Lambda-calcul et langages fonctionnels

Jean Goubault-Larrecq

Abstract

Ceci est la version 2 de la première partie du cours de lambda-calcul, datant du 04 mars 2009. La première version datait du 02 août 1999. Bizarrement, je n'avais pas remarqué l'erreur présente depuis dix ans dans la démonstration du théorème de standardisation.

Un *langage fonctionnel* est juste un langage dans lequel la notion de *fonction* (procédure, sous-programme) est centrale. Cette définition vague recouvre les langages fonctionnels dits *purs*, dans lesquels tout calcul est effectué au moyen d'appels de fonctions (Miranda, Haskell, par exemple); et les langages fonctionnels *impératifs* (Lisp, ML), dans lesquels il est aussi permis d'effectuer des effets de bord (affectations), comme en Pascal ou en C.

Pour prendre un exemple, en OCaml, on peut définir la factorielle par:

```
let rec fact n = if n=0 then 1 else n*fact (n-1);;
```

et calculer la factorielle de 10 en demandant:

```
fact 10;;
```

ce à quoi le système répond `3628800`. On a donc défini la fonction `fact` par une définition proche de la notion mathématique usuelle:

$$n! \hat{=} \begin{cases} 1 & \text{si } n = 0 \\ n \times (n-1)! & \text{sinon} \end{cases}$$

où $\hat{=}$ dénote l'égalité par définition.

Les ingrédients de cette définition sont, d'abord, l'absence d'affectation à des variables, contrairement au programme C typique réalisant la même fonction:

```
int fact (int n)
{
  int res, i;

  for (res = 1, i = 1; i<=n; i++)
    res *= i;
  return res;
}
```

ensuite, la définition de `fact` par une expression utilisant `fact` elle-même — ce qu'on appelle une définition *récursive*. On pourrait d'ailleurs faire la même chose en C:

```
int fact (int n)
{
  if (n==0)
    return 1;
  return n * fact (n-1);
}
```

ce qui montre que C est lui-même un langage fonctionnel, selon notre définition.

Le point intéressant dans ces petits exemples, et que nous développerons, est que la notion de *fonction* est une notion importante en programmation. En fait, elle est tellement fondamentale qu'on peut fonder un langage de programmation universel (au sens où une machine de Turing est universelle) sur la seule notion de fonction: c'est le λ-*calcul pur*, que nous allons étudier dans ce chapitre. Le λ-calcul est aujourd'hui un outil central en informatique et en logique: en fait, la sémantique des langages de programmation, la formalisation des logiques d'ordre supérieur ou de la déduction naturelle bénéficient directement des concepts inhérents au λ-calcul.

Nous examinons principalement le λ-calcul pur dans ce chapitre. Nous en donnons la définition en section 1, prouvons certaines de ses propriétés les plus importantes en section 2, puis nous dégageons les quelques stratégies de réduction les plus caractéristiques des langages de programmation actuels en section 3. Nous définissons ensuite des modèles mathématiques des programmes du λ-calcul (les λ-*termes*) en section 4, ce qui nous permettra de raisonner sémantiquement sur les programmes. Finalement, nous introduirons la notion de *continuation* en section 5, en profiterons pour faire le lien entre sémantiques dénotationnelles en style de passage de continuations et stratégies de réduction.

1 Introduction au λ-calcul

Le λ-calcul a été inventé par le logicien américain Alonzo Church dans les années 1930, dans l'espoir de fournir un fondement au mathématiques plus simple que la théorie des ensembles, et fondé sur la notion de fonction. Ce programme a échoué, car le λ-calcul a un pouvoir d'expression beaucoup plus faible; en revanche, le λ-calcul a exactement le même pouvoir d'expression que les machines de Turing par exemple, ce qui en fait un bon choix pour fonder la notion de *fonction calculable*. La bible du λ-calcul est le livre de Barendregt [Bar84].

1.1 Syntaxe

Les trois constructions principales du λ-calcul sont:

- la *variable* (il faut bien commencer quelque part): nous les noterons x, y, z, etc.
- l'*application*: si u et v sont deux programmes, on peut considérer u comme une fonction et v comme un argument possible, et former l'*application* uv. Ceci correspond à la notation mathématique usuelle $u(v)$ tout en nous économisant quelques parenthèses. On remarquera qu'il n'y a pas de contraintes de typage ici, et qu'il est donc tout à fait légal de former des *auto-applications* xx, par exemple.
- l'*abstraction*: si u est un programme dépendant (ou non) de la variable x, alors on peut former un nouveau programme $\lambda x \cdot u$, qui représente la fonction qui à x associe u.

Par exemple, $\lambda x \cdot x + 1$ est intuitivement la fonction qui à x associe $x + 1$ — sauf que $+$ et 1 ne font pas partie du vocabulaire décrit ci-dessus, un point sur lequel nous reviendrons.

Formellement, fixons-nous un ensemble $\mathcal{V}$ infini dénombrable dit de variables. On définit l'ensemble Λ des λ-*termes* s, t, u, v, $\ldots$, comme étant le plus petit tel que $\mathcal{V} \subseteq \Lambda$, si u et v sont dans Λ alors uv est dans Λ, et si $x \in \mathcal{V}$ et u est dans Λ alors $\lambda x \cdot u$ est dans Λ. Ou, sous forme de déclaration de grammaire:

$$\Lambda ::= \mathcal{V} \mid \Lambda\Lambda \mid \lambda\mathcal{V} \cdot \Lambda$$

On autorisera l'usage de parenthèses pour désambigüer les expressions. D'autre part, les abstractions s'étendent aussi loin qu'il leur est permis, ce qui signifie en particulier que $\lambda x \cdot uv$ dénote $\lambda x \cdot (uv)$ et non $(\lambda x \cdot u)v$. La notation $uu_1 \ldots u_n$ dénotera u si $n = 0$, et sinon $(\ldots((uu_1)u_2)\ldots u_n)$, quant à la notation $\lambda x_1, \ldots, x_n \cdot u$, elle abrégera $\lambda x_1 \cdot \ldots \cdot \lambda x_n \cdot u$.

L'intérêt de ces deux dernières notations est que, bien que toutes les fonctions du λ-calcul soient *unaires* (ne prennent qu'un argument), on peut simuler les fonctions d'arité quelconque par *currification*: la fonction $\lambda x \cdot \lambda y \cdot x + y$, par exemple, est la fonction qui prend un x en argument, et retourne une fonction qui prend un y en argument et retourne $x + y$; autrement dit, c'est la fonction qui prend x, y et retourne leur somme.

1.2 Calculs et réduction

Il s'agit maintenant de donner un sens à ces λ-termes. Une première observation simple est que nous voudrions que $\lambda x \cdot x + 1$ et $\lambda y \cdot y + 1$ dénotent la même fonction: que l'argument s'appelle x ou y, c'est toujours la fonction qui ajoute 1 à son argument. On va donc vouloir confondre les deux termes. Supposons pour cela que nous sachions ce que veut dire remplacer une variable x par un terme v dans un terme u, et notons $u[x := v]$ le résultat. L'égalité entre deux termes dont seuls les noms des variables d'entrée diffèrent s'exprime alors par la règle suivante, dite d'*α-renommage*:

$$(\alpha) \qquad \lambda x \cdot u \quad \alpha \quad \lambda y \cdot (u[x := y])$$

ce qui définit une relation binaire α. Notons $=_\alpha$ la plus petite congruence contenant α, autrement dit la plus petite relation telle que $u\alpha v$ implique $u =_\alpha v$, $=_\alpha$ est réflexive, symétrique, transitive et passe au contexte: $u_1 =_\alpha u_2$ et $v_1 =_\alpha v_2$ impliquent $u_1 v_1 =_\alpha u_2 v_2$, et $u =_\alpha v$ implique $\lambda x \cdot u =_\alpha \lambda x \cdot v$.

Pour plus de simplicité, nous quotientons implicitement Λ par $=_\alpha$, de sorte que $\lambda x \cdot x + 1$ et $\lambda y \cdot y + 1$ seront en fait vus comme deux termes *identiques*.

Sémantiquement, la règle importante est la *β-réduction*:

$$(\beta) \qquad (\lambda x \cdot u)v \quad \beta \quad u[x := v]$$

qui exprime que si on applique la fonction qui à x associe u à l'argument v, on obtient la même chose que si on calcule directement u avec x remplacé par v. Par exemple, $(\lambda x \cdot x + 1)4 \;\beta\; 4 + 1$.

Nous noterons $\rightarrow_\beta$, ou plus souvent $\rightarrow$, la plus petite relation contenant β qui passe au contexte. Nous noterons $\rightarrow^*$ sa clôture réflexive transitive: $u \rightarrow^* v$ si et seulement s'il existe $n \geq 0$, et $n+1$ termes u_0, u_1, $\ldots$, u_n tels que $u = u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_n = v$. Nous noterons $\rightarrow^+$ la clôture transitive de $\rightarrow$: la définition est similaire, mais avec $n > 0$. Finalement, deux termes u et v sont *β-équivalents* si et seulement si $u =_\beta v$, où $=_\beta$ est la plus petite congruence contenant β.

Mais avant d'aller plus avant, il faut d'abord réaliser que la notion de remplacement, ou *substitution* $u[x := v]$ n'est pas aussi simple qu'elle y paraît. Par exemple, si vous utilisez la notion de remplacement textuel, fourni par la commande de recherche et de remplacement de n'importe quel éditeur de texte, vous allez obtenir une absurdité:

$$(\lambda x \cdot y)[x := uv] \quad = \quad \lambda uv \cdot y \quad (?)$$

mais $\lambda uv \cdot y$ n'est pas un terme légal, uv n'étant pas une variable. Le remède est simple: ne remplaçons pas les variables qui suivent un λ, que l'on considère comme des variables *liées*. Mais ça ne résout pas tout:

$$\begin{array}{rcll} (\lambda x \cdot x)[x := y] & = & \lambda x \cdot y & (?) \\ \lambda x \cdot y[y := x] & = & \lambda x \cdot x & (?) \end{array}$$

Dans le premier exemple ci-dessus, le fait de remplacer x par y dans la fonction identité $\lambda x \cdot x$ a produit la fonction constante qui retourne toujours y. Dans le deuxième exemple, c'est le contraire qui se produit.

Le problème dans le premier exemple, c'est que nous avons remplacé une occurrence de la variable x qui était liée par l'en-tête λx: nous devons donc exclure ce cas. Dans le second, c'est plus subtil: nous avons remplacé une variable non liée y par un terme qui contient une variable *libre* x (autrement dit, pas dans la portée d'un en-tête d'abstraction), qui si l'on remplace y par x, va se retrouver subrepticement liée par le λx au-dessus.

Pour éviter ces problèmes, nous conviendrons que les termes sont toujours préalablement α-renommés de sorte à éviter ces problèmes. Formellement (et nous revenons pour cela à la notion de termes hors α-renommage):

Définition 1 *L'ensemble des variables* libres $\mathrm{fv}(u)$ *et celui des variables* liées $\mathrm{bv}(u)$ *est défini par récurrence sur la structure de u par:*

$$\begin{array}{rcllrcl} \mathrm{fv}(x) & \hat{=} & \{x\} & & \mathrm{bv}(x) & \hat{=} & \emptyset \\ \mathrm{fv}(uv) & \hat{=} & \mathrm{fv}(u) \cup \mathrm{fv}(v) & & \mathrm{bv}(uv) & \hat{=} & \mathrm{bv}(u) \cup \mathrm{bv}(v) \\ \mathrm{fv}(\lambda x \cdot u) & \hat{=} & \mathrm{fv}(u) \setminus \{x\} & & \mathrm{bv}(\lambda x \cdot u) & \hat{=} & \mathrm{bv}(u) \cup \{x\} \end{array}$$

On dit que x est substituable *par v dans u si et seulement si $x \notin \mathrm{bv}(u)$ et $\mathrm{fv}(v) \cap \mathrm{bv}(u) = \emptyset$. Dans ce cas, on définit $u[x := v]$ par récurrence sur la structure de u:*

$$\begin{array}{rcl} x[x := v] & \hat{=} & v \\ y[x := v] & \hat{=} & y \quad (y \neq x) \\ (u_1 u_2)[x := v] & \hat{=} & (u_1[x := v])(u_2[x := v]) \\ (\lambda y \cdot u)[x := v] & \hat{=} & \lambda y \cdot (u[x := v]) \end{array}$$

La relation d'α-renommage est alors définie par:

$$(\alpha) \qquad \lambda x \cdot u \quad \alpha \quad \lambda y \cdot (u[x := y])$$

pour toute variable y telle que $x = y$, ou x est substituable par y dans u et y n'est pas libre dans u.

Exercice 1 *Montrer que, formellement, $\lambda x \cdot \lambda y \cdot xy \ \alpha \ \lambda y \cdot \lambda x \cdot yx$ est faux, mais que $\lambda x \cdot \lambda y \cdot xy =_\alpha \lambda y \cdot \lambda x \cdot yx$ est vrai.*

Formellement, on définit en fait $\rightarrow_\beta$ comme la plus petite relation contenant β *et* $=_\alpha$ qui passe au contexte, et de même pour $\rightarrow^*_\beta$, $\rightarrow^+_\beta$, $=_\beta$.

L'exercice suivant montre que l'on peut essentiellement faire comme s'il n'y avait aucun problème dans la définition de la substitution, pourvu que l'on s'arrange toujours pour remplacer u par un α-équivalent avant substitution, et que l'on raisonne toujours modulo α-équivalence. C'est ce que nous supposerons toujours dans la suite.

Exercice 2 *Montrer que pour tous u, x, v il existe un terme u' tel que $u =_\alpha u'$ et x est substituable par v dans u'. Montrer de plus que si u' et u'' sont deux termes vérifiant ces conditions, alors $u'[x := v] =_\alpha u''[x := v]$.*

Nous considérerons aussi de temps en temps la règle d'*η-contraction*:

$$(\eta) \qquad \lambda x \cdot ux \quad \eta \quad u \qquad (x \notin \mathrm{fv}(u))$$

qui exprime que, lorsque u ne dépend pas de x, la fonction qui à x associe u de x est exactement u elle-même. Par exemple, la fonction $\lambda x \cdot \mathit{len}\ x$, où *len* est la fonction longueur sur les listes, est *len* elle-même. Curieusement, (η) est indépendante des relations précédentes, et notamment $\lambda x \cdot ux \neq_\beta u$ lorsque $x \notin \mathrm{fv}(u)$ en général. On note $\rightarrow_{\beta\eta}$, $\rightarrow^*_{\beta\eta}$, $=_{\beta\eta}$ la plus petite relation contenant β, η et $=_\alpha$, sa clôture réflexive transitive, sa clôture transitive et la plus petite congruence la contenant respectivement.

Exercice 3 *Donner un exemple montrant que la condition $x \notin \mathrm{fv}(u)$ est nécessaire, intuitivement, dans la règle (η).*

Une occurrence d'un sous-terme de la forme $(\lambda x \cdot u)v$ dans un terme t est appelé un *β-rédex* dans t. (Ce mot vient de l'anglais "redex", abréviation de "reduction expression".) Dire que $t \rightarrow s$ signifie que s est obtenu à partir de t (modulo α-renommage) en remplaçant un rédex $(\lambda x \cdot u)v$ par son *contractum* $u[x := v]$.

Le but de l'exercice suivant est de montrer qu'un terme peut contenir zéro, un ou plusieurs rédex.

Exercice 4 *Montrer qu'il y a exactement deux termes v tels que $(\lambda x \cdot (\lambda y \cdot xy)x)z \rightarrow v$. Nommons-les v_1 et v_2. Montrer que, pour chaque $i \in \{1, 2\}$, il y a exactement un terme w_i tel que $v_i \rightarrow w_i$. Montrer que $w_1 = w_2$ et qu'il n'y a pas de terme t tel que $w_1 \rightarrow t$.*

On a donc le graphe de réductions*:*

$$\begin{tikzcd} & (\lambda x \cdot (\lambda y \cdot xy)x)z \arrow[dl] \arrow[dr] & \\ v_1 \arrow[dr] & & v_2 \arrow[dl] \\ & w_1 = w_2 & \end{tikzcd}$$

où v_1, v_2, w_1 et w_2 sont à trouver.

Exercice 5 *Dessiner le graphe de réductions de $(\lambda x \cdot (\lambda y \cdot y)x)z$, de $(\lambda f, x \cdot f(fx))(\lambda g, y \cdot gy)$, de $(\lambda x \cdot xx)(\lambda x \cdot xx)$.*

2 Théorèmes principaux, confluence, expressivité

La relation $\to_\beta$ (en abrégé, $\to$) est une *règle de calcul*, elle permet de simplifier des termes jusqu'à obtenir un terme qu'on ne peut plus simplifier. Un tel terme, sur lequel on ne peut pas appliquer la règle (β), est appelé une *forme normale*. Deux questions se posent immédiatement: 1. ce processus 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 ?

2.1 Terminaison

La première question a en fait une réponse négative:

Lemme 1 *La β-réduction ne termine pas en général.*

Preuve : Considérons $\Omega \hat{=} (\lambda x \cdot xx)(\lambda x \cdot xx)$ (on a le droit d'écrire une chose pareille, oui). Il ne contient qu'un rédex, lui-même. Le contracter redonne Ω, ce qui signifie que la seule réduction (maximale) à partir de Ω est la boucle:

$$\Omega \to \Omega \to \ldots \to \Omega \to \ldots$$

qui ne termine pas. ♢

Non seulement elle ne termine pas, mais en fait Ω n'a même pas de forme normale.

Si cette dernière phrase a l'air curieuse, c'est qu'il faut formaliser nos concepts:

Définition 2 *Un terme u est* fortement normalisant *si et seulement si toutes les réductions partant de u sont finies.*

Un terme u est faiblement normalisant *si et seulement si au moins une réduction partant de u est finie.*

Donc Ω n'est non seulement pas fortement normalisant, il n'est même pas faiblement normalisant.

Tous les termes fortement normalisants sont faiblement normalisants, mais le contraire n'est pas vrai:

Exercice 6 *Montrer que $(\lambda x \cdot y)\Omega$ est faiblement normalisant, mais pas fortement normalisant.*

Tant pis pour la terminaison, et en fait c'est normal: n'importe quel langage de programmation (sauf quelques rares exceptions) permet d'écrire des programmes qui bouclent, et ne terminent donc jamais.

2.2 Confluence

La deuxième question est alors: en supposant u faiblement normalisant — autrement dit, u a au moins une forme normale —, cette forme normale est-elle unique ? Cette question a, elle, une réponse affirmative. C'est une conséquence des résultats qui suivent: Si $u \to^* v_1$ et $u \to^* v_2$, alors il existe w tel que $v_1 \to^* w$ et $v_2 \to^* w$. Nous démontrerons ce résultat à la fin de cette section.

On utilisera pour écrire ce théorème la notation diagrammatique plus parlante:

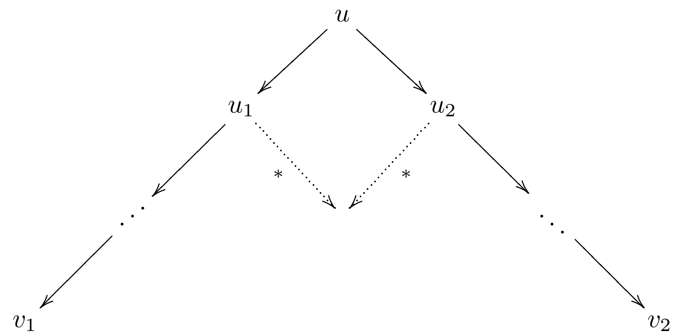
$$\begin{tikzcd}
 & u \arrow[dl, "*"'] \arrow[dr, "*"] & \\
v_1 \arrow[dr, dotted, "*"'] & & v_1 \arrow[dl, dotted, "*"] \\
 & w &
\end{tikzcd}$$

où les lignes pleines indiquent des réductions universellement quantifiées, et les lignes pointillées des réductions dont l'existence est affirmée. Les astérisques sur les flèches dénotent la clôture réflexive transitive.

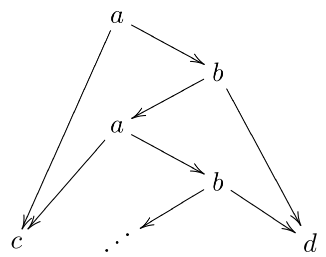
Une technique de preuve qui marche presque est la suivante. On vérifie par énumération exhaustive de tous les cas que:

$$\begin{tikzcd}
& u \arrow[dl] \arrow[dr] & \\
v_1 \arrow[dr, dotted, "*"'] & & v_1 \arrow[dl, dotted, "*"] \\
& w &
\end{tikzcd}$$

(Noter que u se réduit ici vers v_1 ou v_2 *en une étape.*) Puis on colle autant de ces petits diagrammes qu'il en faut pour obtenir:



mais ça ne marche pas: les petits diagrammes à mettre à l'intérieur du cône ci-dessus ne se recollent pas bien les uns aux autres. En fait, supposons qu'on ait quatre termes a, b, c, d tels que $a \to b$, $b \to a$, $a \to c$ et $b \to d$ et c et d sont normaux et distincts, alors on aurait $a \to c$ d'un côté, et $a \to^* d$ (en passant par b). Mais c et d étant normaux et distincts, il n'existe pas de w tel que $c \to^* w$ et $d \to^* w$, et le théorème serait faux. Pourtant, on peut vérifier qu'on a tous les "petits diagrammes" ci-dessus:



On peut réparer ce défaut lorsque la relation $\to$ termine... ce qui ne sera malheuresement pas le cas du λ-calcul:

Lemme 2 (Newman) *Soit $\to$ une relation binaire sur un ensemble A, et supposons que $\to$ est localement confluente et terminante. Alors $\to$ est confluente.*

Preuve : Comme $\to$ termine, on dispose du principe de *récurrence* le long de $\to$: pour démontrer une propriété P des éléments u de A, il suffit de la démontrer sous l'hypothèse (dite de *récurrence*) que $(*)$ $P(v)$ est vraie pour tout v tel que $u \to v$. (Il n'y a pas besoin de cas de base séparé: lorsque u est normal, $(*)$ est trivialement vérifiée, et l'on doit donc vérifier que $P(u)$ est vrai pour tout élément normal.)

