Fondements de la Programmation

CM: Paulin de Naurois - TD: Domenico Ruoppolo

Notes de Cours (P. Boudes, V. Mogbil, P. de Naurois.)

Bilbiographie

Ces notes de cours utilisent le livre de de Neil Jones "Computability and Complexity: from a Programming Perspective" et celui de P. Clote / E. Kranakis "Boolean Functions and Computation Models" pour les modèles de calculs, le livre de Barendregt "The Lambda Calculus, Its Syntax and Semantics" et les notes de cours de J. Goubault-Larrecq, de T. Hardin et de Y. Bertot pour le λ-calcul.

1 Introduction

Thèse de Church-Turing

Toutes les formalisations raisonnables de la notion de calcul et d'algorithme sont équivalentes.

Remarque essentielle: On distingue deux notions:

- Une *fonction*, qui, à chaque entrée (par exemple un entier naturel), associe une valeur (par exemple un entier naturel).
- Un algorithme, qui est une description précise de la façon de calculer un résultat sur une une entrée donnée. Un programme est la représentation d'un algorithme dans un langage de programmation fixé.

Un algorithme réalise une fonction partielle: la valeur associée à chaque entrée est le résultat du calcul de l'algorithme sur cette entrée, si l'algorithme termine. Une fonction peut être réalisée par plusieurs algorithmes différents, ou par aucun. L'ensemble des algorithmes de $\mathbb{N} \to \mathbb{N}$ est dénombrable : il suffit pour s'en persuader de considérer leur écriture sur un alphabet fini. L'ensemble des fonctions de $\mathbb{N} \to \mathbb{N}$ est quant à lui indénombrable - plus précisément, en bijection avec $\mathbb{R}$. Il existe donc infiniment plus de fonctions que d'algorithmes, et seule une infime proportion de ces fonctions est réalisable par des algorithmes.

La Thèse de Church-Turing peut être reformulée comme suit: Quel que soit le formalisme raisonnable choisi pour décrire les algorithmes, les fonctions réalisées sont les mêmes. Pour prouver cette thèse de Church-Turing pour deux formalismes de description des algorithmes, il faut donner une méthode systématique de traduction des instructions de programmation d'un formalisme dans le second: cette méthode systématique est elle-même un algorithme, qu'on appelle *compilateur*.

Modèles de calcul

Voici quelques modèles abstraits de calcul que l'on va étudier :

- Machine abstraite de Turing (MT) et avec langage d'instruction (MTI),
- Machine abstraite à compteurs (MC), à adressage indirect (RAM),
- Machine abstraite parallèles (PRAM),

- Programmation par fonctions récursives (Kleene)
- Programmation fonctionnelle: λ-calcul (Church)

Pour tout ces formalismes, la thèse de Church-Turing est vérifiée. Ces modèles sont très différents mais certains ont des aspects communs : un algorithme est un ensemble fini d'instructions, l'exécution est faite pas-à-pas de façon déterministe, en utilisant une quantité finie de mémoire ou de temps, pour lesquels il n'y a pas de borne a priori sur leur quantité.

Notations Communes

On adopte les notations suivantes pour les modèles impératifs (MT - MTI - MC - RAM - PRAM), issues de livre de N. Jones.

- $M - data$ décrit la structure de donnée des entrées et des sorties - i.e le domaine et le co-domaine des fonctions calculées.
- $M - store$ décrit la structure de données *interne* à la machine: cést une description de sa mémoire. Cette structure dépend du modèle choisi.
- $M - prog$ décrit le langage d'instructions du modèle.

Pour chacun de ces modèles un programme P est défini comme une suite finie d'instructions, numérotées. La dernière instruction est vide, et le programme termine lorsqu'il effectue cette instruction vide.

$$\begin{aligned} P \quad = \quad & 1 : I_1, \\ & 2 : I_2, \\ & \vdots \\ & m : I_m, \\ & m+1 : \end{aligned}$$

Exécutions La mémoire de la machine est initialisée avec l'entrée du programme. Cette initialisation est donnée par la fonction $Readin : M - data \to M - store$, qui décrit l'état initial de la machine sur l'entrée spécifiée. A la fin de l'exécution, le résultat du calcul est lu dans la mémoire de la machine. Cette lecture du résultat est donnée par la fonction $Readout : M - store \to M - data$. Pendant l'exécution du programme, la machine passe d'un état au suivant, par le biais de *transitions*. Chaque état de la machine est donné par un couple (l, σ), où $l \in \{1, \cdots, m+1\}$ est un numéro d'instruction du programme, et $\sigma \in M - store$ est la mémoire. On note une transition d'un état au suivant comme suit: $p \vdash (i, \sigma_i) \to (j, \sigma_j)$.

L'effet d'un programme p sur l'entrée x est $[p](x) = y$ si et seulement si

1. $\sigma_0 = Readin(x)$
2. $p \vdash (1, \sigma_0) \to^* (m+1, \sigma)$
3. $y = Readout(\sigma)$

2 Machines de Turing avec langage d'instructions (MTI)

On travaille sur un alphabet fini, en général binaire $\{0, 1\}$. La machine dispose d'un ruban bi-infini pour mémoriser l'entrée et faire les calculs. Il s'agit d'une suite infinie de cellules contenant chacune un symbole de l'alphabet, ou le symbole Blanc (B); seul un nombre fini de cellules contiennent un symbole non-blanc.

On dispose d'une tête de lecture/écriture pour lire/modifier le symbole courant, et d'instructions de contrôle. Le langage d'instruction permet de déplacer la tête de lecture/écriture sur le ruban d'une cellule, à gauche ou à droite. On peut écrire un symbole sur le ruban, et tester quel est le symbole courant. La convention d'initialisation est que le ruban ne contienne que des symboles B (blanc) sauf à droite de la tête de lecture/écriture où l'on a l'entrée. Il en est de même pour le résultat lorsque l'exécution est terminée.

2.1 Synthèse

- $MTI - data = \{0,1\}^*$
- $MTI - store = \{L\underline{S}R | L, R : String\ S : Symbol\}$, où la position de la tête de lecture/écriture est la cellule soulignée, avec:

$$\begin{array}{rcl} L, R : String & ::= & S\ String \mid \epsilon\ (motvide) \\ S, S' : Symbol & ::= & 0 \mid 1 \mid B \end{array}$$

- $Readin(x) = L\underline{B}xBR$, avec $L = LB$ et $R = BR$
- $Readout(L\underline{B}yBR) = y$, i.e. le mot binaire commençant à droite de la tête de lecture, jusqu'au premier symbole B.
- $MTI - prog ::= right \mid left \mid write\ S \mid if\ S\ goto\ l'\ else\ l"$

2.2 Transitions

$p \vdash (l, L\underline{S}S'R)$	$\rightarrow$	$(l+1, LS\underline{S'}R)$	si $I_l = right$
$p \vdash (l, L\underline{S})$	$\rightarrow$	$(l+1, LS\underline{B})$	si $I_l = right$
$p \vdash (l, LS\underline{S'}R)$	$\rightarrow$	$(l+1, L\underline{S}S'R)$	si $I_l = left$
$p \vdash (l, \underline{S}R)$	$\rightarrow$	$(l+1, \underline{B}SR)$	si $I_l = left$
$p \vdash (l, L\underline{S}R)$	$\rightarrow$	$(l+1, L\underline{S'}R)$	si $I_l = write\ S'$
$p \vdash (l, L\underline{S}R)$	$\rightarrow$	$(l', L\underline{S}R)$	si $I_l = if\ S\ goto\ l'\ else\ l"$
$p \vdash (l, L\underline{S'}R)$	$\rightarrow$	$(l", L\underline{S}R)$	si $I_l = if\ S\ goto\ l'\ else\ l"$

2.3 Exemple

La machine qui réalise la fonction constante 5 en binaire peut s'écrire comme suit:

$$\begin{array}{rcl} P & = & 1 : right, \\ & & 2 : if\ B\ goto\ 4\ else\ 5, \\ & & 3 : write\ B, \\ & & 4 : goto\ 1, \\ & & 5 : write\ 1, \\ & & 6 : left, \\ & & 7 : write 0, \\ & & 8 : left, \\ & & 9 : write 1, \\ & & 10 : left, \\ & & 11 : \end{array}$$

Machine de Turing avec langage d'instructions, à k rubans

Le modèle de la machine de Turing se généralise à plusieurs rubans. Dans ce cas, seul le premier ruban est utilisé pour la lecture de l'éntrée et du résultat du calcul. Les instructions sont les mêmes que précédemment, mais propres à *chaque* ruban.

- $MTI^k - data = \{0,1\}^*$
- $MTI^k - store = Tape^k$, où $Tape = L\underline{S}R$, avec

$$\begin{array}{rcl} L, R : String & ::= & S\ String \mid \epsilon\ (motvide) \\ S, S' : Symbol & ::= & 0 \mid 1 \mid B \end{array}$$

- $Readin(x) = (L\underline{B}xBR, L\underline{B}R, \cdots, L\underline{B}R)$, avec $L = LB$ et $R = BR$
- $Readout(L\underline{B}yBR, Tape_2, \cdots, Tape_k) = y$, i.e. le mot binaire commençant à droite de la tête de lecture du premier ruban, jusqu'au premier symbole B. Le contenu des autres rubans est ignoré.
- $MTI - prog ::= right_i \mid left_i \mid write_i\ S \mid if_i\ S\ goto\ l'\ else\ l"$

3 Machine à Compteurs (MC)

On dispose d'un nombre fini, non borné, de compteurs (registres) $X_0, X_1, X_2, \cdots$ pour mémoriser des entiers naturels. Le langage d'instruction permet de tester si un compteur est à zéro, d'incrémenter/décrémenter un compteur de 1. La convention d'initialisation est que tous les compteurs sont à $\perp$, désignant la valeur "indéterminée", qui est interprétée pour le calcul comme un 0, sauf le premier compteur, contenant l'entrée. Le résultat est aussi donné par ce premier compteur lorsque l'exécution est finie.

