

Rapport TP1

Valeran MAYTIE

1 Prise en main du logiciel Isabelle

On a commencé par utiliser 2 commandes :

- `typ` : détermine si un type est bien formé.
- `term` : détermine si un term est bien formé et donne son type s'il est typable le plus général (Algo W).

D'abord, on a écrit des types bien formés par exemple :

```
typ <int  $\Rightarrow$  int>
typ <' $\alpha$  list  $\Rightarrow$  ' $\beta$  set>
```

Ensuite, on a écrit des termes. Grâce à l'algorithme W implémenté dans Isabelle, on peut obtenir le type le plus général des termes écrits.

Pour débiter, on a écrit deux termes assez simples :

```
term < $\lambda x. x$ >
term <[1, 2, 3::nat]>
```

Le `::nat` dans le deuxième terme sert à spécifier le type contenu dans la liste. Si cette spécification n'est pas faite l'algorithme nous sortira le type `' α list`. Les symboles 1, 2 et 3 ne sont pas des entiers, mais des types généraux : ' α (Pour garder la possibilité de redéfinir ces symboles).

L'éditeur de texte à une super fonctionnalité qui permet d'afficher les fichiers de définition. Il suffit de faire un `Ctrl+clac droit` sur la définition voulue.

On peut écrire des termes assez compliqués comme celui-ci :

```
term <( $\lambda x. \lambda y. (\lambda z. (\lambda x. z x) (\lambda y. z y)) (x y))$ >
```

Isabelle le rejette, car il ne peut pas être typé. En réalité ce terme cache une auto application. Il se réduit en : $\lambda x y. (x y) (\lambda z. (x y) z)$. Or ce genre de λ -term ne sont pas acceptés par le système de type $\lambda 2$ (polymorphisme).

On récupère dans la console l'erreur :

```
Type unification failed: Occurs check!

Type error in application: incompatible operand type

Operator:  z :: ??'a  $\Rightarrow$  ??'b
Operand:   z :: ??'a  $\Rightarrow$  ??'b
```

Pour débiter un terme comme celui-ci, on peut remplacer la variable par une variable libre en mettant une apostrophe après. On peut remarquer que les variables libres sont coloriées en bleu dans l'interface.

2 Axiome

Dans une théorie logique, on va parfois utiliser des axiomes, car certains énoncés jugés correct ne sont pas prouvables. Par exemple le tiers exclu ($u \vee \neg u$) n'est pas prouvable dans les logiques constructives. Pour cela en Isabelle, on peut ajouter des axiomes qui sont des énoncés acceptés par

le cœur d'Isabelle sans preuve. Il faut donc bien faire attention avant d'ajouter ce genre d'énoncé, car accepter un axiome faux va mettre à mal toute la cohérence d'Isabelle.

Par exemple dans le TP, on nous demande de rajouter l'axiome du combinateur de point fixe :

$$\forall f, \exists Y, Y f = f (Y f)$$

Malheureusement cette formule est fausse. Si on applique cette formule à $f = \neg := \lambda x. \perp$ on obtient $Y \neg = \neg (Y \neg)$ ce qui n'est pas vrai.

Un tel λ -terme existe $\lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$, mais il n'est pas typable, il est alors refusé par Isabelle.

3 Encodage de Church

On cherche à encoder les entiers naturels en λ -calcul pur. Pour cela on utilise l'encodage de Church, il est construit avec une fonction qui prend en paramètre deux variables f une fonction et x une variable. Pour représenter un nombre n on compose n fois la fonction f appliquée à x .

Le λ -term ressemble à :

$$\lambda f. \lambda x. f^n x \equiv \lambda f. \lambda x. \underbrace{f (f \dots (f x))}_{n \text{ fois}}$$

En Isabelle nous définissons les entiers naturels de 0 à 5 comme ceci :

```
definition ZERO where "ZERO  ≡ λf x. x "
```

```
definition ONE  where "ONE   ≡ λf x. f x"
```

```
definition TWO  where "TWO   ≡ λf x. f (f x)"
```

```
definition THREE where "THREE ≡ λf x. f (f (f x))"
```

```
definition FOUR  where "FOUR  ≡ λf x. f (f (f (f x)))"
```

```
definition FIVE  where "FIVE  ≡ λf x. f (f (f (f (f x))))"
```

FIGURE 1 – Entier de Church de 0 à 5

On peut définir des opérations sur cet encodage.

$$\begin{aligned} n + 1 &: \lambda n f x. f (n f x) \\ n + m &: \lambda n m f x. m f (n f x) \\ n \times m &: \lambda n m f x. n (m f) x \\ n^m &: \lambda n m f x. m n \end{aligned}$$

Malheureusement les opérations prédécesseur et soustraction sont plus difficiles à écrire.

En Isabelle, on définit uniquement l'addition (Figure-2) car ça sera la seule utile pour notre première preuve.

```
definition PLUS where "PLUS ≡ λn m f x. m f (n f x)"
```

FIGURE 2 – Addition avec l'encodage de Church

Notre première démonstration consiste à prouver que $3 + 2 = 5$ en entier de Church. L'énoncé s'écrit comme ceci : `PLUS TWO THREE = FIVE`. Pour commencer, il faut dérouler toutes les définitions. On utilise donc la tactique `unfolding` avec comme argument la définition à déplier suivit de “`_ def`” (par exemple pour la définition `PLUS` : `PLUS_def`).

Le dépliage va effectuer tous les calculs possibles, on aura donc :

$$\text{PLUS TWO THREE} \rightarrow_{\beta}^* \lambda f x. f (f (f (f (f x))))$$

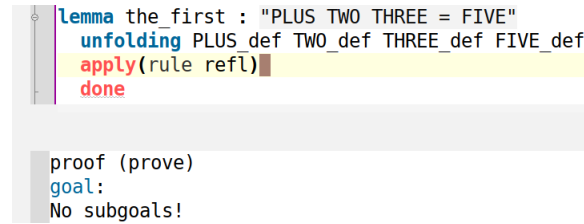
Nous remarquons que `PLUS TWO THREE` se réduit en `FIVE`, il nous reste à monter que `FIVE = FIVE`. Nous avons vu en cours que l'égalité de Isabelle est réflexive ($\forall x, x = x$). Il suffit d'appliquer

le théorème de réflexivité *HOL.refl* que l'on peut trouver à l'aide de la commande `find_theorems "_ = _"`. On peut appliquer ce théorème en utilisant `apply(rule refl)`. La preuve complète de deux lignes :) se trouve ci-dessous (Figure-3).

```
lemma the_first : "PLUS TWO THREE = FIVE"
  unfolding PLUS_def TWO_def THREE_def FIVE_def
  apply(rule refl)
done
```

FIGURE 3 – Preuve que $2 + 3 = 5$ avec l'encodage de Church

Une fois que tous les buts sont prouvés, on a le message suivant indiquant que la preuve est finie :



```
lemma the_first : "PLUS TWO THREE = FIVE"
  unfolding PLUS_def TWO_def THREE_def FIVE_def
  apply(rule refl)
done

proof (prove)
goal:
No subgoals!
```

Les théorèmes prouvés sont rajoutés au cœur d'Isabelle. On peut afficher l'énoncé du théorème en utilisant la commande `thm nom`. Elle va afficher l'énoncé du théorème *nom* dans la console.