Chapter 1

Subtyping

Subtyping is an fundamental concept in programming language theory. It is an important part of the design of an object-oriented language, in particular, in which the relation between a superclass and its subclasses may be seen as a subtyping relation. This chapter develops the theory of subtyping by considering various subtyping relations and discussing the semantics of subtyping.

1.1 Principle of subtyping

The *principle of subtyping* is a principle specifying when a type is a *subtype* of another type. It states that A is a subtype of B if an expression of type A may be used wherever an expression of type B is expected. Formally we write $A \leq B$ if A is a subtype of B, or equivalently, if B is a *supertype* of A.

The principle of subtyping justifies two *subtyping rules*, *i.e.*, inference rules for deducing subtyping relations:

$$\frac{}{A \leq A}\ \mathit{Refl}_{\leq} \qquad \frac{A \leq B \quad B \leq C}{A \leq C}\ \mathit{Trans}_{\leq}$$

The rules $\mathit{Refl}_{\leq}$ and $\mathit{Trans}_{\leq}$ express reflexivity and transitivity of the subtyping relation $\leq$, respectively.

The *rule of subsumption* below is a typing rule which enables us to change the type of an expression to its supertype:

$$\frac{\Gamma \vdash e : A \quad A \leq B}{\Gamma \vdash e : B}\ \mathsf{Sub}$$

It is easy to justify the rule Sub using the principle of subtyping. Suppose $\Gamma \vdash e : A$ and $A \leq B$. Since e has type A (under typing context Γ), the subtyping relation $A \leq B$ allows us to use e wherever an expression of type B is expected, which implies that e is effectively of type B.

There are two kinds of semantics for subtyping: *subset semantics* and *coercion semantics*. Under the subset semantics, $A \leq B$ holds if type A literally constitutes a "subset" of type B. That is, $A \leq B$ holds if a value of type A can also be viewed as a value of type B. The coercion semantics permits $A \leq B$ if there exists a unique method to convert values of type A to values of type B.

As an example, consider three base type nat for natural numbers, int for integers, and float for floating point numbers:

$$\text{base type} \qquad P \quad ::= \quad \mathsf{nat} \mid \mathsf{int} \mid \mathsf{float}$$

If nat and int use the same representation, say, a 32-bit word, a value of type nat also represents an integer of type int. Hence the set of natural numbers of type nat is a subset of the set of integers of type int, which means that $\mathsf{nat} \leq \mathsf{int}$ holds under the subset semantics. If float uses a 64-bit word to represent a floating point number, a value of type int is not a special value of type float because int uses a representation which is incompatible with 64-bit floating point numbers. Thus int is not a subtype of float under the subset semantics, even though integers are a subset of floating point numbers in mathematics. Under the coercion semantics, however, $\mathsf{int} \le \mathsf{float}$ holds if there is a function, *e.g.*, $\mathsf{int2float}$, converting 32-bit integers to 64-bit floating point numbers.

In the next section, we will assume the subset semantics which does not alter the operational semantics and is simpler than the coercion semantics. We will discuss the coercion semantics in detail in Section 1.3

1.2 Subtyping relations

To explain subtyping relations on various types, let us assume two base types nat and int, and a subtyping relation $\mathsf{nat} \le \mathsf{int}$:

$$\begin{array}{lrcl} \mathsf{type} & A & ::= & P \mid A \rightarrow A \mid A \times A \mid A+A \mid \mathsf{ref}\ A \\ \mathsf{base\ type} & P & ::= & \mathsf{nat} \mid \mathsf{int} \end{array}$$

A subtyping relation on two product types tests the relation between corresponding components:

$$\frac{A \le A' \quad B \le B'}{A \times B \le A' \times B'}\ \mathit{Prod}_{\le}$$

For example, $\mathsf{nat} \times \mathsf{nat}$ is a subtype of $\mathsf{int} \times \mathsf{int}$:

$$\frac{\mathsf{nat} \le \mathsf{int} \quad \mathsf{nat} \le \mathsf{int}}{\mathsf{nat} \times \mathsf{nat} \le \mathsf{int} \times \mathsf{int}}\ \mathit{Prod}_{\le}$$

