Cours 1
Langages fonctionnels

Jean-Jacques.Levy@inria.fr

http://jeanjacqueslevy.net

secrétariat de l'enseignement:

Catherine Bensoussan

cb@lix.polytechnique.fr

Aile 00, LIX

tel: 34 67

http://w3.edu.polytechnique.fr/informatique/

Plan du cours

- quatre paradigmes de langages de programmation
  - fonctionnel (ML, Haskell)
  - impératif (Pascal, C, Java, et un peu ML)
  - logique (Prolog)
  - orienté-objet (Java, Ocaml)
- Principes généraux des langages de programmation
- Sémantique (opérationnelle, plutôt que dénotationnelle)
- Vérification de programmes

(en 8 leçons!)

On supposera connue la notion de syntaxe abstraite

Références

- The Lambda Calculus, Its Syntax and Semantics, H. Barendregt, North-Holland, 1981.
- Theories of Programming Languages, J. Reynolds, Cambridge University Press, 1998.
- Foundations for Programming Languages, J. Mitchell, MIT Press, 1996.
- Essentials of Programming Languages, D. Friedman, M. Wand, C.Haynes, McGraw-Hill, 1992
- Semantics of Programming Languages, C. Gunter, MIT Press, 1992.

Conférences

- ACM Conference on Principles of Programming Languages (POPL)
- IEEE Conference on Logic in Computer Science (LICS)

Plan

1. Lambda-calcul non typé
2. Variables libres – liées
3. Substitution de variables libres
4. Réductions
5. Confluence
6. Stratégies de réduction
7. Evaluateurs formels

Programmation fonctionnelle

- Simple, concis et esthétique
- Notation mathématique
- Fonctions $=$ citoyens de 1ère classe
- Fonctions $+$ structures de données $=$ langage puissant
- Transparence référentielle (indépendance du contexte)
- Parallélisation

La lambda notation — Church [30]

Supposons que l'on dispose d'une fonction `integrale` telle que `integrale`(a, b, f) vaut $\int_a^b f(x)dx$

Soit f la fonction $x \mapsto x^2 + 1$. On veut calculer $\int_0^1 f(x)dx$.

Avec la lambda notation, on peut écrire

$f \equiv \lambda x.\ x \times x + 1$

et calculer `integrale`$(0, 1, f)$

ou plus simplement écrire

`integrale`$(0, 1, \lambda x.\ x \times x + 1)$

La lambda notation – bis

Considérons $g(y) = (x + y) \times (x - y)$ et $f(x) = \int_0^x g(y)dy$

Essayons de calculer $\int_0^1 f(x)dx = \int_0^1 \int_0^x (x + y) \times (x - y)\, dxdy$

On écrira $\texttt{integrale}(0, 1, \lambda x.\ \texttt{integrale}(0, x, \lambda y.\ (x + y) \times (x - y)))$

Remarque

On a utilisé

- notation anonyme pour les fonctions
- fonction argument d'une autre fonction
- fonction résultat d'une autre fonction

La lambda notation et les langages de programmation

C'est le coeur de la notation pour des langages comme Lisp, Scheme, ML ou Haskell.

En Lisp/Scheme:

```
(defun integrale (a b f) ... )

(integrale 0 1 (lambda (x) (integrale 0 x (lambda (y) (+ x y)))))
```

En ML:

```
let integrale a b f =  ... ;;

(integrale 0 1 (function x -> (integrale 0 x (function y -> x+y)))) ;;
```

Beaucoup de la problématique de la λ-notation se retrouve dans tous les langages de programmation.

Variables liées – Variables libres

Le nom des variables liées n'a pas d'importance.

$\lambda x.\, x \times x + 1$ est la même fonction que $\lambda x'.\, x' \times x' + 1$

De même pour $\lambda y.\, (x + y) \times (x - y))$ et $\lambda y'.\, (x + y') \times (x - y'))$

Mais pas identique à $\lambda x.\, (x + x) \times (x - x))$!!

On dit que x (resp. y) est **libre** (resp. **lié**) dans $\lambda y.\, (x + y) \times (x - y))$

On peut donc renommer une variable liée à condition de ne pas capturer de variable libre [une variable liée est souvent appelée variable muette en math.]

