CM de Modèles de Calcul

Éric Violard

violard@unistra.fr

Université de Strasbourg
UFR de Mathématiques et d'Informatique
Département d'informatique

Objectif du cours

- Se familiariser avec les principaux modèles mathématiques des ordinateurs et de la notion de calcul : Les machines de Turing et le lambda-calcul.

Bibliographie

- Jean-Louis Krivine. "Lambda-calcul, types et modèles". Masson, 1990.
- Edmond Bianco. "Informatique fondamentale : De la machine de Turing aux ordinateurs modernes". Birkhäuser, 1979.

Plan

Introduction

En 1930, le mathématicien américain Alonzo Church (1903 - 1995) imagine un modèle de calcul basé sur la notion de fonction.

Notion de fonction

La notion de fonction est une notion trés importante en programmation.

Point de vue extensionnel (Fonction $\simeq$ énoncé de problème)

Est-ce que la fonction est définie sur son argument ? si oui, quelle valeur ? On s'intéresse au résultat et pas à la façon de l'obtenir.

Point de vue intensionnel (Fonction $\simeq$ algorithme ou programme)

On s'intéresse aux fonctions comme des algorithmes, c'est-à-dire des procédures précises de calcul. Description des fonctions par leur comportement.

Formalisme

- Le lambda-calcul est un formalisme (un petit langage) pour exprimer des fonctions (associé à un ensemble de règles de calcul).
- Les expressions du lambda-calcul notent des **fonctions anonymes** et sont appelées *lambda-termes*.

