UFR IM²AG
UNIVERSITÉ Grenoble Alpes

Modèles de Calcul [Lambda-Calcul]
Programmation en λ-calcul simplement typé (booléens et entiers)

Pascal Fradet, Jean-François Monin, Catherine Parent-Vigouroux

1 Typage simple avec entiers primitifs : principe et règles

1.1 Rappels de cours

On rappelle les règles élémentaires de typage. On considère ici un λ-calcul *non pur* comprenant des constantes entières 0, 1, 2, ... et les opérations d'addition et de soustraction notées de manière infixe, de sorte que les λ-termes considérés ici seront tous ceux du λ-calcul pur ainsi que, par exemple : 7, $2+3$, $2+x$, $x-y$, $\lambda x.(y+5)-x$. Autrement dit on ajoute au langage de λ-termes les règles de formation suivantes :

— $0, 1, 2, \ldots$ sont des λ-termes ;
— si U et V sont des λ-termes, alors $(U+V)$ et $(U-V)$ sont aussi des λ-termes.

Une expression comme $f\,5$ est acceptable, si f représente une fonction des entiers vers les entiers (ou vers autre chose). En revanche on va refuser $5\,x$ car la constante primitive 5 n'est pas une fonction, ainsi que $(\lambda x.5)+2$ car $\lambda x.5$ n'est pas un entier. On formalise cela au moyen d'un système de *types* comprenant :

— un langage pour exprimer des types τ ;
— un prédicat de typage affirmant que le λ-terme U a pour type τ, noté $U : \tau$;
— un ensemble de règles de typage permettant de conclure que U a pour type τ, par examen de la structure de U ; on suppose au préalable que l'on a assigné un type à chaque variable de U ; on le note en exposant, par exemple $x^{\texttt{nat}}$.

Intuitivement, on aura les types suivants :

— `nat` pour des entiers, par exemple 2 ou $3+5$;
— `nat` → `nat`, par exemple pour les fonctions $\lambda x^{\texttt{nat}}.3$, $\lambda x^{\texttt{nat}}.x$ et $\lambda x^{\texttt{nat}}.y^{\texttt{nat}}$;
— `nat` → (`nat` → `nat`) pour des fonctions prenant un `nat` en entrée et retournant une fonction de `nat` vers `nat`, par exemple $\lambda x^{\texttt{nat}}.\lambda y^{\texttt{nat}}.3$;
— (`nat` → `nat`) → `nat` pour des fonctions prenant une fonction de `nat` vers `nat` en entrée et retournant un `nat`, par exemple $\lambda f^{\texttt{nat}\to\texttt{nat}}.f\,3$;
— (`nat` → `nat`) → (`nat` → `nat`) pour des fonctions prenant une fonction de `nat` vers `nat` en entrée et retournant une fonction de `nat` vers `nat`, par exemple $\lambda f^{\texttt{nat}\to\texttt{nat}}.\lambda x^{\texttt{nat}}.(x+f\,3)$.

Formellement, le langage des types utilisé ici est le suivant.

— `nat` est un type ;
— si τ_1 et τ_2 sont des types déjà construits, alors $(\tau_1 \to \tau_2)$ est un type.

Par convention, $\tau_1 \to \tau_2 \to \tau_3$ désigne $\tau_1 \to (\tau_2 \to \tau_3)$.

1.2 Exercices sur papier

Typer, lorsque cela est possible, les λ-termes suivants en indiquant un type approprié pour les variables déclarées, les variables libres et le type de chaque sous-terme. Lorsque les termes peuvent se réduire, il est instructif d'effectuer les réductions et d'observer que les types sont conservés. La seconde colonne (exercices 7 à 12) est optionnelle.

1/ $2 + x$

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

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

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

5/ $1 + f\ 3$

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

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

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

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

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

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

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

1.3 Rappels de cours (suite)

Pour formaliser les règles de typage on a préalablement besoin de la notion d'*environnement*, qui est une séquence d'associations entre variable et type et qui permet de détermnier le type d'une variable libre dans un terme. Ces environnements sont formés ainsi :

— ε est un environnement (l'environnement vide) ;

— si Γ est un environnement, x une variable et τ un type, alors $\Gamma;(x,\tau)$ est un environnement.

On peut alors dériver des *jugements de typage* qui s'écrivent $\Gamma \vdash U : \tau$ où Γ est un environnement, U est un terme et τ est un type. On commence par le typage d'une variable x dans un environnement, qui s'effectue en parcourant ce dernier de droite à gauche :

— $\Gamma;(x,\tau) \vdash x : \tau$;

— Si $\Gamma \vdash x : \tau$, et si $x \neq y$, alors $\Gamma;(y,\tau') \vdash x : \tau$.

