CS611 Lecture 37 Subtype Polymorphism / Type Inclusion November 30, 2001
Scribe: Michael Brukman and Jeff Hartline Lecturer: Andrew Myers

1 Introduction

We'd like to extend the traditional typed λ-calculus to include object oriented features, such as classes and class inheritance. We will do this first by defining a structure type, and then a subtype property that can be used to relate similar structures. The subtype idea extends beyond structures and we will see how it can be applied to functions and pointers.

2 Record Types

We extend our typed language to included record types and operations:

$$\begin{array}{lll} e & ::= & \ldots \mid \{l_1 = e_1, l_2 = e_2, \ldots, l_n = e_n\} \mid e.l \\ \tau & ::= & \ldots \mid \{l_1.\tau_1, l_2.\tau_2, \ldots, l_n.\tau_n\} \end{array}$$

Also we add the following derivations:

$$\frac{\Gamma \vdash e_i \ : \ \tau_i^{\forall i \in 1..n}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \ : \ \{l_1.\tau_1, \ldots, l_n.\tau_n\}} \qquad \frac{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} \ : \ \{l_1.\tau_1, \ldots, l_n.\tau_n\}}{\Gamma \vdash e.l_i \ : \ \tau_i}$$

Now we want to look at relationships between similar structure types. In some sense, languages like C and Java only consider name equivalence and their subtyping relation correspondingly relies on explicit declarations: `class Child extends Parent`. We thus consider subclass declarations of these languages to be assertions of structural conformance. For a compiler to allow the programmer to make such assertions, however, it needs a way to check that the subclass and superclass classes are indeed structurally compatible. Here's how we define structural equivalence for our structure types.

To do this, we introduce a new typing judgement:

$$\Delta \vdash \tau_1 \leq \tau_2$$

which means that, given context Δ we conclude τ_1 is a subtype of τ_2. We also need to specify how subtyping is used in the type checking derivation. Basically, the idea of a subtype is that when a subtype τ' is desired, a parent type τ is acceptable. This concept is captured formally by the *subsumption rule*:

$$\frac{\Gamma \vdash e : \tau \qquad \vdash \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

Here are some basic properties of the subtype relation: reflexivity and transitivity. Note that the subtype relation is not necessarily antisymmetric.

$$\frac{}{\tau_1 \leq \tau_2} \qquad \frac{\vdash \tau_1 \leq \tau_2 \quad \vdash \tau_2 \leq \tau_3}{\vdash \tau_1 \leq \tau_3}$$

For record types there are two natural subtype relationships: width subtyping and depth subtyping. These two derivation steps, in conjunction with reflexivity and transitivity, completely describe the subtype relationship.

$$\frac{m \geq n}{\vdash \{l_1 : \tau_1, \ldots, l_m : \tau_m\} \leq \{l_1 : \tau_1, \ldots, l_n : \tau_n\}} \qquad \frac{\tau_i \leq \tau_i'^{\forall i \in 1..n}}{\vdash \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1', \ldots, l_n : \tau_n'\}}$$

Observe that the subtyping property on structures is *covariant* with the structure components. That is, a structure S is a subtype of another structure S' if the shared components of S are subtypes of the shared components of S'.

3 Functions

The rule in Eiffel with regard to method subtyping is not sound. It corresponds to the following subtyping rule on functions:

$$\frac{\vdash \tau_1 \leq \tau_1' \qquad \vdash \tau_2 \leq \tau_2'}{\vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

What we really want is for function subtyping to work as follows:

$$\frac{\vdash \tau_1' \leq \tau_1 \qquad \vdash \tau_2 \leq \tau_2'}{\vdash \tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

where the arguments are *contravariant* and the return type is *covariant*. The reason for this can be seen as follows: let's say we have a function $f : \tau \to \tau'$ and we wish to be able to use a function $g : \sigma \to \sigma'$ in its place. What should the restrictions be on types σ and σ' if we are to claim that $g \leq f$?

