# Part XI

# Subtyping and Inheritance

# Chapter 26

# Subtyping

A *subtype* relation is a pre-order[1] on types that validates the *subsumption principle*: if $\sigma$ is a subtype of $\tau$, then a value of type $\sigma$ may be provided whenever a value of type $\tau$ is required. This means that a value of the subtype should "act like" a value of the supertype when used in supertype contexts.

## 26.1 MinML With Subtyping

We will consider two extensions of MinML with subtyping. The first, *MinML with implicit subtyping*, is obtained by adding the following rule of *implicit subsumption* to the typing rules of MinML:

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

With implicit subtyping the typing relation is no longer syntax-directed, since the subsumption rule may be applied to any expression $e$, without regard to its form.

The second, called *MinML with explicit subtyping*, is obtained by adding to the syntax by adding an explicit *cast* expression, $(\tau)\, e$, with the following typing rule:

$$\frac{\Gamma \vdash e : \sigma \quad \sigma <: \tau}{\Gamma \vdash (\tau)\, e : \tau}$$

The typing rules remain syntax-directed, but all uses of subtyping must be explicitly indicated.

[1]A pre-order is a reflexive and transitive binary relation.

We will refer to either variation as MinML$_{<:}$ when the distinction does not matter. When it does, the implicit version is designated MinML$^{i}_{<:}$, the implicit MinML$^{e}_{<:}$.

To obtain a complete instance of MinML$_{<:}$ we must specify the subtype relation. This is achieved by giving a set of *subtyping axioms*, which determine the primitive subtype relationships, and a set of *variance rules*, which determine how type constructors interact with subtyping. To ensure that the subtype relation is a pre-order, we tacitly include the following rules of reflexivity and transitivity:

$$\frac{}{\tau <: \tau} \qquad \frac{\rho <: \sigma \quad \sigma <: \tau}{\rho <: \tau}$$

Note that pure MinML is obtained as an instance of MinML$^{i}_{<:}$ by giving no subtyping rules beyond these two, so that $\sigma <: \tau$ iff $\sigma = \tau$.

The dynamic semantics of an instance of MinML$_{<:}$ must be careful to take account of subtyping. In the case of implicit subsumption the dynamic semantics must be defined so that the primitive operations of a supertype apply equally well to a value of any subtype. In the case of explicit subsumption we need only ensure that there be a means of *casting* a value of the subtype into a corresponding value of the supertype.

The type safety of MinML$_{<:}$, in either formulation, is assured, provided that the following *subtyping safety conditions* are met:

- For MinML$^{e}_{<:}$, if $\sigma <: \tau$, then casting a value of the subtype $\sigma$ to the supertype $\tau$ must yield a value of type $\tau$.

- For MinML$^{i}_{<:}$, the dynamic semantics must ensure that the value of each primitive operation is defined for closed values of *any subtype* of the expected type of its arguments.

Under these conditions we may prove the Progress and Preservation Theorems for either variant of MinML$_{<:}$.

**Theorem 83 (Preservation)**
*For either variant of* MinML$_{<:}$*, under the assumption that the subtyping safety conditions hold, if* $e : \tau$ *and* $e \mapsto e'$*, then* $e' : \tau$*.*

**Proof:** By induction on the dynamic semantics, appealing to the casting condition in the case of the explicit subsumption rule of MinML$^{e}_{<:}$. ■

**Theorem 84 (Progress)**
*For either variant of* MinML<:, *under the assumption that the subtyping safety conditions hold, if* $e : \tau$, *then either* $e$ *is a value or there exists* $e'$ *such that* $e \mapsto e'$.

**Proof:** By induction on typing, appealing to the subtyping condition on primitive operations in the case of primitive instruction steps. ■

## 26.2 Varieties of Subtyping

In this section we will explore several different forms of subtyping in the context of extensions of MinML. To simplify the presentation of the examples, we tacitly assume that the dynamic semantics of casts is defined so that $(\tau)\,v \mapsto v$, unless otherwise specified.

### 26.2.1 Arithmetic Subtyping

In informal mathematics we tacitly treat integers as real numbers, even though $\mathbb{Z} \not\subseteq \mathbb{R}$. This is justified by the observation that there is an injection $\iota : \mathbb{Z} \hookrightarrow \mathbb{R}$ that assigns a canonical representation of an integer as a real number. This injection preserves the ordering, and commutes with the arithmetic operations in the sense that $\iota(m + n) = \iota(m) + \iota(n)$, where $m$ and $n$ are integers, and the relevant addition operation is determined by the types of its arguments.

In most cases the real numbers are (crudely) approximated by floating point numbers. Let us therefore consider an extension of MinML with an additional base type, `float`, of floating point numbers. It is not necessary to be very specific about this extension, except to say that we enrich the language with floating point constants and arithmetic operations. We will designate the floating point operations using a decimal point, writing `+.` for floating point addition, and so forth.[2]

By analogy with mathematical practice, we will consider taking the type `int` to be a subtype of `float`. The analogy is inexact, because of the limitations of computer arithmetic, but it is, nevertheless, informative to consider it.

To ensure the safety of explicit subsumption we must define how to cast an integer to a floating point number, written (`float`) $n$. We simply postu-

[2]This convention is borrowed from O'Caml.

late that this is possible, writing $n.0$ for the floating point representation of the integer $n$, and noting that $n.0$ has type `float`.[3]

To ensure the safety of implicit subsumption we must ensure that the floating point arithmetic operations are well-defined for integer arguments. For example, we must ensure that an expression such as `+.`$(3, 4)$ has a well-defined value as a floating point number. To achieve this, we simply require that floating point operations implicitly convert any integer arguments to floating point before performing the operation. In the foregoing example evaluation proceeds as follows:

$$\texttt{+.}(3,4) \mapsto \texttt{+.}(3.0,4.0) \mapsto 7.0.$$

This strategy requires that the floating point operations detect the presence of integer arguments, and that it convert any such arguments to floating point before carrying out the operation. We will have more to say about this inefficiency in Section 26.4 below.

### 26.2.2 Function Subtyping

Suppose that `int <: float`. What subtyping relationships, if any, should hold among the following four types?

1. $\texttt{int}{\rightarrow}\texttt{int}$
2. $\texttt{int}{\rightarrow}\texttt{float}$
3. $\texttt{float}{\rightarrow}\texttt{int}$
4. $\texttt{float}{\rightarrow}\texttt{float}$

To determine the answer, keep in mind the subsumption principle, which says that a value of the subtype should be usable in a supertype context.

