CS611 Lecture 39: Object Types 12/06/99

Lantian Zheng, Chaitanya Swamy

1 Rules for class translation

In the last lecture, the class translation rule for base classes was definied as :

$$[\![\text{class c } \{\ldots x_i.l_i = e_i \ \ldots\}]\!] \triangleq \lambda u.\{\ldots x_i.l_i = e_i \ldots\}$$

However for multiple inheritance we need a function that takes an object and returns an extended object. So we define the rules for class translation as:

$$\begin{array}{rcl} [\![\text{class c } \{x_i.l_i = e_i\}]\!] & \triangleq & \lambda o.o \texttt{ with } x_i.l_i = e_i \\ [\![\text{class c extends } c_1, c_2 ... c_n \ \{x_i.l_i = e_i\}]\!] & \triangleq & \lambda o.c_1(c_2...(c_n(o))...) \texttt{ with } x_i.l_i = e_i \\ [\![\text{new c}]\!] & \triangleq & c \ \{\} \end{array}$$

2 Typing rules for object-calculus

- **Syntax of typed object-calculus** : We introduce a new object type and modify the syntax of expressions to include type annotations.

$$\begin{array}{rcl} \tau & ::= & ... \mid [l_i.\tau_i{}^{i\in 1..n}] \\ e & ::= & ... \mid x \mid e.l \mid \{x_i : \tau_i.l_i = e_i\} \mid e \texttt{ with } x : \tau.l = e' \end{array}$$

- **Typing rules**

$$\frac{\Gamma[x \mapsto \tau] \vdash e_i : \tau_i}{\Gamma \vdash \{x_i : \tau.l_i = e_i\} : \tau} \qquad \text{where } \tau = [l_i : \tau_i{}^{i\in 1..n}]$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e.l_j : \tau_j} \qquad \text{where } \tau = [l_i : \tau_i{}^{i\in 1..n}], \quad j \in 1..n$$

$$\frac{\Gamma \vdash e : \tau \quad \Gamma[x \mapsto \tau] \vdash e' : \tau_j}{\Gamma \vdash e \texttt{ with } x : \tau.l = e' \ : \ \tau} \qquad \text{where } \tau = [l_i : \tau_i{}^{i\in 1..n}]$$

- **Subtyping rules**

  $[l_1 : \tau_1, ..., l_{n+m} : \tau_{n+m}] \leq [l_1 : \tau_1, ..., l_n : \tau_n]$

  The above rule seems too restrictive. However the following rule :

$$\frac{\forall i \in 1..n \quad \tau_i \leq \tau_i'}{[l_1 : \tau_1, ..., l_{n+m} : \tau_{n+m}] \leq [l_1 : \tau_1', ..., l_n : \tau_n']}$$

  though intuitively seems to make sense, is unsound. Let's look at an example which exposes the bug.

```
point = [x:int, y:int]
cpoint = [x:int, y:int, c:color]
t1 = [p:cpoint, col:color]
t2 = [p:point, col:color]
----------------------------
t1obj={o:t1.p = {cp:cpoint.x=1, cp:cpoint.x=2, cp:cpoint.color=red}, o:t1.col=o.p.c}}
t1obj with o':t2.p = {p':point.x = 2, p':point.y=2}.col
```

The last expression will give an error because the class `point` does not have a `c:color` field, This means that $t_1 \nleqslant t_2$ which is contrary to what the subtyping rule gives. Basically we did the following : Assuming $[p : cpoint, col : color] \leqslant [p : point, col : color]$,

1. Start with the left type, create an object of that type.
2. By subsumption the object has the type on the right too.
3. Update p with a point.

Notice that $o.col$ depends on the fact that $o.p$ is a color point. So in this case, p(in t_1) is *covariant* and cannot be updated with a *point*.

A direct way to fix this problem is to annotate the types of fields and limit the allowed operation. This is called **Variance Annotations**. In detail, we define the object type as $[l_i : \tau_i^{v_i}] - v_i$ is the annotation:

v_i	denotation	allowed operations
0	invariant subtyping	$e.l_i$, e `with` $x.l_i = e'$
$+$	covariant subtyping	$e.l_i$
$-$	contravariant subtyping	e `with` $x.l_i = e'$

The fixed subtyping rule is:

$$\frac{\forall i \in 1..n(v_i = 0 \Rightarrow \tau_i = \tau_i') \land (v_i = + \Rightarrow \tau_i \leqslant \tau_i') \land (v_i = - \Rightarrow \tau_i' \leqslant \tau_i)}{[l_1 : \tau_1^{v_1}, ..., l_{n+m} : \tau_{n+m}^{v_{n+m}}] \leqslant [l_1 : \tau_1'^{v_1}, ..., l_n : \tau_n'^{v_n}]}$$

- **Two examples**

  1. Application of contravariant subtyping
     Let's define an output stream type:

     $$\begin{aligned} Stream_T &\triangleq [d : T^-, print : (T \to u)^+] \\ Stream_S &\triangleq [d : S^-, print : (S \to u)^+] \end{aligned}$$

     We have $(T \leqslant S) \Rightarrow (Stream_S \leqslant Stream_T)$. Since d is contravariant, $e.d$ must be forbidden.
  2. Recursive types and covariant subtyping
     To add a method $move_x : int \to point$ to the type $point$, we need recursive types :

     $$\begin{aligned} Point &\triangleq \mu P.[x : int,\ y : int,\ get_x : int,\ move_x : int \to P] \\ CPoint &\triangleq \mu P.[x : int,\ y : int,\ get_x : int,\ move_x : int \to P,\ c : color] \end{aligned}$$

     If we allow updating $move_x$, we can prove $CPoint \nleqslant Point$ by constructing a counter example. One way to solve the problem is to add the covariant annotation: $move_x : (int \to P)^+$.

3 Some other issues

- Besides recursion, there are other ways to deal with self-returning methods. A natural idea is to handle the type of self in some special way. Several languages have included *SELF* (or *MYSELF*) in their type systems. In these languages, we can define $move_x : int \to SELF$. But still some ad hoc typing rules for *SELF* are needed to avoid unsoundness.
- Dynamic typing mechanism, such as *instanceof* operator in Java, turns out to be useful. (This mechanism can also deal with the conditions about subtype hierarchy.) But it is considered impure for several methodological reasons. First, it violates the object abstraction. Second, it makes code less extensible.
- Interaction between object types and parametric polymorphism : We can have polymorphism along with object types. Polymorphic object types can be $\forall \alpha.\tau$, or we could even have more complicated F-bounded polymorphism: $\forall \alpha \leqslant \tau'.\tau$. This is still an active research area.