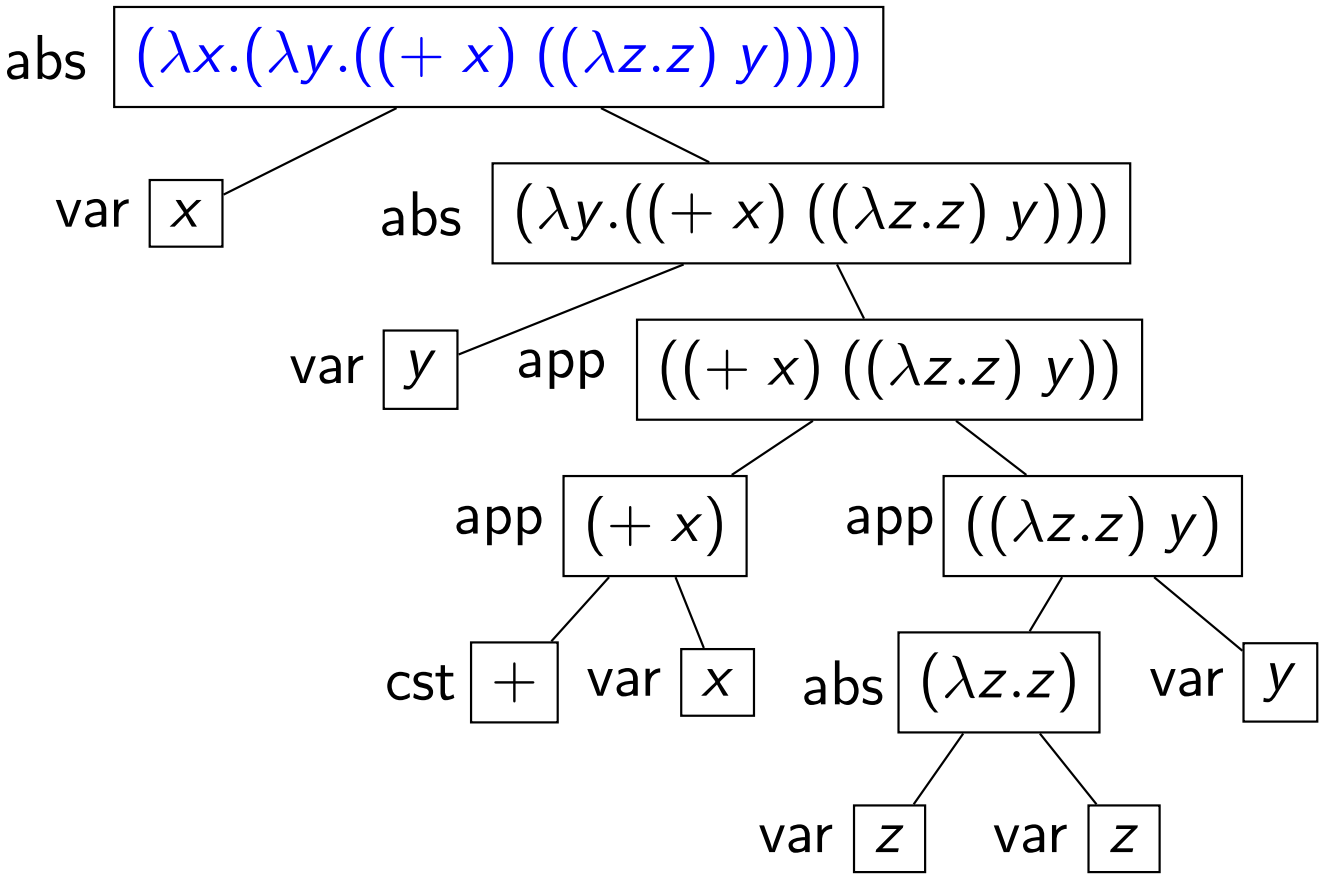
Lambda-termes

Les lambda-termes peuvent être définis (récursivement) comme suit : Un λ-terme est :

- soit une *variable* (typiquement une seule lettre minuscule)
  ex : x, y, z, ...
- soit une *constante* (un nombre, un symbole, ...)
  ex : 0, $+$, π, ...
- soit une *abstraction* de la forme $(\lambda x.M)$
  où x est une variable et M est un λ-terme
  (M est appelé corps de l'abstraction).
- soit une *application* de la forme $(M \_ N)$
  où M et N sont deux λ-termes.

NB : On parle de λ-calcul pur lorsqu'il est sans constantes.

Exemple de λ-terme



Conventions de parenthésage

$$\begin{array}{rcl} \lambda x.(M\ N) & \equiv & \lambda x.M\ N \\ \lambda x.(\lambda y.M) & \equiv & \lambda x.\lambda y.M \\ (M\ N)\ O & \equiv & M\ N\ O \end{array}$$

où $\equiv$ note l'égalité syntaxique et chacun des termes M, N et O soit est une constante, soit est une variable, soit est entourée de parenthèses.

Exemple : $(\lambda x.(\lambda y.((+\ x)\ ((\lambda z.z)\ y)))) \equiv \lambda x.\lambda y.+\ x\ ((\lambda z.z)\ y)$

Interprétation

- constante 0, $+$: nombre, opération (préfixée)
- variable x : variable en maths
- abstraction $\lambda x.M$: fonction anonyme qui à tout x associe M
- application $M\ N$: image de N par M

Exemples

- $\lambda x.x$: la fonction identité qui à tout x associe x.
- $\lambda x.+\ 1\ 2$: la fonction constante qui à tout x associe la somme de 1 et 2.
- $\lambda x.\times\ x\ 2$: la fonction qui à tout nombre associe son double.
- $\lambda y.(\lambda x.x)\ y$: la fonction qui à tout y associe l'image de y par la fonction identité.
- $\lambda x.x\ x$?

Remarque : certains termes ne s'interprétent pas comme une fonction.

Notations différentes, un même concept

Maths	OCaml	λ-calcul
$x \mapsto x$	`fun x -> x`	$\lambda x.x$
$x \mapsto 2x$	`fun x -> 2*x`	$\lambda x. \times\ 2\ x$
$(x \mapsto 2x)(1)$	`(fun x -> 2*x) 1`	$(\lambda x. \times\ 2\ x)\ 1$

Autres exemples

- $\lambda x.\lambda y.x$: la fonction qui étant donné deux arguments x et y (dans cet ordre) associe x.
- $\lambda z.(\lambda x.\lambda y.x)\ 1\ 2$: la fonction qui à tout z associe l'image de 1 et 2 par la fonction précédente (égale à la fonction constante 1).

Autre exemple

- $\lambda x.\lambda y.+\ x\ y$: la fonction qui à tout x et tout y associe la somme de x et y.

En OCaml où le produit cartésien existe, deux écritures sont possibles :

```
fun (x,y) -> x+y  ou  fun x -> fun y -> x+y
```

avec deux types distincts :

```
int * int -> int  ou  int -> int -> int
```

En lambda-calcul, il n'y a pas de produit cartésien : Au lieu d'écrire $\lambda(x,y).\cdots$, on écrit $\lambda x.\lambda y.\cdots$.

Curryfication

La *curryfication* est l'opération qui consiste à transformer une fonction qui prend un n-uplet $(x_1, x_2, \cdots, x_n)$ en une fonction qui prend n arguments $x_1, x_2, \cdots, x_n$ un à un dans cet ordre.