Suppose $f : \texttt{int}{\rightarrow}\texttt{int}$. If we apply $f$ to $x : \texttt{int}$, the result has type `int`, and hence, by the arithmetic subtyping axiom, has type `float`. This suggests that

$$\texttt{int}{\rightarrow}\texttt{int} \texttt{ <: } \texttt{int}{\rightarrow}\texttt{float}$$

is a valid subtype relationship. By similar reasoning, we may derive that

$$\texttt{float}{\rightarrow}\texttt{int} \texttt{ <: } \texttt{float}{\rightarrow}\texttt{float}$$

[3] We may handle the limitations of precision by allowing for a cast operation to fail in the case of overflow. We will ignore overflow here, for the sake of simplicity.

is also valid.

Now suppose that $f$ : $\texttt{float}\rightarrow\texttt{int}$. If $x$ : $\texttt{int}$, then $x$ : $\texttt{float}$ by subsumption, and hence we may apply $f$ to $x$ to obtain a result of type $\texttt{int}$. This suggests that

$$\texttt{float}\rightarrow\texttt{int} \mathrel{\texttt{<:}} \texttt{int}\rightarrow\texttt{int}$$

is a valid subtype relationship. Since $\texttt{int}\rightarrow\texttt{int} \mathrel{\texttt{<:}} \texttt{int}\rightarrow\texttt{float}$, it follows that

$$\texttt{float}\rightarrow\texttt{int} \mathrel{\texttt{<:}} \texttt{int}\rightarrow\texttt{float}$$

is also valid.

Subtyping rules that specify how a type constructor interacts with subtyping are called *variance* principles. If a type constructor *preserves* subtyping in a given argument position, it is said to be *covariant* in that position. If, instead, it *inverts* subtyping in a given position it is said to be *contravariant* in that position. The discussion above suggests that the function space constructor is covariant in the range position and contravariant in the domain position. This is expressed by the following rule:

$$\frac{\tau_1 \mathrel{\texttt{<:}} \sigma_1 \quad \sigma_2 \mathrel{\texttt{<:}} \tau_2}{\sigma_1\rightarrow\sigma_2 \mathrel{\texttt{<:}} \tau_1\rightarrow\tau_2}$$

Note well the inversion of subtyping in the domain, where the function constructor is contravariant, and the preservation of subtyping in the range, where the function constructor is covariant.

To ensure safety in the explicit case, we define the dynamic semantics of a cast operation by the following rule:

$$\frac{}{(\tau_1\rightarrow\tau_2)\, v \mapsto \texttt{fn}\, x\texttt{:}\tau_1\, \texttt{in}\, (\tau_2)\, v\texttt{(}(\sigma_1)\, x\texttt{)}\, \texttt{end}}$$

Here $v$ has type $\sigma_1\rightarrow\sigma_2$, $\tau_1 \mathrel{\texttt{<:}} \sigma_1$, and $\sigma_2 \mathrel{\texttt{<:}} \tau_2$. The argument is cast to the domain type of the function prior to the call, and its result is cast to the intended type of the application.

To ensure safety in the implicit case, we must ensure that the primitive operation of function application behaves correctly on a function of a subtype of the "expected" type. This amounts to ensuring that a function can be called with an argument of, and yields a result of, a subtype of the intended type. One way is to adopt a semantics of procedure call that is independent of the types of the arguments and results. Another is to introduce explicit run-time checks similar to those suggested for floating point arithmetic to ensure that calling conventions for different types can be met.

### 26.2.3 Product and Record Subtyping

In Chapter 19 we considered an extension of MinML with product types. In this section we'll consider equipping this extension with subtyping. We will work with $n$-ary products of the form $\tau_1 \texttt{*} \cdots \texttt{*} \tau_n$ and with $n$-ary records of the form $\{l_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,} l_n \texttt{:} \tau_n\}$. The tuple types have as elements $n$-tuples of the form $\texttt{<} e_1 \texttt{,} \ldots \texttt{,} e_n \texttt{>}$ whose $i$th component is accessed by projection, $e\,\texttt{.}\,i$. Similarly, record types have as elements records of the form $\{l_1 \texttt{:} e_1 \texttt{,} \ldots \texttt{,} l_n \texttt{:} e_n\}$ whose $l$th component is accessed by field selection, $e\,\texttt{.}\,l$.

Using the subsumption principle as a guide, it is natural to consider a tuple type to be a subtype of any of its prefixes:

$$\frac{m > n}{\tau_1 \texttt{*} \cdots \texttt{*} \tau_m \;\texttt{<:}\; \tau_1 \texttt{*} \cdots \texttt{*} \tau_n}$$

Given a value of type $\tau_1 \texttt{*} \cdots \texttt{*} \tau_n$, we can access its $i$th component, for any $1 \leq i \leq n$. If $m > n$, then we can equally well access the $i$th component of an $m$-tuple of type $\tau_1 \texttt{*} \cdots \texttt{*} \tau_m$, obtaining the same result. This is called *width subtyping* for tuples.

For records it is natural to consider a record type to be a subtype of any record type with any subset of the fields of the subtype. This may be written as follows:

$$\frac{m > n}{\{l_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,} l_m \texttt{:} \tau_m\} \;\texttt{<:}\; \{l_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,} l_n \texttt{:} \tau_n\}}$$

Bear in mind that the ordering of fields in a record type is immaterial, so this rule allows us to neglect any subset of the fields when passing to a supertype. This is called *width subtyping* for records. The justification for width subtyping is that record components are accessed by label, rather than position, and hence the projection from a supertype value will apply equally well to the subtype.

What variance principles apply to tuples and records? Applying the principle of subsumption, it is easy to see that tuples and records may be regarded as covariant in all their components. That is,

$$\frac{\forall 1 \leq i \leq n \; \sigma_i \;\texttt{<:}\; \tau_i}{\sigma_1 \texttt{*} \cdots \texttt{*} \sigma_n \;\texttt{<:}\; \tau_1 \texttt{*} \cdots \texttt{*} \tau_n}$$

and

$$\frac{\forall 1 \leq i \leq n \; \sigma_i \;\texttt{<:}\; \tau_i}{\{l_1 \texttt{:} \sigma_1 \texttt{,} \ldots \texttt{,} l_n \texttt{:} \sigma_n\} \;\texttt{<:}\; \{l_1 \texttt{:} \tau_1 \texttt{,} \ldots \texttt{,} l_n \texttt{:} \tau_n\}}.$$

These are called *depth subtyping* rules for tuples and records, respectively.

To ensure safety for explicit subsumption we must define the meaning of casting from a sub- to a super-type. The two forms of casting corresponding to width and depth subtyping may be consolidated into one, as follows:

$$\frac{m \geq n}{(\tau_1 \texttt{*} \cdots \texttt{*} \tau_n)\,\texttt{<}v_1\texttt{,}\ \ldots\texttt{,}v_m\texttt{>} \mapsto \texttt{<}(\tau_1)\,v_1\texttt{,}\ \ldots\texttt{,}(\tau_n)\,v_n\texttt{>}.}$$

An analogous rule defines the semantics of casting for record types.

