Examen du cours "Typage et programmation"

D.E.A. "Programmation"

20 décembre 2001

Exercice 1

Le but de cet exercice est d'ajouter au le langage mini-ML un mécanisme pour faire de l'abstraction de types qui est plus simple, mais moins puissant, que les mécanismes d'abstraction de types au niveau des modules vus dans le dernier cours.

Pour ce faire, nous introduisons dans l'algèbre de types la possibilité de quantifier existentiellement des variables de types :

Types : $\tau ::= \ldots \mid \exists\alpha.\ \tau$

Intuitivement, un terme a le type $\exists\alpha.\ \tau$ s'il existe un certain type τ' tel que le terme est de type $\tau[\alpha \leftarrow \tau']$. Cependant, le reste du programme ne connait pas ce type τ', et donc doit utiliser le terme de type $\exists\alpha.\ \tau$ de manière polymorphe par-rapport à α, c'est-à-dire en supposant que α est distinct de tous les autres types, y compris τ'.

Pour construire et utiliser ces types "existentiels", nous ajoutons deux constructions `pack` et `open` au langage mini-ML polymorphe :

Expressions : $a ::= \ldots \mid \texttt{pack}(a) \mid \texttt{open}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2$
Valeurs : $v ::= \ldots \mid \texttt{pack}(v)$
Contextes : $\Gamma ::= \ldots \mid \texttt{pack}(\Gamma) \mid \texttt{open}\ \Gamma\ \texttt{as}\ x\ \texttt{in}\ a$

Les règles de réduction et de typage de ces constructions sont :

$$\texttt{open}\ (\texttt{pack}\ (v))\ \texttt{as}\ x\ \texttt{in}\ a \stackrel{\varepsilon}{\rightarrow} a[x \leftarrow v] \qquad (\beta_{open})$$

$$\frac{E \vdash a : \tau[\alpha \leftarrow \tau']}{E \vdash \texttt{pack}(a) : \exists\alpha.\tau}\ (\text{pack})$$

$$\frac{E \vdash a_1 : \exists\alpha.\ \tau' \qquad E + \{x : \tau'\} \vdash a_2 : \tau \qquad \alpha \notin \mathcal{L}(E) \qquad \alpha \notin \mathcal{L}(\tau)}{E \vdash \texttt{open}\ a_1\ \texttt{as}\ x\ \texttt{in}\ a_2 : \tau}\ (\text{open})$$

(Rappel : $\mathcal{L}$ désigne l'ensemble des variables de types libres dans un type ou un environnement de typage.)

Exemple : on souhaite construire une liste de paires (valeur imprimable, fonction d'impression), où les valeurs peuvent être de différents types. C'est impossible en mini-ML, mais devient possible avec les types existentiels, en considérant chacune des paires avec le type $\exists\alpha.\ \alpha \times (\alpha \rightarrow \texttt{unit})$.

```
let l = [ pack(42, print_int); pack("Hello", print_string) ] in
let print = fun p -> open p as x in snd(p)(fst(p)) in
List.iter print l
```

On admettra que les propriétés 1.2, 1.3, 1.4 et 2.2 énoncées dans le cours restent vraies lorsqu'on ajoute les types existentiels.

Question 1. On souhaite donner une sémantique "à grands pas" pour mini-ML + existentiels, dans le style du prédicat $a \xrightarrow{v} v$ du chapitre 1 des notes. Donner sans justification les règles définissant ce prédicat pour les nouvelles constructions `pack` et `open`.

Question 2. Montrer que la règle de réduction (β_{open}) préserve le typage : si $E \vdash a : \tau$ et $a \xrightarrow{\varepsilon} a'$ par la règle (β_{open}), alors $E \vdash a' : \tau$.

Question 3. Supposons $\emptyset \vdash v : \exists \alpha.\ \tau$. Que dire de la valeur v ?

Question 4. Montrer la propriété de progression pour mini-ML + types existentiels : si $\emptyset \vdash a : \tau$, alors ou bien a est une valeur, ou bien il existe a' telle que $a \to a'$. (Inutile de re-démontrer les cas qui ne changent pas par-rapport à la preuve de la progression pour mini-ML.)

