lec15: Subtyping

Joshua Dunfield

March 20, 2018

Covers Thursday, 15 March and Tuesday, 20 March

1 Subtyping

1.1 Introduction

Many real programming languages include some form of *subtyping*. You may be most familiar with subtyping in object-oriented languages, where the primary form of subtyping is achieved through inheritance: if class C2 inherits from class C1, then C2 is a subclass of C1, and therefore C2 is a*subtype* of C1.

However, we can interpret subtyping more broadly:

$$S \text{ is a subtype of } T \text{ if every } S \text{ is a } T$$

or

$$S \text{ is a subtype of } T \text{ if } S \subseteq T$$

We cannot really say that, because types S and T are defined by a grammar; what does it mean for one string of symbols to be a subset of another?

We *can* say

S is a subtype of T if,
for every value v such that $\emptyset \vdash v : S$,
we can also derive $\emptyset \vdash v : T$

Most subtyping *systems*—sets of rules deriving a judgment $S <: T$—do not quite reflect this idea. Instead, they *approximate* it, by being sound with respect to it:

If $S <: T$ then, for every value v such that...

but not complete, that is, the following does *not* hold:

If, for every value v such that..., we can derive $S <: T$

An example of a "sound subtyping" that many subtyping systems cannot derive is

$$(\bot \times \mathsf{int}) <: \bot$$

It is true that every value of type $(\bot \times \mathsf{int})$ also has type $\bot$, but only because there are *no* values of type $(\bot \times \mathsf{int})$—because there are no values of type $\bot$.

On the other hand, subtyping systems *can* derive many useful subtypings. For example, if we add a type nat of integers that are greater than or equal to zero, with a typing rule

$$\frac{n \geq 0}{\Gamma \vdash n : \mathsf{nat}}\ \mathsf{natIntro}$$

then every value of type nat also has type int (because we can use our existing rule intIntro), making the following subtyping rule sound.

$$\frac{}{\mathsf{nat} \mathrel{<:} \mathsf{int}}\ \text{sub-nat-int}$$

In the remainder of these notes, we design sound subtyping rules for other types in our language, including $\times$, $\rightarrow$ and $+$.

1.2 Reflexivity

A rule that doesn't say anything interesting is the *reflexivity rule*:

$$\frac{}{S \mathrel{<:} S}\ \text{sub-refl}$$

It says that every type is a subtype of itself. Intuitively, this says that, if a value has type S then it has type S, which is certainly sound.

1.3 Subtyping for pairs

$$\frac{S_1 \mathrel{<:} T_1 \qquad S_2 \mathrel{<:} T_2}{(S_1 \times S_2) \mathrel{<:} (T_1 \times T_2)}\ \text{sub-pair}$$

You can gain some intuition for this rule by drawing the Cartesian plane, interpreting (Pair x y) as the point (x, y) where x and y are integers, and considering the types

- nat $\times$ nat,
- nat $\times$ int,
- int $\times$ nat, and
- int $\times$ int.

Then the rule sub-pair says that the upper-right quadrant (nat $\times$ nat) is a subtype of the three other types, that the right-hand half (nat $\times$ int) is a subtype of the entire plane (int $\times$ int), and that the upper half (int $\times$ nat) is a subtype of the entire plane (int $\times$ int).

Exercise 1. Add a type neg, like pos but negative. Design an appropriate subtyping rule. Design appropriate subtyping rule(s).

Exercise 2. Add a type zero, whose only value is 0. Design an appropriate typing rule. Design appropriate subtyping rule(s).

1.4 Substitutability

The visual intuition of the Cartesian plane may be enough to figure out subtyping for $\times$, but subtyping for some other types will be tricky. We need another source of guidance.

A useful way to approach subtyping is *substitutability*, which asks: If I expect something of type T, when should I allow something of type S instead? If I expect T but allow S, then values of type S are *substitutable* for values of type T, and it is okay for S to be a subtype of T. (See the Liskov–Wing principle. Aside: I was a TA for Jeannette Wing in 2001.)

For example, if I expect something of type nat $\times$ int, I should allow you to give me something of type nat $\times$ nat: I expect something from the right-hand half of the Cartesian plane, and you are giving me something from the upper-right quadrant, which is contained within the right-hand half.

$$\dfrac{\dfrac{}{\mathsf{nat} <: \mathsf{nat}}\ \text{sub-refl} \qquad \dfrac{}{\mathsf{nat} <: \mathsf{int}}\ \text{sub-nat-int}}{(\mathsf{nat} \times \mathsf{nat}) <: (\mathsf{nat} \times \mathsf{int})}\ \text{sub-pair}$$

1.5 Subtyping for functions

It's tempting to write a rule

$$\dfrac{S_1 <: T_1 \qquad S_2 <: T_2}{(S_1 \rightarrow S_2) <: (T_1 \rightarrow T_2)}\ \text{sub-}{\rightarrow}\text{-UNSOUND}$$

Unfortunately, only one of these two premises is okay.

The okay premise is the second one. For example, we need the second premise to show

$$(\mathsf{unit} \rightarrow \mathsf{nat}) <: (\mathsf{unit} \rightarrow \mathsf{int})$$

Under substitutability, if I expect something of type unit $\rightarrow$ int—that is, a function that takes () and returns an integer—I should accept your offer of a function that takes () and returns a natural number, because nat <: int (every natural number is an integer).

However, as John C. Reynolds[1] once said, "something funny happens to the left of the arrow". The premise $S_1 <: T_1$ allows us to derive

$$\dfrac{\mathsf{nat} <: \mathsf{int} \qquad \mathsf{nat} <: \mathsf{nat}}{(\mathsf{nat} \rightarrow \mathsf{nat}) <: (\mathsf{int} \rightarrow \mathsf{nat})}\ \text{sub-}{\rightarrow}\text{-UNSOUND}$$

That is, if I expect a function of type int $\rightarrow$ nat, I should accept a function of type nat $\rightarrow$ nat.

