lambda-calcul typé

Yves Bertot

Février 2008

1 Informations sur les fonctions

Dans le cours précédent, nous avons construit des fonctions et nous les avons considérées comme des objets, de façon à pouvoir les passer en argument à d'autres fonctions sans nous poser de question. La même fonction a d'ailleurs été utilisée pour effectuer des calculs sur les entiers ou effectuer des calculs sur les paires. Avec un λ-terme entre les mains, il était généralement impossible de savoir si ce terme représentait un nombre ou une valeur booléenne. Programmer dans un tel langage est excessivement difficile : la moindre erreur de programmation peut mener à des données complètement absurdes.

Nous avons utilisé des fonctions pour représenter des données. Un avantage majeur de cette approche est que le langage que nous avons construit ainsi ne présentait aucune erreur à l'exécution. La seule chose qui puisse arriver de mal était que l'on tente de normaliser une expression qui n'a pas de forme normale.

Ce problème est quand même assez grave : 0 et la fonction factorielle sont deux expressions intéressantes pour un programmeur, mais la première a une forme normale (on peut se lancer sans risque dans sa normalisation), tandis que la seconde ne possède pas de forme normale en effet cette expression a la caractéristique suivante :

$$Y\ factF \twoheadrightarrow Y\ factF \twoheadrightarrow Y\ factF.$$

En outre, les ordinateurs fournissent habituellement des solutions natives pour certains types de données et il est dommage de ne pas incorporer ces types de données dans un langage. Néanmoins, lorsque de tels types donnés sont incorporés, il viennent avec des fonctions qui n'acceptent pas d'être appliquées à n'importe quoi. Dans la majeure partie des langages de programmation, l'expression $fact + 3$ n'a pas de sens et sera d'ailleurs rejetée à la compilation ou à l'exécution. C'est ce problème que l'introduction de types vise à réduire.

2 Le langage des types

Nous considérons que nous disposons d'un ensemble P de types primitifs. Par exemple, P pourra contenir les types `int`, `bool`, `float`. Nous considérons le langage de types T défini de la façon suivante :

– tout type primitif de P est un type de T,
– si t_1 et t_2 sont deux types alors $t_1 * t_2$ est un type,
– si t_1 et t_2 sont deux types alors $t_1 \to t_2$ est un type.

Les types de la forme $t_1 * t_2$ seront utilisés pour décrire le type des couples, nous utiliserons la notation $\langle e_1, e_2 \rangle$ pour décrire la construction qui permet d'obtenir un couple. Les types de la forme $t_1 \to t_2$ seront utilisés pour décrire le type des fonctions prenant en argument des valeurs de type t_1 et retournant des valeurs de type t_2.

Par exemple, le type $\texttt{int} * \texttt{int}$ représente le type des couples de valeur entières, tandis que le type $\texttt{int} \to \texttt{int}$ représente le type des fonctions qui prennent un entier en argument et retournent un entier. Une fonction à deux argument pourra être décrite alternativement comme une fonction recevant un argument de type "couple d'entier" et retournant un entier ou comme une fonction recevant un entier et retournant une nouvelle fonction de type entier vers entier. La fonction $\texttt{curry}_{int}$ permet de passer d'une forme à l'autre. Elle a le type suivant :

$$((\texttt{int} * \texttt{int}) \to \texttt{int}) \to (\texttt{int} \to (\texttt{int} \to \texttt{int}))$$

En programmation fonctionnelle, il est fréquent de définir des fonctions à plusieurs arguments sans utiliser de couples. On obtient alors des fonctions dont le type est constitué d'une longue séquence de flèches. L'usage est d'enlever les parenthèses autour de ces flèches lorsque ces parenthèses sont à droite, ainsi le type de $\texttt{curry}_{int}$ s'écrit mieux de la façon suivante :

$$((\texttt{int} * \texttt{int}) \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$$

Comme dans le λ-calcul pur, nous verrons dans ce cours une notion de réduction. Nous la noterons $\rightsquigarrow$ pour bien la distinguer de la notion de type décrite ici.