Exemple : la curryfication transforme `fun (x,y) -> x+y` en `fun x -> fun y -> x+y`.

Remarque 1 : cette opération porte le nom de son inventeur, le mathématicien Haskell Curry (1900-1982).
Remarque 2 : cette opération est inversible.

Curryfication

Exemple :

```
# curry2 (fun (x,y) -> x+y);;
- : int -> int -> int = <fun>
# uncurry2 (fun x -> fun y -> x+y);;
- : int * int -> int = <fun>
```

Définition des fonctions curry2 et uncurry2 en OCaml :

```
let curry2 f = fun x -> fun y -> f(x,y);;
let uncurry2 f = fun (x,y) -> f x y;;
```

Notion de réduction

La *réduction* formalise le déroulement du calcul.

Exemple : $(\lambda x.\ \times\ 2\ x)\ \pi$ exprime l'application à π de la fonction qui double son argument, ce qui résulte en 2π qui en λ-calcul s'écrit $\times\ 2\ \pi$.

De fa con plus concise, on dit que $(\lambda x.\ \times\ 2\ x)\ \pi$ *se réduit en* $\times\ 2\ \pi$.

Notation : $(\lambda x.\ \times\ 2\ x)\ \pi\ \rightarrow\ \times\ 2\ \pi$

La relation binaire $\rightarrow$ formalise une étape de calcul.

La réduction est l'enchaînement de plusieurs réductions élémentaires comme celle-ci.

Règles de réduction

Chaque réduction élémentaire est l'application d'une régle qui permet de transformer un sous-terme en un autre.

Le lambda-calcul définit quatres règles de réduction :
Ces règles sont appelées β-, α-, η- et δ-conversion.

δ-conversion

La δ-conversion est la régle qui modélise les opérations impliquant les constantes.

Notation :

$$(op\ c_1\ c_2\ \ldots\ c_n) \underset{\delta}{\rightarrow} c_0$$

où c_0 est le résultat de l'opération op appliquée à c_1, c_2, $\ldots$, c_n.

Exemple :

$$(+\ (\times\ 1\ 2)\ 3) \underset{\delta}{\rightarrow} (+\ 2\ 3) \underset{\delta}{\rightarrow} 5$$

Remarque : On peut appliquer une règle à un sous-terme.

β-conversion

La β-conversion est la régle la plus importante. Elle modélise l'application d'une fonction à un argument.

Exemple : La β-conversion transforme $(\lambda x.\times\ 2\ x)\ \pi$ en $\times\ 2\ \pi$.

Plus généralement, elle consiste à transformer un λ-terme de la forme $(\lambda x.M)\ N$ en le λ-terme M dans lequel la variable x a été substituée par N.

Notation :

$$(\lambda x.M)\ N \underset{\beta}{\rightarrow} M[x := N]$$

La substitution est donc l'opération de base du lambda calcul.

β-conversion

Exemple :

$$
\begin{aligned}
&(\lambda f.\lambda a.f\ (f\ a))\ \lambda x.x \\
&\underset{\beta}{\rightarrow} (\lambda a.f\ (f\ a))[f := \lambda x.x] \equiv \lambda a.(\lambda x.x)\ ((\lambda x.x)\ a) \\
&\underset{\beta}{\rightarrow} \lambda a.(\lambda x.x)\ x[x := a] \equiv \lambda a.(\lambda x.x)\ a \\
&\underset{\beta}{\rightarrow} \lambda a.x[x := a] \equiv \lambda a.a
\end{aligned}
$$

Problème de capture de variable

Exemple : Considérons le λ-terme $(\lambda f.\lambda a.f\ a)\ a\ x$.
D'après son interprétation, il devrait se réduire en $a\ x$.

Or :
$(\lambda f.\lambda a.f\ a)\ a\ x \underset{\beta}{\rightarrow} ((\lambda a.f\ a)[f := a])\ x \equiv (\lambda a.a\ a)\ x$

Et :
$(\lambda a.a\ a)\ x \underset{\beta}{\rightarrow} (a\ a)[a := x] \equiv x\ x \not\equiv a\ x.$

Le résultat est incorrect !

Problème de capture de variable (suite)

La β-conversion $(\lambda f.\lambda \mathbf{a}.f\ \mathbf{a})\ a \underset{\beta}{\rightarrow} \lambda a.a\ a$ est fautive.

