CS611 Lecture 35 **Subtyping and Coercions** 24 November 2003

Scribe: Karen McKeen Lecturer: Radu Rugina

1 Subtyping

We now continue our discussion of types by considering subtyping. The subtyping relation typically occurs in object oriented languages, where subtyping is induced by subclassing. However, there are also other cases of subtyping relations. For instance, the primitive type int is a subtype of the primitive type float.

To formalize the subtyping relation induced by subclassing, we extend our language with records which model classes. A record is a sequence of labelled expressions. Correspondingly, a record type is a sequence of labelled types. Two types with the same component types, but with different labels are regarded differently. Records generalize pairs to tuples with labelled fields. Finally, a field access expression retrieves the contents of particular field:

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

Each $l_i = e_i$ represents a field in the record. The expression $e.l$ selects a the field l from the record e. The evaluation rules for these expressions are left as an exercise.

The typing rules are as follows:

$$\frac{\Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma \vdash \{l_1 = e_1, ..., l_n = e_n\} : \{l_1 : \tau_1, ..., l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, ..., l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

1.1 Definition of Subtyping

We next want to build a subtyping relation for records. We want this relation to be such that, if τ_1 is a subtype of τ_2, written as $\tau_1 <: \tau_2$, then a value of type τ_1 may be used whenever a value of type τ_2 is required. This is formally expressed by the following rule, known as the *subsumption rule*:

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}$$

The subsumption rule is applies only when an expression uses a value. For assignments in imperative languages, this rule does not apply to the assigned variable, on the left and side of the assignment.

The following relations define the subtyping relation:

Width Subtyping: $$\frac{n \geq m}{\{l_1 : \tau_1, ..., l_n : \tau_n\} <: \{l_1 : \tau_1, ..., l_m : \tau_m\}}$$

Depth Subtyping: $$\frac{\tau_i <: \tau_i' \quad 1 \leq i \leq n}{\{l_1 : \tau_1, ..., l_n : \tau_n\} <: \{l_1 : \tau_1', ..., l_n \tau_n'\}}$$

Permutation Subtyping: $$\frac{\{l_i : \tau_i\}_i \text{ is a permutation of } \{l_i' : \tau_i'\}_i}{\{l_1 : \tau_1, ..., l_n : \tau_n\} <: \{l_1' : \tau_1', ..., l_n' \tau_n'\}}$$

Finally, we take the reflexive, transitive closure of the above relations:

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \quad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

Note that if the absence of the permutation rule, $<:$ would be an antisymmetric relation, and therefore a partial order.

1.1.1 Function Subtyping

We next want to derive a subtyping rule for functions. Function subtyping is applicable only in languages where functions are first class values. The conclusion of the function subtyping rule should be of the form: $\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'$. To derive the appropriate premises for this rule, we will use a brief example.

Consider two functions, $f : \tau_1 \to \tau_2$ and $f' : \tau_1' \to \tau_2'$ such that the type of f is a subtype of the type of f'. Assume a program fragment $\mathsf{a} = \mathsf{f'(b)}$, where $\mathsf{a} : \tau_2'$ and $\mathsf{b} : \tau_1'$. Because the type of f is a subtype of the type of f', we should be able to replace f' with f and get a well-typed expression. It means that: Type($\mathsf{f(b)}$) $<:$ Type(a) and Type(b) $<: \tau_1$. Therefore, $\tau_2 <: \tau_2'$ and $\tau_1' <: \tau_1$. These are premise in the subtyping rule for functions:

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}$$

Informally, the above rule says that the arguments must be *contravariant* (have the opposite order) and the return values must be *covariant* (have the same order as the conclusion).

1.2 References

We next explore the interaction of subtyping and mutable heap locations. As in FUN!, we extend expressions with constructs for manipulating references:

$$e \ ::= \ \ldots \ \mid \ \mathsf{ref}\ e \ \mid \ !e \ \mid \ e_1 := e_2 \ \mid \ e_1; e_2$$

Accordingly, we add reference types:

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

We already know the evaluation rules for the constructs that manipulate references. We now want to give typing rules for these expressions. Recall that assignments are only performed for their side effects and that in a sequence, only the value of the second expression is returned. The typing rules are fairly intuitive:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau} \qquad\qquad \frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1; e_2 : \tau_2}$$

The above rules are sound, meaning that if these rules are satisfied, the program execution is free of errors.