Intuitively a pair of natural numbers can be viewed as a pair of integers because a natural number is a special form of integer. Similarly we can show that a subtyping relation on two sum types tests the relation between corresponding components as follows:

$$\frac{A \le A' \quad B \le B'}{A+B \le A'+B'}\ \mathit{Sum}_{\le}$$

The subtyping rule for function types requires a bit of thinking. Consider two functions $f : A \rightarrow \mathsf{nat}$ and $f' : A \rightarrow \mathsf{int}$. An application of f' to an expression e of type A has type int (under a certain typing context Γ):

$$\Gamma \vdash f'\ e : \mathsf{int}$$

If we replace f' by f, we get an application of type nat, but by the rule of subsumption, the resultant application can be assigned type int as well:

$$\frac{\Gamma \vdash f\ e : \mathsf{nat} \quad \mathsf{nat} \le \mathsf{int}}{\Gamma \vdash f\ e : \mathsf{int}}\ \mathsf{Sub}$$

Therefore, for the purpose of typechecking, it is always safe to use f wherever f' is expected, which implies that $A \rightarrow \mathsf{nat}$ is a subtype of $A \rightarrow \mathsf{int}$. The converse, however, does not hold because there is no assumption of $\mathsf{int} \le \mathsf{nat}$. The result is generalized to the following subtyping rule:

$$\frac{B \le B'}{A \rightarrow B \le A \rightarrow B'}\ \mathit{Fun}'_{\le}$$

The rule $\mathit{Fun}'_{\le}$ says that subtyping on function types is *covariant* in return types in the sense that the premise places the two return types in the same direction as in the conclusion (*i.e.*, left B and right B'). Then, by the rules $\mathit{Prod}_{\le}$ and $\mathit{Sum}_{\le}$, subtyping on product types and sum types is also covariant in both components.

Now consider two functions $g : \mathsf{nat} \rightarrow A$ and $g' : \mathsf{int} \rightarrow A$. Perhaps surprisingly, $\mathsf{nat} \rightarrow A$ is *not* a subtype of $\mathsf{int} \rightarrow A$ whereas $\mathsf{int} \rightarrow A$ is a subtype of $\mathsf{nat} \rightarrow A$. To see why, let us consider an application of g to an expression e of type nat (under a certain typing context Γ):

$$\frac{\Gamma \vdash g : \mathsf{nat} \rightarrow A \quad \Gamma \vdash e : \mathsf{nat}}{\Gamma \vdash g\ e : A}\ \rightarrow\!\mathsf{E}$$

Since e can be assigned type int by the rule of subsumption, replacing g by g' does not change the type of the application:

$$\dfrac{\Gamma \vdash g' : \mathsf{int} \rightarrow A \quad \dfrac{\Gamma \vdash e : \mathsf{nat} \quad \mathsf{nat} \leq \mathsf{int}}{\Gamma \vdash e : \mathsf{int}}\ \mathsf{Sub}}{\Gamma \vdash g'\ e : A}\ \rightarrow\!\mathsf{E}$$

Therefore $\mathsf{int} \rightarrow A$ is a subtype of $\mathsf{nat} \rightarrow A$. To see why the converse does not hold, consider another application of g' to an expression e' of type int:

$$\dfrac{\Gamma \vdash g' : \mathsf{int} \rightarrow A \quad \Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash g'\ e' : A}\ \rightarrow\!\mathsf{E}$$

If we replace g' by g, the resultant application does not even typecheck because e' cannot be assigned type nat:

$$\dfrac{\Gamma \vdash g : \mathsf{nat} \rightarrow A \quad \dfrac{\Gamma \vdash e' : \mathsf{int}}{\Gamma \vdash e' : \mathsf{nat}}\ ???}{\Gamma \vdash g\ e' : A}\ \rightarrow\!\mathsf{E}$$

We generalize the result to the following subtyping rule:

$$\dfrac{A' \leq A}{A \rightarrow B \leq A' \rightarrow B}\ \mathit{Fun}''_{\leq}$$

The rule $\mathit{Fun}''_{\leq}$ says that subtyping on function types is *contravariant* in argument types in the sense that the premise reverses the position of the two argument types from the conclusion (*i.e.*, left A' and right A).

We combine the rules $\mathit{Fun}'_{\leq}$ and $\mathit{Fun}''_{\leq}$ into the following subtyping rule:

$$\dfrac{A' \leq A \quad B \leq B'}{A \rightarrow B \leq A' \rightarrow B'}\ \mathit{Fun}_{\leq}$$

The rule $\mathit{Fun}_{\leq}$ says that subtyping on function types is contravariant in argument types and covariant in return types.