Question 5. Justifier la restriction $\alpha \notin \mathcal{L}(\tau)$ dans la règle de typage de la construction `open`. Pour ce faire, enlevez cette condition dans la règle de typage et construisez un terme bien typé dont l'évaluation échoue.

Question 6. Justifier de même la restriction $\alpha \notin \mathcal{L}(E)$ dans la règle de typage de la construction `open`.

Exercice 2

Le but de cet exercice est d'étudier une technique de compilation des langages avec sous-typage, basée sur l'insertion de conversions du sous-type vers le super-type. De ce point de vue, on peut distinguer deux sortes de sous-typage :

- Sans changement de représentation : par exemple, supposons un type `pos` des entiers positifs, qui est sous-type du type `int` des entiers relatifs. Vraisemblablement, un entier positif va avoir la même représentation en machine qu'il soit vu avec le type `pos` ou avec le type `int`. Donc, il n'est pas nécessaire de produire du code de conversion pour une étape de subsomption implicite faisant passer de a : `pos` à a : `int`.
- Avec changement de représentation : c'est le cas par exemple si l'on considère le type `int` comme sous-type du type `float` des nombres flottants. L'entier 3 n'a pas la même représentation dans la machine que le flottant 3.0. Il faut donc, à l'exécution, effectuer une conversion entier vers flottant lorsqu'une expression a : `int` est vue, par subsomption implicite, comme a : `float`.

Dans cet exercice, nous allons voir comment le compilateur peut insérer automatiquement les bonnes conversions entier-flottant en s'appuyant sur les informations de typage du programme.

Le langage source a la syntaxe suivante :

Expressions : $a ::= x \mid c \mid op \mid \texttt{fun}\ x \to a \mid a_1\ a_2$

Opérateurs : $op ::= \texttt{floatofint} \mid \dots$

Types : $\tau ::= \texttt{int} \mid \texttt{float} \mid \tau_1 \to \tau_2$

Pour la relation de sous-typage $\tau_1 <: \tau_2$, nous disons que `int` est sous-type de `float` et étendons ce sous-typage aux types de fonctions de la manière standard.

$$\texttt{int} <: \texttt{float} \qquad \texttt{int} <: \texttt{int} \qquad \texttt{float} <: \texttt{float} \qquad \frac{\sigma' <: \sigma \quad \tau <: \tau'}{\sigma \to \tau <: \sigma' \to \tau'}$$

Si $\tau <: \tau'$, nous définissons la conversion de τ vers τ', notée $\mathcal{C}(\tau, \tau')$, de la manière suivante :

$$\begin{array}{rcl}
\mathcal{C}(\texttt{int}, \texttt{float}) & = & \texttt{floatofint} \\
\mathcal{C}(\texttt{int}, \texttt{int}) & = & \texttt{fun}\ x \to x \\
\mathcal{C}(\texttt{float}, \texttt{float}) & = & \texttt{fun}\ x \to x \\
\mathcal{C}(\sigma \to \tau, \sigma' \to \tau') & = & \texttt{fun}\ f \to \texttt{fun}\ x \to \mathcal{C}(\tau, \tau')(f(\mathcal{C}(\sigma', \sigma)(x)))
\end{array}$$

Enfin, le prédicat de typage et de traduction $E \vdash a : \tau \Longrightarrow b$ est défini par les règles ci-dessous. Ce prédicat est semblable au prédicat de typage habituel $E \vdash a : \tau$, mais calcule en plus une expression b (la “traduction” de a) qui est l'expression source a dans laquelle on a inséré les conversions `floatofint` explicitant les étapes de sous-typage effectuées durant le typage (voir la règle (sub)).