To ensure safety for implicit subsumption we must ensure that projection is well-defined on a subtype value. In the case of tuples this means that the operation of accessing the $i$th component from a tuple must be insensitive to the size of the tuple, beyond the basic requirement that it have size at least $i$. This can be expressed schematically as follows:

$$\texttt{<}v_1\texttt{,}\ \ldots\texttt{,}v_i\texttt{,}\ldots\texttt{>.}i \mapsto v_i.$$

The ellision indicates that fields beyond the $i$th are not relevant to the operation. Similarly, for records we postulate that selection of the $l$th field is insensitive to the presence of any other fields:

$$\texttt{\{}l\texttt{:}v\texttt{,}\ldots\texttt{\}.}l \mapsto v.$$

The ellision expresses the independence of field selection from any "extra" fields.

### 26.2.4 Reference Subtyping

Finally, let us consider the reference types of Chapter 14. What should be the variance rule for reference types? Suppose that $r$ has type $\sigma\,\texttt{ref}$. We can do one of two things with $r$:

1. Retrieve its contents as a value of type $\sigma$.

2. Replace its contents with a value of type $\sigma$.

If $\sigma$ `<:` $\tau$, then retrieving the contents of $r$ yields a value of type $\tau$, by subsumption. This suggests that references are covariant:

$$\frac{\sigma \texttt{ <: } \tau}{\sigma\,\texttt{ref} \overset{?}{\texttt{<:}} \tau\,\texttt{ref}.}$$

On the other hand, if $\tau$ `<:` $\sigma$, then we may store a value of type $\tau$ into $r$. This suggests that references are contravariant:

$$\frac{\tau \mathrel{\texttt{<:}} \sigma}{\sigma\,\texttt{ref} \overset{?}{\mathrel{\texttt{<:}}} \tau\,\texttt{ref}}.$$

Given that we may perform either operation on a reference cell, we must insist that reference types are *invariant*:

$$\frac{\sigma \mathrel{\texttt{<:}} \tau \quad \tau \mathrel{\texttt{<:}} \sigma}{\sigma\,\texttt{ref} \mathrel{\texttt{<:}} \tau\,\texttt{ref}}.$$

The premise of the rule is often strengthened to the requirement that $\sigma$ and $\tau$ be equal:

$$\frac{\sigma = \tau}{\sigma\,\texttt{ref} \mathrel{\texttt{<:}} \tau\,\texttt{ref}}$$

since there are seldom situations where distinct types are mutual subtypes.

A similar analysis may be applied to any mutable data structure. For example, *immutable* sequences may be safely taken to be covariant, but *mutable* sequences (arrays) must be taken to be invariant, lest safety be compromised.

## 26.3 Type Checking With Subtyping

Type checking for MinML<:, in either variant, clearly requires an algorithm for deciding subtyping: given $\sigma$ and $\tau$, determine whether or not $\sigma$ `<:` $\tau$. The difficulty of deciding type checking is dependent on the specific rules under consideration. In this section we will discuss type checking for MinML<:, under the assumption that we can check the subtype relation.

Consider first the explicit variant of MinML<:. Since the typing rules are syntax directed, we can proceed as for MinML, with one additional case to consider. To check whether $(\sigma)\,e$ has type $\tau$, we must check two things:

1. Whether $e$ has type $\sigma$.
2. Whether $\sigma$ `<:` $\tau$.

The former is handled by a recursive call to the type checker, the latter by a call to the subtype checker, which we assume given.

This discussion glosses over an important point. Even in pure MinML it is not possible to determine directly whether or not $\Gamma \vdash e : \tau$. For suppose that $e$ is an application $e_1\texttt{(}e_2\texttt{)}$. To check whether $\Gamma \vdash e : \tau$, we must find the domain type of the function, $e_1$, against which we must check the type of the argument, $e_2$. To do this we define a *type synthesis* function that determines the unique (if it exists) type $\tau$ of an expression $e$ in a context $\Gamma$, written $\Gamma \vdash e \Rightarrow \tau$. To check whether $e$ has type $\tau$, we synthesize the unique type for $e$ and check that it is $\tau$.

This methodology applies directly to $\mathsf{MinML}^{e}_{<:}$ by using the following rule to synthesize a type for a cast:

$$\frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma \mathrel{\texttt{<:}} \tau}{\Gamma \vdash (\tau)\, e \Rightarrow \tau}$$

Extending this method to $\mathsf{MinML}^{i}_{<:}$ is a bit harder, because expressions no longer have unique types! The rule of subsumption allows us to weaken the type of an expression at will, yielding many different types for the same expression. A standard approach is define a type synthesis function that determines the *principal* type, rather than the *unique* type, of an expression in a given context. The principal type of an expression $e$ in context $\Gamma$ is the *least* type (in the subtyping pre-order) for $e$ in $\Gamma$. Not every subtype system admits principal types. But we usually strive to ensure that this is the case whenever possible in order to employ this simple type checking method.

The rules synthesizing principal types for expressions of $\mathsf{MinML}^{i}_{<:}$ are as follows:

$$\frac{(\Gamma(x) = \tau)}{\Gamma \vdash x \Rightarrow \tau} \qquad \frac{}{\Gamma \vdash n \Rightarrow \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{true} \Rightarrow \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} \Rightarrow \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 \Rightarrow \sigma_1 \quad \sigma_1 \mathrel{\texttt{<:}} \tau_1 \quad \cdots \quad \Gamma \vdash e_n \Rightarrow \sigma_n \quad \sigma_n \mathrel{\texttt{<:}} \tau_n}{\Gamma \vdash o(e_1, \ldots, e_n) \Rightarrow \tau}$$

where $o$ is an $n$-ary primitive operation with arguments of type $\tau_1, \ldots, \tau_n$, and result type $\tau$. We use subsumption to ensure that the argument types are subtypes of the required types.

$$\frac{\Gamma \vdash e \Rightarrow \sigma \quad \sigma \mathrel{\texttt{<:}} \texttt{bool} \quad \Gamma \vdash e_1 \Rightarrow \tau_1 \quad \tau_1 \mathrel{\texttt{<:}} \tau \quad \Gamma \vdash e_2 \Rightarrow \tau_2 \quad \tau_2 \mathrel{\texttt{<:}} \tau}{\Gamma \vdash \texttt{if}\, e\, \texttt{then}\, e_1\, \texttt{else}\, e_2\, \texttt{fi} \Rightarrow \tau}$$

We use subsumption to ensure that the type of the test is a subtype of `bool`. Moreover, we rely on explicit specification of the type of the two clauses of the conditional.[4]

$$\frac{\Gamma[f{:}\tau_1 \to \tau_2][x{:}\tau_1] \vdash e \Rightarrow \tau_2}{\Gamma \vdash \texttt{fun}\, f\, \texttt{(}x\texttt{:}\tau_1\texttt{)}\texttt{:}\tau_2\, \texttt{is}\, e\, \texttt{end} \Rightarrow \tau_1 \to \tau_2}$$

[4]This may be avoided by requiring that the subtype relation have least upper bounds "whenever necessary"; we will not pursue this topic here.

$$\frac{\Gamma \vdash e_1 \Rightarrow \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 \Rightarrow \sigma_2 \quad \sigma_2 \mathrel{\texttt{<:}} \tau_2}{\Gamma \vdash e_1\texttt{(}e_2\texttt{)} \Rightarrow \tau}$$

We use subsumption to check that the argument type is a subtype of the domain type of the function.

