Analyse des Programmes et Sémantique (4)

Prof[r] Jacques Malenfant

(cours de J.Malenfant modifié et enseigné en 2013 par Basile Starynkévitch)

janvier-avril 2013

MI030 - APS

Cours (en M1) du Professeur Jacques Malenfant
http://pagesperso-systeme.lip6.fr/Jacques.Malenfant/ Professeur en informatique au Laboratoire d'Informatique de Paris 6

Enseigné en 2013 par Basile Starynkevitch
http://starynkevitch.net/Basile/
basile@starynkevitch.net & basile.starynkevitch@cea.fr
ingénieur chercheur au CEA, LIST - http://www-list.cea.fr/
travaille sur gcc-melt.org

♠ Nota Bene

Les transparents à fond rose (pages numérotées ♠) et les mots ♠ signalés ainsi ♣ sont de Basile Starynkevitch (dont les opinions n'engagent que lui) ♠

La plupart des transparents sont [recopiés de ceux] de J.Malenfant 2012, que je remercie. Ces transparents sont disponibles sous http://starynkevitch.net/Basile/

Un peu de λ-calcul et des révisions

(On est un peu “en avance”, mais le cours du 4 mars 2013 n’aura pas lieu)
quelques rudiments de λ-calcul
quelques révisions et compléments, notamment sur Prolog
Un parseur un peu plus robuste de BOPL en Prolog/Eclipse/CLP
`github.com/bstarynk/BOPL-APS` devrait

- gérer la position (numéro de ligne)
- accepter et ignorer des lignes de commentaires commençant **#**
- utiliser des **`struct`** **sucre syntaxique** pour les lexèmes et l’AST :

  %%% tokens with location for delimiters (like period, lparen, ...)
  :- export struct(tDelim(loc,cont)).
  %% so code tDelim{cont:C} or tDelim{loc:L,cont:C} or tDelimL, C

!! très bogué ce lundi 11 février 2013 matin

Pourquoi étudier le λ-calcul ?

- Fondement des langages fonctionnels.
- Outil de base de la sémantique dénotationnelle pour représenter les fonctions mathématiques.

♠ antérieur à l'informatique, mais largement étudié, et avec moultes variantes... ♣

♠ le partiel ne portera pas sur le λ-calcul ♣

1 Syntaxe du λ-calcul

2 Sémantique opérationnelle des λ-expressions

3 Stratégies de réduction

4 variétés de λ-expressions

1. Variables (lettres minuscules) v
2. Constantes prédéfinies c (valeurs, opérateurs, ..., *i.e.*, les δ-*règles*)
3. Application de fonctions $(f\ a)$ où f est la fonction et a son argument (combinateurs)
4. λ-abstractions $\lambda v. \ldots$ (définitions de fonctions)

♠ Presque toutes les valeurs sont fonctionnelles ♣

Grammaire des λ-expressions :

$$e ::= v \mid c \mid (e_1\ e_2) \mid \lambda v.e$$

qui sont aussi appelées des λ-*termes*.

♠ informatiquement, une valeur fonctionnelle est généralement une fermeture (= clôture = “closure”) mélant code et données (analogie avec les objets !) ♣

λ-calcul inspirant des languages de programmation

- en ML ou Ocaml : `fun x -> x + 3` $\equiv \lambda x.(x+3)$
- en Lisp ou Scheme : `(lambda (x) (+ x 3))`

Création **dynamique** de valeurs fonctionnelles : `fun d -> (fun x -> x + d)` appliqué à 3 “crée” la fonction `fun y -> y + 3`

Création de valeur fonctionnelle “impossible” en C, C++, Java !

