CS 6110 Lecture 28 More Subtyping, Proof Normalization, and Minimal Typing 8 April 2009

Lecturer: Andrew Myers

Our goal in exploring subtyping was partly to understand the nature of object-oriented languages. However, we haven't seen any types that look much like objects yet. We can add *record types* that look a great deal like objects, and get some useful insights.

1 Records

A record is a collection of named fields, each with its own type. We extend the grammar of e and τ for adding support for record types:

$$\begin{aligned} e &::= \dots \ \mid\ \{x_1 = e_1, \dots, x_n = e_n\} \ \mid\ e.x \\ v &::= \dots \ \mid\ \{x_1 = v_1, \dots, x_n = v_n\} \\ \tau &::= \dots \ \mid\ \{x_1 : \tau_1, \dots, x_n : \tau_n\} \end{aligned}$$

We add following the rule to the small-step semantics:

$$\frac{}{\{x_1 = v_1, \dots, x_n = v_n\}.x_i \longrightarrow v_i}$$

and the following typing rules:

$$\frac{\Gamma \vdash e_i : \tau_i \quad ^{(\forall i \in 1..n)}}{\Gamma \vdash \{x_1 = e_1, \dots, x_n = e_n\} : \{x_1 : \tau_1, \dots, x_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{x_1 : \tau_1, \dots, x_i : \tau_i, \dots, x_n : \tau_n\}}{\Gamma \vdash e.x_i : \tau_i}$$

What we can see from this is that the record type $\{x_1 : \tau_1, \dots, x_n : \tau_n\}$ acts a lot like the product type $\tau_1 * \cdots * \tau_n$. This suggests that subtyping on records should behave like subtyping on products.
There are actually three types of reasonable subtyping rules for records:

- Depth subtyping: a covariant subtyping relation between two records that have the same number of fields.

$$\frac{\tau_i \le \tau_i' \quad ^{\forall i \in 1..n}}{\{x_1 : \tau_1, \dots, x_n : \tau_n\} \le \{x_1 : \tau_1', \dots, x_n : \tau_n'\}}$$

- Width subtyping: a subtyping relation between two records that have different number of fields.

$$\frac{}{\{x_1 : \tau_1, \dots, x_{n+1} : \tau_{n+1}\} \le \{x_1 : \tau_1, \dots, x_n : \tau_n\}}$$

  Observe that in this case, the subtype has more components than the supertype. This is the kind of subtyping that most programmers are familiar with in a language like Java. It allows programmers to define a subclass with more fields and methods than in the superclass, and have the objects of the subclass be (soundly) treated as objects of the superclass.

- Permutation subtyping: a relation between records with the same fields, but in a different order. Most languages don't support this kind of subtyping because it prevents compile-time mapping of field names to fixed offsets within memory (or to fixed indices in a tuple). Notice that permutation subtyping is not antisymmetric.

$$\frac{\{a_1, \dots, a_n\} = \{1, \dots, n\}}{\{x_1 : \tau_1, \dots, x_n : \tau_n\} \le \{x_{a_1} : \tau_{a_1}, \dots, x_{a_n} : \tau_{a_n}\}}$$

The depth and width subtyping rules for records can be combined to yield a single equivalent rule that handles all transitive applications of both rules:

$$\frac{m \leq n \quad \tau_i \leq \tau_i' \ \ (\forall i \in 1 \ldots n)}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_1 : \tau_1', \ldots, x_m : \tau_m'\}}$$

Records can be viewed as tagged product types of arbitrary length; the analogous extension for sum types are variants. The depth subtyping rule for variants is the same as that given above for records (replacing the records with variants). The width subtyping rule is however different and we will see why this is so. Suppose we used a width subtyping rule of the same form as given above. Recall that if $\tau_1 \leq \tau_2$, then this implies that anything of type τ_1 can be used in a context expecting something of type τ_2. Suppose we now had a case statement that did pattern matching on something of type τ_2; our subtyping relation says that we can pass in something of type τ_1 to this case statement and still have it work. However, since τ_2 has fewer components than τ_1 and the case statement was originally written for an object of type τ_2, there will be values of τ_1 for which no corresponding pattern match exists. Thus, for variants, the direction of the $\leq$ symbol in the premise of the width subtyping rule given above needs to be reversed i.e., for variants, the subtype will have fewer components than the supertype.