Explication : La variable a de l'argument a été confondue avec la variable $\mathbf{a}$ du corps de l'abstraction. On dit que la variable a de l'argument *a été capturée* dans le corps de l'abstraction.

Donc, il faut ajouter une condition à l'application de la β-conversion.

Cette condition repose sur la distinction entre les deux occurrences de la variable a : l'une (a) est libre, l'autre ($\mathbf{a}$) est liée.

Variables libres et variables liées

L'objet qui permet de définir une variable liée est appelé un lieur.

En lambda-calcul, λ est un lieur.

Définition

- Une occurrence d'une variable x est *liée* dans un λ-terme M si elle apparaît dans M à l'intérieur d'un sous-terme de la forme $\lambda x.E$.
- Une occurrence d'une variable x est *libre* dans un λ-terme M si elle n'est pas liée dans M.

Exemple

Soit M, le λ-terme :

$$(\lambda \mathbf{x}.\mathbf{x}\ y)\ \lambda \mathbf{y}.x\ \lambda \mathbf{x}.\mathbf{x}\ (\mathbf{y}\ \mathbf{x})$$

Remarque 1 : Cette notion de variable libre ou liée est toujours relative à un terme.
Remarque 2 : Une variable peut apparaître une fois libre et une autre fois liée dans un même terme.
Autre exemple : $x\ (\lambda \mathbf{x}.\mathbf{x})$

Combinateur

On appelle *combinateur* un terme qui n'a pas variable libre.

Exemples :

$\Delta \stackrel{\text{def}}{=} \lambda x.x\ x$

$\Omega \stackrel{\text{def}}{=} (\lambda x.x\ x)\ (\lambda x.x\ x)$

$Y \stackrel{\text{def}}{=} \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$

Problème de capture

$$(\lambda x.M)\ N \underset{\beta}{\rightarrow} M[x := N]$$

- Le problème survient lorsque :

  Il existe une variable y libre dans N et liée dans M.

  Exemple : $(\lambda f.\lambda a.f\ a)\ a$

  La variable a est libre dans a et liée dans $\lambda a.f\ a$.

- On ne doit pas appliquer la β-conversion dans ce cas.

Substitution

$$M[x := N]$$

On ne substitue par N que les occurrences libres de x dans M.

Exemple :

$(x\ (\lambda x.x))[x := y] \equiv y\ (\lambda x.x)$

Retour sur notre exemple

On ne peut donc pas réduire directement le λ-terme $(\lambda f.\lambda a.f\ a)\ a\ x$ à cause du problème de capture de la variable a.

Cependant, on peut remarquer que le sous-terme $\lambda f.\lambda a.f\ a$ a la même interprétation que $\lambda f.\lambda b.f\ b$ obtenue en renommant la variable a par b, et poursuivre la réduction :

$$(\lambda f.\lambda b.f\ b)\ a\ x$$
$$\underset{\beta}{\rightarrow} ((\lambda b.f\ b)[f := a])\ x \equiv (\lambda b.a\ b)\ x$$
$$\underset{\beta}{\rightarrow} (a\ b)[b := x] \equiv a\ x.$$

Ce qui donne le bon résultat !

α-conversion

On vient d'identifier un autre règle de réduction, la α-conversion qui consiste en le renommage d'une variable liée.

$$\lambda x.M \underset{\alpha}{\rightarrow} \lambda y.M[x := y]$$

où y est une nouvelle variable.

Exemple : $\lambda f.\lambda a.f\ a \underset{\alpha}{\rightarrow} \lambda f.\lambda b.(f\ a)[a := b] \equiv \lambda f.\lambda b.f\ b$

η-conversion

La η-conversion est la régle la plus rarement employée. Elle remplace un terme de la forme $\lambda x.(F\ x)$ par F s'il n'y a pas d'occurrence libre de x dans F. (Dans ce cas, $\lambda x.(F\ x)$ et F ont la même interprétation.)

$$\lambda x.(F\ x) \underset{\eta}{\rightarrow} F$$

si x n'est pas libre dans F.

Règles de calcul

$(\lambda x.M)\ N \underset{\beta}{\rightarrow} M[x := N]$ s'il n'existe pas de variable dont une occurrence est libre dans N et une autre est liée dans M.

$\lambda x.M \underset{\alpha}{\rightarrow} \lambda y.(M[x := y])$ où y est une variable qui n'apparaît pas dans M.

$\lambda x.M\ x \underset{\eta}{\rightarrow} M$ s'il n'y a pas d'occurrence libre de x dans M.

