Notes on subsumption and subtyping

Introduction

These notes review our discussion of subsumption and subtyping in type-and-effect systems.

Subsumption

Our discussion of type and effect systems included the *subsumption* rule.

$$\frac{\Gamma \vdash t : T \mathbin{\&} E_1 \quad E_1 \subseteq E_2}{\Gamma \vdash t : T \mathbin{\&} E_2}$$

This rule should seem counter-intuitive: the conclusion has more effects---that is, makes a weaker statement about the term e---that the premises. And, in the absense of abstraction, we could mostly work around the need for subsumption. But once we introduction abstractions (i.e., functions), subsumption becomes unavoidable. Consider the following example:

```
if isz x then throw[Int] else get - x
```

The two branches of the `if` have different effects, so these effects must be merged in the type of the expression. We have written the typing rule for `if` to take this possibility into account, and so we can conclude that this example has type $Int \overset{\{\text{get,throw}\}}{\rightarrow} Int$. However, if we add an extra layer of abstraction, we have a problem:

```
if isz x then \y : Int -> throw[Int] else \y : Int -> get - x
```

Now, the naive derivations of the two branches are

$$x : \texttt{Int} \vdash \backslash y : \texttt{Int} \rightarrow \texttt{throw[Int]} : \texttt{Int} \overset{\{\text{throw}\}}{\rightarrow} \texttt{Int} \mathbin{\&} \emptyset$$

and

$$x : \texttt{Int} \vdash \backslash y : \texttt{Int} \rightarrow \texttt{get} : \texttt{Int} \overset{\{\text{get}\}}{\rightarrow} \texttt{Int} \mathbin{\&} \emptyset$$

The problem is no longer that the effects do not match, but that the types themselves do not match. To construct a typing for this term, we need to use the subsumption rule in the derivation of each branch, to conclude that they can both have the type $\mathtt{Int} \overset{\{\text{throw},\text{get}\}}{\rightarrow} \mathtt{Int}$.

If that were as far as it went, this would not be too bad. But consider another example. Suppose that g_1 has type $(\mathtt{Int} \overset{\{\text{throw}\}}{\rightarrow} \mathtt{Int}) \overset{E}{\rightarrow} \mathtt{Int}$ and g_2 has type $(\mathtt{Int} \overset{\{\text{get}\}}{\rightarrow} \mathtt{Int}) \overset{E}{\rightarrow} \mathtt{Int}$, and consider the term

```
let f = \x : Int -> x in if isz y then g₁ f else g₂ f
```

Now, we have a problem. If we inline the body of `f` in the two branch of the `if` statement, then the resulting term can type. In the first branch, we use subsumption to conclude that `\x : Int -> x` has type $\mathtt{Int} \overset{\{\text{throw}\}}{\rightarrow} \mathtt{Int}$, and in the second branch that it has type $\mathtt{Int} \overset{\{\text{get}\}}{\rightarrow} \mathtt{Int}$. But we cannot give `f` a single type that encompasses both of these uses, and so it seems that we cannot type this term.

Subtyping

The key relationship that we need to capture is one among types, corresponding to (hypothetical) uses of the subsumption rule. This is an example of a more general pattern called *subtyping*, which appears throughout programming languages, particularly in object-oriented programming languages.

To capture this relationship, we start by introducing a new relation on types, $T_1 <: @_2$. The intuition behind this relation is that any time you expect a value of type T_2, you can substitute a value of type T_1. An alternative view is that the values of type T_1 are a subset of the values of type T_2. For example, suppose we had a type `Pos` that contained only the positive integers. Then, we would have that $\mathtt{Pos} <: \mathtt{Int}$.

We can formally define the $<:$ relation using natural deduction rules, as follows.

$$\frac{}{\mathtt{Int} <: \mathtt{Int}} \qquad \frac{}{\mathtt{Bool} <: \mathtt{Bool}}$$

The base types are subtypes only of themselves.

$$\frac{T_1 <: T_2 \quad U_1 <: U_2}{T_1 \times U_1 <: T_2 \times U_2} \qquad \frac{T_1 <: T_2 \quad U_1 <: U_2}{T_1 + U_1 <: T_2 + U_2}$$

Product and sum types are subtypes if their components are subtypes. Consider the product types: if $T_1 \times U_1 <: T_2 \times U_2$, then we expect to be able to use a $T_1 \times U_1$ value anywhere we expect a $T_2 \times U_2$ value. What does this mean? The primary thing we can do with a term of product type is to access its components, so it must be the case that the components of the $T_1 \times U_1$ value can be used where the components of the $T_2 \times U_2$ value are expected. The intuition for sums is similar.

$$\frac{T_2 <: T_1 \quad U_1 <: U_2 \quad E_1 \subseteq E_2}{T_1 \xrightarrow{E_1} U_1 <: T_2 \xrightarrow{E_2} U_2}$$

Functions are the interesting case. Suppose that $T_1 \xrightarrow{E_1} U_1 <: T_2 \xrightarrow{E_2} U_2$; this means that we should be able to use a function of type $T_1 \xrightarrow{E_1} U_1$ any time we expect a function of type $T_2 \xrightarrow{E_2} U_2$. What does this require? It requires that the result U_1 be usable wherever we expected a result U_2, so $U_1 <: U_2$; it also requires that the effects E_1 be no more than the expected effects E_2, so $E_1 \subseteq E_2$. Finally, it requires that any argument valid for $T_2 \xrightarrow{E_2} U_2$ also be valid for $T_1 \xrightarrow{E_1} U_1$; that is, that $T_2 <: T_1$. Note that this is reversed from the other relations.

Typing with subtyping

Having defined the subtyping relation, we need to introduce it to the type system. As with subsumption, the most straightforward way to do so is by introducing a new typing rule.

$$\frac{\Gamma \vdash t : T_1 \quad T_1 <: T_2}{\Gamma \vdash t : T_2}$$

This captures the intuition of subtyping directly: any time you need a term of type T_2, a term of type T_1 will do.

However, from an implementation perspective, this rule is problematic. Up to this point, our type systems have been *syntax-directed*. That is, the typing rule to be applied at each point is determined by the syntax of the term. This is not true for the subtyping (or subsumption) rules: they can be applied at any point in a derivation. However, we do not want to over-eagerly apply subtyping or subsumption, as doing so loses information about the term.

To resolve this problem, we observe that, while we *could* apply subtyping at arbitrary points, it is only necessary to do so when we are comparing types for equality. This leads us to an alternative approach, in which we do not include the subtyping rule in its full generality, but instead build subtyping into the other rules where it could usefully be applied.

$$\frac{\Gamma \vdash t_1 : T_1 \xrightarrow{E_3} U \mathbin{\&} E_1 \quad \Gamma \vdash t_2 : T_2 \mathbin{\&} E_2 \quad T_2 <: T_1}{\Gamma \vdash t_1\ t_2 : U \mathbin{\&} E_1 \cup E_2 \cup E_3}$$

$$\frac{\Gamma \vdash t_1 : \texttt{Bool} \mathbin{\&} E_1 \quad \Gamma \vdash t_2 : T_2 \mathbin{\&} E_2 \quad \Gamma \vdash t_3 : T_3 \mathbin{\&} E_3 \quad T_2 <: T \quad T_3 <: T}{\Gamma \vdash \texttt{if}\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 : T \mathbin{\&} E_1 \cup E_2 \cup E_3}$$