En effet, supposons le contraire, c'est-à-dire qu'il existe une propriété P telle que pour tout u, si $(*)$ $P(v)$ est vraie pour tout v tel que $u \to v$, alors $P(u)$ est vraie; mais qu'il existe un élément u_0 tel que $P(u_0)$

soit fausse. On note qu'il existe nécessairement un u_1 tel que $u_0 \to u_1$ et tel que $P(u_1)$ soit faux: sinon, $(*)$ impliquerait que $P(u_0)$ soit vrai. De même, il existe un u_2 tel que $u_1 \to u_2$ et $P(u_2)$ est faux. Par récurrence sur k, on en déduit l'existence d'une réduction infinie $u_0 \to u_1 \to \ldots \to u_k \to \ldots$ où $P(u_k)$ est faux: contradiction.

Soit $u \to^* v_1$ et $u \to^* v_2$. Montrons que v_1 et v_2 ont un réduit commun, par récurrence sur u le long de $\to$. Si la longueur de réduction de u à v_1 vaut 0, c'est évident (prendre v_2), et de même si la longueur de réduction de u à v_2 vaut 0. Sinon, les deux réductions données se décomposent en $u \to u_1 \to^* v_1$ et $u \to u_2 \to^* v_2$ respectivement, et l'on a:

$$\begin{tikzcd}
& & u \arrow[dl] \arrow[dr] & & \\
& u_1 \arrow[dl, "*"'] \arrow[dr, dotted, "*"] & & u_2 \arrow[dl, dotted, "*"'] \arrow[dr, "*"] & \\
v_1 \arrow[dr, dotted, "*"'] & & v \arrow[dl, dotted, "*"'] & & v_2 \arrow[ddll, dotted, "*"] \\
& w_1 \arrow[dr, dotted, "*"'] & & & \\
& & w & &
\end{tikzcd}$$

où le losange du haut est par l'hypothèse de confluence locale, et les deux plus bas sont par hypothèse de récurrence, sur u_1 et u_2 respectivement. $\diamondsuit$

Il y différentes techniques pour montrer le théorème [Bar84]. Une de celles-ci est donnée à l'exercice 8.

Exercice 7 (Confluence forte) *Supposons que la relation binaire $\to$ soit* fortement confluente*, c'est-à-dire que si $u \to u_1$ et $u \to u_2$, alors u_1 et u_2 se réécrivent au même terme en* au plus une étape*. Montrer que $\to$ est confluente.*

Exercice 8 (Réductions parallèles) *On définit une nouvelle règle de réduction $\Rightarrow$ par:*

1. *$u \Rightarrow u$ pour tout terme u;*
2. *si $u \Rightarrow u'$ et $v \Rightarrow v'$, alors $uv \Rightarrow u'v'$;*
3. *si $u \Rightarrow u'$, alors $\lambda x \cdot u \Rightarrow \lambda x \cdot u'$;*
4. *si $u \Rightarrow u'$ et $v \Rightarrow v'$, alors $(\lambda x \cdot u)v \Rightarrow u'[x := v']$.*

Autrement dit, $\Rightarrow$ est la plus petite relation binaire vérifiant ces conditions et contenant le α-renommage. Vérifier que:

$$\begin{tikzcd}
& u \arrow[dl, Rightarrow] \arrow[dr, Rightarrow] & \\
u_1 \arrow[dr, Rightarrow] & & u_2 \arrow[dl, Rightarrow] \\
& v &
\end{tikzcd}$$

Montrer que $\to \subseteq \Rightarrow \subseteq \to^*$, et en déduire le théorème 1.

Pour démontrer le théorème 1, nous allons passer par le théorème dit des *développements finis*. Ceci introduira au passage une technique de démonstration de la terminaison qui formera la base de techniques plus avancées que l'on verra en partie 2.

Considérons pour ceci le calcul suivant, qu'on appellera le *λ^*-calcul*. Les λ^*-termes sont définis par la grammaire:

$$\Lambda^* ::= \mathcal{V} \mid \Lambda^*\Lambda^* \mid \lambda\mathcal{V} \cdot \Lambda^* \mid (\lambda^*\mathcal{V} \cdot \Lambda^*)\Lambda^*$$

de nouveau, à α-équivalence près. Il y a une construction supplémentaire par rapport au λ-calcul: la construction $(\lambda^* x \cdot u)v$. Attention: ceci n'est pas l'application d'un terme de la forme $\lambda^* x \cdot u$ à v: il n'y a tout simplement pas de terme de la forme $\lambda^* x \cdot t$ dans le langage. La notation peut prêter à confusion, donc, et il serait peut-être plus clair de noter la construction $(\lambda^* x \cdot u)v$ sous une forme plus proche de la notation OCaml **let** $x = v$ **in** u. Néanmoins, nous conservons la notation $(\lambda^* x \cdot u)v$ par commodité: la technique qui consiste à effacer les étoiles, que nous utiliserons ci-dessous, est bien plus simple à expliquer avec cette notation. Formellement, l'*effacement d'étoiles* est la fonction E définie par:

$$E(x) = x \quad E(uv) = E(u)E(v) \quad E(\lambda x \cdot u) = \lambda x \cdot E(u) \quad E((\lambda^* x \cdot u)v) = (\lambda x \cdot E(u))E(v)$$

Le λ^*-calcul n'a qu'une règle de réduction, la *β^*-réduction*:

$$(\beta^*) \qquad (\lambda^* x \cdot u)v \to u[x := v]$$

On notera que, du coup, $(\lambda x \cdot u)v$ n'est *jamais* un rédex en λ^*-calcul.

Lemme 3 (Développements finis) *La β^*-réduction termine.*

Preuve : Nous allons tenter de montrer que le λ^*-terme t est fortement normalisant par récurrence structurelle sur t. Ceci échoue lorsque t est de la forme $(\lambda^* x \cdot u)v$, il est possible que t se réduise en $(\lambda^* x \cdot u')v'$ avec $u \to^* u'$, $v \to^* v'$, puis en $u'[x := v']$, sur lequel on ne sait plus rien dire.

À la place, nous allons montrer que $t\sigma$ est fortement normalisant *pour toute substitution σ fortement normalisante*. Une *substitution* est une liste $[x_1 := t_1, \ldots, x_n := t_n]$ où les variables $x_1, \ldots, x_n$ sont deux à deux disjointes. L'application de la substitution à t est définie comme pour la substitution ordinaire: $x_i\sigma = t_i$ $(1 \leq i \leq n)$, $y\sigma = y$ pour toute variable hors de $\{x_1, \ldots, x_n\}$, $(st)\sigma = (s\sigma)(t\sigma)$, et $(\lambda x \cdot t)\sigma = \lambda x \cdot (t\sigma)$, où x n'est libre dans aucun t_i, et est différent de x_i, pour tout i, $1 \leq i \leq n$. On notera qu'il s'agit d'une substitution *parallèle*, et qu'en général, par exemple, $t[x_1 := t_1, x_2 := t_2] \neq t[x_1 := t_1][x_2 := t_2]$. La substitution σ est *fortement normalisante* si et seulement si tous les t_i le sont, $1 \leq i \leq n$.

On démontre que $t\sigma$ est fortement normalisant pour toute substitution fortement normalisante, par récurrence structurelle sur t. Lorsque t est une variable x_i, alors $t\sigma = t_i$ est fortement normalisant par hypothèse. Lorsque t est une autre variable y, $t\sigma = y$ est fortement normalisant. Lorsque t est une application uv, toute réduction infinie partant de $t\sigma = (u\sigma)(v\sigma)$ devrait effectuer une infinité de réductions à partir de $u\sigma$ ou à partir de $v\sigma$, puisque jamais aucune règle ne s'applique aux applications en λ^*-calcul (on se rappellera que la construction $(\lambda^* x \cdot u)v$ n'est *pas* une application). Mais, par hypothèse de récurrence, on ne peut faire qu'un nombre fini de réductions depuis $u\sigma$ et depuis $v\sigma$.

Finalement, lorsque t est de la forme $(\lambda^* x \cdot u)v$, toute réduction infinie à partir de $t\sigma$ est nécessairement de la forme $t\sigma = (\lambda^* x \cdot u\sigma)(v\sigma) \to^* (\lambda^* x \cdot u')v' \to u'[x := v'] \to \ldots$, où $u\sigma \to^* u'$ et $v\sigma \to^* v'$, et la réduction $(\lambda^* x \cdot u')v' \to u'[x := v']$ est la première qui utilise (β^*) au sommet du terme. Une telle réduction finit nécessairement par arriver, car $u\sigma$ et $v\sigma$ sont fortement normalisants par hypothèse de récurrence. Mais il est facile de voir que $u\sigma[x := v\sigma] = u\sigma'$ où, si σ s'écrit $[x_1 := t_1, \ldots, x_n := t_n]$, σ' est la substitution parallèle $[x_1 := t_1, \ldots, x_n := t_n, x := v\sigma]$. On démontre en effet que l'égalité tient dès que $x \notin \{x_1, \ldots, x_n\}$ et x n'est libre dans aucun t_i, $1 \leq i \leq n$, par récurrence structurelle sur u. Or, par hypothèse σ, donc $t_1, \ldots, t_n$ sont fortement normalisants; $v\sigma$ est fortement normalisant par hypothèse de récurrence; donc σ' est fortement normalisante, ce qui permet d'affirmer que $u\sigma'$ est fortement normalisant, de nouveau par hypothèse de récurrence. Or $u\sigma' = u\sigma[x := v\sigma] \to^* u'[x := v']$, et il existe une réduction infinie partant de $u'[x := v']$ (voir plus haut): contradiction. $\Diamond$

Il est facile, quoique pénible, de vérifier que la β^*-réduction est localement confluente. Par le lemme de Newman (lemma 2), tout λ^*-terme u a donc une unique forme normale $u\downarrow$. Remarquons que $u\downarrow$ ne peut pas contenir d'étoile: les étoiles apparaissent uniquement sur les β^*-rédex. Donc $u\downarrow$ est en fait un λ-terme. Définissons une relation $\rightarrow_1$ sur les λ-termes par: $u \rightarrow_1 v$ si et seulement s'il existe un λ^*-terme u' tel que $u = E(u')$ et $v = u'\downarrow$. Ceci revient à ajouter des étoiles sur certains rédex de u, puis à les réduire jusqu'à ce que tous aient été éliminés.

La relation $\rightarrow_1$ est fortement confluente (voir l'exercice 7). En effet, si $u \rightarrow_1 u_1$ et $u \rightarrow_1 u_2$, c'est qu'on peut rajouter des étoiles à certains rédex de u de sorte que le λ^*-terme obtenu se réduise en u_1, et des étoiles à certains rédex (en général différents) de u de sorte que le λ^*-terme obtenu se réduise en u_2. Pour fixer les idées, imaginons que l'on colorie les étoiles: les étoiles ajoutées dans le premier cas sont bleues, celles du second cas sont rouges. Il est facile de vérifier que, si l'on pose u' le terme où l'on a rajouté à la fois les étoiles bleues et les étoiles rouges, alors la réduction $u \rightarrow_1 u_1$ se relève en une réduction de u' vers un terme u'_1 qui est u_1, avec possiblement des étoiles rouges ici ou là; et que de même u' se réduit en λ^*-calcul à un terme u'_2 qui est u_2 avec des étoiles bleues ici ou là. Comme le λ^*-calcul est confluent et terminant, u'_1 et u'_2 ont la même forme normale pour (β^*), disons v: mais alors $u_1 \rightarrow_1 v$ et $u_2 \rightarrow_1 v$.

Théorème 1 (Confluence) *Si $u \rightarrow^* v_1$ et $u \rightarrow^* v_2$, alors il existe w tel que $v_1 \rightarrow^* w$ et $v_2 \rightarrow^* w$.*

Preuve : Comme $\rightarrow_1$ est fortement confluente, elle est confluente (exercice 7). Or $\rightarrow \subseteq \rightarrow_1 \subseteq \rightarrow^*$: si $u \rightarrow v$, alors il suffit de poser une étoile sur l'unique rédex contracté, ce qui montre que $u \rightarrow_1 v$; si $u \rightarrow_1 v$, alors $u' \rightarrow^* v$ en λ^*-calcul pour un certain u' tel que $E(u') = u$; mais l'effacement des étoiles s'étend aux réductions, montrant que $u \rightarrow^* v$.

Donc si $u \rightarrow^* v_1$ et $u \rightarrow^* v_2$, on a $u \rightarrow_1^* v_1$ et $u \rightarrow_1^* v_2$. Comme $\rightarrow_1$ est confluente, il existe un w tel que $v_1 \rightarrow_1^* w$ et $v_2 \rightarrow_1^* w$. On en déduit $v_1 (\rightarrow^*)^* w$, donc $v_1 \rightarrow^* w$, et de même $v_2 \rightarrow^* w$. ♢

Corollaire 1 (Unicité des formes normales) *Si u_1 et u_2 sont deux formes normales de u, alors $u_1 = u_2$.*

Preuve : Par le théorème 1, il existe w tel que $u_1 \rightarrow^* w$ et $u_2 \rightarrow^* w$. Comme u_i est normal ($i \in \{1, 2\}$), $u_i \rightarrow^* w$ implique $u_i = w$. Donc $u_1 = u_2$. ♢

Donc si l'on estime que la valeur d'un programme u est sa forme normale (par exemple, la valeur de $(\lambda x \cdot x + 1)4$ serait 5, en supposant que $4 + 1 \rightarrow^* 5$ et que 5 est normal), alors ce corollaire nous permet d'affirmer qu'en effet cette valeur est uniquement définie.

Exercice 9 *Démontrer que la relation $\rightarrow_1$ ci-dessus coïncide en fait avec la relation $\Rightarrow$ de l'exercice 8.*

2.3 Pouvoir expressif

La question que nous abordons maintenant est de savoir quelles fonctions on peut représenter en λ-calcul. Dans nos exemples, nous avons souvent utilisé des symboles qui ne faisaient pas partie du vocabulaire du λ-calcul pur, comme $+$, 1, 4 ou 5. Il y a deux façons de les intégrer dans le langage:

1. les ajouter au langage, et se donner quelques règles de réduction supplémentaires comme $4 + 1 \rightarrow 5$. Ceci demande à redémontrer la confluence, par exemple, mais est une façon de faire standard pour définir de vrais langages de programmation.

2. Ou bien les définir dans le langage, c'est-à-dire trouver des termes dans le λ-calcul pur qui aient le comportement attendu. Il s'agit de l'approche la plus simple d'un point de vue logique, et bien que n'étant pas d'une utilité informatique immédiate, elle apporte de nombreux éléments.

On va commencer par coder les entiers naturels dans le λ-calcul. Il y a plusieurs façons de le faire, mais la plus simple et la plus connue est d'utiliser les *entiers de Church*:

Définition 3 *Pour tout $n \in \mathbb{N}$, on pose $\lceil n \rceil \hat{=} \lambda f, x \cdot f^n x$, où $f^n t$ est défini par: $f^0 t \hat{=} t$, $f^{n+1} t \hat{=} f(f^n t)$.*

On pose d'autre part:

1. $S \hat{=} \lambda n, f, x \cdot f(nfx)$;

2. $\lceil + \rceil \hat{=} \lambda m, n \cdot mSn$;

3. $\lceil \times \rceil \hat{=} \lambda m, n, f \cdot m(nf)$;

4. $\lceil exp \rceil \hat{=} \lambda m, n \cdot mn$.

L'entier de Church $\lceil n \rceil$ est donc la fonctionnelle qui prend une fonction f et un argument x, et qui retourne f composée n fois appliquée à x.

On sait donc calculer sommes, produits et exponentielles:

Exercice 10 *Montrer que* $S\lceil n \rceil \to^* \lceil n+1 \rceil$, *autrement dit* S *représente la fonction successeur, qui à* n *associe* $n+1$. *De même, montrer que* $\lceil + \rceil \lceil m \rceil \lceil n \rceil \to^* \lceil m+n \rceil$, $\lceil \times \rceil \lceil m \rceil \lceil n \rceil \to^* \lceil mn \rceil$, $\lceil exp \rceil \lceil m \rceil \lceil n \rceil \to^* \lceil n^m \rceil$.

On dispose aussi des booléens. Pour les coder, l'idée c'est qu'un booléen sert à faire des tests if ... then ... else. Si on choisit $\mathbf{V} \hat{=} \lambda x, y \cdot x$ pour le vrai et $\mathbf{F} \hat{=} \lambda x, y \cdot y$, on voit que le test "if b then x else y" se représente juste par l'application bxy. En effet, "if $\mathbf{V}$ then x else y" est $\mathbf{V}xy$, qui se réduit en x, et "if $\mathbf{F}$ then x else y" est $\mathbf{F}xy$, qui se réduit en y.

On peut donc presque coder la factorielle, l'exemple du début de ce chapitre. Pour cela, il nous manque quand même quelques ingrédients, notamment le test d'égalité entre entiers et le calcul du prédécesseur $n-1$ d'un entier n. C'est le sujet des exercices qui suivent.

Exercice 11 *On code les couples* $\langle u, v \rangle$ *par la fonction* $\lambda z \cdot zuv$. *On définit les projections* $\pi_1 \hat{=} \lambda s \cdot s\mathbf{V}$, $\pi_2 \hat{=} \lambda s \cdot s\mathbf{F}$. *Montrer que* $\pi_1 \langle u, v \rangle \to^* u$ *et* $\pi_2 \langle u, v \rangle \to^* v$. *Montrer par contre qu'en général,* $\langle \pi_1 s, \pi_2 s \rangle \neq_\beta s$.

Exercice 12 (☡) *On veut définir une fonction prédecesseur, c'est-à-dire un* λ*-terme* P *tel que* $P(S\lceil n \rceil) \to^* \lceil n \rceil$ *pour tout entier* n. *Par convention, on posera que* $P(\lceil 0 \rceil) \to^* \lceil 0 \rceil$.

Montrer que $P \hat{=} \lambda k \cdot \pi_2(k(\lambda s \cdot \langle S(\pi_1 s), \pi_1 s \rangle)\langle \lceil 0 \rceil, \lceil 0 \rceil \rangle)$ *est une fonction prédecesseur acceptable (effectuer une récurrence sur* n*). Décrire intuitivement comment le prédécesseur est calculé étape par étape. Que pensez-vous de l'efficacité de l'algorithme ?*

Exercice 13 *On définit* $\lceil let \rceil\ x = u\ \lceil in \rceil\ v$ *par* $(\lambda x \cdot v)u$. *Montrer que* $\lceil let \rceil\ x = u\ \lceil in \rceil\ v \to v[x := u]$. *En déduire que* $\lceil let \rceil \ldots \lceil in \rceil \ldots$ *est un mécanisme de nommage (de* u *par la variable* x*) correct.*

Exercice 14 *On veut maintenant fabriquer une construction de pattern-matching "case* n *of* $0 \Rightarrow u \mid Sm \Rightarrow f(m)$*" qui calcule* u *si* $n = 0$, *et sinon calcule* $f(m)$, *où* m *est le prédécesseur de* n. *On va coder ceci par un terme* $\lceil intcase \rceil$ *tel que* $\lceil intcase \rceil\ \lceil n \rceil\ uf$ *représente le résultat désiré, autrement dit:*

$$\begin{array}{rcl} \lceil intcase \rceil\ \lceil 0 \rceil\ uf & \to^* & u \\ \lceil intcase \rceil\ \lceil n+1 \rceil\ uf & =_\beta & f\lceil n \rceil \end{array}$$

Montrer que l'on peut prendre $\lceil intcase \rceil \hat{=} \lambda k, x, f \cdot k(\lambda z \cdot f(Pk))x$. *Montrer qu'en général ce terme ne vérifie pas* $\lceil intcase \rceil\ \lceil n+1 \rceil\ uf \to^* f\lceil n \rceil$.

Exercice 15 *On va définir une fonction de soustraction partielle* $\dot{-}$ *par* $m \dot{-} n \hat{=} \max(m-n, 0)$. *Montrer que* $\lceil \dot{-} \rceil \hat{=} \lambda m, n \cdot nPm$ *en est une réalisation correcte.*

Exercice 16 *Montrer que* $\lceil zero? \rceil \hat{=} \lambda n \cdot n(\lambda z \cdot \mathbf{F})\mathbf{V}$ *est une fonction de test de nullité, autrement dit:*

$$\begin{array}{rcl} \lceil zero? \rceil\ \lceil 0 \rceil & \to^* & \mathbf{V} \\ \lceil zero? \rceil\ \lceil n+1 \rceil & \to^* & \mathbf{F} \end{array}$$

En déduire que $\lceil \leq \rceil \hat{=} \lambda m, n \cdot \lceil zero? \rceil(\lceil \dot{-} \rceil mn)$ *est un terme qui réalise le test* $\leq$ *d'inégalité.*

Exercice 17 *Définir* $\lceil\wedge\rceil$*, un* λ*-terme prenant deux booléens* b *et* b' *et retournant leur conjonction* $b \wedge b'$ *(le "et"). Faire la même chose pour le "ou"* $\vee$*, pour l'implication* $\Rightarrow$*, la négation* $\neg$*.*

Exercice 18 *Sachant que* $m = n$ *si et seulement si* $m \leq n$ *et* $n \leq m$*, déduire des exercices précédents un terme de test d'égalité entre entiers.*

Exercice 19 (B. Maurey) *Étant donnés deux termes* a *et* b *dans lesquels* x *n'est pas libre, et posant* $F \hat{=} \lambda f, g \cdot gf$*, on définit* $G_{a,b} \hat{=} \lambda n, m \cdot nF(\lambda x \cdot a)(mF(\lambda x \cdot b))$*. Montrer que:*

$$\begin{aligned} G_{a,b}\lceil 0\rceil\ \lceil m\rceil &\to^* a \\ G_{a,b}\lceil n+1\rceil\ \lceil m\rceil &=_\beta G_{b,a}\lceil m\rceil\ \lceil n\rceil \end{aligned}$$

En déduire que $G_{\mathbf{V},\mathbf{F}}$ *est une réalisation correcte de* $\leq$*, et* $G_{\mathbf{F},\mathbf{V}}$ *une réalisation correcte de* $>$*.*

Il nous manque enfin un ingrédient pour pouvoir coder la factorielle (et en fait n'importe quelle fonction calculable sur les entiers): la récursion. C'est un point délicat, et nous allons venir à la solution par étapes.