Le typage d'un λ-terme quelconque s'effectue en décomposant sa structure. Le cas d'une variable a déjà été présenté, il reste l'application, l'abstraction et les opérations entières :

— si $\Gamma \vdash U : \tau_1 \to \tau_2$, et si $\Gamma \vdash V : \tau_1$, alors $\Gamma \vdash U\,V : \tau_2$;

— si $\Gamma \vdash U : \texttt{nat}$, et si $\Gamma \vdash V : \texttt{nat}$, alors $\Gamma \vdash U + V : \texttt{nat}$ et $\Gamma \vdash U - V : \texttt{nat}$;

— si $\Gamma;(x,\tau_1) \vdash U : \tau_2$, alors $\Gamma \vdash (\lambda x^{\tau_1}.U) : \tau_1 \to \tau_2$.

Ces règles de typage peuvent être présentées synthétiquement sous forme de **règles d'inférence** comme suit.

$$\frac{}{\Gamma;(x,\tau) \vdash x : \tau} \qquad \frac{\Gamma \vdash x : \tau \quad x \neq y}{\Gamma;(y,\tau') \vdash x : \tau}$$

$$\frac{\Gamma \vdash U : \tau_1 \to \tau_2 \quad \Gamma \vdash V : \tau_1}{\Gamma \vdash U\,V : \tau_2} \qquad \frac{\Gamma;(x,\tau_1) \vdash U : \tau_2}{\Gamma \vdash (\lambda x^{\tau_1}.U) : \tau_1 \to \tau_2}$$

$$\frac{}{\Gamma \vdash 0 : \texttt{nat}} \qquad \frac{}{\Gamma \vdash 1 : \texttt{nat}} \qquad \frac{}{\Gamma \vdash 2 : \texttt{nat}} \qquad \text{etc.}$$

$$\frac{\Gamma \vdash U : \texttt{nat} \quad \Gamma \vdash V : \texttt{nat}}{\Gamma \vdash U + V : \texttt{nat}} \qquad \frac{\Gamma \vdash U : \texttt{nat} \quad \Gamma \vdash V : \texttt{nat}}{\Gamma \vdash U - V : \texttt{nat}}$$

1.4 Exercices sur papier

1.4.1 Typage de variables

Donner le type de x (et justifier) dans les environnements suivants :

1/ $\varepsilon;(x,\texttt{nat});(y,\texttt{nat} \to \texttt{nat})$

2/ $\varepsilon;(x,\texttt{nat});(y,\texttt{nat} \to \texttt{nat});(x,\texttt{nat} \to \texttt{nat})$

3/ $\varepsilon;(x,\texttt{nat} \to \texttt{nat});(y,\texttt{nat} \to \texttt{nat});(x,\texttt{nat})$

4/ ε

1.4.2 Typage de λ-termes

Dériver les jugements de typage adéquats pour les λ-termes donnés dans la série 1.2.

2 Évaluations en Coq

Dans la suite nous allons continuer à utiliser Coq, mais de façon différente par rapport aux fois précédentes. Auparavant, Coq était simplement un support de programmation dans lequel le λ-calcul pur et non typé était défini au moyen de structures de données représentant sa syntaxe abstraite et de fonctions de manipulation programmées à dessein.

En fait, en tant que langage de programmation, Coq *est* lui-même une version typée du λ-calcul. Plus précisément, 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).

Attention à la syntaxe :
l'abstraction typée $\lambda x^{\texttt{nat}}.3$ est notée dans la syntaxe du λ-calcul typé de Coq

```
fun x : nat => 3 .
```

Dans certains cas, Coq est capable d'inférer le type d'une variable. Par exemple dans $\lambda x.2+x$ la variable x est nécessairement de type `nat`, au lieu de `fun x : nat => 2 + x` on peut donc écrire plus simplement : `fun x => 2 + x`.
Nous allons utiliser le système de preuves Coq pour vérifier la bonne construction de λ-termes et les évaluer si possible.

Lancer Coq sous Linux de la façon habituelle au moyen **coqide**.
Contrairement aux fois précédentes, **NE PAS** effectuer le `Require Import untypedLC.`
Rappel : 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 $\lambda x.2+x$:

```
Definition pl2 := fun x => 2 + x.
```

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

```
Print pl2.
```

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

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

soit utiliser la définition précédente `pl2` pour $\lambda x.2+x$:

```
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.
```

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 par la suite. 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. Les variables typées déclarées dans une section constituent un environnement. Une section permet donc de définir un environnement qui sera utilisé pour toutes les définitions effectuées dans cette section.

