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:

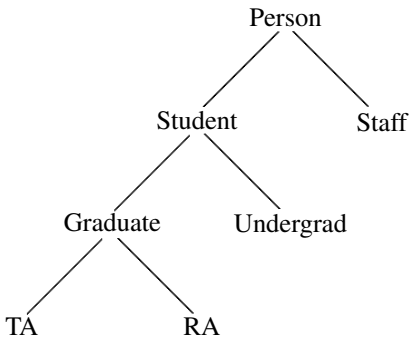

Figure 1: A Subtype Hierarchy

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:

The $\leq$ symbol is typically used to denote the subtype relationship. Thus, Staff $\leq$ Person, RA $\leq$ Student and so on.[1]

1.1 Subtyping as inclusion

The statement $\tau_1 \leq \tau_2$ means that a τ_1 can be used wherever a τ_2 is expected. One way to think of this is in terms of sets of values corresponding to these types. Any value of type τ_1 must also be a value of type τ_2. Assuming a type interpretation $\mathcal{T}[\![\tau]\!]$ that gives the set of elements, we understand $\tau_1 \leq \tau_2$ to mean $\mathcal{T}[\![\tau_1]\!] \subseteq \mathcal{T}[\![\tau_2]\!]$.

1.2 Subtyping as coercion

Another view of subtyping is when a value of the type τ_2 is expected and a value of τ_1 is supplied instead, the compiler or run-time system is able to automatically *coerce* the supplied value to the correct type. Given a subtyping relationship $\tau_1 \leq \tau_2$, we can define a *coercion function* $\Theta(\tau_1 \leq \tau_2) : \tau_1 \to \tau_2$ that shows how to do this coercion. A language that supports subtyping can then be translated via coercion to a language that does not have subtyping.

[1] Some people, like Pierce, use the symbol $<:$, but in this class, we will stick to $\leq$.

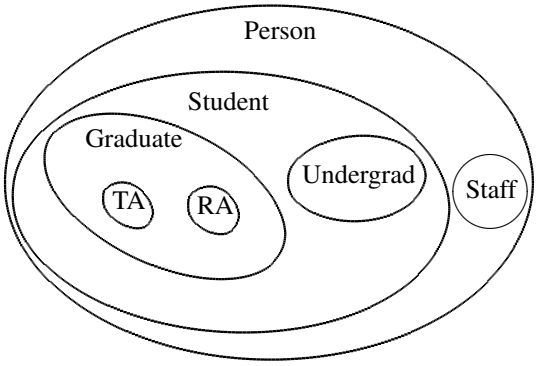

Figure 2: Subtypes as Subsets

2 Subtyping Rules

The informal interpretation of this subtype relation $\tau_1 \leq \tau_2$ is that anything of type τ_1 can be used in a context that expects something of type τ_2. This is formalized as the *subsumption* rule:

$$\frac{\Gamma \vdash e : \tau \quad \vdash \tau \leq \tau'}{\Gamma \vdash e : \tau'} \text{ (SUB)}$$

Notice that the right premise in this rule is actually a side condition, relying on a separate, new judgement of the form $\vdash \tau_1 \leq \tau_2$. We still have to define this judgement.

There are two general rules regarding the subtyping relationship:

$$\frac{}{\tau \leq \tau} \text{ (REFL)}$$

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

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:

- The unit type 1: Being the top type, any type can be treated as a subtype of 1. If a context expects something of type 1, it can never be disappointed by a different value. Therefore, $\forall \tau.\ \tau \leq 1$. In Java, this is much like the type **void** that is the type of statements and of methods that return no value.

  The coercion function for this subtyping relationship is trivial:

  $$\Theta[\![\tau \leq 1]\!] = \lambda x{:}\tau.\, \mathbf{null}$$

- We can also introduce a bottom type 0 that is a universal subtype. This type can be accepted by any context in lieu of any other type: i.e., $\forall \tau.\ 0 \leq \tau$. This type is useful for describing the result of a computation that never produces a value. For example, it never terminates, or it transfers control somewhere else rather than producing a value.

The type hierarchy thus looks as shown in Figure 3.

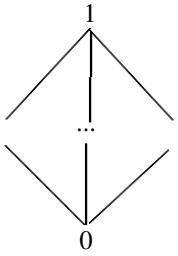

Figure 3: Type Hierarchy for Typed Lambda Calculus

3 Subtyping Rules for Product and Sum Types

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

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

These rules showing that subtyping on products and sum is *covariant*: the direction of subtyping in the arguments to the type constructor $(+, *)$ is same as the direction on the output of the type. In general, we say that a type constructor F has subtyping that is covariant in one of its arguments τ if whenever $\tau \leq \tau'$, it is the case that $F(\tau) \leq F(\tau')$.

