Scribe: Michael Clarkson, Nikita Proskourine Lecturer: Andrew Myers

1 Motivation

Object-oriented programming (OOP) emerged as a dominant programming paradigm in the 1990s. Programmers utilized OOP techniques to achieve better extensibility, modularity, and reusability of code. One of the key features of OOP is the *subtype relationship*.

As an informal example of subtypes, consider Figure 1:

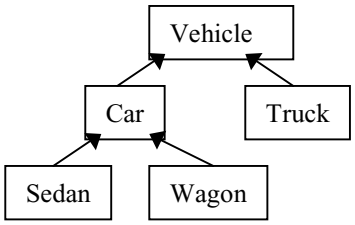

Figure 1: Example subtype hierarchy

In this figure, Car and Truck are both subtypes of Vehicle, and Wagon and Sedan are both subtypes of Car.

Subtyping was introduced in the language SIMULA 67[1] along with other important OO features such as inheritance and classes. The language was intended for "easy generation of simulation programs for discrete event systems"[2]. The types of systems the language designers were interested in modeling came from what we would now call operations research, such as transportation systems and production lines. Subtyping was a key feature in the language, since it allowed coding of algorithms that could be applied to several data types. For example, a function that manipulated Vehicles could also be applied to Sedans and Trucks.

Subtyping has since been incorporated into many other languages, notably C++ (1986), Modula-3 (1993), and Java (1995). Untyped object-oriented languages like Smalltalk have also used some of the ideas from Simula.

2 The subtype relationship

We will notate the subtype relationship as

$$\tau_1 \leq \tau_2$$

which should be read as "τ_1 is a subtype of τ_2", or alternately, "all values of type τ_1 are of type τ_2". Continuing our earlier example, we could write Car $\leq$ Vehicle, since all Cars are Vehicles.

2.1 Subtypes as subsets

One interpretation of the subtype relationship is as the subset relationship. For example, $\{c|c : \mathsf{Vehicle}\} \subseteq \{v|v : \mathsf{Vehicle}\}$, as shown in the Venn diagram in Figure 2.

[1] SIMUlation LAnguage, originally meant to be a package built on ALGOL 60 but by the early 1970s developed into a language with its own compiler

[2] Dahl, Ole-Johan and Nygaard, Kristen. *SIMULA: A Language for Programming and Description of Discrete Event Systems. Introduction and User's Manual*. May 1965. NCC.

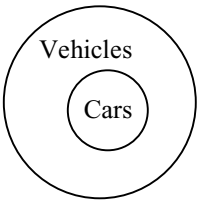

Figure 2: Subtypes as subsets

2.2 Using product and sum constructors with subtypes

Using record notation, which we formalize later in this lecture, we can describe the internals of types Car and Vehicle as:

```
Vehicle  =  { location : float×float, maxSpeed : float }
    Car  =  { location : float×float, maxSpeed : float, doors : int, passengers : int }
```

This leads to the counter-intuitive observation that although Car $\leq$ Vehicle, fields(Car) $\supseteq$ fields(Vehicle).

The records we use here have named fields, making them similar to a product type where the projection functions operate on names rather than indices. We can also use sum types to describe the relationships between our example types:

```
Vehicle  =  Car | Truck
    Car  =  Sedan | Wagon
```

This leads to some awkwardness. To use s:Sedan as a Vehicle, we have to explicitly inject it as Vehicle(Car(s)). In the other direction, to find out what type of vehicle v:Vehicle is, we have to write several case patterns:

```
case v of
  Vehicle (Car(Sedan(s)))   ⇒ ...
  Vehicle (Car(Wagon(w)))   ⇒ ...
  Vehicle (Truck(t))        ⇒ ...
```

These cases and injections lead to code that is inherently difficult to extend. For example, each time we add a new subclass of Vehicle, we have to add new cases. Subtype polymorphism will help us solve this problem in an elegant manner.

2.3 Subtyping rules

We will notate a subtyping judgment as:

$$\Delta \vdash \tau_1 \leq \tau_2$$

The Δ may be necessary as a context for type variables in a language with recursive types or with parametric polymorphism, but for now we will dispense with it.

