Scribe: André Allavena and Filip Radliński Lecturer: Radu Rugina

1 Subtyping for Functions

Last time defined a subtyping relation, $\tau \leq \tau'$ whereby a value of type τ can be used whenever we require a value of type τ'. We also defined width, depth and permutation subtyping. But consider subtype for arrow types, in other words for function. Intuitively, we could write the following (incorrect) rule:

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

However, consder two examples:

$$\left.\begin{array}{ll} f : \tau \to \tau_2 & f' : \tau \to \tau_2' \\ y = f'(x) & f'(x) : \tau_2' \end{array}\right\} \quad f(x) : \tau_2 \Rightarrow f(x) : \tau_2'$$

Above, f and f' have the same domain but a different range. We can use a value of type τ_2 instead of τ_2', so our intuition as to the first premise of the rule is correct.

Now consider the following functions:

$$f : \tau_1 \to \tau \qquad f' : \tau_1' \to \tau$$

We want $f(x)$ and $f'(x)$ to both be valid, so we must require the following:

$$x : \tau_1' \leq x : \tau_1$$

In other words, our intuition was exactly wrong. The correct rule for arrow types is as follows:

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

The result types are in the same order in the premise and conclusion so we say they are "covarient". The parameter types are in the opposite order, so we say they are "antivariant". In all the other rules we have seen so far, for example for records, all the fields are covariant.

2 Top Type

Consider having a Top type such that $\forall \tau.\ \tau \leq Top$. It turns out such a type is useful, and for example equivalent to the Java `Java.object` type.

Consider the following example:

$$(\mathsf{if}\ e\ \mathsf{then}\ e_1 : \tau_1\ \mathsf{else}\ e_2 : \tau_2) : \tau$$

We want $\tau_1 \leq \tau$ and $\tau_2 \leq \tau$. If type Top exists, we are guaranteed to have such a τ. If it doesn't exists, we might not have such a τ, for example if $\tau_1 = \textbf{bool}$ and $\tau_2 = \textbf{int}$.

3 References

Let us extend the our language once more:

$$\begin{array}{lcl} e & ::= & \ldots \mid \textbf{ref}\ e \mid\ !e \mid e_0 := e_1 \\ \tau & ::= & \ldots \mid \textbf{ref}\ \tau \end{array}$$

It is very easy to derive the typing rules for these new expressions:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \textbf{ref}\ e : \textbf{ref}\ \tau} \qquad \frac{\Gamma \vdash e : \textbf{ref}\ \tau}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_0 : \textbf{ref}\ \tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 := e_1 : \mathbf{1}}$$

Note that above 1 refers to the type of the value *unit*, not the numerical value one.

How do we use subtyping with references? Consider for example:

$$\frac{\tau \leq \tau'}{\textbf{ref}\ \tau \leq \textbf{ref}\ \tau'}$$

Despite being intuitive, this rule is unsound (meaning we can write programs which type-check correctly but produce runtime errors). Let us see why with the following example.

$$\begin{aligned}
\texttt{Point} &= \{x : \textbf{int},\ y : \textbf{int}\} \\
\texttt{CPoint} &= \{x : \textbf{int},\ y : \textbf{int},\ c : \textbf{int}\} \ \leq\ \texttt{Point} \\
\texttt{3DPoint} &= \{x : \textbf{int},\ y : \textbf{int},\ z : \textbf{int}\} \ \leq\ \texttt{Point}
\end{aligned}$$

$$\begin{array}{ll}
\texttt{let}\ x' = \textbf{ref}\ \texttt{CPoint} = \textbf{ref}\ \texttt{CPoint}(\ldots)\ \texttt{in} & \\
\texttt{let}\ x = \textbf{ref}\ \textbf{ref}\ \texttt{CPoint} = \textbf{ref}\ x'\ \texttt{in} & (*\textbf{ref}\ \textbf{ref}\ \texttt{CPoint} \leq \textbf{ref}\ \textbf{ref}\ \texttt{Point}*) \\
\texttt{let}\ x' = \textbf{ref}\ \texttt{CPoint} = \textbf{ref}\ \texttt{CPoint}(\ldots)\ \texttt{in} & \\
\quad y := \textbf{ref}\ \texttt{3DPoint}(\ldots); & (*\textbf{ref}\ \texttt{3DPoint} \leq \textbf{ref}\ \texttt{Point}*) \\
\quad x.c; &
\end{array}$$

At the point $x.c$, we will have a run time error because now x references a `3DPoint` and not a `CPoint` anymore. Despite this, with the above rule the program would type-check correctly.