2.1 λ-calcul simplement typé

Nous allons maintenant nous intéresser à une variante du λ-calcul ou chaque fonction définie par abstraction contient une information sur le type attendu. Les expression du λ calcul auront donc la forme suivante :

$$e ::= x | \langle e_1, e_2 \rangle | \lambda x : t.\ e | e_1\ e_2 | \mathit{fst} | \mathit{snd}$$

Nous allons chercher à déterminer le type des expressions de ce langage, mais pour le faire nous serons obligés de disposer d'informations sur le type des variables libres dans ces expressions. Pour parler de ces informations nous utiliserons un contexte, constitué d'une séquence de couples composés de variables et de types. Ces contextes seront notés avec la variable Γ, le contexte vide sera noté $\emptyset$ et le contexte auquel on ajoute le couple associant la variable x avec le type t sera noté $\Gamma, x : t$.

L'algorithme de typage est décrit par les points suivants :

1. dans le contexte $\Gamma, x : t$ l'expression x a le type t,

2. si l'expression x a le type t dans le contexte Γ et si $x \neq y$, alors l'expression x a le type t dans le contexte $\Gamma, y : t'$,
3. si l'expression e_1 a le type t_1 dans contexte Γ et l'expression e_2 a le type t_2 dans le même contexte, alors l'expression $\langle e_1, e_2 \rangle$ a le type $t_1 * t_2$ dans le contexte Γ,
4. si l'expression e a le type t' dans le contexte $\Gamma, x : t$ alors l'expression $\lambda x : t.\ e$ a le type $t \to t'$ dans le contexte Γ,
5. si l'expression e_1 a le type $t \to t'$ dans le contexte Γ et e_2 a le type t dans ce contexte, alors $e_1\ e_2$ a le type t' dans ce contexte,
6. pour tous types t_1 et t_2 l'expression *fst* a le type $t_1 * t_2 \to t_1$ dans tout contexte Γ,
7. pour tous types t_1 et t_2 l'expression *snd* a le type $t_1 * t_2 \to t_2$ dans tout contexte Γ.

Donnons quelques exemples :

- l'expression y a le type `int` dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int}$, (par application de la règle 1 ci-dessus),
- l'expression x a le type `int` dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int}$ (par application des règles 2 et 1),
- l'expression $\langle x, y \rangle$ a le type $\texttt{int} * \texttt{int}$ dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int}$, $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int}$ (règle 3),
- l'expression f a le type $\texttt{int} * \texttt{int} \to \texttt{int}$ dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int}$ (règles 2 et 1),
- l'expression $f\ \langle x, y \rangle$ a le type `int` dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int}$ (règle 5),
- l'expression $\lambda y : \texttt{int}.\ f\langle x, y \rangle$ a le type $\texttt{int} \to \texttt{int}$ dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}$ (règle 4),
- l'expression $\lambda x : \texttt{int}.\ \lambda y : \texttt{int}.\ f\langle x, y \rangle$ a le type $\texttt{int} \to \texttt{int} \to \texttt{int}$ dans le contexte $\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}$,
- $\lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y \rangle$ a le type $(\texttt{int} * \texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}$ dans tout contexte Γ.

