1 Introduction

In this lecture, we attempt to extend the typed λ-calculus to support objects. In particular, we explore the concept of *subtyping* in detail, which is one of the key features of object-oriented (OO) languages.

Subtyping was first introduced in Simula, the first object-oriented programming language. Its inventors Ole-Johan Dahl and Kristen Nygaard later 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 closely tied to inheritance and polymorphism and offers a formal way of studying them. It is best illustrated by means of an example. This is an example of a hierarchy that describes a

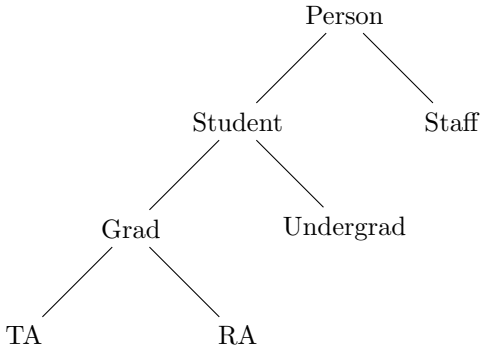

Figure 1: A Subtype Hierarchy

subtype relationship between different types. In this case, the types Student and Staff are both subtypes of Person. Alternatively, one can say that Person is a *supertype* of Student and Staff. Similarly, TA is a subtype of the Student and Person types, and so on. The subtype relationship is normally a preorder (reflexive and transitive) on types.

A subtype relationship can also be viewed in terms of subsets. If σ is a subtype of τ, then all elements of type σ are automatically elements of type τ.

The $\leq$ symbol is typically used to denote the subtype relationship. Thus, Staff $\leq$ Person, RA $\leq$ Student, etc. Sometimes the symbol $<:$ is used, but we will stick with $\leq$.

2 Basic Subtyping Rules

Formally, we write $\sigma \leq \tau$ to indicate that σ is a subtype of τ. In denotational semantics, this is equivalent to saying $[\![\sigma]\!] \subseteq [\![\tau]\!]$. The informal interpretation of this subtype relation is that anything of type σ can be used in a context that expects something of type τ. This is known as the *subsumption* rule:

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma \leq \tau}{\Gamma \vdash e : \tau}$$

Two further general rules are:

$$\tau \leq \tau \qquad\qquad \frac{\sigma \leq \tau \qquad \tau \leq \rho}{\sigma \leq \rho}$$

which say that $\leq$ is reflexive and transitive, respectively, thus a preorder. In many cases, antisymmetry holds as well, making the subtyping relation a partial order, but this is not always true.

The subtyping rules governing the types 1 and 0 are interesting:

- 1 (unit): Every type is a subtype of 1, that is, $\tau \leq 1$ for all types τ, thus 1 is the top type. If a context expects something of type 1, then it can accept any type. In Java, this is equivalent to the type Object.
- 0 (void): Every type is a supertype of 0, i.e., $0 \leq \tau$ for all types τ, thus 0 is the bottom type. The type 0 can be accepted by any context in lieu of any other type. In Java, this is the type of null.

2.1 Products and Sums

The subtyping rules for product and sum types are quite intuitive:

$$\frac{\sigma \leq \sigma' \qquad \tau \leq \tau'}{\sigma * \tau \leq \sigma' * \tau'} \qquad\qquad \frac{\sigma \leq \sigma' \qquad \tau \leq \tau'}{\sigma + \tau \leq \sigma' + \tau'}$$

These rules say that the product and sum type constructors are monotone with respect to the subtype relation.

