5 *Understanding Subtypes*

Thus far we have assumed that only object types have subtypes, and that subtypes are formed only by adding new methods to object types. In this chapter we provide some insight into ways that subtyping can be extended to more types, and how the subtyping relation on object types can be made richer.

SUBTYPE

Recall from Chapter 2 that type `S` is a *subtype* of a type `T`, written `S` <: `T`, if an expression of type `S` can be used in any context that expects an element of type `T`. Another way of putting this is that any expression of type `S` can masquerade as an expression of type `T`.

SUBSUMPTION RULE

This definition can be made more concrete by introducing a rule stating that if `S` <: `T` and expression `e` has type `S`, then `e` also has type `T`. This rule, usually termed the *subsumption rule*, provides a mechanism for informing the type checker that an expression of a subtype can masquerade as an element of a supertype.

Subtyping provides added flexibility in constructing legal expressions of a language. Let `x` be a variable holding values of type `T`. If `e` is an expression of type `T`, then of course `x := e` is a legal assignment statement.

Now suppose that `S` is a subtype of `T` and `e'` has type `S`. Then `e'` can masquerade as an element of type `T`, and hence `x := e'` will also be a legal assignment statement. Similarly an actual parameter of type `S` may be used in a function or procedure call when the corresponding formal parameter's type is declared to be `T`.

In most pure object-oriented languages, objects are represented as implicit references. Assignment and parameter passing are interpreted as binding new names to existing objects, i.e., as ways of creating sharing. Because elements of a supertype and subtype both take the same amount of space (the space to hold a reference) there is no implementation difficulty in using elements of the subtype in contexts expecting elements of the supertype. The difficulty instead is determining when using an element of another type is logically correct, *i.e.*, all operations expected for the supertype make sense for the subtype.

How can we determine when one type is a subtype of another? A careful theoretical analysis of this topic would take us far afield from the aims of this text into complex issues of domain theory in denotational semantics. Instead we will present intuitive arguments for determining when one type is a subtype of another. The subtyping rules in this section are based on those given by Cardelli [Car88].

5.1 Subtyping for non-object types

Because object types have similarities to records of functions, we begin with examining the simpler cases of subtyping for record and function types, holding off on object types until later in this chapter. We also include a discussion of references (*i.e.*, the types of variables in programming languages) here, in order to prepare for the later discussion of instance variables in objects. This will also be useful in discussing subtyping for arrays and mutable records.

5.1.1 Record types

In order to keep the initial discussion as simple as possible, we deal in this subsection only with immutable (or "read-only") records of the sort found in functional programming languages like ML. While one can create records in a single operation, the only operations that may be applied to existing immutable record values are to extract the values of particular fields. No operations are available to update particular fields of these records. Because the operations do not depend on the order of the fields, we consider record types that differ only in the order of their fields as identical.

An object can be interpreted as a record whose fields include their methods. Because methods may not be updated in objects, the study of immutable records will be important to our understanding of object types. We discuss in Section 5.1.3 the impact of allowing updatable fields.

Records associate values to labels so that the values may be extracted using the name of the label. The type of a record specifies the type of the value corresponding to each label. For example, we can define the record type

```
SandwichType = {| bread: BreadType; filling: FoodType|}.
```

An example of an element of type `SandwichType` is

```
s: Sandwich := {| bread: BreadType := rye;
                filling: FoodType := pastrami |}
```

Because these records are immutable, the only operations available on `s` are the extraction of values held in the `bread` and `filling` fields via expressions `s.bread` and `s.filling`.

Suppose that we are given that `CheeseType` <: `FoodType`. Let

```
CheeseSandwichType = {| bread: BreadType;
                        filling: CheeseType;
                        sauce: SauceType |}
```

and

```
cs: CheeseSandwich := {| bread: BreadType := white;
                        filling: CheeseType := cheddar;
                        sauce: SauceType := mustard |}
```

We claim that `CheeseSandwichType` <: `SandwichType`.

For elements of `CheeseSandwichType` to successfully masquerade as elements of `SandwichType`, expressions of type `CheeseSandwichType` need to support all of the operations applicable to expressions of type `SandwichType`. Since the only operation available on these records is extracting fields, it is straightforward to show this.

A record `cs` of type `CheeseSandwichType` has the `bread` and `filling` fields expected of a value of type `SandwichType`. Moreover, the results of extracting the `bread` field from values of each of the two sandwich types each have type `BreadType`. The result of extracting the `filling` field from a record of type `CheeseSandwichType` is of type `CheeseType`, which is not the same as `FoodType`. However, because `CheeseType` <: `FoodType`, it can masquerade as a value of type `FoodType`.