3.1 Synthèse

- $MC - data = \mathbb{N}$
- $MC - store = \{\sigma | \sigma : \mathbb{N} \to \mathbb{N} \cup \{\perp\}\}$, où $\sigma(i)$ dénote le contenu du compteur X_i pour $i \in \mathbb{N}$.
- $Readin(x) = \sigma : \left\{ \begin{array}{ll} 0 & \to x \\ i > 0 & \to \perp \end{array} \right.$
- $Readout(\sigma) = \sigma(0)$
- $MTI - prog ::= X_i := X_i + 1 \mid X_i := X_i - 1 \mid if\ X_i = 0\ goto\ l'\ else\ l"$.

3.2 Transitions

$p \vdash (l, \sigma)$	$\to$	$\left(l+1, \sigma' : \left\{ \begin{array}{ll} i & \to \sigma(i)+1 \\ j \neq i & \to \sigma(j) \end{array} \right.\right)$	si $I_l = X_i := X_i + 1$
$p \vdash (l, \sigma)$	$\to$	$\left(l+1, \sigma' : \left\{ \begin{array}{ll} i & \to \sigma(i)-1 \\ j \neq i & \to \sigma(j) \end{array} \right.\right)$	si $I_l = X_i := X_i - 1$ et $\sigma(i) \neq 0$
$p \vdash (l, \sigma)$	$\to$	$(l+1, \sigma)$	si $I_l = X_i := X_i - 1$ et $\sigma(i) = 0$
$p \vdash (l, \sigma)$	$\to$	(l', σ)	si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) = 0$
$p \vdash (l, \sigma)$	$\to$	$(l", \sigma)$	si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) \neq 0$

3.3 Exemple

La machine qui calcule le modulo à 2 de son entrée peut s'écrire comme suit:

$$\begin{aligned} P \quad = \quad & 1: \ if\ X_0 = 0\ goto\ 7\ else\ 2, \\ & 2: \ X_0 = X_0 - 1, \\ & 3: \ if\ X_0 = 0\ goto\ 6\ else\ 4, \\ & 4: \ X_0 = X_0 - 1, \\ & 5: \ goto\ 1, \qquad\qquad (if\ X_0 = 0\ goto\ 1\ else\ 1) \\ & 6: \ X_0 = X_0 + 1, \\ & 7: \end{aligned}$$

3.4 Extension de l'ensemble d'instructions

Les instructions suivantes, dont l'interprétation est évidente, sont facilement programmables avec les MC, et peuvent être utilisées dans l'écriture des programmes:

$$X_i := X_j \mid goto\ l \mid if\ X_i < X_j\ goto\ l'\ else\ l" \mid if\ X_i = X_j\ goto\ l'\ else\ l"$$
$$x_i := constante \mid if\ X_i = constante\ goto\ l'\ else\ l" \mid if\ X_i < constante\ goto\ l'\ else\ l"$$

Elles pourront également être utilisées pour les SRAM et les CRAM.

4 Machine à adressage indirect (SRAM)

Ces machines sont une extension des machines abstraites à compteurs qui est plus proche du langage machine usuel. La mémoire est toujours formée de registres, mais on dispose d'un ensemble d'instructions plus riche. Les différentes définitions des RAM contiennent essentiellement la possibilité de:

- copier le contenu d'un registre dans un autre,
- faire de l'adressage indirect: lire la valeur d'un registre comme une "adresse mémoire" c'est à dire un numéro d'un autre registre, et accéder à cet autre registre (en lecture et/ou en écriture),
- faire des opérations élémentaires sur les valeurs des registres

Dans ce modèle, Il n'y a pas de limite à la taille des mots ni à l'espace d'adresses mémoires : le nombre de registres est potentiellement infini et chacun peut contenir un entier naturel de taille arbitraire. Bien qu'un programme ne puisse traiter directement qu'un nombre fini de registres, l'adressage indirect permet l'accès à un nombre arbitraire de registres.

4.1 Synthèse

Ce modèle diffère de la machine à compteurs (MC) uniquement par le jeu d'instructions

$$\begin{aligned} SRAM - prog \quad ::= \quad & X_i := X_i + 1 \mid X_i := X_i - 1 \mid if\ X_i = 0\ goto\ l'\ else\ l" \\ \mid \quad & X_i := X_j \mid X_i :=< X_j > \ \mid \ < X_i >:= X_j \end{aligned}$$

4.2 Transitions

$p \vdash (l,\sigma)$	$\rightarrow$	$\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(i)+1 \\ j \neq i & \rightarrow \sigma(j) \end{cases}\right)$	si $I_l = X_i := X_i + 1$
$p \vdash (l,\sigma)$	$\rightarrow$	$\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(i)-1 \\ j \neq i & \rightarrow \sigma(j) \end{cases}\right)$	si $I_l = X_i := X_i - 1$ et $\sigma(i) \neq 0$
$p \vdash (l,\sigma)$	$\rightarrow$	$(l+1, \sigma)$	si $I_l = X_i := X_i - 1$ et $\sigma(i) = 0$
$p \vdash (l,\sigma)$	$\rightarrow$	(l', σ)	si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) = 0$
$p \vdash (l,\sigma)$	$\rightarrow$	$(l", \sigma)$	si $I_l = if\ X_i = 0\ goto\ l'\ else\ l"$ et $\sigma(i) \neq 0$
$p \vdash (l,\sigma)$	$\rightarrow$	$\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(\sigma(j)) \\ k \neq i & \rightarrow \sigma(k) \end{cases}\right)$	si $I_l = X_i :=< X_j >$
$p \vdash (l,\sigma)$	$\rightarrow$	$\left(l+1, \sigma' : \begin{cases} \sigma(i) & \rightarrow \sigma(j) \\ k \neq \sigma(i) & \rightarrow \sigma(k) \end{cases}\right)$	si $I_l =< X_i >:= X_j$
$p \vdash (l,\sigma)$	$\rightarrow$	$\left(l+1, \sigma' : \begin{cases} i & \rightarrow \sigma(j) \\ k \neq i & \rightarrow \sigma(k) \end{cases}\right)$	si $I_l = X_i := X_j$

4.3 Exemple

La machine qui vérifie si une liste d'entiers contient un 0 peut s'écrire comme suit. Initialement, X_0 contient la taille n de la liste, et les registres $X_1, \cdots, X_n$ contiennent les valeurs des éléments de la liste. Les registres de travail sont notés $R_1 \cdots, R_k$, pour $k \in \mathbb{N}$.

La machine renvoie 1 (sur le registre X_0) si la liste $X_1, \cdots, X_n$ contient un 0, et 0 sinon.

$$
\begin{aligned}
P \quad = \quad & 1 : R_1 := X_0, \\
& 2 : if\ R_1 = 0\ goto\ 7\ else\ 3, \\
& 3 : R_2 :=< R_1 >, \\
& 4 : if\ R_2 = 0\ goto\ 9\ else\ 5, \\
& 5 : R_1 := R_1 - 1, \\
& 6 : goto\ 2, \\
& 7 : X_0 := 0, \\
& 8 : goto\ 10, \\
& 9 : X_0 := 1, \\
& 10 :
\end{aligned}
$$

5 Machine abstraite parallèle (PRAM)

Il s'agit d'un modèle abstrait de calcul massivement parallèle pour le développement d'algorithmes. Une machine parallèle est constituée d'un nombre arbitrairement grand, non borné, de processeurs, qui effectue des calculs en parallèle. Chaque processeur dispose d'une mémoire locale, isolée de celle des autres processeurs, et d'un PID (identifiant unique, qui lui est propre, et auquel il a accès). Cette mémoire locale est accessible par adressage direct ou indirect. En outre, la communication entre les différents processeurs est assurée par le biais d'un mémoire globale, partagée en lecture et en écriture entre tous les processeurs. Là encore, la mémoire partagée est accessible en adressage direct ou indirect.

On dispose d'un unique programme pour tous les processeurs, c'est un modèle synchrone : tous les processeurs passent d'une instruction à la suivante au même moment, les horloges sont synchronisées. L'exécution du même programme est différenciée pour chaque processeur grâce à la valeur de son PID: les branchements conditionnels liées à ce PID induiront des sauts différents pour chaque processeur.

Les modèles de PRAM sont basés sur les différents mode de lecture/écriture par plusieurs processeurs sur le même registre de la mémoire globale : EREW, CREW ou CRCW selon que la lecture (Read) ou

l'écriture (Write) est Exclusive ou Concurrente. Dans le cas d'écriture concurrente en mémoire globale du modèle commun, on convient que tous les processeurs écrivent la même chose. Pour comprendre la puissance de calcul d'une PRAM, il suffit de savoir qu'une CRCW calcul n'importe quelle fonction booléenne en 4 instructions.