**Theorem 85**

1. *If* $\Gamma \vdash e \Rightarrow \sigma$*, then* $\Gamma \vdash e : \sigma$.
2. *If* $\Gamma \vdash e : \tau$*, then there exists* $\sigma$ *such that* $\Gamma \vdash e \Rightarrow \sigma$ *and* $\sigma \mathrel{\texttt{<:}} \tau$.

**Proof:**

1. By a straightforward induction on the definition of the type synthesis relation.
2. By induction on the typing relation.

■

## 26.4 Implementation of Subtyping

### 26.4.1 Coercions

The dynamic semantics of subtyping sketched above suffices to ensure type safety, but is in most cases rather impractical. Specifically,

- Arithmetic subtyping relies on run-time type recognition and conversion.
- Tuple projection depends on the insensitivity of projection to the existence of components after the point of projection.
- Record field selection depends on being able to identify the $l$th field in a record with numerous fields.
- Function subtyping may require run-time checks and conversions to match up calling conventions.

These costs are significant. Fortunately they can be avoided by taking a slightly different approach to the implementation of subtyping. Consider, for example, arithmetic subtyping. In order for a mixed-mode expression such as $\texttt{+.}(3, 4)$ to be well-formed, we must use subsumption to weaken the types of 3 and 4 from `int` to `float`. This means that type conversions are required exactly insofar as subsumption is used during type checking — a use of subsumption corresponds to a type conversion.

Since the subsumption rule is part of the static semantics, we can insert the appropriate conversions during type checking, and omit entirely the need to check for mixed-mode expressions during execution. This is called a *coercion interpretation* of subsumption. It is expressed formally by augmenting each subtype relation $\sigma \mathrel{\texttt{<:}} \tau$ with a function value $v$ of type $\sigma{\rightarrow}\tau$ (in pure MinML) that *coerces* values of type $\sigma$ to values of type $\tau$. The augmented subtype relation is written $\sigma \mathrel{\texttt{<:}} \tau \leadsto v$.

Here are the rules for arithmetic subtyping augmented with coercions:

$$\frac{}{\tau \mathrel{\texttt{<:}} \tau \leadsto \texttt{id}_\tau} \qquad \frac{\rho \mathrel{\texttt{<:}} \sigma \leadsto v \quad \sigma \mathrel{\texttt{<:}} \tau \leadsto v'}{\rho \mathrel{\texttt{<:}} \tau \leadsto v\texttt{;}v'}$$

$$\frac{}{\texttt{int} \mathrel{\texttt{<:}} \texttt{float} \leadsto \texttt{to\_float}} \qquad \frac{\tau_1 \mathrel{\texttt{<:}} \sigma_1 \leadsto v_1 \quad \sigma_2 \mathrel{\texttt{<:}} \tau_2 \leadsto v_2}{\sigma_1{\rightarrow}\sigma_2 \mathrel{\texttt{<:}} \tau_1{\rightarrow}\tau_2 \leadsto v_1{\rightarrow}v_2}$$

These rules make use of the following auxiliary functions:

1. Primitive conversion: `to_float`.
2. Identity: $\texttt{id}_\tau = \texttt{fn}\, x\texttt{:}\tau\, \texttt{in}\, x\, \texttt{end}$.
3. Composition: $v\texttt{;}v' = \texttt{fn}\, x\texttt{:}\tau\, \texttt{in}\, v'\texttt{(}v\texttt{(}x\texttt{))}\, \texttt{end}$.
4. Functions: $v_1{\rightarrow}v_2 =$
   $\texttt{fn}\, f\texttt{:}\sigma_1{\rightarrow}\sigma_2\, \texttt{in}\, \texttt{fn}\, x\texttt{:}\tau_1\, \texttt{in}\, v_2\texttt{(}f\texttt{(}v_1\texttt{(}x\texttt{)))}\, \texttt{end}\, \texttt{end}$.

The coercion interpretation is type correct. Moreover, there is at most one coercion between any two types:

**Theorem 86**

1. *If $\sigma \mathrel{\texttt{<:}} \tau \leadsto v$, then $\vdash^- v : \sigma{\rightarrow}\tau$.*
2. *If $\sigma \mathrel{\texttt{<:}} \tau \leadsto v_1$ and $\sigma \mathrel{\texttt{<:}} \tau \leadsto v_2$, then $\vdash^- v_1 \cong v_2 : \sigma{\rightarrow}\tau$.*

**Proof:**

1. By a simple induction on the rules defining the augmented subtyping relation.
2. Follows from these equations:
   (a) Composition is associative with `id` as left- and right-unit element.

(b) $\mathtt{id}\rightarrow\mathtt{id} \cong \mathtt{id}$.

(c) $(v_1\rightarrow v_2)\,\mathtt{;}\,(v_1'\rightarrow v_2') \cong (v_1'\,\mathtt{;}\,v_1)\rightarrow(v_2\,\mathtt{;}\,v_2')$.

■

The type checking relation is augmented with a translation from $\mathsf{MinML}^i_{<:}$ to pure MinML that eliminates uses of subsumption by introducing coercions:

$$\frac{\Gamma \vdash e : \sigma \leadsto e' \quad \sigma \mathrel{\mathtt{<:}} \tau \leadsto v}{\Gamma \vdash e : \tau \leadsto v\mathtt{(}e'\mathtt{)}}$$

The remaining rules simply commute with the translation. For example, the rule for function application becomes

$$\frac{\Gamma \vdash e_1 : \tau_2\rightarrow\tau \leadsto e_1' \quad \Gamma \vdash e_2 : \tau_2 \leadsto e_2'}{\Gamma \vdash e_1\mathtt{(}e_2\mathtt{)} : \tau \leadsto e_1'\mathtt{(}e_2'\mathtt{)}}$$

**Theorem 87**

1. *If $\Gamma \vdash e : \tau \leadsto e'$, then $\Gamma \vdash e' : \tau$ in pure MinML.*

2. *If $\Gamma \vdash e : \tau \leadsto e_1$ and $\Gamma \vdash e : \tau \leadsto e_2$, then $\Gamma \vdash e_1 \cong e_2 : \tau$ in pure MinML.*

3. *If $e : \mathtt{int} \leadsto e'$ is a complete program, then $e \Downarrow n$ iff $e' \Downarrow n$.*

The coercion interpretation also applies to record subtyping. Here the problem is how to implement field selection efficiently in the presence of subsumption. Observe that in the absence of subtyping the type of a record value reveals the *exact* set of fields of a record (and their types). We can therefore implement selection efficiently by ordering the fields in some canonical manner (say, alphabetically), and compiling field selection as a projection from an offset determined statically by the field's label.