Subtyping on reference types is unusual in that it is neither covariant nor contravariant. Let us figure out the relation between two types A and B when $\mathsf{ref}\ A \leq \mathsf{ref}\ B$ holds:

$$\dfrac{???}{\mathsf{ref}\ A \leq \mathsf{ref}\ B}\ \mathit{Ref}_{\leq}$$

Suppose that an expression e has type $\mathsf{ref}\ A$ and another expression e' type $\mathsf{ref}\ B$. By the principle of subtyping, we should be able to use e wherever e' is used. Since e' has a reference type, there are two ways of using e': dereferencing it and assigning a new value to it.

As an example of the first case, consider a well-typed expression $f\ (!e')$ where f has type $B \rightarrow C$ for some type C. By the assumption $\mathsf{ref}\ A \leq \mathsf{ref}\ B$, expression $f\ (!e)$ is also well-typed. Since f has type $B \rightarrow C$ and $!e$ has type A, the type of $!e$ changes from A to B, which implies $A \leq B$. As an example of the second case, consider a well-typed expression $e' := v$ where v has type B. By the assumption $\mathsf{ref}\ A \leq \mathsf{ref}\ B$, expression $e := v$ is also well-typed. Since v has type B and e has type $\mathsf{ref}\ A$, the type of v changes from B to A, which implies $B \leq A$. These two observations lead to the following subtyping rule for reference types:

$$\dfrac{A \leq B \quad B \leq A}{\mathsf{ref}\ A \leq \mathsf{ref}\ B}\ \mathit{Ref}_{\leq}$$

Thus we say that subtyping on reference types is *non-variant* (*i.e.*, neither covariant nor contravariant).

Another way of looking at the rule $\mathit{Ref}_{\leq}$ is by interpreting $\mathsf{ref}\ A$ as an abbreviation of a function type. We may think of $\mathsf{ref}\ A$ as $? \rightarrow A$ for some unknown type $?$ because dereferencing an expression of type $\mathsf{ref}\ A$ requires no additional argument (hence $? \rightarrow$) and returns an expression of type A (hence $\rightarrow A$). Therefore ref $A \leq$ ref B implies $? \to A \leq ? \to B$, which in turn implies $A \leq B$ by the rule $Fun_{\leq}$. We may also think ref A as $A \to \mathsf{unit}$ because assigning a new value to an expression of type ref A requires a value of type A (hence $A \to$) and returns a unit (hence $\to \mathsf{unit}$). Therefore ref $A \leq$ ref B implies $A \to \mathsf{unit} \leq B \to \mathsf{unit}$, which in turn implies $B \leq A$ by the rule $Fun_{\leq}$.

While we have not investigated array types, subtyping on array types follows the same pattern as subtyping on reference types, since an array, like a reference, allows both read and write operations on it. If we use an array type array A for arrays of elements of type A, we obtain the following subtyping rule:

$$\frac{A \leq B \quad B \leq A}{\mathsf{array}\ A \leq \mathsf{array}\ B}\ Array_{\leq}$$

Interestingly the Java language adopts a subtyping rule in which subtyping on array rules is covariant in element types:

$$\frac{A \leq B}{\mathsf{array}\ A \leq \mathsf{array}\ B}\ Array_{\leq}'$$

While it is controversial whether the rule $Array_{\leq}'$ is a flaw in the design of the Java language, using the rule $Array_{\leq}'$ for subtyping on array types incurs a runtime overhead which would otherwise be unnecessary. To be specific, lack of the condition $B \leq A$ in the premise implies that whenever a value of type B is written to an array of type array A, the runtime system must verify a subtyping relation $B \leq A$, which incurs a runtime overhead of dynamic tag-checks.

1.3 Coercion semantics for subtyping

Under the coercion semantics, a subtyping relation $A \leq B$ holds if there exists a unique method to convert values of type A to values of type B. As a witness to the existence of such a method, we usually use a λ-abstraction, called a *coercion function*, of type $A \to B$. We use a *coercion subtyping judgment*

$$A \leq B \Rightarrow f$$

to mean that $A \leq B$ holds under the coercion semantics with a coercion function f of type $A \to B$. For example, a judgment $\mathsf{int} \leq \mathsf{float} \Rightarrow \mathsf{int2float}$ holds if a coercion function int2float converts integers of type int to floating point number of type float.

