UFR IM²AG
UNIVERSITÉ Grenoble Alpes

Modèles de Calcul [Lambda-Calcul]
Programmation par points fixes

Pascal Fradet Jean-François Monin Catherine Parent-Vigouroux

1 Notion de normalisation

On rappelle qu'un rédex dans un terme T est une opportunité de réduction dans T, c'est-à-dire l'occurrence d'une combinaison de la forme $(\lambda x.\mathrm{U})\mathrm{V}$ dans T. Il convient de noter qu'en général, T peut comporter un nombre quelconque de rédexes. Autrement dit, il peut exister plusieurs termes T' , T'' , T''' etc. tels que $\mathrm{T} \xrightarrow{\beta} \mathrm{T}'$, $\mathrm{T} \xrightarrow{\beta} \mathrm{T}''$, $\mathrm{T} \xrightarrow{\beta} \mathrm{T}'''$, etc. À leur tour, T', T'', T''' etc... peuvent comporter plusieurs rédexes, et ainsi de suite. Cela correspond à l'idée qu'il peut y avoir plusieurs manières d'effectuer un calcul.

Définitions

On dit qu'un terme T est :

- *normal* ou *sous forme normale* s'il n'existe aucun terme T' tel que $\mathrm{T} \xrightarrow{\beta} \mathrm{T}'$ (ou encore, si T ne comporte aucun rédex ; le calcul est terminé) ;
- *faiblement normalisant* s'il existe une séquence de termes $\mathrm{T}_0, \mathrm{T}_1, \ldots \mathrm{T}_n$ telle que T_n est normal, $\mathrm{T} = \mathrm{T}_0$ et $\mathrm{T}_i \xrightarrow{\beta} \mathrm{T}_{i+1}$ pour tout $i < n$; intuitivement, le calcul *peut* terminer – il existe une suite de réductions à partir de T pouvant s'écrire $\mathrm{T} = \mathrm{T}_0 \xrightarrow{\beta} \mathrm{T}_1 \xrightarrow{\beta} \ldots \xrightarrow{\beta} \mathrm{T}_n$ qui aboutit à une forme normale ; on dit alors que T_n est une *forme normale de* T ;
- *fortement normalisant* si toute séquence de termes $\mathrm{T}_0, \mathrm{T}_1, \ldots \mathrm{T}_i \ldots$ telle que $\mathrm{T} = \mathrm{T}_0$ et $\mathrm{T}_i \xrightarrow{\beta} \mathrm{T}_{i+1}$ est finie ; intuitivement, le calcul termine quelle que soit la manière de s'y prendre – toute suite de réductions $\mathrm{T} = \mathrm{T}_0 \xrightarrow{\beta} \mathrm{T}_1 \xrightarrow{\beta} \ldots \xrightarrow{\beta} \mathrm{T}_i \xrightarrow{\beta} \ldots$ aboutit fatalement à un terme T_n qui est normal.

En particulier, tout terme fortement normalisant est faiblement normalisant, et tout terme normal est fortement normalisant.

1. Vérifier que les constructeurs définis dans les fiches précédentes pour représenter des types de données (booléens, entiers de Church, produits, sommes, listes, arbres) sont des termes normaux.
2. Vérifier que les fonctions définies sur ces structures de données (exemples : opérations booléennes, opérations sur les entiers, test à 0) sont également des termes normaux.
3. En revanche, en appliquant les secondes sur les premiers, on obtient des termes qui ne sont pas normaux ; observer cependant que les calculs terminent toujours. Autrement dit, les termes construits jusqu'ici sont ... *[compléter la phrase]*.

2 Combinateur paradoxal

On rappelle les définitions de λ-termes suivantes :

- $\Delta \stackrel{\text{def}}{=} \lambda x.\, x\, x$
- $\Omega \stackrel{\text{def}}{=} \Delta\, \Delta$, appelé le *combinateur paradoxal.*

1. Observer que, par définition, $\Omega = \Delta\ \Delta = (\lambda x.\, x\ x)\ \Delta$. Indiquer le rédex présent dans ce λ-terme et trouver le λ-terme vers lequel il se réduit.