Thus no matter which label from `FoodType` is extracted from a value of `CheeseSandwichType`, the result can masquerade as the corresponding type of `SandwichType`. Hence `CheeseSandwichType` is a subtype of `SandwichType`. The extra fields in `CheeseSandwichType` are irrelevant as we only need to know that enough fields of the appropriate types are available in order to masquerade as a `Sandwich` type.

Figure 5.1 illustrates a slightly more abstract version of this argument. In that figure a record `r': {| m: S'; n: T'; p: U' q: V' |}` is masquerading

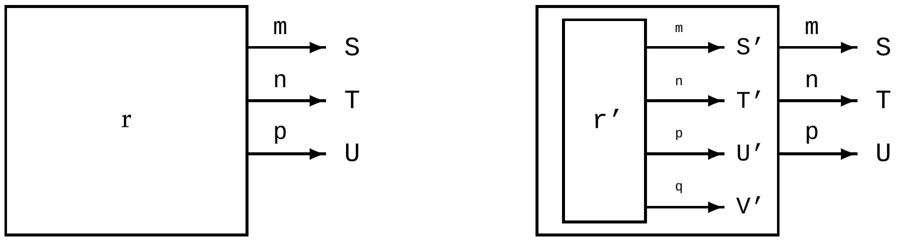

Figure 5.1 A record `r`: {| `m`: `S`; `n`: `T`; `p`: `U` |}, and another record `r'`: {| `m`: `S'`; `n`: `T'`; `p`: `U'`; `q`: `V'` |} masquerading as an element of type {| `m`: `S`; `n`: `T`; `p`: `U` |}.

as a record of type {| `m: S; n: T; p: U` |}. We illustrate this by placing the figure representing `r'` inside a box (think costume) which has the same interface as an element of type {| `m: S; n: T; p: U` |}.

For the masquerade to be successful, the value of the `m` field of `r'`, for example, must be able to masquerade as a value of type `S`. Similarly for the `n` and `p` fields. Again, notice that the subtype may have more labeled fields (*e.g.*, the `q` field) than the supertype, since the extra fields don't get in the way of any of the operations applicable to the supertype.

Thus one record type is a subtype of another if the first has all of the fields of the second (and perhaps more), and the types of the corresponding fields are subtypes. Notice that the ordering of the fields is irrelevant in determining subtyping. We identify record types that are the same up to the ordering of fields.

We write this more formally as follows. Let $\{\!|\ \mathtt{l}_i\colon \mathtt{T}_i\ |\!\}_{1\leq i\leq n}$ represent the type of a record with labels $\mathtt{l}_i$ of type $\mathtt{T}_i$ for $1 \leq i \leq n$. Then,

$$\{\!|\ \mathtt{l}_j\colon \mathtt{T}_j |\!\}_{1\leq j\leq n} <: \{\!|\ \mathtt{l}_i\colon \mathtt{U}_i |\!\}_{1\leq i\leq k},\ \textit{if } k \leq n \textit{ and for all } 1 \leq i \leq k, \mathtt{T}_i <: \mathtt{U}_i.$$

By this definition, `CheeseSandwichType` <: `SandwichType`.

It is sometimes convenient to break up the subtyping for records into two pieces: breadth and depth subtyping rules. One record type is a *breadth subtype* of another if the first has all of the fields of the second (and perhaps more). A record type is a *depth subtype* of another if they have exactly the same fields, but the types of the corresponding fields are subtypes.

BREADTH SUBTYPE

DEPTH SUBTYPE

Again, the general subtyping rule above is appropriate for record values in which the only operations available are extracting labeled fields. Later we

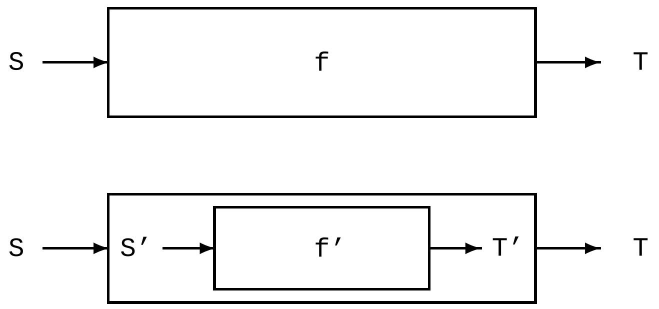

Figure 5.2 A function `f`: `S` → `T`, and another function `f'`: `S'` → `T'` masquerading as having type `S` → `T`.

discuss how the subtyping rule would change if operations were available to update the fields.

5.1.2 Function types

The proper definition of subtyping for function types has provoked great controversy and confusion, so it is worth a careful look. As discussed earlier, we write `S` → `T` for the type of functions that take a parameter of type `S` and return a result of type `T`. If (`S'` → `T'`) <: (`S` → `T`), then we should be able to use an element of the first functional type in any context in which an element of the second type would type check.

