Fondements de la programmation

Exercices 8 lambda-calcul

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

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

où α est un type atomique.

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

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

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

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

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

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

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

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

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

Ces règles définissent un système d'inférence similaire au calcul des séquents. Une inférence de typage est un arbre dont chaque noeud est un instance de règle, chaque feuille une instance d'axiome, de telle sorte que chaque arête mette en relation un même séquent à chaque extrémité.

Une autre façon pratique de voir le typage des lambda-termes est de ne considérer que des lambda-termes dans lesquels les variables liées sont choisies toutes différentes par α-renommage (on ne peut pas écrire $(\lambda x.x\ \lambda x.x)$, on écrit $(\lambda x.x\ \lambda y.y)$) et dans lesquels on a choisi pour chaque variable un type unique (on annote les occurrences de variables dans le terme avec leur type). Ce choix est l'équivalent du choix d'un contexte de typage étendu aux variables liées. Si le lambda-terme est typable dans un tel contexte, il a alors un type unique et on peut trouver son type en suivant la structure syntaxique du terme comme dans 1.

Un lambda-terme pur peut ne correspondre à aucun terme typé comme il peut correspondre à plusieurs termes typés (il peut n'y avoir aucune façon de le décorer avec des types comme il peut y en avoir plusieurs).

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

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

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

1 Exercices

1.1 Inférence de type

Donner un type à chacun des termes suivants, lorsque c'est possible :

1. $\lambda x.x$
2. $\lambda xf.(f\ (f\ (f\ x)))$
3. $\lambda xyz.((x\ y)\ z)$
4. $((\lambda x.x)\ (\lambda x.x))$
5. $(x\ y)$
6. $(x\ x)$
7. $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$
8. Les entiers de Church et les opérations que nous avons défini sur ces entiers.

1.2 Terme non typable

Donner un exemple de terme non typable qui se réduit en un terme typable.

1.3 Termes d'un type donné

Pour chacun des types suivants, donner un terme clos (sans variable libre) de ce type.

1. $\alpha \to \alpha$
2. $(\alpha \to \alpha) \to (\alpha \to \alpha)$
3. $\alpha \to ((\alpha \to \beta) \to \beta)$.

1.4 Prédécesseur (difficile)

Trouver un λ-terme pred qui calcule le prédécesseur sur les entiers de Church, c'est à dire tel que appliqué à un entier $\overline{n}$ de Church se β-réduit en $\overline{n-1}$ si $n > 0$ et en $\lambda fx.x$ (c'est à dire $\overline{0}$) sinon.

1.5 Combinateur de point fixe Y de Church

Soit :

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

Dans le lambda-calcul pur, pour f terme quelconque, montrer que $(Y\,f) \equiv_\beta (f(Y\,f))$.

1.6 Fonction récursive en lambda-calcul

Définir la fonction factorielle en lambda-calcul pur (sur les entiers de Church). Indication : cette fonction doit être le point fixe d'un fonction définie par cas.