2. En déduire que Ω n'est pas faiblement normalisant.
3. Soit $\mathrm{T} \stackrel{\text{def}}{=} (\lambda x y.\, x)\left((\lambda n m f z.\, (n\, f\, (m\, f\, z)))\, (\lambda g x.\, g\, x)\, (\lambda h y.\, h\, y)\right)\Omega$.
   Indiquer différentes réductions de T.
   T est-il faiblement normalisant ? fortement normalisant ?

3 Combinateur de point fixe

On pose :

— $\Delta' \stackrel{\text{def}}{=} \lambda f.\lambda x.\, f(x\, x)$

— $\mathbf{Y} \stackrel{\text{def}}{=} \lambda f.(\Delta'\, f)\, (\Delta'\, f)$
   Y est appelé le combinateur de point fixe de Curry.

Soit U un λ-terme quelconque.

1. Observer que $\mathbf{Y}\,\mathrm{U} \xrightarrow{\beta} (\Delta'\,\mathrm{U})\,(\Delta'\,\mathrm{U}) \xrightarrow{\beta} (\lambda x.\,\mathrm{U}(x\, x))(\Delta'\,\mathrm{U})$. Indiquer le rédex le plus à gauche présent dans ce λ-terme et trouver le λ-terme vers lequel il se réduit.
2. En déduire que $\mathbf{Y}\mathrm{U}$ n'est pas fortement normalisant. Donner des exemples de U pour lesquels $\mathbf{Y}\mathrm{U}$ est faiblement normalisant et des exemples de U pour lesquels $\mathbf{Y}\mathrm{U}$ n'est pas faiblement normalisant.
3. Soit $\mathrm{T_U} \stackrel{\text{def}}{=} (\lambda x y.\, x)\left((\lambda n m f z.\, (n\, f\, (m\, f\, z)))\, (\lambda g x.\, g\, x)\, (\lambda h y.\, h\, y)\right)(\mathbf{Y}\,\mathrm{U})$.
   Indiquer différentes réductions de $\mathrm{T_U}$.
   $\mathrm{T_U}$ est-il faiblement normalisant ? fortement normalisant ?

Remarque. La raison pour laquelle **Y** est appelé combinateur de point fixe est la suivante. Si on considère que deux λ-termes U et V sont égaux si l'on peut passer de l'un à l'autre par des β-réductions, dans un sens ou dans l'autre (mathématiquement on dira qu'ils sont dans la même classe d'équivalence de la relation obtenue par clôture réflexive, symétrique et transitive de $\xrightarrow{\beta}$) les questions précédentes nous amènent à conclure $\mathrm{U}\,(\mathbf{Y}\,\mathrm{U}) = \mathrm{Y}\,\mathrm{U}$. Autrement dit, $\mathbf{Y}\,\mathrm{U}$ est un point fixe de U, c'est-à-dire une solution de l'équation $\mathrm{U}(x) = x$.

4 Codage de définitions récursives par point fixe

Illustrons la notion de *définition récursive* par l'exemple bien connu de la fonction factorielle (dans une syntaxe à la Coq) :

$$\mathsf{fact} \stackrel{\text{def}}{=} \textbf{fun}\ n\ \Rightarrow\ \textbf{if}\ n = 0\ \textbf{then}\ 1\ \textbf{else}\ n \times \mathsf{fact}\,(n-1) \tag{1}$$

Plus généralement, ce que l'on appelle habituellement une définition récursive s'écrit sous la forme

$$\mathsf{r} \stackrel{\text{def}}{=} \mathit{expression}_{\mathsf{r}} \tag{2}$$

étant entendu que le nom r de ce que l'on prétend ainsi définir apparaît dans l'expression du membre droit de ladite « définition ». Dit ainsi, il s'agit d'une escroquerie : une définition ne peut reposer que sur des notions préalablement définies, et donc en aucun cas faire à référence des objets qui ne sont pas encore définis ! C'est d'ailleurs exactement le message d'erreur que vous obtiendrez en Coq si vous essayez :

```
Definition fact := fun n => match n with 0 => 1 | S p => n * fact p end.
```