In the presence of record subtyping this simple technique breaks down, because the type no longer reveals the fields of a record, not their types. For example, every expression of record type has the record type $\{\}$ with no fields whatsoever! This makes it difficult to predict statically the position of the field labelled $l$ in a record. However, we may restore this important property by using coercions. Whenever the type of a record is weakened using subsumption, insert a function that creates a new record that exactly matches the supertype. Then use the efficient record field selection method just described.

Here, then, are the augmented rules for width and depth subtyping for records:

$$\frac{m > n}{\{l_1 \mathtt{:} \tau_1 \mathtt{,} \ldots \mathtt{,} l_m \mathtt{:} \tau_m\} \mathrel{\mathtt{<:}} \{l_1 \mathtt{:} \tau_1 \mathtt{,} \ldots \mathtt{,} l_n \mathtt{:} \tau_n\} \rightsquigarrow \mathtt{drop}_{m,n,l,\tau}}$$

$$\frac{\sigma_1 \mathrel{\mathtt{<:}} \tau_1 \rightsquigarrow v_1 \quad \ldots \quad \sigma_n \mathrel{\mathtt{<:}} \tau_n \rightsquigarrow v_n}{\{l_1 \mathtt{:} \sigma_1 \mathtt{,} \ldots \mathtt{,} l_n \mathtt{:} \sigma_n\} \mathrel{\mathtt{<:}} \{l_1 \mathtt{:} \tau_1 \mathtt{,} \ldots \mathtt{,} l_n \mathtt{:} \tau_n\} \rightsquigarrow \mathtt{copy}_{n,l,\sigma,v}}$$

These rules make use of the following coercion functions:

$$\begin{aligned} \mathtt{drop}_{m,n,l,\sigma} &= \\ &\mathtt{fn}\, x\mathtt{:}\{l_1\mathtt{:}\sigma_1\mathtt{,}\ldots\mathtt{,}l_m\mathtt{:}\sigma_m\}\,\mathtt{in}\,\{l_1\mathtt{:}x\mathtt{.}l_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}x\mathtt{.}l_n\}\,\mathtt{end}\end{aligned}$$

$$\begin{aligned} \mathtt{copy}_{n,l,\sigma,v} &= \\ &\mathtt{fn}\, x\mathtt{:}\{l_1\mathtt{:}\sigma_1\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}\sigma_n\}\,\mathtt{in}\,\{l_1\mathtt{:}v_1\mathtt{(}x\mathtt{.}l_1\mathtt{)}\mathtt{,}\ldots\mathtt{,}l_n\mathtt{:}v_n\mathtt{(}x\mathtt{.}l_n\mathtt{)}\}\,\mathtt{end}\end{aligned}$$

In essence this approach represents a trade-off between the cost of subsumption and the cost of field selection. By creating a new record whenever subsumption is used, we make field selection cheap. On the other hand, we can make subsumption free, provided that we are willing to pay the cost of a search whenever a field is selected from a record.

But what if record fields are mutable? This approach to coercion is out of the question, because of *aliasing*. Suppose that a mutable record value $v$ is bound to two variables, $x$ and $y$. If coercion is applied to the binding of $x$, creating a new record, then future changes to $y$ will not affect the new record, nor vice versa. In other words, uses of coercion changes the semantics of a program, which is unreasonable.

One widely-used approach is to increase slightly the cost of field selection (by a constant factor) by separating the "view" of a record from its "contents". The view determines the fields and their types that are present for each use of a record, whereas the contents is shared among all uses. In essence we represent the record type $\{l_1 \mathtt{:} \tau_1 \mathtt{,} \ldots \mathtt{,} l_n \mathtt{:} \tau_n\}$ by the product type

$$\{l_1\mathtt{:int,}\ldots\mathtt{,}l_n\mathtt{:int}\}\mathtt{*}(\tau\,\mathtt{array}).$$

The field selection $l\mathtt{.}e$ becomes a two-stage process:

$$\mathtt{snd(}e\mathtt{)[fst(}e\mathtt{).}l\mathtt{]}$$

Finally, coercions copy the view, without modifying the contents. If $\sigma = \{l_1 \mathtt{:} \sigma_1 \mathtt{,} \ldots \mathtt{,} l_n \mathtt{:} \sigma_n\}$ and $\tau = \{l_1\mathtt{:int,}\ldots\mathtt{,}l_n\mathtt{:int}\}$, then

$$\mathtt{drop}_{m,n,l,\sigma} = \mathtt{fn}\,x\,\mathtt{in}\,\mathtt{(drop}_{m,n,l,\tau}\mathtt{(fst(}x\mathtt{)),snd(}x\mathtt{))}\,\mathtt{end}.$$

# Chapter 27

# Inheritance and Subtyping in Java

In this note we discuss the closely-related, but conceptually distinct, notions of *inheritance*, or *subclassing*, and *subtyping* as exemplified in the Java language. Inheritance is a mechanism for supporting *code re-use* through incremental extension and modification. Subtyping is a mechanism for expressing *behavioral relationships* between types that allow values of a subtype to be provided whenever a value of a supertype is required.

In Java inheritance relationships give rise to subtype relationships, but not every subtype relationship arises via inheritance. Moreover, there are languages (including some extensions of Java) for which subclasses do not give rise to subtypes, and there are languages with no classes at all, but with a rich notion of subtyping. For these reasons it is best to keep a clear distinction between subclassing and subtyping.

## 27.1 Inheritance Mechanisms in Java

### 27.1.1 Classes and Instances

The fundamental unit of inheritance in Java is the *class*. A class consists of a collection of *fields* and a collection of *methods*. Fields are assignable variables; methods are procedures acting on these variables. Fields and methods can be either *static* (per-class) or *dynamic* (per-instance).[1] Static fields are per-class data. Static methods are just ordinary functions acting on static fields.

[1]Fields and methods are assumed dynamic unless explicitly declared to be static.