On détaille ici les CRAM. Une CRAM est une suite $\{R_i\}$, $i \in \mathbb{N}$, de machines RAM. Chaque machine R_k a sa propre mémoire locale: un ensemble infini de registres X_i^k, $i \in \mathbb{N}$, pouvant contenir un entier naturel. La mémoire globale un ensemble infini de registres G_i, $i \in \mathbb{N}$, accessibles simultanément en lecture/écriture par différents processeur. En cas de tentative d'écriture simultanée d'une valeur différente par plusieurs processeurs, la résolution du conflit se fait par une gestion de priorité: seul le processeur de PID le plus petit réalise l'écriture. Les autres continuent leur exécution normalement, ils ne sont pas bloqués.

5.1 Synthèse

- $CRAM - data = \mathbb{N}$
- $CRAM - store = \{\sigma | \sigma : \mathbb{N} \times \mathbb{N} \to \mathbb{N} \cup \{\bot\}\}$, où $\sigma(i,k)$ dénote le contenu du compteur X_i^k pour $i \in \mathbb{N}$ et $k > 0$, et $\sigma(i,0)$ dénote le contenu du compteur global G_i.
- $Readin(x) = \sigma : \begin{cases} (0,0) & \to x \\ (i,k) \neq (0,0) & \to \bot \end{cases}$ (entrée dans le registre G_0)
- $Readout(\sigma) = \sigma(0,0)$
- $$\begin{array}{rcl} CRAM - prog & ::= & X_i := X_i + 1 \mid X_i := X_i - 1 \mid if\ X_i = 0\ goto\ l'\ else\ l" \\ & \mid & X_i := X_j \mid X_i :=< X_j > \mid < X_i >:= X_j \\ & \mid & X_i := PID \mid X_i :=< X_j >^g \mid < X_i >^g := X_j \end{array}$$

  Lorsqu'on effectue une instruction sur un registre local X_i pour une machine R_k de la CRAM, la notation X_i désigne le registre X_i^k.

5.2 Transitions

Outre les transitions des SRAM, on a également les transitions suivantes:

$p \vdash (l,\sigma)$	$\to$	$\left(l+1, \sigma' : \begin{cases} (i,k) & \to k \\ (j,k), j \neq i & \to \sigma(j,k) \end{cases}\right)$	si $I_l = X_i := PID$
$p \vdash (l,\sigma)$	$\to$	$\left(l+1, \sigma' : \begin{cases} (i,k) & \to \sigma(\sigma(j,k),k) \\ (t,k), t \neq i & \to \sigma(t,k) \end{cases}\right)$	si $I_l = X_i :=< X_j >$
$p \vdash (l,\sigma)$	$\to$	$\left(l+1, \sigma' : \begin{cases} (i,k) & \to \sigma(\sigma(j,k),0) \\ (t,k), t \neq i & \to \sigma(t,k) \end{cases}\right)$	si $I_l = X_i :=< X_j >^g$
$p \vdash (l,\sigma)$	$\to$	$\left(l+1, \sigma' : \begin{cases} (\sigma(i,k),k) & \to \sigma(j,k) \\ (t,k), t \neq \sigma(i,k) & \to \sigma(t,k) \end{cases}\right)$	si $I_l =< X_i >:= X_j$
$p \vdash (l,\sigma)$	$\to$	$\left(l+1, \sigma' : \begin{cases} (\sigma(i,k),0) & \to \sigma(j,k) \\ (t,k), t \neq \sigma(i,k) & \to \sigma(t,k) \end{cases}\right)$	si $I_l =< X_i >^g := X_j$

5.3 Exemple

La machine qui recherche si une valeur e appartient à une liste d'entiers peut s'écrire comme suit. Initialement, la valeur à rechercher est dans le registre G_0, et les registres $G_1, \cdots, G_n$ contiennent les valeurs des éléments de la liste.

La machine renvoie 1 (sur le registre G_0) si la liste $G_1, \cdots, G_n$ contient la valeur recherchée, et 0 sinon.

$$
\begin{array}{rcl}
P & = & 1 : X_0 := G_0, \\
 & & 2 : G_0 := 0, \\
 & & 3 : X_1 := PID, \\
 & & 4 : X_2 :=< X_1 >^g, \\
 & & 5 : if\ X_2 = X_0\ goto\ 6\ else\ 7, \\
 & & 6 : G_0 := 1, \\
 & & 7 :
\end{array}
$$

6 Programmation par algèbre de fonctions

On change ici de présentation en calculant une fonction à l'aide d'une algèbre de fonctions $[\xi; OP]$. C'est à dire que l'on utilise des fonctions de base et des schémas pour définir d'autres fonctions : il faut les comprendre comme autant d'instructions de ce langage car chaque schéma de programmation exprime comment on construit une fonction, c'est un algorithme. Un exemple de fonction est la fonction constante zéro, notée 0, un exemple de schéma est la composition (Comp) à partir des fonctions $h, g_1, ..., g_m$ définie par

$$f(x_1, \cdots, x_n) = h(g_1(x_1, \cdots, x_n), \cdots, g_m(x_1, \cdots, x_n)).$$

On note alors $f = Comp(h, g_1, \cdots, g_m)$ et on dit que f est obtenu par composition à partir des fonctions $h, g_1, \cdots, g_m$.

Si on note $\mathcal{F}$ l'espace des fonctions, un opérateur est une fonctionelle $\mathcal{O} : \mathcal{F} \to \mathcal{F}$. Si ξ est un ensemble de fonctions et OP est un ensemble d'opérateurs, alors $[\xi; OP]$ dénote le plus petit ensemble de fonctions contenant ξ et clos sous les opérateurs de OP: c'est la clôture de ξ par OP.

6.1 Fonctions Primitives Récursives

Les fonctions primitives récursives, de $\mathbb{N}^k \to \mathbb{N}$ (vues en cours de calculabilité) sont définies comme l'algèbre de fonctions $PR = [0, \Pi, s; Comp, PR]$, où :

- 0 est la fonction constante 0,
- Π est l'ensemble de fonctions de projection π_i^k, où $\pi_i^k(x_1, \cdots, x_k) = x_i$,
- s est la fonction successeur: $s(x) = x + 1$,
- $Comp$ est le schéma de composition évoqué plus haut, et
- PR est le schéma de récursion primitive défini comme suit $f = PR(h, g)$ si et seulement si:

$$
\begin{array}{rcl}
f(0, x_1, \cdots, x_n) & = & g(x_1, \cdots, x_n) \\
f(s(x), x_1, \cdots, x_n) & = & h(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n))
\end{array}
$$

6.1.1 Exemples

On définit l'addition comme suit:

Soit $g = \pi_1^1$, soit $h = Comp(s, \pi_3^3)$, l'addition est alors $add = PR(h, g)$. Autrement dit, on a:

$$
\begin{array}{rcll}
add(0, y) & = & y & (= \pi_1^1(y)) \\
add(s(x), y) & = & s(add(x, y)) & (= s(\pi_3^3(x, y, add(x, y))))
\end{array}
$$

De la même manière, on peut définir la multiplication:

$$\begin{aligned} mul(0,y) &= 0 \\ mul(s(x),y) &= add(y, mul(x,y)) \end{aligned}$$

Et l'exponentielle y^x:

$$\begin{aligned} exp(0,y) &= s(0) \\ exp(s(x),y) &= mul(y, exp(x,y)) \end{aligned}$$

6.2 Fonctions Primitives Récursives sur les Notations

On considère ici des fonctions sur les mots finis d'un alphabet fini - usuellement les mots booléens.

Pour un alphabet fini $A = \{a_1, \cdots, a_l\}$, on prend les fonctions de base suivantes:

- la fonction constante ϵ (le mot vide)
- Π, l'ensemble de fonctions de projection π_i^k, où $\pi_i^k(x_1, \cdots, x_k) = x_i$,
- S l'ensemble des fonctions successeur sur les mots de l'alphabet: Pour $i = 1, \cdots, l$, $s_i(x) = a_i.x$, le mot obtenu en plaçant la lettre a_i en tête du mot x.

Les fonctions récursives primitives sur les notations sont alors l'ensemble $PR_{not} = [\epsilon, \Pi, S; Comp, PR_{not}]$, où PR_{not} est le schéma de récursion primitive sur les notations suivant:

$$\begin{aligned} f(\epsilon, x_1, \cdots, x_n) &= g(x_1, \cdots, x_n) \\ f(s_1(x), x_1, \cdots, x_n) &= h_1(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n)) \\ &\vdots \\ f(s_l(x), x_1, \cdots, x_n) &= h_l(x, x_1, \cdots, x_n, f(x, x_1, \cdots, x_n)) \end{aligned}$$

Modulo le codage des entiers par des mots sur un alphabet fini, l'ensemble des fonctions primitives récursives et l'ensemble des fonctions primitives récursives sur les notations sont identiques.

6.3 Fonctions Récursives - Fonctions Récursives sur les Notations

L'ensemble des fonctions récursives est obtenu en enrichissant les opérateurs avec le schéma de *minimisation* suivant, qui renvoie le plus petit entier y annulant f, s'il existe. S'il n'existe pas, la valeur renvoyée est $\perp$:

$$\mu(f)(x_1, \cdots, x_n) = \begin{cases} y & \text{si } f(y, x_1, \cdots, x_n) = 0 \text{ et } \forall z < y, f(z, x_1, \cdots, x_n) \neq 0 \\ \perp & \text{si } \forall y, f(y, x_1, \cdots, x_n) \neq 0 \end{cases}$$

La valeur de retour $\perp$ dénote un résultat indéterminé. Du point de vue de la sémantique d'exécution, $\perp$ dénote la non-terminaison du calcul. C'est donc une valeur absorbante pour le calcul: si une fonction a un de ses paramètres égal à $\perp$, son résultat est également $\perp$.