Coercion functions for these rules are as follows. The definition of Θ is well-typed by induction on the derivation of the subtyping relationship.

$$\Theta(\tau_1 * \tau_2 \leq \tau_1' * \tau_2') = \lambda x{:}\tau_1 * \tau_2.\, (\Theta(\tau_1 \leq \tau_1')\ (\#1\ x), \Theta(\tau_2 \leq \tau_2')\ (\#2\ x))$$
$$\Theta(\tau_1 + \tau_2 \leq \tau_1' + \tau_2') = \lambda x{:}\tau_1 + \tau_2.\, \mathbf{case}\ x\ \mathbf{of}\ (y_1{:}\tau_1).\mathbf{inl}\ (\Theta(\tau_1 \leq \tau_1')\ y_1) \mid (y_2{:}\tau_2).\mathbf{inr}\ (\Theta(\tau_2 \leq \tau_2')\ y_2)$$

4 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:

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'}\ (\textsc{BrokenFunSub})$$

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

$\mathbf{let}\ f : \tau_1 \to \tau_2 = f_1\ \mathbf{in}$
  $\mathbf{let}\ f' : \tau_1' \to \tau_2' = f_2\ \mathbf{in}$
    $\mathbf{let}\ t : \tau_1' = v_1\ \mathbf{in}$
      $f'(t')$

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.

We can derive the correct subtyping rule for functions by thinking about Figure 4. The outer box represents a context expecting a function of type $\tau_1' \to \tau_2'$. The inner box is a function of type $\tau_1 \to \tau_2$. The arrows show the direction of flow of information. So arguments from the outside of type τ_1' are passed to a function expecting τ_1. Therefore we need $\tau_1' \leq \tau_1$. Results of type τ_2 are returned to a context expecting τ_2'. Therefore we need $\tau_2 \leq \tau_2'$. The rule is:

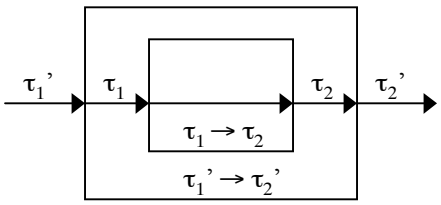

Figure 4: Function subtyping

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \text{ (FUNSUB)}$$

The function subtyping rule is our first *contravariant* rule—the direction of the subtyping relation is reversed in the premise for the first argument (τ_1).

A coercion function for this relationship can be defined too:

$$\Theta(\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2') = \lambda f{:}\tau_1 \to \tau_2.\, (\lambda x{:}\tau_1'.\, \Theta(\tau_2 \leq \tau_2')\ (f\ (\Theta(\tau_1' \leq \tau_1)\ x)))$$

Notice that If we tried to write this coercion function for BROKENFUNSUB, we wouldn't be able to, because we would have no way to convert the argument x from τ_1 to τ_1'.

The rule for function subtyping determines how object-oriented languages can soundly permit subclasses to override the types of methods. If we write a declaration $\mathbf{C}$ **extends** $\mathbf{D}$ in a Java program, for example, it had better be the case that the types of methods in $\mathbf{C}$ are subtypes of the corresponding types in $\mathbf{D}$. Checking this is known as checking the *conformance* of $\mathbf{C}$ with $\mathbf{D}$.

It is sound for object-oriented languages to check conformance by using a more restrictive rule than FUNSUB, and Java does: as of Java 1.5, it allows the return types of methods to be refined covariantly in subclasses (the $\tau_2 \leq \tau_2'$ part of the rule above), which is sound. Java doesn't, however, permit contravariant generalization of method arguments, probably because it would interact poorly with the Java overloading mechanism.

It took a surprisingly long time for everyone to agree on the right subtyping rule for functions. The broken rule BROKENFUNSUB was actually used for conformance checking in the language Eiffel. Run-time type-checking had to be added later to make the language type-safe. More recent work on *family inheritance* mechanisms such as *virtual classes* and *nested inheritance* shows how to soundly permit some of the covariant overriding that the Eiffel designers wanted.

5 A subtyping semantics by type-preserving translation *

We can give a semantics for subtyping as a *type-preserving translation* from the language with subtyping to the language without subtyping. A type-preserving translation translates well-typed source terms to well-typed target terms. The fact that the translation is well-typed can be seen by induction on the typing derivation; in fact, we write the translation as operating on typing derivations, just as we did for denotational semantics. Therefore, $\mathcal{E}[\![\Gamma \vdash e : \tau]\!]$ is the translation of e in context Γ to a well-typed term. The type preservation property is the following:

$$\mathcal{G}[\![\Gamma]\!] \vdash \mathcal{E}[\![\Gamma \vdash e : \tau]\!] : \mathcal{T}[\![\tau]\!]$$

where we define translations $\mathcal{G}[\![\Gamma]\!]$ and $\mathcal{T}[\![\tau]\!]$ that explain how to translate contexts and types respectively. In this case the translations of contexts and types are trivial: $\mathcal{G}[\![\Gamma]\!] = \Gamma$ and $\mathcal{T}[\![\tau]\!] = \tau$ respectively. Throughout the translation, we use coercion functions $\Theta(\tau_1 \leq \tau_2)$, which we know have the type $\tau_1 \to \tau_2$, by induction on the derivation of $\tau_1 \leq \tau_2$. First, consider the case where the typing derivation of e ends in subsumption:

$$\mathcal{E}[\![\Gamma \vdash e : \tau]\!] = \Theta(\tau' \leq \tau)\ \mathcal{E}[\![\Gamma \vdash e : \tau']\!]$$

