Modèles de Calcul [Lambda-Calcul]
Fiche 1 : Syntaxe & portées

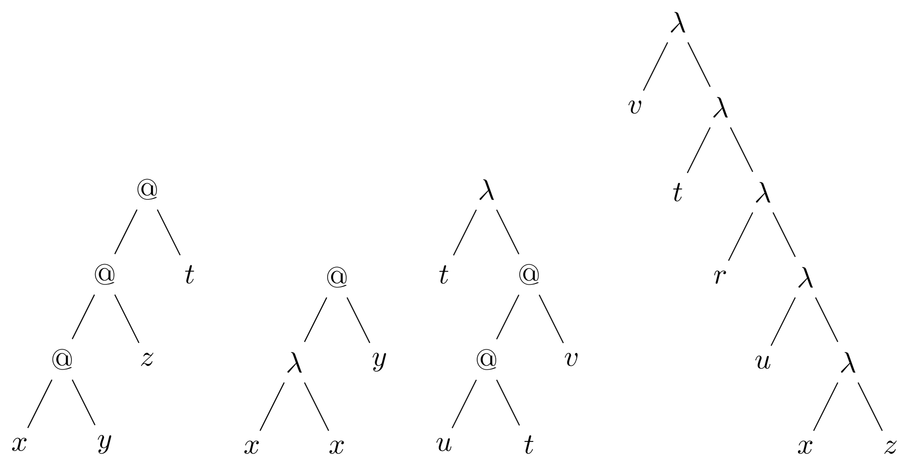
Clément Charpentier, Jean-François Monin, Catherine Parent-Vigouroux

Rappels

Soit V un ensemble infini dénombrable de variables. On définit inductivement l'ensemble Λ des λ-termes par $\Lambda := x|(\lambda x.\Lambda)|(\Lambda\Lambda)$ avec $x \in V$.

1 Arbres syntaxiques

1. Donner le λ-terme correspondant à chacun de ces arbres syntaxiques :



2. Pour chacun des termes suivants, donner l'arbre qui lui correspond :

1/ $(\lambda x.(\lambda v.x))$
2/ $(\lambda x.(x\ y))$
3/ $(x\ (y\ t))$
4/ $(\lambda v.(\lambda t.(\lambda r.((u\ x)\ z))))$
5/ $((x\ y)\ (((z\ t)\ t)\ v))$
6/ $(((\lambda v.(((u\, t)\ v)\ r))\ (y\ z))\ (\lambda x.x))$

3. Simplifier au maximum le parenthèsage des termes de cet exercice.

2 Abréviations

Donner la notation complètement parenthésée et non abrégée des λ-termes suivants :

1/ $(x\ y\ z)$
2/ $\lambda v.u\ x\ z$
3/ $\lambda vxy.u$
4/ $\lambda vtr.(u\ x\ z)$
5/ $\lambda x.(x\ (\lambda y.y\ x))$
6/ $(u\ x\ (y\ z)(\lambda v.v\ y))$
7/ $(\lambda xyz.(x\ z\ (y\ z))\ u\ v\ w)$
8/ $(w(\lambda xyz.(x\ z\ (y\ z))\ u\ v))$

3 Définitions de fonctions

Rendre à chaque terme sa définition en français.

1/ $\lambda x.\, x$

2/ $\lambda x.\, y$

3/ $\lambda x.\lambda v.\, x$

4/ $\lambda x.\lambda y.\, (x\ y)$

5/ $\lambda x.\, (x\ x)$

1/ Fonction qui, à deux arguments, associe le premier.

2/ Fonction identité.

3/ Fonction qui applique son argument à lui-même.

4/ Fonction qui, à deux arguments, associe l'application du premier au deuxième.

5/ Fonction qui, à un argument, associe la valeur d'une variable.

4 Variables libres / liées

Pour chaque λ-terme, donner les occurrences libres et liées, puis marquer la portée de chaque variable x.