On note $REC : [0, \Pi, s; Comp, PR, \mu]$ l'ensemble des fonctions récursives.

Ce schéma est également applicable au cas des fonctions sur les mots d'un alphabet fini. En munissant l'alphabet $A = \{a_1, \cdots, a_l\}$ d'une relation d'ordre, et en prenant l'ordre lexicographique induit sur les mots finis de A, on obtient:

$$\mu_{not}(f)(x_1, \cdots, x_n) = \begin{cases} y & \text{si } f(y, x_1, \cdots, x_n) = \epsilon \text{ et } \forall z < y, f(z, x_1, \cdots, x_n) \neq \epsilon \\ \perp & \text{si } \forall y, f(y, x_1, \cdots, x_n) \neq \epsilon \end{cases}$$

L'ensemble des fonctions récursives sur les notations est alors $REC_{not} = [\epsilon, \Pi, S; Comp, PR_{not}, \mu_{not}]$. Ces deux ensembles de fonctions REC et REC_{not} sont là encore identiques, et vérifient la thèse de Church-Turing: ils coïncident avec l'ensemble des fonctions calculables par Machine de Turing, par Machines à Compteurs, Machines RAM, PRAM, etc.

7 Lambda-Calcul Pur

La notion de fonction est centrale en informatique, au point que l'on peut fonder un langage de programmation universel (au même sens que les machines de Turing) sur cette seule notion : c'est le λ-calcul pur.

7.1 Syntaxe

Etant donné un ensemble V de noms de variables (notées en minuscules), les *termes* du λ-calcul pur sont définis par

$$M, N ::= x \mid (MN) \mid \lambda x.M,$$

où $x \in V$. Le terme (MN) est appelé *application* d'une fonction M à un argument N, et le terme $\lambda x.M$, qui dénote la fonction qui associe le terme M à l'argument x, est appelé *abstraction*. Dans le cas de l'abstraction $\lambda x.M$, toutes les occurrences de la variable x dans le sous-terme M sont alors *liées* dans M.

A ce stade, il est important de noter que l'on ne différencie pas dans la syntaxe les notions de fonctions et d'argument: un même terme peut dénoter à la fois une fonction lorsqu'il est placé à gauche d'une application, et un argument lorsqu'il est passé à droite d'une application.

Notations : Afin d'alléger la notation, les parenthèses et les λ non-indispensables à la compréhension sont souvent omis. Dans ce cas, les conventions de notation sont les suivantes:

- Lorsque ce n'est pas ambigu, les parenthèses autour des applications sont omises. Par exemple, la notation MN dénote le terme (MN).
- La portée des abstractions s'étend aussi loin que possible (vers la droite). Par exemple, la notation $\lambda x.MN$ dénote le terme $\lambda x.(MN)$, et non pas le terme $((\lambda x.M)N)$.
- Pour les suites d'abstractions, la notation $\lambda x_1, x_2, \cdots, x_n.M$ dénote le terme $\lambda x_1.\lambda x_2.\cdots.\lambda x_n.M$.
- Pour les suites d'applications, la notation $MN_1N_2\cdots N_m$ dénote le terme $(\cdots((MN_1)N_2)\cdots N_m)$ lorsque $m \geq 1$, et simplement M lorsque $m = 0$.

Le troisième point de notation permet de dénoter des fonctions à n arguments, par le procédé de *Curryfication*: la fonction, qui à deux arguments x et y, associe le terme M, est ainsi notée $\lambda x, y.M$. Elle correspond en réalité au terme $\lambda x.\lambda y.M$, c'est à dire à la fonction, qui à un argument x associe une fonction: la fonction qui à un argument y associe M. Ce procédé de Curryfication est central dans les langages de programmation fonctionnels. Par exemple, en `Caml`, la fonction `map ('a -> 'b) -> 'a list -> 'b list` est définie dans la librairie `List` sous forme curryfiée. Ainsi, pour une fonction `f: 'a -> 'b`, l'application `map f` est de type `'a list -> 'b list`, et désigne la fonction, qui à une liste `l` d'éléments de type `'a`, associe la liste obtenue en applicant la fonction `f` à tous les éléments de `l`.

Exemples. Voici des termes duλ-calcul pur, appelés combinateurs:

$$\begin{array}{llll} I = & \lambda x.x & \Delta = & \lambda x.x\ \ x \\ K = & \lambda x, y.x & S = & \lambda x, y, z.x\ z\ (y\ z) \end{array}$$

On peut comprendre ces termes de la façon suivante. K représente une fonction qui prend un argument et retourne une fonction constante, et Δ reçoit un argument et l'applique à lui-même. Cet argument doit donc être à la fois une fonction et une donnée. Avoir des fonctions qui reçoivent des fonctions en argument n'est pas étranger à la pratique informatique : par exemple, un compilateur ou un système d'exploitation

reçoivent des programmes en argument. De même appliquer une fonction à elle-même est ce que peut faire un compilateur : on peut compiler un compilateur avec lui-même.

Lorsqu'on écrit un lambda-terme sous forme d'arbre syntaxique, les $\lambda x. \ldots$ sont des nœuds unaires, les applications sont des nœuds binaires, habituellement dénotés par le symbole @, et les variables correspondent aux feuilles. On peut alors ajouter à cet arbre syntaxique des arcs orientés. Pour chaque variable liée on dessine un arc allant au lambda qui la lie (le premier lambda en remontant vers la racine).

Avec les arcs lieurs on obtient une arbre appelé arbre du lambda-terme. Un exemple est donné Figure **??**.

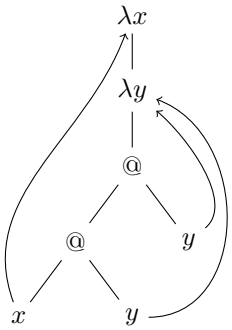

Figure 1: arbre du terme $\lambda xy.((x\ y)\ y)$

7.2 Substitution - α-conversion

Le terme $I = \lambda x.x$ dénote la fonction identité. Il en va de même pour le terme $\lambda y.y$. On considère donc l'égalité entre les termes qui diffèrent uniquement par le nom des variables utilisés dans les abstractions.

$$\lambda x.M =_\alpha \lambda y.M[y/x],$$

où $M[y/x]$ est la substitution dans le terme M de x par y, dont la définition suit.

Pour un terme T, on définit par induction structurelle sur T l'ensemble des variables *libres* $fv(T)$, et des variables *liées* $bv(T)$:

$$\begin{array}{llll} fv(x) & = x & bv(x) & = \emptyset \\ fv(MN) & = fv(M) \cup fv(N) & bv(MN) & = bv(M) \cup bv(N) \\ fv(\lambda x.M) & = fv(M) \setminus \{x\} & bv(\lambda x.M) & = bv(M) \cup \{x\} \end{array}$$

On note alors $M[N/x]$ la substitution dans M de la variable x par le terme N si $x \notin bv(N)$ et $fv(M) \cap bv(N) = \emptyset$, i.e.

$$\begin{array}{ll} x[N/x] & = N \\ y[N/x] & = y \text{ si } y \neq x \\ (MN)[P/x] & = (M[P/x]N[P/x]) \\ (\lambda y.M)[N/x] & = \lambda y.(M[N/x]) \end{array}$$

L'α-conversion d'un terme est alors donnée par la substitution d'un nom de variable liée à une abstraction, comme suit

$$\lambda x.M \to_\alpha \lambda y.M[y/x],$$

et l'α-équivalence $=_\alpha$ est la plus petite congruence (réflexive, symétrique, transitive) contenant $\to_\alpha$ et qui passe au contexte (si $P =_\alpha Q$ alors $\lambda x.P =_\alpha \lambda x.Q$, etc). Dans la suite on considère l'espace des termes du λ-calcul pur quotienté par $=_\alpha$. En pratique, on prendra soin d'utiliser l'α-conversion chaque fois que c'est possible pour donner des noms de variable différents à chaque abstraction.

Pour la représentation des λ-termes sous forme d'arbres: deux termes dont les arbres sont égaux sauf éventuellement pour le nom de leurs variables liées sont dits α-équivalents. On peut toujours renommer

n'importe quelle variable liée d'un terme ou d'un sous-terme pour obtenir un terme α-équivalent. Ceci permet d'obtenir un λ-terme équivalent, dont les abstractions portent sur des noms de variables tous différents. Dans ce cas, on peut omettre les arcs lieurs dans la représentation arborescente du terme.

Sur les arbres, la *substitution* d'une variable libre x par un terme N dans un terme M consiste en remplacer dans l'arbre de M chaque feuille x non liée par l'arbre de N.

7.3 β-réduction

La règle d'exécution du λ-calcul est la β-réduction, notée $\rightarrow_\beta$, définie comme suit:

$$(\lambda x.M)N \rightarrow_\beta M[N/x],$$

où x n'apparait pas comme variable liée dans un sous-terme de N. Evidemment, si x apparaît comme variable liée dans un sous-terme de N, l'α-conversion permet de se ramener au cas précédent.

On note $\rightarrow_\beta^c$ le passage au contexte de la règle $\rightarrow_\beta$: $M \rightarrow_\beta^c N$ si et seulement si il existe un sous-terme M' de M avec $M' \rightarrow_\beta N'$, et N est obtenu à partir de M en remplaçant le sous-terme M' par N'. On note alors $\rightarrow_\beta^*$ la clôture réflexive et transitive de $\rightarrow_\beta^c$, c'est-à-dire $M \rightarrow_\beta^* N$ si et seulement si il existe $n \in \mathbb{N}$, et $n+1$ termes $M_0, \cdots, M_n$ tels que $M = M_0 \rightarrow_\beta^c M_1 \rightarrow_\beta^c \cdots \rightarrow_\beta^c M_n = N$.