On veut définir $\lceil fact\rceil$ comme une fonction d'elle-même, autrement dit on veut:

$$\lceil fact\rceil =_\beta \lambda n \cdot \lceil if\rceil(\lceil zero?\rceil n)\ \lceil 1\rceil\ (\lceil\times\rceil n\ (\lceil fact\rceil(Pn)))$$

où $\lceil if\rceil\ b\ x\ y$ abrège bxy, dans un but de lisibilité. Il s'agit d'une équation à résoudre, et son côté droit est une fonction de l'inconnue $\lceil fact\rceil$. En fait, on peut même écrire le côté droit sous la forme $F\lceil fact\rceil$, où F est le λ-terme:

$$\lambda f \cdot \lambda n \cdot \lceil if\rceil(\lceil zero?\rceil n)\ \lceil 1\rceil\ (\lceil\times\rceil n\ (f(Pn)))$$

(On a renommé l'inconnue $\lceil fact\rceil$ par une variable f.)

Notre équation devient alors: $\lceil fact\rceil =_\beta F\lceil fact\rceil$. Ce genre d'équation est appelé une *équation de point fixe*, car si $\lceil fact\rceil$ en est une solution, elle est par définition un point fixe de la fonction représentée par le λ-terme F.

Le miracle du λ-calcul est que tous les termes ont des points fixes; encore mieux, ces points fixes sont définissables dans le λ-calcul lui-même:

Théorème 2 *Tout* λ*-terme* F *a un point fixe* x*, autrement dit un* λ*-terme* x *tel que* $x =_\beta Fx$*.*

En fait, il existe un λ*-terme* Y *sans variable libre tel que pour tout* F*,* YF *soit un point fixe de* F*. Un tel* Y *est appelé un* combinateur de point fixe*.*

Preuve : Ce théorème est surprenant, mais voyons comment on peut le démontrer, en utilisant le fait que nous connaissons déjà un terme qui boucle, à savoir $\Omega \hat{=} (\lambda x \cdot xx)(\lambda x \cdot xx)$.

On veut que $YF =_\beta F(YF)$, pour tout F, donc en particulier lorsque F est une variable. On peut donc définir Y sous la forme $\lambda f \cdot A(f)$, où $A(f)$ est un terme à trouver de variable libre f. Ce qui faisait boucler Ω, c'était une savante dose d'auto-application. On va réutiliser l'astuce, et essayer de trouver $A(f)$ sous la forme de l'application d'un terme $B(f)$ à lui-même. Autrement dit, on veut que $B(F)(B(F)) =_\beta F(B(F)(B(F)))$. Pour cela, on va supposer que le premier $B(F)$ à gauche est une abstraction $\lambda x \cdot u$, et que u est l'application de F à un terme inconnu, de sorte à obtenir le F en tête de $F(B(F)(B(F)))$. En somme, on suppose que $B(F) \hat{=} F(C(F,x))$, où $C(F,x)$ est un terme à trouver. En simplifiant juste le rédex de gauche, ceci se ramène à $F(C(F, B(F))) =_\beta F(B(F)(B(F)))$, ce qui sera réalisé dès que $C(F, B(F)) = B(F)(B(F))$. On peut par exemple poser $C(f,x) = xx$. En résumé, on a trouvé le combinateur de point fixe:

$$Y \hat{=} \lambda f \cdot (\lambda x \cdot f(xx))(\lambda x \cdot f(xx))$$

Ce combinateur s'appelle le *combinateur de point fixe de Church*. $\diamondsuit$

On notera que $YF =_\beta F(YF)$, mais il est faux que $YF \to^* F(YF)$. Il arrive que l'on souhaite cette dernière réduction. En fait, c'est réalisable: le combinateur Θ de point fixe de Turing a comme propriété que $\Theta F \to^* F(\Theta F)$.

On peut l'obtenir par un raisonnement similaire, c'est-à-dire en utilisant un choix d'arguments parmi: 1. si on veut $ab \rightarrow^* c$ et a est inconnu, alors on peut choisir a sous forme d'une abstraction; 2. si on veut $a \rightarrow^* f(b)$ et a est inconnu, où f est une variable, on peut choisir a de la forme $f(c)$, avec c à trouver tel que $c \rightarrow^* b$; 3. on peut toujours choisir un terme inconnu a de la forme bb, avec b à trouver.

Voici le raisonnement menant au combinateur de Turing. Ce qui faisait boucler Ω, c'était une savante dose d'auto-application. On va réutiliser l'astuce, et essayer de trouver Θ sous la forme de l'application d'un terme A à lui-même. On cherche donc A tel que $AAF \rightarrow^* F(AAF)$. Pour que AAF se réduise, quel que soit F, donc même lorsque F est une variable, il est raisonnable de penser que A va devoir s'écrire $\lambda g \cdot B(g)$ pour un certain terme $B(g)$ à trouver ayant g pour variable libre. Alors $AAF \rightarrow B(A)F$, qui doit se réduire en F appliqué à quelque chose. On va donc choisir $B(g) \hat{=} \lambda h \cdot h(C(g,h))$, où $C(g,h)$ est à trouver. Ce faisant, $AAF \rightarrow B(A)F = (\lambda h \cdot h(C(A,h)))F \rightarrow F(C(A,F))$, et une solution est alors de définir $C(A,F) \hat{=} AAF$, autrement dit $C(g,h) \hat{=} ggh$. En résumé, on obtient:

$$\Theta \hat{=} (\lambda g, h \cdot h(ggh))(\lambda g, h \cdot h(ggh))$$

La découverte de combinateurs de points fixes est un exercice de voltige, et si vous n'avez pas tout suivi, essayez simplement de vérifier que $YF =_\beta F(YF)$ pour tout F, et que $\Theta F \rightarrow^* F(\Theta F)$ pour tout F.

Il est aussi clair que l'on ne codera pas les fonctions récursives dans une réalisation informatique du λ-calcul par l'utilisation de Y ou de Θ tels que définis ci-dessus. Par contre, on supposera souvent qu'on a un *mécanisme* permettant de réécrire YF en $F(YF)$, pour Y un terme sans variable libre fixé à l'avance (typiquement une constante que l'on aura rajouté au langage). Ce mécanisme ne change rien aux propriétés du λ-calcul, car on peut le coder dans le λ-calcul lui-même: c'est tout l'intérêt des constructions du théorème 2.

Exercice 20 () *On code les listes de termes comme suit. La liste $u_1, \ldots, u_n$ est représentée par $[u_1, \ldots, u_n] \hat{=} \lambda f, x \cdot fu_1(fu_2 \ldots (fu_n x) \ldots)$. En particulier, la liste vide est $\lceil nil \rceil \hat{=} \lambda f, x \cdot x$, et la fonction $\lceil cons \rceil$ qui prend un élément u_0 et une liste $[u_1, \ldots, u_n]$ et retourne $[u_0, u_1, \ldots, u_n]$ est $\lceil cons \rceil \hat{=} \lambda a, \ell, f, x \cdot fa(\ell f x)$.*

Montrer que $\lceil hd \rceil \hat{=} \lambda \ell \cdot \ell(\lambda y, z \cdot y)\lceil nil \rceil$ est telle que:

$$\begin{aligned} \lceil hd \rceil(\lceil cons \rceil a \ell) &\rightarrow^* a \\ \lceil hd \rceil \, \lceil nil \rceil &\rightarrow^* \lceil nil \rceil \end{aligned}$$

et donc calcule le premier élément de la liste en argument, quand elle est non vide.

Montrer que $\lceil map \rceil \hat{=} \lambda g, \ell \cdot \ell(\lambda a, \ell' \cdot \lceil cons \rceil (ga)\ell')\lceil nil \rceil$ est telle que:

$$\lceil map \rceil \; g \; [u_1, \ldots, u_n] =_\beta [gu_1, \ldots, gu_n]$$

(On pourra utiliser le fait que si $\ell = [u_1, u_2, \ldots, u_n]$, alors $\ell hr =_\beta hu_1(\ell' hr)$, où $\ell' = [u_2, \ldots, u_n]$, et effectuer une récurrence sur n.)

Montrer que $\lceil tl \rceil \hat{=} \lambda \ell \cdot \pi_2(\ell(\lambda s, p \cdot \langle \lceil cons \rceil s(\pi_1 p), \pi_1 p \rangle)\langle \lceil nil \rceil, \lceil nil \rceil \rangle)$ envoie la liste $[u_1, u_2, \ldots, u_n]$ vers $[u_2, \ldots, u_n]$. (Ceci est analogue au codage du prédécesseur dans les entiers.)

Que fait $\lceil app \rceil \hat{=} \lambda \ell, \ell' \cdot \ell \lceil cons \rceil \ell'$?

3 Stratégies de réduction et langages de programmation

On a vu qu'un λ-terme pouvait contenir plusieurs rédex. Si l'on vu calculer la forme normale d'un terme u (si elle existe), on va pouvoir l'obtenir en choisissant un rédex dans u, en le contractant, et en répétant le processus jusqu'à ce qu'il n'y ait plus de rédex. Ce choix du rédex à chaque étape de réduction est appelé une *stratégie* de réduction.

3.1 Stratégies internes

Prenons le cas d'un langage de programmation usuel, comme OCaml ou Pascal ou C. Dans ces langages, un terme de la forme uv (noté $u(v)$ en Pascal ou en C) s'évalue typiquement en calculant d'abord la valeur f de u, qui doit être une fonction, puis en calculant la valeur a de v, puis en appliquant f à a. (En général, u est un identificateur qui dénote déjà une fonction, sans qu'on ait à calculer quelle fonction c'est exactement.) Remis dans un cadre λ-calculatoire, on peut dire qu'on a d'abord réduit les rédex de u, puis quand il n'y en a plus eu, on a réduit les rédex de v, enfin on regarde ce qu'est devenu u, et s'il est devenu un terme de la forme $\lambda x \cdot t$, on applique cette dernière fonction à v pour donner $t[x := v]$, et on continue.

Ce genre de stratégie est appelée une stratégie *interne* (innermost en anglais), parce qu'elle réduit d'abord les rédex les plus à l'intérieur des termes d'abord. La stratégie particulière présentée préfère de plus les rédex de gauche à ceux de droite: elle ne réduit les rédex de droite que quand il n'y en a plus à gauche. Il s'agit d'une stratégie *interne gauche*. Une stratégie *interne droite* est évidemment envisageable, et se rapprocherait de la sémantique de OCaml. Un petit test qui le montre est de taper:

```
let a = ref 0;;
let f x y = ();;
f (a:=1) (a:=2);;
a;;
```

qui crée une référence (une cellule de valeur modifiable par effet de bord), puis une fonction bidon `f` qui ignore ses deux arguments. Ensuite on calcule `f` appliquée à deux arguments qui affectent 1, resp. 2 à `a`. Le langage étant séquentiel, la valeur finale de `a` permettra de savoir lequel des arguments a été évalué en dernier. En OCaml, la réponse est 1, montrant que c'est l'argument de gauche qui a été évalué en dernier.

Évidemment, en λ-calcul on n'a pas d'affectations, et le fait qu'une stratégie interne soit gauche ou droite ne change rien au résultat final.

3.2 Stratégies externes

On peut aussi décider d'utiliser une stratégie *externe*, qui réduit les rédex les plus externes d'abord. Autrement dit, alors qu'une stratégie interne réduit $(\lambda x \cdot u)v$ en normalisant $\lambda x \cdot u$ vers $\lambda x \cdot u'$, en normalisant v vers v', puis en réduisant $u'[x := v']$, une stratégie externe commence par contracter $(\lambda x \cdot u)v$ en $u[x := v]$, qu'il réduit à son tour.

Vu au travers du filtre des langages de programmation usuels, voici ce que ces stratégies donnent pour l'évaluation de $fact(1 + 2)$, en supposant la définition de $fact$ donnée au début de ce chapitre, et les règles de réduction typiques de $+$, $\times$, etc. Pour se fixer les idées, on utilisera celles de $\lceil fact \rceil$, $\lceil + \rceil$, $\lceil \times \rceil$.

On commence par regarder la stratégie externe gauche. Comme $\lceil fact \rceil = Y(\lambda f \cdot \lambda n \cdot \lceil if \rceil(\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (f(Pn))))$, par une stratégie externe on aura $\lceil fact \rceil \to^* \lambda n \cdot \lceil if \rceil(\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (\lceil fact \rceil(Pn)))$. On obtient alors:

$$\begin{array}{rl} & \lceil fact \rceil(\lceil + \rceil\ \lceil 1 \rceil\ \lceil 2 \rceil) \\ \to^* & (\lambda n \cdot \lceil if \rceil(\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (\lceil fact \rceil(Pn))))(\lceil + \rceil\ \lceil 1 \rceil\ \lceil 2 \rceil) \\ \to & \lceil if \rceil(\lceil zero? \rceil(\lceil + \rceil\ \lceil 1 \rceil\ \lceil 2 \rceil))\ \lceil 1 \rceil\ (\lceil \times \rceil(\lceil + \rceil\ \lceil 1 \rceil\ \lceil 2 \rceil)\ (\lceil fact \rceil(P(\lceil + \rceil\ \lceil 1 \rceil\ \lceil 2 \rceil)))) \end{array}$$

mais ici on voit que l'argument $(\lceil + \rceil\ \lceil 1 \rceil\ \lceil 2 \rceil)$ a été copié en trois endroits. Chaque occurrence devra être réduite en $\lceil 3 \rceil$ indépendamment des autres; autrement dit, on va devoir calculer $1 + 2 = 3$ trois fois !

Pour voir ça plus clairement, reprenons une notation plus proche de la notation OCaml, et en particulier plus agréable. Nous réécrivons la réduction ci-dessus et la complétons:

$$
\begin{array}{rl}
 & \texttt{fact}\ (1+2) \\
\to^* & (\lambda n \cdot \texttt{if}\ n=0\ \texttt{then}\ 1\ \texttt{else}\ n \times \texttt{fact}\ (n-1))(1+2) \\
\to & \texttt{if}\ (1+2)=0\ \texttt{then}\ 1\ \texttt{else}\ (1+2) \times \texttt{fact}\ ((1+2)-1) \\
 & \ldots \text{où l'on passe quelques temps à réduire } (1+2)=0 \text{ en } \mathbf{F} \ldots \\
\to^* & \texttt{if}\ \mathbf{F}\ \texttt{then}\ 1\ \texttt{else}\ (1+2) \times \texttt{fact}\ ((1+2)-1) \\
\to & (1+2) \times \texttt{fact}\ ((1+2)-1) \\
\to^* & \ldots \text{où } 1+2 \text{ n'est toujours pas évalué à } 3 \ldots \\
 & \ldots \text{on réduit comme ça jusqu'à la fin de la récursion} \ldots \\
\to^* & (1+2) \times (((1+2)-1) \times ((((1+2)-1)-1) \times 1)) \\
\to^* & \ldots \text{et après un temps certain, on obtient le résultat final,} \\
\to^* & 6
\end{array}
$$

3.3 Stratégies internes faibles, appel par valeur

Maintenant, examinons ce qui se passe avec une stratégie interne. Nous allons commencer par supposer que la factorielle $\lceil fact \rceil$ s'évalue par la stratégie interne en $\lambda n \cdot \lceil if \rceil (\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (\lceil fact \rceil (Pn))$, et que ceci est normal. (Il y a un problème avec cette hypothèse, que nous résoudrons par la suite.) Alors le même calcul, par la stratégie interne, est beaucoup plus direct. Dans un souci de lisibilité, nous l'écrivons en notation pseudo-OCaml:

$$
\begin{array}{rl}
 & \texttt{fact}\ (1+2) \\
\to^* & (\lambda n \cdot \texttt{if}\ n=0\ \texttt{then}\ 1\ \texttt{else}\ n \times \texttt{fact}\ (n-1))(1+2) \\
\to^* & (\lambda n \cdot \texttt{if}\ n=0\ \texttt{then}\ 1\ \texttt{else}\ n \times \texttt{fact}\ (n-1))(3) \\
\to & \texttt{if}\ 3=0\ \texttt{then}\ 1\ \texttt{else}\ 3 \times \texttt{fact}\ (3-1)) \\
\to^* & \texttt{if}\ \mathbf{F}\ \texttt{then}\ 1\ \texttt{else}\ 3 \times \texttt{fact}\ (3-1)) \\
\to & 3 \times \texttt{fact}\ (3-1) \\
\to^* & 3 \times (\lambda n \cdot \texttt{if}\ n=0\ \texttt{then}\ 1\ \texttt{else}\ n \times \texttt{fact}\ (n-1))(2) \\
\to^* & \ldots \text{par le même procédé} \ldots \\
 & 3 \times (2 \times \texttt{fact}\ (2-1)) \\
\to^* & 3 \times (2 \times (1 \times 1)) \\
\to^* & 6
\end{array}
$$

ce qui correspond beaucoup plus à ce à quoi on s'attend de la part d'un langage de programmation.

Cependant, dans l'exemple ci-dessus, nous avons triché deux fois.

Première tricherie: Nous avons supposé que $\lceil fact \rceil$ s'évaluait par la stratégie interne en $\lambda n \cdot \lceil if \rceil (\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (\lceil fact \rceil (Pn))$, et que ceci était normal. Mais c'est faux ! En fait, $\lceil fact \rceil$ est de la forme YF, pour un certain F, et on a:

$$YF \to (\lambda x \cdot F(xx))(\lambda x \cdot F(xx)) \to F((\lambda x \cdot F(xx))(\lambda x \cdot F(xx))) \to F(F((\lambda x \cdot F(xx))(\lambda x \cdot F(xx)))) \to \ldots$$

qui ne termine jamais !

Le remède est que dans les langages de programmation usuels, on ne réduit pas sous les λ. Autrement dit, on considère que $\lambda x \cdot u$ est déjà suffisamment évalué. De telles stratégies s'appellent des *stratégies de réduction faible*. Elles ne sont évidemment pas suffisantes pour obtenir une forme normale — par exemple, aucune règle de réduction faible ne s'applique pour réduire $\lambda x \cdot (\lambda y \cdot y)x$. Mais ça n'est pas très important dans un langage de programmation; en un sens, par programme on veut calculer des entiers, des listes, des structures de données en somme. Si un programme retourne une fonction, c'est qu'il attend encore au moins un argument pour pouvoir continuer le calcul.

La stratégie de réduction interne faible peut maintenant se formaliser par les règles suivantes, où $u \triangleright v$ signifie "u se réduit en une étape en v" par cette stratégie:

$$\frac{}{(\lambda x \cdot u)V \rhd u[x := V]}\ (\beta\rhd) \quad \frac{u \rhd u'}{uv \rhd u'v}\ (App_1\rhd) \quad \frac{v \rhd v'}{uv \rhd uv'}\ (App_2\rhd)$$

L'argument V de l'abstraction dans la règle $(\beta\rhd)$ est contraint à être une *valeur*. On définit ici les valeurs V par récurrence sur la structure de V comme étant les variables, les applications V_1V_2 d'une valeur V_1 qui n'est pas une abstraction à une valeur V_2, et les abstractions quelconques $\lambda x \cdot u$. De façon équivalente:

$$V ::= \mathcal{V}V \dots V \mid \lambda x \cdot \Lambda$$

Ces règles sont une façon commode de dire que $\rhd$ est la plus petite relation binaire telle que $(\lambda x \cdot u)V \rhd u[x := V]$ pour tout terme u et toute valeur V, et passant aux contextes *applicatifs* (règles $(App_1\rhd)$ et $(App_2\rhd)$), mais ne passant pas nécessairement aux contextes d'abstraction. Formellement, les formules (ici, de la forme $u \rhd v$) au-dessus des barres sont des *prémisses*, qu'il faut vérifier avant de pouvoir appliquer la règle et d'en déduire la *conclusion* qui est en-dessous de la barre.

Pour corriger le problème de non-terminaison de la définition de $\lceil fact \rceil$ en tant que point fixe, on va définir $\lceil fact \rceil \hat{=} Y_v[F]$ au lieu de YF, où F est la fonctionnelle définissant $\lceil fact \rceil$, soit:

$$F \hat{=} \lambda f \cdot \lambda n \cdot \lceil if \rceil (\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (f(Pn)))$$

et $Y_v[F]$ est définie de sorte d'une part que $Y_v[F]V \rhd^* F(Y_v[F])V$ pour toute valeur V, et d'autre part que $Y_v[F]$ soit une *valeur*. On peut y arriver en effectuant quelques η-expansions dans YF, comme le montre l'exercice 21. Selon la stratégie interne faible, on aura alors:

$$\begin{array}{rcl} \lceil fact \rceil & = & Y_v[F] \\ & \rhd^* & (\lambda f \cdot \lambda n \cdot \lceil if \rceil (\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (f(Pn))))(Y_v[F]) \\ & = & (\lambda f \cdot \lambda n \cdot \lceil if \rceil (\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (f(Pn))))\lceil fact \rceil \\ & \rhd & \lambda n \cdot \lceil if \rceil (\lceil zero? \rceil n)\ \lceil 1 \rceil\ (\lceil \times \rceil n\ (\lceil fact \rceil (Pn))) \end{array}$$

où le dernier terme est une valeur, et donc la définition de $\lceil fact \rceil$ ne boucle plus.