Indécidabilité de l’égalité de deux valeurs fonctionnelles

Curryfication : une fonction à deux arguments est comprise comme une fonctionnelle (qui au premier argument associe la fonction partielle acceptant le second)

Quelques définitions

- *Variable liée* : la variable v dans $\lambda v.e$
  ♠ les occurrences de v dans e sont liées (“bound”) ; une (occurrence de) variable non liée est libre. ♣
- *Corps de fonction* : le terme e dans $\lambda v.e$
- *Opérateur* : le terme e_1 dans $(\underline{e_1}\ e_2)$
- *Opérande* : le terme e_2 dans $(e_1\ \underline{e_2})$

♠ en λ-calcul pur les rôles opérateur/operande sont “symétriques” (appliquer une fonction à elle même, se renvoyer soi-même...) ♣

Exemples de fonctions

$\lambda x.x$	fonction identité (polymorphique)	$(\lambda x.x\ 5) \Rightarrow 5$
$\lambda x.x+1$	fonction successeur	$((\lambda x.x+1)\ 5) \Rightarrow 6$
$\lambda f.\lambda x.(f\ (f\ x))$	fonction double	$(((\lambda f.\lambda x.(f\ (f\ x)))\ (\lambda x.x+1))\ 5)$ $\Rightarrow 7$

où $+$ est une constante fonctionnelle dont l'évaluation est définie par des δ-règles.

♠ Noter que $(\lambda x.x)(\lambda x.x) \Rightarrow \lambda x.x$ ♣

Plus de définitions ; notations

1. *Lettres majuscules* : méta-variables représentant des λ-expressions.
2. *Associativité à gauche* : $E_1\ E_2\ E_3 \equiv ((E_1\ E_2)\ E_3)$
3. *Portée maximale à droite des définitions de fonctions* :
   $\lambda x.E_1\ E_2\ E_3 \equiv \lambda x.(E_1\ E_2\ E_3)$
   et non $(\lambda x.E_1)\ E_2\ E_3$
4. *Curryfication des fonctions à plusieurs arguments* : $\lambda x\,y\,z.e \equiv \lambda x.\lambda y.\lambda z.e$
   mais $\lambda(x\,y\,z).e$ prend un argument qui est un triplet.
5. *Nommage des fonctions, mais par simple substitution textuelle (hors du λ-calcul)* :
   $Twice = \lambda f.\lambda x.(f\ (f\ x))$
   $(Twice\ (\lambda x.x+1)\ 5) \Rightarrow ((\lambda f.\lambda x.(f\ (f\ x)))\ (\lambda x.x+1)\ 5) \Rightarrow 7$

Fonctions curryfiées

Ainsi nommées selon le mathématicien Curry qui les a étudiées la première fois.
Soit `sum(a, b) = a + b` une fonction à représenter en λ-calcul, deux représentations sont possibles :

- $\lambda(a,b).a+b$ de type $int \times int \rightarrow int$
- $\lambda a.\lambda b.a+b$ de type $int \rightarrow int \rightarrow int$ (version curryfiée)

Le résultat de $((\lambda a.\lambda b.a+b)\ 5) \Rightarrow \lambda b.5+b$ est alors vu comme la fonction qui ajoute 5 à son argument.

♠ Curryfication en Ocaml, pas en Scheme ou Lisp. L'efficacité "impose" des optimisations dans le compilateur Ocaml ♣

1 Syntaxe du λ-calcul

2 Sémantique opérationnelle des λ-expressions

3 Stratégies de réduction

Mode d'évaluation

- Évaluation, opérationnellement fondée sur la *réécriture* ou la *réduction* des λ-termes.
- Opération fondamentale : substitution des variables libres par des expressions dans une application pour réaliser le passage des paramètres.
- Exemple : $((\lambda a.\lambda b.a+b)\ 5) \Rightarrow \lambda b.5+b$
  on a substitué la variable *a* par la valeur 5 dans le corps de la fonction.
- Pour les opérations « hors λ-calcul », comme l'addition, elles sont gérées par des règles de réduction spécifiques appelées δ-*règles*.

Substitution et capture de variables

Définition (occurrence libre, liée)

Un occurrence d'une variable v dans une λ-expression est dite *liée* si elle est dans la portée d'un λv, sinon elle est dite *libre*.

Notation ($E[v \to E_1]$)

La notation $E[v \to E_1]$ représente la substitution de toutes les occurrences libres de la variable v dans la λ-expression E par la λ-expression E_1.