On appelle *redex* (reduction expression) et *contractum* les termes gauches et droit de la règle $\rightarrow_\beta$. A travers le passage au contexte de la β-reduction, un terme T peut donc contenir aucun, un seul ou plusieurs redex: chaque sous-terme de T de la forme $(\lambda x.M)N$ est un redex de T. Un terme T sans redex est appelé *forme normale*. Si en outre ce terme T est obtenu par β-réduction d'un terme M, on dit que T est la forme normale de M, et que M est (faiblement) *normalisant*.

Remarques

- Le λ-calcul est un calcul non-déterministe: à chaque étape de calcul, il peut exister plusieurs choix possibles: si le terme courant possède plusieurs redex, on peut β-réduire l'un ou l'autre de ces redex, et le résultat d'une étape de β-réduction peut être différent suivant le redex choisi.
- La β-réduction ne termine pas en général. Par exemple, le terme $\Omega = (\lambda x.x\ x)(\lambda x.x\ x)$ admet la suite (infinie) de β-réductions $\Omega \rightarrow_\beta \Omega \rightarrow_\beta \cdots \rightarrow_\beta \Omega$.

Un terme qui n'admet que des suites finies de β-réductions est dit *fortement normalisant*.

7.4 Normalisation, Confluence

Lorsqu'un terme est normalisant, on peut se poser la question de l'unicité de la forme normale. Cela revient à étudier la notion de confluence globale (voir Figure **??**) pour la relation $\rightarrow_\beta$:

$$\text{si } P \leftarrow_\beta^* M \rightarrow_\beta^* Q, \text{ alors il existe } N \text{ tel que } P \rightarrow_\beta^* N \leftarrow_\beta^* Q.$$

Lorsque P et Q sont obtenus par une seule étape de réduction, on parle de *confluence locale*, et lorsqu'en plus N est de plus obtenu en au plus une étape de réduction, on parle de *confluence forte*.

Remarque. La confluence locale ne permet pas d'obtenir a elle seule la confluence: On peut construire un terme normalisant se réduisant comme décrit à la Figure **??**. Il faut ajouter la terminaison pour éviter ce problème, comme l'énonce le lemme de Newman.

Lemme de Newman. Si une relation binaire sur un ensemble A est fortement normalisante (i.e. il n'y a pas de réduction infinie depuis un élément de A) et localement confluente, alors est globalement confluente.

L'exemple du terme Ω donné à la section précédente montre que le λ-calcul *n'est pas* fortement normalisant. Le lemme de Newman ne permet donc pas d'établir la confluence globale. Néanmoins, Taït et Löf ont démontré

La β-réduction est fortement confluente.

La conséquence immédiate est *l'unicité des formes normales*: Si P et Q sont deux formes normales d'un même terme M, alors $P = Q$.

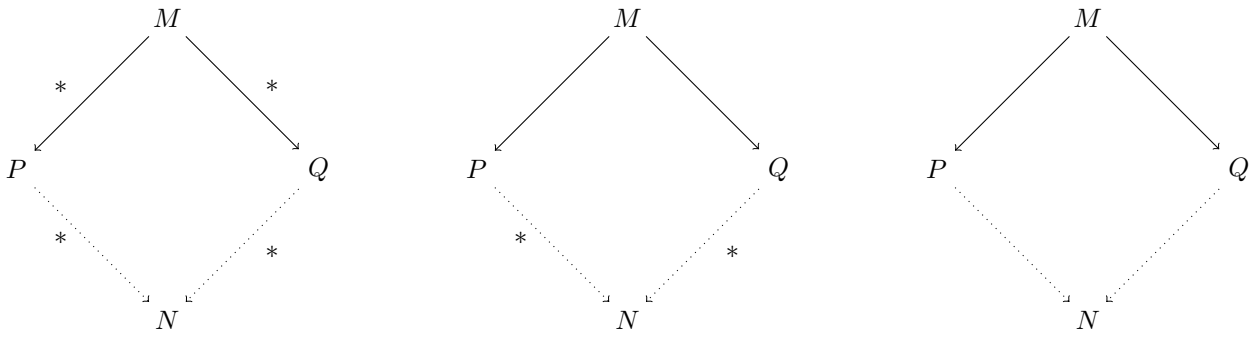

Figure 2: Confluence globale, confluence locale et confluence forte.

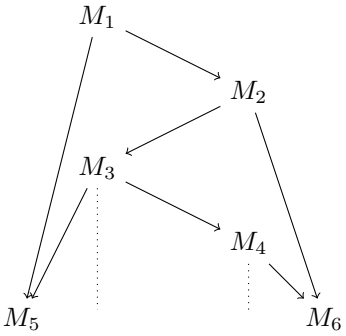

Figure 3: La confluence locale n'est pas suffisante à la confluence globale.

7.5 Stratégies de réduction

Comme on l'a vu précédemment, le λ-calcul est localement non-déterministe: un terme M donné peut contenir plusieurs redex, et il peut donc y avoir plusieurs choix possibles d'étapes de β-réduction. Ces choix possibles peuvent avoir une influence sur la terminaison ou non du calcul, et sur le nombre d'étapes de calcul nécéssaire à l'obtention de la forme normale (si elle existe).

Une *Stratégie d'évaluation* est un algorithme de choix du redex à réduire. On distingue principalement deux stratégies d'évaluation:

- Stratégie d'*appel par nom*, ou *externe gauche*. Le principe est de rechercher à chaque étape le redex le plus à gauche du terme à réduire - autrement dit le plus externe dans l'arbre syntactique du terme. Cette stratégie traite une application (MN) comme suit :

  1. elle réduit l'opérateur M en forme normale, et s'arrête si on a pas obtenu $\lambda x.M'$,
  2. ensuite, elle réduit $(\lambda x.M')N$

  Si un terme M admet une forme normale, celle-ci sera toujours obtenue par la stratégie d'appel par nom.

- Stratégie d'*appel par valeur*. Le principe cette fois est de rechercher la forme normale (la valeur - de la forme $\lambda x.t$) de l'argument d'une application avant de chercher à réduire l'application. Cette stratégie traite une application (MN) comme suit :

  1. elle réduit l'opérateur M en forme normale, et s'arrête si on a pas obtenu $\lambda x.M'$,

2. elle réduit ensuite l'argument N en forme normale N',
3. et enfin, elle réduit $(\lambda x.M')N'$

Malgré qu'elle échoue parfois à trouver la forme normale, cette stratégie est, en moyenne, plus efficace que l'appel par nom.

Exemples

1. Prenons le terme $(\lambda x.(x\ x))\ (\lambda y.y\ \lambda z.z)$.

   - En appel par nom:

$$\begin{aligned}(\lambda x.(x\ x))\ (\lambda y.y\ \lambda z.z) &\rightarrow (\lambda y.y\ \lambda z.z)\ (\lambda y.y\ \lambda z.z)\\ &\rightarrow \lambda z.z\ (\lambda y.y\ \lambda z.z)\\ &\rightarrow (\lambda y.y\ \lambda z.z)\\ &\rightarrow \lambda z.z\end{aligned}$$

   - En appel par valeur:

$$\begin{aligned}(\lambda x.(x\ x))\ (\lambda y.y\ \lambda z.z) &\rightarrow (\lambda x.(x\ x)\ \lambda z.z)\\ &\rightarrow \lambda z.z\ \lambda z.z\\ &\rightarrow \lambda z.z\end{aligned}$$

   Sur cet exemple, qui duplique son argument, la stratégie d'appel par valeur est plus efficace: on duplique uniquement la valeur, et l'argument est évalué une seule fois.

2. Prenons le terme $\lambda x.\lambda y.x\ z\ \Omega$.

   - En appel par nom:

$$\begin{aligned}\lambda x.\lambda y.x\ z\ \Omega &\rightarrow \lambda y.z\ \Omega\\ &\rightarrow z\end{aligned}$$

   - En appel par valeur:

$$\begin{aligned}\lambda x.\lambda y.x\ z\ \Omega &\rightarrow \lambda x.\lambda y.x\ z\ \Omega\\ &\vdots\\ &\rightarrow \lambda x.\lambda y.x\ z\ \Omega\ldots\end{aligned}$$

   Sur cet exemple, qui ignore l'un de ses arguments, la stratégie d'appel par valeur ne termine pas, alors que la stratégie d'appel par nom termine (et est donc plus efficace): l'argument problématique n'a pas besoin d'être évalué avant d'être ignoré.

La stratégie d'appel par valeur est utilisée par les langages `LISP`, `SCHEME` et `ML`. Le langage `HASKELL` utilise une variante de l'appel par nom dite d'appel par nécessité, que l'on nomme aussi évaluation paresseuse. Cette stratégie d'appel par nécéssité consiste à mémoriser dans une table les valeurs déjà évaluées, et à réduire en une étape un terme en sa valeur lorsqu'il est déjà présent dans la table.

7.6 Structures de données

Dans le λ-calcul pur, la notion de "donnée" n'existe pas en tant que telle: le langage n'étant pas typé, et on ne dispose pas de types de base `int`, `char`, etc. pour représenter les objets du calcul. Ces objets vont donc être représentés par des λ-termes particuliers, par le biais d'un encodage que l'on donne ici. Il est important de noter que cet encodage n'est pas unique, et qu'il existe de nombreuses manières équivalentes de représenter ces objets.