Exercice 21 *Définissons* $Y_v[F] \hat{=} \lambda y \cdot (\lambda x \cdot F(\lambda z \cdot xxz))(\lambda x \cdot F(\lambda z \cdot xxz))y$. *Montrer que* $Y_v[F] \rhd^* F(Y_v[F])V$ *pour toute valeur* V.

Deuxième tricherie: Le calcul que l'on a fait de la factorielle de 3 par stratégie interne, maintenant modifiée en une stratégie interne faible, n'est toujours pas correct, même si on remplace $\rightarrow$ par $\rhd$. En effet, la réduction:

$$\begin{array}{rl} & \texttt{if } \mathbf{F} \texttt{ then } 1 \texttt{ else } 3 \times \texttt{fact } (3-1) \\ \rhd & 3 \times \texttt{fact } (3-1) \end{array}$$

est incorrecte: ce n'est pas une réduction par stratégie interne. En effet, revenons à une écriture plus formelle, alors la ligne du dessus est $\mathbf{F}\lceil 1 \rceil (\lceil \times \rceil\ \lceil 3 \rceil\ (\lceil fact \rceil (P\lceil 3 \rceil)))$, mais la réduction par stratégie interne demande que l'on calcule d'abord la valeur $\lceil 6 \rceil$ de $\lceil \times \rceil\ \lceil 3 \rceil\ (\lceil fact \rceil (P\lceil 3 \rceil))$ avant de conclure que l'on peut normaliser $\mathbf{F}\lceil 1 \rceil \lceil 6 \rceil$ en $\lceil 6 \rceil$. Or ce que nous avons fait ici, c'est utiliser une étape de stratégie *externe*, pour éviter de garder l'argument $\lceil 1 \rceil$ en attente, puisque son destin est d'être éliminé de toute façon.

Une conclusion provisoire est que les langages de programmation utilisent en fait une stratégie interne faible pour les appels de fonction normaux, mais une stratégie externe (faible elle aussi) pour les tests. La raison est pragmatique: il est plus avantageux de calculer de façon interne les appels de fonctions normaux, parce que leurs arguments devront en général être utilisés par la fonction, et que s'ils sont utilisés plusieurs fois, on n'aura pas à recalculer leur valeur plusieurs fois; mais d'autre part, les arguments en position "then" et "else" d'un `if` ne sont jamais utilisés plus d'une fois, et en fait l'un des deux ne sera pas utilisé, il est donc plus intéressant d'utiliser une stratégie externe dans ce cas, ce qui économise l'évaluation de l'argument qui ne sera jamais évalué.

3.4 Standardisation, appel par nom

Il est bon de remarquer, cependant, que changer de stratégie n'est pas innocent: certains termes ont une réduction interne qui termine mais aucune réduction externe finie. (Prendre l'exemple de l'exercice 6.) En revanche, et aussi bizarre que ça paraisse, la réduction externe gauche, aussi inefficace qu'elle puisse être en pratique, est optimale dans le sens suivant:

Théorème 3 (Standardisation) *Si u est faiblement normalisant, alors la stratégie externe gauche calcule la forme normale de u par une réduction finie.*

Autrement dit, la stratégie externe est peut-être lente, mais elle est sûre: elle ne se laisse pas embarquer dans des bouclages inutiles.

Preuve :

Avant de commencer, faisons quelques remarques sur les stratégies externes:

- D'abord, on peut écrire tout λ-terme u de façon unique sous la forme $\lambda x_1, \ldots, x_n \cdot hu_1 \ldots u_m$, où $hu_1 \ldots u_m$ n'est pas une abstraction, h n'est pas une application, et n et m valent possiblement 0. Le sous-terme h est appelé la *tête* de u, et la notation $\lambda x_1, \ldots, x_n \cdot hu_1 \ldots u_m$ est la *forme de tête* de u.

  En effet, si u est une abstraction $\lambda x \cdot u'$, on prend $x_1 = x$ et on continue le processus sur u'; après un nombre fini détapes, on aboutit ainsi à écrire u sous la forme $\lambda x_1, \ldots, x_n \cdot v$, où v n'est pas une abstraction. Dans ce cas, soit v est directement la tête h, soit v s'écrit $v'u_m$, et on recommence le processus avec v' à la place de v. Ce processus s'arrête lorsque v' devient un terme qui n'est pas une application, et on obtient ainsi v sous la forme désirée $hu_1 \ldots u_m$. (Exercice: formaliser cet argument.)

- Si la tête h de u est une abstraction $\lambda x \cdot v$, alors nécessairement $m \geq 1$ (sinon $hu_1 \ldots u_m$ serait une abstraction), et le sous-terme $(\lambda x \cdot v)u_1$ de u est un rédex. En fait, dans ce cas $(\lambda x \cdot v)u_1$ est un rédex le plus externe possible, et c'est le plus à gauche dans u de tous les rédex externes.

  Dans ce cas spécial, on dit que $(\lambda x \cdot v)u_1$ est le *rédex de tête* de u, et l'étape de réduction:

  $$u = \lambda x_1, \ldots, x_n \cdot (\lambda x \cdot v)u_1 \ldots u_m \to \lambda x_1, \ldots, x_n \cdot v[x := u_1]u_2 \ldots u_m$$

  est appelée *réduction de tête*.

  Notons $\to_t$ la réduction de tête en une étape, et $\to_t^*$ sa clôture réflexive transitive.

- Si par contre la tête h de u n'est pas une abstraction, auquel cas il n'y a pas de rédex de tête et le terme u est dit *normal de tête* (car il n'y a pas de réduction de tête), alors h est une variable. Cette variable est appelée la *variable de tête* de u.

  La propriété importante est que dans une forme normale de tête $u = \lambda x_1, \ldots, x_n \cdot hu_1 \ldots u_m$, les seules réductions possibles prennent place dans les u_i, et en fait sans aucune interférence entre les différents u_i. L'en-tête $\lambda x_1, \ldots, x_n \cdot h \ldots$ ne bougera plus jamais dans la réduction.

Nous donnons maintenant une preuve du théorème due à René David. On peut définir de façon astucieuse la notion de réduction *standard* $\to_s^*$ de u vers v, comme étant la plus petite relation binaire telle que:

1. si v est une variable, alors pour tout terme u tel que $u \to_t^* v$, on a $u \to_s^* v$;
2. sinon, v s'écrit sous forme de tête $\lambda x_1, \ldots, x_n \cdot v_0 v_1 \ldots v_m$, avec $n \geq 1$ ou $m \geq 1$. Alors, pour tout u tel que u se réduit par réduction de tête en la forme de tête $\lambda x_1, \ldots, x_n \cdot u_0 u_1 \ldots u_m$, si $u_i \to_s^* v_i$ pour tout i, $0 \leq i \leq n$, alors $u \to_s^* v$.

Ceci est une définition correcte de $u \to_s^* v$, par récurrence sur la taille de v. En effet, dans le cas 2, par la condition $n \geq 1$ ou $m \geq 1$, tous les v_i sont strictement plus petits que v.

Il est clair que la stratégie externe gauche est une stratégie standard. En effet, la stratégie externe gauche commence par calculer la forme normale de tête (par $\to_t^*$) $\lambda x_1, \ldots, x_n \cdot u_0 u_1 \ldots u_m$, puis normalise u_1 par la stratégie externe gauche, puis $u_2, \ldots, $ enfin u_m, u_0 étant une variable, sa normalisation est immédiate. La notion de réduction standard est plus souple en ce sens qu'on ne demande pas que u_0 soit une variable, et que l'on peut alterner des réductions dans u_1 avec des réductions dans $u_2, \ldots, u_m$, et même dans u_0.

On va démontrer que: (*) si $u \to^* v$, alors $u \to_s^* v$, pour tous termes u, v. Ceci impliquera le théorème: si u a une forme normale v, alors $u \to_s^* v$ par (*), et il ne reste plus qu'à démontrer par récurrence sur la taille de v que cette réduction standard induit une réduction externe gauche de u vers v. Si v est une variable, alors cette réduction standard est une réduction de tête, qui est bien une réduction externe gauche. Sinon, écrivons v sous forme normale de tête $\lambda x_1, \ldots, x_n \cdot v_0 v_1 \ldots v_n$. Alors nécessairement la réduction standard de u vers v commence par effectuer une réduction de tête de u vers une forme de tête $\lambda x_1, \ldots, x_n \cdot u_0 u_1 \ldots u_n$ suivie de réductions standard $u_0 \to_s^* v_0$, $u_1 \to_s^* v_1$, $\ldots$, $u_n \to_s^* v_n$. Ces réductions standard induisent des réductions externes gauches par hypothèse de récurrence, donc la réduction:

$$\begin{array}{rl} u \to_t^* & \lambda x_1, \ldots, x_n \cdot u_0 u_1 u_2 \ldots u_n \\ \to^* & \lambda x_1, \ldots, x_n \cdot v_0 u_1 u_2 \ldots u_n \\ \to^* & \lambda x_1, \ldots, x_n \cdot v_0 v_1 u_2 \ldots u_n \\ \to^* & \lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots u_n \\ \to^* & \ldots \\ \to^* & \lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots v_n = v \end{array}$$

est une réduction externe gauche.

Montrons maintenant (*). En première lecture, il est conseillé de sauter la démonstration, qui est assez technique. L'idée est de permuter les applications de la règle (β), et la difficulté est de le faire dans le bon ordre.

On va démontrer (*) par étapes. Le point important est le numéro 4 ci-dessous, les précédents sont des points techniques:

1. D'abord, on remarque que si $u = v$, alors $u \to_s^* v$; autrement dit, $\to_s^*$ est réflexive. Ceci se démontre par récurrence sur la taille de v. Si v est une variable, $u = v$ implique $u \to_t^* v$, donc $u \to_s^* v$. Sinon, on peut écrire v en forme de tête $\lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots v_m$, avec $n \geq 1$ ou $m \geq 1$, mais alors $u = \lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots v_n$ se réduit de tête (en 0 étape) en la forme de tête $\lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots v_n$, avec $v_i \to_s^* v_i$ pour tout i par hypothèse de récurrence.

2. Ensuite, si $u \to_t^* v$, alors $u \to_s^* v$. Ceci est encore par récurrence sur la taille de v. Si v est une variable, c'est évident, par la clause 1 de la définition de $\to_s^*$. Sinon, écrivons v en forme de tête $\lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots v_m$, avec $n \geq 1$ ou $m \geq 1$. Comme u se réduit par réduction de tête en v, et que $v_i \to_s^* v_i$ pour tout i par le point 1 ci-dessus, il s'ensuit que $u \to_s^* v$.

3. Si $u' \to_s^* w'$ et $u_1 \to_s^* w_1$, alors $u'[x := u_1] \to_s^* w'[x := w_1]$. Ceci se démontre par récurrence sur la taille de w'.

   Si w' est la variable x, alors $u' \to_t^* x$, donc $u'[x := u_1] \to_t^* u_1$. De $u_1 \to_s^* w_1$, on en déduit que $u'[x := u_1] \to_s^* w_1 = w'[x := w_1]$.

   Si w' est une autre variable y, alors $u' \to_t^* y = w'[x := w_1]$, et l'on conclut par le point 2.

   Sinon, puisque $u' \to_s^* w'$, on peut écrire $u' \to_t^* \lambda x_1, \ldots, x_n \cdot u'_0 u'_1 \ldots u'_m$, $u'_0 \to_s^* w'_0$, $\ldots$, $u'_m \to_s^* w'_m$ et $w' = \lambda x_1, \ldots, x_n \cdot w'_0 w'_1 \ldots w'_m$. Par α-renommage, on peut supposer que $x_1, \ldots, x_n$ sont fraîches. Par hypothèse de récurrence on a $u'_0[x := u_1] \to_s^* w'_0[x := w_1]$, $u'_1[x := u_1] \to_s^* w'_1[x := w_1]$, $\ldots$, $u'_m[x := u_1] \to_s^* w'_m[x := w_1]$. Donc $u'[x := u_1] \to_s^* w'[x := w_1]$.

4. Si $u \to_s^* w \to v$, alors $u \to_s^* v$. La démonstration est par récurrence sur la taille de w.

   Si $u \to_s^* w$ par la clause 1 de la définition de $\to_s^*$, c'est que w est une variable, et est donc normale. Mais ceci contredit $w \to v$, donc ce cas ne s'applique pas.

Sinon, $u \to_s^* w$ par la clause 2, donc u se réduit par réduction de tête en une forme de tête $\lambda x_1, \ldots, x_n \cdot u_0 u_1 u_2 \ldots u_n$, w a la forme de tête $\lambda x_1, \ldots, x_n \cdot w_0 w_1 w_2 \ldots w_n$ et $u_i \to_s^* w_i$ pour tout i, et $n \geq 1$ ou $m \geq 1$. La réduction de w vers v peut maintenant se produire en deux types d'endroits.

Cas 1: elle se produit dans l'un des w_j. Alors $v = \lambda x_1, \ldots, x_n \cdot v_0 v_1 v_2 \ldots v_n$, avec $v_i = w_i$ pour tout $i \neq j$, et $w_j \to v_j$ sinon. Noter que $u_j \to_s^* w_j \to v_j$, et que la taille de w_j est plus petite que celle de w, donc on peut appliquer l'hypothèse de récurrence et conclure que $u_j \to_s^* v_j$. D'autre part, pour $i \neq j$ on a $u_i \to_s^* w_i = v_i$. Donc dans tous les cas, pour tout i, $u_i \to_s^* v_i$. Il s'ensuit que $u \to_s^* v$, en utilisant la clause 2 de la définition de $\to_s^*$.

Cas 2: la réduction de w vers v est une réduction de tête; autrement dit, w_0 est une abstraction $\lambda x \cdot w'$ et la réduction donnée de u vers v est de la forme:

$$\begin{aligned} u \to_t^* \quad & \lambda x_1, \ldots, x_n \cdot (\lambda x \cdot u') u_1 u_2 \ldots u_n \\ \to^* \quad & \lambda x_1, \ldots, x_n \cdot (\lambda x \cdot w') w_1 w_2 \ldots w_n \\ \to \quad & \lambda x_1, \ldots, x_n \cdot w'[x := w_1] w_2 \ldots w_n = v \end{aligned}$$

avec $u' \to_s^* w'$, $u_1 \to_s^* w_1$, ..., $u_n \to_s^* w_n$. Noter en particulier que la tête de u, qui se réécrit en $\lambda x \cdot w'$ est bien une abstraction, donc nécessairement de la forme $\lambda x \cdot u'$ avec $u' \to_s^* w'$: si ce n'était pas le cas, la tête de u n'étant jamais une application ne pourrait être qu'une variable, qui serait normal et donc ne pourrait se réduire vers $\lambda x \cdot w'$.

Mais du coup on pouvait déjà directement réduire $(\lambda x \cdot u')u_1$ sans attendre que u_1 se réduise en w_1, et produire la réduction:

$$u \to_t^* \lambda x_1, \ldots, x_n \cdot (\lambda x \cdot u') u_1 u_2 \ldots u_n \to_t \lambda x_1, \ldots, x_n \cdot u'[x := u_1] u_2 \ldots u_n$$

Par le point 3 ci-dessus, $u'[x := u_1] \to_s^* w'[x := w_1]$, et comme $u_2 \to_s^* w_2$, ..., $u_n \to_s^* w_n$, il s'ensuit que $u \to_s^* v$ par la clause 2 de la définition de $\to_s^*$.

Nous démontrons maintenant (*), à savoir que $u \to^* v$ implique $u \to_s^* v$, par récurrence sur la longueur k de la réduction donnée de u vers v. Si cette longueur est 0, alors $u = v$ et on conclut par le point 1. Sinon, c'est que cette réduction s'écrit $u \to^* w \to v$, où la réduction de u à w est de longueur $k - 1$. Par hypothèse de récurrence, $u \to_s^* w$, et par le point 4 ci-dessus, $u \to_s^* v$. $\diamondsuit$

La stratégie externe gauche (*leftmost outermost strategy* en anglais) est souvent appelée *réduction gauche*, ou stratégie d'*appel par nom*. Cette dernière dénomination vient du langage Algol, et caractérise le fait que dans $(\lambda x \cdot u)v$, on remplace d'abord x par v dans u avant de réduire v — le terme non réduit v étant pris comme nom de sa valeur.

Exercice 22 *Montrer que si $u \to^* x$, où x est une variable, alors il existe une réduction de tête de u à x. (Regarder la preuve du théorème 3.)*

Exercice 23 **( )** *Utiliser la notion de réduction standard, et en particulier le point (*) de la démonstration du théorème 3 pour montrer directement que le λ-calcul est confluent.*

D'un point de vue pratique, sachant que ni les stratégies externes (appel *par valeur*) ni les stratégies internes (appel par nom) ne sont des panacées, pourquoi ne pas utiliser d'autres stratégies ? Une qui semble intéressante est la stratégie d'*appel par nécessité*, utilisée dans les langages comme Miranda ou Haskell. L'idée, c'est de partager tous les sous-termes d'un λ-terme, et de considérer les λ-termes comme des *graphes*. On verra comment faire cela dans la troisième partie. Disons tout de suite que ce n'est pas, en fait, une panacée.

4 Modèles, sémantique dénotationelle

La sémantique qu'on a donnée du λ-calcul est par réduction. Que ce soit la sémantique générale de la β-réduction ou la sémantique de la stratégie interne faible, cette sémantique est donnée par un ensemble de règles, plus la condition que la réduction est la plus petite relation binaire obéissant aux règles. On dit qu'il s'agit d'une sémantique *opérationnelle*.

Il arrive que l'on ait envie d'un autre style de sémantique, plus synthétique. Intuitivement, on aimerait pouvoir définir la sémantique du λ-calcul par une fonction $u \mapsto [\![u]\!]$ qui à tout terme u associe un vrai objet mathématique, dans un domaine D de valeurs. Par exemple, si $u = \lambda x \cdot x$, on aimerait dire que $[\![u]\!]$ est la fonction identité (sur un sous-ensemble de D).

$[\![\_]\!]$ sera alors un *modèle* s'il fournit une interprétation *saine* de la β-conversion, autrement dit si:

$$u =_\beta v \Rightarrow [\![u]\!] = [\![v]\!]$$

L'intérêt d'une telle approche, c'est qu'elle va simplifier les preuves, dans la plupart des cas. Par exemple, on pourra montrer que deux termes u et v ne sont pas convertibles en exhibant un modèle du λ-calcul suffisamment simple dans lequel $[\![u]\!] \neq [\![v]\!]$. De façon plus subtile, on utilisera des modèles dans lesquels une valeur spéciale $\perp$ sera réservée pour représenter la valeur des programmes qui ne terminent pas, et on pourra alors assurer qu'un λ-terme u termine rien qu'en calculant $[\![u]\!]$ et en vérifiant que le résultat est différent de $\perp$, c'est-à-dire sans avoir à réduire u pendant un temps indéterminé.

L'autre intérêt d'exhiber des modèles, c'est que souvent ils seront trop riches, et contiendront des valeurs qui ne sont des valeurs d'aucun λ-terme. Ceci permet soit de montrer que certaines constructions sémantiquement saines ne sont pas réalisables par programme dans le λ-calcul. Par contrecoup, ceci permettra de suggérer des extensions du λ-calcul, permettant de réaliser ces constructions, tout en garantissant à la fois que ces extensions sont cohérentes avec le λ-calcul et nécessaires pour réaliser la construction visée.

4.1 Quelques remarques liminaires

Avant de passer à plus compliqué, observons qu'il y a au moins deux modèles très simples du λ-calcul:

- le modèle trivial: D est un singleton $\{*\}$, et $[\![u]\!] = *$ pour tout u. Un tel modèle n'apportant pas grand-chose à notre compréhension, nous nous efforcerons de trouver des modèles *non triviaux*.
- Le *modèle de termes* $\Lambda/=_\beta$: ses éléments sont les classes d'équivalence de λ-termes modulo β-équivalence. Celui-ci est exactement celui qui nous intéresse, mais il n'apporte aucune facilité de calcul: pour raisonner dessus, on est obligé de passer par l'analyse des propriétés de $\rightarrow$ (confluence, standardisation, etc., sont les outils de base).

Nous allons donc essayer de trouver des modèles intermédiaires, qui soient non triviaux et sur lesquels on peut calculer.

Une première idée est la suivante: on prend un ensemble D de valeurs (sémantiques, à ne pas confondre avec la notion de valeurs de la section 3), et on essaie d'interpréter les termes dans D.

Commençons par les variables: que doit valoir $[\![x]\!]$? Tout choix étant arbitraire, on va supposer qu'on se donne toujours en paramètre une *valuation* ρ qui à toute variable associe sa valeur. On définira donc non pas $[\![u]\!]$ pour tout terme u, mais $[\![u]\!]\rho$ comme une fonction de u et de ρ. Clairement, on posera $[\![x]\!]\rho \hat{=} \rho(x)$.