Renommage des paramètres d'une fonction

Les paramètres dans les fonctions de Pascal, Java, ML, etc. sont aussi des variables liées. On peut leur donner un nom arbitraire

```
let g y = (x + y) * (x - y) ;;
```

est identique à

```
let g' y' = (x + y') * (x - y') ;;
```

mais n'est pas équivalent à

```
let h x = (x + x) * (x - x) ;;
```

Idem en Java pour le nom du paramètre y dans

```
static int g (int y) { return (x + y) * (x - y); }
```

PCF – petit langage fonctionnel [Plotkin 74]

Termes

M, N, P	$::=$	x	variable
	$\mid$	$\lambda x.M$	abstraction
	$\mid$	$M\,N$	application
	$\mid$	$\underline{n}$	constante entière
	$\mid$	$M \otimes N$	$\otimes \in \{+, -, \times, \div\}$
	$\mid$	$\texttt{ifz}\, M\ \texttt{then}\ N\ \texttt{then}\ N'$	conditionnelle
	$\mid$	$\mu x.M$	définition récursive

Exemples

$\lambda x.x \times x + 1$
$\lambda x.\lambda y.(x + y) \times (x - y)$
$\lambda f.\lambda x.f(fx)$
$(\lambda f.\lambda x.f(fx))(\lambda x.x + 1)$
$(\mu f.\lambda x.\, \texttt{ifz}\, x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x - 1))\ 5$

Syntaxe abstraite de PCF

```
type nom = string;;

type term =
  Lambda of nom * term
| Application of term * term
| Variable of nom
| Entier of int
| Ifz of term * term * term
| Plus of term * term
| Times of term * term
| Div of term * term
| Minus of term * term
| Mu of nom * term ;;
```

Cf. `http://logical.inria.fr/~dowek/Cours/lp1.html`

Convention

On a tendance à supprimer les parenthèses à gauche des applications. Donc $M_1 M_2 \cdots M_n$ désignera $(\cdots(M_1 M_2)\cdots M_n)$.

Variables libres

$$
\begin{array}{ll}
FV(x) = \{x\} & FV(\lambda x.M) = FV(\mu x.M) = FV(M) - \{x\} \\
FV(\underline{n}) = \emptyset & FV(M\,N) = FV(M \otimes N) = FV(M) \cup FV(N) \\
\multicolumn{2}{l}{FV(\texttt{ifz}\, M \,\texttt{then}\, N \,\texttt{else}\, N') = FV(M) \cup FV(N) \cup FV(N')}
\end{array}
$$

Variables liées

$$
\begin{array}{ll}
BV(x) = \emptyset & BV(\lambda x.M) = BV(\mu x.M) = BV(M) \cup \{x\} \\
BV(\underline{n}) = \emptyset & BV(M\,N) = BV(M \otimes N) = BV(M) \cup BV(N) \\
\multicolumn{2}{l}{BV(\texttt{ifz}\, M \,\texttt{then}\, N \,\texttt{else}\, N') = BV(M) \cup BV(N) \cup BV(N')}
\end{array}
$$

Substitution

$$x[x\backslash P] = P$$
$$y[x\backslash P] = y$$
$$(\lambda x.M)[x\backslash P] = \lambda x.M$$
$$(\lambda y.M)[x\backslash P] = \lambda y'.M[y\backslash y'][x\backslash P] \qquad (y' \neq x \textbf{ et } y' \notin FV(P) \cup FV(M))$$
$$(M\,M')[x\backslash P] = M[x\backslash P]\ M'[x\backslash P]$$

$$\underline{n}[x\backslash P] = \underline{n}$$
$$(M \otimes M')[x\backslash P] = M[x\backslash P] \otimes M'[x\backslash P]$$
$$(\texttt{ifz}\,M\ \texttt{else}\ N\ \texttt{else}\ N')[x\backslash P] = \texttt{ifz}\,M[x\backslash P]\ \texttt{then}\ N[x\backslash P]\ \texttt{else}\ N'[x\backslash P]$$

$$(\mu x.M)[x\backslash P] = \mu x.M$$
$$(\mu y.M)[x\backslash P] = \mu y'.M[y\backslash y'][x\backslash P] \qquad (y' \neq x \textbf{ et } y' \notin FV(P) \cup FV(M))$$

Exemples

$(x+y)[y\backslash x] = x + x$
$(\lambda x.x+y)[y\backslash x] = \lambda x'.x' + x$
$(\texttt{ifz}\,y\ \texttt{then}\ x+y\ \texttt{else}\ z)[y\backslash x] = \texttt{ifz}\,x\ \texttt{then}\ x+x\ \texttt{else}\ z$
$(\mu x.x+y)[y\backslash x] = \mu x'.x' + x$

Règles de réductions

α	$\lambda x.M \equiv \lambda x'.M[x\backslash x']$	$(x' \notin FV(M))$
β	$(\lambda x.M)N \rightarrow M[x\backslash N]$	
op	$\underline{m} \otimes \underline{n} \rightarrow \underline{m \otimes n}$	$(m \otimes n \geq 0)$
cond1	$\texttt{ifz}\,\underline{0}\,\texttt{then}\ M\ \texttt{else}\ N \rightarrow M$	
cond2	$\texttt{ifz}\,\underline{n+1}\,\texttt{then}\ M\ \texttt{else}\ N \rightarrow N$	
μ	$\mu x.M \rightarrow M[x\backslash \mu x.M]$	

