CS 4120/5120 Lecture 11 Subtype Polymorphism 22 Feb 2016

Lecturer: Andrew Myers

1 Introduction

We now explore the concept of *subtyping*, 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:

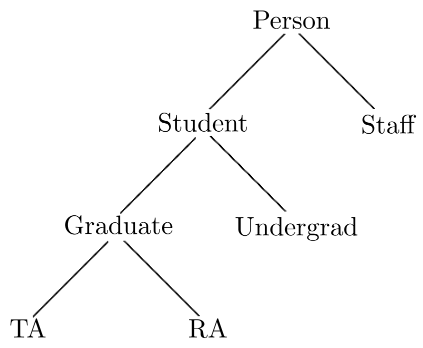

Figure 1: A Subtype Hierarchy

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:

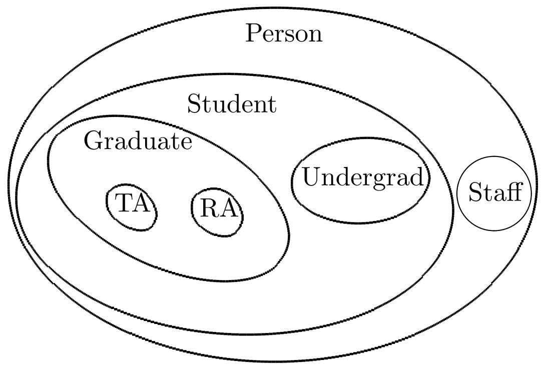

Figure 2: Subtypes as Subsets

The $\leq$ symbol is typically used to denote the subtype relationship. Thus, Staff $\leq$ Person, RA $\leq$ Student and so on.[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 that $\mathcal{T}[\![\tau]\!]$ is the set of elements of type τ, we understand $\tau_1 \leq \tau_2$ to mean $\mathcal{T}[\![\tau_1]\!] \subseteq \mathcal{T}[\![\tau_2]\!]$.

2 Subtyping Rules

The informal interpretation of the subtype relationship $\tau_1 \leq \tau_2$ 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:

$$\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau'}{\Gamma \vdash e : \tau'}\ (\textsc{Sub})$$

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

The subsumption rule is a perfectly well-defined typing rule, but it has one serious problem for practical application: it is not syntax-directed. Given a judgment to be derived, we can't tell whether to use the subsumption rule or a rule whose conclusion matches the syntax of the desired judgment.

There are two general rules regarding the subtyping relationship:

$$\frac{}{\tau \leq \tau}\ (\textsc{Refl})$$

$$\frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ (\textsc{Trans})$$

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.
- 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.

3 Subtyping on Product Types (Tuples)

The simplest kind of data structure is a pair (a 2-tuple), with the type $\tau_1 * \tau_2$. It represents a pair (v_1, v_2) where v_1 is a τ_1 and v_2 is a τ_2. The operations supported by a pair are to extract the first (left) component or the second (right). The most permissive sound subtyping rule for this type is quite intuitive:

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

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

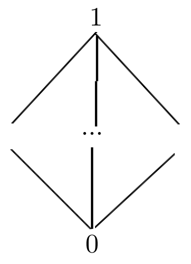

Figure 3: Type Hierarchy for Typed Lambda Calculus

This rule is *covariant*: the direction of subtyping in the arguments to the type constructor ($*$) is same as the direction on the result of the constructor. 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 $F(\tau) \leq F(\tau')$.

4 Records

We are particularly interested in how to handle subtyping in the context of object-oriented languages. *Record types* correspond closely to object types and yield some useful insights.

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

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

with the following typing rules:

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

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

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 $\tau_1 * \cdots * \tau_n$. Records can be viewed as tagged product types of arbitrary length. 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.

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

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

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

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:

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

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:

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

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

```
let $f : \tau_1 \to \tau_2 = f_1$ in
  let $f' : \tau_1' \to \tau_2' = f_2$ in
    let $t : \tau_1' = v_1$ 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.

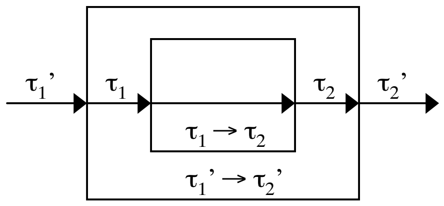

Figure 4: Function subtyping

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:

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

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).

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 `C extends D` in a Java program, for example, it had better be the case that the types of methods in `C` are subtypes of the corresponding types in `D`. Checking this is known as checking the *conformance* of `C` with `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.

6 Subtyping Rules for Arrays

What about subtyping on array types?

$$\tau \quad ::= \quad \ldots \ \mid \ \tau\,\texttt{[]}$$

For the subtyping rule, our first impulse might be to write down a covariant rule:

$$\frac{\tau_1 \leq \tau_2}{\tau_1\,\texttt{[]} \leq \tau_2\,\texttt{[]}}$$

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

```
let x:Square[] = new Square[0]
let y:Shape[]= x
y[0] := circle
x[0].corners() // do something that only squares can do!
```

Even though this code type-checks with the given subtyping rule for array types, it will cause a run-time error, because in the last line `x[0]` does not evaluate to a square. To avoid this problem, the subtyping relation on array types should be *invariant* in τ.

Since the equivalent code type-checks in Java, you might be wondering how Java can get away with a covariant subtyping rule. The answer is that Java uses extra run-time checking. At the third line, the assignment to `y[0]` will failed with a run-time exception. This not only makes code less robust but also imposes a significant run-time cost on use of arrays.