Many modern languages include a feature called *subtyping*. Subtyping is particularly well known in object oriented languages, and indeed was first introduced in the first object oriented language: Simula. The inventors of Simula, Ole-Johan Dahl and Kristen Nygaard, went on to win the Turing award for their contribution to the field of object-oriented programming. Simula introduced a number of innovative features that have become the mainstay of modern OO languages including objects, subtyping and inheritance.

The concept of subtyping is that one type may be more precise than another. A type τ_1 is a subtype of τ_2 if τ_1 is more precise than τ_2. Any instance of τ_1 must then also be an instance of τ_2, but not necessarily vice versa. If we think of a type as encapsulating the set of terms that have that type, a subtyping relationship says that the set of terms described by τ_1 is a subset of those described by τ_2.

The concept of subtyping is closely tied to inheritance and polymorphism and offers a formal way of studying them. It is well illustrated by means of an example.

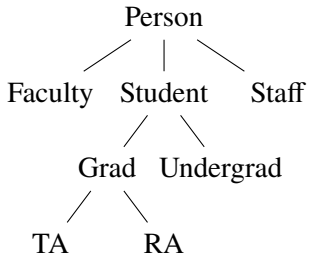

Figure 1: A Subtype Hierarchy

This example hierarchy describes a subtype relationship between different types. In this case, Student, Faculty, and Staff are all subtypes of Person. One could also say that Person is a *supertype* of all of Student, Faculty, and Staff. Similarly, TA is a subtype of Student and Person. This is because the subtype relationship must be reflexive and transitive, together making it a *preorder*.

Notationally, we will write $\tau <: \tau'$ to mean that τ is a subtype of τ'. Textbooks and papers sometimes also use other symbols (the most common being $\leq$), but we will stick with $<:$ here.

1 Basic Subtyping

Informally, the statement $\tau <: \tau'$ means that τ is more specific than τ', meaning anything that has type τ should be usable wherever we need or expect something of type τ'. To make this sensible, we require that $<:$ create a *preorder*. That is, it must be reflexive and transitive, obeying the following rules.

$$\frac{}{\tau <: \tau} \qquad \frac{\tau_1 <: \tau_2 \qquad \tau_2 <: \tau_3}{\tau_1 <: \tau_3}$$

Type systems usually formalize this intuition with a rule called a *subsumption* rule that allows a supertype to *subsume* terms of any of its subtypes.

$$[\textsc{Subsume}]\ \frac{\Gamma \vdash e : \tau' \qquad \tau' <: \tau}{\Gamma \vdash e : \tau}$$

Some languages also have maximal and minimal types, which we will call $\mathbf{1}$ and $\mathbf{0}$, respectively. The subtyping rules for these types are interesting.

- **1** (unit): Every type is a subtype of the maximal type **1**. That is, $\tau <: \mathbf{1}$ for every type τ. If a context expects something of type **1**, it can accept any value. In Java, this is equivalent to the type Object.
- **0** (void): The minimal type **0** is a subtype of every type. That is, $\mathbf{0} <: \tau$ for every type τ. If a context expects anything, it can accept a value of type **0**. In Java, this is the type of null.

Now let us investigate how subtyping works on some of the data types we have seen previously.

1.1 Products and Sums

When is something a subtype of a product type $\tau_1 \times \tau_2$? Since a product type has a pair of values, any value provided where we expect a product must itself have two values. Moreover, the first value must be usable whenever a τ_1 is expected and the second must be usable whenever a τ_2 is expected. In other words, the subtype must itself be a pair, and its elements must be subtypes of τ_1 and τ_2, respectively. This insight produces the following subtyping rule.

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2' <: \tau_2}{\tau_1' \times \tau_2' <: \tau_1 \times \tau_2}$$

For sum types, a similar logic applies. To provide a context with a $\tau_1' + \tau_2'$, then context must expect to receive a sum type where the left option can handle—expects a supertype of—τ_1' and the right option is a supertype of τ_2'. That is,

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2' <: \tau_2}{\tau_1' + \tau_2' <: \tau_1 + \tau_2}.$$

These rules say the product and sum type constructors ($\cdot \times \cdot$ and $\cdot + \cdot$, respectively) are *monotone* with respect to the subtyping relation. When the subtyping relationship is monotone—it goes in the same direction in the premise and the conclusion—it is called a *covariant* subtyping relationship.