Grâce aux valuations, on va pouvoir définir facilement la valeur des abstractions. Pour toute valuation ρ, pour toute variable x et toute valeur $v \in D$, définissons $\rho[x := v]$ comme étant la valuation qui à x associe v, et à toute autre variable y associe la valeur qu'elle avait par ρ, soit $\rho(y)$. En somme, $\rho[x := v]$ c'est "ρ, sauf que x vaut maintenant v". Le valeur $[\![\lambda x \cdot u]\!]\rho$ est alors juste la fonction[1] qui prend une valeur $v \in D$ et retourne $[\![u]\!](\rho[x := v])$.

Il y a juste un problème ici, c'est que la valeur de $\lambda x \cdot u$ est une fonction de D vers D. Mais comme c'est censé être aussi une valeur, elle doit être dans D aussi. En clair, on veut trouver un domaine D suffisamment gros pour que:

$$(D \rightarrow D) \subseteq D$$

où $D \rightarrow D$ dénote l'ensemble des fonctions de D vers D. Malheureusement:

[1] En français, on devrait dire une application, qui est une fonction définie sur tout son ensemble de départ. Comme le mot "application" est déjà pris par ailleurs, nous utiliserons les anglicismes "fonction partielle" pour fonction et "fonction totale" pour application, et conviendrons que "fonction" signifie "fonction totale".

Lemme 4 *Si $(D \to D) \subseteq D$, alors D est trivial.*

Preuve : D ne peut pas être vide, car sinon $D \to D$ contiendrait un élément (la fonction vide) qui n'est pas dans D. Si D n'était pas trivial, alors son cardinal α serait donc au moins 2. Mais alors le cardinal de $D \to D$ serait $\alpha^\alpha \geq 2^\alpha > \alpha$ par le théorème de Cantor, ce qui entraîne que $D \to D$ ne peut pas être contenu dans D. $\diamondsuit$

En fait, on veut aussi que $D \subseteq (D \to D)$, mais c'est plus facile à obtenir. La raison est que l'on veut pouvoir interpréter $[\![uv]\!]\rho$ comme l'application de $[\![u]\!]\rho$ à $[\![v]\!]\rho$, mais pour cela il faut que toute valeur possible pour $[\![u]\!]\rho$ (dans D) puisse être vue comme une fonction de D dans D.

Il faut donc trouver d'autres solutions. Une idée due à Dana Scott est de se restreindre à des domaines D munie d'une structure supplémentaire, et à demander à ce que l'espace de fonctions de D vers D préserve la structure. Ceci permet de court-circuiter l'argument de circularité. Par exemple, si on prend $D = \mathbb{R}$ avec sa structure d'espace topologique, et qu'on ne garde que les fonctions continues de $\mathbb{R}$ vers $\mathbb{R}$, alors il n'y a pas plus de fonctions continues que de réels. (La raison est que le cardinal des réels est $2^{\aleph_0}$, et que les fonctions continues de $\mathbb{R}$ vers $\mathbb{R}$ sont déterminées de façon unique comme les prolongements par continuité de leurs restriction à $\mathbb{Q}$. Il n'y en a donc pas plus que de fonctions de $\mathbb{Q}$ vers $\mathbb{R}$, soit pas plus que $(2^{\aleph_0})^{\aleph_0} = 2^{\aleph_0}$.) Malheureusement, même si le problème de cardinalité est ainsi vaincu, ça ne suffit pas pour résoudre le problème entièrement.

4.2 Les ordres partiels complets (cpo)

La solution de Scott est de considérer des domaines D qui sont des *cpo*, ou *ordres partiels complets* (cpo signifie "complete partial order"):

Définition 4 *Un* ordre partiel *est un couple $(D, \leq)$, où $\leq$ est une relation d'ordre sur D.*

Un majorant d'une partie E de D est un élément x de D tel que $y \leq x$ pour tout y dans E. Une borne supérieure *de E est un majorant minimal $\bigsqcup E$; si elle existe, elle est unique.*

Une chaîne *dans $(D, \leq)$ est une suite infinie croissante $(x_i)_{i \in \mathbb{N}}$ d'éléments de D (soit: $i \leq j$ implique $x_i \leq x_j$).*

On dira qu'un tel ordre partiel est complet *si et seulement si D a un élément minimum $\perp$ et si tout chaîne a une borne supérieure.*

Une fonction f de $(D, \leq)$ vers $(D', \leq')$ est dite monotone *si et seulement si $x \leq y$ implique $f(x) \leq' f(y)$. Elle est* continue *si et seulement si elle préserve les bornes supérieures de chaînes:*

$$f(\bigsqcup C) = \bigsqcup \{f(c) \mid c \in C\}$$

pour toute chaîne C dans D.

L'idée est qu'une chaîne est une suite d'approximations de la valeur que l'on souhaite calculer, et l'ordre $\leq$ est l'ordre "est moins précis que". L'élément $\perp$ représente alors l'absence totale d'information, et la borne supérieure d'une chaîne représente la valeur finale d'un calcul.

Précisément, on peut voir les chaînes émerger à partir de l'analyse de la réduction d'un terme. Par exemple, si on prend le terme $J \hat{=} YG$, avec $G \hat{=} \lambda x, y, z \cdot y(xz)$, on a les réductions suivantes, avec à chaque étape ce qu'on peut conclure sur le résultat final R du calcul, s'il existe:

$$\begin{array}{rll}
J = & YG & R = ? \\
\to^* & G(YG) & R = ? \\
\to & \lambda y, z \cdot y(YGz) & R = \lambda y, z \cdot y? \\
\to^* & \lambda y, z \cdot y(G(YG)z) & R = \lambda y, z \cdot y? \\
\to^* & \lambda y, z \cdot y(\lambda z' \cdot z(YGz')) & R = \lambda y, z \cdot y(\lambda z' \cdot z?) \\
\to^* & \lambda y, z \cdot y(\lambda z' \cdot z(\lambda z'' \cdot z'(YGz''))) & R = \lambda y, z \cdot y(\lambda z' \cdot z(\lambda z'' \cdot z'?)) \\
& \ldots &
\end{array}$$

où les points d'interrogation dénotent des termes encore inconnus. (Noter qu'il s'agit d'isoler la partie du terme à chaque ligne qui est en forme normale de tête, cf. théorème 3). On peut ordonner les "termes partiels" contenant des points d'interrogation par une relation $\leq$ telle que $u \leq v$ si et seulement si v résulte de u par le remplacement de points d'interrogations par des termes partiels. La colonne de droite définit alors bien une chaîne dans l'espace des termes partiels , où $? = \bot$. Pour que de telles chaînes aient des bornes supérieures, on est obligé d'enrichir l'espace des termes partiels par des termes infinis. Par exemple, la borne supérieure de la chaîne ci-dessus est:

$$\lambda z_0, z_1 \cdot z_0(\lambda z_2 \cdot z_1(\lambda z_3 \cdot z_2(\ldots(\lambda z_{k+1} \cdot z_k(\lambda z_{k+2} \cdot z_{k+1}(\ldots$$

après un α-renommage adéquat. On note en général plutôt Ω le point d'interrogation, et l'espace des arbres finis ou infinis ainsi obtenus s'appelle l'espace des *arbres de Böhm*.

Exercice 24 *Construire la suite d'arbres de Böhm associée à la réduction standard de $\lambda x \cdot x$. Quelle est sa borne supérieure ?*

Exercice 25 *Construire la suite d'arbres de Böhm associée à la réduction standard de $(\lambda x \cdot xx)(\lambda x \cdot xx)$. Quelle est sa borne supérieure ? Justifier pourquoi on appelle ce terme Ω.*

Le modèle des arbres de Böhm — pour vérifier qu'il s'agit effectivement d'un modèle, on consultera [Bar84] — est relativement peu informatif encore: il code essentiellement la syntaxe et la réduction, modulo le théorème de standardisation.

On va maintenant construire quelques modèles $(D, \leq)$, donc quelques cpo tels que $[D \to D] = D$, où $[D \to D]$ est l'espace des fonctions continues de D dans D. Une première observation, c'est que nous avons juste besoin que $[D \to D]$ et D soient isomorphes, en fait même seulement qu'il existe deux fonctions continues:

$$i : D \to [D \to D] \qquad r : [D \to D] \to D$$

telles que $i \circ r$ soit l'identité sur $[D \to D]$. Un tel domaine D est appelé un *domaine réflexif.*

En effet, on définira alors:

$$\begin{aligned} [\![x]\!]\rho = &\quad \rho(x) \\ [\![uv]\!]\rho = &\quad i([\![u]\!]\rho)([\![v]\!]\rho) \\ [\![\lambda x \cdot u]\!]\rho = &\quad r(v \mapsto [\![u]\!](\rho[x := v])) \end{aligned}$$

où $v \mapsto e$ dénote la fonction qui à la valeur v associe la valeur e (pour ne pas provoquer de confusions avec la notation des λ-termes, nous ne l'écrivons pas $\lambda v \cdot e$).

On a alors:

Lemme 5 (Correction) *Si $u =_\beta v$, alors $[\![u]\!]\rho = [\![v]\!]\rho$ pour tout ρ.*

Preuve : Il suffit de montrer le résultat lorsque $u \to v$. On le montre alors par récurrence sur la profondeur du rédex contracté dans u pour obtenir v. Le cas inductif, où cette profondeur est non nulle, est trivial. Dans le cas de base, on doit démontrer que $[\![(\lambda x \cdot s)t]\!]\rho = [\![s[x := t]]\!]\rho$. Mais $[\![(\lambda x \cdot s)t]\!]\rho = i([\![\lambda x \cdot s]\!]\rho)([\![t]\!]\rho) = i(r(v \mapsto [\![s]\!](\rho[x := v])))([\![t]\!]\rho) = (v \mapsto [\![s]\!](\rho[x := v])([\![t]\!]\rho)$ (puisque $i \circ r$ est l'identité) $= [\![s]\!](\rho[x := [\![t]\!]\rho])$, et il ne reste plus qu'à montrer que ce dernier vaut $[\![s[x := t]]\!]\rho$. Ceci se fait par une récurrence structurelle facile sur s. $\diamondsuit$

4.3 Le modèle $\mathbb{P}\omega$

Une première construction d'un modèle, due à Plotkin et Scott, et qui est très concrète, est le modèle $\mathbb{P}\omega$ des parties de l'ensemble $\mathbb{N}$ des entiers naturels (on note aussi $\mathbb{N} = \omega$, d'où le nom), avec l'ordre $\subseteq$.

La construction du modèle $\mathbb{P}\omega$ est fondée sur quelques remarques. D'abord, on peut représenter tout couple $\langle m, n\rangle$ d'entiers par un entier, par exemple par la formule:

$$\langle m, n\rangle \hat{=} \frac{(m+n)(m+n+1)}{2} + m$$

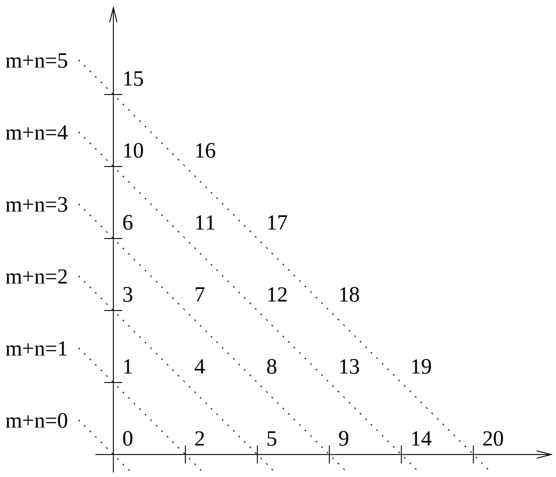


Figure 1: Codage des couples d'entiers

La valeur de $\langle m, n\rangle$ en fonction de m en abscisse, et de n en ordonnée, est donnée en figure 1.

Ensuite, on peut représenter tout ensemble fini $e \hat{=} \{n_1, \ldots, n_k\}$ d'entiers par le nombre binaire $[e] \hat{=} \sum_{i=1}^{k} 2^{n_i}$. Réciproquement, tout entier m peut être écrit en binaire, et représente l'ensemble fini e_m de tous les entiers n tels que le bit numéro n de m est à 1: cf. figure 2, où l'on a écrit l'ensemble $\{1, 3, 4, 7, 9, 10\}$ sous la forme du nombre binaire 11010011010, soit 1690 en décimal — autrement dit, $[\{1, 3, 4, 7, 9, 10\}] = 1690$ et $e_{1690} = \{1, 3, 4, 7, 9, 10\}$.

	10	9	8	7	6	5	4	3	2	1	0	
{1, 3, 4, 7, 9, 10} =	1	1	0	1	0	0	1	1	0	1	0	= 1690

Figure 2: Codage des ensembles finis

Exercice 26 *Trouver une formule pour* $\pi_1 : \langle m, n\rangle \mapsto m$ *et pour* $\pi_2 : \langle m, n\rangle \mapsto n$.

L'astuce principale dans la construction de $\mathbb{P}\omega$ est que la continuité des fonctions de $\mathbb{P}\omega$ dans $\mathbb{P}\omega$ permet de les représenter comme limites d'objets finis:

Lemme 6 *Pour toute fonction* f *de* $\mathbb{P}\omega$ *dans* $\mathbb{P}\omega$, f *est continue si et seulement si pour tout* $x \in \mathbb{P}\omega$, $f(x)$ *est l'union de tous les* $f(y)$, y *partie finie de* x.

Preuve : Seulement si: supposons f continue, et soit x un ensemble non vide. Énumérons ses éléments, par exemple en ordre croissant n_1, n_2, …, n_k, … (si la suite s'arrête à l'indice k, on poserait $n_k = n_{k+1} = n_{k+2} = \ldots$); posons $y_k \hat{=} \{n_1, n_2, \ldots, n_k\}$ pour tout k, ceci définit une chaîne dont la borne supérieure est x. Comme f est continue, $f(x) = \bigcup_k f(y_k) \subseteq \bigcup_{y \text{ finie } \subseteq x} f(y)$; réciproquement, $\bigcup_{y \text{ finie } \subseteq x} f(y) \subseteq f(x)$ car f est monotone, donc $f(y) \subseteq f(x)$ pour tout $y \subseteq x$. Finalement, si x est vide, $f(x) = \bigcup_{y \text{ finie } \subseteq x} f(y)$ est évident.

Si: supposons que $f(x) = \bigcup_{y \text{ finie } \subseteq x} f(y)$, et soit $(y_k)_{k \in \mathbb{N}}$ une chaîne, $x = \bigcup_k y_k$. Donc $f(x)$ est l'union des $f(y)$, y finie partie de x, et contient en particulier tous les $f(y_k)$, donc $f(x) \supseteq \bigcup_k f(y_k)$; réciproquement, $f(x) \subseteq \bigcup_k f(y_k)$ car toute partie finie y de x est incluse dans un y_k. $\diamondsuit$

L'idée est alors que toute fonction continue est définie par ses valeurs sur les ensembles finis, et que les ensembles finis sont codables par des entiers. On définit donc:

$$r : [\mathbb{P}\omega \to \mathbb{P}\omega] \to \mathbb{P}\omega \qquad f \mapsto \{\langle m, n\rangle \mid n \in f(e_m)\}$$

et son inverse à gauche:

$$i : \mathbb{P}\omega \to [\mathbb{P}\omega \to \mathbb{P}\omega] \qquad e \mapsto (x \mapsto \{n \in \mathbb{N} \mid \exists y \text{ fini } \subseteq x \cdot \langle [y], n\rangle \in e\}$$

Théorème 4 *La fonction $i \circ r$ est l'identité sur $[\mathbb{P}\omega \to \mathbb{P}\omega]$, i et r sont continues. De plus, la fonction $(u, \rho) \mapsto [\![u]\!]\rho$ est bien définie.*

Autrement dit, on a bien défini un modèle. Le fait que la fonction $(u, \rho) \mapsto [\![u]\!]\rho$ soit bien définie signifie notamment que, dans la définition de $[\![\lambda x \cdot u]\!]\rho$, la fonction $v \mapsto [\![u]\!](\rho[x := v])$ est bien continue — ce qui n'est pas totalement immédiat. Finalement, il est clair que $\mathbb{P}\omega$ est non trivial.

Exercice 27 *Démontrer le théorème 4. (En cas de panne, se référer à [Bar84], pp.469–476.)*

Exercice 28 *Calculer $[\![\lambda x \cdot x]\!]\rho$ dans $\mathbb{P}\omega$, et montrer qu'il est non vide.*

Exercice 29 *Calculer $[\![\lambda x, y \cdot xy]\!]\rho$ dans $\mathbb{P}\omega$, et montrer qu'il est différent de $[\![\lambda x \cdot x]\!]\rho$. En déduire que la règle (η) n'est pas déductible de la (β)-équivalence, autrement dit $\lambda x \cdot ux \neq_\beta u$ en général.*

On peut aussi montrer ce dernier résultat syntaxiquement: $\lambda x, y \cdot xy$ et $\lambda x \cdot x$ sont η-équivalents mais normaux et différents, donc par β-équivalents, par la confluence du λ-calcul. La preuve sémantique via $\mathbb{P}\omega$ remplace la démonstration combinatoire de la confluence du λ-calcul par la démonstration sémantique que $\mathbb{P}\omega$ est un modèle.

4.4 Sémantiques d'un λ-calcul enrichi

Une famille plus générale de modèles, due à Scott, est présentée en annexe A. Elle permet de montrer le résultat suivant:

Théorème 5 (Scott) *Soit $(D_0, \leq_0)$ un cpo quelconque. Alors il existe un cpo $(D_\infty, \leq_\infty)$ contenant $(D_0, \leq_0)$ et tel que $D_\infty = [D_\infty \to D_\infty]$ à isomorphisme près.*

Un intérêt que l'on peut avoir dans les cpo ne tient pas tant au fait qu'on puisse fabriquer des domaines D réflexifs, mais au fait que finalement ils modélisent une notion de calcul par *approximations successives*.

On peut alors fabriquer des cpo non nécessairement réflexifs pour modéliser des concepts informatiques intéressants. Par exemple, le cpo des booléens est $\mathbb{B}_\perp \hat{=} \{\mathbf{F}, \mathbf{V}, \perp\}$, avec l'ordre $\perp \leq \mathbf{F}$, $\perp \leq \mathbf{V}$, mais $\mathbf{F}$ et $\mathbf{V}$ incomparables. En clair, et pour paraphraser l'exemple des arbres de Böhm plus haut, un programme qui doit calculer un booléen soit n'a pas encore répondu (sa valeur courante est $\perp$) soit a répondu $\mathbf{F}$ (et on sait tout sur la valeur retournée, donc $\mathbf{F}$ doit être un élément maximal), soit a répondu $\mathbf{V}$ (de même). Comme $\mathbf{F}$ et $\mathbf{V}$ sont deux valeurs différentes, et maximales en termes de précision, elles doivent être incomparables.

Le cpo des entiers, de même, est $\mathbb{N}_\perp \hat{=} \mathbb{N} \cup \{\perp\}$, avec l'ordre dont les seules instances non triviales sont $\perp \leq n$, $n \in \mathbb{N}$. Les entiers sont incomparables entre eux pour les mêmes raisons que les booléens, et son diagramme de Hasse est donné en figure 3. Un tel cpo, dont la hauteur est inférieure ou égale à 1, est dit *plat*.

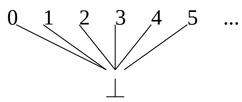


Figure 3: Le cpo plat des entiers naturels

Exercice 30 *Si D_1 et D_2 sont deux cpo, montrer que $[D_1 \to D_2]$ n'est jamais plat, sauf si $D_1 = \{\perp\}$ et D_2 est plat.*

On peut alors construire sémantiquement un langage plus riche que le λ-calcul pur, mais qui contienne toujours le λ-calcul. Par exemple, on peut décider d'ajouter à la syntaxe des termes du λ-calcul des constantes `F` et `V`, la constante `0` et un symbole `S` représentant le successeur.

Sémantiquement, on a besoin d'un domaine réflexif (pour avoir les fonctions i et r) contenant l'union de $\mathbb{B}_\perp$ et $\mathbb{N}_\perp$. C'est facile: appliquer le théorème 5 à $D_0 \hat{=} \mathbb{B}_\perp \cup \mathbb{N}_\perp$ (un cpo plat encore). On peut alors définir:

$$\begin{array}{rcl}
[\![\mathtt{T}]\!]\rho & \hat{=} & \mathbf{V} \\
[\![\mathtt{F}]\!]\rho & \hat{=} & \mathbf{F} \\
[\![\mathtt{0}]\!]\rho & \hat{=} & 0 \\
[\![\mathtt{S}]\!]\rho & \hat{=} & r(v \mapsto v+1)
\end{array}$$

Il y a quand même un problème dans cette définition, à savoir que $v+1$ n'est défini que quand v est un entier, et donc la fonction $v \mapsto v+1$ n'est pas définie. Si $v = \perp$, c'est-à-dire que typiquement v est le “résultat” d'un programme qui ne termine pas, intuitivement on va prendre $v+1 = \perp$, autrement dit le calcul de $v+1$ ne terminera pas non plus: une telle fonction, qui envoie $\perp$ sur $\perp$, est dite *stricte*.

Si v n'est pas dans $\mathbb{N}_\perp$, alors une convention possible est de considérer que $v+1$ est un programme absurde, qui ne termine pas, autrement dit $v+1 = \perp$. En pratique, un tel programme absurde plante, ou s'interrompt sur un message d'erreur, et c'est en ce sens qu'il ne termine pas: il n'arrive jamais au bout du calcul.

Regardons maintenant les fonctions que l'on peut définir sur les booléens. Le “ou” $\vee$ doit avoir la propriété que $\mathbf{F} \vee \mathbf{F} = \mathbf{F}$ et $\mathbf{V} \vee x = x \vee \mathbf{V} = \mathbf{V}$ pour tout $x \in \mathbb{B}$, mais qu'en est-il si $x = \perp$?