Suppose we have a function `f` with type `S` → `T`. In order to use a function, `f'`, with type `S'` → `T'`, in place of `f`, the function `f'` must be able to accept an argument of type `S` and return a value of type `T`. See Figure 5.2.

To masquerade successfully as a function of type `S` → `T`, function `f'` must be able to be applied to an argument, `s`, of type `S`. Because the domain of `f'` is `S'`, it can be applied to elements of type `S` as long as `S` <: `S'`. In that case, using subsumption, `s` can be treated as an element of type `S'`, making `f'(s)` type-correct.

On the other hand, if the output of `f'` has type `T'`, then `T'` <: `T` will guarantee that the output of `f'` can be treated as an element of type `T`. Summarizing,

$$(\texttt{S}' \to \texttt{T}') <: (\texttt{S} \to \texttt{T}), \;\; \textit{if}\, \texttt{S} <: \texttt{S}' \;\textit{and}\; \texttt{T}' <: \texttt{T}$$

If we assume, as before, that `CheeseType` <: `FoodType`, it follows that

$$\texttt{(Integer} \rightarrow \texttt{CheeseType)} <: \texttt{(Integer} \rightarrow \texttt{FoodType)}$$

but

$$\texttt{(FoodType} \rightarrow \texttt{Integer)} <: \texttt{(CheeseType} \rightarrow \texttt{Integer)}$$

In the latter case, if `f'`: `FoodType` → `Integer`, then `f'` can be applied to an expression of type `CheeseType`, since that expression can masquerade as being of type `FoodType`.

The reverse is not true, since if `f`: `CheeseType` → `Integer`, it may not be possible to apply `f` to an argument of type `FoodType`. The body of `f` may apply an operation that is only defined for expressions of type `CheeseType`. For example, suppose `melt` is a function that can be applied to elements of type `CheeseType`, but not `FoodType`. Then if `melt` is applied to the parameter in the body of `f`, an execution error would arise if the actual parameter was of type `FoodType` and not `CheeseType`.

Procedure types may be subtyped as though they were degenerate function types that always return a default type `Void`.

The subtype ordering of parameter types in function subtyping is the reverse of what might initially have been expected, while the output types of functions are ordered in the expected way. We say that subtyping for parameter types is *contravariant* (*i.e.*, goes the opposite direction of the relation being proved), while the subtyping for result types of functions is *covariant* (*i.e.*, goes in the same direction).

The contravariance for parameter types can be initially confusing, because it is always permissible to replace an actual parameter by another whose type is a subtype of the original. However the key is that in the subtyping rule for function types, it is the function, *not* the actual parameter, which is being replaced.

Let us look at one last example to illustrate why contravariance is appropriate for type changes in the parameter position of functions and procedures. The contravariant rule for procedures tells us that it is possible to replace a procedure, `p`, of type `CheeseType` → `Void` by a procedure, `p'`, of type `FoodType` → `Void`.

The procedure `p` can be applied to any value, `cheese`, of type `CheeseType`. Because `CheeseType` <: `FoodType`, the value `cheese` can masquerade as an element of type `FoodType`. As a result, `p'` can also be applied to the value `cheese`. Thus `p'`, and indeed any procedure of type `FoodType` → `Void`, can masquerade as an element of type `CheeseType` → `Void`.

5.1.3 Types of variables

Variables holding values of type `T` have very different properties than simple values of type `T`. Variables holding values of type `T` may be the targets (left sides) of assignments, while values of type `T` may only be the sources (right sides) of such assignments. Obviously, an expression representing a value, *e.g.*, 3, of type integer may not be a target of an assignment statement.

Thus we need to distinguish values of type `T` from variables holding values of type `T`. Because variables are sometimes referred to as references, we will denote the type of variables holding values of type `T` as `Ref T`. Thus a variable `x` holding integer values will have type `Ref Integer`, while the number 17 has type `Integer`.

Variables in programming languages typically represent two kinds of values. This can be seen by examining the meaning of the statement

```
x := x + 1
```

The `x` on the left side of the assignment represents a location in memory that can hold values, while the `x` on the right represents the value stored in that location. These values are sometimes referred to as the *l-value* and *r-value* of the variable. The *l*-value (so-called because it is used as the value of variables to the left of an assignment) represents the location of the variable, while the *r*-value (used for variables occurring on the right side of an assignment) represents the value stored in the variable.

L-VALUE
R-VALUE

To make this distinction clearer as we examine variables, we will use the notation `val x` to stand for the *r*-value of a variable `x`, while an unqualified `x` will represent the *l*-value of the variable. Thus, we would re-write the above assignment as:

```
x := val x + 1
```

In the rest of this subsection we show that the variable (reference) types have only trivial subtypes. We begin as usual with an example.