The basic rules for subtyping are:

$$\frac{}{\vdash \tau \leq \tau} \quad \text{Reflexivity}$$

$$\frac{\vdash \tau_1 \leq \tau_2 \quad \vdash \tau_2 \leq \tau_3}{\vdash \tau_1 \leq \tau_3} \quad \text{Transitivity}$$

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

The intuition behind the subsumption rule is that given an expression with a hole:

$$\ldots [\cdot]_{\tau'} \ldots$$

we can plug in an expression of type τ without changing the type of the enclosing context.

Given that $\leq$ is reflexive and transitive, it is natural to ask whether it is also anti-symmetric, and thus usable as the ordering relation $\sqsubseteq$ in a partial order. However, the answer to this question depends upon the type system of the language in question. In some languages, we do in fact have the rule:

$$\frac{\vdash \tau_1 \leq \tau_2 \quad \vdash \tau_2 \leq \tau_1}{\vdash \tau_1 = \tau_2} \quad \text{True anti-symmetry}$$

where $=$ means syntactic identity. In most languages, though, this is relaxed to type equivalence:

$$\frac{\vdash \tau_1 \leq \tau_2 \quad \vdash \tau_2 \leq \tau_1}{\vdash \tau_1 \cong \tau_2} \quad \text{Equivalence in terms of subtyping}$$

In the latter (and usual) case, we have a relation that is only reflexive and transitive. Though not enough to directly define a partial order, we can define a *preorder* with this relation. A preorder gives a set of equivalence classes ordered by the relation, from which we can define a partial order where the elements are the equivalence classes.

3 Records and variants

3.1 Syntax and typing rules

Let us introduce a record type:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\} \\ e & ::= & \ldots \mid \{l_1 = e_1, l_2 = e_2, \ldots, l_n = e_n\} \mid e.l \end{array}$$

These rules allow us to define a record and select fields from a record. This seems to describe just a fancy, labelled, version of a product type, but when we later introduce subtyping we'll get a richer type. The typing rules for records are:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \cdots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \{l_1 = e_1, l_2 = e_2, \ldots, l_n = e_n\} : \{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\}}$$

$$\frac{\Gamma \vdash e : \{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\}}{\Gamma \vdash e.l_i : \tau_i}$$

While records are extensions of products, variant types are extensions of sums. They are somewhat like ML's datatype:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid [l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n] \\ e & ::= & \ldots \mid \mathsf{inj}[\tau](l = e) \mid \mathsf{case}\ e\ \mathsf{of}\ l_1(x) \Rightarrow e_1, l_2(x) \Rightarrow e_2, \ldots, l_n(x) \Rightarrow e_n \end{array}$$

A given variant value has only one of the types possible for that variant.

3.2 Subtype relationships

How we define subtype relationships between these types has implications on the implementation of the type checker. There are two different forms of subtyping we would like to consider: width subtyping and depth subtyping.

Width subtyping means that a subtype has extra fields:

$$\frac{m \leq n}{\{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_m : \tau_m\}} \quad \text{Width subtyping (records)}$$

While depth subtyping means that the fields of a subtype are themselves subtypes:

$$\frac{\tau_i \leq \tau_i' \quad i \in 1 \ldots n}{\{l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n\} \leq \{l_1 : \tau_1', l_2 : \tau_2', \ldots, l_n : \tau_n'\}} \quad \text{Depth subtyping (records)}$$

These rules are implicitly saying that the order of the fields matters: we can extend only on the right side. The benefit of this order is that we can compute addresses for the fields at compile time. In some languages (CLU), the order of the fields doesn't matter, but this is expensive.

