Lecture Notes on Subtyping

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 14
October 19, 2004

Subtyping is a fundamental idea in the design of programming languages. It can allow us to write more concise and readable programs by eliminating the need to convert explicitly between elements of types. It can also be used to express more properties of programs. Finally, it is absolutely fundamental in object-oriented programming where the notion of subclass is closely tied to the notion of subtype.

Which subtyping relationships we want to integrate into a language depends on many factors. Besides some theoretical properties we want to satisfy, we also have to consider the pragmatics of type-checking and the operational semantics. In this lecture we are interested in isolating the fundamental principles that must underly various forms of subtyping. We will then see different instances of how these principles can be applied in practice.

We write $\tau \leq \sigma$ to express that τ is a subtype of σ. The fundamental principle of subtyping says:

> *If $\tau \leq \sigma$ then wherever a value of type σ is required, we can use a value of type τ instead.*

This can be refined into two more specific statements, depending on the form of subtyping used.

> **Subset Interpretation.** *If $\tau \leq \sigma$ then every value of type τ is also a value of type σ.*

As an example, consider both empty and non-empty lists as subtypes of the type of lists. This is because an empty list is clearly a list, and a non-empty lists is also a list. One can see that with a subset interpretation of subtyping one can track properties of values.

> **Coercion Interpretation.** *If $\tau \leq \sigma$ then every value of type τ can be converted (coerced) to a value of type σ in a unique way.*

As an example, consider integers as a subtype of floating point numbers. This interpretation is possible because there is a unique way we can convert an integer to a corresponding floating point representation (ignoring questions of size bounds). Therefore, coercive subtyping allows us to omit explicit calls to functions that perform the coercion. However, we have to be careful to guarantee the coerced value is unique, because otherwise the result of a computation may be ambiguous. For example, if we want to say that both integers and floating point numbers are also a subtype of strings, and the coercion yields the printed representation, we violate the uniqueness guarantee. This is because we can coerce `3` to `"3"` since $\mathsf{int} \leq \mathsf{string}$ or `3` to `3.0` and then to `"3.0"` using first $\mathsf{int} \leq \mathsf{float}$ and then $\mathsf{float} \leq \mathsf{string}$. We call a language that satisfies the uniqueness property *coherent*; incoherent languages are poor from the design point of view and can lead to many practical problems. We therefore require the coherence from the start.

Note that both forms of subtyping satisfy the fundamental principle, but that the coercion interpretation is more difficult to achieve than subset interpretation, because we have to verify uniqueness of coercions. Because it is somewhat richer, we concentrate in this lecture on working out a concrete system of subtyping under the coercion interpretation.

First, some general laws that are independent of whether we choose a subset or coercion interpretation. The defining property of subtyping can be expressed in the calculus by the rule of *subsumption*.

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

Secondly, we have reflexivity and transitivity of subtyping.

$$\frac{}{\tau \leq \tau}\ refl \qquad\qquad \frac{\tau_1 \leq \tau_2 \quad \tau_2 \leq \tau_3}{\tau_1 \leq \tau_3}\ trans$$

Let us carefully justify these principles. Under the subset interpretation $\tau \leq \tau$ follows from $A \subseteq A$ for any set of values A. Transitivity follows from the transitivity of the subset relation. Under the coercion interpretation, the identity function coerces from τ to τ for any τ. And we can validate transitivity by composition of functions.

To make the latter considerations concrete, we annotate the subtyping judgment with a coercion and we calculate this coercion in each case. We write $f : \tau \leq \sigma$ is f is a coercion from τ to σ. Note that coercions f are always closed, that is, contain no free variables, so no context is necessary.

$$\frac{}{\lambda x.x : \tau \leq \tau}\,refl \qquad \frac{f : \tau_1 \leq \tau_2 \quad g : \tau_2 \leq \tau_3}{\lambda x.g(f(x)) : \tau_1 \leq \tau_3}\,trans$$

The three laws we have are essentially all the general laws that can be formulated in this manner. Coherence is stated in a way that is similar to a substitution principle.

Coherence. *If $f : \tau \leq \sigma$ and $g : \tau \leq \sigma$ then $f \simeq g : \tau \rightarrow \sigma$.*

Here, extensional equality $f \simeq g : \tau$ is defined inductively on type τ. Note that all coercions should terminate and not have any effects. We show the cases for functions, pairs, and primitive types.

1. $e \simeq e' : \mathsf{int}$ if $e \mapsto^* \mathtt{num}(n)$ and $e' \mapsto^* \mathtt{num}(n')$ and $n = n'$.
2. $e \simeq e' : \tau_1 \times \tau_2$ if $\mathtt{fst}(e) \simeq \mathtt{fst}(e') : \tau_1$ and $\mathtt{snd}(e) \simeq \mathtt{snd}(e') : \tau_2$.
3. $e \simeq e' : \tau_1 \rightarrow \tau_2$ if for any $e_1 \simeq e_1' : \tau_1$ we have $e\,e_1 \simeq e'\,e_1' : \tau_2$.