1.2 Records

Recall the grammar for types and terms with record types:

$$\begin{array}{lcl} \ell & \in & \mathbf{Lab} \\ \tau & ::= & \cdots \mid \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \\ e & ::= & \cdots \mid \{\ell_1 = e_1, \ldots, \ell_n = e_n\} \mid e.\ell \end{array}$$

where $n \geq 1$. The ℓ_is are called *field identifiers*, are assumed to be distinct, and can appear in any order without changing the type. We had the following rule in the small-step operational semantics

$$\frac{1 \leq i \leq n}{\{\ell_1 = v_1, \ldots, \ell_n = v_n\}.\ell_i \longrightarrow v_i}$$

and the following typing rules

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \cdots \qquad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{\ell_1 = e_1, \ldots, \ell_n = e_n\} : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}} \qquad \frac{\Gamma \vdash e : \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} \qquad 1 \leq i \leq n}{\Gamma \vdash e.\ell_i : \tau_i}$$

To build the subtyping rules for record types, we need to consider what constitutes a "more precise" type that can be used when a record type is expected (or, conversely, what would be "less precise" than a record type). If a context expects a record of type $\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}$, then any value it receives must have two properties: (1) the value must have all of the fields $\ell_1, \ldots, \ell_n$, and (2) the type of field ℓ_i must be usable whenever a τ_i is expected.

Putting this intuition together points to two different subtyping rules for records.

- *Depth subtyping* is a typing relation where the records have the same fields, but those fields have a covariant subtyping relationship:

$$\frac{\tau_1' <: \tau_1 \qquad \cdots \qquad \tau_n' <: \tau_n}{\{\ell_1 : \tau_1', \ldots, \ell_n : \tau_n'\} <: \{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\}}$$

- *Width subtyping* is a relation where the subtype can have more fields than the supertype, but the types must be the same on fields in both records:

$$\frac{m \leq n}{\{\ell_1 : \tau_1, \ldots, \ell_n : \tau_n\} <: \{\ell_1 : \tau_1, \ldots, \ell_m : \tau_m\}}$$

  Note that in this case the subtype has *more* fields. That is because a context expecting fields $\ell_1, \ldots, \ell_m$ requires that they all be present, but can safely ignore the extra fields $\ell_{m+1}, \ldots, \ell_n$. This is analogous to the relationship between a subclass and a superclass; the subclass must have all of the fields of the superclass and may also have more.

It is possible to combine depth and width subtyping of records into a single, somewhat more complicated rule:

$$\frac{m \leq n \qquad \tau_1' <: \tau_1 \qquad \cdots \qquad \tau_m' <: \tau_m}{\{\ell_1 : \tau_1', \ldots, \ell_n : \tau_n'\} <: \{\ell_1 : \tau_1, \ldots, \ell_m : \tau_m\}}$$

Records as Indexed Products. Mathematically, record types can be viewed as product types whose components are indexed by the field labels ℓ. The operators $.\ell$ are the corresponding projections. Thus if Δ : **Lab** $\rightharpoonup$ **Type** is a partial function with finite domain $\mathrm{dom}(\Delta)$ associating a type $\Delta(\ell)$ with each element ℓ in its domain, we might write

$$\prod_{\ell \in \mathrm{dom}(\Delta)} \Delta(\ell)$$

for a record type, or just $\prod \Delta$ for short. An expression of this type would be a tuple indexed by $\mathrm{dom}(\Delta)$:

$$(e_\ell \mid \ell \in \mathrm{dom}(\Delta)).$$

In this view, the typing rules would take the form

$$\frac{\forall \ell \in \mathrm{dom}(\Delta).\, \Gamma \vdash e_\ell : \Delta(\ell)}{\Gamma \vdash (e_\ell \mid \ell \in \mathrm{dom}(\Delta)) : \prod \Delta} \qquad \frac{\Gamma \vdash e : \prod \Delta}{\Gamma \vdash e.\ell : \Delta(\ell)}$$

and the subtyping rule would take the form

$$\frac{\mathrm{dom}(\Delta_2) \subseteq \mathrm{dom}(\Delta_1) \qquad \forall \ell \in \mathrm{dom}(\Delta_2).\, \Delta_1(\ell) <: \Delta_2(\ell)}{\prod \Delta_1 <: \prod \Delta_2}$$

2 Function Subtyping

Based on what we have seen so far, our first impulse might be to write down something like the following to describe the subtyping relation on functions:

$$\frac{\tau_1' <: \tau_1 \qquad \tau_2' <: \tau_2}{\tau_1' \to \tau_2' <: \tau_1 \to \tau_2}$$

However, this would be incorrect. To see why, assume $\tau_1' <: \tau_1$ and $\tau_2' <: \tau_2$ and consider the functions

$$f : \tau_1 \to \tau_2$$
$$f' : \tau_1' \to \tau_2'$$

If a value v has type τ_1, we can apply $f\ v$ and have it produce a τ_2. If the above subtyping rule were correct, we would be able to use f' in place of f and apply $f'\ v$ without any type concerns. But that is not always safe! If τ_1 is a strict supertype of τ_1', then v may not have type τ_1' at all, and $f'\ v$ will crash. For instance, say $\tau_1 = \mathsf{int}$ and $\tau_1' = \mathsf{nat}$ and $v = -42$. In this case f can handle any integers, including negative ones, but f' can only handle natural numbers, meaning $f'\ -42$ will not work.