2 Subtyping Rules for References

Here are the extensions to the grammar of e and τ for adding support for references:

$$\begin{array}{rcl} e & ::= & \ldots \mid \textbf{ref}\ e \mid\ !e \mid e_1 := e_2 \\ \tau & ::= & \ldots \mid \tau\ \textbf{ref} \end{array}$$

where we add the following typing rules:

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

As for the subtyping rule, once again, our first impulse would be to write down something of the following form:

$$\frac{\tau_1 \leq \tau_2}{\tau_1\ \textbf{ref} \leq \tau_2\ \textbf{ref}}$$

However, this is incorrect. To see why, consider the following example:

```
**let** $x : \mathit{Square}\ \textbf{ref} = \textbf{ref}\ \mathit{square}$ **in**
    **let** $y : \mathit{Shape}\ \textbf{ref} = x$ **in**
        $(y := \mathit{circle};\ (!x).\mathit{side})$
```

Even though this code type-checks with the given subtyping rule for reference types, it will cause a run-time error, because in the last line x does not refer to a square anymore. To avoid this problem, we do not introduce a subtyping relation between two ref types until they contain exactly the same type (the reflexive relationship). Thus, the correct subtyping rule for references is thus neither covariant nor contravariant in τ, but rather *invariant* in τ.

Arrays

What about other mutable types, such as arrays? For example, Java has an array type written $\tau[\,]$. When is it safe to use a $\tau[\,]$ as a $\tau'[\,]$? Since an array is very close to being a tuple of refs, we expect that sound array subtyping should also be invariant.

Apparently guided by a desire to support the use arrays as immutable sets, the designers of Java made array subtyping covariant. The result is that soundness is achieved in the Java type system only by run-time type checks whenever a value is assigned into a array of objects. An assignment of a value of type τ' into a array that is statically of type $\tau'[]$ might turn out to be assigning into an array of type $\tau[]$ where $\tau \le \tau'$. The necessary run-time checks to catch this make Java arrays slower than in other languages. Further, their use can result in unexpected errors at run time.

3 Proof normalization

The subsumption rule poses a problem for implementers of languages with subtyping. Because it can be used at any time during type checking, type checking is no longer syntax-directed. We can solve this problem by folding the uses of the subsumption rule into existing typing rules, restoring the syntax-directed property.

The key is to notice that typing derivations can be put into a normal form. To see how this works, we consider just $\lambda^{\to}$ with subtyping. We show that any typing derivation in this language can be reduced to a normal form in which subsumption only appears in two places:

1. a single use at the root of the derivation
2. on the right-hand (argument) premise of the application rule.

Suppose we have an arbitrary typing derivation that uses subsumption. Then we apply the following reductions to put it into this normal form.

First, we can eliminate successive uses of subsumption by taking advantage of the transitivity of subtyping:

$$\frac{\dfrac{\dfrac{A}{\Gamma \vdash e : \tau''} \quad \dfrac{B}{\tau'' \le \tau'}}{\Gamma \vdash e : \tau'}\,(\textsc{Sub}) \quad \dfrac{C}{\tau' \le \tau}}{\Gamma \vdash e : \tau}\,(\textsc{Sub}) \quad \longrightarrow \quad \frac{\dfrac{A}{\Gamma \vdash e : \tau''} \quad \dfrac{\dfrac{B}{\tau'' \le \tau'} \quad \dfrac{C}{\tau' \le \tau}}{\tau'' \le \tau}\,(\textsc{Trans})}{\Gamma \vdash e : \tau}\,(\textsc{Sub})$$

We can also eliminate the gratuitous use of reflexive subtyping:

$$\frac{\dfrac{A}{\Gamma \vdash e : \tau} \quad \dfrac{}{\tau \le \tau}}{\Gamma \vdash e : \tau} \quad \longrightarrow \quad \frac{A}{\Gamma \vdash e : \tau}$$

If subsumption is used in the premise of the typing rule for a lambda, we can push it down toward the root:

$$\frac{\dfrac{\dfrac{A}{\Gamma, x{:}\tau' \vdash e : \tau''} \quad \dfrac{B}{\tau'' \le \tau}}{\Gamma, x{:}\tau' \vdash e : \tau}\,(\textsc{Sub})}{\Gamma \vdash \lambda x{:}\tau'.\, e : \tau' \to \tau}\,(\textsc{Fun}) \quad \longrightarrow \quad \frac{\dfrac{\dfrac{A}{\Gamma, x{:}\tau' \vdash e : \tau''}}{\Gamma \vdash \lambda x{:}\tau'.\, e : \tau' \to \tau''}\,(\textsc{Fun}) \quad \dfrac{\dfrac{}{\tau' \le \tau'} \quad \dfrac{B}{\tau'' \le \tau}}{\tau' \to \tau'' \le \tau' \to \tau}}{\Gamma \vdash \lambda x{:}\tau'.\, e : \tau' \to \tau}\,(\textsc{Sub})$$

