CS 6110 Lecture 29 Type-Preserving Translation, Proof Normalization, and Principal Typing 5 April 2013

Lecturer: Andrew Myers

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 would naturally expect that sound array subtyping should also be invariant.

Apparently guided by a desire to support the use of 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.

1 Type-preserving translation

We can give a semantics to subtyping by defining a translation from the typed lambda calculus (with a subsumption rule) to the typed lambda calculus (without). The translation will be defined so that a well-typed term in the source language always translates to a well-typed term in the target language. We show that this translation does preserve typing by induction on the source-language typing derivation.

...

2 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 $\lambda^{\rightarrow}$ 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:

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

We can also eliminate the gratuitous use of reflexive subtyping:

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

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

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

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:

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

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

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:

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

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):

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

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.