variables sont muettes, on aurait pu écrire:

$$\text{Fun } x \text{ -> } x * x$$

La variable « muette » est déjà connue:

$$\int_0^1 x^2 dx \quad \int_0^1 u^2 du \quad \dots$$

Que faire avec une fonction ?

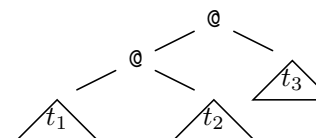
L'appliquer (à un argument) notation t1 t2.

$$(\text{Fun } x \text{ -> } x*x) \ 2$$

L'exécution du terme/programme ci dessus, doit bien renvoyer 4.

Note 1 La notation t1 t2 cache l'existence d'un symbole binaire "@": parfois explicité: @(t1, t2).

Note 2 L'application « associe à gauche », c'est à dire que t1 t2 t3 est à comprendre comme (t1 t2) t3. Soit:



Sémantique de l'application

Ou plutôt du passage d'argument ou β réduction

$$(\mathbf{Fun} \ x \ \rightarrow \ t_1) \ t_2 \longrightarrow [t_2 \backslash x]t_1$$

Où la notation $[t_2 \backslash x]t_1$ veut dire: « t_1 où x est remplacé par t_2 » (on y reviendra).

Par exemple:

$(\mathbf{Fun} \ x \ \rightarrow \ x*x) \ 2$

Devient

$2*2$

Qui, comme chacun sait, vaut 4.

Fonctions de première classe

► L'argument d'une fonction peut être une fonction.

$\mathbf{Fun} \ f \ \rightarrow \ f \ 0$

Appliquée à $\mathbf{Fun} \ x \ \rightarrow \ 1+x$ la fonction ci-dessus renvoie: 1.

► Le résultat d'une fonction peut être une fonction.

$\mathbf{Fun} \ n \ \rightarrow \ \mathbf{Fun} \ m \ \rightarrow \ n + m$

Appliquée à 1, la fonction ci-dessus renvoie: $\mathbf{Fun} \ m \ \rightarrow \ 1+m$, une fonction qui ajoute 1 à son argument.

Remarque : Avec les fonctions qui renvoient des fonctions, on a aussi les fonctions à plusieurs arguments (curryfication).

Exemple classique de fonctions de première classe

La composition.

$\mathbf{Fun} \ f \ \rightarrow \ \mathbf{Fun} \ g \ \rightarrow \ \mathbf{Fun} \ x \ \rightarrow \ f \ (g \ x)$

Ainsi:

$(\mathbf{Fun} \ f \ \rightarrow \ \mathbf{Fun} \ g \ \rightarrow \ \mathbf{Fun} \ x \ \rightarrow \ f \ (g \ x))$
 $(\mathbf{Fun} \ n \ \rightarrow \ n+1)$
 $(\mathbf{Fun} \ m \ \rightarrow \ 2*m)$

Doit valoir (deux β -réductions, sur f puis g)

$\mathbf{Fun} \ x \ \rightarrow \ (\mathbf{Fun} \ n \ \rightarrow \ n+1) \ ((\mathbf{Fun} \ m \ \rightarrow \ 2*m) \ x)$

Et donc (deux autres β -réductions, sur m puis n)

$\mathbf{Fun} \ x \ \rightarrow \ 2*x+1$

Ce qui correspond bien à la composition.

La définition locale

Il est plus clair d'écrire:

$\mathbf{Let} \ comp = \mathbf{Fun} \ f \ \rightarrow \ \mathbf{Fun} \ g \ \rightarrow \ \mathbf{Fun} \ x \ \rightarrow \ f \ (g \ x) \ \mathbf{In}$
 $\mathbf{Let} \ inc = \mathbf{Fun} \ n \ \rightarrow \ n+1 \ \mathbf{In}$
 $\mathbf{Let} \ twice = \mathbf{Fun} \ m \ \rightarrow \ 2*m \ \mathbf{In}$
 $comp \ inc \ twice$

En gros, l'idée c'est:

$\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \sim (\mathbf{Fun} \ x \ \rightarrow \ t_2) \ t_1$

Soit comme règle sémantique :

$\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \longrightarrow [t_1 \backslash x]t_2$

La récursion

Comment définir la fonction puissance ($\mathbb{N}^2 \rightarrow \mathbb{N}$), qui à x et n entiers associe:

$$\underbrace{x \times x \times \cdots \times x}_{n \text{ fois}}$$

► En mathématiques (si on évite le \cdots vite ambigu): récurrence.