L'usage est de décrire ces règles de typage en utilisant le style de règle d'inférence emprunté à la logique. Les règles 1 à 7 se retrouvent sous la forme suivante :

$$\frac{}{\Gamma, x : t \vdash x : t}\ (1) \qquad \frac{\Gamma \vdash x : t \qquad x \neq y}{\Gamma, y : t \vdash x : t}\ (2)$$

$$\frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2}{\Gamma \vdash \langle e_1, e_2 \rangle : t_1 * t_2}\ (3)$$

$$\frac{\Gamma, x : t \vdash e : t'}{\Gamma \vdash \lambda x : t.\ e : t \to t'}\ (4)$$

$$\frac{\Gamma \vdash e_1 : t \to t' \qquad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1\ e_2 : t'}\ (5)$$

$$\frac{}{\Gamma \vdash \mathit{fst} : t_1 * t_2 \to t_1}\ (6) \qquad \frac{}{\Gamma \vdash \mathit{snd} : t_1 * t_2 \to t_2}\ (7)$$

Lorsque l'on représente les règles de typage de cette manière, il est possible de décrire également le procédé complet de typage d'une expression par un figure, que l'on appelle un arbre de dérivation.

Par exemple le typage de $\lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle$ peut être représenté par la dérivation suivante :

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, \cdots, x : \texttt{int} \vdash x : \texttt{int}}\,(1)}{\Gamma, \cdots \vdash x : \texttt{int}}\,(2) \quad \dfrac{}{\Gamma, \cdots, y : \texttt{int} \vdash y : \texttt{int}}\,(1)}{\Gamma, \cdots \vdash \langle x, y\rangle : \texttt{int} * \texttt{int}}\,(3)}{\Gamma, \cdots \vdash f : \texttt{int} * \texttt{int} \to \texttt{int}}\,(2)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int}, y : \texttt{int} \vdash f\langle x, y\rangle}\,(5)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int}, x : \texttt{int} \vdash \lambda y : \texttt{int}.\ f\langle x, y\rangle : \texttt{int} \to \texttt{int}}\,(4)}{\Gamma, f : \texttt{int} * \texttt{int} \to \texttt{int} \vdash \lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle : \texttt{int} \to \texttt{int} \to \texttt{int}}\,(4)}{\Gamma \quad \vdash \lambda f : \texttt{int} * \texttt{int} \to \texttt{int}.\lambda x : \texttt{int}.\lambda y : \texttt{int}.\ f\langle x, y\rangle : (\texttt{int} * \texttt{int} \to \texttt{int}) \to \texttt{int} \to \texttt{int} \to \texttt{int}}\,(4)$$

La raison pour laquelle on emploie un formalisme issu de la logique est reliée au théorème suivant : *si l'on remplace les flèches par des implications et les étoiles par des conjonctions, alors toutes les formules qui sont le type d'une expression du λ-calcul typé dans le contexte vide sont vraies.* C'est aussi pour cette raison que le système restreint à S et K porte le nom de *logique combinatoire.*

2.2 Réduction typée

Pour équiper le λ-calcul typé d'une réduction, nous allons nous contenter de considérer que les termes du λ-calcul typé se réduisent comme les termes du λ-calcul non typé, par β-réduction. Une propriété notable de la réduction est un théorème de stabilité : la réduction d'un terme se fait à type constant, ce qui peut se démontrer par récurrence sur la taille des expressions considérées.

Nous n'allons pas détailler cette preuve, mais seulement en vérifier deux lemmes.

Premièrement, si e_1 a le type $t \to t'$ et se réduit en e_1' de même type, alors l'expression $e_1\ e_2$ est bien typée si et seulement si $e_1'\ e_2$ est bien typée.

Deuxièmement, si e_1 a le type t' dans le contexte $\Gamma, x : t$ et si e_2 a le type t alors, l'expression $(\lambda x : t.\ e_1)e_2$ est bien typée et a le type t' dans le contexte Γ. Il reste à vérifier que $e_1[e_2/x]$ est bien typé dans le contexte Γ, ce qui est naturel pour les deux raisons suivantes :

1. Dans $e_1[e_2/x]$ la variable x n'apparait plus, donc la déclaration $x : t$ n'est plus nécessaire dans le contexte de typage,
2. on remplace la variable x de type t par une expression e_2 de type t.