- Attention cependant aux captures de variables !
- S'il y a capture, la subsitution *n'est pas valide*, et alors il faut d'abord renommer les variables susceptibles d'être capturées :

$(\lambda x.v)[v \to \lambda y.y + x] \Rightarrow \lambda x.\lambda y.y + x$  capture de x !

$(\lambda x.v)[v \to \lambda y.y + z] \Rightarrow \lambda x.\lambda y.y + z$ OK, pas de capture...

Calcul de l'ensemble des variables libres

Définition (variables libres)

L'ensemble des *variables libres* dans une λ-expression e, noté $VL(e)$ est défini par :

- $VL(c) = \emptyset$ $\quad c \in$ *constante*
- $VL(x) = \{x\}$ pour toute variable x
- $VL(E_1\ E_2) = VL(E_1) \cup VL(E_2)$
- $VL(\lambda x.e) = VL(e) - \{x\}$

Une λ-expression e telle que $VL(e) = \emptyset$ est dite *close*.

Nota : c'est une définition dirigée par la syntaxe !

Calcul des substitutions

Définition (substitution)

La substitution $E[v \to E_1]$ est définie par :

- $v[v \to E_1] = E_1$
- $x[v \to E_1] = x$ si $x \neq v$
- $c[v \to E_1] = c$ si c est une constante
- $(E\ E')[v \to E_1] = (E[v \to E_1]\ E'[v \to E_1])$
- $(\lambda v.E)[v \to E_1] = \lambda v.E$
- $(\lambda x.E)[v \to E_1] = \lambda x.(E[v \to E_1])$ si $x \neq v$ et $x \notin VL(E_1)$
- $(\lambda x.E)[v \to E_1] = \lambda z.(E[x \to z][v \to E_1])$
  si $x \neq v$ et $x \in VL(E_1)$ et $z \neq x$ et $z \notin VL(E_1)$

Exemple de calcul d'une substitution

$$
\begin{aligned}
(\lambda y.(\lambda f.f\ x)\ y)[x \to f\ y] &= (\lambda y.(\lambda f.f\ x)\ y)[y \to z][x \to f\ y] \\
&= (\lambda z.(\lambda f.f\ x)\ z)[x \to f\ y] \\
&= \lambda z.((\lambda f.f\ x)[x \to f\ y])\ (z[x \to f\ y]) \\
&= \lambda z.((\lambda f.f\ x)[x \to f\ y])\ z \\
&= \lambda z.((\lambda f.f\ x)[f \to g][x \to f\ y])\ z \\
&= \lambda z.((\lambda g.g\ x)[x \to f\ y])\ z \\
&= \lambda z.(\lambda g.g\ x[x \to f\ y])\ z \\
&= \lambda z.(\lambda g.g\ (f\ y))\ z
\end{aligned}
$$

La réduction des λ-expressions 1

- Évaluation ≡ simplification : réduire une expression jusqu'à ce que plus aucune règle ne s'applique.
- Règle principale : β-réduction pour l'application de fonction.
- Seconde règle : α-conversion pour le renommage des variables pour éviter les captures.

Définition (α-conversion)

Si v et w sont des variables et E une λ-expression, alors l'*alpha*-conversion est définie par :

$$\lambda v.E \;\Rightarrow_\alpha\; \lambda w.E[v \to w]$$

si w n'apparaît pas dans E.

Les termes $\lambda v.E$ et $\lambda w.E[v \to w]$ sont dits α-*congruents*.

La réduction des λ-expressions 2

Définition (β-réduction)

Si v est une variable et E ; E_1 des λ-expressions, alors la β-réduction est définie par :

$$(\lambda v.E)\ E_1 \quad \Rightarrow_\beta \quad E[v \rightarrow E_1]$$

Le terme $(\lambda v.E)\ E_1$ est appelé β-redex.