$$E \vdash x : E(x) \Longrightarrow x \ \text{(var)} \qquad E \vdash c : TC(c) \Longrightarrow c \ \text{(const)} \qquad E \vdash \mathit{op} : TC(\mathit{op}) \Longrightarrow \mathit{op} \ \text{(op)}$$

$$\frac{E + \{x : \tau'\} \vdash a : \tau \Longrightarrow b}{E \vdash \texttt{fun}\ x \to a : \tau' \to \tau \Longrightarrow \texttt{fun}\ x \to b} \ \text{(fun)}$$

$$\frac{E \vdash a_1 : \tau' \to \tau \Longrightarrow b_1 \quad E \vdash a_2 : \tau' \Longrightarrow b_2}{E \vdash a_1\ a_2 : \tau \Longrightarrow b_1\ b_2} \ \text{(app)} \qquad \frac{E \vdash a : \tau \Longrightarrow b \quad \tau <: \tau'}{E \vdash a : \tau' \Longrightarrow \mathcal{C}(\tau, \tau')(b)} \ \text{(sub)}$$

Question 1. On se donne un opérateur `cos` de type `float` $\to$ `float`. Donner une dérivation de typage et de traduction de l'expression `cos`(1) où 1 est une constante de type `int`. Quelle est la traduction de cette expression ?

Question 2. Justifier la définition donnée plus haut de la conversion entre deux types fonctionnels $\mathcal{C}(\sigma \to \tau, \sigma' \to \tau')$. (On pourra prendra l'exemple d'une fonction de type `float` $\to$ `int` que l'on veut utiliser dans un contexte qui attend une fonction de type `int` $\to$ `float`.)

Question 3. On note $E \vdash b : \tau$ le système de typage simple, sans sous-typage, défini par les règles suivantes :

$$E \vdash x : E(x) \qquad E \vdash c : TC(c) \qquad E \vdash \mathit{op} : TC(\mathit{op})$$

$$\frac{E + \{x : \tau'\} \vdash b : \tau}{E \vdash \texttt{fun}\ x \to b : \tau' \to \tau} \qquad \frac{E \vdash b_1 : \tau' \to \tau \quad E \vdash b_2 : \tau'}{E \vdash b_1\ b_2 : \tau}$$

Montrer que pour tous types τ, τ' tels que $\tau <: \tau'$, et tout environnement E, on a

$$E \vdash \mathcal{C}(\tau, \tau') : \tau \to \tau'$$

(Notez bien qu'il n'y a pas de règle de subsomption dans le système de type utilisé pour typer les coercions et les traductions.).

Question 4. En déduire que si $E \vdash a : \tau \Longrightarrow b$, alors $E \vdash b : \tau$.

Cohérence de la traduction

Dans le reste de cet exercice, on s'intéresse au problème de la *cohérence* de la traduction : dans la mesure où il y a plusieurs dérivations de typage-traduction possibles pour une même expression (on peut appliquer la règle (sub) à différents endroits), on peut obtenir plusieurs traductions différentes pour une même expression de départ ; est-ce que toutes ces traductions sont cohérentes, c'est-à-dire calculent le même résultat ?

Question 5. On considère à nouveau l'expression `cos` 1. Donner une autre dérivation de typage et de traduction que celle trouvée à la question 2.1. Comparer les deux traductions ainsi obtenues.

On note $\approx$ la relation de $\beta\eta$-équivalence entre expressions. Cette équivalence est obtenue en appliquant les deux axiomes β et η ci-dessous n'importe où à l'intérieur d'un terme, y compris sous un `fun`.

$$(\texttt{fun}\ x \to a)\ b \approx a[x \leftarrow b] \ \ (\beta) \qquad\qquad (\texttt{fun}\ x \to a\ x) \approx a \ \ (\eta)$$

On admettra sans démonstration que si deux expressions sont $\beta\eta$-équivalentes, alors elles calculent le même résultat. Dans la suite de l'exercice, on va montrer que toutes les traductions possibles pour une même expression sont $\beta\eta$-équivalentes.