Remarques

- **la règle α est une relation d'équivalence, et non une réduction. L'égalité des termes est souvent considérée à α-équivalence près.**
- **la règle β représente l'application d'un argument à une fonction**

Exemples de réductions

$(\lambda x.\lambda y.x+y)2\ 3 \to (\lambda y.2+y)3 \to 2+3 \to 5$

$(\lambda f.\lambda x.f(fx))(\lambda x.x+1)3 \to (\lambda x.(\lambda x.x+1)((\lambda x.x+1)x))3 \to$
$(\lambda x.x+1)((\lambda x.x+1)3) \to (\lambda x.x+1)(3+1) \to (\lambda x.x+1)4 \to 4+1 \to 5$

$(\mu f.\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1))3 \to$

$(\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times (\mu f.\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1))(x-1))3 \to$

$\texttt{ifz}\,3\ \texttt{then}\ 1\ \texttt{else}\ 3 \times (\mu f.\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1))(3-1) \to$

$3 \times (\mu f.\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1))(3-1) \to$

$3 \times (\mu f.\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1))2 \to \cdots$

$3 \times 2 \times 1 \times \texttt{ifz}\,0\ \texttt{then}\ 1\ \texttt{else}\ 0 \times (\mu f.\lambda x.\,\texttt{ifz}\,x\ \texttt{then}\ 1\ \texttt{else}\ x \times f(x-1))(0-1) \to$

$3 \times 2 \times 1 \times 1 \to \cdots 6$

Stratégies de réduction

Plusieurs sous-termes réductibles (redex = ***reductible expressions*****) peuvent se trouver dans un même terme.**

Théorème [Church-Rosser] Si $M \to^* N$ et $M \to^* N'$, il existe toujours un P tel que $N \to^* P$ et $N' \to^* P$.

On dit aussi que le système de réduction est confluent.

La démonstration est compliquée.

Ce théorème assure l'unicité du résultat (forme normale) d'un calcul.

Appel par nom

L'ordre d'évaluation peut influer sur la terminaison. Par exemple

$(\lambda x.1)(\mu x.x) \to (\lambda x.1)(\mu x.x) \to \cdots$

mais pourtant

$(\lambda x.1)(\mu x.x) \to 1$

Théorème [Curry] Si M a une forme normale, la réduction normale (qui réduit toujours le redex le plus externe et le plus à gauche) atteint cette forme normale.

La démonstration est aussi compliquée.

Réduction normale $=$ appel par nom dans langages de programmation

Appel par valeur

Valeurs

$$
\begin{array}{rcll}
V, V' & ::= & x & \text{variable} \\
 & | & \lambda x.M & \text{abstraction} \\
 & | & \underline{n} & \text{constante entière}
\end{array}
$$

La stratégie de réduction par appel par valeur consiste à:

- pour une application: à trouver la valeur de l'argument de gauche d'une application, puis celle de l'argument de droite, puis à appliquer la règle β
- pour un opérateur arithmétique: à trouver la valeur des arguments avant de calculer la valeur finale
- pour une conditionnelle: à trouver la valeur du prédicat puis à appliquer la bonne règle Cond1 ou Cond2.

Sémantique opérationnelle *bigstep* de l'appel par valeur

On peut définir rigoureusement l'appel par valeur pour PCF avec des règles d'inférence.

$$\vdash \lambda x.M = \lambda x.M \qquad \frac{\vdash M[x\backslash \mu x.M] = V}{\vdash \mu x.M = V}$$

$$\frac{\vdash M = \lambda x.P \qquad \vdash N = V' \qquad \vdash P[x\backslash V'] = V}{\vdash M\,N = V}$$

$$\vdash \underline{n} = \underline{n} \qquad \frac{\vdash M = \underline{m} \qquad \vdash N = \underline{n} \qquad m \otimes n \geq 0}{\vdash M \otimes N = \underline{m \otimes n}}$$

$$\frac{\vdash M = \underline{0} \qquad \vdash M = V}{\vdash \texttt{ifz}\, M\, \texttt{then}\, N\, \texttt{then}\, N' = V} \qquad \frac{\vdash M = \underline{n+1} \qquad \vdash N = V'}{\vdash \texttt{ifz}\, M\, \texttt{then}\, N\, \texttt{then}\, N' = V'}$$