2.3 Terminaison des réductions

Si on observe la règle 5 qui régit le typage de l'application d'une fonction on s'aperçoit que le type de la fonction est un terme strictement plus

grand que le type de l'argument. Pour cette raison, il n'est pas possible de typer une expression de la forme $x\ x$. Pour cette raison, l'expression $\Delta \equiv \lambda x.\ x\ x$ n'est pas typable et donc pas l'expression $\Omega \equiv \Delta\Delta$, ni l'expression $Y = (\lambda zx.\ x(z\ z\ x))\lambda zx.\ x(z\ z\ x)$. Les expressions que nous avons vues dans le cours sur le λ-calcul pur qui présentent des dérivations infinies ne sont pas typables. En fait ceci se généralise en un théorème très important : *toutes les expressions typables du λ-calcul simplement typé sont fortement normalisantes.* Dit autrement, il n'y a jamais de dérivations infinies avec le λ-calcul simplement typé. La démonstration de ce théorème est assez complexe et nous ne l'aborderons pas dans ce cours. Une démonstration de ce théorème peut être trouvée dans [3], on trouve également une généralisation de ce résultat dans [4].

Le fait d'avoir éliminé les causes de non-termination peut sembler un grand progrès, mais en chemin on perd la possibilité de définir des fonctions récursives. Une première solution est de ré-introduire la terminaison tout en conservant le typage en ajoutant une constante Y au langage, qui va correspondre au combinateur de point fixe tel que nous l'avons vu dans notre étude du λ-calcul pur. Cette constante doit présenter la réduction suivante :

$$(Y\ f) \rightarrow f(Y\ f)$$

Suivant les différentes expressions, Y doit être une fonction de forme $\theta \rightarrow \psi$. D'après le membre gauche, l'argument doit être une fonction donc $\theta = \sigma \rightarrow \sigma'$. Si nous voulons encore disposer de la propriété de stabilité des types, il faut également assurer que le type de $Y\ f$ coincide avec le type de $f(Y\ f)$ donc $\sigma' = \psi$. Enfin, pour que $f(Y\ f)$ soit bien typé il faut en outre que le type de $(Y\ f)$ coïncide avec le type en entrée de f, soit $\alpha = \psi$. Tout réuni, nous trouvons que Y doit avoir le type $(t \rightarrow t) \rightarrow t$. Nous ajoutons donc la règle de typage *pour tout type t, Y a le type $(t \rightarrow t) \rightarrow t$.*

Par exemple, nous pouvons travailler dans le contexte suivant :

Γ, *plus*, *sub*, *mult* : $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$, *le* : $\texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{bool}$,
if : $\texttt{bool} \rightarrow \texttt{int} \rightarrow \texttt{int} \rightarrow \texttt{int}$

et définir la fonction factorielle, de façon bien typée, en utilisant la même approche que pour le λ-calcul pur. On construit d'abord une fonctionnelle $factF$ de type

$$factF : (\texttt{int} \rightarrow \texttt{int}) \rightarrow (\texttt{int} \rightarrow \texttt{int})$$

$$factF \equiv \lambda f : \texttt{int} \rightarrow \texttt{int}.\lambda x : \texttt{int}.\ \texttt{if}(\texttt{le}\ x\ 0)\ 1\ (\texttt{mult}\ x\ (f\ (\texttt{sub}\ x\ 1)))$$

Ajouter de la sorte un opérateur de point fixe apporte donc le retour de la récursion générale et de la possibilité pour les programmes de « boucler ». Cette approche avec opérateur de point fixe est celle fournie dans la majeure partie des langages fonctionnels typés, dont `ML`, `OCaml` et `Haskell` sont les exemples les plus connus.