► Dans les langages de programmation:

▷ Boucle (Java):

```
int r = 1 ;
for (int k = 1 ; k <= n ; k++) r := r * x ;
```

▷ Définition récursive (Caml):

```
let rec pow x n =
  if n = 0 then 1 else x * pow x (n-1)
```

Définition « par point fixe »

En PCF on définit la fonction `pow` comme égale à

```
Fun x -> Fun n -> Ifz n Then 1 Else x * pow x (n-1)
```

Mais `pow` ne *peut pas* être utilisée dans sa propre définition.

On écrit plutôt $p = \Phi p$ avec:

Let $\Phi =$

```
Fun pow ->
  Fun x -> Fun n -> Ifz n Then 1 Else x * pow x (n-1)
```

À comprendre comme: la fonction p est point fixe de Φ .

On laisse de côté pour le moment la question de l'existence/unicité de ce point fixe et on note :

```
Fix pow ->
  Fun x -> Fun n -> Ifz n Then 1 Else x * pow x (n-1)
```

Réduction du Fix

Un moyen de contourner l'existence/unicité : définir l'effet du **Fix**.

$$\mathbf{Fix} \ x \rightarrow t \longrightarrow [\mathbf{Fix} \ x \rightarrow t \backslash x]t$$

L'idée est en fait très simple:

```
(Fix fact -> Fun n -> Ifz n Then 1 Else n*fact (n-1)) 3
```

Règle du **Fix**, remplacer `fact` par `Fix fact -> ...`

```
(Fun n -> Ifz n Then 1 Else n * (Fix fact -> ...) (n-1)) 3
```

β -règle (remplacer `n` par 3).

```
Ifz 3 Then 1 Else 3 * (Fix fact -> ...) (3-1)
```

Comme 3 n'est pas nul, et que 3-1 vaut 2:

```
3 * (Fix fact -> ...) 2
```

Soit en gros `3 * fact 2` c'est gagné. Au final `3 * 2 * 1 * 1`.

Un peu de rigueur: syntaxe

Les termes de PCF sont définis, par

- Une infinité de constantes 0, 1, etc.
- Quatre symboles d'opération +, -, *, / (binaires, notés de façon infixé dans les termes $t_1 + t_2$).
- Un symbole ternaire (**Ifz**, notation des termes `Ifz t_1 Then t_2 Else t_3`).
- Un ensemble de variables.
- Le symbole binaire de fonction, termes notés `Fun $x \rightarrow t$` (le premier argument est une variable, liée dans t).
- Le symbole binaire du point fixe, termes notés `Fix $x \rightarrow t$` .
- La définition, ternaire, termes notés `Let $x = t_1$ In t_2` .

Syntaxe abstraite de PCF

```

type var = string

type op = Add | Sub | Mul | Div

type t =
  | Num of int
  | Var of var
  | Op of op * t * t
  | Ifz of t * t * t
  | Let of var * t * t
  | App of t * t
  | Fun of var * t
  | Fix of var * t

```

Un peu de rigueur: sémantique

Le but est de définir une relation $t \hookrightarrow v$ entre les programmes (termes de PCF) et leur valeurs.