Ouverture d'une section de nom `sec_1` :

```
Section sec_1.
```

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.
```

Fermeture de la section `sec_1` :

```
End sec_1.
```

Tester et évaluer les λ-termes typables proposés ci-dessus en section **1** ainsi que $(\lambda y.\lambda x.\ x - y)\ 8\ 10$. Selon le temps disponible, vous pouvez vous limiter aux termes les plus simples et réserver les autres pour la préparation à l'examen.

Pour disposer d'un type et l'utiliser dans la suite d'une section, on écrira :

```
Variable T : Set.
```

Par exemple, si on a ensuite besoin d'une variable libre *y* de type T, on pourra poursuivre ainsi :

```
Variable y : T.
```

3 Codage des booléens et de l'opérateur if

Nous revenons au codage des booléens inventé par Church, cette fois dans le cadre du λ-calcul typé de Coq. Dans le langage de types utilisé ci-dessous, on n'utilisera pas `nat` comme type de base, mais à la place un type T arbitraire non interprété. On se place donc dans une section débutant ainsi :

```
Section type_booleen.

Variable T: Set.

(* les booleens sont des fonctions a deux parametres *)
Definition cbool := T -> T -> T.

(* codage de true *)
Definition ctr : cbool := fun x y => x.

(* codage de false *)
Definition cfa : cbool := fun x y => y.
```

1. Coder en Coq le combinateur `cif`, réalisant un *if_then_else*.
2. Vérifier (en utilisant `Compute`) que l'évaluation de `cif` sur *true* (`ctr`) et sur *false* (`cfa`) est correcte. Il convient à cet effet d'appliquer ces termes sur deux variables de type T.

4 Codage des opérateurs booléens

On rappelle les λ-termes correspondant aux opérateurs booléens `not`, `and` et `or`.

— `cnot` $\stackrel{\text{def}}{=} \lambda b.\lambda xy.\, b\ y\ x$

— `cand` $\stackrel{\text{def}}{=} \lambda ab.\lambda xy.\, a\ (b\ x\ y)\ y$

— `cor` $\stackrel{\text{def}}{=} \lambda ab.\lambda xy.\, a\ x\ (b\ x\ y)$

1. Coder l'opérateur `cnot` en Coq.
2. Peut-on typer `cnot1` $\stackrel{\text{def}}{=} \lambda b.\, b$ `cfa ctr` de façon satisfaisante, c.-à-d. avec le même type que `cnot` ?
3. Vérifier avec `Compute` que `cnot ctr` se réduit bien en `cfa`.
4. Coder les opérateurs `cand` et `cor`.

5 Codage des entiers de Church avec typage simple

Nous allons reprendre le codage des entiers de Church dans le λ-calcul typé de Coq. On rappelle qu'ils se définissent comme suit en λ-calcul pur :

— $c_0 \stackrel{\text{def}}{=} \lambda fx.x$

— $c_1 \stackrel{\text{def}}{=} \lambda fx.f\ x$

— $c_2 \stackrel{\text{def}}{=} \lambda fx.f\ (f\ x)$

— $c_3 \stackrel{\text{def}}{=} \lambda fx.f\ (f\ (f\ x))$

— ...

Pour coder ces termes en Coq, il faut leur donner un type. On observe que ce sont des fonctions à deux paramètres f et x qui renvoient f appliqué itérativement n fois à x, n étant l'entier ainsi codé. Un type simple possible en Coq est donc :

```
Variable T: Set.
Definition cnat := (T->T) -> T->T.
```

Remarquer qu'une autre écriture possible de `cnat` est possible :

```
Definition cnat := (T->T) -> (T->T).
```

Cela revient à considérer un entier de Church comme une fonction qui prend en argument une fonction f et rend f itérée un certain nombre de fois, c'est-à-dire $f \circ f \ldots \circ f$.

1. Coder en Coq le type `cnat` et les 3 premiers entiers c_0, c_1 et c_2.
2. On définit la fonction successeur d'un entier de Church par le λ-terme
   `cS` $\stackrel{\text{def}}{=} \lambda n.\lambda fx.(f\ (n\ f\ x))$.
   Coder cette fonction en Coq, puis l'évaluer pour quelques entiers de Church.
3. Pour éviter d'avoir à introduire une constante c_n pour chaque n dont on aurait besoin, on choisit de définir une notation simplifiée pour le terme $\lambda fx.f\ (f\ \ldots x)$ où f est appliquée n fois. Pour cela, on définit la composition de deux fonctions et un itérateur de composition. Il n'est pas demandé de comprendre les détails techniques des lignes suivantes, mais de les copier de façon à utiliser la notation utile introduite à la fin. Attention à la saisie du caractère « ° », qui est différent de « o ».

```
(* Définition de la composition de g et f *)
Definition compo : (T->T) -> (T->T) -> (T->T) :=
  fun g f => fun x => g (f x).