Classes give rise to *instances*, or *objects*, that consist of the dynamic methods of the class together with fresh copies (or instances) of its dynamic fields. Instances of classes are created by a *constructor*, whose role is to allocate and initialize fresh copies of the dynamic fields (which are also known as *instance variables*). Constructors have the same name as their class, and are invoked by writing `new` $C$`(`$e_1, \ldots, e_n$`)`, where $C$ is a class and $e_1, \ldots, e_n$ are arguments to the constructor.[2] Static methods have access only to the static fields (and methods) of its class; dynamic methods have access to both the static and dynamic fields and methods of the class.

The components of a class have a designated *visibility* attribute, either `public`, `private`, or `protected`. The public components are those that are accessible by all clients of the class. Public static components are accessible to any client with access to the class. Public dynamic components are visible to any client of any instance of the class. Protected components are "semi-private; we'll have more to say about protected components later.

The components of a class also have a *finality* attribute. Final fields are not assignable — they are read-only attributes of the class or instance. Actually, final dynamic fields can be assigned exactly once, by a constructor of the class, to initialize their values. Final methods are of interest in connection with inheritance, to which we'll return below.

The components of a class have *types*. The type of a field is the type of its binding as a (possibly assignable) variable. The type of a method specifies the types of its arguments (if any) and the type of its results. The type of a constructor specifies the types of its arguments (if any); its "result type" is the instance type of the class itself, and may not be specified explicitly. (We will say more about the type structure of Java below.)

The public static fields and methods of a class $C$ are accessed using "dot notation". If $f$ is a static field of $C$, a client may refer to it by writing $C$`.`$f$. Similarly, if $m$ is a static method of $C$, a client may invoke it by writing $C$`.`$m$`(`$e_1$`,...,`$e_n$`)`, where $e_1, \ldots, e_n$ are the argument expressions of the method. The expected type checking rules govern access to fields and invocations of methods.

The public dynamic fields and methods of an instance $c$ of a class $C$ are similarly accessed using "dot notation", *albeit* from the instance, rather than the class. That is, if $f$ is a public dynamic field of $C$, then $c$`.`$f$ refers to the $f$ field of the instance $c$. Since distinct instances have distinct fields, there is no essential connection between $c$`.`$f$ and $c'$`.`$f$ when $c$ and $c'$ are

[2]Classes can have multiple constructors that are distinguished by overloading. We will not discuss overloading here.

distinct instances of class $C$. If $m$ is a public dynamic method of $C$, then $c$.$m$($e_1, \ldots, e_n$) invokes the method $m$ of the instance $c$ with the specified arguments. This is sometimes called *sending a message $m$ to instance $c$ with arguments $e_1, \ldots, e_n$*.

Within a dynamic method one may refer to the dynamic fields and methods of the class via the pseudo-variable `this`, which is bound to the instance itself. The methods of an instance may call one another (or themselves) by sending a message to `this`. Although Java defines conventions whereby explicit reference to `this` may be omitted, it is useful to eschew these conveniences and always use `this` to refer to the components of an instance from within code for that instance. We may think of `this` as an implicit argument to all methods that allows the method to access to object itself.

### 27.1.2 Subclasses

A class may be defined by *inheriting* the visible fields and methods of another class. The new class is said to be a *subclass* of the old class, the *superclass*. Consequently, inheritance is sometimes known as *subclassing*. Java supports *single inheritance* — every class has at most one superclass. That is, one can only inherit from a single class; one cannot combine two classes by inheritance to form a third. In Java the subclass is said to `extend` the superclass.

There are two forms of inheritance available in Java:

1. *Enrichment*. The subclass enriches the superclass by providing additional fields and methods not present in the superclass.

2. *Overriding*. The subclass may re-define a method in the superclass by giving it a new implementation in the subclass.

Enrichment is a relatively innocuous aspect of inheritance. The true power of inheritance lies in the ability to override methods.

Overriding, which is also known as *method specialization*, is used to "specialize" the implementation of a superclass method to suit the needs of the subclass. This is particularly important when the other methods of the class invoke the overridden method by sending a message to `this`. If a method $m$ is overridden in a subclass $D$ of a class $C$, then all methods of $D$ that invoke $m$ via `this` will refer to the "new" version of $m$ defined by the override. The "old" version can still be accessed explicitly from the subclass by referring to `super.`$m$. The keyword `super` is a pseudo-variable that may be used to refer to the overridden methods.

Inheritance can be controlled using visibility constraints. A sub-class $D$ of a class $C$ automatically inherits the private fields and methods of $C$ without the possibility of overriding, or otherwise accessing, them. The public fields and methods of the superclass are accessible to the subclass without restriction, and retain their `public` attribute in the subclass, unless overridden. A `protected` component is "semi-private" — accessible to the subclass, but not otherwise publically visible.[3]

Inheritance can also be limited using finality constraints. If a method is declared `final`, it may not be overridden in any subclass — it must be inherited as-is, without further modification. However, if a final method invokes, via `this`, a non-final method, then the behavior of the final method can still be changed by the sub-class by overriding the non-final method. By declaring an entire class to be final, no class can inherit from it. This serves to ensure that any instance of this class invokes the code from this class, and not from any subclass of it.

Instantiation of a subclass of a class proceeds in three phases:

1. The instance variables of the subclass, which include those of the superclass, are allocated.

2. The constructor of the superclass is invoked to initialize the superclass's instance variables.

3. The constructor of the subclass is invoked to initialize the subclass's instance variables.

The superclass constructor can be explicitly invoked by a subclass constructor by writing `super(`$e_1, \ldots, e_n$`)`, but *only* as the very first statement of the subclass's constructor. This ensures proper initialization order, and avoids certain anomalies and insecurities that arise if this restriction is relaxed.

### 27.1.3 Abstract Classes and Interfaces

An *abstract class* is a class in which one or more methods are declared, but left unimplemented. Abstract methods may be invoked by the other methods of an abstract class by sending a message to `this`, but since their implementation is not provided, abstract classes do not themselves have instances. Instead the obligation is imposed on a subclass of the abstract

[3]Actually, Java assigns `protected` components "package scope", but since we are not discussing packages here, we will ignore this issue.

class to provide implementations of the abstract methods to obtain a *concrete* class, which does have instances. Abstract classes are useful for setting up "code templates" that are instantiated by inheritance. The abstract class becomes the locus of code sharing for all concretions of that class, which inherit the shared code and provide the missing non-shared code.

Taking this idea to the extreme, an *interface* is a "fully abstract" class, which is to say that

- All its fields are `public static final` (*i.e.*, they are constants).
- All its methods are `abstract public`; they must be implemented by a subclass.

Since interfaces are a special form of abstract class, they have no instances.