As a particular example of subtyping, consider $\mathsf{int} \leq \mathsf{float}$. We call the particular coercion $\mathsf{itof} : \mathsf{int} \rightarrow \mathsf{float}$.

$$\frac{}{\mathsf{int} \leq \mathsf{float}} \qquad \frac{}{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}$$

In order to use these functions, consider two versions of the addition operation: one for integers and one for floating point numbers. We avoid overloading here, which is subject of another lecture.

$$\frac{\Gamma \vdash e_1 : \mathsf{int} \quad \Gamma \vdash e_2 : \mathsf{int}}{\Gamma \vdash e_1 + e_2 : \mathsf{int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{float} \quad \Gamma \vdash e_2 : \mathsf{float}}{\Gamma \vdash e_1 +\!. \; e_2 : \mathsf{float}}$$

Now an expression such as $2 + 3.0$ is ill-typed, since the second argument is a floating point number and floating point numbers in general cannot be coerced to integers. However, the expression $2 +\!. \; 3.0$ is well-typed because the first argument 2 can be coerced to the floating point number 2.0 by applying itof. Concretely:

$$\dfrac{\dfrac{\dfrac{}{\vdash 2 : \mathsf{int}} \quad \dfrac{}{\mathsf{int} \leq \mathsf{float}}}{\vdash 2 : \mathsf{float}} \quad \dfrac{}{\vdash 3.0 : \mathsf{float}}}{\vdash 2 +\!. \; 3.0 : \mathsf{float}}$$

So far we have avoided a discussion of the operational semantics, but we can see that (a) under the subset interpretation the operational semantics remains the same as without subtyping, and (b) under the coercion interpretation the operational semantics must apply the coercion functions. That is, we cannot define the operational semantics directly on expressions, because only the subtyping derivation will contain the necessary information on how and where to apply the coercions. We do not formalize the translation from subtyping derivations with coercions to the language without, but we show it by example. In the case above we have

$$\dfrac{\dfrac{\dfrac{}{\vdash 2 : \mathsf{int}} \quad \dfrac{}{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}}{\vdash \mathsf{itof}(2) : \mathsf{float}} \quad \dfrac{}{\vdash 3.0 : \mathsf{float}}}{\vdash \mathsf{itof}(2) +\!. \; 3.0 : \mathsf{float}}$$

The subsumption rule with annotations then looks like

$$\dfrac{\Gamma \vdash e : \tau \quad f : \tau \leq \sigma}{\Gamma \vdash f(e) : \sigma}$$

so we interpret $f : \tau \leq \sigma$ as $f : \tau \rightarrow \sigma$. However, typing derivations are not unique. As written, however, it is not quite right because the source code does not contain f, only the result of type checking. This process is generally called *elaboration* and will occupy us further in the next section. Writing out coercions as we did above, we could have

$$\dfrac{\dfrac{\dfrac{}{\vdash 2 : \mathsf{int}} \quad \dfrac{\dfrac{}{\lambda x.x : \mathsf{int} \leq \mathsf{int}} \quad \dfrac{}{\mathsf{itof} : \mathsf{int} \leq \mathsf{float}}}{\lambda y.\mathsf{itof}((\lambda x.x)(y)) : \mathsf{int} \leq \mathsf{float}}}{\vdash (\lambda y.\mathsf{itof}((\lambda x.x)(y)))(2) : \mathsf{float}} \quad \dfrac{}{\vdash 3.0 : \mathsf{float}}}{\vdash (\lambda y.\mathsf{itof}((\lambda x.x)(y)))(2) +\!. \; 3.0 : \mathsf{float}}$$

This alternative compilation will behave identically to the first one, itof and $\lambda y.\mathsf{itof}((\lambda x.x)(y))$ are observationally equivalent. To see this, apply both sides to a value v. Then the one side yiels $\mathsf{itof}(v)$, the other side

$$(\lambda y.\mathsf{itof}((\lambda x.x)(y)))v \mapsto \mathsf{itof}((\lambda x.x)v) \mapsto \mathsf{itof}(v)$$

The fact that the particular chosen typing derivation does not affect the behavior of the compiled expressions (where coercions are explicit) is the subject of the coherence theorem for a language. This is a more precise expression of the "uniqueness" required in the defining property for coercive subtyping.