(* Un raccourci syntaxique pour ecrire g ° f au lieu de (compo g f ) *)
```

```
Notation "g ° f" := (compo g f) (at level 10).

(* Un itérateur de f, n fois *)
(* L'écriture 0, 1, 2, 3 etc. des entiers natifs de Coq
   représente en réalité O, S O, S (S O), S (S (S O)) etc. *)
Fixpoint iter (f:T->T) (n: nat) :=
  match n with
  | O => fun x => x
  | S p => f ° (iter f p)
  end.

(* Utilisation de cet itérateur pour construire un cnat a partir d'un nat standard *)
Definition cnat_of : nat -> cnat := fun n => fun f => (iter f n).

(* Raccourci pour écrire le nᵉ entier de Church : [n]N au lieu de (cnat_of n) *)
Notation "[ X ]N " := (cnat_of X) (at level 5).
(* par exemple [3]N signifie (cnat 3) et donc (après réduction) λf x. f (f (f x))  *)
```

Copier ces différents énoncés Coq, tester le raccourci syntaxique pour les entiers de Church puis essayer la fonction successeur sur des entiers plus grands que 3.

6 Opérations sur les entiers de Church avec typage simple

1. L'addition de deux entiers de Church est définie par : $\lambda nm.\lambda fx.n\ f\ (m\ f\ x)$.
   Coder en Coq cette fonction, la tester sur des exemples.
2. (Optionnel) La multiplication de deux entiers de Church est définie par : $\lambda nm.\lambda f.n\ (m\ f)$.
   Coder en Coq cette fonction et la tester sur des exemples.
3. (Optionnel) Le test à zéro d'un entier de Church est défini par : $\lambda n.\lambda xy.n\ (\lambda z.y)\ x$.
   Coder en Coq cette fonction `tz` en utilisant le type `cbool` vu au §**3** ci-dessus comme type résultat de cette fonction de test à zéro.

7 (Optionnel) Réductions pas à pas en mode preuve interactive

Pour détailler pas à pas le processus de réduction déclenché par un calcul, on va utiliser le mode preuve interactive de Coq qui fonctionne ainsi : on énonce un `Theorem` (un théorème à démontrer) puis on enchaine l'application de *tactiques* élémentaires qui correspondent à l'usage de règles valides jusqu'à ce qu'il n'y ait plus rien à démontrer.

Illustrons l'idée sur le calcul de `cnot ctr` : on va démontrer l'égalité `cnot ctr = cfa` en utilisant des tactiques de réduction élémentaires – elles porteront sur premier membre puisque dans le second il n'y a rien à réduire. Notre objectif est d'atteindre le but `cfa = cfa` qui pourra être démontré par la tactique `reflexivity`. Auparavant, notre première tactique s'appelle `cbv` pour *Call By Value*. Nous allons l'utiliser avec, en argument, un nom de réduction qui sera `beta` ou `delta`. Suivant le cas, des β-réductions ou des δ-réductions (expansion de définitions) sont effectuées sur les termes du théorème à démontrer. Pour `delta`, on peut en option (avec `[ ...]`) spécifier le ou les noms de constantes à expanser (par défaut toutes les définitions sont expansées). Sur notre exemple, on obtient la preuve suivante :

```
Theorem not_true : cnot ctr = cfa.
Proof. (* marque le debut de la preuve *)
  cbv delta [cnot]. (* les tactiques commencent par des minuscules *)
  cbv beta.
  cbv delta [ctr].
  cbv beta.
  cbv delta [cfa].
  (* A ce stade on a une egalite entre deux termes syntaxiquement identiques *)
  reflexivity.
Qed. (* marque la fin de la preuve *)
```

1. Exécuter cette preuve, pas à pas, et observer les effets de chaque tactique.
2. Qu'est-ce que la δ-réduction ?
3. Qu'est-ce que la β-réduction ?
4. Produire une preuve analogue pour démontrer `cnot cfa = ctr`.
5. Observer qu'en fait, invoquer `reflexivity` dès le début suffirait pour faire cette démonstration : cet emploi de `reflexivity` cache donc des δ-réductions et des β-réductions.
6. Démontrer une ou plusieurs des propriétés suivantes (table de vérité du `and`) :
   `cand cfa cfa = cfa`, `cand cfa ctr = cfa`, `cand ctr cfa = cfa`, `cand ctr ctr = ctr`.
7. Démontrer les propriétés de la table de vérité du `or`.
8. Démontrer
   `Remark cS_compo : forall n, cS n = fun f => f ° (n f).`
   puis
   `Theorem cS_is_successor : forall n, cS [n]N = [S n]N.`