The utility of interfaces stems from their role in `implements` declarations. As we mentioned above, a class may be declared to extend a *single* class to inherit from it.[4] A class may also be declared to `implement` *one or more* interfaces, meaning that the class provides the public methods of the interface, with their specified types. Since interfaces are special kinds of classes, Java is sometimes said to provide *multiple inheritance of interfaces*, but only *single inheritance of implementation*. For similar reasons an interface may be declared to extend multiple interfaces, provided that the result types of their common methods coincide.

The purpose of declaring an interface for a class is to support writing generic code that works with *any* instance providing the methods specified in the interface, *without* requiring that instance to arise from any particular position in the inheritance hierarchy. For example, we may have two unrelated classes in the class hierarchy providing a method $m$. If both classes are declared to implement an interface that mentions $m$, then code programmed against this interface will work for an instance of *either* class.

The literature on Java emphasizes that interfaces are *descriptive* of behavior (to the extend that types alone allow), whereas classes are *prescriptive* of implementation. While this is surely a noble purpose, it is curious that interfaces are *classes* in Java, rather than *types*. In particular interfaces are unable to specify the public fields of an instance by simply stating their types, which would be natural were interfaces a form of type. Instead fields in interfaces are forced to be constants (public, static, final), precluding their use for describing the public instance variables of an object.

[4]Classes that do not specify a superclass implicitly extend the class `Object` of all objects.

## 27.2 Subtyping in Java

The Java type system consists of the following types:

1. *Base types*, including `int`, `float`, `void`, and `boolean`.
2. *Class types* $C$, which classify the instances of a class $C$.
3. *Array types* of the form $\tau$ `[ ]`, where $\tau$ is a type, representing mutable arrays of values of type $\tau$.

The basic types behave essentially as one would expect, based on previous experience with languages such as C and C++. Unlike C or C++ Java has true array types, with operations for creating and initializing an array and for accessing and assigning elements of an array. All array operations are safe in the sense that any attempt to exceed the bounds of the array results in a checked error at run-time.

Every class, whether abstract or concrete, including interfaces, has associated with it the type of its instances, called (oddly enough) the *instance type* of the class. Java blurs the distinction between the class as a program structure and the instance type determined by the class — class names serve not only to identify the class but also the instance type of that class. It may seem odd that abstract classes, and interfaces, all define instance types, even though they don't have instances. However, as will become clear below, even abstract classes have instances, indirectly through their concrete subclasses. Similarly, interfaces may be thought of as possessing instances, namely the instances of concrete classes that implement that interface.

### 27.2.1 Subtyping

To define the Java subtype relation we need two auxiliary relations. The *subclass* relation, $C \lhd C'$, is the reflexive and transitive closure of the *extends* relation among classes, which holds precisely when one class is declared to extend another. In other words, $C \lhd C'$ iff $C$ either coincides with $C'$, inherits directly from $C'$, or inherits from a subclass of $C'$. Since interfaces are classes, the subclass relation also applies to interfaces, but note that multiple inheritance of interfaces means that an interface can be a subinterface (subclass) of more than one interface. The *implementation* relation, $C \blacktriangleleft I$, is defined to hold exactly when a class $C$ is declared to implement an interface that inherits from $I$.

The Java subtype relation is inductively defined by the following rules. Subtyping is reflexive and transitive:

$$\frac{}{\tau \texttt{ <: } \tau} \tag{27.1}$$

$$\frac{\tau \texttt{ <: } \tau' \quad \tau' \texttt{ <: } \tau''}{\tau \texttt{ <: } \tau''} \tag{27.2}$$

Arrays are *covariant* type constructors, in the sense of this rule:

$$\frac{\tau \texttt{ <: } \tau'}{\tau\,\texttt{[ ]} \texttt{ <: } \tau'\,\texttt{[ ]}} \tag{27.3}$$

Inheritance implies subtyping:

$$\frac{C \lhd C'}{C \texttt{ <: } C'} \tag{27.4}$$

Implementation implies subtyping:

$$\frac{C \blacktriangleleft I}{C \texttt{ <: } I} \tag{27.5}$$

Every class is a subclass of the distinguished "root" class `Object`:

$$\frac{}{\tau \texttt{ <: Object}} \tag{27.6}$$

The array subtyping rule is a structural subtyping principle — one need not explicitly declare subtyping relationships between array types for them to hold. On the other hand, the inheritance and implementation rules of subtyping are examples of nominal subtyping — they hold when they are declared to hold at the point of definition (or are implied by further subtyping relations).

### 27.2.2 Subsumption

The subsumption principle tells us that if $e$ is an expression of type $\tau$ and $\tau \texttt{ <: } \tau'$, then $e$ is also an expression of type $\tau'$. In particular, if a method is declared with a parameter of type $\tau$, then it makes sense to provide an argument of any type $\tau'$ such that $\tau' \texttt{ <: } \tau$. Similarly, if a constructor takes a parameter of a type, then it is legitimate to provide an argument of a subtype of that type. Finally, if a method is declared to return a value of type $\tau$, then it is legitimate to return a value of any subtype of $\tau$.

This brings up an awkward issue in the Java type system. What should be the type of a conditional expression $e \, \texttt{?} \, e_1 \, \texttt{:} \, e_2$? Clearly $e$ should have type `boolean`, and $e_1$ and $e_2$ should have the same type, since we cannot in general predict the outcome of the condition $e$. In the presence of subtyping, this amounts to the requirement that the types of $e_1$ and $e_2$ have an *upper bound* in the subtype ordering. To avoid assigning an excessively weak type, and to ensure that there is a unique choice of type for the conditional, it would make sense to assign the conditional the *least upper bound* of the types of $e_1$ and $e_2$. Unfortunately, two types need not have a least upper bound! For example, if an interface $I$ extends incomparable interfaces $K$ and $L$, and $J$ extends both $K$ and $L$, then $I$ and $J$ do not have a least upper bound — both $K$ and $L$ are upper bounds of both, but neither is smaller than the other. To deal with this Java imposes the rather *ad hoc* requirement that either the type of $e_1$ be a subtype of the type of $e_2$, or *vice versa*, to avoid the difficulty.

A more serious difficulty with the Java type system is that the array subtyping rule, which states that the array type constructor is *covariant* in the type of the array elements, violates the subsumption principle. To understand why, recall that we can do one of two things with an array: retrieve an element, or assign to an element. If $\tau \texttt{ <: } \tau'$ and $A$ is an array of type $\tau \, \texttt{[ ]}$, then retrieving an element of $A$ yields a value of type $\tau$, which is by hypothesis an element of type $\tau'$. So we are OK with respect to retrieval. Now consider array assignment. Suppose once again that $\tau \texttt{ <: } \tau'$ and that $A$ is an array of type $\tau \, \texttt{[ ]}$. Then $A$ is also an array of type $\tau' \, \texttt{[ ]}$, according to the Java rule for array subtyping. This means we can assign a value $x$ of type $\tau'$ to an element of $A$. But this violates the assumption that $A$ is an array of type $\tau \, \texttt{[ ]}$ — one of its elements is of type $\tau'$.

