lec13: Subtyping

Jana Dunfield

March 25, 2022

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:

$$\text{If } S <: T \text{ then, for every value } v \text{ such that...}$$

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

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

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.

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

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

1.2 Reflexivity

The *reflexivity rule* says that every type is a subtype of itself.

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

Intuitively, this says that, if a value has type S then it has type S, which is certainly sound.

1.3 Subtyping for pairs

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

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 (Liskov and Wing 1994).

For example, if I expect something of type $\mathsf{nat} \times \mathsf{int}$, I should allow you to give me something of type $\mathsf{nat} \times \mathsf{nat}$: 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

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

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

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

Under substitutability, if I expect something of type $\mathsf{unit} \to \mathsf{int}$—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 $\mathsf{nat} <: \mathsf{int}$ (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

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

That is, if I expect a function of type $\mathsf{int} \to \mathsf{nat}$, I should accept a function of type $\mathsf{nat} \to \mathsf{nat}$.

An example of a function that has (or that we would like to have) type $\mathsf{int} \to \mathsf{nat}$ is

```
absf = (Lam x (Ite (< x 0) (- 0 x) x))
```

If I call absf, I will always get a natural number, even when I pass a negative number. This function also has type $\mathsf{nat} \to \mathsf{nat}$. (Making this expression *actually* have that type is a little complicated.

[1] John Reynolds's last student, Neel Krishnaswami, wrote about him shortly after his death: `http://semantic-domain.blogspot.ca/2013/04/john-c-reynolds-june-1-1935-april-28.html`. 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.

It's easier if we add an absolute value expression to the language, and then define absf to be `(Lam x (Abs x))`.)

However, another example of a function of type nat → 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 int → nat, and you give me idf of type nat → nat, I may 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:

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

This rule properly disallows (nat → nat) <: (int → 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*.

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

Let's say that I expect a function of type nat → nat. Maybe I expect something like idf. If you give me a function of type int → 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 nat → nat; I won't use the extra power, but it does no harm. Our correct rule sub-→ says that's okay:

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

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.)

$$\frac{}{(\text{nat} \to \text{int}) \to \text{unit} <: (\text{int} \to \text{int}) \to \text{unit}}$$

1.6 Subtyping for sums

A value of type $T_1 + T_2$ is either

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

1.8 Backwards subsumption?

Reversing the premises of type-subsume is wrong; it will cause type preservation to break (that is, the type preservation theorem becomes false):

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

However, a "downcast" feature appears in a number of programming languages, including Java:

$$\frac{\Gamma \vdash e : T \qquad S <: T}{\Gamma \vdash (\mathsf{Downcast}\ e\ S) : S} \text{ type-downcast}$$

Assuming suitable reduction rules, this "downcast" construct can be compatible with type preservation. If the downcast succeeds (the value really does have type S and not only type T), we get a value of type S:

$$\frac{\emptyset \vdash v : S}{(\mathsf{Downcast}\ v\ S) \mapsto_R v}\ \text{red-downcast}$$

If the downcast fails, then reduction is not possible. This is compatible with type preservation, because type preservation only applies when the expression can be stepped.

But progress, as currently stated, breaks: for example, $(\mathsf{Downcast}\ -\ -\ 3\ \mathsf{nat})$ has type nat, but is not a value and it does not step to anything. To "patch" this, we would need to amend the statement of progress to include a third possibility. Progress would then say that either (1) the expression is a value, (2) it steps, or (3) it "fails" due to a failed cast.

Many languages already need something like this new version of progress; adding a division operator would require the semantics to account for failing due to division by zero, adding exceptions would require accounting for failing due to an uncaught exception, and so on.

Generally speaking, type systems prevent certain categories of errors but not all of them; if we give programmers the power to attempt a downcast that might fail, we have to accept the possibility of failure.