L'évaluation d'une λ-expression est constituée d'une suite de β-réductions, possiblement entrelacée d'α-conversion.
Elle est notée $\Rightarrow_\beta^*$

informatiquement

L' α-conversion est très fréquente en interne dans un compilateur (renommage des variables....)
La β-réduction s'apparente à l'appel de fonction

Les δ-règles

- Le λ-calcul est très minimal, en n'admettant que la β-réduction comme règle de réduction.
- Comment représenter les calculs de base, comme les entiers naturels et l'addition ?
- Il existe un encodage des entiers naturels sous la forme de λ-termes et de l'addition comme λ-expression, mais son intérêt est purement théorique.
- En pratique, on trouve plus commode d'étendre le λ-calcul pour inclure en quelque sorte des types de données élémentaires et leurs opérations sous la forme de constantes.
- Ces opérations n'étant pas formulées comme des fonctions du λ-calcul, leur « traitement » est délégué à des règles de réductions spécifiques ajoutées à la β-réduction.
- Exemple :
  Constantes : valeurs dans $\mathbb{N}$ et l'opération *add*.
  δ-règles :
  $$(add\ \Lambda_1\ \Lambda_2) \rightarrow_\delta \Lambda_1 + \Lambda_2 \text{ si } \Lambda_1, \Lambda_2 \in \mathbb{N}$$
  où '+' représente l'addition sur les entiers naturels.

1 Syntaxe du λ-calcul

2 Sémantique opérationnelle des λ-expressions

3 Stratégies de réduction

Évaluation des λ-expressions

But de l'évaluation : réduire la λ-expression à une forme la plus simple possible, et considérer le λ-terme résultant comme la valeur de l'expression.

Définition (forme normale)

Une λ-expression est dite en *forme normale* si elle ne contient aucun β-redex (ni aucune δ-règle applicable), si bien qu'elle ne peut être réduite davantage par β-réduction (ou application d'une δ-règle).

Quatre questions fondamentales :

1. Est-ce que toute λ-expression peut être réduite à une forme normale ?
2. Existe-t'il plus d'une façon (séquences de réductions) de réduire une λ-expression ?
3. S'il existe plus d'une façon de réduire une λ-expression, donnent-elles toutes la même forme normale ?
4. Existe-t'il une façon de réduire les λ-expression qui garantisse l'obtention d'une forme normale ?

Réponses aux questions...

Avant d'aller plus avant, donnons quelques réponses informelles aux quatre questions précédentes :

1. Non, toutes les λ-expressions ne sont pas réductibles à une forme normale. Exemple :

$$((\lambda x.x\ x)\ (\lambda x.x\ x)) \Rightarrow_\beta ((\lambda x.x\ x)\ (\lambda x.x\ x)) \Rightarrow_\beta \ldots$$

♠ Il y a pire $(\lambda x.xxx)(\lambda x.xxx)$ ♣

- Oui, il existe souvent plusieurs façons de réduire une même λ-expression. Par exemple, l'expression $(((\lambda x.\lambda y.y + ((\lambda z.x \times z)\ 3))\ 7)\ 5)$ possède les deux séquences de réduction suivantes :

$$\begin{aligned}(((\lambda x.\lambda y.y + ((\lambda z.x \times z)\ 3))\ 7)\ 5) &\Rightarrow_\beta ((\lambda y.y + ((\lambda z.7 \times z)\ 3))\ 5)\\ &\Rightarrow_\beta 5 + ((\lambda z.7 \times z)\ 3)\\ &\Rightarrow_\beta 5 + 7 \times 3)\\ &\Rightarrow_\beta 5 + 21)\\ &\Rightarrow_\beta 26\end{aligned}$$

ou encore :

$$\begin{aligned}(((\lambda x.\lambda y.y + ((\lambda z.x \times z)\ 3))\ 7)\ 5) &\Rightarrow_\beta (((\lambda x.\lambda y.y + (x \times 3)\ 7)\ 5)\\ &\Rightarrow_\beta ((\lambda y.y + (7 \times 3))\ 5)\\ &\Rightarrow_\beta ((\lambda y.y + 21)\ 5)\\ &\Rightarrow_\beta (5 + 21)\\ &\Rightarrow_\beta 26\end{aligned}$$

- Lorsqu'elles arrivent à donner une forme normale, toutes les stratégies de réduction donnent la même.
- Oui, il existe une stratégie de réduction qui trouve toujours la forme normale, mais elle a aussi ses inconvénients si on veut l'implanter sur ordinateur.