With no further provisions the language would not be type safe. It is a simple matter to contrive an example involving arrays that incurs a run-time type error ("gets stuck"). Java avoids this by a simple, but expensive, device — every array assignment incurs a "run-time type check" that ensures that the assignment does not create an unsafe situation. In the next subsection we explain how this is achieved.

### 27.2.3 Dynamic Dispatch

According to Java typing rules, if $C$ is a sub-class of $D$, then $C$ is a sub-type of $D$. Since the instances of a class $C$ have type $C$, they also, by subsumption, have type $D$, as do the instances of class $D$ itself. In other words, if the static type of an instance is $D$, it might be an instance of class $C$ or an instance of class $D$. In this sense the static type of an instance is at best an approximation of its dynamic type, the class of which it is an instance.

The distinction between the static and the dynamic type of an object is fundamental to object-oriented programming. In particular method specialization is based on the dynamic type of an object, not its static type. Specifically, if $C$ is a sub-class of $D$ that overrides a method $m$, then invoking the method $m$ of a $C$ instance $o$ will always refer to the overriding code in $C$, even if the static type of $o$ is $D$. That is, method dispatch is based on the dynamic type of the instance, not on its static type. For this reason method specialization is sometimes called *dynamic dispatch*, or, less perspicuously, *late binding*.

How is this achieved? Essentially, every object is tagged with the class that created it, and this tag is used to determine which method to invoke when a message is sent to that object. The constructors of a class $C$ "label" the objects they create with $C$. The method dispatch mechanism consults this label when determining which method to invoke.[5]

The same mechanism is used to ensure that array assignments do not lead to type insecurities. Suppose that the static type of $A$ is $C$`[]`, and that the static type of instance $o$ is $C$. By covariance of array types the dynamic type of $A$ might be $D$`[]` for some sub-class $D$ of $C$. But unless the dynamic type of $o$ is also $D$, the assignment of $o$ to an element of $A$ should be prohibited. This is ensured by an explicit run-time check. In Java *every single array assignment incurs a run-time check* whenever the array contains objects.[6]

### 27.2.4 Casting

A *container class* is one whose instances "contain" instances of another class. For example, a class of lists or trees or sets would be a container class in this sense. Since the operations on containers are largely (or entirely) independent of the type of their elements, it makes sense to define containers generally, rather than defining one for each element type. In Java this is achieved by exploiting subsumption. Since every object has type `Object`, a general container is essentially a container whose elements are of type `Object`. This allows the container operations to be defined once for all

[5]In practice the label is a pointer to the vector of methods of the class, and the method is accessed by indexing into this vector. But we can just as easily imagine this to be achieved by a case analysis on the class name to determine the appropriate method vector.

[6]Arrays of integers and floats do not incur this overhead, because numbers are not objects.

element types. However, when retrieving an element from a container its static type is `Object`; we lost track of its dynamic type during type checking. If we wish to use such an object in any meaningful way, we must recover its dynamic type so that message sends are not rejected at compile time.

Java supports a safe form of *casting*, or *change of type*. A cast is written `(`$\tau$`)` $e$. The expression $e$ is called the *subject* of the cast, and the type $\tau$ is the *target type* of the cast. The type of the cast is $\tau$, provided that the cast makes sense, and its value is that of $e$. In general we cannot determine whether the cast makes sense until execution time, when the dynamic type of the expression is available for comparison with the target type. For example, every instance in Java has type `Object`, but its true type will usually be some class further down the type hierarchy. Therefore a cast applied to an expression of type `Object` cannot be validated until execution time.

Since the static type is an attenuated version of the dynamic type of an object, we can classify casts into three varieties:

1. *Up casts*, in which the static type of the expression is a subtype of the target type of the cast. The type checker accepts the cast, and no run-time check is required.

2. *Down casts*, in which the static type of the expression is a *supertype* of the target type. The true type may or may not be a subtype of the target, so a run-time check is required.

3. *Stupid casts*, in which the static type of the expression rules out the possibility of its dynamic type matching the target of the cast. The cast is rejected.

Similar checks are performed to ensure that array assignments are safe.

Note that it is up to the programmer to maintain a sufficiently strong invariant to ensure that down casts do not fail. For example, if a container is intended to contain objects of a class $C$, then retrieved elements of that class will typically be down cast to a sub-class of $C$. It is entirely up to the programmer to ensure that these casts do not fail at execution time. That is, the programmer must maintain the invariant that the retrieved element really contains an instance of the target class of the cast.

## 27.3 Methodology

With this in hand we can (briefly) discuss the methodology of inheritance in object-oriented languages. As we just noted, in Java subclassing entails subtyping — the instance type of a subclass is a subtype of the instance type of the superclass. It is important to recognize that this is a methodological commitment to certain uses of inheritance.

Recall that a subtype relationship is intended to express a form of behavioral equivalence. This is expressed by the subsumption principle, which states that subtype values may be provided whenever a supertype value is required. In terms of a class hierarchy this means that a value of the subclass can be provided whenever a value of the superclass is required. For this to make good sense the values of the subclass should "behave properly" in superclass contexts — they should not be distinguishable from them.

But this isn't necessarily so! Since inheritance admits overriding of methods, we can make almost arbitrary[7] changes to the behavior of the superclass when defining the subclass. For example, we can turn a stack-like object into a queue-like object (replacing a LIFO discipline by a FIFO discipline) by inheritance, thereby changing the behavior drastically. If we are to pass off a subclass instance as a superclass instance using subtyping, then we should refrain from making such drastic behavioral changes.

The Java type system provides only weak tools for ensuring a behavioral subtyping relationship between a subclass and its superclass. Fundamentally, the type system is not strong enough to express the desired constraints.[8]. To compensate for this Java provides the finality mechanism to limit inheritance. Final classes cannot be inherited from at all, ensuring that values of its instance type are indeed instances of that class (rather than an arbitrary subclass). Final methods cannot be overridden, ensuring that certain aspects of behavior are "frozen" by the class definition.

Nominal subtyping may also be seen as a tool for enforcing behavioral subtyping relationships. For unless a class extends a given class or is declared to implement a given interface, no subtyping relationship holds. This helps to ensure that the programmer explicitly considers the behavioral subtyping obligations that are implied by such declarations, and is therefore an aid to controlling inheritance.

[7]Limited only by finality declarations in the superclass.

[8]Nor is the type system of any other language that I am aware of, including ML