Par exemple, en `OCaml` la construction

$$\texttt{let rec}\ f\ x\ \texttt{=}\ e\ \texttt{in}\ e_1$$

serait équivalente à une construction de la forme suivante (sauf que la fonction *fix* n'est pas donnée dans le langage, il faut la définir) :

$$\texttt{let } f \texttt{ = } \textit{fix} \texttt{ (fun } f \texttt{ -> (fun x -> } e \texttt{ in } e_1\texttt{))}$$

2.4 Types récursifs et récursion primitive

Une approche alternative pour maintenir la propriété de terminaison dans les langages est d'autoriser l'utilisateur à définir de nouveaux types de données en fournissant pour chacun des types construits un opérateur de récursion spécialisé. Nous n'allons présenter cette technique que sur un exemple.

On considère le type `bin` des arbres binaires construit par les constantes `leaf` et `node` de la façon suivante :

1. `leaf` a le type `bin`.
2. si n est une valeur de type `int` et t_1 et t_2 sont des valeurs de type `bin` alors `node` n t_1 t_2 est une valeur de type `bin`.

Plutôt que de fournir un récurseur Y pour tous les besoins possibles, nous proposons de fournir un opérateur R_b spécialisé pour les calculs récursifs sur les arbres binaires. Pour exprimer cette utilisation, nous allons donner à R_b le type suivant, pour tout type t :

$$R_b : t \to (\texttt{int} \to \texttt{bin} \to t \to \texttt{bin} \to t \to t) \to \texttt{bin} \to t$$

et les règles de réduction suivantes :

$$R_b\ v\ f\ \texttt{leaf} \to v \qquad R_b\ v\ f\ (\texttt{node}\ n\ t_1\ t_2) \to f\ n\ t_1\ (R_b\ v\ f\ t_1)\ t_2\ (R_b\ v\ f\ t_2)$$

Intuitivement, on autorise seulement les appels récursifs sur les sous-arbres d'un arbre binaire. Les sous-arbres d'un arbre binaire sont nécessairement plus petit que lui, en n'autorisant ainsi que des appels récursifs qui décroissent dans la structure des arbres, on assure que les calculs récursifs terminent toujours. On parle de récurrence structurelle.

Par exemple on peut décrire la fonction qui additionne toutes les valeurs dans un arbre par la fonction suivante :

$$R_b\ 0\ (\lambda n : \texttt{int}, t_1 : \texttt{bin}, v_1 : \texttt{int}, t_2 : \texttt{bin}, v_2 : \texttt{int}.(\textit{plus}\ n(\textit{plus}\ v_1\ v_2)))$$

Les nombres naturels sont un cas particulier de types récursifs, construits à l'aide des opérateurs 0 de type `nat` et *succ* de type `nat` $\to$ `nat`, le récurseur pour les nombres naturels prend alors la forme suivante :

$$R_{\mathbb{N}} : t \to (\texttt{nat} \to t \to t) \to \texttt{nat} \to t$$

$$R_{\mathbb{N}}\ v\ f\ 0 \to v \qquad R_{\mathbb{N}}\ v\ f\ (\textit{succ}\ n) \to f\ n\ (R_{\mathbb{N}}\ v\ f\ n)$$

Si on limite le type t aux types non fonctionnels (c'est-à-dire les types construits seulement avec les types primitifs et la construction de couples $\langle \cdot, \cdot \rangle$),

alors cet opérateur de récursion fournit exactement la récursion primitive, très étudiée dans les cours de calculabilité.

Les récurseurs associés aux types de données ne se retrouvent dans pratiquement aucun véritable langage de programmation. On retrouve néanmoins ces notions dans les systèmes de démonstration par ordinateurs dont l'objectif est de démontrer des propriétés de programmes fonctionnels, voir par exemple le système Coq [1] pour lequel un ouvrage en cours de publication peut être trouvé à l'adresse suivante :