By induction on the typing derivation (Since $\Gamma \vdash e : \tau' \prec \Gamma \vdash e : \tau$), we can assume $\mathcal{E}[\![\Gamma \vdash e : \tau']\!] : \tau'$, so the right-hand side is a well-typed application, with type τ, as required. The remainder of the translation is as follows:

$$\begin{aligned}
\mathcal{E}[\![\Gamma \vdash \mathbf{null} : 1]\!] &= \mathbf{null} \quad : 1 \\
\mathcal{E}[\![\Gamma, x{:}\tau \vdash x : \tau]\!] &= x \quad : \tau \\
\mathcal{E}[\![\Gamma \vdash e_0\ e_1 : \tau]\!] &= \mathcal{E}[\![e_0 : \tau' \to \tau]\!]\ \mathcal{E}[\![e_1 : \tau']\!] \\
\mathcal{E}[\![\Gamma \vdash \lambda x{:}\tau'.\, e : \tau' \to \tau]\!] &= \lambda x{:}\tau'.\, \mathcal{E}[\![\Gamma, x{:}\tau' \vdash e : \tau]\!]
\end{aligned}$$

In each case, we can see that the right-hand side of the translation uses translations of the premises of the appropriate typing rule. Therefore, by induction on the typing derivation, the right-hand side has the correct type.

For example, in the case $\mathcal{E}[\![\Gamma \vdash \lambda x{:}\tau'.\, e : \tau' \to \tau]\!]$, we have by the induction hypothesis that $\Gamma, x{:}\tau' \vdash \mathcal{E}[\![\Gamma, x{:}\tau' \vdash e]\!] : \tau'$, and therefore we can conclude $\Gamma \vdash \lambda x{:}\tau'.\, \mathcal{E}[\![\Gamma, x{:}\tau' \vdash e : \tau]\!] : \tau' \to \tau$.

This the the same process we went through to argue that the denotational semantics was sound for $\lambda^{\to}$, but now we have a type system on the right-hand side.

6 Records

Our goal in exploring subtyping was partly to understand the nature of object-oriented languages. However, we haven't seen any types that look much like objects yet. We can add *record types* that look a great deal like objects, and get some useful insights.

A record is a collection of named fields, each with its own type. We extend the grammar of e and τ for adding support for record types:

$$\begin{aligned}
e &::= \ldots \mid \{x_1 = e_1, \ldots, x_n = e_n\} \mid e.x \\
v &::= \ldots \mid \{x_1 = v_1, \ldots, x_n = v_n\} \\
\tau &::= \ldots \mid \{x_1 : \tau_1, \ldots, x_n : \tau_n\}
\end{aligned}$$

We add following the rule 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:

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

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

What we can see from this is that the record type $\{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ acts a lot like the product type $\tau_1 * \cdots * \tau_n$. This suggests that subtyping on records should behave like subtyping on products.
There are actually three types of reasonable subtyping rules for records:

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

$$\frac{\tau_i \le \tau_i' \ \ ^{\forall i \in 1..n}}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \le \{x_1 : \tau_1', \ldots, x_n : \tau_n'\}}$$

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

$$\frac{}{\{x_1 : \tau_1, \ldots, x_{n+1} : \tau_{n+1}\} \leq \{x_1 : \tau_1, \ldots, x_n : \tau_n\}}$$

  Observe that in this case, the subtype has more components than the supertype. This is the kind of subtyping that most programmers are familiar with in a language like Java. It allows programmers to define a subclass with more fields and methods than in the superclass, and have the objects of the subclass be (soundly) treated as objects of the superclass.

- Permutation subtyping: a relation between records with the same fields, but in a different order. Most languages don't support this kind of subtyping because it prevents compile-time mapping of field names to fixed offsets within memory (or to fixed indices in a tuple). Notice that permutation subtyping is not antisymmetric.

$$\frac{\{a_1, \ldots, a_n\} = \{1, \ldots, n\}}{\{x_1 : \tau_1, \ldots, x_n : \tau_n\} \leq \{x_{a_1} : \tau_{a_1}, \ldots, x_{a_n} : \tau_{a_n}\}}$$

The depth and width subtyping rules for records can be combined to yield a single equivalent rule that handles all transitive applications of both rules:

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

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.

7 Subtyping Rules for References

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

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

where we add the following typing rules:

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

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

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

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

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

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. To avoid this problem, we do not introduce a subtyping relation between two ref types until they contain exactly the same type (the reflexive relationship). Thus, the correct subtyping rule for references is thus neither covariant nor contravariant in τ, but rather *invariant* in τ.