We now want to derive a subtyping rule for references, whose conclusion is $\mathsf{ref}\ \tau <: \mathsf{ref}\ \tau'$. Intuitively, we would expect τ and τ' to be covariant:

$$\frac{\tau <: \tau'}{\mathsf{ref}\ \tau <: \mathsf{ref}\ \tau'}$$

However this turns out to be incorrect. We demonstrate that this rule is unsound with an example. Consider the following record types.

Point = {x:int, y:int}	(the supertype),
CPoint = {x:int, y:int, c:int}	(colored point), and
3DPoint = {x:int, y:int, z:int}	(three dimensional point).

With our definitions, we have the following subtyping relations: CPoint $<:$ Point and 3DPoint $<:$ Point. Now we will write a short program which is well-typed, but whose execution yields an error. We assume that we have a let construct where we specify the type of the variable we introduce: $\mathsf{let} x : \tau = e\ \mathsf{in}\ e'$. Consider the program:

```
let Z: ref CPoint = ref {x=0, y=0, c=0} in
    let X: ref ref CPoint = ref Z in
        let Y: ref ref Point = X in
            (Y := ref {x=0, y=0, z=0};
            (!(!X)).c)
```

The above program is well-typed. In particular, the assignments Y = X and Y := ref {x=0, y=0, z=0} are well-typed because of the covariant subtyping rule for references, along with the subsumption rule. However, the execution of the program runs into an error when it reaches the assignment (!(!X)).c, because !(!X) is a 3DPoint and has no field c. The problem is that we were able to change to location that X point to and make it inconsistent with the type of X.

In the presence of mutable references, reference subtyping must be *type invariant*. The correct (sound) rule in this case is:

$$\frac{\tau <: \tau' \quad \tau' <: \tau}{\mathsf{ref}\ \tau <: \mathsf{ref}\ \tau'}$$

Note, however, that if references are not mutable (heap locations can only be read), then the covariant subtyping rule for references is sound!

2 Coercions

Subtyping allows us to use values from a subtype $\tau_1 < \tau_2$ whenever values from the supertype τ_2 are required. This may cause a problem since values in τ_1 may have a different internal representation that in τ_2. Consider, for instance, the relation int <: float. Then an integer value 1 can be used whenever a floating point value is required. However, the floating point representation of 1 is different than its integer representation. Hence, the compiler, must insert subtyping *coercions* in the generated code, to adjust the internal representation of a value when going from the subtype domain to the supertype domain.

To express coercions formally, we define a function $\Theta[\![\tau <: \tau']\!] \in \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]$ which describes the change in the internal representation when going from the domain of values of the subtype $\mathcal{T}[\![\tau]\!]$ to the domain of values of the supertype $\mathcal{T}[\![\tau']\!]$. We define the coercion function Θ inductively, according to the rules that define the subtyping relation. We only present two of the rules, corresponding to the transitivity and function subtyping rules:

$$\begin{aligned}
\Theta\left[\!\!\left[\ \frac{\tau <: \tau' \quad \tau' <: \tau''}{\tau <: \tau''}\ \right]\!\!\right] &= \Theta[\![\tau' <: \tau'']\!] \circ \Theta[\![\tau <: \tau']\!] \\
\Theta\left[\!\!\left[\ \frac{\tau_1' <: \tau_1 \quad \tau_2 <: \tau_2'}{\tau_1 \to \tau_2 <: \tau_1' \to \tau_2'}\ \right]\!\!\right] &= \lambda f \in \mathcal{T}[\![\tau_1 \to \tau_2]\!].\, \lambda x \in \mathcal{T}[\![\tau_1']\!].\, \Theta[\![\tau_2 <: \tau_2']\!]\ f(\Theta[\![\tau_1' <: \tau_1]\!]\ x)
\end{aligned}$$

We leave the remaining rules as an exercise.

The compiler then applies these coercions whenever a subtyping rule is applied using the subsumption rule. If $\mathcal{TC}[\![e]\!]$ is the function that inserts the coercions for expression e, then this function recursively walks expression e in the typing proof and applies the coercion Θ whenever it encounters the subsumption rule:

$$\mathcal{TC}\left[\!\!\left[\ \frac{\Gamma \vdash e : \tau_1 \quad \tau_1 <: \tau_2}{\Gamma \vdash e : \tau_2}\ \right]\!\!\right] = \Theta[\![\tau_1 <: \tau_2]\!]\ \mathcal{TC}[\![\Gamma \vdash e.\tau_1]\!]$$