At this point we have general laws for typing (subsumption) and subtyping (reflexivity and transitivity). But how does subtyping interact with pairs, functions, and other constructs? We start with pairs. We can coerce a value of type $\tau_1 \times \tau_2$ to a value of type $\sigma_1 \times \sigma_2$ if we can coerce the individual components appropriately. That is:

$$\frac{\tau_1 \leq \sigma_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}$$

With explicit coercions:

$$\frac{f_1 : \tau_1 \leq \sigma_1 \quad f_2 : \tau_2 \leq \sigma_2}{\lambda p.\mathtt{pair}(f_1(\mathtt{fst}(p)), f_2(\mathtt{snd}(p))) : \tau_1 \times \tau_2 \leq \sigma_1 \times \sigma_2}$$

Functions are somewhat trickier. We know that $\mathsf{int} \leq \mathsf{float}$. It should therefore be clear that $\mathsf{int} \rightarrow \mathsf{int} \leq \mathsf{int} \rightarrow \mathsf{float}$, because we can coerce the output of the function of the left to the required output for the function on the right. So

$$\lambda h.\lambda x.\mathsf{itof}(h(x)) : \mathsf{int} \rightarrow \mathsf{int} \leq \mathsf{int} \rightarrow \mathsf{float}$$

Perhaps surprisingly, we have

$$\mathsf{float} \rightarrow \mathsf{int} \leq \mathsf{int} \rightarrow \mathsf{int}$$

because we can obtain a function of the type on the right from a function on the left by coercing the argument:

$$\lambda h.\lambda x.h(\mathsf{itof}(x)) : \mathsf{float} \rightarrow \mathsf{int} \leq \mathsf{int} \rightarrow \mathsf{int}$$

Putting these two ideas together we get

$$\lambda h.\lambda x.\mathsf{itof}(h(\mathsf{itof}(x))) : \mathsf{float} \rightarrow \mathsf{int} \leq \mathsf{int} \rightarrow \mathsf{float}$$

In the general case, we obtain the following rule:

$$\frac{\sigma_1 \leq \tau_1 \quad \tau_2 \leq \sigma_2}{\tau_1 \rightarrow \tau_2 \leq \sigma_1 \rightarrow \sigma_2}$$

With coercion functions:

$$\frac{f_1 : \sigma_1 \leq \tau_1 \quad f_2 : \tau_2 \leq \sigma_2}{\lambda h.\lambda x.f_2(h(f_1(x))) : \tau_1 \to \tau_2 \leq \sigma_1 \to \sigma_2}$$

The fact that the subtyping relationship flips in the left premise is called *contravariance*. We say that function subtyping is contravariant in the argument and covariant in the result. Subtyping of pairs, on the other hand, is covariant in both components.

Mutable reference can be neither covariant nor contravariant. As simple counterexamples, consider the following pieces of code.

The first one assumes that $\tau\ \mathsf{ref} \leq \sigma\ \mathsf{ref}$ if $\sigma \leq \tau$, that is reference subtyping is contravariant.

```
let val r = ref 2.1 (* r : float ref *)
in
   !r
end : int         (* using float ref <: int ref *)
```

Clearly, this is incorrect and violates preservation.

Conversely if we assume subtyping is covariant, that is, $\tau\ \mathsf{ref} \leq \sigma\ \mathsf{ref}$ if $\tau \leq \sigma$, then

```
let val r = ref 3 (* r : int ref *)
in
   r := 2.1;    (* using int ref <: float ref *)
   !r
end : int
```

To avoid these counterexamples we make mutable references *non-variant*.

$$\frac{\tau \leq \sigma \quad \sigma \leq \tau}{\tau\ \mathsf{ref} \leq \sigma\ \mathsf{ref}}$$

More detailed analyses of references are possible. In particular, we can decompose them into "sources" from which we can only read and "sinks" to which we can only write. Sources are covariant and sinks are contravariant. Since we can both read from and write to mutable references, they must be non-variant. We will not develop this formally here.

Note that non-variance of references is an important issue in object-oriented languages. For example, in Java every element of an array acts like a reference and should therefore be non-variant. However, in Java, arrays are co-variant, so run-time checks on types of assigments to arrays or mutable fields are necessary in order to save type preservation. In particular, every time one writes to an array of objects in Java, a dynamic tag-check is required, because arrays are co-variant in the element type. Fortunately, this is possible because in Java and other object-oriented languages there is enough information at run-time to perform this check reasonably efficiently.

There are other type constructors that must be non-variant. For example, if we define (in ML)

```
datatype 'a func = F of 'a -> 'a
```

then the rule

$$\frac{\tau \leq \sigma \quad \sigma \leq \tau}{\tau\ \mathsf{func} \leq \sigma\ \mathsf{func}}$$

is forced by the occurrence of `'a` in both a co-variant and contra-variant position in the argument to the constructor `F`.

Conversely, a type such as

```
datatype 'a singleton = Unit of unit
```

has only a single element, `Unit`, independently of the instantiation of `'a`. Therefore we can pose, for arbitrary τ and σ, that

$$\frac{}{\tau\ \mathsf{singleton} \leq \sigma\ \mathsf{singleton}}$$

without fear of compromising soundness.