En Pascal, le ou est strict en ses deux arguments. Autrement dit, $x \vee y$ est calculé en calculant x, y, puis en en prenant la disjonction. En C, le ou est strict en son premier argument mais pas en son second: $x||y$ est calculé en calculant x; si x vaut $\mathbf{V}$, alors $\mathbf{V}$ est retourné comme valeur, sinon c'est celle de y qui est calculée. En particulier, $\mathbf{V} \vee \perp = \mathbf{V}$. La table de vérité complète est:

`||`	**V**	**F**	$\perp$
V	**V**	**V**	**V**
F	**V**	**F**	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$

Il y a encore d'autres possibilités. Une qui maximise le nombre de cas où $x \vee y$ termine est donnée par la table de vérité:

$\mid$	**V**	**F**	$\perp$
V	**V**	**V**	**V**
F	**V**	**F**	$\perp$
$\perp$	**V**	$\perp$	$\perp$

Cette fonction $\mid$ est en effet continue, donc sémantiquement acceptable. Elle est connue sous le nom de “ou parallèle”, car on peut la réaliser en évaluant x sur un processeur 1, y sur un processeur 2. Le premier des deux qui répond $\mathbf{V}$ interrompt l'autre, et $\mathbf{V}$ est retourné. Si les deux répondent $\mathbf{F}$, alors $\mathbf{F}$ est retourné. Sinon, rien n'est retourné, autrement dit la valeur finale est $\perp$.

Il est remarquable que, alors que le ou de Pascal et le `||` de C étaient simulables en λ-calcul, le ou parallèle ne l'est pas. La raison est que le λ-calcul est *intrinsèquement séquentiel.* Ceci ce manifeste mathématiquement par le fait que les valeurs sémantiques des fonctions définissables par des λ-termes sont non seulement continues, mais *stables*:

Définition 5 *Deux éléments x et y d'un cpo sont dits* compatibles *si et seulement s'il existe z dans le cpo tel que $x \leq z$ et $y \leq z$.*

Supposons que le cpo D ait la propriété que pour tous éléments compatibles x et y, il existe une borne inférieure $x \sqcap y$. Une fonction $f : D \to D'$ est stable *si et seulement si elle est continue, et pour tous x, y compatibles dans D la borne inférieure $f(x) \sqcap f(y)$ existe et est égale à $f(x \sqcap y)$.*

*On munit l'espace des fonctions stables de D vers D' de l'*ordre de Berry*: $f \leq_s g$ si et seulement si $x \leq y$ implique que $f(y) \sqcap g(x)$ existe et égale $f(x)$.*

Exercice 31 *Montrer que le ou parallèle n'est pas stable. En déduire qu'on ne peut pas le coder en λ-calcul enrichi avec* `T`*,* `F`*,* `0`*,* `S`*.*

On peut coder le test "if ... then ... else ...". Sémantiquement, c'est la fonction $if(x, y, z)$ telle que $if(\mathbf{V}, y, z) = y$, $if(\mathbf{F}, y, z) = z$, et si x n'est pas un booléen alors $if(x, y, z) = \bot$. Noter que cette fonction est stricte en son premier argument, mais pas en son deuxième et en son troisième.

Par contre, des fonctions comme $+$ et $\times$ seront strictes en tous les arguments. Même $\times$ est en général choisie stricte, ce qui signifie que $0 \times \bot = \bot$ et non 0. On remarquera qu'une fonction appelée par valeur ($+$, $\times$) est nécessairement stricte, car cette fonction attend que ses arguments soient évalués avant de continuer le calcul; alors qu'une fonction en appel par nom peut être non stricte. (Le cpo que nous choisissons pour comprendre ces notions dans un cadre d'analyse des réductions est celui dont les valeurs sont les arbres de Böhm.) Le cadre sémantique est plus souple et accomode les deux notions dans une seule définition. De plus, les détails réels des réductions importent peu, finalement, du moment que les équations sémantiques sont respectées.

Par exemple, un λ-calcul avec booléens et entiers, en appel par valeur sauf sur le "if ... then ... else ...", serait défini comme suit. On prend un cpo D tel que:

$$D = (\mathbb{N} \oplus \mathbb{B} \oplus [D \to D])_\bot$$

modulo un isomorphisme qui sera laissé implicite dans la suite, où $\oplus$ dénote l'union disjointe d'ensembles ordonnés, et $D_\bot$ dénote l'ensemble ordonné obtenu à partir de D en ajoutant un élément $\bot$ plus bas que tous les éléments de D, et on définit $[\![\_]\!]$ comme en figure 4.

$$\begin{array}{rcl}
[\![x]\!]\rho & \hat{=} & \rho(x) \\
[\![\lambda x \cdot u]\!]\rho & \hat{=} & (v \mapsto \left\{ \begin{array}{ll} \bot & \text{si } v = \bot \\ [\![u]\!](\rho[x := v]) & \text{sinon} \end{array} \right.) \\
[\![uv]\!]\rho & \hat{=} & \left\{ \begin{array}{ll} f([\![v]\!]\rho) & \text{si } f \hat{=} [\![u]\!]\rho \in [D \to D] \\ \bot & \text{sinon} \end{array} \right. \\
[\![\mathtt{T}]\!]\rho & \hat{=} & \mathbf{V} \\
[\![\mathtt{F}]\!]\rho & \hat{=} & \mathbf{F} \\
[\![\mathtt{0}]\!]\rho & \hat{=} & 0 \\
[\![\mathtt{S}]\!]\rho & \hat{=} & r(v \mapsto \left\{ \begin{array}{ll} v+1 & \text{si } v \in \mathbb{N} \\ \bot & \text{sinon} \end{array} \right.) \\
[\![if(u, v, w)]\!]\rho & \hat{=} & \left\{ \begin{array}{ll} [\![v]\!]\rho & \text{si } [\![u]\!]\rho = \mathbf{V} \\ [\![w]\!]\rho & \text{si } [\![u]\!]\rho = \mathbf{F} \\ \bot & \text{sinon} \end{array} \right.
\end{array}$$

Figure 4: Sémantique dénotationnelle de l'appel par valeur

Dans les langages dits *paresseux*, comme Miranda ou Haskell, qui simule un appel par nom d'une façon un peu plus efficace (voir partie 3), la seule différence serait qu'il n'attend pas d'évaluer l'argument, et donc on aurait juste:

$$[\![\lambda x \cdot u]\!]\rho \quad \hat{=} \quad (v \mapsto [\![u]\!](\rho[x := v]))$$

On voit alors qu'on peut réaliser une fonction $\lambda x \cdot u$ qui est normalement en appel par nom par une stratégie d'appel par valeur exactement quand les deux définitions de sa sémantique coïncident, autrement dit quand cette fonction est *stricte*. L'appel par valeur étant dans ce cas moins coûteux en temps et en espace, un bon compilateur pourra compiler les fonctions détectées comme strictes par une stratégie d'appel par valeur. Cette détection de fonctions strictes est appelée en anglais la “strictness analysis”.

5 Style de passage de continuations

Les modèles sémantiques dénotationnels que l'on a vus jusqu'ici — dits *en style direct* — ont l'avantage d'oublier tous les détails liés à l'ordre d'évaluation, excepté les points importants comme la terminaison (valoir $\perp$ ou pas). C'est parfois aussi un inconvénient, et on aimerait parfois pouvoir exprimer une partie de la stratégie d'évaluation. Par exemple, dans la sémantique proposée en figure 4, rien ne dit que pour évaluer uv, on doive évaluer u avant v.

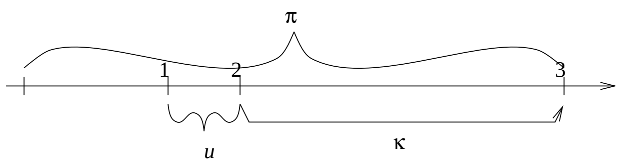
C'est gênant notamment si l'on enrichit notre λ-calcul avec des constructions telles que les *affectations* (`setq` en Lisp, `:=` ou `<-` en Caml, `:=` en Pascal, `=` en C), ou les *sauts non locaux* (`try` et `raise` en Caml, `goto` en Pascal, `setjmp` et `longjmp` en C). Prenons le cas des affectations tout d'abord, parce qu'il est plus simple à expliquer. Supposons qu'on veuille calculer la valeur de uv, et que l'évaluation de u affecte une référence x à 1, et l'évaluation de v affecte x à 2. Si u est évalué avant v, alors x vaudra 2 à la fin du calcul, et si c'est v, alors x vaudra 1 à la fin du calcul: la sémantique des deux cas n'est donc pas la même.

5.1 Le concept de continuation

Une façon de spécifier l'ordre d'évaluation en sémantique dénotationnelle est de faire prendre à la fonction d'évaluation un deuxième argument, la *continuation* κ, qui est une fonction de D vers D, et de convenir que:

$$[\![u]\!]\rho\kappa$$

est la valeur non pas de u mais d'un programme tout entier dont un sous-terme est u: κ dénote la suite du calcul une fois qu'on aura évalué u, et si la valeur de u est d, alors la valeur du programme tout entier sera $\kappa(d)$. Graphiquement, on peut imaginer que l'on a un programme π, dont l'exécution va consister en une suite d'instructions, et qu'au milieu de cette suite on trouve une sous-suite d'instructions pour évaluer u:



Lorsque l'exécution se trouve au début de u (au point 1), ce graphique dit que ce qu'il reste à faire pour trouver la valeur de π au point 3, c'est calculer la valeur de u, et une fois qu'on l'aura (au point 2), il suffira d'appliquer κ, la suite du calcul.

Formellement, examinons déjà comme on peut modéliser une stratégie (faible) d'appel par valeur gauche sur les λ-termes purs par ce moyen (le codage des booléens, des entiers, etc., est laissé en exercice), et nous verrons ensuite comment ce style de sémantique permet de modéliser les constructions de style `setjmp`/`longjmp`, puis les affectations.

Tout d'abord, $[\![x]\!]\rho\kappa$, lorsque x est une variable, doit évaluer x, et passer sa valeur $\rho(x)$ à κ. On va donc définir $[\![x]\!]\rho\kappa \hat{=} \kappa(\rho(x))$.

Dans le cas des applications, $[\![uv]\!]\rho\kappa$ doit d'abord évaluer u, puisque nous avons décidé d'une stratégie gauche. $[\![uv]\!]\rho\kappa$ vaudra donc $[\![u]\!]\rho\kappa'$ pour une continuation κ' à trouver. Cette continuation κ' est une fonction qui prend la valeur f de u, et va ensuite calculer v. Donc κ' doit être de la forme $(f \mapsto [\![v]\!]\rho'\kappa'')$ pour un environnement ρ' et une continuation κ'' à trouver. Mais v s'évalue dans le même environnement que uv, donc $\rho' = \rho$. D'autre part, κ'' doit être une fonction qui prend la valeur d de v, applique la valeur f de u (que l'on supposera être une fonction) à d, et continue le reste du calcul (κ) avec la valeur $f(d)$. En résumé:

$$[\![uv]\!]\rho\kappa \hat{=} [\![u]\!]\rho(f \mapsto [\![v]\!]\rho(d \mapsto \kappa(f(d))))$$

Cette formule peut avoir l'air illisible, mais voici un truc pour déchiffrer ce genre de notations intuitivement: $[\![u]\!]\rho(f \mapsto \ldots$ se lit "évaluer u dans l'environnement ρ, ceci donne une valeur f; ensuite ...". D'autre part, appliquer κ à une valeur d se lit: "retourner d". La formule ci-dessus se lit donc:

Évaluer u dans l'environnement ρ, ceci donne une valeur f; ensuite,
évaluer v dans l'environnement ρ, ceci donne une valeur d; ensuite,
retourner $f(d)$.

Pour évaluer une λ-abstraction $\lambda x \cdot u$ dans l'environnement ρ et la continuation κ, on va d'abord calculer la valeur de $\lambda x \cdot u$, et ensuite la retourner par κ. Cette valeur est une fonction qui prend une valeur d pour x, et ensuite évalue u ...mais avec quelle continuation ? L'idée, c'est qu'une fois que le calcul de u sera terminé, la continuation devra juste retourner cette valeur: la continuation cherchée est l'identité id. En somme:

$$[\![\lambda x \cdot u]\!]\rho\kappa \hat{=} \kappa(d \mapsto [\![u]\!](\rho[x := d])id)$$

Cette sémantique, qui est notre premier exemple d'une sémantique *par passage de continuations*, est résumée en figure 5. Nous avons affiné la définition dans le cas de l'application uv, pour tenir compte de ce qui se passe lorsque f n'est pas une fonction dans le domaine sémantique D considéré. Noter qu'on ne retourne pas $\perp$ (on n'a pas écrit $\kappa(\perp)$), mais on interrompt l'exécution complète du programme, qui retourne immédiatement $\perp$. (C'est un plantage.)

$$\begin{array}{rcl} [\![x]\!]\rho\kappa & \hat{=} & \kappa(\rho(x)) \\ [\![uv]\!]\rho\kappa & \hat{=} & [\![u]\!]\rho\left(f \mapsto [\![v]\!]\rho\left(d \mapsto \left\{ \begin{array}{ll} \kappa(f(d)) & \text{si } f \text{ est une fonction} \\ \perp & \text{sinon} \end{array} \right.\right)\right) \\ [\![\lambda x \cdot u]\!]\rho\kappa & \hat{=} & \kappa(d \mapsto [\![u]\!](\rho[x := d])id) \end{array}$$

Figure 5: Une sémantique de l'appel par valeur en style de passage de continuations

On notera que l'on aurait pu définir l'application autrement, par exemple:

$$[\![u]\!]\rho\left(f \mapsto \left\{ \begin{array}{ll} [\![v]\!]\rho(d \mapsto \kappa(f(d))) & \text{si } f \text{ est une fonction} \\ \perp & \text{sinon} \end{array} \right.\right)$$

Avec cette autre définition, si f n'est pas une fonction, la machine plante avant même d'évaluer v.

5.2 Sauts non locaux et captures de continuations

Voyons maintenant comment on peut coder un mécanisme à la `setjmp/longjmp` de C. Rappelons que `setjmp` capture une copie du contexte d'évaluation courant (le compteur de programme, le pointeur de pile, les registres) et la met dans une structure k, que nous représenterons par une variable du λ-calcul, puis retourne 0; `longjmp` appliqué à k et à une valeur d réinstalle k comme contexte d'évaluation courant, ce qui à pour effet de faire un saut non local à l'endroit du programme où se trouve le `setjmp` qui à défini k, et de faire retourner la valeur d par ce dernier. Ceci permet de gérer des conditions d'erreur, par ex:

```
x = setjmp (k);
if (x==0) /* retour normal */
  {
    /* faire un certain nombre de calculs, eventuellement imbriques,
       qui arrivent a la ligne suivante : */
    if (erreur)
      longjmp (k, errno); /* longjmp ne retourne pas,
                             et saute a l'endroit du setjmp ci-dessus */
    /* si pas d'erreur, continuer... */
```

```
  }
else
  { /* ici, on est revenu sur erreur, avec x==errno */
    printf ("Erreur no. %d.\n", x);
  }
```

Comme beaucoup de constructions en C, celle-ci manque de l'élégance des constructions similaires de langages fonctionnels, et la plus proche est une construction, appelée `call-with-current-continuation` ou `call/cc`, et qui a été introduite dans le langage Scheme, un dialecte Lisp. (Le manque d'élégance de la construction C est due à deux choses. D'une part, si `errno` vaut 0, on ne peut pas distinguer un retour normal de `setjmp` d'un retour sur erreur dans l'exemple ci-dessus. D'autre part, `longjmp` a une sémantique indéfinie s'il est appelé après que la fonction dans laquelle `setjmp` a été appelé a retourné.)

L'idée est simple: le concept concret de contexte d'évaluation courant n'est rien d'autre que la continuation κ ! On peut donc définir en première approche une construction $\texttt{callcc}\ k\ \texttt{in}\ u$ qui capture à la façon de `setjmp` le contexte (la continuation) courante, la met dans k et évalue u; et une construction $\texttt{throw}\ k\ v$ qui évalue k et v, et réinstalle la continuation k pour faire retourner la valeur de v par l'instruction `callcc` qui a défini k. Sémantiquement:

$$
\begin{aligned}
[\![\texttt{callcc}\ k\ \texttt{in}\ u]\!]\rho\kappa &\mathrel{\hat{=}} [\![u]\!](\rho[k := \kappa])\kappa \\
[\![\texttt{throw}\ k\ v]\!]\rho\kappa &\mathrel{\hat{=}} [\![k]\!]\rho(\kappa' \mapsto [\![v]\!]\rho\kappa')
\end{aligned}
$$

Remarquer que dans la définition de `throw`, la continuation κ n'intervient pas dans le côté droit de l'égalité: elle est tout simplement ignorée, car c'est la continuation κ', valeur de k, qui va être utilisée pour continuer le calcul. Le fait que κ n'intervienne pas signifie que $\texttt{throw}\ k\ v$ ne retourne jamais.

On peut tester sur quelques exemples que le comportement est bien le comportement attendu:

Exercice 32 *Montrer que* `callcc k in 3` *retourne* 3*; autrement dit, en supposant que l'on a une constante* `3` *telle que* $[\![\texttt{3}]\!]\rho\kappa = \kappa(3)$*, montrer que* $[\![\texttt{callcc k in 3}]\!]\rho\kappa = \kappa(3)$*.*

Exercice 33 *Selon cette sémantique, que valent* `callcc k in throw k 3`*,* `callcc k in (throw k 3) 2`*,* `callcc k in 1+callcc k' in throw k' (throw k 3)` *? (On supposera que* `1` *et l'addition ont leurs sémantiques naturelles, que l'on précisera.)*

Mais en fait, non, le comportement des continuations n'est pas le comportement attendu en général. Considérons le programme $\texttt{callcc}\ k\ \texttt{in}\ (\lambda x \cdot \texttt{throw}\ k\ (\lambda y \cdot y))\texttt{3}$. On s'attend à ce qu'il ait la même sémantique que le β-réduit $\texttt{callcc}\ k\ \texttt{in throw}\ k\ (\lambda y \cdot y)$, autrement dit qu'il retourne la fonction identité, mais ce n'est pas le cas. En effet, formellement:

$$
\begin{aligned}
& [\![\texttt{callcc}\ k\ \texttt{in}\ (\lambda x \cdot \texttt{throw}\ k\ (\lambda y \cdot y))\texttt{3}]\!]\rho\kappa \\
= \ & [\![(\lambda x \cdot \texttt{throw}\ k\ (\lambda y \cdot y))\texttt{3}]\!](\rho[k := \kappa])\kappa \\
= \ & [\![\lambda x \cdot \texttt{throw}\ k\ (\lambda y \cdot y)]\!](\rho[k := \kappa])(f \mapsto [\![\texttt{3}]\!](\rho[k := \kappa])(d \mapsto \kappa(f(d)))) \\
= \ & [\![\lambda x \cdot \texttt{throw}\ k\ (\lambda y \cdot y)]\!](\rho[k := \kappa])(f \mapsto \kappa(f(3))) \\
= \ & (f \mapsto \kappa(f(3)))(d \mapsto [\![\texttt{throw}\ k\ (\lambda y \cdot y)]\!](\rho[k := \kappa, x := d])id) \\
= \ & \kappa([\![\texttt{throw}\ k\ (\lambda y \cdot y)]\!](\rho[k := \kappa, x := 3])id) \\
= \ & \kappa([\![k]\!](\rho[k := \kappa, x := 3])(\kappa' \mapsto [\![\lambda y \cdot y]\!](\rho[k := \kappa, x := 3])\kappa')) \\
= \ & \kappa((\kappa' \mapsto [\![\lambda y \cdot y]\!](\rho[k := \kappa, x := 3])\kappa')\kappa) \\
= \ & \kappa([\![\lambda y \cdot y]\!](\rho[k := \kappa, x := 3])\kappa) \\
= \ & \kappa(\kappa(d \mapsto [\![y]\!](\rho[k := \kappa, x := 3, y := d])id)) \\
= \ & \kappa(\kappa(d \mapsto d))
\end{aligned}
$$

alors que:

$$\begin{array}{rl} & [\![\texttt{callcc}\ k\ \texttt{in}\ \texttt{throw}\ k\ (\lambda y\cdot y)]\!]\rho\kappa \\ = & [\![\texttt{throw}\ k\ (\lambda y\cdot y)]\!](\rho[k:=\kappa])\kappa \\ = & [\![k]\!](\rho[k:=\kappa])(\kappa'\mapsto[\![\lambda y\cdot y]\!](\rho[k:=\kappa])\kappa') \\ = & (\kappa'\mapsto[\![\lambda y\cdot y]\!](\rho[k:=\kappa])\kappa')\kappa \\ = & [\![\lambda y\cdot y]\!](\rho[k:=\kappa])\kappa \\ = & \kappa(d\mapsto[\![y]\!](\rho[k:=\kappa,y:=d])id) \\ = & \kappa(d\mapsto d) \end{array}$$

Non seulement les deux valeurs sémantiques ne sont pas identiques — donc nous n'avons plus un modèle correct de la β-réduction —, mais celle de $\texttt{callcc}\ k\ \texttt{in}\ (\lambda x\cdot\texttt{throw}\ k\ (\lambda y\cdot y))\texttt{3}$ est en fait parfaitement absurde: $\kappa(\kappa(d\mapsto d))$ signifie que si κ est la suite du programme, au lieu de retourner simplement la fonction $d\mapsto d$, on va retourner $d\mapsto d$, effectuer la suite du programme, et une fois que le programme aura terminé et retourné une valeur d', on réexécutera la même suite κ du programme cette fois-ci sur la valeur d' !