Question 6. On considère une dérivation de typage-traduction qui se termine par une règle (sub) suivie d'une règle (fun) :

$$\dfrac{\dfrac{\begin{array}{c}\text{dérivation } D\\ \vdots\end{array} \qquad \ldots <: \ldots}{\ldots}\ (\text{sub})}{E \vdash a : \tau \Longrightarrow b}\ (\text{fun})$$

Montrer que l'on peut permuter les règles (sub) et (fun), obtenant une dérivation de la forme suivante :

$$\dfrac{\dfrac{\begin{array}{c}\text{dérivation } D\\ \vdots\end{array}}{\ldots}\ (\text{fun}) \qquad \ldots <: \tau}{E \vdash a : \tau \Longrightarrow b'}\ (\text{sub})$$

Montrer que $b' \approx b$.

Question 7. Même question que 2.5 si la dérivation se termine par une règle (sub) à gauche d'une règle (app) :

$$\dfrac{\dfrac{\begin{array}{c}\text{dérivation } D_1\\ \vdots\end{array} \qquad \ldots <: \ldots}{\ldots}\ (\text{sub}) \qquad \begin{array}{c}\text{dérivation } D_2\\ \vdots\end{array}}{E \vdash a : \tau \Longrightarrow b}\ (\text{app})$$

Mettre cette dérivation sous la forme suivante :

$$\dfrac{\dfrac{\begin{array}{c}\text{dérivation } D_1\\ \vdots\end{array} \qquad \dfrac{\begin{array}{c}\text{dérivation } D_2\\ \vdots\end{array} \qquad \ldots <: \ldots}{\ldots}\,(\text{sub})}{\ldots}\,(\text{app}) \qquad \ldots <: \tau}{E \vdash a : \tau \Longrightarrow b'}\,(\text{sub})$$

et montrer que $b' \approx b$.

Question 8. Montrer la transitivité du sous-typage : si $\tau_1 <: \tau_2$ et $\tau_2 <: \tau_3$, alors $\tau_1 <: \tau_3$ et de plus la conversion $\mathcal{C}(\tau_1, \tau_3)$ est $\beta\eta$-équivalente à la composition des conversions $\mathcal{C}(\tau_1, \tau_2)$ et $\mathcal{C}(\tau_2, \tau_3)$:

$$\mathcal{C}(\tau_1, \tau_3) \approx \texttt{fun}\ x \to \mathcal{C}(\tau_2, \tau_3)\ (\mathcal{C}(\tau_1, \tau_2)(x))$$

(Indication : montrer qu'il n'y a que trois cas à considérer : cas (1) $\tau_1 = \tau_2$, cas (2) $\tau_2 = \tau_3$, et cas (3) τ_1, τ_2 et τ_3 sont trois types de fonctions.)

Question 9. On appelle dérivation normalisée de $E \vdash a : \tau \Longrightarrow b$ une dérivation dans laquelle la règle (sub) ne peut apparaître que juste au dessus à droite de la règle (app), pour sous-typer le type de l'argument de la fonction, mais nulle part ailleurs :

$$\dfrac{\begin{array}{c}\text{dérivation normalisée}\\ \vdots\end{array} \qquad \dfrac{\begin{array}{c}\text{dérivation normalisée}\\ \vdots\end{array} \qquad \ldots <: \ldots}{\ldots}\,(\text{sub})}{\ldots}\,(\text{app})$$

Remarquez que les dérivations normalisées sont entièrement déterminées par E et a : il existe au plus un τ et un b tels qu'on puisse dériver $E \vdash a : \tau \Longrightarrow b$ avec une dérivation normalisée.

Montrer que si $E \vdash a : \tau \Longrightarrow b$, alors il existe une dérivation de cet énoncé de la forme suivante : une dérivation normalisée suivie d'une application finale de la règle (sub).

$$\dfrac{\dfrac{\begin{array}{c}\text{dérivation normalisée}\\ \vdots\end{array}}{\ldots} \qquad \ldots <: \tau}{E \vdash a : \tau \Longrightarrow b'}\,(\text{sub})$$

Montrer que de plus $b \approx b'$.

Question 10. En déduire que si $E \vdash a : \tau \Longrightarrow b_1$ et $E \vdash a : \tau \Longrightarrow b_2$, alors $b_1 \approx b_2$.