If subsumption appears in the left premise of the typing rule for an application, we can push it down toward the root *and* into the right premise:

$$\frac{\dfrac{\dfrac{A}{\Gamma \vdash e_0 : \tau_0' \to \tau_0} \quad \dfrac{\dfrac{B}{\tau' \le \tau_0'} \quad \dfrac{C}{\tau_0 \le \tau}}{\tau_0' \to \tau_0 \le \tau' \to \tau}}{\Gamma \vdash e_0 : \tau' \to \tau}\,(\textsc{Sub}) \quad \dfrac{D}{\Gamma \vdash e_1 : \tau'}}{\Gamma \vdash e_0\; e_1 : \tau}\,(\textsc{App}) \quad \longrightarrow$$

$$\frac{\dfrac{\dfrac{A}{\Gamma \vdash e_0 : \tau_0' \to \tau_0} \quad \dfrac{\dfrac{D}{\Gamma \vdash e_1 : \tau'} \quad \dfrac{B}{\tau' \le \tau_0'}}{\Gamma \vdash e_1 : \tau_0'}\,(\textsc{Sub})}{\Gamma \vdash e_0\; e_1 : \tau_0}\,(\textsc{App}) \quad \dfrac{C}{\tau_0 \le \tau}}{\Gamma \vdash e_0\; e_1 : \tau}\,(\textsc{Sub})$$

These reductions all have the effect of pushing subsumption toward the bottom of the derivation except where it is used for the right-hand premise of App. The total height of all uses of subsumption within the derivation (other than on right-hand premises in App) must decrease in each reduction. And we can see that for any derivation not in normal form, one of the above reductions will always be possible. Therefore there reductions will always terminate in a normal form.

Once we have the derivation in normal form, the single use of Sub at the bottom will mean that we derive the smallest possible type for the program (smallest in the subtyping ordering $\leq$), then apply subsumption. If we're willing to accept this *minimal typing* for the term, then we don't need the final use of subsumption at all. The minimal type for a term in this language is a *principal type*: a "best possible" type that allows it to be well-typed in any possible context that could use it. Having a principal type is an important property for a type system, because it means that the type checker doesn't have to try all possible types for a term, and hence engage in exponential search.

Besides the final use of subsumption at the root, all other uses of subsumption are attached to uses of the rule App:

$$\dfrac{\dfrac{\dots}{\Gamma \vdash e_0 : \tau' \to \tau} \quad \dfrac{\dfrac{\dots}{\Gamma \vdash e_1 : \tau_1} \quad \dfrac{\dots}{\tau_1 \leq \tau'}}{\Gamma \vdash e_1 : \tau'}\ (\textsc{Sub})}{\Gamma \vdash e_0\ e_1 : \tau}\ (\textsc{App})$$

But we can capture the effect of these parts of the derivation by a single AppSub rule that folds in the possible use of subsumption (it also works if the App didn't use subsumption):

$$\dfrac{\dfrac{\dots}{\Gamma \vdash e_0 : \tau' \to \tau} \quad \dfrac{\dots}{\Gamma \vdash e_1 : \tau_1} \quad \dfrac{\dots}{\tau_1 \leq \tau'}}{\Gamma \vdash e_0\ e_1 : \tau}\ (\textsc{AppSub})$$

This rule is clearly admissible, since anything that can be proved with it can be proved with the original rules. And since we can reduce all typing derivations to derivations in normal form, we can type-check all terms using just this rule, plus the usual rules for typechecking lambdas and variables. With just three rules, one for each syntactic form, type checking is once again syntax-directed!

In general, when we design type checkers for languages with subtyping, we need to figure out how to fold subsumption into typing rules with multiple premises. This is usually not difficult.

Though it's straightforward for the types we have seen so far, deriving the relation $\vdash \tau_1 \leq \tau_2$ can also be an issue for languages with complex subtyping rules. For example, with the three subtyping rules for records, which one should we apply first? We can also normalize the derivation of subtyping to arrive at combined rules that avoid having to make the choice.