7.6.1 Nombres entiers

On utilise les entiers de Church. Un entier n est codé par le nombre de compositions d'une fonction f sur un argument x, comme suit.

- $\underline{0} = \lambda f.\lambda x.x$, et
- $\underline{n} = \lambda f.\lambda x.\ \underbrace{f \cdots f}_{n}\ x$.

On définit alors les fonctions arithmétiques usuelles

- `succ` $\underline{n} = \underline{n+1}$ par $\lambda n.\lambda f.\lambda x.f\ (n\ f\ x)$,
- `plus` $\underline{m}\ \underline{n} = \underline{m+n}$ par $\lambda m.\lambda n.\lambda f.\lambda x.m\ f\ (n\ f\ x)$
- `mult` $\underline{m}\ \underline{n} = \underline{m*n}$ par $\lambda m.\lambda n.\lambda f.m\ f\ (n\ f)$
- `sub`, `power`, ...

7.6.2 Booléens

On définit les booléens par la projection d'une paire sur sa première ou sa deuxième coordonnée:

- `true` $= \lambda x.\lambda y.x$, et
- `false` $= \lambda x.\lambda y.y$ (on remarque alors que `false` et $\underline{0}$ sont identiques).

On définit alors les fonctions booléennes usuelles

- `and` $= \lambda p.\lambda q.p\ q\ p$,
- `or` $= \lambda p.\lambda q.p$ `true` q,
- `not` $= \lambda p.p$ `false` `true`,
- `ifthenelse` $= \lambda p.\lambda a.\lambda b.p\ a\ b$, ...

et les prédicats sur les entiers (de Church)

- `iszero` $= \lambda n.n\ (\lambda x.$`false`$)$ `true`,
- `leq` $= \lambda m.\lambda n.$`izero` (`sub` $m\ n$), ...

7.6.3 Listes

On définit les listes de n éléments:

$$[M_1, \cdots, M_n] = \lambda f.\lambda x.(f\ M_1(f\ M_2(\cdots(f\ M_n\ x)\cdots)))$$

et les fonctions usuelles sur les listes, `cons`, `append`, `head`, `tail`, `length`, ...

7.7 Combinateurs de point fixe, Récursion

Définition. On note $=_\beta$ la clôture transitive, réflexive et symétrique de $\to_\beta$. On dit alors qu'un terme T est un *Combinateur de point fixe* si, pour tout terme M, on a $M\ (T\ M) =_\beta T\ M$. Notons qu'il existe de nombreux combinateurs de point fixe. On définit le *combinateur de point fixe de Curry*:

$$\mathtt{Y} = \lambda f.\,((\lambda x.(f(x\ x)))\ (\lambda x.(f(x\ x))))$$

Les combinateurs de point fixe sont utiles pour exprimer en λ-calcul les constructions de fonction par récurrence. Prenons par exemple la définition récursive primitive de l'addition suivante:

$$\begin{array}{lcl} \mathtt{add}\ \underline{m}\ \underline{n} & = & \underline{n} \text{ si } \underline{m} = \underline{0} \\ \mathtt{add}\ \underline{m}\ \underline{n} & = & \mathtt{succ}\ (\mathtt{add}\ (\mathtt{pred}\ \underline{m})\ \underline{n}) \text{ sinon} \end{array}$$

Cette définition *n'appartient pas* au langage du λ-calcul. Elle peut cependant s'exprimer en λ-calcul à l'aide d'un combinateur de point fixe: Soit `fix` un combinateur de point fixe. On définit alors l'addition récursive par le terme

$$\mathtt{add} = \mathtt{fix}\ F \text{ où } F \text{ est le terme } (\lambda f.\lambda m.\lambda n.\mathtt{ifthenelse}\ (\mathtt{ifzero}\ m)\ n\ (\mathtt{succ}\ (f\ (\mathtt{pred}\ m)\ n)))\,,$$

qui vérifie $\mathtt{add}\ \underline{m}\ \ \underline{n}\ \to_\beta^* \underline{m+n}$.

D'une manière similaire, on peut aussi utiliser un opérateur de point fixe pour exprimer co-inductivement le schéma de minimisation des fonctions récursives (Section 6.3).

Une fonction $\Phi : \mathbb{N}^k \to \mathbb{N}$ est *λ-définissable* s'il existe un terme T du lambda calcul pur tel que :

$$\forall n_1, \cdots, n_k \in \mathbb{N}\ T\ \underline{n_1}\ \cdots\ \underline{n_k} \to_\beta^* \underline{\Phi(n_1, \cdots, n_k)}.$$

L'utilisation des structures de données (entiers de Church, etc...) et des combinateurs de point fixe pour encoder la récursion primitive et la minimisation a ainsi permis à Kleene d'établir que les fonctions λ-définissables contiennent les fonctions récursives : le λ-calcul, comme les autres modèles vu jusqu'ici, vérifie ainsi (une direction de) la thèse de Church-Turing.

7.8 Machine Abstraite de Krivine

La *machine abstraite de Krivine (KAM)* est un algorithme, exprimé dans un formalisme de programmation impératif (machine à pile), qui implémente une stratégie d'évaluation du λ-calcul. Elle permet donc compléter la thèse de Church-Turing pour le λ-calcul pur, en donnant l'autre direction de l'équivalence. On donne cet algorithme pour les deux principales stratégies d'évaluation évoquées.

7.8.1 KAM par nom

La *machine abstraite de Krivine (KAM)* implémente l'appel par nom en λ-calcul. On définit 4 notions :

- termes: $T := x \mid (T\ T) \mid \lambda x.T$ (λ-termes) .
- environnement: $E := \emptyset \mid (x, C) \cup E$ (dictionnaire associant des clôtures à des noms de variables)
- clôture : $C := (T, E)$ (un environnement associé à un terme)
- pile : $\pi := \varepsilon \mid C\ ::\ \pi$ (pile de clôtures)

L'état de la machine est représenté par un triplet fait d'un terme, d'un environnement et d'une pile. On se donne un terme initial (un programme) avec un environnement et une pile vide. Une étape de réduction fait passer d'un état à l'état suivant en appliquant l'une des trois règles :

$$\frac{\begin{array}{cccc}(MN) & & e & \pi \end{array}}{\begin{array}{cccc} M & & e & (N,e) :: \pi \end{array}}\ \text{push}$$

$$\frac{x \qquad e \cup (x, (M, e')) \qquad \pi}{M \qquad e' \qquad \pi} \text{ deref}$$

$$\frac{\lambda x.M \qquad e \qquad c :: \pi}{M \qquad e \cup (x, c) \qquad \pi} \text{ pop}$$

La règle applicable dépend uniquement de la forme du λ-terme à réduire: application, abstraction, ou nom de variable. La KAM est donc déterministe.

Exemple: l'évaluation du terme $((\lambda x.x)\ y)$ sur la KAM par nom se déroule ainsi:

$$\cfrac{\cfrac{\cfrac{((\lambda x.x)\ y) \qquad \emptyset \qquad \varepsilon}{\lambda x.x \qquad \emptyset \qquad (y, \emptyset)}\text{ push}}{x \qquad (x, (y, \emptyset)) \qquad \varepsilon}\text{ pop}}{y \qquad \emptyset \qquad \varepsilon}\text{ deref}$$

Le résultat est y (plus aucune règle ne s'applique).

7.8.2 KAM par valeur

La *KAM par valeur* est alors définie en modifiant les piles de façon à ce qu'elles contiennent deux sortes d'éléments: les fonctions et les arguments. $\pi := \varepsilon \mid F(C) :: \pi \mid A(C) :: \pi$

Les règles sont alors, par ordre de priorité décroissante,

$$\frac{(MN) \qquad e \qquad \pi}{M \qquad e \qquad A(N, e) :: \pi} \text{ push}$$

$$\frac{\lambda x.M \qquad e \qquad A(N, e') :: \pi}{N \qquad e' \qquad F(\lambda x.M, e) :: \pi} \text{ swap}$$

$$\frac{x \qquad e \cup (x, (M, e')) \qquad \pi}{M \qquad e' \qquad \pi} \text{ deref}$$

$$\frac{v \qquad e \qquad F(\lambda x.M, e') :: \pi}{M \qquad e' \cup (x, (v, e)) \qquad \pi} \text{ pop}$$

Dans cette dernière règle, v désigne une valeur c'est à dire ici un lambda-terme qui n'est pas une application (donc une variable ou un lambda).

Par exemple, l'évaluation du terme $((\lambda x.x)\ y)$ sur la KAM par valeur se déroule ainsi:

$$\cfrac{\cfrac{\cfrac{\cfrac{((\lambda x.x)\ y) \qquad \emptyset \qquad \varepsilon}{\lambda x.x \qquad \emptyset \qquad A(y, \emptyset)}\text{ push}}{y \qquad \emptyset \qquad F(\lambda x.x, \emptyset)}\text{ swap}}{x \qquad (x, (y, \emptyset)) \qquad \varepsilon}\text{ pop}}{y \qquad \emptyset \qquad \varepsilon}\text{ deref}$$

Le résultat est y (plus aucune règle ne s'applique).

8 λ-calcul simplement typé

Le λ-calcul simplement typé est dérivé du λ-calcul pur par l'adjonction d'un système de typage inductif sur la structure syntaxique des termes.

Un *type simple* est, soit un élément d'un ensemble de types atomiques, soit une flèche entre deux types :

$$A := \alpha \mid A \to B$$

où α est un type atomique (ensemble choisi au départ).