An example of a function of type int $\rightarrow$ nat is

$$\text{absf} \quad = \quad \texttt{(Lam x (Abs x))}$$

If I call absf, I will always get a natural number, even when I pass a negative number. This function also has type nat $\rightarrow$ nat.

[1] His last student, Neel Krishnaswami, wrote about him shortly after his death. When I met John for the first time, I was impressed that he seemed genuinely interested in what I thought about Java, even though I was an undergraduate student and he was one of the greatest researchers in the field.

However, another example of a function of type $\mathsf{nat} \rightarrow \mathsf{nat}$ is the identity function:

$$\text{idf} \quad = \quad (\texttt{Lam}\ x\ x)$$

If I pass a negative number like −5 to idf, I will get −5.

Therefore, if I expect a function like absf of type $\mathsf{int} \rightarrow \mathsf{nat}$, and you give me idf of type $\mathsf{nat} \rightarrow \mathsf{nat}$, I will be unhappy.

To fix the subtyping rule and make it sound, we could require the argument types, S_1 and S_2, to be the same:

$$\frac{S_1 = T_1 \qquad S_2 <: T_2}{(S_1 \rightarrow S_2) <: (T_1 \rightarrow T_2)}\ \text{sub-}{\rightarrow}\text{-sound-but-weak}$$

This rule properly disallows $(\mathsf{nat} \rightarrow \mathsf{nat}) <: (\mathsf{int} \rightarrow \mathsf{nat})$. But it is not as strong as it could be. It turns out that S_1 and T_1 don't have to be the same; rather, T_1—the type from the right-hand side of the conclusion—must be a subtype of S_1—which is from the left-hand side of the conclusion. This "swapping" is called *contravariance*.

$$\frac{T_1 <: S_1 \qquad S_2 <: T_2}{(S_1 \rightarrow S_2) <: (T_1 \rightarrow T_2)}\ \text{sub-}{\rightarrow}$$

Let's say that I expect a function of type $\mathsf{nat} \rightarrow \mathsf{nat}$. Maybe I expect something like idf. If you give me a function of type $\mathsf{int} \rightarrow \mathsf{nat}$, you are giving me a more powerful tool—a function that can take *any* integer, not only a positive integer. I will only pass natural numbers to the function, because I think it has type $\mathsf{nat} \rightarrow \mathsf{nat}$; I won't use the extra power, but it does no harm. Our correct rule sub-→ says that's okay:

$$\frac{\dfrac{}{\mathsf{nat} <: \mathsf{int}}\ \text{sub-nat-int} \qquad \dfrac{}{\mathsf{nat} <: \mathsf{nat}}\ \text{sub-refl}}{(\mathsf{int} \rightarrow \mathsf{nat}) <: (\mathsf{nat} \rightarrow \mathsf{nat})}\ \text{sub-}{\rightarrow}$$

■ **Exercise 3.** Complete the following derivation. (Yes, this is possible! Rule sub-→ swaps the argument types, and you need to use sub-→ *twice*, so the types get swapped twice.)

$$\frac{}{(\mathsf{nat} \rightarrow \mathsf{int}) \rightarrow \mathsf{unit} <: (\mathsf{int} \rightarrow \mathsf{int}) \rightarrow \mathsf{unit}}$$

1.6 Subtyping for sums

A value of type $T_1 + T_2$ is either

1. $(\texttt{Inj}_1\ v_1)$ where v_1 has type T_1, or
2. $(\texttt{Inj}_2\ v_2)$ where v_2 has type T_2.

If I expect a value of type $T_1 + T_2$, and you give me a value of type $S_1 + S_2$, I should accept it as long as every value of type S_1 is also a value of type T_1, and the same for S_2 and T_2. When

I eliminate $T_1 + T_2$ using a Case, I expect the variable x_1 to have type T_1 in one branch, and the variable x_2 will have type T_2 in the other branch. If you give me an x_1 of type S_1, that's okay as long as $S_1 <: T_1$.

$$\frac{S_1 <: T_1 \qquad S_2 <: T_2}{(S_1 + S_2) <: (T_1 + T_2)}\ \text{sub-+}$$

For example, if I expect a value v to have type $\mathsf{int} + \mathsf{unit}$, then I expect *either*

1. $v = (\mathtt{Inj}_1\ n)$ where n is an integer, or
2. $v = (\mathtt{Inj}_2\ ())$.

If you give me a v of type $\mathsf{nat} + \mathsf{unit}$, then you are guaranteeing that either

1. $v = (\mathtt{Inj}_1\ n)$ where n is an integer *and* $n \geq 0$, or
2. $v = (\mathtt{Inj}_2\ ())$.

The first part of your guarantee is stronger than what I need, because I only need to know that n is an integer, but that's okay.

$$\frac{\dfrac{}{\mathsf{nat} <: \mathsf{int}}\ \text{sub-nat-int} \qquad \dfrac{}{\mathsf{unit} <: \mathsf{unit}}\ \text{sub-refl}}{(\mathsf{nat} + \mathsf{unit}) <: (\mathsf{nat} + \mathsf{unit})}\ \text{sub-+}$$

1.7 Subsumption rule

Defining subtyping rules is only of theoretical interest unless we incorporate subtyping into our type system. We can add a rule known as *subsumption*.

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T}\ \text{type-subsume}$$

Adding this rule has some interesting consequences: if we know that, say, an expression e has the form $(\mathtt{Call}\ e_1\ e_2)$, we no longer know that the rule concluding a derivation $\Gamma \vdash e : T$ has to be $\rightarrow$Elim, because type-subsume could have been used instead.