Conclusion, reference types are *not* covariant. So for object oriented languages, mutable fields (i.e. those that can be written to) are type invariant, while unmutable fields (such as methods) are type covariant.

4 Coercions

Consider two types τ and τ' such that $\tau \leq \tau'$.

By this we mean that we can use an expression of type τ whenever we can use one of type τ'. However, there are a number of difficulties when we actually implement this. Consider the following program:

```
int x;
float y=x;
```

If we assume, as is usually the case, $\mathsf{int} \leq \mathsf{float}$, at the second line we need to convert the representation for integers to that for floats before we can assign the value of x to y. The steps to perform this subtyping conversion, which must occur at runtime, are called *coercions*.

We can also think of coercions when applied to permutation subtyping, that is records where the fields can be in a different order. When performing assignments of records, we must search through the fields in one of the values to assign them to the other. Ideally, we would like to be able to implement statically computed offsets, eliminating the need for this search.

We represent the coercions as follows:

$$\theta[\![\tau \leq \tau']\!] \in \mathcal{T}[\![\tau]\!] \to \mathcal{T}[\![\tau']\!]$$

This coercion characterizes how we convert the representation of type τ to a representation of type τ'. Hence we have the following coercions:

$$\theta[\![\tau \leq \tau']\!] = \lambda x : \tau.\ x$$

$$\theta\left[\!\!\left[\ \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ \right]\!\!\right] = \lambda x : \tau_1.\ \theta[\![\tau_2 \leq \tau_3]\!]\Big(\theta[\![\tau_1 \leq \tau_2]\!]x\Big)$$

$$\theta\left[\!\!\left[\ \frac{\tau_1' \leq \tau_1 \quad \tau_2' \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}\ \right]\!\!\right] = \lambda f : \tau_1 \to \tau_2.\ \lambda x : \tau_1'.\ \underbrace{\theta[\![\tau_2 \leq \tau_2']\!]}_{\text{convert to } \tau_2'}\ \underbrace{f\Big(\underbrace{\theta[\![\tau_1' \leq \tau_1]\!]x}_{\text{of type } \tau_1}\Big)}_{\text{of type } \tau_2}$$

$$\begin{aligned} &\theta[\![\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1, \ldots, l_m : \tau_m\}]\!] = && \text{where } n \geq m \\ &\qquad = \lambda x : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}.\ \{l_1 = x.l_1, \ldots, l_m = x.l_m\} \end{aligned}$$

$$\begin{aligned} &\theta[\![\{l_1 : \tau_1, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1', \ldots, l_n : \tau_n'\}]\!] = && \text{where } \tau_i \leq \tau_i', \forall i = 1..n \\ &\qquad = \lambda x : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}.\ \{l_1 = \theta[\![\tau_1 \leq \tau_1']\!]x.l_1, \ldots, l_n = \theta[\![\tau_n \leq \tau_n']\!]x.l_n\} \end{aligned}$$

$$\theta[\![\ \underbrace{\{l_i : \tau_i\}_i}_{\text{permutation}}\ \leq \{l_j : \tau_j\}_j]\!] = \lambda x : \{l_i : \tau_i\}_i.\ \{l_{p(i)} : \tau_{p(i)}\}_i$$

We can use these coercions to build translations. We represent the translations as:

$$\mathcal{T}[\![\Gamma \vdash e : \tau]\!]$$

First note that there is a potential for confusion here we are using $\mathcal{T}$ to have a completely different meaning to that used before. The above statement represents the function to translate the well typed expression e which is of type τ.

An example of a translation function is:

$$\mathcal{T}[\![\Gamma \vdash e_0 e_1 : \tau]\!] = \mathcal{T}[\![\Gamma \vdash e_0 : \tau' \to \tau]\!]\mathcal{T}[\![\Gamma \vdash e_1 : \tau']\!]$$

This gives a recursive definition for the translation function for applications, but does not make use of the coercions.

If we consider the subtyping rule:

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

The translation function for this rules is:

$$\mathcal{T}[\![\Gamma \vdash e : \tau']\!] = \theta[\![\tau \to \tau']\!]\Big(\mathcal{T}[\![\Gamma \vdash e : \tau]\!]\Big)$$

It is in such translation functions that we make the conversion between types explicit in the generated code.

Next lecture will be on polymorphism.