Actually, this incorrect typing rule was implemented in the language Eiffel, and runtime type checking had to be added later to make the language type safe.

To get the correct typing rule, we need to think about when it is safe to use f' in place of f. Succinctly, if every valid input to f is also a valid input of f' and every output of f' is also a valid output of f, then it is safe to use f' when f was expected. Translating that intuition to types, f takes input of type τ_1 while f' takes input of type τ_1'. So the intuition translate to requiring that every value of type τ_1 can be seen as a value of type τ_1': in other words, $\tau_1 <: \tau_1'$. For the output types, requiring every value f' can produce (type τ_2') to be a valid output of f (type τ_2) translates to $\tau_2' <: \tau_2$. The correct subtyping rule for functions is therefore:

$$\frac{\tau_1 <: \tau_1' \qquad \tau_2' <: \tau_2}{\tau_1' \to \tau_2' <: \tau_1 \to \tau_2}.$$

In the subtyping rules for all previous constructors, subtype ordering was preserved—the types were covariant. This remains true for *output* types on functions, but *input* types vary in the opposite direction. We say that the function type construct $\to$ is *contravariant* on its domain and *covariant* on its codomain.

3 References

To discuss subtyping of references, we first need typing rules for references. They are fairly straightforward and presented below.

$$\begin{aligned} \tau &::= \cdots \mid \mathsf{ref}\ \tau \\ e &::= \cdots \mid \mathsf{ref}\ e \mid e_1 := e_2 \mid\ !e \end{aligned}$$

$$[\textsc{Ref}]\ \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \mathsf{ref}\ \tau} \qquad [\textsc{Assign}]\ \frac{\Gamma \vdash e_1 : \mathsf{ref}\ \tau \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \mathsf{unit}} \qquad [\textsc{Deref}]\ \frac{\Gamma \vdash e : \mathsf{ref}\ \tau}{\Gamma \vdash\ !e : \tau}$$

As for subtyping, our first impulse might once again be to make the rule covariant. That is, if $\tau' <: \tau$, then $\mathsf{ref}\ \tau' <: \mathsf{ref}\ \tau$. As with functions, however, this would be incorrect. To see why, consider the following code snippet with types Square, Circle, and Shape where Square, Circle <: Shape. Assume sq : Square and c : Circle.

$$\begin{aligned} &\mathsf{let}\ x : \mathsf{ref\ Square} = \mathsf{ref}\ sq\ \mathsf{in} \\ &\mathsf{let}\ y : \mathsf{ref\ Shape} = x\ \mathsf{in} \\ &\quad y := c\ ;\ (!x).\mathsf{side} \end{aligned}$$

With the covariant typing rule above, this code will be well-typed. Unfortunately, it places a circle c in the location that x points to, and c.side will not be defined, causing an error. This problem actually exists in Java when using arrays, as the language incorrectly uses covariant subtyping for arrays. As a result, a runtime check is necessary.

As a demonstration, say the file `BadArray.java` contains the following code.

```
public class BadArray {
  public static void main(String[] args) {
    String[] a = new String[1];
    Object[] b = a;
    b[0] = Integer.valueOf(1);
  }
}
```

We can then run the following commands with the following outputs.

```
$ javac BadArray.java
$ java BadArray
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
        at BadArray.main(BadArray.java:5)
```

This example shows that ref cannot safely vary covariantly because an assignment will use it as an input, which can cause problems. But what about contravariance? If we try that, we end up with a similar problem, but the code is even simpler:

$$\begin{array}{l}\mathsf{let}\ y : \mathsf{ref}\ \mathsf{Shape} = \mathsf{ref}\ c\ \mathsf{in}\\ \mathsf{let}\ x : \mathsf{ref}\ \mathsf{Square} = y\ \mathsf{in}\\ \quad (!x).\mathsf{side}\end{array}$$

In other words, dereferencing treats a ref τ as an output, meaning it cannot safely be contravariant. The only option, then is *invariance*, which means the type can only change based on equivalence in the subtyping relation:

$$\frac{\tau <: \tau' \qquad \tau' <: \tau}{\mathsf{ref}\ \tau' <: \mathsf{ref}\ \tau}$$

Note that, for most languages, the subtyping relationship is antisymmetric—meaning $\tau <: \tau'$ and $\tau' <: \tau$ if and only if $\tau = \tau'$—which also makes it a partial order, not just a preorder. In that case, invariance requires equality of the two types. Some languages, however, are more permissive, so we do not force that restriction.