$op\ c_1\ c_2\ \ldots\ c_n \underset{\delta}{\rightarrow} c_0$ où op est un opérateur, les c_i $(i = 0..n)$ sont des constantes et c_0 exprime le résultat de l'opération.

Exemple de réduction

$$
\begin{aligned}
&(\lambda x.x\ (\lambda x.\lambda b.b\ c\ x)\ (\lambda x.x)\ \lambda x.\lambda y.x)\ \lambda f.\lambda c.f\ c \\
&\xrightarrow[\beta]{} (\lambda f.\lambda c.f\ c)\ (\lambda x.\lambda b.b\ c\ x)\ (\lambda x.x)\ \lambda x.\lambda y.x \\
&\xrightarrow[\alpha]{} (\lambda f.\lambda d.f\ d)\ (\lambda x.\lambda b.b\ c\ x)\ (\lambda x.x)\ \lambda x.\lambda y.x \\
&\xrightarrow[\beta]{} (\lambda d.(\lambda x.\lambda b.b\ c\ x)\ d)\ (\lambda x.x)\ \lambda x.\lambda y.x \\
&\xrightarrow[\beta]{} (\lambda x.\lambda b.b\ c\ x)\ (\lambda x.x)\ \lambda x.\lambda y.x \\
&\xrightarrow[\beta]{} (\lambda b.b\ c\ \lambda x.x)\ \lambda x.\lambda y.x \\
&\xrightarrow[\beta]{} (\lambda x.\lambda y.x)\ c\ \lambda x.x \\
&\xrightarrow[\beta]{} (\lambda y.c)\ \lambda x.x \\
&\xrightarrow[\beta]{} c
\end{aligned}
$$

Forme normale

Les règles de réduction permettent de simplifier des termes jusqu'à obtenir un terme qu'on ne peut plus simplifier (autrement que par α-conversion).
Un tel terme est appelé une *forme normale*.

Exemples

Parmi ces λ-termes, lesquels sont en forme normale :

- $\lambda x.x$
- $((\lambda x.\lambda y.x)\ v)\ w$
- $(\lambda x.\lambda y.x\ v)\ w$
- $\lambda x.\lambda y.x\ v\ w$

Redex

Étant donné un λ-terme, il peut exister plusieurs sous-termes que l'on peut choisir de réduire par β-conversion.

Exemple : $(\lambda x.(\lambda y.\lambda x. + x\ y)\ 4\ x)\ 3$

Un tel sous-terme (qui est donc de la forme $(\lambda x.M)\ N$) est appelé *redex*.

On peut indiquer l'endroit dans un λ-terme où se trouve un redex en écrivant le symbole $\curlywedge$ en dessous de l'application (comme ceci $(\lambda x.M) \curlywedge N$).

Exemple : $(\lambda x.(\lambda y.\lambda x. + x\ y) \curlywedge 4\ x) \curlywedge 3$

Questions

1. Le processus de réduction termine-t-il toujours, autrement dit aboutit-on toujours à une forme normale ?
2. La forme normale est-elle unique ? En général, combien a-t-on de formes normales d'un même terme ?

Terminaison

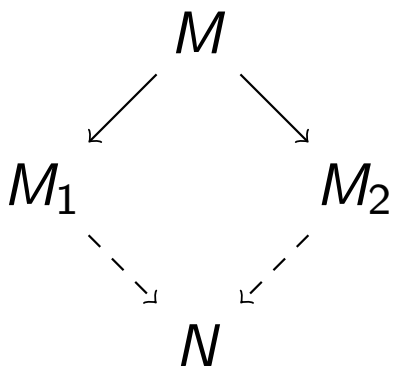
La réduction ne termine pas toujours.

Exemple : Soit Ω, le λ-terme : $(\lambda x.x\ x)\ (\lambda x.x\ x)$. On a :

$$\begin{aligned}
\Omega &\stackrel{\text{def}}{=} (\lambda x.x\ x)_{\lambda}(\lambda x.x\ x) \\
&\underset{\beta}{\rightarrow} (x\ x)[x := \lambda x.x\ x] \equiv (\lambda x.x\ x)_{\lambda}(\lambda x.x\ x) \\
&\underset{\beta}{\rightarrow} (\lambda x.x\ x)_{\lambda}(\lambda x.x\ x) \\
&\underset{\beta}{\rightarrow} (\lambda x.x\ x)_{\lambda}(\lambda x.x\ x) \\
&\underset{\beta}{\rightarrow} \ldots
\end{aligned}$$

Résultat théorique de Church-Rosser

Si un même λ-terme M se réduit en un λ-terme M_1 (en choisissant certains redex) et en un autre λ-terme M_2 (en faisant d'autres choix de redex), alors il existe un λ-terme N tel que M_1 et M_2 se réduisent en N.
Autrement dit, la réduction est confluente.