Base de départ, six axiomes de réduction élémentaires:

$$\begin{aligned}
 & \text{(\b{F}un } x \rightarrow t_1) t_2 \longrightarrow [t_2 \backslash x]t_1 & \text{\b{L}et } x = t_1 \text{ \b{I}n } t_2 \longrightarrow [t_1 \backslash x]t_2 \\
 & \text{\b{F}ix } x \rightarrow t \longrightarrow [(\text{\b{F}ix } x \rightarrow t) \backslash x]t & \text{\b{I}fz } 0 \text{ \b{T}hen } t_1 \text{ \b{E}lse } t_2 \longrightarrow t_1 \\
 & \text{\b{I}fz } n \text{ \b{T}hen } t_1 \text{ \b{E}lse } t_2 \longrightarrow t_2 \ (n \neq 0) & n_1 \text{ op } n_2 \longrightarrow n_1 \text{ op } n_2
 \end{aligned}$$

Sémantique « à petits pas »

- Les termes réductibles ou *radicaux*. Ex. $1 + 1$, $(\text{\b{F}un } x \rightarrow x + x) y$.
Contracter un radical (règle $t_1 \longrightarrow t_2$) c'est remplacer t_1 par t_2 . Ex. $2, y + y$.

- Plus « Passage au contexte »: contracter les radicaux dans les termes.

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \qquad \frac{t_2 \longrightarrow t'_2}{t_1 t_2 \longrightarrow t_1 t'_2}$$

$$\frac{t \longrightarrow t'}{\text{\b{F}un } x \rightarrow t \longrightarrow \text{\b{F}un } x \rightarrow t'} \qquad \dots$$

Sémantique à petits pas

- Rappel ? La fermeture réflexive-transitive \mathcal{R}^* d'une relation \mathcal{R} .

$$t \mathcal{R} t \qquad \frac{t_1 \mathcal{R} t_2}{t_1 \mathcal{R}^* t_2} \qquad \frac{t_1 \mathcal{R}^* t_2 \quad t_2 \mathcal{R}^* t_3}{t_1 \mathcal{R}^* t_3}$$

- Les valeurs v sont définies comme les termes irréductibles (ou formes normales)

$$v \in \mathcal{NF} \Leftrightarrow \nexists w, v \longrightarrow w$$

- Définition de la sémantique $t \hookrightarrow v$ ssi:

$$t \longrightarrow^* v \text{ et } v \in \mathcal{NF}$$

Autrement dit: réduire les radicaux de t , jusqu'à plus soif.

Exemples

Valeur de :

- ▶ **Fix** $x \rightarrow x$? Pas de valeur car $t \rightarrow t$ et pas le choix.
- ▶ **(Fun** $y \rightarrow 3)$ 2 ? 3 en une β -réduction.
- ▶ **(Fun** $y \rightarrow 3)$ **(Fix** $x \rightarrow x)$? 3 en une β -réduction, si on est malin,
- ▶ **Fix** $n \rightarrow 1+n$? Pas de valeur, réduction de t sur $1+t$, $1+(1+t)$, $1+(1+(1+t))$, etc.

Exemple plus sérieux

Let $pow2 = \text{Fix } p \rightarrow \text{Fun } n \rightarrow \text{Ifz } n \text{ Then } 1 \text{ Else } 2 * p (n-1) \text{ In } pow2 \ 2$

Réduire liaison:

(Fix $p \rightarrow \text{Fun } n \rightarrow \text{Ifz } n \text{ Then } 1 \text{ Else } 2 * p (n-1)) \ 2$

Notons $P = \text{Fix } p \rightarrow \dots$. Réduire point fixe:

(Fun $n \rightarrow \text{Ifz } n \text{ Then } 1 \text{ Else } 2 * P (n-1)) \ 2$

β -réduction.

