1 Introduction

In this lecture, we make an attempt to extend the typed λ-calculus for it to support more advanced data structures such as records and references. In particular, we explore the concept of *subtyping* in detail, which is one of the key features of object-oriented languages.

Subtyping was first introduced in SIMULA, considered 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 those of inheritance and polymorphism and offers a formal way of studying them. It is best illustrated by means of an example:

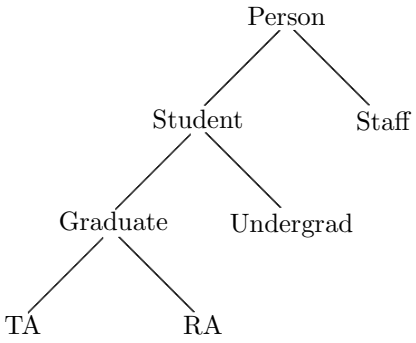

Figure 1 - A Subtype Hierarchy

This is an example of a subtype hierarchy, which describes the relationship between different entities. In this case, the Student and Staff types are both subtypes of the Person type (alternately, Person is the supertype of Student and Staff). Similarly, TA is a subtype of the Student and Person types and so on. A subtype relationship can also be thought of in terms of subsets. For example, this example can be visualized with the help of the following Venn diagram:

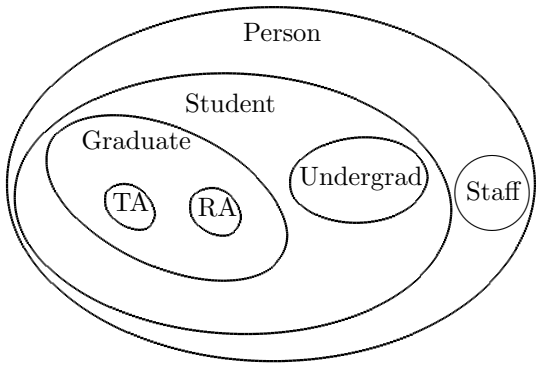

Figure 2 - Subtypes as Subsets

The $\leq$ symbol is typically used to denote the subtype relationship. Thus, Staff $\leq$ Person, RA $\leq$ Student and so on. Sometimes, the symbol $<:$ is also used; in this class, we will stick to the former. We will now study some properties of the subtyping relationship and describe it more formally.

2 Basic Subtyping Rules

Formally we notate the subtype relationship as: $\tau_1 \leq \tau_2$. In set notation, this is equivalent to $[\![\tau_1]\!] \subseteq [\![\tau_2]\!]$. The informal interpretation of this subtype relation is that anything of type τ_1 can be used in a context that expects something of type τ_2. This is known as the *subsumption* rule:

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}$$

There are two further general rules covering the subtyping relationship:

$$\frac{}{\tau \leq \tau} \qquad \text{(Reflexivity)}$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3} \qquad \text{(Transitivity)}$$

Since the $\leq$ relation is both reflexive and transitive, it is a pre-order. In most cases, anti-symmetry holds as well, making the subtyping relation a partial order, but this is not always true. The subtype relationships governing the 1 and 0 types are interesting:

- **unit** type: Being the top type, any type is a subtype of **unit**. If a context expects something of type **unit**, then it can accept any type i.e., $\forall\tau.\ \tau \leq \textbf{unit}$. In Java, this is much like the **Object** type.
- We can also introduce a bottom type **void** that is a universal subtype. This type can be accepted by any context in lieu of any other type: i.e., $\forall\tau.\ \textbf{void} \leq \tau$. This type is useful for describing the result of a computation that never terminates, or that transfers control somewhere else rather than producing a value.

The type hierarchy thus looks like the following:

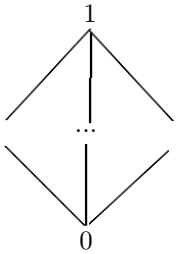

Figure 3 - Type Hierarchy for Typed Lambda Calculus

3 Subtyping Rules for Product and Sum Types

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

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 * \tau_2 \leq \tau_1' * \tau_2'} \qquad \frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 + \tau_2 \leq \tau_1' + \tau_2'}$$

4 Subtyping Rules for Records

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

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

We also had the following rule added to the small-step semantics:

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

and the following typing rules:

$$\frac{\Gamma \vdash e_i : \tau_i \quad (\forall i : 1 \ldots n)}{\Gamma \vdash \{x_1 = e_1, \ldots, x_n = e_n\} : \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{x_1 : \tau_1, \ldots, x_i : \tau_i, \ldots, x_n : \tau_n\}}{\Gamma \vdash e.x_i : \tau_i}$$

There are two types of subtyping rules for records:

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

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2' \quad \ldots \quad \tau_n \leq \tau_n'}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_1 : \tau_1', \ldots, x_n : \tau_n'\}}$$

- Width subtyping: a subtyping relation between two records that have different number of fields.

$$\frac{m \leq n}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_1 : \tau_1, \ldots, x_m : \tau_m\}} \qquad \text{(the } \leq \text{ in the premise is an integer comparison)}$$

  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, where the former has more components than the latter.

