15-312 Foundations of Programming Languages
Recitation 8: Subtyping

Daniel Spoonhower
`spoons+@cs`

October 22, 2003

1 Subtyping for Function Types

Recall our discussion of subtyping for function types from class along with the following rule:

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

And our coercion,

$$\frac{f : \sigma_1 \leq \tau_1 \quad g : \tau_2 \leq \sigma_2}{\lambda x.\lambda y.g\,(x\,f(y)) : \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

Consider the following, a more graphical representation.

$$\begin{tikzcd}
\tau_1 \arrow[r, "x"] & \tau_2 \arrow[d, "g"] \\
\sigma_1 \arrow[u, "f"] \arrow[r, dashed] & \sigma_2
\end{tikzcd}$$

The broken arrow across the bottom of the square is the coercion that we would like to give as witness to the conclusion of our inference rule for subtyping above. From the diagram, it's clear that this coercion is exactly the composition of f, x, and g. (Notice that the arrows representing f and g go in opposite directions, another clue that one should be covariant and the other contravariant.)

1.1 An Example with Records

Let's consider another example to make things concrete. Without (just yet) getting into the details of record subtyping, take the following relation as given.

$$\frac{}{\{x : \texttt{int}, y : \texttt{int}, z : \texttt{int}\} \leq \{x : \texttt{int}, y : \texttt{int}\}}$$

Call the type on the left `point3d` and on the right, `point2d`. Now say that we define a function `dilateBy5` : `point2d` $\to$ `point2d` that dilates a point in the plane by multiplying each coordinate of the point by the scalar 5. What sort of points might this function accept an argument? What sorts of points might we expect to see as a result?

Consider the same questions for a function `liftTo7` : `point2d` $\rightarrow$ `point3d` that takes a point in two dimensions (x, y) and returns a point in three dimensions $(x, y, 7)$ lifted out of the xy-plane.

2 References, Revisited

In our discussion in class, we noted that subtyping on references should be *both* covariant and contravariant. Remember, however, that our counterexample to covariance was through the use of an assignment, while our counterexample to contravariance was a dereference. Given the apparent orthogonality of these two examples, we might think there is a way of teasing apart the covariant portion of references from the contravariant part.

We will introduce two new type constructors $\tau\,\texttt{source}$ and $\tau\,\texttt{sink}$ to represent expressions from which (and to) we can read (and write) values. We will define assignment and derefencing as for references, but we will only assign to sinks and dereference sources.

$$\frac{\Gamma \vdash e : \tau\,\texttt{source}}{\Gamma \vdash \texttt{!}e : \tau} \quad \frac{\Gamma \vdash e_1 : \tau\,\texttt{sink} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 \texttt{:=} e_2 : \texttt{unit}}$$

We can then give the expected subtyping rules:

$$\frac{\tau \leq \sigma}{\tau\,\texttt{source} \leq \sigma\,\texttt{source}} \quad \frac{\sigma \leq \tau}{\tau\,\texttt{sink} \leq \sigma\,\texttt{sink}}$$

Finally, we can express the idea we started with, that references are both sources and sinks (since we can both read from and write to them).

$$\frac{}{\tau\,\texttt{ref} \leq \tau\,\texttt{source}} \quad \frac{}{\tau\,\texttt{ref} \leq \tau\,\texttt{sink}}$$

(There should be something reminiscent of function subtyping here. Notice that sinks, which provide an *input* to a reference cell, are contravariant, while a source, the output of a cell, is covariant.)

3 Safety

How do we know that our formulation of subtyping with sources and sinks is correct? Below we will sketch a proof of safety for a language with references and the subtyping relations described above.

3.1 Preservation

Recall the preservation theorem for our language with references.

Theorem 1 (Presevation). *If* $\Lambda; \cdot \vdash e : \tau$ *and* $\Lambda; \cdot \vdash M : \Lambda$ *and* $\langle M, e\rangle \mapsto \langle M', e'\rangle$ *then for some* $\Lambda' \geq \Lambda$ *and memory* M' *we have* $\Lambda'; \cdot \vdash e' : \tau$ *and* $\Lambda'; \cdot \vdash M' : \Lambda'$.

As usual our proof will follow by rule induction. The interesting cases will be for derefencing and assigning to locations. We will only have time to consider the former.

$$\frac{M = (M_1, l = v, M_2)}{\langle M, \texttt{!}l\rangle \mapsto \langle M, v\rangle}$$

By inversion on the typing rule, we know that if $\Lambda; \cdot \vdash \texttt{!}l : \tau$ then $\Lambda; \cdot \vdash l : \tau\,\texttt{source}$. We would like to show that $\Lambda; \cdot \vdash v : \tau$. We seem to get stuck when we try to show that $\Lambda = (\Lambda_1, l : \tau, \Lambda_2)$. The problem is that we can't know that Λ maps l to τ, it might not! Since we don't have anywhere else to go, it seems like an appropriate time to try a lemma. Before we continue, though, let's state an inversion lemma.

Lemma 2 (Inversion on Subtyping). *If* $\sigma \leq \tau\,\texttt{source}$ *then either* $\sigma = \tau\,\texttt{ref}$ *or* $\sigma = \tau'\,\texttt{source}$ *for some* $\tau' \leq \tau$.

Proof. By inspection of the subtyping rules. □

Lemma 3. *If* $\Lambda; \cdot \vdash l : \tau\,\texttt{source}$ *then* $\sigma \leq \tau$ *and* $\Lambda = (\Lambda_1, l : \sigma, \Lambda_2)$.

Proof. By rule induction on the given derivation. Note that we will use the inversion lemma above. □

With this lemma in hand, the rest of the above case is straightforward.

3.2 Progress

We won't have time to cover the changes to our progress proof, but if you take the time to think about it, note that our value inversion lemma must change. For example,

$$\textit{If } \Lambda; \Gamma \vdash v : \tau_1 \to \tau_2 \textit{ and } v \ \mathsf{value} \textit{ then } v = \texttt{fn}(\sigma_1, x.e).$$