1/ $(\lambda x.\, x\ y)$

2/ $(\lambda vxy.\, u\ v\ y)$

3/ $(\lambda x.(x\ (\lambda y.\, y\ x))$

4/ $((\lambda xyz.\, x\ z\ (y\ z))\ u\ x\ w)$

5/ $(u\ x\ (y\ z)\ \lambda x.\, x\ y)$

5 Typage élémentaire

On rappelle les règles élémentaires de typage. On suppose ici que l'on a un type primitif `nat` pour les entiers naturels 0, 1, 2...

- Pour typer une abstraction $\lambda x.U$, on déclare le type de la variable x en le mettant en exposant de x ; par exemple si ce type est `nat`, on écrit $\lambda x^{\mathtt{nat}}.\, U$;
  si le type de U est lui même `nat`, le type de $\lambda x^{\mathtt{nat}}.\, U$ est alors $\mathtt{nat} \rightarrow \mathtt{nat}$: c'est une fonction de `nat` vers `nat`.
- Lorsqu'un λ-terme comporte une variable libre, il faut également la typer ; par exemple $\lambda x^{\mathtt{nat}}.\, y^{\mathtt{nat}}$ est de type $\mathtt{nat} \rightarrow \mathtt{nat}$; en revanche on n'écrira pas $\lambda x^{\mathtt{nat}}.\, x^{\mathtt{nat}}$, mais $\lambda x^{\mathtt{nat}}.\, x$, simplement.
- Plus généralement, les types sont de la forme
  - `nat`
  - $\mathtt{nat} \rightarrow \mathtt{nat}$, par exemple pour les fonctions $\lambda x^{\mathtt{nat}}.\, 3$, $\lambda x^{\mathtt{nat}}.\, x$ et $\lambda x^{\mathtt{nat}}.\, y^{\mathtt{nat}}$
  - $\mathtt{nat} \rightarrow (\mathtt{nat} \rightarrow \mathtt{nat})$: fonctions prenant un `nat` en entrée et retournant une fonction de `nat` vers `nat`, par exemple $\lambda x^{\mathtt{nat}}.\lambda y^{\mathtt{nat}}.\, 3$
  - $(\mathtt{nat} \rightarrow \mathtt{nat}) \rightarrow \mathtt{nat}$: fonctions prenant une fonction de `nat` vers `nat` en entrée et retournant un `nat`, par exemple $\lambda f^{\mathtt{nat}\rightarrow\mathtt{nat}}.\, x^{\mathtt{nat}}$
  - $(\mathtt{nat} \rightarrow \mathtt{nat}) \rightarrow (\mathtt{nat} \rightarrow \mathtt{nat})$: fonctions prenant une fonction de `nat` vers `nat` en entrée et retournant une fonction de `nat` vers `nat`, par exemple $\lambda f^{\mathtt{nat}\rightarrow\mathtt{nat}}.\lambda x^{\mathtt{nat}}.\, 3$
  - etc., c.-à-d. tous les types de la forme `nat` ou $\tau_1 \rightarrow \tau_2$ où τ_1 et τ_2 sont des types déjà construits.
- Si un λ-terme U a le type $\tau_1 \rightarrow \tau_2$, alors on peut l'appliquer à un terme V de type τ_1 et le type de $U\,V$ est τ_2.
  Par exemple le type de $(\lambda x^{\mathtt{nat}}.\, 3)\, 1$ est `nat` car $\lambda x^{\mathtt{nat}}.\, 3$ est de type $\mathtt{nat} \rightarrow \mathtt{nat}$ et 1 est de type `nat`.

- On se donne également la possibilité d'écrire des additions et soustractions entre entiers naturels : si U et V sont des λ-termes de type `nat`, alors $U+V$ et $U-V$ sont aussi des λ-termes de type `nat`.

Typer, lorsque cela est possible, les λ-termes suivants en indiquant un type approprié pour les variables déclarées et le type de chaque sous-terme.