Cependant il est possible de voir (2) comme une définition *indirecte*. Pour cela, il convient de commencer par isoler les occurrences du nom r dans l'expression servant de membre droit à cette « définition », ou plus précisément de voir $\mathit{expression}_{\mathsf{r}}$ comme une expression paramétrée par r. Renommons ensuite r en x par exemple [1], pour faire apparaître l'expression en membre droit comme une fonction de x.

1. N'importe quel nouveau nom peut convenir pourvu qu'il n'apparaisse pas déjà dans l'expression.

Attention ! Dans ce qui suit x n'est pas un nat mais une *fonction* de nat vers nat.
Dans l'exemple (1), l'expression en membre droit devient

$$\textbf{fun}\ n\ \Rightarrow\ \ \textbf{if}\ n=0\ \textbf{then}\ 1\ \textbf{else}\ n\times x(n-1) \tag{3}$$

Faisons de cette expression une fonction explicite de x, disons $\mathsf{f}(x)$. Dans notre exemple cela revient à poser

$$\mathsf{f}(x)\ \stackrel{\text{def}}{=}\ \textbf{fun}\ n\ \Rightarrow\ \ \textbf{if}\ n=0\ \textbf{then}\ 1\ \textbf{else}\ n\times x(n-1) \tag{4}$$

ou encore, dans un langage de programmation issu du λ-calcul :

$$\mathsf{f}\ \stackrel{\text{def}}{=}\ \textbf{fun}\ x\ \Rightarrow\ \textbf{fun}\ n\ \Rightarrow\ \ \textbf{if}\ n=0\ \textbf{then}\ 1\ \textbf{else}\ n\times x(n-1) \tag{5}$$

Ainsi en Coq, la définition suivante est acceptée :

```
Definition f := fun x => fun n => match n with 0 => 1 | S p => n * x p end.
```

La définition auxiliaire f est parfois appelée la *fonctionnelle associée à la définition récursive de* r considérée (une fonctionnelle est une fonction de fonctions).

Une fois que la fonctionnelle f associée à r est posée, (2) se comprend comme l'affirmation suivante :

$$\mathsf{r}\ =\ \mathsf{f}(\mathsf{r}) \tag{6}$$

Autrement dit, comme l'affirmation que r est un point fixe de f, c'est-à-dire une solution de l'équation

$$x\ =\ \mathsf{f}(x) \tag{7}$$

Pour que cela puisse être accepté comme une définition de r, encore faut-il s'assurer
- — que l'équation (7) a bien une solution ;
- — que cette solution est unique.

Cela n'a rien d'évident a priori. Oublions un instant les points fixes sur les fonctions et considérons des points fixes sur des objet plus simples, les entiers. Par exemple, les équations suivantes :

$$x = x+1 \tag{8}$$
$$x = 2-x \tag{9}$$
$$x = x*x \tag{10}$$
$$x = x \tag{11}$$

ont respectivement 0, 1, 2 et une infinité de solutions. Le même genre de phénomène est susceptible d'arriver sur les points fixes de fonctions.

Ici, on indique simplement que moyennant certaines conditions sur le corps de f, l'existence et l'unicité d'un tel point fixe[2] sont assurées. Dans le cadre de Coq, cela explique l'utilisation du mot clé `Fixpoint` pour définir des fonctions récursives, par exemple

```
Fixpoint fact n := match n with 0 => 1 | S p => n * fact p end.
```

Cette définition est acceptée parce que l'appel récursif s'effectue sur un argument (p) qui est un sous-terme du paramètre n de la fonction (au moment de l'appel récursif, n a été décomposé en S p). Cela marche sur les entiers primitifs de Coq, mais ne se transposerait pas sur les entiers de Church.