Corollaire au théorème de confluence

Si un même λ-terme M se réduit en deux λ-termes M_1 et M_2 sous forme normale, alors M_1 et M_2 sont syntaxiquement égales à des renommages (α-conversions) près.

Stratégies de réduction

Le choix du redex à chaque étape d'une réduction est appelé *stratégie de réduction*. Une telle stratégie défini un ordre dans lequel les redex sont réduits.

Il y a en a deux classiques :

- L'ordre normal (NOR) :
  redex le plus à l'extérieur et le plus à gauche.
- L'ordre applicatif (AOR) :
  redex le plus à l'intérieur et le plus à gauche.

Exemple

Réduction de $(\lambda x.(\lambda a. \times\ a\ a)\ x)\ ((\lambda y.y)\ 2)$.

avec NOR :

$$(\lambda x.(\lambda a. \times\ a\ a)\ x)_{\curlywedge}((\lambda y.y)\ 2)$$
$$\underset{\beta}{\rightarrow} (\lambda a. \times\ a\ a)_{\curlywedge}((\lambda y.y)\ 2)$$
$$\underset{\beta}{\rightarrow} \times((\lambda y.y)_{\curlywedge}2)\ ((\lambda y.y)\ 2)$$
$$\underset{\beta}{\rightarrow} \times 2\ ((\lambda y.y)_{\curlywedge}2) \underset{\beta}{\rightarrow} \times\ 2\ 2 \underset{\delta}{\rightarrow} 4$$

avec AOR :

$$(\lambda x.(\lambda a. \times\ a\ a)_{\curlywedge}x)\ ((\lambda y.y)\ 2)$$
$$\underset{\beta}{\rightarrow} (\lambda x. \times\ x\ x)\ ((\lambda y.y)_{\curlywedge}2)$$
$$\underset{\beta}{\rightarrow} (\lambda x. \times\ x\ x)_{\curlywedge}2 \underset{\beta}{\rightarrow} \times\ 2\ 2 \underset{\delta}{\rightarrow} 4$$

Exemple

Réduction de $(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)\ (\lambda z.z\ z))$.

avec NOR :

$(\lambda x.\lambda y.y)_{\curlywedge}((\lambda z.z\ z)\ (\lambda z.z\ z))$

$\underset{\beta}{\rightarrow}\ \lambda y.y$

avec AOR :

$(\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_{\curlywedge}(\lambda z.z\ z))$

$\underset{\beta}{\rightarrow}\ (\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_{\curlywedge}(\lambda z.z\ z))$

$\underset{\beta}{\rightarrow}\ (\lambda x.\lambda y.y)\ ((\lambda z.z\ z)_{\curlywedge}(\lambda z.z\ z))$

$\underset{\beta}{\rightarrow}\ \ldots$

Résultat théorique de Curry

Si un λ-terme a une forme normale, on peut l'obtenir en réduisant toujours le redex le plus à l'extérieur et le plus à gauche.
Autement dit, l'ordre normal de réduction conduit à coup sûr à la forme normale lorsqu'elle existe.

NB : AOR est généralement plus rapide que NOR, mais avec AOR, on est pas sûr d'atteindre la forme normale lorsqu'elle existe.

Pouvoir expressif du λ-calcul pur

On peut représenter par des λ-termes les données usuelles des langages de programmation comme les booléens, les entiers, les couples, et on peut aussi représenter toutes les fonctions récursives.

Dans ce contexte, on utilise parfois la relation d'égalité $=$ entre 2 λ-termes pour signifier que ces λ-termes ont le même comportement.
Cette relation peut être formellement définie comme la fermeture transitive des conversions : On dira que $M = N$ ssi $M \equiv N$ ou bien s'il existe une suite finie de λ-termes : M_1, M_2, ..., M_n telle que $M_1 \equiv M$, pour i = 1,..,n-1, $M_i \to M_{i+1}$ ou bien $M_{i+1} \to M_i$, et $M_n \equiv N$.