2 Coercion interpretation

The previous sections have assumed the *value interpretation* of subtyping, stated above as

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

That is, S <: T if the values of type S are a subset of the values of type T. In practice, the rules we have for subtyping may not be "complete": we might have a case where the values of type S are a subset of T but we cannot derive S <: T. But subtyping should be "sound"—that is, the converse (the converse of $A \supset B$ is $B \supset A$) should hold:

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

This characterization works for some languages, but does not describe how subtyping works in many other languages. Consider a subtyping rule

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

Every integer is a real number, so mathematically this rule seems as valid as sub-nat-int (because every natural number is an integer). However, computers do not represent integers the same way as floating-point numbers: the integer 2 would be represented in binary as 000...10, while the floating-point number 2 would be represented differently (since floating-point numbers have two parts: a mantissa and an exponent).

On my laptop, the C program below prints `0.000000`. (Changing `n = 2` to `n = 999999999` causes `n as float: 0.004724` to be printed.

```
#include <stdio.h>

int main (int argc, char **argv)
{
  int n = 2;
  int *p = &n;
  float *q = (float *)p;
  printf("sizeof n: %ld, sizeof *q: %ld\n", sizeof n, sizeof *q);
  printf("n = %d; n as float: %f\n", n, *q);
}
```

To account for the behaviour of languages that automatically *cast*, *coerce* or *convert* from a subtype to a supertype *with a different representation*—like converting from int to float—we can change the statement above as follows:

> If S is a subtype of T then,
> for every value v such that $\emptyset \vdash v : S$,
> there exists a coercion f such that we can also derive $\emptyset \vdash f(v) : T$.

This new statement characterizes the *coercion interpretation* of subtyping: if S <: T, then there must be a way to convert from ("coerce") values of type S to values of type T. For the subtyping between int and float, the coercion f would be a machine-level instruction to convert an integer to a floating-point number.

What properties should f have?

For starters, it should be a mathematical function, that is, if $f(v) = v_1$ and $f(v) = v_2$, then $v_1 = v_2$. We shouldn't get two different floating-point numbers from the same integer.

Beyond that, the properties of f are determined by our language design. We might hope that different integers are converted to different floating-point numbers, but this may not be the case; for example, on my laptop's C compiler, `int` and `float` are both 32 bits. To represent every `int` uniquely in a floating-point number, we would need 32 bits of mantissa, which would leave no room for the exponent. The subtyping int <: double would probably avoid this, because I'd expect a 64-bit floating-point number to use at least 32 bits for the mantissa. So requiring that f be a one-to-one function would exclude subtypings such as int <: float, while allowing (probably) int <: double.

Another requirement, which real languages often don't meet, is that the coercion itself be unique. Consider the rules

$$\frac{}{\text{int} <: \text{double}}\ \text{sub-int-double} \qquad \frac{}{\text{int} <: \text{string}}\ \text{sub-int-string} \qquad \frac{}{\text{double} <: \text{string}}\ \text{sub-double-string}$$

A reasonable way to convert the integer `2` to a string would be the string `"2"`. A reasonable way to convert the double `2.0` to a string would be the string `"2.0"`. There are two derivations of int <: string:

$$\dfrac{\dfrac{}{\text{int} <: \text{string}}\ \text{sub-int-string}}{\text{int} <: \text{string}} \qquad \dfrac{\dfrac{\dfrac{}{\text{int} <: \text{double}}\ \text{sub-int-double} \quad \dfrac{}{\text{double} <: \text{string}}\ \text{sub-double-string}}{\text{int} <: \text{string}}\ \text{sub-trans}}{\text{int} <: \text{string}}$$

Here, I used a rule which I somehow forgot to present, a transitivity rule:

$$\frac{S_1 <: S_2 \qquad S_2 <: S_3}{S_1 <: S_3}\ \text{sub-trans}$$

If we associate an appropriate coercion to each subtyping rule, compiling using the sub-int-string derivation would cause 2 to be converted to the string `"2"`, while the sub-trans derivation would cause 2 to be converted to the string `"2.0"`. This violates a property called *coherence*:

> If derivation $\mathcal{D}_1$ of S <: T generates the coercion f_1
> and derivation $\mathcal{D}_2$ of S <: T generates the coercion f_2
> then $f_1 = f_2$.

References

Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994. `http://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf`.