Un lambda-terme t est typable lorsqu'on peut lui donner un type (ici parmi les types simples). Les lambda-termes typables sont définis par induction sur la syntaxe.

Intuitivement cela se fait comme ceci (mais nous allons voir qu'il y a un problème).

$$t \quad := \quad x^A \mid \overbrace{\lambda x^A.t^B}^{A\to B} \mid \overbrace{(u^{A\to B} v^A)}^{B} \tag{1}$$

Une variable est toujours typable, et on peut lui donner n'importe quel type. Si u est un lambda-terme typable auquel on peut donner le type $A \to B$ et si v est un lambda-terme typable auquel on peut donner le type A, alors $(u\ v)$ est un lambda-terme typable et on peut lui donner le type B. Enfin si t est un lambda-terme typable auquel on peut donner le type B et x est une variable à laquelle on a donné le type A *en typant* t, alors $\lambda x.t$ est typable, de type $A \to B$. Le point important ici, est que toutes les occurrences de la variable libre x doivent avoir ce même type A à l'intérieur de t. Mais comment l'exprimer correctement ?

Cela se fait en introduisant les notions de contexte de typage et de jugement de typage dans l'induction. Un contexte de typage, est un tableau associatif (un dictionnaire), dont les clés sont des variables et dont les valeurs sont des types simples. Un contexte de typage associe donc un unique type simple à chaque variable d'un ensemble fini donné de variables. On note traditionnellement $x : A$, l'association du type A à la variable x, Γ un contexte de typage et on utilise la virgule pour dénoter l'union disjointe de contextes. Ainsi $\Gamma, x : A$ signifie le contexte de typage formé en prenant un contexte Γ non défini pour la clé x et en l'étendant en donnant à la clé x la valeur A.

Un jugement de typage $\Gamma \vdash t : A$ permet d'associer un type A à un terme t dans un contexte Γ. On peut alors formaliser l'induction précédente en disant qu'un jugement de typage est valide s'il peut être dérivé grâce aux règles suivantes :

$$\frac{}{\Gamma, x : A \vdash x : A}\ \text{id}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ \text{abs.}$$

$$\frac{\Gamma \vdash u : A \to B \qquad \Gamma \vdash v : A}{\Gamma \vdash (u\ v) : B}\ \text{app.}$$

Le point important est que pour la règle d'application, u et v ont été typés en utilisant un même contexte, c'est à dire en donnant le même type aux variables de même nom. Si un jugement de typage est valide, le contexte de typage contient nécessairement toutes les variables libres du terme.

Ces règles définissent un système d'inférence similaire au calcul des séquents (pour la partie implicative de la logique propositionnelle intuinionniste). Ce système s'appelle deduction naturelle. Une inférence de typage est un arbre dont chaque nœud est une instance de règle, chaque feuille une instance d'axiome, de telle sorte que chaque arête mette en relation un même séquent à chaque extrémité. L'équivalence entre ce système de typage et la déduction naturelle signifie que les programmes typés du lambda-calcul sont la même chose que les preuves de la logiques constructive. Cette observation prend le nom de correspondence preuve-programmes, ou correspondance types-formules, ou encore isomorphisme de Curry-Howard.

Une autre façon pratique de voir le typage des lambda-termes est de ne considérer que des lambda-termes dans lesquels les variables liées sont choisies toutes différentes par α-renommage (on ne peut pas écrire $(\lambda x.x\ \lambda x.x)$, on écrit $(\lambda x.x\ \lambda y.y)$) et dans lesquels on a choisi pour chaque variable un type unique (on annote les occurrences de variables dans le terme avec leur type). Ce choix est l'équivalent du choix d'un contexte de typage étendu aux variables liées. Si le lambda-terme est typable dans un tel contexte, il a alors un type unique et on peut trouver son type en suivant la structure syntaxique du terme comme dans **??**.

Un lambda-terme pur peut ne correspondre à aucun terme typé comme il peut correspondre à plusieurs termes typés (il peut n'y avoir aucune façon de le décorer avec des types comme il peut y en avoir plusieurs). D'une façon générale, les termes contenant des sous termes de la forme (MM) sont non-typables: M ne peut être à la fois de type A et de type $A \to A$. Ainsi, le combinateur de point fixe de Curry Y est non-typable.

Les types sont conservés par β-conversion. (La substitutution est faite en utilisant un terme de même type que la variable substituée).

Il existe des termes non typables qui se réduisent en des termes typables.

Le λ-calcul simplement typé est fortement normalisant : n'importe quelle série de réductions amène à une forme normale (et cette forme normale est unique pour tous les chemins de réduction). Il n'est donc pas Turing complet.

9 Extensions du λ-calcul simplement typé

Unicité du typage. Nous utilisons désormais une version du lambda-calcul simplement typé où chaque variable liée par un lambda est annotée par un type, comme ceci : $\lambda x[T].t$, de telle sorte que T soit le type associé à la variable x au moment du jugement de typage :

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x[A].t : A \to B}\ \text{abs.}$$

De cette façon, étant donné un contexte de typage Γ, et un terme t ainsi annoté, il existe au plus un type A tel que $\Gamma \vdash t : A$.

Nous allons maintenant donner des *extensions du lambda-calcul simplement typé* de façon à en faire un langage de programmation, Turing-complet et ayant les caractéristiques d'un langage de type `ML` (les langages `ML` comme `SML`, `CAML` sont inspirés des extensions du lambda-calcul et en particulier du langage `PCF`, (*programming computable functions*).

9.1 Mise en séquence

`Unit`. On ajoute une valeur `unit` de type, `Unit`.

$$\begin{aligned} t &:= \ldots \mid \texttt{unit} \\ v &:= \ldots \mid \texttt{unit} \\ T &:= \ldots \mid \texttt{Unit} \end{aligned}$$

Et la règle de typage correspondante:

$$\frac{}{\Gamma \vdash \texttt{unit} : \texttt{Unit}}$$

Ceci nous permet d'introduire une *forme dérivée*:

$$t_1; t_2 \ \equiv\ (\lambda x[\texttt{Unit}].t_2\ \ t_1) \quad (x \notin fv(t_2)).$$

Cette mise en séquence, $t_1; t_2$ a pour effet d'évaluer t_1, de jeter son résultat, puis d'évaluer t_2. Les règles d'évaluation induites sur la forme sont :

$$\frac{t_1 \;\to\; t_1'}{t_1; t_2 \;\to\; t_1'; t_2}$$

$$\frac{}{\texttt{unit}; t_2 \;\to\; t_2}$$

Et la règle de typage induite sur la forme:

$$\frac{\Gamma \vdash t_1 : \texttt{Unit} \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash t_1; t_2 : B}$$

9.2 `Let in`

On étend le λ-calcul simplement typé avec la forme syntaxique `Let in`:

$$t := \ldots \mid \texttt{let}\ x = t\ \texttt{in}\ t$$

Les règles d'évaluation induites sur la forme sont :

$$\frac{}{\texttt{let}\ x = v_1\ \texttt{in}\ t_2 \;\to\; t_2[v_1/x]}$$

$$\frac{t_1 \;\to\; t_1'}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \;\to\; \texttt{let}\ x = t_1'\ \texttt{in}\ t_2}$$

Et la règle de typage induite sur la forme:

$$\frac{\Gamma \vdash t_1 : A \qquad \Gamma, x : A \vdash t_2 : B}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : B}$$

Au premier abord, cette forme $\texttt{let}\ x = t_1\ \texttt{in}\ t_2$ est juste du sucre syntaxique pour $(\lambda x[T_1].t_2\ t_1)$. Pour *désugariser* il faudra toutefois trouver l'annotation de type pour λx, donc faire l'inférence de type avant.

9.3 Récursion

On a vu que le combinateur de point fixe de Curry n'est pas simplement typable: la méthode utilisée pour exprimer la récursion primitive en λ-calcul pur ne s'applique donc pas au λ-calcul simplement typé. On étend le λ-calcul simplement typé comme suit: pour chaque type A on se donne un opérateur sur les termes `fix` tel que :

$$t := \ldots \mid (\texttt{fix}\ t)$$

Les règles d'évaluation induites sur la forme sont :

$$\frac{}{(\texttt{fix}\ \lambda x[A].t_2) \;\to\; t_2[(\texttt{fix}\ \lambda x[A].t_2)/x]}$$

$$\frac{t_1 \;\to\; t_1'}{\texttt{fix}\ t_1 \;\to\; \texttt{fix}\ t_1'}$$

Et la règle de typage induite sur la forme:

$$\frac{\Gamma \vdash t_1 : A \to A}{\Gamma \vdash (\texttt{fix}\ t_1) : A}$$

Forme dérivée `let rec`:

$$\texttt{letrec}\ x = t_1\ \texttt{in}\ t_2 \;\equiv\; \texttt{let}\ x = (\texttt{fix}\ \lambda x[T_1].t_1)\ \texttt{in}\ t_2$$