1/ $x+2$

2/ $\lambda x.\ x+2$

3/ $(\lambda x.\, x+2)\ 3$

4/ $(\lambda x.\lambda y.\ x-y)\ 10\ 8$

5/ $f\ 3\ +\ 1$

6/ $\lambda f.\,(f\ 3\ +\ 1)$

7/ $\lambda f.\,(f\ 10\ 8\ +\ 1)$

8/ $(\lambda x.\, x+7)\, y$

9/ $(\lambda x.\, x+7)\, y\, 4$

10/ $(\lambda v.\, u\, v\, w)\, 4$

11/ $(\lambda u.\, u\, v\, w)\, 4$

12/ $(\lambda f.(f\ 1))\ (\lambda x.\ x+2)$

6 Évaluations en Coq

Rappel: Coq est un assistant à la preuve qui embarque un λ-calcul typé avec constantes. Parmi ces dernières on trouve des entiers naturels notés 0, 1, 2, ... de type `nat` et des fonctions prédéfinies comme l'addition et la soustraction notées de façon usuelle (infixe). La syntaxe pour l'abstraction typée $\lambda x^{\mathtt{nat}}.3$ est `fun x : nat => 3`.

Dans certains cas, Coq est capable d'inférer le type d'une variable. Par exemple dans $\lambda x.\, x+2$ la variable x est nécessairement de type `nat`, au lieu de `fun x : nat => x + 2` on peut donc écrire plus simplement : `fun x => x + 2`.

Nous allons utiliser le système de preuves Coq pour vérifier la bonne construction de λ-termes et les évaluer si possible.
Pour lancer Coq, sous Linux, taper **coqide**.
Toute phrase Coq se termine par un point.

Commençons par quelques termes sans variables libres (appelés aussi *termes clos*).

Pour donner un nom au λ-terme $(\lambda x.\, x+2)$:

```
Definition f := fun x => x + 2.
```

Pour voir le λ-terme $(\lambda x.\, x+2)$ et son type :

```
Print pl2.
```

Pour l'évaluer :

```
Compute pl2.
```

Pour définir le λ-terme $((\lambda x.\, x+2)\ 3)$ on peut alors soit procéder directement :

```
Definition t := (fun x => x + 2) 3.
```

soit utiliser la définition précédente pour $(\lambda x.\, x+2)$:

```
Definition t' := pl2 3.
```

Pour les évaluer :

```
Compute t.
Compute t'.
```

Pour définir le λ-terme $(\lambda x.\lambda y.\ x - y)$ 8 10 :

```
Definition v := (fun x => fun y => x - y) 8 10.
```

Voici une autre définition possible, correspondant à la syntaxe abrégée $(\lambda xy.\ x - y)$ 8 10 :

```
Definition u := (fun x y => x - y) 8 10.
```

Cependant dans le cas général on souhaite définir des termes comportant des variables libres. Il faut annoncer celles-ci à l'avance, avec leur type, et aussi indiquer jusqu'où de telles variables sont visibles. À cet effet on va utiliser un mécanisme de Coq appelé *section* : on ouvrira une section, qui sera fermée à la fin. La portée d'une variable déclarée dans une section débute à l'endroit de la déclaration et se termine à la fin de cette section.

Ouverture d'une section :

```
Section td1.
```

Fermeture d'une section (à faire à la fin du TP) :

```
End td1.
```

Pour disposer d'une variable libre de type `nat`, visible jusqu'à la fin de la section en cours :

```
Variable y : nat.
```

La variable libre y peut être utilisée dans les définitions qui suivent, par exemple :

```
Definition ply := fun x => x + y.
```

1. Tester et évaluer l'ensemble des λ-termes proposés ci-dessus en section **5**, ainsi que $(\lambda y.\lambda x.\ x - y)$ 8 10.

2. Coder en Coq chacun des λ-termes de la section 4, en prenant soin de déclarer les variables libres nécessaires. Faire deux versions : une avec des variables libres de type `nat` (dans un premier temps, on va jouer uniquement avec ce type), puis une seconde avec des variables libres d'un type quelconque `T`.

   Pour disposer d'un type et l'utiliser dans la suite de la section :

   ```
   Variable T : Set.
   Variable y : T.
   ```

   Pour chaque version, ouvrir une section et la fermer à la fin.