2.2 Records

Recall our extensions to the grammar of e and τ for adding support for records types:

$$\begin{array}{rcl} e & ::= & \cdots \mid \{x_1 = e_1, \ldots, x_n = e_n\} \mid e.x \\ \tau & ::= & \cdots \mid \{x_1 : \tau_1, \ldots, x_n : \tau_n\} \end{array}$$

where $n \geq 1$. The x_i are called *field identifiers* and are assumed to be distinct. Also, the order in which they are listed in $\{x_1 = e_1, \ldots, x_n = e_n\}$ and $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ does not matter. We also had the following rule added to the small-step semantics:

$$\{x_1 = v_1, \ldots, x_n = v_n\}.x_i \ \rightarrow \ v_i$$

as well as the following typing rules:

$$\frac{\Gamma \vdash e_i : \tau_i, \ 1 \leq i \leq n}{\Gamma \vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}} \qquad\qquad \frac{\Gamma \vdash e : \{\ldots, x : \tau, \ldots\}}{\Gamma \vdash e.x : \tau}$$

There are two subtyping rules for records:

- *Depth subtyping*: this defines the subtyping relation between two records that have the same number of fields.

$$\frac{\sigma_i \leq \tau_i, \ 1 \leq i \leq n}{\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \leq \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

- *Width subtyping*: this defines the subtyping relation between two records that have different number of fields.

$$\frac{m \leq n}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_1 : \tau_1, \ldots, x_m : \tau_m\}} \tag{1}$$

  where the $\leq$ in the premise is integer comparison. Observe that in this case, the subtype has more components than the supertype. This is analogous to the relationship between a subclass and a superclass in which the subclass has all the components of the superclass, which it inherits from the superclass, but perhaps has some components that the superclass does not have.

The depth and width subtyping rules for records can be combined into a single rule:

$$\frac{m \le n \qquad \sigma_i \le \tau_i,\ 1 \le i \le m}{\{x_1 : \sigma_1, \ldots, x_n : \sigma_n\} \le \{x_1 : \tau_1, \ldots, x_m : \tau_m\}} \tag{2}$$

Mathematically, record types can be viewed as product types whose components are indexed by the field identifiers x. The operators $.x$ are the corresponding projections. Thus if Δ is a partial function with finite domain $\mathsf{dom}\,\Delta$ associating a type $\Delta(x)$ with each element x in its domain, we might write

$$\prod_{x \in \mathsf{dom}\,\Delta} \Delta(x)$$

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

$$(e_x \mid x \in \mathsf{dom}\,\Delta).$$

In this view, the typing rules would take the form

$$\frac{\Gamma \vdash e_x : \Delta(x),\ x \in \mathsf{dom}\,\Delta}{\Gamma \vdash (e_x \mid x \in \mathsf{dom}\,\Delta) : \prod \Delta} \qquad\qquad \frac{\Gamma \vdash e : \prod \Delta}{\Gamma \vdash e.x : \Delta(x)}$$

and the subtyping rule (2) would take the form

$$\frac{\mathsf{dom}\,\Delta_1 \subseteq \mathsf{dom}\,\Delta_2 \qquad \Delta_2(x) \le \Delta_1(x),\ x \in \mathsf{dom}\,\Delta_1}{\prod \Delta_2 \le \prod \Delta_1}$$

2.3 Variants

The analogous extension for sum types is known as *variant records* or just *variants*. These are just Δ-indexed coproducts

$$\sum_{x \in \mathsf{dom}\,\Delta} \Delta(x) \;=\; \sum \Delta.$$

In OCaml, one would write

```
type t = I of int | S of string | P of int * string
```

to declare the type of a variant record whose elements consist of the (tagged) union of three sets. Here $\mathsf{dom}\,\Delta = \{\mathtt{I}, \mathtt{S}, \mathtt{P}\}$ with $\Delta(\mathtt{I}) = \mathtt{int}$, $\Delta(\mathtt{S}) = \mathtt{string}$, and $\Delta(\mathtt{P}) = \mathtt{int * string}$.