9.4 Types de données composées

On peut construire des types de données composées, le plus simple d'entre eux étant les *paires* :

$$\begin{array}{rcl} t & := & \ldots \mid \{t,t\} \mid t.1 \mid t.2 \\ v & := & \ldots \mid \{t,t\} \\ T & := & \ldots \mid T_1 \times T_2 \end{array}$$

$$i = 1,2 \ \frac{t \ \rightarrow \ t'}{t.i \ \rightarrow \ t'.i}$$

$$\frac{}{\{v_1,v_2\}.i \ \rightarrow \ v_i} \ i = 1,2$$

$$\frac{t_1 \ \rightarrow \ t_1'}{\{t_1,t_2\} \ \rightarrow \ \{t_1',t_2\}}$$

$$\frac{t_2 \ \rightarrow \ t_2'}{\{v_1,t_2\} \ \rightarrow \ \{v_1,t_2'\}}$$

Règles de typage des paires :

$$\frac{\Gamma \vdash t_1 : A \qquad \Gamma \vdash t_2 : B}{\Gamma \vdash \{t_1,t_2\} : A \times B}$$

$$\frac{\Gamma \vdash t : T_1 \times T_2}{\Gamma \vdash t.i : T_i}$$

9.5 Type Somme, Pattern Matching

On se donne un ensemble L de mots ou étiquettes. Un *type somme* est la donnée d'un dictionnaire fini dont les clés sont des mots et dont les valeurs sont des types: $\{\mathsf{mot}_1 : T_1, \ldots, \mathsf{mot}_n : T_n\}$. On étend les types avec les types sommes et les termes avec le pattern-matching.

$$\begin{array}{rcl} T & := \ldots & \mid \{\mathsf{mot}_1 : T_1, \ldots, \mathsf{mot}_n : T_n\} \\ t & := \ldots & \mid \texttt{match } t \texttt{ with} \\ & & \quad \texttt{case } \mathsf{mot}_1 \ x_1 \Rightarrow t_1 \\ & & \quad \vdots \\ & & \quad \texttt{case } \mathsf{mot}_n \ x_n \Rightarrow t_n \\ & & \mid \mathsf{mot}\ t \ \texttt{as} \ T \\ v & := \ldots & \mid \mathsf{mot}\ t \ \texttt{as} \ T \end{array}$$

Remarque : sans ce " `as` T " on perdrait l'unicité du type associé à un terme (on aurait par exemple Vrai `unit` de type $\{\mathsf{Vrai} : \texttt{Unit}\}$ mais aussi de type $\{\mathsf{Vrai} : \texttt{Unit}, \mathsf{Faux} : \texttt{Unit}\}$, ou de type $\{\mathsf{Vrai} : \texttt{Unit}, \mathsf{Saispas} : \texttt{Unit}\}$ etc.).

Règles de réduction.

$$\frac{t \ \rightarrow \ t'}{\texttt{match } t \texttt{ with...} \ \rightarrow \ \texttt{match } t' \texttt{ with...}}$$

$$\frac{t \;\to\; t'}{\texttt{mot}\; t \;\texttt{as}\; T \;\to\; \texttt{mot}\; t' \;\texttt{as}\; T}$$

$$\frac{}{\begin{array}{ll}\texttt{match mot}_i\; t \;\texttt{as}\; T \;\texttt{with} & \texttt{case mot}_1\; x_1 \Rightarrow t_1 \\ & \vdots \\ & \texttt{case mot}_n\; x_n \Rightarrow t_n \\ \to\; t_i[t/x_i] & \end{array}}$$

Règles de typage.

$$\frac{\Gamma \vdash t : \{\ldots, m_i : T_i, \ldots\} \qquad \Gamma, x_i : T_i \vdash t_i : T \quad (\forall i)}{\Gamma \vdash \texttt{match}\; t \;\texttt{as}\; \{\ldots, m_i : T_i, \ldots\} \;\texttt{with} \qquad \ldots\; \texttt{case}\; m_i\; x_i \Rightarrow t_i \ldots \quad : T}$$

$$\frac{\Gamma \vdash t_i : T_i}{\Gamma \vdash \texttt{m}_i\; t_i \;\texttt{as}\; \{\ldots, m_i : T_i, \ldots\} : \{\ldots, m_i : T_i, \ldots\}}$$

9.6 Types Polymorphes à la Milner

On a vu précédemment que le lambda-calcul simplement typé permet de donner des types différents à un même terme; il en va de même pour les extensions précédemment décrites. Par exemple, la fonction identité $\lambda x.x$ peut prendre le type $A \to A$, ou indifféremment le type $(A \to B) \to (A \to B)$, en fonction des choix réalisés au niveau des règles (id) de la dérivation de type. Ces types simples sont appelés *mono-types*.

Dans le cadre de ces mono-types, pour représenter la même fonction, appliquée sur des arguments de types différents, on est contraint d'utiliser différentes copies de cette fonction, chacune typée spécifiquement de façon à s'appliquer au type de l'argument qu'on lui passe; ainsi, pour la fonction identité, il faudrait une fonction identité de type `int` $\to$ `int` pour exprimer l'identité sur les entiers, une fonction identité de type `char` $\to$ `char`, etc... Pourtant, les dérivations de types sont identiques, modulo le choix de l'environnement initial utilisé dans les règles (id).

L'objectif du *typage polymorphe* est d'internaliser au niveau des types et des règles de typage ces différentes possibilités, de façon à obtenir un type le plus générique possible, qui puisse s'appliquer aux différents cas particuliers de types d'arguments utilisables. On présente ici le polymorphisme utilisé dans le langage `Caml` et ses dérivés.

On étend la grammaire des types comme suit:

$$\begin{array}{rrl} \text{Mono-types:} & A = & \alpha \mid A \to A \\ \text{Poly-types:} & T = & A \mid \forall \alpha.T \end{array}$$

On remarque que le quantificateur $\forall$ des poly-types ne peut apparaître qu'à l'extérieur des flèches $\to$. Tous comme pour les λ-termes, on considère les types polymorphes α-équivalent par renommage de leurs variables liées (aux quantificateurs $\forall$). Revenons à la fonction identité: dans un système polymorphe on donnera alors le type $\forall\alpha.\ \alpha \to \alpha$ à cette fonction. Ce type sera ensuite spécialisé, à chaque appel de la fonction, pour s'adapter au type de son argument. Ceci nous conduit à la notion de *relation de spécialisation* $\sqsubseteq$:

$$\forall\alpha.T \sqsubseteq T[C/\alpha] \quad \text{pour tout mono-type } C, \text{ où } \alpha \text{ est libre dans l'expression } T.$$

Ainsi, pour notre fonction identité, nous avons $\forall\alpha.\ \alpha \to \alpha \sqsubseteq$ `int` $\to$ `int`, pour dénoter la spécialisation de cette fonction aux arguments de type `int`, $\forall\alpha.\ \alpha \to \alpha \sqsubseteq$ `char` $\to$ `char`, pour les arguments de type `char`, etc.

Un type polymorphe peut également se spécialiser sur des arguments ayant eux-même un poly-type. Il faut dans ce cas être attentif à respecter la grammaire des types polymorphes, et éviter de mettre des quantificateurs à l'intérieur des flèches $\to$. Ainsi,

$$\forall\alpha.T \sqsubseteq T', \quad \text{où} \quad \begin{array}{l}\alpha \text{ est libre dans l'expression } T,\\ C \text{ est un poly-type, et}\\ T' \text{ est obtenu à partir de } T[C/\alpha] \text{ en déplaçant}\\ \text{tous les quantificateurs à l'extérieur des flèches } \rightarrow.\end{array}$$

Pour notre fonction identité, on peut alors l'appliquer à elle-même via α-équivalence et via la spécialisation suivante: $\forall\alpha.\ \alpha \rightarrow \alpha \sqsubseteq \forall\beta.\ \beta \rightarrow \beta$, où le type $\forall\beta.\ (\beta \rightarrow \beta)$ est obtenu à partir de $(\forall\beta.\ \beta \rightarrow \forall\beta.\beta)$ en déplaçant les deux quantificateurs $\forall\beta$ à l'extérieur de la flèche (et en les fusionnant puisqu'ils sont alors identiques). Cette spécialisation permet de donner deux poly-types syntaxiquement différents, (quoique α-équivalents), aux deux occurences de la fonction identité, et de typer ensuite l'application de la fonction identité à elle même.

Les règles de typage sont alors étendues avec les deux règles suivantes:

$$\frac{\Gamma \vdash t : T' \qquad T' \sqsubseteq T}{\Gamma \vdash t : T}\ \text{inst.}$$

$$\frac{\Gamma \vdash t : T \qquad \alpha \text{ n'est pas libre dans } \Gamma}{\Gamma \vdash t : \forall\alpha.T}\ \text{gen.}$$

Combinée avec la forme `let in`, qui oblige à typer un terme *avant* de l'appliquer, le polymorphisme permet d'appliquer une même fonction à des termes de types différents, comme dans l'expression:

$$\texttt{let}\ f = \lambda x.x\ \texttt{in}\ \{f\ v_1, f\ v_2\},$$

où v_1 et v_2 sont deux valeurs de type différents.

9.7 Propriétés du λ-calcul typé

Théorème de préservation. Si $\Gamma \vdash t : T$ et $t \ \rightarrow\ t'$, alors $\Gamma \vdash t' : T$. Ceci est valable pour le lambda-calcul simplement typé comme pour les extensions présentées ici. La réciproque est en générale fausse.

Théorème de progression. Si $\vdash t : T$ (notez le contexte vide), alors soit t est une valeur, soit il existe t' tel que $t \ \rightarrow\ t'$.

Type safety . Un langage est dit sûr au niveau du typage *type safe* lorsqu'il possède les deux propriétés précédentes (préservation et progression), éventuellement en élargissant la propriété de progression de façon à permettre au calcul de terminer sur des erreurs (des exceptions) aussi bien que sur des valeurs. Notez que la progression n'implique pas la terminaison.

Théorème de normalisation forte. En lambda-calcul simplement typé la β-réduction termine toujours (sans avoir besoin d'utiliser une stratégie particulière). Evidement ce résultat est faux pour le calcul étendu avec le `let rec`.