Notes on Subtyping

Prakash Panangaden

23$^{\text{rd}}$ March 2020

Subtyping introduces a new relation between types. We denote it by the symbol $\lhd$. It is two-place relation between types. We will think exclusively in terms of monomorphic types since the interaction between parametric polymorphism and subtyping is subtle.

When we write $A \lhd B$ we mean that A is a subtype of B. This is just notation, what does it mean? It means that *any context that needs a value of type B will be content with a value of type A*. You can think that A-values have every feature that B-values have and possibly more. Subtyping is particularly subtle when it comes to functions; in this case it is fruitful to think in terms of contracts. We will formalize this below but first, some examples.

A *crude* view of subtyping is to think of it in terms of sets. A *type* is a collection of values sharing some common structural feature. A subtype is just a subset. I want to emphasize that this is a very simple-minded and *not very useful* way of thinking about types. Consider the type of mammals. The type of dogs is a subtype of the type of mammals. This is sometimes called the "is-a" relation and one sees so-called ontological diagrams with different collections of things connected by the *is-a* relation: "a Volkswagen is a car", "a car is vehicle", "a vehicle is a machine". This view does not really tell us much of computational interest.

Much more fruitful is to think of types as *contracts.* A declaration like $x : \mathsf{int}$ promises certain structural and computation properties of x. It tells us, for example, that $x + 1$ will make sense. Now these contracts just tell you what properties the value has to have. *It does not rule out the possibility that it has more properties.* Thus, for example, in some languages (but not OCaml) int is a subtype of float; or to use our shiny new notation

$$\mathsf{int} \lhd \mathsf{float}.$$

In these languages if the surrounding context expects a floating point number it is happy to obtain an integer. In such languages, expressions like $2 + 3.14159$ are acceptable. The run-time system automatically converts the integer 2 to 2.0. This is called an *implicit coercion.* It is relatively easy to put this into a language with run-time checking of types (like Scheme or Python) but not so easy when one wants to assign types at compile time. That is why in OCaml there are *explicit functions* to convert an integer to a float. This is called *explicit* *coercion.* Note that the reverse subtyping relation: float $\lhd$ int, does **not** hold in any language that I know[1]. There is no sensible *universal way* to convert a real number to an integer; one can truncate, round, take the ceiling etc.

Subtyping gets more interesting with record types. **Warning: this is not what happens in OCaml**. Consider a type called person which is a record type with fields name of type string, SIN of type integer, age of type integer. Now consider other record types called student and employee. Perhaps all these record types are part of a database system. Both student and employee have all the same fields as person but student has **in addition** the fields, studentid and program. The employee type has all the same fields as person but has, in addition, fields called title and salary. Now say a program or context expects a value of type person. Perhaps it is a method that has its input declared to be of type person. What the method is promising (and this is enforced by the type checker in, for example, Java) is that it will only use the fields that are declared to be in the person record type. Well, all these fields are present in student and in employee so the method should be perfectly happy if it gets a value of type student or a value of type employee. This is exactly what happens in Java when we define the type person as an interface and define classes student and employee that *implement* the interface person. It is equally what happens if we define a class called person and then define classes student and employee that *extend* the class person. In Java (as you know!) there are other things that happen when we define a class as an extension of another class; this is called *inheritance* and will be discussed later. In the next class we will take a close look at inheritance in Java and what happens with the type system.

Let me take this opportunity to scotch a common source of confusion. How can student be a subtype of person when the student record is obviously "bigger". This is just sloppy talk leading to confused thinking. An **individual** student record is bigger than **an individual** person record; the **collection** of **all** student records is smaller than the **collection** of **all** person records.

The above example illustrates that two different types that are not related to each other may be subtypes of the same type. In our notation we have

$$\mathsf{student} \lhd \mathsf{person} \text{ and } \mathsf{employee} \lhd \mathsf{person}$$

but student $\not\lhd$employee and employee $\not\lhd$student. Could it happen that one type is simultaneously a subtype of two other types that are not related to each other? Yes! Consider the example of teaching assistants. In most universities, these are employees who are *required to be* students. We can define a type ta that is a subtype of both employee and student; it is, of course, also a subtype of person. This situation is called a subtyping diamond and is illustrated in Figure 1.

This kind of situation causes problems with inheritance so it is banned in Java: this is why Java has inheritance mixed with pure subtyping. In C++ mul-

[1] If there is a language like that I am much happier not knowing about it.

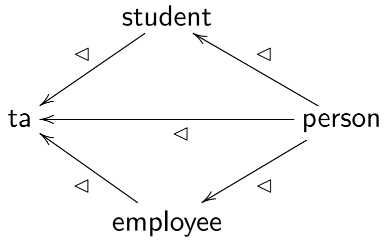

Figure 1: A subtyping diamond

tiple inheritance is allowed and predictably this leads to a mess for all concerned.

