Cours de Compilation-Exercices

Analyse sémantique avancée

Master d'Informatique M1 2010–2011

1 décembre 2010

1 Surcharge à la Java

Dans Java, la surcharge est résolue statiquement en ne prenant en compte que le type des arguments de la méthode. À cause du sous-typage, plusieurs méthodes peuvent s'appliquer aux mêmes arguments. L'ambiguité est levée s'il existe parmi toutes les méthodes applicables exactement une méthode dont le profil est plus petit que tous les autres. Le profil d'une méthode à n arguments est formé de la classe dans laquelle la méthode est déclarée et des types des arguments. Un profil $(C, \tau_1, \ldots, \tau_n)$ est plus petit qu'un profil $(D, \sigma_1, \ldots, \sigma_p)$ si $n = p$, que C est une sous-classe de D et que chaque τ_i est un sous-type de σ_i.

1. Parmi les exemples suivants, dire s'ils sont correctement formés et si oui indiquer à quelle déclaration fait référence chaque utilisation de méthode.

   (a)
   ```
   class A {
       void g(A a) { g(a,a); }
       void g(A a, A b) {g(a); }
       void g(B b, B a) {g(a); }
   }
   class B extends A {
       void g(B b) { g(b,b); }
       void g() { g(this); }
       void g(int n) { g(new A()); }
       }
   ```

   (b)
   ```
   class A { }
   class B extends A {
       void g() { g(this,this); }
       void g(A a, B b) { }
       void g(B b, A a) { }
   }
   ```

   (c)
   ```
   class A {
       void f(B b) {  }
   }
   class B extends A {
       void f(A a) {  }
   }
   class Main {
       static void main(A a) { a.f(a); }
   }
   ```

2. On se propose d'implanter une fonction qui étant donné un profil p et un ensemble de profils lp (représenté par exemple par une liste de profils) cherche s'il existe un seul profil minimal p' dans lp qui est plus grand que p. Ce problème peut se formuler de manière plus générale, on se donne un ordre partiel $\sqsubseteq$ sur un ensemble A, en pratique une fonction booléenne. Étant donné une liste l d'éléments de A, construire une liste l' des éléments minimaux de l, ie $l' \subseteq l$, deux élements différents de l' sont incomparables et tout élément de l est plus grand qu'un élément dans l'.
   (a) Ecrire une fonction `minimal` qui effectue cette opération.
   (b) On suppose donnée une fonction qui compare deux types de Java suivant la relation de sous-typage. Un profil est représenté par une liste de types. Écrire la fonction `resoud_surcharge` qui étant donnés un profil d'utilisation d'une méthode m et une liste de profils possibles pour une méthode m, résoud la surcharge lorsque cela peut être fait de manière non ambiguë.
3. Surcharge sur les types de base. Dans Java les types de base suivent la hiérarchie suivante `byte` $\leq$ `short` $\leq$ `int`, `char` $\leq$ `int` et `int` $\leq$ `long` $\leq$ `float` $\leq$ `double`. La conversion d'un type vers un type plus grand est faite implicitement par le compilateur qui introduit si nécessaire des opérations de conversion.
   Ecrire une fonction qui étant donné un arbre de syntaxe abstraite `Op`$(e_1, \texttt{Plus}, e_2)$ produit un arbre avec l'opération adéquate (`PlusInt`, `Pluschar`, `Plusfloat`...) et les conversions associées (`Int2Float` ...)

2 Inférence de types polymorphes

Nous considérons ici un mini langage fonctionnel PtiCaml comportant uniquement les expressions suivantes :

⟨*expr*⟩	:=	⟨*simple_expr*⟩ | `function` ⟨*ident*⟩ `->` ⟨*expr*⟩ | ⟨*expr*⟩ ⟨*op*⟩ ⟨*expr*⟩
		| ⟨*simple_expr*⟩ ⟨*simple_expr*⟩$^+$ | `let` ⟨*ident*⟩ `=` ⟨*expr*⟩ `in` ⟨*expr*⟩
		| `if` ⟨*expr*⟩ `then` ⟨*expr*⟩ `else` ⟨*expr*⟩
⟨*simple_expr*⟩	:=	`(` ⟨*expr*⟩ `)` | ⟨*ident*⟩ | ⟨*const*⟩
⟨*op*⟩	:=	`+` | `-` | `+.` | `-.` | `=`
⟨*const*⟩	::=	`true` | `false` | ⟨*entier*⟩ | ⟨*réel*⟩
⟨*type*⟩	:=	`bool` | `int` | `float` | α | ⟨*type*⟩ `->` ⟨*type*⟩

1. Définir en OCaml l'arbre de syntaxe abstraite des expressions et des types.

2.1 Typage monomorphe

Dans cette première partie, nous nous limitons au typage sans polymorphisme.

1. Quel est le type de l'expression suivante :

```
let id = function x -> x in
let a = id 1 in
id true
```

2. Définir le système de type pour PtiCaml monomorphe.
3. Écrire un typeur pour PtiCaml avec typage monomorphe et dans lequel les paramètres de fonctions sont annotés par leur type (`function (`⟨*ident*⟩ `:` ⟨*type*⟩`) ->` ⟨*expr*⟩).

Nous voulons maintenant réaliser un typeur avec inférence de type (les arguments des fonctions ne sont plus annotés avec leur type).

1. Donner précisément la dérivation de typage de l'expression suivante :

```
function g -> function h -> function x -> g ((h x) + 1) + h (x + 2)
```

2. On représentera une substitution par une liste d'association. Définir une fonction `subst` qui applique une substitution à un type, une fonction `subst_env` qui applique une substitution à tous les types d'un environnement de typage et une fonction `compose` qui compose deux substitutions.
3. Définir une fonction `unify` qui calcule la substitution qui unifie deux types. `unify t1 t2` retourne une substitution `s` telle que `subst s t1 = subst s t2`.
4. Écrire un typeur pour PtiCaml avec typage monomorphe.

2.2 Typage polymorphe

Nous voulons maintenant introduire du typage polymorphe. Pour cela, l'environnement de typage associe à chaque variable un schéma de type (de la forme $\forall\alpha_1, \ldots, \alpha_n.\tau$) et les règles de typage des variables et du `let` deviennent :

$$\frac{\tau \in \texttt{inst}(\Gamma(x))}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \forall\bar{\alpha}.\tau_1 \vdash e_2 : \tau_2 \quad \bar{\alpha} = \texttt{vars}(\tau_1) \setminus \texttt{fv}(\Gamma)}{\Gamma \vdash \texttt{let } x{=}e_1 \texttt{ in } e_2 : \tau_2}$$

où `vars` est l'ensemble des variables apparaissant dans un type, `fv` est l'union des ensembles des variables libres des schémas de type présents dans un environnement et où $\texttt{inst}(\forall\alpha_1, \ldots, \alpha_n.\tau)$ est l'ensemble des types obtenus en substituant dans τ les variables α_i par des types quelconques.

1. Définir un type `schema` qui représente un schéma de type.
2. Écrire une fonction `specialize` qui à partir d'un schéma de type calcule un nouveau type où toutes les variables de type quantifiées universellement sont remplacées par des variables fraîches.
3. Écrire une fonction `generalize` qui à partir d'un environnement de typage et d'un type va calculer le schéma de type correspondant.
4. Définir la fonction `subst_sc` qui applique une substitution à un schéma de type.
5. Écrire un typeur pour PtiCaml avec typage polymorphe.