Le problème est dans la façon dont nous avons défini la sémantique de l'abstraction en figure 5, bien que cela ne soit pas facile à voir sur l'exemple. Raisonnons donc par analogie, en identifiant continuation et pile courante (plus compteur de programme). La sémantique de la figure 5 demande à évaluer $\lambda x\cdot u$ dans la pile κ en retournant une fonction qui doit exécuter u dans la pile id, mais c'est absurde: si jamais la fonction doit être appelée un jour dans le contexte d'une autre pile κ', alors u doit être évalué dans la pile κ'. Il est donc nécessaire de modifier la définition de sorte qu'elle prenne en paramètre la pile κ' de l'appelant en plus de la valeur d de l'argument x:

$$[\![\lambda x\cdot u]\!]\rho\kappa\hat{=}\kappa(\kappa',d\mapsto[\![u]\!](\rho[x:=d])\kappa')$$

et la sémantique de l'application uv doit tenir compte de ce changement: la valeur f de u est maintenant une fonction qui prend non seulement la valeur d de v en argument, mais aussi la pile courante κ; $f(\kappa,d)$ calcule alors la valeur u avec x valant d dans la continuation κ. En particulier, comme κ est passée à f, c'est $f(\kappa,d)$ qui sera chargée de calculer tout le reste du programme, et on n'a donc pas besoin de calculer $\kappa(f(\kappa,d))$. On pose alors:

$$[\![uv]\!]\rho\kappa\hat{=}[\![u]\!]\rho(f\mapsto[\![v]\!]\rho(d\mapsto f(\kappa,d)))$$

En résumé, on a obtenu la sémantique de la figure 6.

$$\begin{array}{rcl} [\![x]\!]\rho\kappa & \hat{=} & \kappa(\rho(x)) \\ [\![uv]\!]\rho\kappa & \hat{=} & [\![u]\!]\rho\left(f\mapsto[\![v]\!]\rho\left(d\mapsto\begin{cases} f(\kappa,d) & \text{si } f \text{ est une fonction binaire} \\ \bot & \text{sinon}\end{cases}\right)\right) \\ [\![\lambda x\cdot u]\!]\rho\kappa & \hat{=} & \kappa((\kappa',d)\mapsto[\![u]\!](\rho[x:=d])\kappa') \\ [\![\texttt{callcc}\ k\ \texttt{in}\ u]\!]\rho\kappa & \hat{=} & [\![u]\!](\rho[k:=\kappa])\kappa \\ [\![\texttt{throw}\ k\ v]\!]\rho\kappa & \hat{=} & [\![k]\!]\rho(\kappa'\mapsto[\![v]\!]\rho\kappa') \end{array}$$

Figure 6: Une sémantique en passage par valeur avec des sauts non locaux

Exercice 34 *On pose* $\texttt{let}\ x=u\ \texttt{in}\ v\hat{=}(\lambda x\cdot v)u$. *Calculer* $[\![\texttt{let}\ x=u\ \texttt{in}\ v]\!]\rho\kappa$.

Exercice 35 *On appelle* valeur au sens de Plotkin*, ou* P-valeur*, les λ-termes qui sont soit des variables soit des abstractions. Montrer que si V est une P-valeur, $[\![(\lambda x\cdot u)V]\!]\rho\kappa=[\![u[x:=V]]\!]\rho\kappa$. (On dit que la règle (β_v): $(\lambda x\cdot u)V\to u[x:=V]$ est valide.) Montrer que ce résultat ne se généralise pas au cas où V n'est pas une P-valeur.*

En Scheme, les constructions $\texttt{callcc}\ldots\texttt{in}$ et $\texttt{throw}$ ne sont pas présentes telles quelles. Comme k est une variable liée dans $\texttt{callcc}\ k\ \texttt{in}\ u$, on peut en effet s'arranger pour la lier via un λ, et définir $\texttt{callcc}\, k\, \texttt{in}\, u = \texttt{call/cc}(\lambda x \cdot u)$, où $\texttt{call/cc}$ est un opérateur primitif du langage. La sémantique (provisoire) de $\texttt{call/cc}$ est:

$$[\![\texttt{call/cc}]\!]\rho\kappa \hat{=} \kappa((\kappa', f) \mapsto f(\kappa', \kappa'))$$

En fait, il est traditionnel lorsqu'on étudie les opérateurs de capture de continuation de faire de la valeur de la variable k de continuation directement une valeur de fonction. En effet, la seule utilisation possible d'une continuation stockée dans k est d'appeler $\texttt{throw}\ k\ v$. Posons $k' \hat{=} \texttt{throw}\ k \hat{=} \lambda x \cdot \texttt{throw}\ k\ x$: la valeur de k' est la seule chose que $\texttt{call/cc}$ a besoin de stocker. On définit donc (définitivement):

$$[\![\texttt{call/cc}]\!]\rho\kappa \hat{=} \kappa((\kappa', f) \mapsto f(\kappa', (\_, d') \mapsto \kappa'(d')))$$

où $(\_, d') \mapsto \kappa'(d')$ est la fonction qui prend une continuation κ'', une valeur d' et retourne $\kappa'(d')$ après avoir jeté κ''.

Exercice 36 *Montrer que, si k est une variable telle que $\rho(k) = f$, alors $[\![\lambda x \cdot \texttt{throw}\ k\ x]\!]\rho\kappa = \kappa((\_, d) \mapsto f(d))$.*

Exercice 37 *Vérifier que, si k n'est pas libre dans u, alors $[\![\texttt{call/cc}(\lambda k' \cdot u)]\!]\rho\kappa = [\![\texttt{callcc}\ k\ \texttt{in}\ u[k' := \lambda x \cdot \texttt{throw}\ k\ x]]\!]\rho\kappa$. (Pour faciliter le calcul, on remarquera que $\lambda x \cdot \texttt{throw}\ k\ x$ est une P-valeur, donc par l'exercice 35, il suffit de montrer que $[\![\texttt{call/cc}(\lambda k' \cdot u)]\!]\rho\kappa = [\![\texttt{callcc}\ k\ \texttt{in}\ (\lambda k' \cdot u)(\lambda x \cdot \texttt{throw}\ k\ x)]\!]\rho\kappa$.)*

Exercice 38 *L'opérateur* $\texttt{call/cc}$ *connaît de nombreuses variantes, notamment l'opérateur $\mathcal{C}$ de Felleisen, dont la sémantique est:*

$$[\![\mathcal{C}]\!]\rho\kappa \hat{=} \kappa((\kappa', f) \mapsto f(\_ \mapsto \bot, (\_, d') \mapsto \kappa'(d')))$$

où $\_ \mapsto \bot$ est la fonction qui retourne toujours $\bot$. Montrer que, pour tout t:

$$[\![\mathcal{C}t]\!]\rho\kappa = [\![t]\!]\rho(f \mapsto f(\_ \mapsto \bot, (\_, d') \mapsto \kappa(d')))$$

Exercice 39 *On dit que u* retourne normalement *dans l'environnement ρ si et seulement si, pour toute continuation κ, $[\![u]\!]\rho\kappa = \kappa(d)$ pour une certaine valeur d, appelée* la valeur *de u.*

Montrer que si u retourne normalement dans $\rho[k := (\_, d') \mapsto \kappa(d')]$, alors $[\![\mathcal{C}(\lambda k \cdot u)]\!]\rho\kappa = \bot$. (Autrement dit, si u retourne normalement, alors $\mathcal{C}(\lambda k \cdot u)$ plante.)

Exercice 40 *Montrer que l'on peut réaliser* $\texttt{call/cc}$ *à l'aide de $\mathcal{C}$. Plus précisément, on montrera que $\texttt{call/cc}(\lambda k \cdot u)$ et $\mathcal{C}(\lambda k \cdot ku)$ ont la même sémantique.*

L'opérateur $\mathcal{C}$ est souvent plus pratique pour les manipulations formelles que $\texttt{call/cc}$. Faisons quelques calculs, dans l'espoir de découvrir quelques égalités sémantiques entre termes. Notamment, que vaut $\mathcal{C}(\lambda k \cdot t)$ appliqué à v ? Intuitivement, on va récupérer dans k la continuation consistant à appliquer la valeur f de t à la valeur d de v, et ensuite appliquer la continuation courante κ à $f(d)$. Mais on peut récupérer la continuation courante dans une variable k' en écrivant $\mathcal{C}(\lambda k' \cdot \ldots)$, et alors le calcul consistant à calculer t avec comme continuation la fonction qui prend la valeur f de t, l'applique à v et continue le calcul (par k') est donc $t[k := \lambda f \cdot k'(fv)]$ — ceci du moins tant que v est une P-valeur. Ceci suggère que $\mathcal{C}(\lambda k \cdot t)v = \mathcal{C}(\lambda k' \cdot t[k := \lambda f \cdot k'(fv)]$, au moins si v est une P-valeur. Plus généralement, posons $u \hat{=} \lambda k \cdot t$, et comparons les deux quantités $\mathcal{C}uv$ et $\mathcal{C}(\lambda k' \cdot u(\lambda f \cdot k'(fv)))$:

$$\begin{aligned}
[\![\mathcal{C}uv]\!]\rho\kappa = {} & [\![\mathcal{C}u]\!]\rho(g \mapsto [\![v]\!]\rho(d \mapsto g(\kappa, d))) \\
= {} & [\![u]\!]\rho(f \mapsto f(_ \mapsto \bot, (_, d') \mapsto (g \mapsto [\![v]\!]\rho(d \mapsto g(\kappa, d)))(d'))) \\
& \text{par l'exercice 38} \\
= {} & [\![u]\!]\rho(f \mapsto f(_ \mapsto \bot, (_, d') \mapsto [\![v]\!]\rho(d \mapsto d'(\kappa, d))))
\end{aligned}$$

Tandis que:

$$
\begin{array}{rl}
& [\![\mathcal{C}(\lambda k' \cdot u(\lambda f \cdot k'(fv)))]\!]\rho\kappa \\
= & [\![\lambda k' \cdot u(\lambda f \cdot k'(fv))]\!]\rho(f \mapsto f(\_ \mapsto \bot, (\_, d') \mapsto \kappa(d'))) \\
= & (f \mapsto f(\_ \mapsto \bot, (\_, d') \mapsto \kappa(d')))((\kappa', d'') \mapsto [\![u(\lambda f \cdot k'(fv))]\!](\rho[k' := d''])\kappa') \\
= & [\![u(\lambda f \cdot k'(fv))]\!](\rho[k' := (\_, d') \mapsto \kappa(d')])(\_ \mapsto \bot) \\
= & [\![u]\!](\rho[k' := (\_, d') \mapsto \kappa(d')])(g \mapsto [\![\lambda f \cdot k'(fv)]\!](\rho[k' := (\_, d') \mapsto \kappa(d')])(d'' \mapsto g(\_ \mapsto \bot, d''))) \\
= & [\![u]\!]\rho(g \mapsto [\![\lambda f \cdot k'(fv)]\!](\rho[k' := (\_, d') \mapsto \kappa(d')])(d'' \mapsto g(\_ \mapsto \bot, d''))) \\
& \text{à condition que } k' \text{ ne soit pas libre dans } u \\
= & [\![u]\!]\rho(g \mapsto (d'' \mapsto g(\_ \mapsto \bot, d''))((\kappa', g') \mapsto [\![k'(fv)]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])\kappa')) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![k'(fv)]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])\kappa')) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![k']\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g']) \\
& \qquad\qquad (f' \mapsto [\![fv]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])(d \mapsto f'(\kappa', d))))) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto \\
& \qquad\qquad (f' \mapsto [\![fv]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])(d \mapsto f'(\kappa', d)))((\_, d') \mapsto \kappa(d')))) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![fv]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])(d \mapsto \kappa(d)))) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![f]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g']) \\
& \qquad\qquad (f' \mapsto [\![v]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])(d'' \mapsto f'(d \mapsto \kappa(d), d''))))) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto (f' \mapsto [\![v]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])(d'' \mapsto f'(d \mapsto \kappa(d), d'')))(g'))) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![v]\!](\rho[k' := (\_, d') \mapsto \kappa(d'), f := g'])(d'' \mapsto g'(d \mapsto \kappa(d), d'')))) \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![v]\!]\rho(d'' \mapsto g'(d \mapsto \kappa(d), d'')))) \\
& \text{à condition que } k' \text{ et } f \text{ ne soient pas libres dans } v \\
= & [\![u]\!]\rho(g \mapsto g(\_ \mapsto \bot, (\kappa', g') \mapsto [\![v]\!]\rho(d'' \mapsto g'(\kappa, d'')))) \\
& \text{car } (d \mapsto \kappa(d)) = \kappa
\end{array}
$$

On a donc démontré:

Lemme 7 $[\![\mathcal{C}uv]\!]\rho\kappa = [\![\mathcal{C}(\lambda k' \cdot u(\lambda f \cdot k'(fv)))]\!]\rho\kappa$, *où* k' *et* f *sont deux variables nouvelles.*

Notons que ceci est correct même quand v n'est pas une P-valeur. On peut directement formaliser le comportement de l'opérateur $\mathcal{C}$ par des règles de réécriture inspirées de la sémantique:

$$
\begin{array}{lccl}
(\mathcal{C}_L) & \mathcal{C}uv & \to & \mathcal{C}(\lambda k' \cdot u(\lambda f \cdot k'(fv))) \\
(\mathcal{C}_R) & V(\mathcal{C}u) & \to & \mathcal{C}(\lambda k' \cdot u(\lambda x \cdot k'(Vx)))
\end{array}
$$

où dans la seconde, V est une P-valeur.

Exercice 41 ( ) *Montrer que la règle* $(\mathcal{C}_R)$ *est valide. (On utilisera le fait que toute P-valeur* V *retourne normalement, cf. exercice 39, et en fait que* $[\![V]\!]\rho\kappa = \kappa(d)$ *pour une valeur* d *qui ne dépend que de* V *et de* ρ*.) Montrer que l'hypothèse selon laquelle* V *est une P-valeur est essentielle.*

Nous avons ignoré les problèmes de définition de domaine de valeurs. Mais ceci se règle aisément. Nous avons besoin d'un domaine D, les continuations κ seront des éléments de $[D \to D]$, et les valeurs des fonctions seront des éléments de $[[D \to D] \times D \to D]$. Il suffit donc de prendre un domaine D tel que:

$$D = (\mathbb{N} \cup [[D \to D] \times D \to D])_\bot$$

à isomorphisme près, et où $\mathbb{N}$ peut être remplacé par n'importe quel autre ensemble de valeurs de base. (Noter que les continuations ne sont pas incluses dans le domaine des valeurs, car les continuations capturées par $\mathcal{C}$ sont codées comme des fonctions dans $[[D \to D] \times D \to D]$.)

5.3 Ajout d'effets de bord

Nous allons maintenant voir quelques conséquences intéressantes de cette façon de voir la sémantique des langages en appel par valeur.

La première est qu'il est maintenant facile de donner une sémantique à un langage enrichi d'effets de bord, sous la forme d'affectations. Voici, dans le style de ML, les constructions que nous voulons ajouter: certains termes dénoteront des *références*, c'est-à-dire des adresses en mémoire pointant vers des valeurs; la construction $!u$ retourne la valeur stockée à l'adresse u, la construction $u := v$ stocke la valeur de v à l'adresse u, et enfin `ref` u alloue une adresse l, y stocke la valeur de u, et retourne l. Tant qu'à imiter les langages impératifs comme Pascal ou C, nous allons aussi introduire une construction de séquencement $u; v$ qui évalue u, puis évalue et retourne la valeur de v.

Pour donner une sémantique formelle à ces constructions, on peut modifier notre fonction $[\![\_]\!]$ de sorte qu'elle prenne maintenant non seulement un environnement ρ, une continuation κ, mais aussi une *mémoire* $\sigma \in [L_\bot \to D]$, où L est un ensemble d'*adresses* ("locations" en anglais).

Noter que les continuations doivent maintenant non seulement prendre en argument la valeur calculée par un terme u, mais aussi le nouvel état de la mémoire. En résumé, on aura les conventions de typage:

$$\begin{array}{rcl} \rho & \in & \mathcal{V} \to D \\ \sigma & \in & [L_\bot \to D] \\ \kappa & \in & [[L_\bot \to D] \times D \to A] \end{array}$$

où A est un domaine de *réponses*. Auparavant, A était juste le domaine D des valeurs, mais il est aussi envisageable de vouloir prendre pour A le produit du domaine D des valeurs avec celui, $[L_\bot \to D]$, des mémoires, par exemple, selon ce que l'on souhaite observer à la fin du calcul.

Les valeurs de fonctions devront maintenant prendre une continuation, une valeur pour son argument, *et* une mémoire courante, et retourner une réponse. Notre domaine sémantique doit aussi être enrichi par des adresses, et c'est pourquoi nous posons:

$$\begin{array}{rcl} D & \hat{=} & (\mathbb{N} \oplus L \oplus [Cont \times Mem \times D \to A])_\bot \\ Cont & \hat{=} & [Mem \times D \to A] \\ Mem & \hat{=} & [L_\bot \to D] \end{array}$$

où $\mathbb{N}$ pourrait être remplacé par n'importe quel autre ensemble de valeurs de base. On peut alors définir notre sémantique comme en figure 7. (Exercice: vérifier que la définition est cohérente avec le typage donné par les équations de domaine pour D, $Cont$ et Mem.)

Exercice 42 *Commenter la sémantique de la figure 6 en langage courant. En particulier, dire dans quel ordre cette sémantique spécifie que doivent être évaluées les expressions, dans quelles continuations et quelles mémoires elles sont évaluées.*

On peut bien sûr encore donner une sémantique à l'opérateur $\mathcal{C}$ dans ce contexte:

$$[\![\mathcal{C}]\!]\rho\sigma\kappa \hat{=} \kappa(\sigma, (\kappa', \sigma', f) \mapsto f(\_ \mapsto \bot, \sigma', (\_, \sigma'', d') \mapsto \kappa'(\sigma'', d')))$$

Exercice 43 *Soit $*$ un identificateur distingué (le* handler d'exception courant*). On définit les constructions suivantes:*

$$\begin{array}{rcl} \texttt{try}\ u\ \texttt{with}\ x \Rightarrow v & \hat{=} & \texttt{let}\ k_0 = !*\ \texttt{in} \\ & & \quad \mathcal{C}(\lambda k \cdot * := (\lambda x \cdot * := k_0; kv); \\ & & \quad\quad k(\texttt{let}\ y = u\ \texttt{in}\ * := k_0; y)) \\ \texttt{raise}\ v & \hat{=} & !*\,v \end{array}$$

$$
\begin{array}{rcl}
[\![x]\!]\rho\sigma\kappa & \hat{=} & \kappa(\sigma, \rho(x)) \\
[\![uv]\!]\rho\sigma\kappa & \hat{=} & [\![u]\!]\rho\sigma\left((\sigma', f) \mapsto [\![v]\!]\rho\sigma'\left((\sigma'', d) \mapsto \begin{cases} f(\kappa, \sigma'', d) & \text{si } f \text{ est une fonction} \\ \bot & \text{sinon} \end{cases}\right)\right) \\
[\![\lambda x \cdot u]\!]\rho\sigma\kappa & \hat{=} & \kappa(\sigma, (\kappa', \sigma', d) \mapsto [\![u]\!](\rho[x := d])\sigma'\kappa') \\
[\![!u]\!]\rho\sigma\kappa & \hat{=} & [\![u]\!]\rho\sigma\left((\sigma', l) \mapsto \begin{cases} \kappa(\sigma', \sigma'(l)) & \text{si } l \text{ est dans le domaine de } \sigma' \\ \bot & \text{sinon} \end{cases}\right) \\
[\![u := v]\!]\rho\sigma\kappa & \hat{=} & [\![u]\!]\rho\sigma\left((\sigma', l) \mapsto [\![v]\!]\rho\sigma'\left((\sigma'', d) \mapsto \begin{cases} \kappa(\sigma''[l := d], d) & \text{si } l \in L \\ \bot & \text{sinon} \end{cases}\right)\right) \\
[\![\texttt{ref } u]\!]\rho\sigma\kappa & \hat{=} & [\![u]\!]\rho\sigma((\sigma', d) \mapsto \kappa(\sigma'[l := d], l)) \qquad (l \text{ hors du domaine de } \sigma') \\
[\![u; v]\!]\rho\sigma\kappa & \hat{=} & [\![u]\!]\rho\sigma((\sigma', \_) \mapsto [\![v]\!]\rho\sigma'\kappa)
\end{array}
$$

Figure 7: Une sémantique en passage par valeur avec affectations

(Voir l'exercice 34 pour la construction `let`*.) Justifier intuitivement pourquoi il s'agit bien d'une gestion d'exceptions à la OCaml — et sous quelles conditions.*

Alors qu'il était possible, quoique difficile, de trouver une sémantique par réduction pour $\mathcal{C}$ (les règles $\mathcal{L}_L$ et $\mathcal{L}_R$), il est pratiquement infaisable d'en trouver une satisfaisante pour les affectations.

References

[Bar84] Henk Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, Amsterdam, 1984.

A Modèles D_∞

Fixons un cpo $(D_0, \leq_0)$, et construisons une suite de cpo $(D_k, \leq_k)$ par:

$$D_{k+1} \hat{=} [D_k \to D_k]$$

avec $\leq_{k+1}$ l'ordre point à point, soit: pour tous $f, g \in [D_k \to D_k]$, $f \leq_{k+1} g$ si et seulement si $f(x) \leq_k g(x)$ pour tout $x \in D_k$.