Ifz $2 \text{ Then } 1 \text{ Else } 2 * P (2-1)$

$2 * P (2-1)$

$2 * (\text{Fun } n \rightarrow \text{Ifz } n \text{ Then } 1 \text{ Else } 2 * P (n-1)) (2-1)$

$2 * (\text{Ifz } 2-1 \text{ Then } 1 \text{ Else } 2 * P ((2-1)-1))$

$2 * (\text{Ifz } 1 \text{ Then } 1 \text{ Else } 2 * P ((2-1)-1))$

$2 * (2 * P ((2-1)-1))$

$2 * (2 * ((\text{Fun } n \rightarrow \text{Ifz } n \text{ Then } 1 \text{ Else } 2 * P (n-1)) ((2-1)-1)))$

$2 * (2 * (\text{Ifz } ((2-1)-1) \text{ Then } 1 \text{ Else } 2 * P (((2-1)-1)-1)))$

$2 * (2 * (\text{Ifz } (1-1) \text{ Then } 1 \text{ Else } 2 * P (((2-1)-1)-1)))$

$2 * (2 * (\text{Ifz } 0 \text{ Then } 1 \text{ Else } 2 * P (((2-1)-1)-1)))$

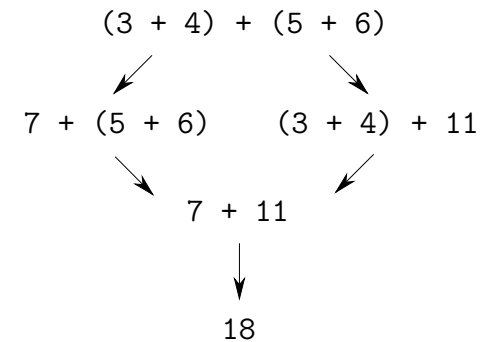
$2 * (2 * 1)$

Deux multiplications, ouf 4.

Le déterminisme

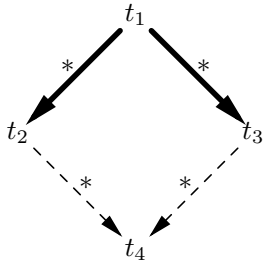
Dans un terme, il peut y avoir plusieurs radicaux. La réduction opère donc un choix.

Ce choix a-t-il un impact ? Les valeurs (formes normales) sont elles uniques ?



Confluence

L'unicité des formes normales découle du résultat plus fort de *confluence* (Théorème, admis).



Qui dit en gros,

- ▶ Deux calculs divergents (t_2 ou t_3).
- ▶ Ne le sont jamais définitivement (il existe t_4).

Retour sur la substitution

On a dit : $[t_1 \backslash x]t_2$ = remplacer x par t_1 dans t_2 .

La substitution donne le sens des variables.

Elle permet aussi de donner un sens aux expressions qui possèdent des variables libres (ou non-closes).

Par ex. $x+1$. possède un sens clair dès que l'on donne une valeur à x .

Rappel ? définition des variables libres

$$\mathcal{F}(n) = \emptyset \quad \mathcal{F}(x) = \{x\} \quad \mathcal{F}(t_1 \text{ op } t_2) = \mathcal{F}(t_1) \cup \mathcal{F}(t_2)$$

$$\mathcal{F}(t_1 t_2) = \mathcal{F}(t_1) \cup \mathcal{F}(t_2) \quad \mathcal{F}(\mathbf{Fun } x \rightarrow t) = \mathcal{F}(t) \setminus \{x\}$$

$$\mathcal{F}(\mathbf{Fix } x \rightarrow t) = \mathcal{F}(t) \setminus \{x\}$$

$$\mathcal{F}(\mathbf{Let } x = t_1 \text{ In } t_2) = \mathcal{F}(t_1) \cup (\mathcal{F}(t_2) \setminus \{x\})$$

Remplacer x par t_1

Est-ce si simple ? En l'absence de lieurs, oui.

$$[t \backslash x]x = t \quad [t \backslash x]y = y \quad [t \backslash x](t_1 \text{ op } t_2) = ([t \backslash x]t_1) \text{ op } ([t \backslash x]t_2)$$

$$[t \backslash x](t_1 t_2) = ([t \backslash x]t_1) ([t \backslash x]t_2) \quad [t \backslash x](\mathbf{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3) =$$

$$\mathbf{Ifz } [t \backslash x]t_1 \text{ Then } [t \backslash x]t_2 \text{ Else } [t \backslash x]t_3$$

Soit par exemple,

$$[t \backslash x](x + y) = t + y$$

En présence de lieurs

La substitution donne la valeur des variables *libres*. On propose donc:

$$[t \backslash x](\mathbf{Fun } x \rightarrow t_1) = \mathbf{Fun } x \rightarrow t_1$$

$$[t \backslash x](\mathbf{Fun } y \rightarrow t_1) = \mathbf{Fun } y \rightarrow [t \backslash x]t_1 \quad \text{etc.}$$

Let $y = t$ **In** **Fun** $x \rightarrow x + y$

En substituant...