The subtyping rules for the coercion subtyping judgment are given as follows:

$$\frac{}{A \leq A \Rightarrow \lambda x\!:\!A.\, x}\ Refl^{\mathsf{C}}_{\leq} \qquad \frac{A \leq B \Rightarrow f \quad B \leq C \Rightarrow g}{A \leq C \Rightarrow \lambda x\!:\!A.\, g\ (f\ x)}\ Trans^{\mathsf{C}}_{\leq}$$

$$\frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A \times B \leq A' \times B' \Rightarrow \lambda x\!:\!A \times B.\, (f\ (\mathsf{fst}\ x), g\ (\mathsf{snd}\ x))}\ Prod^{\mathsf{C}}_{\leq}$$

$$\frac{A \leq A' \Rightarrow f \quad B \leq B' \Rightarrow g}{A{+}B \leq A'{+}B' \Rightarrow \lambda x\!:\!A{+}B.\, \mathsf{case}\ x\ \mathsf{of}\ \mathsf{inl}\ y_1.\, \mathsf{inl}_{B'}\ (f\ y_1) \mid \mathsf{inr}\ y_2.\, \mathsf{inr}_{A'}\ (g\ y_2)}\ Sum^{\mathsf{C}}_{\leq}$$

$$\frac{A' \leq A \Rightarrow f \quad B \leq B' \Rightarrow g}{A \to B \leq A' \to B' \Rightarrow \lambda h\!:\!A \to B.\, \lambda x\!:\!A'.\, g\ (h\ (f\ x))}\ Fun^{\mathsf{C}}_{\leq}$$

Unlike the subset semantics which does not change the operational semantics, the coercion semantics affects the way that expressions are evaluated. Suppose that we are evaluating a well-typed expression e with the following typing derivation which uses a coercion subtyping judgment:

$$\frac{\Gamma \vdash e : A \quad A \leq B \Rightarrow f}{\Gamma \vdash e : B}\ \mathsf{Sub}^{\mathsf{C}}$$

Since the rule $\mathsf{Sub}^{\mathsf{C}}$ tacitly promotes the type of e from A to B, we do not have to insert an explicit call to the coercion function f to make e typecheck. The result of evaluating e, however, is correct only if an explicit call to f is made after evaluating e. For example, if e is an argument to another function g of type $B \rightarrow C$ (for some type C), $g\ e$ certainly typechecks but may go wrong at runtime. Therefore the type system inserts an explicit call to a coercion function each time the rule $\mathsf{Sub}^{\mathsf{C}}$ is used. In the above case, the type system replaces e by $f\ e$ after typechecking e using the rule $\mathsf{Sub}^{\mathsf{C}}$.

A potential problem with the coercion semantics is that the same subtyping relation may have several coercion functions which all have the same type but exhibit different behavior. As an example, consider the following subtyping relations:

$$\mathsf{int} \leq \mathsf{float} \Rightarrow \mathsf{int2float} \quad \mathsf{int} \leq \mathsf{string} \Rightarrow \mathsf{int2string} \quad \mathsf{float} \leq \mathsf{string} \Rightarrow \mathsf{float2string}$$

$\mathsf{int} \leq \mathsf{string}$ and $\mathsf{float} \leq \mathsf{string}$ imply that integers and floating point numbers are automatically converted to strings in all contexts expecting strings. Then the same subtyping judgment $\mathsf{int} \leq \mathsf{string}$ has two coercions functions: $\mathsf{int2string}$ and $\lambda x\!:\!\mathsf{int}.\ \mathsf{float2string}\ (\mathsf{int2float}\ x)$. The two coercion functions, however, behave differently. For example, the first converts 0 to `"0"` whereas the second converts 0 to `"0.0"`.

We say that a type system for subtypes is *coherent* if all coercion functions for the same subtyping relation exhibit the same behavior. In the example above, we can recover coherence by specifying that $\mathsf{float2string}$ converts 0.0 to `"0"`, instead of `"0.0"`, and similarly for all other forms of floating point numbers. We do not further discuss coherence which is difficult to prove for more complex type systems.

Algorithmic subtyping