`ftp://ftp-sop.inria.fr/lemme/Yves.Bertot/bouquin/coqart.ps.gz`

3 Inférence de types

La programmation fortement typée apporte une assurance supplémentaire, mais la nécessité de fournir le type de toutes les variables a rapidement été ressenti comme une contrainte trop lourde. Pour alléger cette contrainte on peut étendre le langage avec une λ-abstraction non typée $\lambda x.\ e$ et lui donner la règle de typage : *pour tous types* t *et* t'*, si* e *a le type* t' *dans le contexte* $\Gamma, x : t$ *alors* $\lambda x.\ e$ *a le type* $t \to t'$. En fait, nous généralisons ici aux abstractions un procédé que nous nous étions déjà autorisé pour les projecteurs des couples *fst* et *snd*, pour l'opérateur de point fixe Y et pour les récurseurs associés à des types récursifs.

Lorsque l'on considère une expression composée avec plusieurs abstractions non typées, la question que l'on se pose est de choisir un type pour chacune de ces abstractions typées de façon que l'expression entière soit bien typée. Une solution est de se reposer sur un algorithme d'unification, après avoir donné à chaque abstraction non typée un type variable. Le programme de vérification de type remplace la vérification d'égalité entre deux types par l'ajout d'une équation dans un jeu de contraintes à résoudre. Bien sûr on donne également un type à toutes les occurrences de constantes polymorphes apparaissant dans le terme, par exemple toute occurrence de *fst* reçoit le type $T_1 * T_2 \to T_1$, où T_1 et T_2 sont de nouvelles variables, on fait de même avec *snd* et Y, si cette constante fait partie du langage. Ensuite on effectue le typage comme d'habitude, sauf que l'on ajoute une contrainte dans une liste de contraintes pour chaque application, où il faut vérifier que l'argument de l'application a bien un type égal au type attendu par la fonction.

Par exemple observons le typage de l'expression $\lambda x.(plus\ x\ x)$, l'annotation avec des variables de types donne l'expression suivante, $\lambda x : T.\ (plus\ x\ x)$ et le typage avance de la manière suivante :

1. on démarre avec une liste de contraintes vides,
2. *plus* a le type $\texttt{int} \to \texttt{int} \to \texttt{int}$ dans le contexte $\emptyset, x : T$,
3. x a le type T dans le contexte $\emptyset, x : T$,
4. l'application *plus* x est bien typée dans le contexte $\Gamma, x : T$ si $\texttt{int} = T$ (on ajoute donc cette contrainte à la liste de contraintes), cette expression a le type $\texttt{int} \to \texttt{int}$,

5. l'application $plus\ x\ x \equiv (plus\ x)\ x$ est bien typée dans le contexte $\Gamma, x : T$ si $\texttt{int} = T$, cette expression a le type `int`
6. l'expression $\lambda x : T.\ (plus\ x\ x)$ est bien typée si les contraintes $\{\texttt{int} = T, \texttt{int} = T\}$ ont une solution σ et a alors le type $\sigma(T) \to \texttt{int}$.

La solution d'un système de contraintes, lorsqu'elle existe est une subtitution d'un type pour chacune des variables de types, ici la solution est la substitution qui remplace T par `int`. L'expression considérée a donc le type $\texttt{int} \to \texttt{int}$.

La résolution des contraintes se fait à l'aide d'un algorithme d'unification, déjà connu dans le domaine de la démonstration sur ordinateur et de Prolog.

4 Typage polymorphe

Lorsque l'on dispose d'une valeur de type t et d'une fonction de type $t \to t$, on peut vouloir appliquer cette fonction deux fois sur cette valeur, ce que l'on écrira de la façon suivante :

$$\lambda v\ f.f(f\ v).$$

Cette fonction travaille toujours de la même façon, quel que soit le type t considéré, qui ne joue au fond aucun rôle. On pourrait donc vouloir appliquer la même fonction en deux endroits différents, une fois pour le type `int` et une fois pour le type `bool`, par exemple, l'expression choisie aurait la forme suivante :

$\lambda c.(\lambda g.\lambda b : \texttt{bool}, f_1 : \texttt{bool} \to \texttt{bool}, n : \texttt{int}, f_2 : \texttt{int} \to \texttt{int}.c\ (g\ b\ f_1)\ (g\ n\ f_2))$
$\lambda v\ f.f(f\ v)$.