The depth subtyping rule for variants is the same as that for records (replacing the records with variants). However, the width subtyping rule is different: instead of $m \le n$ in (1) and (2), we should take $n \le m$. Suppose we used the width subtyping rule (1), the same form as for records. Intuitively, if $\sigma \le \tau$, then this implies that anything of type σ can be used in a context expecting something of type τ. Suppose we now had a match statement that did pattern matching on something of type τ. Our subtyping relation says that we can pass in something of type σ to this match statement and it will still work. However, since τ has fewer components than σ and the match statement was originally written for an object of type τ, there will be values of σ for which no corresponding case exists. Thus, for variants, the direction of the $\le$ symbol in the premise needs to be reversed. In other words, for variants, the subtype should have *fewer* components than the supertype. This leads to the combined rule

$$\frac{\mathsf{dom}\,\Delta_1 \subseteq \mathsf{dom}\,\Delta_2 \qquad \Delta_1(x) \le \Delta_2(x),\ x \in \mathsf{dom}\,\Delta_1}{\sum \Delta_1 \le \sum \Delta_2}$$

3 Function Subtyping

Based on the subtyping rules we have encountered so far, our first impulse might be to write down something of the following form to describe the subtyping relation for functions:

$$\frac{\sigma \leq \sigma' \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

However, this is incorrect. To see why, consider the following code fragment:

$$\begin{array}{l}\mathsf{let}\ f : \sigma \to \tau = g\ \mathsf{in}\\ \mathsf{let}\ f' : \sigma' \to \tau' = g'\ \mathsf{in}\\ \mathsf{let}\ t : \sigma' = v\ \mathsf{in}\\ f'(t)\end{array}$$

Suppose $\sigma \leq \sigma'$ and $\tau \leq \tau'$. By the rule above, we would have $\sigma \to \tau \leq \sigma' \to \tau'$, thus we should be able to substitute f for f' and the resulting program should be type correct, provided the original one was. But it is not: if σ' is a strict supertype of σ and the value v is of type σ' but not of type σ, then f will crash, since it expects an input of type σ and it is not getting one.

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

The correct subtyping rule for functions is:

$$\frac{\sigma' \leq \sigma \qquad \tau \leq \tau'}{\sigma \to \tau \leq \sigma' \to \tau'}$$

Note that the ordering on the domain is reversed in the premise. Succinctly stated, a function f of type $\sigma \to \tau$ can safely take any input of any subtype σ' of σ, and produces a value that can be taken to be of any supertype τ' of τ; thus we are free to regard f as a function of type $\sigma' \to \tau'$.

In all the type constructors we had seen so far, the subtyping relation was preserved. With the function space constructor, however, the subtyping relation on the domain is reversed. We say that the function type constructor $\to$ is *contravariant* in the domain and *covariant* in the codomain.

4 References

Recall the extensions to the grammar of e and τ for adding support for references:

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

The typing rules are

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \mathsf{ref}\ e : \tau\ \mathsf{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \mathsf{ref}}{\Gamma \vdash\ !e : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau\ \mathsf{ref} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : 1}$$

As for subtyping, once again our first impulse might be to write down something like the following:

$$\frac{\sigma \leq \tau}{\sigma\ \mathsf{ref} \leq \tau\ \mathsf{ref}}$$

However, again this would be incorrect. The problem is that to be safe, a ref appearing on the left-hand side of an assignment operator should be contravariant. Consider the following example:

$\mathsf{let}\ x : \mathit{Square}\ \mathsf{ref} = \mathsf{ref}\ \mathit{square}\ \mathsf{in}$
$\mathsf{let}\ y : \mathit{Shape}\ \mathsf{ref} = x\ \mathsf{in}$
$(y := \mathit{circle}\ ;\ (!x).\mathit{side})$

Even though this code type-checks with the given subtyping rule for reference types, it is not type-correct, since in the last line x does not refer to a square anymore. This problem actually exists in Java when using arrays, as the designers incorrectly used the rule given above. Consequently, a runtime check is necessary.

```
public class Test {
  public static void main(String[] args) {
    B[] b = new B[1];
    A[] a = b;
    a[0] = new A();
  }
}

class A {}
class B extends A {}

Exception in thread ‘‘main’’ java.lang.ArrayStoreException: A
  at Test.main(Test.java:5)
```

In order to overcome this problem we must use the correct rule below:

$$\frac{\sigma \leq \tau \qquad \tau \leq \sigma}{\sigma\ \mathsf{ref} \leq \tau\ \mathsf{ref}}$$

The subtyping rule for references is thus both covariant and contravariant.