Chapitre 9

Implémentation prototype : TypI

Comme dans le cas de la récursion, le développement en parallèle d'un interpréteur nous a permis de faciliter la mise au point de l'algorithme d'inférence grâce à l'expérimentation rapide et automatisée. Ce prototype, baptisé TypI, implémente strictement l'algorithme à rang fini présenté jusqu'ici et permet de suivre et de diriger son exécution pas à pas. Il est écrit en Objective Caml et peut être téléchargé à l'adresse suivante :

`http://www-sop.inria.fr/mimosa/Pascal.Zimmer/typi.html`

Nous allons maintenant présenter les possibilités de ce logiciel sur un exemple.

9.1 Exemple d'interaction

Au lancement, le programme rappelle quel est le rang maximal autorisé (10 par défaut). Il est possible de donner un entier en ligne de commande et de choisir ainsi un rang maximal quelconque.

```
     Intersection Types Inference System

Using default maximal rank: 10

Term:
```

Le système demande alors quel est le λ-terme à analyser. On utilise la syntaxe suivante :
- $\lambda x M$ s'écrit "`\x M`" et $[M, N]$ s'écrit "`[M,N]`"
- afin de faciliter l'entrée, certaines constantes sont prédéfinies :

constante	valeur
`I`	$\lambda x\ x$
`D`	$\lambda x\ (x\ x) = \Delta$
`F`	$\lambda x \lambda y\ y$
`K`	$\lambda x \lambda y\ x$
`S`	$\lambda x \lambda y \lambda z\ (x\ z\ (y\ z))$
`@`	$\lambda f \lambda x\ (f\ x)$
`0, 1, 2 ...`	$\lambda f \lambda x\ (f\ (f \ldots (f\ x)))$ (les entiers de Church)

constante	valeur
`succ`	$\lambda n \lambda f \lambda x\ (f\ (n\ f\ x))$
`add`	$\lambda m \lambda n \lambda g \lambda z\ (m\ g\ (n\ g\ z))$
`mult`	$\lambda m \lambda n \lambda g\ (m\ (n\ g))$
`exp`	$\lambda m \lambda n\ (n\ m)$
`mkpair`	$\lambda x \lambda y \lambda b\ (b\ x\ y)$
`fst`	$\lambda p\ (p\ (\lambda x \lambda y\ x))$
`snd`	$\lambda p\ (p\ (\lambda x \lambda y\ y))$

En guise d'exemple simple, nous allons inférer le typage de $I\ (\lambda x\ \Delta x)$. Le système commence par détailler les abréviations, et par décorer l'arbre syntaxique avec des variables fraîches et par le type de chaque noeud. Les variables de type sont de la forme "`t_i`" et plus généralement "`t_i_j_k`..." dans la suite après duplication. Les types intersection sont notés "`(t_1,...,t_2)`" et l'intersection vide "`()`". Le système construit ensuite l'ensemble initial des contraintes :

```
Term: I (\x D x)

Tree:
((\x x:t_0):t_0 -> t_0 (\x ((\x (x:t_1 x:t_2):t_3):(t_1,t_2) -> t_3 x:t_4):t_5)
:t_4 -> t_5):t_6

Constraints:
 [ 1]  t_2 -> t_3 = t_1   [t_2]
D[ 2]  t_4 -> t_5 = (t_1,t_2) -> t_3   [t_4]
D[ 3]  (t_4 -> t_5) -> t_6 = t_0 -> t_0   [t_1,t_2,t_3,t_4,t_5]

Current proof rank: 2

[S]tep(n)   [G]o   [A]ll:
```

On rentre alors dans la boucle principale d'interaction. A chaque étape, le système donne la liste des contraintes restant à résoudre. Celles-ci sont numérotées à partir de 1, et un `D` indique celles qui peuvent être décomposées à cet instant précis. Pour information, le système indique aussi le rang courant de l'arbre de typage (si celui-ci dépasse le rang maximal admis, le programme s'arrête et conclut à l'impossibilité de typer le terme pour ce rang).

