Subtyping

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 is a subtype of T if every S is a T
- or
- S is a subtype of T if S ⊆ 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 ∅ ⊢ v : S,
we can also derive ∅ ⊢ 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

(⊥ × int) <: ⊥

It is true that every value of type (⊥ × int) also has type ⊥, but only because there are no values of type (⊥ × int)—because there are no values of type ⊥.

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

\[
\frac{n \geq 0}{\Gamma \vdash n : \text{nat}} \quad \text{natIntro}
\]
then every value of type $\text{nat}$ also has type $\text{int}$ (because we can use our existing rule $\text{int} \text{Intro}$), making the following subtyping rule sound.

\[
\text{nat} <: \text{int} \quad \text{sub-nat-int}
\]

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

### 1.2 Reflexivity

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

\[
S <: S \quad \text{sub-refl}
\]

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

\[
\frac{S_1 <: T_1 \quad S_2 <: T_2}{(S_1 \times S_2) <: (T_1 \times T_2)} \quad \text{sub-pair}
\]

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

- $\text{nat} \times \text{nat}$,
- $\text{nat} \times \text{int}$,
- $\text{int} \times \text{nat}$, and
- $\text{int} \times \text{int}$.

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

- **Exercise 1.** Add a type $\text{neg}$, like $\text{pos}$ but negative. Design an appropriate subtyping rule. Design appropriate subtyping rule(s).

- **Exercise 2.** Add a type $\text{zero}$, whose only value is 0. Design an appropriate typing rule. Design appropriate subtyping rule(s).
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.\(^1\) Aside: I was a TA for Jeannette Wing in 2001.)

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

\[
\frac{\text{nat} < : \text{nat}}{\text{sub-refl}} \quad \frac{\text{nat} < : \text{int}}{\text{sub-nat-int}} \quad \frac{(\text{nat} \times \text{nat}) < : (\text{nat} \times \text{int})}{\text{sub-pair}}
\]

### 1.5 Subtyping for functions

It’s tempting to write a rule

\[
\frac{S_1 < : T_1 \quad S_2 < : T_2}{(S_1 \to S_2) < : (T_1 \to T_2)} \text{ sub-\rightarrow-UNSOUND}
\]

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

\[
(\text{unit} \to \text{nat}) < : (\text{unit} \to \text{int})
\]

Under substitutability, if I expect something of type $\text{unit} \to \text{int}$—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 $\text{nat} < : \text{int}$ (every natural number is an integer).

However, as John C. Reynolds\(^2\) once said, “something funny happens to the left of the arrow”. The premise $S_1 < : T_1$ allows us to derive

\[
\frac{\text{nat} < : \text{int} \quad \text{nat} < : \text{nat}}{(\text{nat} \to \text{nat}) < : (\text{int} \to \text{nat})} \text{ sub-\rightarrow-UNSOUND}
\]

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

An example of a function of type $\text{int} \to \text{nat}$ is

\[
\text{absf} = (\text{Lam} x (\text{Abs} x))
\]

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

---

\(^1\)Barbara Liskov and Jeannette Wing. A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems, 16(6), 1994. [http://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf](http://www.cs.cmu.edu/~wing/publications/LiskovWing94.pdf)

\(^2\)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.
§1 Subtyping

However, another example of a function of type \( \text{nat} \rightarrow \text{nat} \) is the identity function:

\[
idf = (\text{Lam } x \ x)
\]

If I pass a negative number like \(-5\) to idf, I will get \(-5\).

Therefore, if I expect a function like absf of type \( \text{int} \rightarrow \text{nat} \), and you give me idf of type \( \text{nat} \rightarrow \text{nat} \), 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:

\[
\begin{align*}
S_1 &= T_1 \\
S_2 &<: T_2 \\
(S_1 \rightarrow S_2) &<: (T_1 \rightarrow T_2)
\end{align*}
\]

This rule properly disallows \( (\text{nat} \rightarrow \text{nat}) <: (\text{int} \rightarrow \text{nat}) \). 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.

\[
\begin{align*}
T_1 &<: S_1 \\
S_2 &<: T_2 \\
(S_1 \rightarrow S_2) &<: (T_1 \rightarrow T_2)
\end{align*}
\]

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

\[
\begin{align*}
\text{nat} &<: \text{int} \\
\text{nat} &<: \text{nat} \\
(\text{int} \rightarrow \text{nat}) &<: (\text{nat} \rightarrow \text{nat})
\end{align*}
\]

Exercise 3. Complete the following derivation. (Yes, this is possible! Rule \( \text{sub-} \rightarrow \) swaps the argument types, and you need to use \( \text{sub-} \rightarrow \) twice, so the types get swapped twice.)

\[
(\text{nat} \rightarrow \text{int}) \rightarrow \text{unit} <: (\text{int} \rightarrow \text{int}) \rightarrow \text{unit}
\]

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 $S_1 <: T_1$.

$$S_1 <: T_1 \quad S_2 <: T_2 \quad \text{sub}+ \quad (S_1 + S_2) <: (T_1 + T_2)$$

For example, if I expect a value $v$ to have type `int + unit`, then I expect either

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

If you give me a $v$ of type `nat + unit`, then you are guaranteeing that either

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

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.

$$\text{nat} <: \text{int} \quad \text{sub-nat-int} \quad \text{unit} <: \text{unit} \quad \text{sub-refl} \quad (\text{nat} + \text{unit}) <: (\text{nat} + \text{unit}) \quad \text{sub}+$$

### 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`.

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

Adding this rule has some interesting consequences: if we know that, say, an expression $e$ has the form `(Call e1 e2)`, we no longer know that the rule concluding a derivation $\Gamma \vdash e : T$ has to be $\rightarrow\text{Elim}$, 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):

$$\Gamma \vdash e : T \quad S <: T \quad \text{type-supersume}$$

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

$$\Gamma \vdash e : T \quad S <: T \quad \text{type-downcast}$$
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 $S$:

$$
\emptyset \vdash v : S \\
(\text{Downcast } v S) \rightsquigarrow_R v \quad \text{red-downcast}
$$

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, $(\text{Downcast } -- 3 \text{ nat})$ 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.

References