Cette expression n'est bien typée, car l'une des utilisation de g impose que le type de v soit `bool` tandis que l'autre utilisation impose que le type de v soit `int`. Ici, il semble que le typage impose une duplication de code.

La solution de ce problème est de généraliser à du code écrit par l'utilisateur la solution fournie pour *fst*, *snd* ou Y. Plutôt que d'exprimer que la fonction $\lambda v\ f.\ f(f\ v)$ a le type $T \to (T \to T) \to T$ pour un T donné, nous voulons exprimer que cette fonction a le type $t \to (t \to t) \to t$ pour tout type t. Ainsi, certaines fonctions peuvent avoir un type quantifié universellement, on parle alors de type *polymorphe*.

Nous allons maintenant ajouter une nouvelle construction dans notre langage, qui aura la syntaxe suivante :

$$\mathsf{let}\ x = e\ \mathsf{in}\ e'$$

Opérationnellement, le sens à donner à une telle expression est celui d'un redex, c'est à dire que le comportement à l'exécution des deux expressions suivantes devrait être similaire :

$$\mathsf{let}\ x = e\ \mathsf{in}\ e' \sim (\lambda x.\ e')e$$

Pour le typage néanmoins, le typage polymorphe s'exprime par la règle suivante :

$$\frac{\Gamma \vdash e : \theta \quad \Gamma \vdash e'[e/x] : t}{\Gamma \vdash \mathsf{let}\ x = e\ \mathsf{in}\ e' : t}$$

Donc pour typer une expression $\mathsf{let}\ x = e\ \mathsf{in}\ e'$ dans un contexte Γ il faut d'abord vérifier si l'expression e est bien typée dans ce contexte et ensuite vérifier si chacune des utilisations de e est bien typée, indépendamment des autres utilisations. Cette règle de typage effectue donc la duplication de code avant de faire la vérification de typage, ce qui permet d'éviter aux programmeurs d'effectuer cette duplication eux-mêmes.

Cet algorithme de typage n'est pas très efficace, car il fait appel à une substitution qui peut provoquer une croissance excessive de la taille du terme à typer. On trouvera dans [2] un algorithme plus efficace, que G. Dowek décrit dans son cours[1].

Le typage polymorphe est la discipline de type fournie dans les langages comme ML, OCaml, ou Haskell. Ces langages permettent également de faire des définitions de types récursifs polymorphes, par exemple les listes d'objets de type arbitraire (mais tous de même type), les arbres binaires contenant des objets de type arbitraire, etc. Le typage s'adapte également très bien à une structure de contrôle fournie pour les types récursifs, que l'on appelle l'appel par filtrage. L'appel par filtrage est l'un des points forts des langages fonctionnels typés, mais nous n'aurons malheureusement pas le temps de le décrire dans ce cours.

Les langages fonctionnels de la famille de ML (OCaml, etc.) possèdent également des variables mutables, proches de ce que l'on trouve dans les langages impératifs, comme C ou FORTRAN, mais ces aspects impératifs seront étudiés dans d'autres cours.

Références

[1] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner. The Coq Proof Assistant Reference Manual – Version V6.1. Technical Report RT-0203, INRIA, August 1997. revised version distributed with Coq.

[2] Luís Damas and Robin Milner. Principal type-schemes for functional programs. In *ninth ACM symposium on Principles of Programming Languages*, pages 207–212. ACM, 1982.

[3] Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and types*. Cambridge University Press, 1989.

[4] Jean-Louis Krivine. *Lambda Calcul, types et modèles*. Masson, 1990.

[1] `http ://pauillac.inria.fr/~dowek/Cours/tlp.ps.gz`