- ▶ Si t est 4 on a **Fun** $x \rightarrow x + 4$.
- ▶ Si t est z on a **Fun** $x \rightarrow x + z$.
- ▶ Si t est x a-t-on **Fun** $x \rightarrow x + x$?

Il ne vaut mieux pas ! Considérer la confluence de:

Let $x = 1$ **in** **Let** $y = x$ **in** **Fun** $x \rightarrow x + y$

Alpha-équivalence

Dans l'exemple $[x \setminus y](\mathbf{Fun} \ x \ -> \ x + y)$, nous sommes victimes de la *capture* de la variable libre x , qui devient liée en traversant la liaison $\mathbf{Fun} \ x \ -> \dots$

Or

- ▶ Si l'argument de la fonction est nommé z on peut substituer :

$$[x \setminus y](\mathbf{Fun} \ z \ -> \ z + y) = \mathbf{Fun} \ z \ -> \ z + x.$$

- ▶ Les fonctions $\mathbf{Fun} \ x \ -> \ x + y$ et $\mathbf{Fun} \ z \ -> \ z + y$ sont les mêmes (changement de la variable muette).

L'alpha-équivalence est l'égalité modulo le changement des variables muettes.

On considère les termes « modulo alpha-équivalence ».

Une définition de la substitution

Choisir un bon représentant des classes d'alpha-équivalence.

$$[t \setminus x](\mathbf{Fun} \ y \ -> \ t_2) = \mathbf{Fun} \ z \ -> \ [t \setminus x]([z \setminus y]t_2)$$

$$[t \setminus x](\mathbf{Fix} \ y \ -> \ t_2) = \mathbf{Fix} \ z \ -> \ [t \setminus x]([z \setminus y]t_2)$$

$$[t \setminus x](\mathbf{Let} \ y = t_1 \ \mathbf{In} \ t_2) = \mathbf{Let} \ z = [t \setminus x]t_1 \ \mathbf{In} \ [t \setminus x]([z \setminus y]t_2)$$

Où z est tel que $z \notin \mathcal{F}(t)$ (éviter capture vue) et $z \notin \mathcal{F}(t_2)$ (éviter autre capture).

Plus les définitions pour les symboles sans lieux (page 27).

Sémantique et exécution de programme

Dans un terme, il peut y avoir un choix de radicaux.

- On veut modéliser le calcul d'un ordinateur déterministe.
- Or, certains choix de radicaux de réduction calculent la forme normale, d'autres non.

```
Let loop = Fix f -> Fun x -> f x In
Let k = Fun x -> Fun y -> y In
k (loop 0) 1
```

La sémantique doit répondre à : vaut 1 ou ne termine pas ?

- En outre, peu d'intérêt à distinguer:

```
Fun x -> x + 11      Fun x -> x + (7+4)
```

- En outre, toutes les variables d'un programme ont une définition.

Programmes et réduction faible

1. Dans un programme, il n'y a pas d'inconnu : on ne réduit que des termes clos.
2. Les fonctions sont des fonction et point.

On réduit des termes clos et on *utilise jamais* la règle :

$$\frac{t \longrightarrow t'}{\mathbf{Fun} \ x \ -> \ t \longrightarrow \mathbf{Fun} \ x \ -> \ t'}$$

Tout cela revient à définir les valeurs v ainsi :

- ▶ Une constante entière n .
- ▶ Ou bien une fonction $\mathbf{Fun} \ x \ -> \ t$ (t terme quelconque, avec $\mathcal{F}(t) \subseteq \{x\}$).