Booléens

On peut commencer par représenter la conditionnelle :
si <cond> alors <e1> sinon <e2>
comme une fonction ternaire modélisée par une expression de la forme : $COND \stackrel{\text{def}}{=} \lambda c.\lambda v.\lambda f.E(c, v, f)$
Les constantes booléennes seront modélisées par des termes à définir, notés VRAI et FAUX.

Booléens II

On prend pour $E(c, v, f)$ l'expression la plus simple possible :
$E(c, v, f) = (c\ v\ f)$. Alors : $COND \stackrel{\text{def}}{=} \lambda c.\lambda v.\lambda f.c\ v\ f$
et on doit avoir :
$COND\ VRAI$ <e1> <e2> $\rightarrow$ $VRAI$ <e1> <e2> $\rightarrow$ <e1>
$COND\ FAUX$ <e1> <e2> $\rightarrow$ $FAUX$ <e1> <e2> $\rightarrow$ <e2>
ce qui s'obtient avec :
$VRAI \stackrel{\text{def}}{=} \lambda x.\lambda y.x$
$FAUX \stackrel{\text{def}}{=} \lambda x.\lambda y.y$

Les opérateurs booléens (NON, ET, OU) peuvent être définis en utilisant la conditionnelle.

Entiers

Plusieurs méthodes sont possibles, voici celle de Church :
L'entier n est modélisé par le λ-terme $\underline{n}$:

$$\underline{n} \stackrel{\text{def}}{=} \lambda f.\lambda x.f^n x$$

$$\text{où } f^n x \equiv \underbrace{f\ (f \ldots (f}_{n \times}\ x) \ldots)$$

Ainsi :
$\lambda f.\lambda x.x$ représente l'entier 0.
$\lambda f.\lambda x.f\ x$ représente l'entier 1.
$\lambda f.\lambda x.f\ (f\ x)$ représente l'entier 2.
...

Entiers II

$$\boxed{\underline{n}\ f\ x = f^n\ x}$$

Fonction SUCC (successeur) :

On a : $\underline{n+1}\ f\ x = f^{n+1}\ x = f\ (f^n\ x) = f\ (\underline{n}\ f\ x)$

D'où : $SUCC \stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$

Entiers III

Fonction ADD (addition) :

On a :
$\underline{n+m}\ f\ x = f^{n+m}\ x = f^n\ (f^m\ x) = f^n\ (\underline{m}\ f\ x) = \underline{n}\ f\ (\underline{m}\ f\ x)$

D'où : $ADD \stackrel{\text{def}}{=} \lambda n.\lambda m.\lambda f.\lambda x.n\ f\ (m\ f\ x)$

Entiers IV

Fonction MULT (multiplication) :

On a : $\underline{nm}\ f\ x$

$$= \underbrace{\underline{n} + \underline{n} + \ldots + \underline{n}}_{m\times}\ f\ x = \underbrace{\underline{n}\ f\ (\underline{n}\ f\ (\ldots(\underline{n}\ f}_{m\times}\ x)\ldots))$$

$$= (\underline{n}\ f)^m\ x = \underline{\mathrm{m}}\ (\underline{n}\ f)\ x$$

D'où : $MULT \stackrel{\mathrm{def}}{=} \lambda n.\lambda m.\lambda f.\lambda x.m\ (n\ f)\ x$
qui peut se simplifier (par η-conversion) en :
$MULT = \lambda n.\lambda m.\lambda f.m\ (n\ f)$

Entiers V

Fonction POW (puissance) :

On a : $\underline{n^m}\ f\ x$

$$= \underbrace{\underline{n} \times \underline{n} \times \ldots \times \underline{n}}_{m\times}\ f\ x = \underbrace{\underline{n}\ (\underline{n}\ (\ldots (\underline{n}}_{m\times}\ f) \ldots))\ x$$

$$= \underline{n}^m\ f\ x = \underline{\mathrm{m}}\ \underline{n}\ f\ x$$

D'où : $POW \stackrel{\text{def}}{=} \lambda n.\lambda m.\lambda f.\lambda x.m\ n\ f\ x$
qui peut se simplifier (par η-conversion) en :
$POW = \lambda n.\lambda m.m\ n$

Entiers VI

Fonction ISZERO (test d'égalité à 0) :

$ISZERO \stackrel{\text{def}}{=} \lambda n.n\ (\lambda x.FAUX)\ VRAI$

Fonction PRED (prédecesseur) :

$PRED \stackrel{\text{def}}{=}$

$\lambda n.\lambda f.\lambda x.FST\ (n\ (\lambda p.PAIR\ (SND\ p)\ (f\ (SND\ p)))\ (\lambda t.t\ x\ x))$