Sémantique opérationnelle *bigstep* de l'appel par nom

En changeant légèrement la sémantique, on obtient l'appel par nom

$$\vdash \lambda x.M = \lambda x.M \qquad \frac{\vdash M[x\backslash\mu x.M] = V}{\vdash \mu x.M = V}$$

$$\frac{\vdash M = \lambda x.P \qquad \vdash P[x\backslash N] = V}{\vdash M\,N = V}$$

$$\vdash \underline{n} = \underline{n} \qquad \frac{\vdash M = \underline{m} \qquad \vdash N = \underline{n} \qquad m \otimes n \geq 0}{\vdash M \otimes N = \underline{m \otimes n}}$$

$$\frac{\vdash M = \underline{0} \qquad \vdash M = V}{\vdash \mathtt{ifz}\, M \;\mathtt{then}\; N \;\mathtt{then}\; N' = V} \qquad \frac{\vdash M = \underline{n+1} \qquad \vdash N = V'}{\vdash \mathtt{ifz}\, M \;\mathtt{then}\; N \;\mathtt{then}\; N' = V'}$$

Remarquons qu'alors on peut aussi faire plus fort en évaluant les opérateurs arithmétiques par nom avec des règles comme $0 \times M \to 0$, etc.

Le lambda calcul pur [Church 30]

Termes

$$
\begin{array}{rcll}
M, N, P & ::= & x & \textbf{variable} \\
& \mid & \lambda x.M & \textbf{abstraction} \\
& \mid & MN & \textbf{application}
\end{array}
$$

Découverte de Church: suffisant pour calculer toutes les fonctions calculables $\Rightarrow$ thèse de Church sur le modèle unique de la calculabilité.

Exemples

$(\lambda x.xx)(\lambda x.xy) \rightarrow (\lambda x.xy)(\lambda x.xy) \rightarrow (\lambda x.xy)y \rightarrow yy$

$(\lambda x.xx)(\lambda x.xx) \rightarrow (\lambda x.xx)(\lambda x.xx) \rightarrow \cdots$

$(\lambda x.xxx)(\lambda x.xxx) \rightarrow (\lambda x.xxx)(\lambda x.xxx)(\lambda x.xxx) \rightarrow \cdots$

$\mu x.M = (\lambda y.(\lambda x.M)(yy))(\lambda y.(\lambda x.M)(yy)) \rightarrow^* M[x\backslash\mu x.M]$

$Y = \lambda f.(\lambda y.f(yy))(\lambda y.f(yy))$ et $\mu x.M \simeq Y(\lambda x.M)$

Arithmétique – entiers de Church

$$\texttt{vrai} = \lambda x.\lambda y.x$$

$$\texttt{faux} = \lambda x.\lambda y.y$$

$$\langle M, N\rangle = \lambda z.zMN$$

$$(M)_0 = M\,\texttt{vrai}$$

$$(M)_1 = M\,\texttt{faux}$$

$$\underline{0} = \lambda x.x$$

$$\underline{n+1} = \langle \texttt{faux}, n\rangle$$

$$\texttt{zero} = \lambda x.x\,\texttt{vrai}$$

$$\texttt{ifz}\,P\,\texttt{then}\,M\,\texttt{else}\,N = (\texttt{zero}\,P)MN$$

$$\mu x.M = (\lambda y.(\lambda x.M)(yy))(\lambda y.(\lambda x.M)(yy))$$

Church avait un calcul plus compliqué avec les entiers de Church $\underline{n} = \lambda f.\lambda x.f^n(x)$. Certaines fonctions (par exemple le prédécesseur) sont plus longues à coder.

Programmer avec les entiers de Church est stupide. Nous ne les mentionnons que pour leur application à l'expressibilité du lambda calcul pur.

Encore plus fort

On peut montrer que les lambda termes $K = \lambda x.\lambda y.x$ et $S = \lambda x.\lambda y.\lambda z.xz(yz)$ sont suffisants pour exprimer toutes les lambda expressions (logique combinatoire de **[Curry]**).

Mieux on peut tout exprimer à partir de la seule expression $X = \lambda x.xKSK$ **[Bohm]**.

Par exemple

$K = (XX)X$
$S = X(XX)$
$I = \lambda x.x = SKK = X(XX)((XX)X)((XX)X)$

On peut donc ne vivre qu'avec des X.

Exercice 1 Dans le lambda calcul pur, donner une expression de $\lambda x.x + 1$, $\lambda x.x - 1$, $\lambda x.\lambda y.x + y$, $\lambda x.\lambda y.x \times y$. Et avec les entiers de Church?

Evaluateur de PCF

Deux techniques possibles

- évaluer le programme source avec les techniques développées auparavant
- en appel par nom ou par valeur
- faire la différence sur quelques exemples

Les langages comme Haskell font de l'appel par nom (en ne recalculant pas deux fois la valeur d'un argument de fonction). [appel par nécessité]

Les langages comme ML, Pascal, C, ou Java font de l'appel par valeur.

En TD

- on donnera un analyseur syntaxique pour PCF
- écrire un évaluateur symbolique de PCF
- donner une définition rigoureuse *smallstep* de l'appel par nom et de l'appel par valeur.

La prochaine fois

- interpréteurs (avec environnements)
- types dans les lambda termes
- polymorphisme
- unification et inférence de type