2. Ou plus exactement, celles d'un *plus petit* point fixe. On utilise ici une relation de comparaison entre fonctions qui est : $f \leq g$ ssi f est moins définie que g ; plus formellement : pour tout x, si le calcul de $f\,x$ peut terminer et produire une valeur, alors $g\,x$ peut aussi terminer, en produisant la même valeur. On ne détaille pas davantage ici.

Remarque : dans les fiches précédentes, on a utilisé le mot clé `Fixpoint` pour définir la fonction `iter`.

On retiendra également que la notion de *point fixe* est souvent utilisée pour donner un sens mathématique aux définitions récursives.

Pour revenir au λ-calcul, nous avons vu à la section précédente que le combinateur **Y** permet précisément de calculer un point fixe d'une fonction U. Ainsi, pour reprendre l'exemple de la factorielle, on peut poursuivre la recette ébauchée par les équations (3), (4) et (5). Il convient alors de traduire la dernière en λ-calcul en utilisant le codage des entiers de Church, avec les fonctions test à 0, multiplication et prédécesseur.

Le mot employé (*point fixe*) correspond à une idée statique des fonctions, mais nous avons beaucoup mieux. On va voir en effet qu'en *appliquant* $\mathbf{Y}\mathsf{f}$ à un argument a, l'expression obtenue peut se β-*réduire* de sorte que les termes successifs correspondent exactement au calcul qui s'effectue avec les mécanismes d'exécution des langages de programmation usuels.

1. Traduire (5) en λ-calcul pur et non typé
$$\mathsf{f} \stackrel{\text{def}}{=} \lambda x.\lambda n.\ \ldots \tag{12}$$
2. On définit $\mathsf{fact} \stackrel{\text{def}}{=} \mathbf{Y}\mathsf{f}$. Observer que fact n'est pas sous forme normale et trouver des termes f_1 , f_2 , … tels que $\mathsf{fact} \xrightarrow{\beta} f_1 \xrightarrow{\beta} f_2 \xrightarrow{\beta} \ldots$.
   Remarque : ces réductions sont possibles en λ-calcul et ne se terminent pas ; si on fait la correspondance avec les mécanismes usuels d'évaluation d'expressions contenant des fonctions définies récursivement, tout se passe comme si on repérait $\mathbf{Y}\mathsf{f}$ et qu'on évite de réduire cette expression « pour rien », c'est à dire lorsqu'elle n'est pas elle-même appliquée à un argument.
3. Réduire $\mathsf{fact}\, c_0$, $\mathsf{fact}\, c_1$ et $\mathsf{fact}\, c_2$ où c_0, c_1 et c_2 sont les entiers de Church pour 0, 1 et 2.
   Observer les opportunités de réduire $\mathbf{Y}\mathsf{f}$ et dessiner le treillis des réductions possibles.

5 Récursion bornée

Il arrive que l'on soit capable de connaître à l'avance la profondeur des appels récursifs nécessaires à l'évaluation d'une fonction récursive r sur un argument a, ou un majorant m de ce nombre d'appels. Si f est la fonctionnelle associée à r, on remarque qu'il suffit d'itérer f $m+1$ fois sur une fonction f_0 quelconque (car inutilisée) et sur a pour obtenir $\mathsf{r}\, a$:
$$\mathsf{r}\, a \;=\; \underbrace{\mathsf{f}\,(\ldots(\mathsf{f}}_{m+1}\, f_0)\ldots)\, a \tag{13}$$

Par exemple pour calculer fact n il suffit de $n+1$ appels à f. Il n'est alors pas nécessaire de faire appel à **Y**.

1. Donner un λ-terme pour calculer la factorielle de 2 sans utiliser **Y**. Indication : penser aux entiers de Church.
2. Donner une définition générale sans **Y** de fact en λ-calcul, et la réaliser en Coq.