Couples

Il s'agit de modéliser des fonctions qui créent des couples et en renvoient les éléments. Nous noterons ces fonctions *PAIR*, et *FST* et *SND*.
On peut prendre : $PAIR \stackrel{\text{def}}{=} \lambda x.\lambda y.\lambda t.t\ x\ y$
On doit ensuite vérifier :
$FST\ (PAIR\ \langle e1\rangle\ \langle e2\rangle) \rightarrow FST\ (\lambda t.t\ \langle e1\rangle\ \langle e2\rangle) \rightarrow \langle e1\rangle$
$SND\ (PAIR\ \langle e1\rangle\ \langle e2\rangle) \rightarrow SND\ (\lambda t.t\ \langle e1\rangle\ \langle e2\rangle) \rightarrow \langle e2\rangle$

Cela s'obtient si *FST* (resp. *SND*) applique son argument à *VRAI* (resp. *FAUX*). D'où :
$FST \stackrel{\text{def}}{=} \lambda t.t\ VRAI$
$SND \stackrel{\text{def}}{=} \lambda t.t\ FAUX$

Récursivité

Il s'agit de modéliser des fonctions définies en fonction d'elles-mêmes : dont le nom apparaît dans la définition.

Exemple : fac(n) = if n=0 then 1 else n$\times$fac(n-1)

On ne peut pas simplement modéliser cette fonction par :

$FAC = \lambda n.COND\ (ISZERO\ n)\ 1\ (\times\ n\ (FAC\ (PRED\ n)))$

(en utilisant un modéle des booléens et des entiers)
car en λ-calcul, les fonctions sont anonymes !

Récursivité II

Mais on peut écrire l'équation précédente sous une forme où la récursivité s'exprime plus simplement, on a :

$$FAC = H\ FAC$$

avec $H \stackrel{\text{def}}{=} \lambda f.\lambda n.COND\ (ISZERO\ n)\ 1\ (\times\ n\ (f\ (PRED\ n)))$

Cette équation met en évidence que *FAC* est un *point fixe* de *H*.

La solution pour représenter *FAC* est donc un point fixe de *H*.

Récursivité III

Soit un λ-terme Y qui prend en argument une fonction et qui retourne un point fixe de cette fonction : Un tel λ-terme est appelé *combinateur de point fixe* et vérifie $Y\ H = H\ (Y\ H)$.

Alors, notre problème est résolu : $FAC \stackrel{\text{def}}{=} Y\ H$

Récursivité IV

Vérification : calcul de $FAC\ \underline{2}$

$$
\begin{aligned}
&FAC\ \underline{2} \stackrel{\text{def}}{=} Y\,\text{人}\,H\ \underline{2}\\
&= H\,\text{人}\,(Y\ H)\ \underline{2}\\
&= (\lambda n.COND\ (ISZERO\ n)\ \underline{1}\ (\times\ n\ (Y\ H\ (PRED\ n))))\ \underline{2}\\
&= COND\,\text{人}\,(ISZERO\ \underline{2})\ \underline{1}\ (\times\ \underline{2}\ (Y\ H\ (PRED\ \underline{2})))\\
&= \times\ \underline{2}\ (COND\,\text{人}\,(ISZERO\ \underline{1})\ \underline{1}\ (\times\ \underline{1}\ (Y\ H\ (PRED\ \underline{1}))))\\
&= \times\ \underline{2}\ (\times\ \underline{1}\ (COND\,\text{人}\,(ISZERO\ \underline{0})\ \underline{1}\ (\times\ \underline{0}\ (Y\ H\ (PRED\ \underline{0})))))\\
&= \times\ \underline{2}\ (\times\ \underline{1}\ \underline{1}) = \underline{2}
\end{aligned}
$$

Récursivité V

Reste à définir un combinateur de point fixe Y.

Le combinateur de Curry :

$$Y \stackrel{\text{def}}{=} \lambda f.(\lambda x.f\ (x\ x))\ (\lambda x.f\ (x\ x))$$

est tel que $Y = H\ (Y\ H)$

On peut aussi utiliser le combinateur de Turing :

$$\Theta \stackrel{\text{def}}{=} (\lambda x.\lambda y.y\ (x\ x\ y))\ (\lambda x.\lambda y.y\ (x\ x\ y))$$

qui vérifie $\Theta\ H \rightarrow H\ (\Theta\ H)$.