The depth and width subtyping rules for records can in fact be combined to yield a single equivalent rule:

$$\frac{m \leq n \quad \tau_i \leq \tau_i' \quad (\forall i : 1 \ldots n)}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_1 : \tau_1', \ldots, x_m : \tau_m'\}}$$

Records can be viewed as tagged product types of arbitrary length; the analogous extension for sum types are variants. The depth subtyping rule for variants is the same as that given above for records (replacing the records with variants). The width subtyping rule is however different and we will see why this is so. Suppose we used a width subtyping rule of the same form as given above. Recall that if $\tau_1 \leq \tau_2$, then this implies that anything of type τ_1 can be used in a context expecting something of type τ_2. Suppose we now had a case statement that did pattern matching on something of type τ_2; our subtyping relation says that we can pass in something of type τ_1 to this case statement and still have it work. However, since τ_2 has fewer components than τ_1 and the case statement was originally written for an object of type τ_2, there will be values of τ_1 for which no corresponding pattern match exists. Thus, for variants, the direction of the $\leq$ symbol in the premise of the width subtyping rule given above needs to be reversed i.e., for variants, the subtype will have fewer components than the supertype.

5 Function Subtyping

Based on the subtyping rules we have encountered up to this point, our first impulse is perhaps to write down something like the following to describe the subtyping relation for functions:

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

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

$$\begin{array}{l}\textbf{let } f : \tau_1 \to \tau_2 = f_1 \textbf{ in} \\ \quad \textbf{let } f' : \tau_1' \to \tau_2' = f_2 \textbf{ in} \\ \quad\quad \textbf{let } t : \tau_1' = v_1 \textbf{ in} \\ \quad\quad\quad f'(t')\end{array}$$

In the example above, since $f \leq f'$, we should be able to use f where f' was expected. Therefore we should be able to call $f(t')$. But f expects an input of type τ_1 and gets instead an input of type τ_1', so we should be able to use τ_1' where τ_1 is expected, which in fact implies that we should have $\tau_1' \leq \tau_1$ instead of $\tau_1 \leq \tau_1'$ as given.

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. Thus, the correct subtyping rule for functions is:

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}$$

All the subtyping rules we had seen thus far were *covariant* i.e., the subtyping relation related subexpressions in the premise in the same direction as in the conclusion. In general, we say that a type constructor F is covariant in one of its arguments τ if whenever $\tau \leq \tau'$, it is the case that $F(\tau) \leq F(\tau')$. The function subtyping rule is our first *contravariant* rule—the direction of the subtyping relation is reversed in the premise.

6 Subtyping Rules for References

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

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

where we add the following typing rules:

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

As for the subtyping rule, once again, our first impulse would be to write down something of the following form:

$$\frac{\tau_1 \leq \tau_2}{\tau_1 \textbf{ ref} \leq \tau_2 \textbf{ ref}}$$

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

$$\begin{array}{l}\textbf{let } x : \mathit{Square} \textbf{ ref} = \textbf{ref } \mathit{square} \textbf{ in} \\ \quad \textbf{let } y : \mathit{Shape} \textbf{ ref} = x \textbf{ in} \\ \quad\quad (y := \mathit{circle};\ (!x).\mathit{side})\end{array}$$

Even though this code type-checks with the given subtyping rule for reference types, it will cause a run-time error, because in the last line x does not refer to a square anymore. This problem actually exists in Java when using arrays. Apparently guided by a desire for simplicity, the designers used the rule given above, with the result that soundness is achieved in the Java type system only by run-time type checks whenever a value is assigned into a array of objects. To avoid this problem, we use the rule below:

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_1}{\tau_1 \textbf{ ref} \leq \tau_2 \textbf{ ref}}$$

which equivalently can be written as:

$$\frac{}{\tau\ \mathbf{ref} \leq \tau\ \mathbf{ref}}$$

The correct subtyping rule for references is thus neither covariant nor contravariant in τ, but rather *invariant* in τ.