Une stratégie

C'est une sous-relation (\Rightarrow, \subseteq) déterministe de la relation de réduction (\rightarrow).

Définissons « l'appel par nom », dont la principale caractéristique est de ne pas réduire les arguments des fonctions.

Les axiomes ne changent pas (voir slide 18).

$$\frac{t_1 \rightarrow_n t'_1}{t_1 t_2 \rightarrow_n t'_1 t_2}$$

$$\frac{t_1 \rightarrow_n t'_1}{t_1 \text{ op } t_2 \rightarrow_n t'_1 \text{ op } t_2} \quad \frac{t_2 \rightarrow_n t'_2}{v_1 \text{ op } t_2 \rightarrow_n v_1 \text{ op } t'_2}$$

$$\frac{t_1 \rightarrow_n t'_1}{\text{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3 \rightarrow_n \text{Ifz } t'_1 \text{ Then } t_2 \text{ Else } t_3}$$

Autre présentation

Le radical réduit est le plus extérieur (et le plus à gauche).

$$\mathcal{R}_n(t) = t, \text{ si } t \text{ radical} \quad \mathcal{R}_n(t_1 t_2) = \mathcal{R}_n(t_1)$$

$$\mathcal{R}_n(\text{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3) = \mathcal{R}_n(t_1)$$

$$\mathcal{R}_n(v \text{ op } t) = \mathcal{R}_n(t) \quad \mathcal{R}_n(t_1 \text{ op } t_2) = \mathcal{R}_n(t_1)$$

- Une étape: contracter $\mathcal{R}_n(t)$.
- Itérer (\rightarrow_n^*) jusqu'à trouver une valeur.

Présentation directe de l'appel par nom

En sémantique dite à grands pas, ou naturelle.

$$\frac{t_1 \hookrightarrow_n \text{Fun } x \rightarrow t_3 \quad [t_2 \backslash x]t_3 \hookrightarrow_n v}{t_1 t_2 \hookrightarrow_n v}$$

$$\text{Fun } x \rightarrow t \hookrightarrow_n \text{Fun } x \rightarrow t$$

$$\frac{[t_1 \backslash x]t_2 \hookrightarrow_n v}{\text{Let } x = t_1 \text{ In } t_2 \hookrightarrow_n v} \quad \frac{[(\text{Fix } x \rightarrow t) \backslash x]t \hookrightarrow_n v}{\text{Fix } x \rightarrow t \hookrightarrow_n v}$$

$$\frac{t_1 \hookrightarrow_n 0 \quad t_2 \hookrightarrow_n v}{\text{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3 \hookrightarrow_n v} \quad \frac{t_1 \hookrightarrow_n n(n \neq 0) \quad t_3 \hookrightarrow_n v}{\text{Ifz } t_1 \text{ Then } t_2 \text{ Else } t_3 \hookrightarrow_n v}$$

$$\frac{t_1 \hookrightarrow_n n_1 \quad t_2 \hookrightarrow_n n_2}{t_1 \text{ op } t_2 \hookrightarrow_n n_1 \text{ op } n_2} \quad n \hookrightarrow_n n$$

Quelle présentation choisir ?

On préfère souvent la sémantique naturelle plus concise, en particulier pour programmer un évaluateur (TP).

Mais, ne dit rien des termes qui ne terminent pas, ne dit rien des termes « bloqués » (ex. 1 appliqué à 2).

Les résultats théoriques sont donc plus précis en sémantique par petit pas.

Équivalences (admisses)

- Si $t \hookrightarrow_n v$, alors $t \rightarrow_n^* v$.
- Si $t \rightarrow_n^* v$ (v valeur), alors $t \hookrightarrow_n v$.