Trois commandes sont alors proposées :

- `s` permet de décomposer la première contrainte décomposable dans la liste. Il aussi possible de demander la décomposition d'une autre contrainte que celle par défaut en indiquant son numéro (par exemple, `s3`).
- `g` rentre dans un mode non-interactif, en décomposant les contraintes une à une sans demander l'avis de l'utilisateur, jusqu'à finir le typage ou aboutir à un échec. Cette commande revient à taper systématiquement `s`.
- `a` rentre dans un mode non-interactif dans lequel toutes les stratégies de décomposition possibles sont testées de façon récursive. Etant donné qu'il y en a autant que de réductions normalisantes, l'explosion combinatoire réserve cette commande à de la vérification pour des petits termes (à l'origine dans un but de débogage de l'interpréteur).

Nous allons ici choisir de décomposer d'abord la 3^{e} contrainte. Le système donne les substitutions que cela induit et met à jour la liste des contraintes :

```
[S]tep(n)   [G]o   [A]ll: s3

Decomposing: (t_4 -> t_5) -> t_6 = t_0 -> t_0   [t_1,t_2,t_3,t_4,t_5]
Substituting: t_6 => t_0   []
Substituting: t_0 => t_4 -> t_5   [t_1,t_2,t_3,t_4,t_5]

Constraints:
 [ 1]  t_2 -> t_3 = t_1   [t_2]
D[ 2]  t_4 -> t_5 = (t_1,t_2) -> t_3   [t_4]

Current proof rank: 2
```

```
[S]tep(n)   [G]o   [A]ll:
```

Ici, une seule décomposition est possible ; elle conduit cette fois-ci à une duplication, ce que le système nous signale.

```
[S]tep(n)   [G]o   [A]ll: s

Decomposing: t_4 -> t_5 = (t_1,t_2) -> t_3   [t_4]
Duplicating: dupl(2,t_4)
Substituting: t_5 => t_3   []
Substituting: t_1 => t_4_1   [t_4_1]
Substituting: t_2 => t_4_2   [t_4_2]

Constraints:
 [ 1]  t_4_2 -> t_3 = t_4_1   [t_4_2]

Current proof rank: 3

[S]tep(n)   [G]o   [A]ll:
```

Enfin, la dernière contrainte n'est pas décomposable par (R_n) ou (R_0) ; un dernier `s` va donc appliquer la règle (R_f) pour résoudre les contraintes restantes :

```
[S]tep(n)   [G]o   [A]ll: s

Substituting: t_4_1 => t_4_2 -> t_3

Current proof rank: 3
```

Ensuite, le système commence par récapituler toutes les substitutions et duplications effectuées (il s'agit du $\overline{S}$ global) :

```
Substitutions:
t_6 => t_0
t_0 => t_4 -> t_5
dupl(2,t_4)
t_5 => t_3
t_1 => t_4_1
t_2 => t_4_2
t_4_1 => t_4_2 -> t_3
```

Puis, il termine en donnant l'arbre de typage complet pour le terme, sous la forme d'une déduction logique : chaque ligne numérotée est soit un axiome, soit se déduit à partir des lignes précédentes dont les numéros sont donnés. Les rangs de l'arbre de typage et du type à la racine sont également mentionnés.

```
Proof:
(1): x:((t_4_2 -> t_3),t_4_2) -> t_3 |- x : ((t_4_2 -> t_3),t_4_2) -> t_3
(2): (1) => |- \x x : (((t_4_2 -> t_3),t_4_2) -> t_3) -> ((t_4_2 -> t_3),t_4_2)
 -> t_3
(3): x:t_4_2 -> t_3 |- x : t_4_2 -> t_3
(4): x:t_4_2 |- x : t_4_2
(5): (3) & (4) => x:t_4_2 -> t_3, x:t_4_2 |- x x : t_3
```

```
(6): (5) => |- \x x x : ((t_4_2 -> t_3),t_4_2) -> t_3
(7): x:t_4_2 -> t_3 |- x : t_4_2 -> t_3
(8): x:t_4_2 |- x : t_4_2
(9): (6) & (7) & (8) => x:t_4_2 -> t_3, x:t_4_2 |- (\x x x) x : t_3
(10): (9) => |- \x (\x x x) x : ((t_4_2 -> t_3),t_4_2) -> t_3
(11): (2) & (10) => |- (\x x) (\x (\x x x) x) : ((t_4_2 -> t_3),t_4_2) -> t_3

Proof rank: 3
Final type rank: 2
```