Let us start writing some formal typing rules. Suppose that $A \lhd B$. We have the following typing rule

$$\frac{\Gamma \vdash e : A \quad A \lhd B}{\Gamma \vdash e : B}.$$

This says that if you can derive that e has type A (in the context Γ) and if $A \lhd B$ then one can deduce that e has type B. This means that any function (method) expecting an input of type B will be happy to receive e as an input.

Here are some basic properties of subtyping:

$$\frac{}{T \lhd T} \qquad \frac{S \lhd R \quad R \lhd T}{S \lhd T}.$$

The first makes a trivial statement but it needs to be stated. The second says that the subtyping relation is transitive. What about type constructors? The following rule for forming tuples should be intuitive:

$$\frac{S_1 \lhd T_1 \quad S_2 \lhd T_2}{S_1 * S_2 \lhd T_1 * T_2}.$$

Similarly we have a rule for lists

$$\frac{S \lhd T}{S \text{ list} \lhd T \text{ list}.}$$

These rules say that these type constructors *preserve* the subtyping relation. We call such type constructors **covariant**.

Things are more interesting with records. First we consider the case where the field names are exactly the same. We have the following rule:

$$\frac{S_1 \lhd T_1 \quad S_2 \lhd T_2 \quad \ldots \quad S_k \lhd T_k}{\{x_1 : S_1; \ldots ; x_k : S_k\} \lhd \{x_1 : T_1; \ldots ; x_k : T_k\}}$$

This is straightforward and is just like the previous type constructors. It is called *depth subtyping*. It is also a covariant rule: subtyping is preserved. However consider the example we have been looking at above where there is subtyping between record types and the field names are *not* exactly the same. We have the following rule:

$$\frac{k \leq n}{\{x_1 : T_1; \ldots ; x_n : T_n\} \lhd \{x_1 : T_1; \ldots ; x_k : T_k\}}.$$

The relation $\leq$ is just the usual numerical inequality. One can see that the relation looks like it is getting flipped; of course $\leq$ is not subtyping but still it looks like the relation has gotten flipped. Indeed this is what happens; this phenomenon is called **contravariance**.

Now we are ready to discuss function types. In object-oriented programming functions are called *methods* but I will call them functions here.

Here it really pays off to think of a type declaration as a contract. When we declare $f : T_1 \to T_2$ we are saying that "if f is given input of type T_1 it will deliver output of type T_2." Now suppose that I have declared a function f to be of type $T_1 \to T_2$ and I am given an implementation of type $S_1 \to S_2$, what should the subtyping relationships between S_1, T_1 and S_2, T_2 be? Here is the correct rule:

$$\frac{\Gamma \vdash f : S_1 \to S_2 \quad T_1 \lhd S_1 \quad S_2 \lhd T_2}{\Gamma \vdash f : T_1 \to T_2}.$$

or equivalently

$$\frac{T_1 \lhd S_1 \quad S_2 \lhd T_2}{(S_1 \to S_2) \lhd (T_1 \to T_2)}.$$

Notice that this is neither covariant nor contravariant but is of mixed variance. Let us justify this rule using the concept of types as contracts. The first version of the rule is a bit easier to think about. We have a function f implemented in such a way that $f : S_1 \to S_2$ but the declaration said that f should have type $T_1 \to T_2$. Can we use this implementation of f in the context where it was declared? According to the contract $T_1 \to T_2$ we may pass any value of type T_1 and we expect to get back a value of type T_2. The actual implementation expects values of type S_1 but if $T_1 \lhd S_1$ this is fine. The implementation of f outputs values in type S_2, but if $S_2 \lhd T_2$ the calling context is happy as well.

Let us work through an example. Imagine that we have a programming language called G flat in which we have types int, real and $\mathsf{complex}$. Assume that in this language we have the following subtyping relations:

$$\mathsf{int} \lhd \mathsf{real} \lhd \mathsf{complex}.$$

We declare a function f to be of type $\mathsf{real} \to \mathsf{int}$. Can we safely use functions of type $\mathsf{complex} \to \mathsf{real}$ in this context in place of f? The context has signed a contract that says it will pass in real numbers and expects integers back. Well this clearly will not do. The implementation can cope with the real number input since it can handle any complex number. However, it will output a real number and the context was expecting an integer so this will not work. This shows that you cannot have a fully contravariant rule for the arrow type constructor. Can we use functions of type $\mathsf{int} \to \mathsf{int}$? No, we cannot. Can we use functions of type $\mathsf{complex} \to \mathsf{int}$? Yes, we can. Please work through the contract analogy to convince yourself of these last two statements.