First, we should note that the function g should be able to take any types that f can, thus we must have g's argument types must be at least as general as f's are, ie. $\tau \leq \sigma$, so arguments are contravariant. Second, since anything that our function g returns must be acceptable in the context that f would have been used, the return value of g must be a subtype of the value that f would have returned, or $\sigma' \leq \tau'$, so return values are covariant.

Given our type hierarchy, where $\mathsf{Car} \leq \mathsf{Vehicle}$, we can construct function subtyping that is broken and will result in a run-time error, but typechecks under Eiffel's unsound inference rule:

```
fun f (v: Vehicle) = v.wheels()     (* f: Vehicle -> int *)
fun g (c: Car) = c.passengers()     (* g: Car -> int, so by Eiffel's rule,
                                       g is a subtype of f *)
let x = Vehicle(...) in
(* this code is legal *)
let y = f(x) in
(* but we can use g in place of f!
 * this code typechecks because g is a subtype of f, and f can accept a
 * Vehicle, but g will call the passengers() function on x, which it
 * does not have. This results in a runtime error. *)
g(x)
```

This shows that, albeit counterintuitive at first, the function arguments do need to be contravariant, while return values are covariant, as expected.

4 Partial order on types

Since any type is as good as unit, we can say that any type is a subtype of unit. Nontermination (which we can denote by 0) gives us less information than a terminating program of any type, and because a non-terminating program will never violate any type constraint – it will never terminate and hence can be assigned any type – we can say that 0 is a subtype of all other types:

$$\frac{}{\vdash \tau \leq 1} \qquad \frac{}{\vdash 0 \leq \tau}$$

5 Refs

We can add refs to our language with the following type annotation:

$\tau ::= \ \ldots \mid \tau \ \mathsf{ref}$

and these inference rules:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \mathsf{ref}}{\Gamma \vdash !e : \tau} \qquad \frac{\Gamma \vdash e1 : \tau\ \mathsf{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}$$

How would we present subtyping on refs? Intuitively, it should be:

$$\frac{\vdash\ \tau\ \leq\ \tau'}{\vdash\ \tau\ \mathsf{ref}\ \leq\ \tau'\mathsf{ref}}$$

But this rule is not sound. Consider the following example, where Car and Truck are subtypes of Vehicle, but not related to each other:

```
let x :Car ref = Car(...) in
let y :Vehicle ref = ref x in
(y := ref Truck(...); (!x).passengers()
```

meets with a problem: after assignment y := ref Truck(...) we now have both x and y pointing to a Truck. That may be fine for y which is a Vehicle, but not for x which is a Car, so x is acting like a Car with a Truck object backing it, which is clearly an error. Hence, a typing rule on refs cannot be covariant.

Clearly, the ref typing rule cannot be contravariant either, since that would mean that:

$$\mathsf{Car}\ \leq\ \mathsf{Vehicle}\ \Rightarrow\ \mathsf{Vehicle\ ref}\ \leq\ \mathsf{Car\ ref}$$

which will break easily:

```
let x: Vehicle ref = ref Vehicle(...) in
let y: Car ref = ref x in
!y.passengers()
```

Hence, subtyping on refs is *invariant*, which for object oriented languages such as Java means that inheriting classes cannot change the type of a mutable field variable, while they may change the types of functions as we discussed above.

It is interesting to note, however, that the original Java spec allowed to change the function return values covariantly, but this spec was never implemented in any officially released Java compiler from Sun. Rather than implement this useful feature, the spec was later changed to disallow this in Java, which would've been a cool and useful feature. This reminds us that while we are looking at the weakest sound rules for subtyping, language designers may choose to make the subtyping relation more restructive than is necessary for reasons such as efficient implementation.

6 Conclusion

The subtyping feature is fairly easy to add to the standard typed lambda calculus. Here, we have added subtyping for record types, function types, and ref types. Records have covariant subtypes, functions have covariant arguments and contravariant return types, and references are invariant.