Variants have equivalent rules for width and depth subtyping:

$$\frac{\tau_i \leq \tau_i' \quad i \in 1 \ldots n}{[l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n] \leq [l_1 : \tau_1', l_2 : \tau_2', \ldots, l_n : \tau_n']} \quad \text{Depth subtyping (variants)}$$

$$\frac{n \leq m}{[l_1 : \tau_1, l_2 : \tau_2, \ldots, l_n : \tau_n] \leq [l_1 : \tau_1, l_2 : \tau_2, \ldots, l_m : \tau_m]} \quad \text{Width subtyping (variants)}$$

Note that in the width rule for variants $n \leq m$, in contrast to $m \leq n$ in the width rule for records.

4 Subtyping functions

We may try to express subtype relationship between functions as:

$$\frac{\tau_1 \leq \tau_1' \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \quad \text{Function subtyping (INCORRECT)}$$

This was the original typing rule in the language Eiffel. However, it was discovered that this rule made the type system unsound! Instead, we need to change $\tau_1 \leq \tau_1'$ to $\tau_1' \leq \tau_1$. The reversal of subtyping relationship between τ_1 and τ_1' is illustrated by Figure 3. We have to be able to use τ_2 as τ_2' and τ_1 as τ_1'.

The correct rule is thus:

$$\frac{\tau_1' \leq \tau_1 \quad \tau_2 \leq \tau_2'}{\tau_1 \to \tau_2 \leq \tau_1' \to \tau_2'} \quad \text{Function subtyping}$$

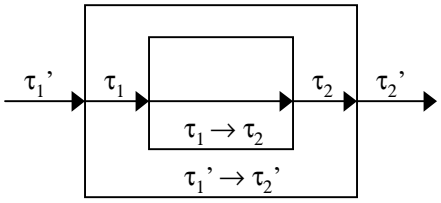

Figure 3: Function subtyping

Function subtyping is *covariant* in the result type and *contravariant* in the argument type. Another way to say this is that functions are antimonotonic in the argument type. This is different from record and variant types, where subtyping was always covariant.

5 Type hierarchy

In addition to types discussed above we have subtype relationships for the unit and 0 types:

$\overline{\tau \leq \mathsf{unit}}$ all you can do with unit is pass it along; any value is good enough

$\overline{0 \leq \tau}$ because the context expecting the τ never receives a value that breaks it

The remaining class of types to consider is the recursive types. The algorithm for type-checking recursive types is a lot like our type equivalence algorithm, applied to infinite trees. In fact, if in that algorithm we replace the equivalence relation with subtyping, and modify the premise for functions to reflect the contravariance in the argument type, we get a correct type-checking algorithm for recursive types.

Now we have a hierarchy of types as shown in Figure **??**, with unit and 0 as the top and bottom elements, respectively.

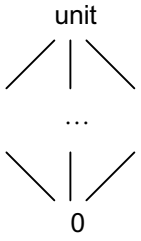

Figure 4: Type hierarchy

6 Objects

We started this lecture by talking about OOP. We can think of classes as recursive object types. Most languages use name-subtyping, so we get only the types we explicitly declare. But we can think of typing rules as rules for deciding when the "extends" in Java, for example, is valid.

```
class C extends D { Car f(); }
```

Java automatically gives width subtyping. Depth subtyping is allowed only when fields have exactly the same type - this is a restriction introduced in version 1.1. In Java 1.0 you could define class D in the code snippet above with a function f returning Vehicle ($\mathsf{Car} \leq \mathsf{Vehicle}$).

While prohibiting depth subtyping on methods is a restriction, disallowing subtyping of member variables is a necessity. Java objects contain possibilities of mutability. In this fragment:

```
class C extends D { Car x; }
```

the variable is actually a reference. To model this, we introduce a type of reference to τ, with the rule:

$$\frac{e : \mathsf{ref}\tau}{!e : \tau}$$

Suppose we allowed depth subtyping on references. Then add to the previous code fragment the following definition:

```
class D { Vehicle X; }
```

Now suppose we execute the following code:

```
D d = new C();
d.x = new Truck();
```

Oops, we just stuck a Truck where we expected a Car! So the fields have to be exactly of the same type:

$$\overline{\mathsf{ref}\tau \leq \mathsf{ref}\tau}$$

Such a subtyping relationship is called *invariant*.