Stratégie de réduction 1

- Par « façons de réduire » une λ-expression, on se réfère au fait qu'il peut exister plus d'un β-redex dans l'expression, et alors l'ordre dans lequel on réduit ces redex suscite les questions précédentes.

Définition (stratégie de réduction)

Une *stratégie de réduction* définit un ordre dans lequel les β-redex sont réduits pour tenter d'obtenir une forme normale.

- Les deux stratégies de réduction les plus connues sont :

  ordre applicatif : réduit toujours le β-redex le plus à gauche et le plus « profond » dans l'arborescence formée par le terme.

  ordre normal : réduit toujours le β-redex le plus à gauche et le plus « extérieur » dans l'arborescence formée par le terme.

Stratégie de réduction 2

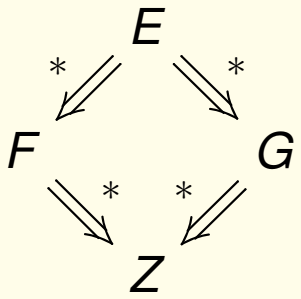
- L'ordre applicatif doit son nom au fait qu'il correspond à l'ordre d'évaluation courant des appels de fonction : on évalue d'abord les arguments (de gauche à droite) qui sont des sous-termes (plus profond) du terme application, puis on appelle la fonction, c'est-à-dire qu'on fait la β-réduction du terme applicatif lui-même.
- L'ordre normal, pour sa part, effectue d'abord la β-réduction (réduction de l'application) et donc passe les arguments sans les évaluer (termes plus profond). Si un paramètre de la fonction est utilisé à plusieurs endroits dans son corps, cet ordre d'évaluation va donc le répéter à chaque occurrence, ce qui demandera donc une implantation plus astucieuse si on veut éviter de le réduire plusieurs fois pendant l'évaluation du corps de la fonction.

Théorème de Church-Rosser I

Theorème (Church-Rosser I)

Pour toutes λ-expressions E, F, G, si $E \Rightarrow^* F$ et $E \Rightarrow^* G$, alors il existe une λ-expressions Z telle que $F \Rightarrow^* Z$ et $G \Rightarrow^* Z$.

C'est la propriété dite de confluence, du *losange* ou encore du *diamant* :



Corollaire

Corollaire

Pour toutes λ-expressions E, F, G, si $E \Rightarrow^* F$ et $E \Rightarrow^* G$, et si F et G sont des formes normales, alors F et G sont des variantes du même λ-terme à des α-conversions près.

Théorème de Church-Rosser II

Théorème (Church-Rosser II)

Pour toutes λ-expressions E et N où N est une forme normale, si $E \Rightarrow^* N$ alors il existe une réduction selon la stratégie en ordre normal de E à N.

Retour sur la quatrième question...

- Ceci répond plus complètement à la quatrième question posée plus tôt : l'existence d'une stratégie de réduction qui trouve toujours la forme normale si elle existe.
- En fait, en conjonction avec la réponse à la première question qui montre qu'il existe des termes qui n'ont pas de forme normale, la stratégie de réduction en ordre normal soit trouve la forme normale si elle existe, soit ne s'arrête pas, c'est-à-dire peut continuer indéfiniment à appliquer la β-réduction à des termes qui n'ont pas de forme normale, comme celui montré à la question 1.

à suivre dans le cours suivant

Quelques révisions sur Prolog :
Satisfaire, puis re-satifaire un [sous-] but
les boites à 4 coins :
call $\rightarrow$ (entrant), $\rightarrow$ exit (sortant),
fail $\leftarrow$ (sortant), $\leftarrow$ redo (entrant)