On peut construire une fonction $r_0 : [D_0 \to D_0] \to D_0$ par: $r_0(f) = f(\bot_0)$; et on peut construire une fonction $i_0 : D_0 \to [D_0 \to D_0]$ par: $i_0(x) = (v \mapsto x)$ (une fonction constante). Par récurrence, on définit $r_{k+1} : [D_{k+1} \to D_{k+1}] \to D_{k+1}$ par: $r_{k+1}(f) \hat{=} r_k \circ f \circ i_k$; et $i_{k+1} : D_{k+1} \to [D_{k+1} \to D_{k+1}]$ par: $i_{k+1}(x) \hat{=} i_k \circ x \circ r_k$. On a donc un diagramme infini:

$$
\begin{tikzcd}
D_0 \arrow[r, shift left, "i_0"] & D_1 \arrow[l, shift left, "r_0"] \arrow[r, shift left, "i_1"] & D_2 \arrow[l, shift left, "r_1"] \arrow[r, shift left, "i_2"] & \cdots \arrow[l, shift left, "r_2"] \arrow[r, shift left, "i_k"] & D_{k+1} \arrow[l, shift left, "r_k"] \arrow[r, shift left, "i_k+1"] & \cdots \arrow[l, shift left, "r_{k+1}"]
\end{tikzcd}
$$

avec $r_k \circ i_k$ l'identité sur D_k. (On remarquera que cette composition est dans le mauvais sens, vu qu'on vu obtenir $i \circ r = id$ au final. On verra plus tard comment ceci se résout.)

Exercice 44 *Montrer par récurrence sur k que r_k et i_k sont continues.*

Posons, pour $k \leq l$, $r_{kl} : D_l \to D_k \hat{=} r_k \circ r_{k+1} \circ \ldots \circ r_{l-1}$. Le système formé par les D_k et les r_{kl}, $k \leq l$, est appelé un *système projectif* de cpo. Intuitivement, ceci revient à voir les D_l se projeter via r_{kl} dans D_k. On peut alors calculer la *limite projective* de ce système, qui est un ensemble le plus petit possible D_∞ se projetant sur tous les D_k via des fonctions $r_{k\infty}$:

Définition 6 *On pose:*

$$\begin{array}{rcl} D_\infty & \hat{=} & \{(d_k)_{k \in \mathbb{N}} \mid \forall k \leq l \cdot d_k = r_{kl}(d_l)\} \\ r_{k\infty}(d_0, d_1, \ldots, d_k, \ldots) & \hat{=} & d_k \\ (d_k)_{k \in \mathbb{N}} \leq_\infty (d'_k)_{k \in \mathbb{N}} & \Leftrightarrow & \forall k \in \mathbb{N} \cdot d_k \leq_k d'_k \end{array}$$

Une caractérisation équivalente est:

$$D_\infty = \{(d_k)_{k \in \mathbb{N}} \mid \forall k \cdot d_k = r_k(d_{k+1})\}$$

Lemme 8 *$(D_\infty, \leq_\infty)$ est un cpo, $r_{k\infty} : D_\infty \to D_k$ est continue pour tout k, et $r_{kl} \circ r_{l\infty} = r_{k\infty}$ pour tout $k \leq l$.*

Preuve : $(D_\infty, \leq_\infty)$ est clairement un ordre partiel, et si $(d^i)_{i \in \mathbb{N}}$ est une chaîne, avec $d^i = (d^i_j)_{j \in \mathbb{N}}$, alors sa borne supérieure $\bigsqcup_i d^i$ ne peut être que $d \hat{=} (\bigsqcup_i d^i_0, \bigsqcup_i d^i_1, \ldots)$; il ne reste qu'à montrer que $d \in D_\infty$, autrement dit que $r_{kl}(\bigsqcup_i d^i_l) = \bigsqcup_i d^i_k$ pour tout $k \leq l$. Ceci revient à montrer que $r_k(\bigsqcup_i d^i_{k+1}) = \bigsqcup_i d^i_k$ pour tout k, sachant que $d^i_k = r_k(d^i_{k+1})$; mais r_k est continue, par l'exercice 44.

Il est pratiquement évident que $r_{k\infty}$ est continue: $r_{k\infty}(\bigsqcup_i d^i) = r_{k\infty}(\bigsqcup_i d^i_0, \bigsqcup_i d^i_1, \ldots) = \bigsqcup_i d^i_k = \bigsqcup_i r_{k\infty}(d^i)$. Finalement, $r_{kl} \circ r_{l\infty} = r_{k\infty}$ pour tout $k \leq l$ par les propriétés générales des limites projectives (mais vous pouvez le vérifier directement). $\diamondsuit$

Donc D_∞ se projette sur chaque D_k via les $r_{k\infty}$. Ceci prend en compte la structure formée à partir des projections r_k. On peut aussi tenir compte des i_k, et définir la construction duale: posons, pour $k \leq l$, $i_{kl} : D_k \to D_l \hat{=} i_{l-1} \circ \ldots \circ i_{k+1} \circ i_k$. (Noter que i_{kl} va de D_k dans D_l, alors que r_{kl} va de D_l dans D_k !) Le système formé par les D_k et les i_{kl}, $k \leq l$, est appelé un *système inductif* de cpo. Ceci revient à voir non pas D_l se projeter dans D_k, mais D_k s'injecter dans D_l via i_{kl}.

On pourrait calculer la *limite inductive* de ce système, qui est en un sens le plus grand ensemble s'injectant dans tous les D_k, et qui est construit comme le quotient de la somme directe des D_k (dont les éléments sont des couples (k, d_k) avec $d_k \in D_k$) par la relation d'équivalence $\sim$ engendrée par $(k, d_k) \sim (l, d_l)$ si $k \leq l$ et $i_{kl}(d_k) = d_l$. Il se trouve que dans ce cas précis, la limite inductive des D_k est exactement la même que la limite projective D_∞ (ce n'est en général pas le cas des limites injectives et projectives); c'est pourquoi nous n'étudierons pas cette limite inductive en tant que telle.

Lemme 9 *Soit $i_{k\infty} : D_k \to D_\infty$ définie par:*

$$i_{k\infty}(d) \hat{=} (r_{0k}(d), r_{1k}(d), \ldots, r_{(k-1)k}(d), d, i_{k(k+1)}(d), i_{k(k+2)}(d), \ldots)$$

pour tout $d \in D_k$. Alors $i_{k\infty}$ est continue, et $i_{l\infty} \circ i_{kl} = i_{k\infty}$ pour tout $k \leq l$. De plus, $r_{k\infty} \circ i_{k\infty}$ est l'identité sur D_k.

Preuve : La fonction $i_{k\infty}$ est bien définie, au sens où $i_{k\infty}(d)$ est bien dans D_∞, c'est-à-dire que $r_j(r_{(j+1)k}(d)) = r_{jk}(d)$ pour tout $j \leq k-2$ (par définition de r_{jk}), $r_{(k-1)k}(d) = r_{k-1}(d)$ (par définition), $d = r_k(i_{k(k+1)}(d))$ (car $i_{k(k+1)} = i_k$ et $r_k \circ i_k = id$), et $i_{kl}(d) = r_l(i_{k(l+1)}(d))$ pour tout $l \geq k+1$ (car $r_l \circ i_l = id$).

La continuité de $i_{k\infty}$ provient du fait que tous les r_{jk}, $j \leq k$ et tous les i_{kl}, $k \leq l$, sont continus, par l'exercice 44.

Pour vérifier que $i_{l\infty} \circ i_{kl} = i_{k\infty}$ pour tout $k \leq l$, on remarque d'abord que pour tous $j \leq k \leq l$:

$$\begin{array}{lll} r_{jk} \circ r_{kl} = r_{jl} & r_{jl} \circ i_{kl} = r_{jk} & r_{jj} = id \\ i_{kl} \circ i_{jk} = i_{jl} & r_{kl} \circ i_{jl} = i_{jk} & i_{jj} = id \end{array}$$

Les deux équations du milieu se démontrent en utilisant que $r_m \circ i_m = id$ pour tout m. Alors si $k \leq l$, pour tout $d \in D_k$ on a:

$$\begin{aligned}
(i_{l\infty} \circ i_{kl})(d) = & \ (r_{0l}(i_{kl}(d)), r_{1l}(i_{kl}(d)), \ldots, r_{(l-1)l}(i_{kl}(d)), i_{kl}(d), i_{l(l+1)}(i_{kl}(d)), i_{l(l+2)}(i_{kl}(d)), \ldots) \\
= & \ (r_{0l}(i_{kl}(d)), \ldots, r_{(k-1)l}(i_{kl}(d)), r_{kl}(i_{kl}(d)), r_{(k+1)l}(i_{kl}(d)), \ldots, r_{(l-1)l}(i_{kl}(d)), \\
& \ i_{kl}(d), i_{l(l+1)}(i_{kl}(d)), i_{l(l+2)}(i_{kl}(d)), \ldots) \\
= & \ (r_{0k}(d), r_{1k}(d), \ldots, r_{(k-1)k}(d), d, i_{k(k+1)}(d), \ldots, i_{k(l-1)}(d), i_{kl}(d), i_{k(l+1)}(d), i_{k(l+2)}(d), \ldots) \\
= & \ i_{k\infty}(d)
\end{aligned}$$

Finalement, $(r_{k\infty} \circ i_{k\infty})(d) = d$ par construction. ♢

Il s'ensuit que l'on peut considérer que D_k est inclus dans D_∞ à isomorphisme près: l'isomorphisme est $i_{k\infty}$ de D_k vers son image, et son inverse est la restriction de $r_{k\infty}$ à l'image de $i_{k\infty}$. En somme, D_∞ est une sorte d'union de tous les D_k; pour être précis, de complété de cette union: un élément $d \hat{=} (d_0, d_1, \ldots, d_k, \ldots)$ est en effet la limite (la borne supérieure) des d_k, $k \in \mathbb{N}$. C'est ce que dit le lemme suivant:

Lemme 10 *Pour tout $d \in D_\infty$, $d = \bigsqcup_k (i_{k\infty} \circ r_{k\infty})(d)$, et $(i_{k\infty} \circ r_{k\infty})_{k \in \mathbb{N}}$ est une chaîne.*

Preuve : Notons d'abord que: (a) pour tout k, pour tout $f \in D_{k+1}$, $i_k(r_k(f)) \leq_{k+1} f$. En effet, ceci signifie que pour tout $x \in D_k$, $i_k(r_k(f))(x) \leq_k f(x)$. Montrons-le par récurrence sur k. Pour $k = 0$, $i_0(r_0(f))(x) = i_0(f(\bot_0))(x) = f(\bot_0) \leq_0 f(x)$ puisque f est monotone. Sinon, $i_{k+1}(r_{k+1}(f))(x) = i_k(r_{k+1}(f)(r_k(x))) = i_k(r_k(f(i_k(r_k(x))))) \leq_k f(i_k(r_k(x)))$ (par récurrence) $\leq_k f(x)$ (par récurrence et monotonie de f).

Soit $d = (d_0, d_1, \ldots, d_k, \ldots) \in D_\infty$. On observe que: (b) $i_k(d_k) \leq_{k+1} d_{k+1}$ pour tout k. En effet, $d_k = r_k(d_{k+1})$ puisque $d \in D_\infty$ et on applique (a). Donc: (c) $i_{kl}(d_k) \leq_l d_l$ pour tout $k \leq l$. Alors:

$$\begin{aligned}
\textstyle\bigsqcup_k (i_{k\infty} \circ r_{k\infty})(d) = & \ \textstyle\bigsqcup_k (i_{k\infty}(d_k)) \\
= & \ \textstyle\bigsqcup_k (r_{0k}(d_k), r_{1k}(d_k), \ldots, r_{(k-1)k}(d_k), d_k, i_{k(k+1)}(d_k), i_{k(k+2)}(d_k), \ldots) \\
= & \ \textstyle\bigsqcup_k (d_0, d_1, \ldots, d_{k-1}, d_k, i_{k(k+1)}(d_k), i_{k(k+2)}(d_k), \ldots) \\
= & \ (\bigsqcup(d_0), \bigsqcup(i_{01}(d_0), d_1), \ldots, \bigsqcup(i_{0k}(d_0), i_{1k}(d_1), \ldots, i_{(k-1)k}(d_{k-1}), d_k), \ldots) \\
= & \ (d_0, d_1, \ldots, d_k, \ldots) = d
\end{aligned}$$

par la remarque (c).

Finalement, $(i_{k\infty} \circ r_{k\infty})_{k \in \mathbb{N}}$ est une chaîne car $k \leq l$ implique $i_{k\infty} \circ r_{k\infty} \leq i_{l\infty} \circ r_{l\infty}$. En effet:

$$\begin{aligned}
(i_{k\infty} \circ r_{k\infty})(d) = & \ (d_0, d_1, \ldots, d_{k-1}, d_k, i_{k(k+1)}(d_k), i_{k(k+2)}(d_k), \ldots, i_{kl}(d_k), i_{k(l+1)}(d_k), \ldots) \\
\leq & \ (d_0, d_1, \ldots, d_{k-1}, d_k, d_{k+1}, d_{k+2}, \ldots, d_l, i_{k(l+1)}(d_k), \ldots) \\
= & \ (i_{l\infty} \circ r_{l\infty})(d)
\end{aligned}$$

par (a). ♢

On peut maintenant définir i et r sur D_∞:

Définition 7 *Soit i la fonction de D_∞ vers $[D_\infty \to D_\infty]$ qui à $(y_0, y_1, \ldots, y_k, \ldots)$ associe la fonction qui à $(x_0, x_1, \ldots, x_k, \ldots) \in D_\infty$ associe $(y_1(x_0), y_2(x_1), \ldots, y_{k+1}(x_k), \ldots) \in D_\infty$.*

Soit r la fonction de $[D_\infty \to D_\infty]$ vers D_∞ qui à f associe $(d_0, d_1, \ldots, d_k, \ldots)$ défini par: $d_0 = r_{0\infty}(f(i_{0\infty}(\bot_0)))$, $d_{k+1} \in D_{k+1} = [D_k \to D_k]$ est la fonction $r_{k\infty} \circ f \circ i_{k\infty}$.

Lemme 11 *Les fonctions i et r de la définition 7 sont bien définies, continues. De plus, $i \circ r$ est l'identité sur $[D_\infty \to D_\infty]$ et $r \circ i$ est l'identité sur D_∞. En particulier, D_∞ et $[D_\infty \to D_\infty]$ sont isomorphes.*

Preuve : Pour montrer que i est bien définie, il faut montrer que pour tout $y \hat{=} (y_0, y_1, \ldots, y_k, \ldots)$ de D_∞, $i(y)$ en tant que fonction qui à $x = (x_0, \ldots, x_k, \ldots)$ associe $(y_1(x_0), y_2(x_1), \ldots, y_{k+1}(x_k), \ldots)$ est bien continue. Ceci est ne présente pas de difficulté, car les bornes supérieures sont prises composante par composante et chaque y_k est continue. De même, i est continue parce que $(\bigsqcup_j y_{k+1}^j)(x_k) = \bigsqcup_j (y_{k+1}^j(x_k))$ par définition. De même, r est bien définie car $r_0(d_1) = d_1(\bot_0) = r_{0\infty}(f(i_{0\infty}(\bot_0))) = d_0$, et $r_{k+1}(d_{k+2}) = r_k \circ d_{k+2} \circ i_k$ (par définition de r_{k+1}) $= r_k \circ r_{(k+1)\infty} \circ f \circ i_{(k+1)\infty} \circ i_k = r_{k\infty} \circ f \circ i_{k\infty} = d_{k+1}$, donc $r(d)$

est bien dans D_∞. De plus, r est continue parce que toutes les $r_{k\infty}$ et les $i_{k\infty}$ sont continues, et que la composition $\circ$ est continue.

Avant de continuer, remarquons que dans un cpo: (a) si on a une suite $a_{jj'}$ telle que pour tous j, j', il existe un k tel que $a_{jj'} \leq a_{kk}$, alors $\bigsqcup_{j,j'} a_{jj'} = \bigsqcup_k a_{kk}$. En effet $\bigsqcup_k a_{kk} \leq \bigsqcup_{j,j'} a_{jj'}$ parce que tous les a_{kk} sont des $a_{jj'}$, et réciproquement $\bigsqcup_{j,j'} a_{jj'} \leq \bigsqcup_{k_{jj'}} a_{k_{jj'}k_{jj'}}$ (où $k_{jj'}$ est un k tel que $a_{jj'} \leq a_{kk}$, pour tous j, j') $\leq \bigsqcup_k a_{kk}$.

Soit f une fonction continue f de D_∞ dans D_∞, $d \in D_\infty$ et posons $a_{jj'} \hat{=} (i_{j\infty} \circ r_{j\infty})(f((i_{j'\infty} \circ r_{j'\infty})(d)))$. Alors comme $i_{j\infty} \circ r_{j\infty}$ forme une chaîne (lemme 10) et que f est monotone, $a_{jj'} \leq a_{kk}$ pour $k = \max(j, j')$. Donc par (a), on a:

$$\begin{array}{rcl}\bigsqcup_k i_{k\infty}(r_{k\infty}(f(i_{k\infty}(r_{k\infty}(d))))) & = & \bigsqcup_{j,j'} (i_{j\infty} \circ r_{j\infty})(f((i_{j'\infty} \circ r_{j'\infty})(d))) \\ & = & \bigsqcup_j (i_{j\infty} \circ r_{j\infty})(f(\bigsqcup_{j'} (i_{j'\infty} \circ r_{j'\infty})(d))) \\ & & \text{par continuité de } i_{j\infty} \circ r_{j\infty} \text{ et de } f \\ & = & \bigsqcup_j (i_{j\infty} \circ r_{j\infty})(f(d)) = f(d) \qquad \text{par le lemme 10}\end{array}$$

En résumé: (b) $f(d) = \bigsqcup_k i_{k\infty}(r_{k\infty}(f(i_{k\infty}(r_{k\infty}(d)))))$.

Calculons maintenant $i \circ r$. Pour toute $f \in [D_\infty \to D_\infty]$, montrons que $i(r(f)) = f$. Pour ceci, il suffit de montrer que $i(r(f))(d) = f(d)$ pour tout $d \in D_\infty$. Remarquons d'abord que: (c) $i(r(f))(d) = \bigsqcup_k i_{k\infty}(r_{k\infty}(i(r(f))(d))$ par le lemme 10. Calculons donc $r_{k\infty}(i(r(f))(d))$. Posons $r(f) \hat{=} (y_0, \ldots, y_k, \ldots)$; et $d \hat{=} (d_0, \ldots, d_k, \ldots)$, autrement dit $d_k = r_{k\infty}(d)$ (puisque $r_{k\infty}$ ne fait que récupérer la composant numéro k). Alors $r_{k\infty}(i(r(f))(d)) = y_{k+1}(d_k)$ (par définition de i) $= y_{k+1}(r_{k\infty}(d))$. Or par définition $y_{k+1} = r_{k\infty} \circ f \circ i_{k\infty}$, donc $r_{k\infty}(i(r(f))(d)) = r_{k\infty}(f(i_{k\infty}(r_{k\infty}(d))))$. Par (c):

$$\begin{array}{rcl}i(r(f))(d) & = & \bigsqcup_k i_{k\infty}(r_{k\infty}(i(r(f))(d)) \\ & = & \bigsqcup_k i_{k\infty}(r_{k\infty}(f(i_{k\infty}(r_{k\infty}(d)))) \\ & = & f(d)\end{array}$$

en utilisant la remarque (b). C'était la partie la plus compliquée à montrer.

Calculons maintenant $r \circ i$. Fixons $y \hat{=} (y_0, y_1, \ldots, y_k, \ldots)$ dans D_∞. La fonction $f \hat{=} i(y)$ envoie tout $x \hat{=} (x_0, x_1, \ldots, x_k, \ldots)$ vers $(y_1(x_0), y_2(x_1), \ldots, y_{k+1}(x_k), \ldots)$. Et $r(i(y)) = r(f)$ est alors un élément $(d_0, d_1, \ldots, d_k, \ldots)$ tel que, d'une part, $d_0 = r_{0\infty}(f(i_{0\infty}(\perp_0))) = y_1(x_0)$ avec x_0 la composante 0 de $i_{0\infty}(\perp_0)$, soit $x_0 = r_{0\infty}(i_{0\infty}(\perp_0)) = \perp_0$, donc $d_0 = y_1(\perp_0) = y_0$; d'autre part, d_{k+1} est la fonction qui à x_k associe $r_{k\infty}(f(i_{k\infty}(x_k))) = y_{k+1}(r_{k\infty}(i_{k\infty}(x_k))) = y_{k+1}(x_k)$, donc $d_{k+1} = y_{k+1}$. Donc $(r \circ i)(y) = y$, pour tout y. $\diamondsuit$

Exercice 45 *Montrer que la règle (η) est valide dans D_∞, autrement dit que $[\![\lambda x \cdot ux]\!]\rho = [\![u]\!]\rho$ dès que x n'est pas libre dans u. (On pourra commencer par le prouver dans le cas où u est une variable $y \neq x$, et remarquer ensuite que $(\lambda x \cdot yx)[y := u] = \lambda x \cdot ux$.)*

Exercice 46 *Supposons que D_0 n'est pas trivial, montrer alors que dans D_∞, $[\![\Omega]\!]\rho = \perp$ et que $[\![\lambda x \cdot x]\!] = r(id) \neq \perp$. Déduire de l'exercice précédent que le λ-calcul avec $\beta\eta$-réduction est* cohérent*, au sens où il existe deux termes u et v tels que $u \neq_{\beta\eta} v$ (ici, Ω et $\lambda x \cdot x$).*

Ce dernier résultat peut aussi se montrer en prouvant que la $\beta\eta$-réduction est confluente, et en remarquant ensuite qu'Ω et $\lambda x \cdot x$ n'ont aucun rédex commun.