Une autre stratégie : l'appel par valeur

Par rapport à l'appel par nom, deux règles changent :

$$\frac{t_1 \hookrightarrow_n \mathbf{Fun} \ x \rightarrow t_3 \quad [t_2 \setminus x]t_3 \hookrightarrow_n v}{t_1 \ t_2 \hookrightarrow_n v} \quad \frac{[t_1 \setminus x]t_2 \hookrightarrow_n v}{\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \hookrightarrow_n v}$$

Deviennent

$$\frac{t_1 \hookrightarrow_v \mathbf{Fun} \ x \rightarrow t_3 \quad t_2 \hookrightarrow_v v_2 \quad [v_2 \setminus x]t_3 \hookrightarrow_v v}{t_1 \ t_2 \hookrightarrow_v v}$$

$$\frac{t_1 \hookrightarrow_v v_1 \quad [v_1 \setminus x]t_2 \hookrightarrow_v v}{\mathbf{Let} \ x = t_1 \ \mathbf{In} \ t_2 \hookrightarrow_v v}$$

Note : Les arguments sont réduits avant la substitution.

Appel par valeur

Parfois caractérisé comme « contracter le radical le plus profond » (et le plus à gauche). Ce n'est pas tout à fait exact.

► Le **Ifz** ne change pas. Autrement dit on ne *remplace pas*

$$\frac{t_1 \hookrightarrow_n 0 \quad t_2 \hookrightarrow_n v_2}{\mathbf{Ifz} \ t_1 \ \mathbf{Then} \ t_2 \ \mathbf{Else} \ t_3 \hookrightarrow_n v_2}$$

par

$$\frac{t_1 \hookrightarrow_v 0 \quad t_2 \hookrightarrow_v v_2 \quad t_3 \hookrightarrow_v v_3}{\mathbf{Ifz} \ t_1 \ \mathbf{Then} \ t_2 \ \mathbf{Else} \ t_3 \hookrightarrow_v v_2}$$

► Autrement dit, il reste un peu d'appel par nom.

Idem pour **Fix** $x \rightarrow t$. Le terme t n'est pas réduit, même si il contient un radical « plus profond ».

Manque d'expressivité de l'appel par valeur

On définit (**Let**) la fonction **fi**.

Fun $x \rightarrow \mathbf{Fun} \ y \rightarrow \mathbf{Fun} \ z \rightarrow \mathbf{Ifz} \ x \ \mathbf{Then} \ z \ \mathbf{Else} \ y$

► En appel par nom, on a l'équivalence :

$$\mathbf{fi} \ t_1 \ t_2 \ t_3 \hookrightarrow_n v \Leftrightarrow \mathbf{Ifz} \ t_1 \ \mathbf{Then} \ t_3 \ \mathbf{Else} \ t_2 \hookrightarrow_n v$$

► Mais pas en appel par valeur. Considérer :

Ifz 0 **Then** 1 **Else** (**Fix** $x \rightarrow x$) 1

Et

fi 0 (**Fix** $x \rightarrow x$) 1

Noter :

Fix $x \rightarrow x \rightarrow \mathbf{Fix} \ x \rightarrow x \rightarrow \mathbf{Fix} \ x \rightarrow x \rightarrow \dots$

Comparaison des stratégies

Appel par nom : Algol 60, Haskell. Appel par valeur : tout les autres langages (ML).

On démontre :

► Que l'appel par nom est *sûr*.

$$(t \longrightarrow^* v) \Rightarrow (t \hookrightarrow_n v)$$

Exemple simple :

(**Fun** $x \rightarrow \mathbf{Fun} \ y \rightarrow y$) (**Fix** $x \rightarrow x$) 1

► Que l'appel par nom autorise les transformations de programme arbitraires.

$$t \longrightarrow^* t' \Rightarrow (t \hookrightarrow_n v \Leftrightarrow t' \hookrightarrow_n v')$$

Avec $v = v'$ pour les entiers, $v \longleftrightarrow^* v'$ en gal.

On constate que l'appel par valeur est plus facile à comprendre.

▶ TP, divers évaluateurs.

▶ La prochaine fois, interprétation.