Suppose once more that `CheeseType` <: `FoodType`, `apple` is a value of type `FoodType`, `fv` is a variable with type `Ref FoodType`, and `cheddar` is a value of type `CheeseType`. Then the assignment

```
fv := apple
```

is type-correct because `apple` has type `FoodType`. It follows that

```
fv := cheddar
```

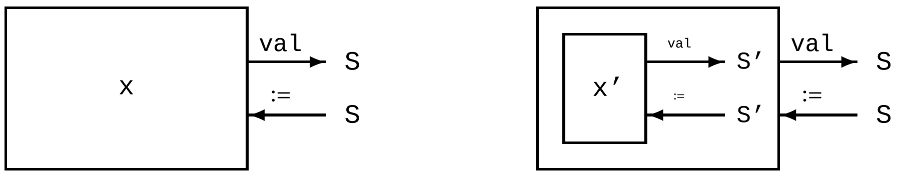

Figure 5.3 A variable `x:Ref S`, and another variable `x′:Ref S′` masquerading as having type `Ref S`.

is also type-correct, because we can always replace a value of type `FoodType` by a value of a subtype. That is, using `cheddar` in a slot expecting a value of type `FoodType` is safe because `CheeseType` <: `FoodType`.

Suppose `cv` is a variable with type `Ref CheeseType`. We noted above that `fv := apple` is fine, but replacing `fv` by `cv` in the assignment statement to obtain `cv := apple` results in a type error.

For example, suppose again that `melt` is a function that can be applied to cheeses but not general foods like apples. Thus an execution error would result if `melt` were applied to `cv` and it held a value, `apple`, that was not of type `CheeseType`.

Thus it is *not* type-correct to replace a *variable* holding values of a given type by a *variable* holding values of a subtype.

As suggested in the example above, the fact that variables may be the targets of assignments will have a great impact on the subtype properties (or rather the lack of them) of reference types. In particular, the example illustrates that `Ref CheeseType` cannot be a subtype of `Ref FoodType`, even though `CheeseType` <: `FoodType`.

Suppose we wish variable `x′` with type `Ref S′` to masquerade as a variable holding values of type `S`. See Figure 5.3 for a graphic illustration.

As indicated earlier, a variable `x` holding values of type `S` has two values: an *l-value* and an *r-value*, where the latter value is obtained by writing `val x`. Thus two operations are applicable to variables, assignment statements with the variable on the left, and `val` expressions. In the first of these, the variable occurs in a value-receiving context, while in the second it occurs in a value-supplying context.

The second of the two operations is represented in the figure by the arrow labeled "`val`" coming out of the variable (because it supplies a value). If `x` is a variable with type `Ref S`, then `val x` returns a value of type `S`.

For a variable `x'` holding values of type `S'` to be able to masquerade as a value of type `S` in all contexts of this kind, we need `S'` <: `S`. This should be clear from the right diagram in the figure, where in order for `x'` to provide a compatible value using the `val` operator, we need `S'` <: `S`.

A value-receiving context is one in which a variable holding values of type `S` is the target of an assignment, *e.g.*, a statement of the form `x := e`, for `e` an expression of type `S`. This is represented in the figure by an arrow labeled "`:=`" going into the variable.

In this context we will be interpreting the variable as a reference or location (*i.e.*, the *l*-value) in which to store a value. We have already seen that an assignment `x := e` is type safe if the type `S` of `e` is a subtype of the type declared to be held in the variable `x`. Thus if we wish to use a variable holding values of type `S'` in all contexts where the right side of the assignment is a value of type `S`, we must ensure that `S` <: `S'`. Again this should be clear from the right diagram in the figure.

Going back to the example at the beginning of this section, suppose we have an assignment statement,

```
cv := cheddar
```

for `cv` a variable holding values of type `CheeseType` and `cheddar` a value of type `CheeseType`. If `fv` is a variable holding values of type `FoodType`, then we can insert `fv` in place of `cv` in the assignment statement, obtaining

```
fv := cheddar
```

Because `CheeseType` <: `FoodType`, this assignment is legal. However the assignment `cv := apple` would *not* be legal.

Thus for a variable holding values of type `S'` to masquerade as a variable holding values of type `S` in value-supplying (*r*-value) contexts we must have

`S'` <: `S`

while it can masquerade in value-receiving (*l*-value) contexts only if

`S` <: `S'`

It follows that there are no non-trivial[1] subtypes of variable (reference) types.

Thus,

`Ref S'` <: `Ref S`, *if* `S'` ≃ `S`,

1. A subtype is trivial if it is equivalent to the supertype in the sense that they are each subtypes of each other.

where `S'` $\simeq$ `S` abbreviates `S'` <: `S` and `S` <: `S'`. We can think of $\simeq$ as defining an equivalence class of types including such things as pairs of record types that differ only in the order of fields. It is common to ignore the differences between such types and to consider them equivalent.

We can get a deeper understanding of the behavior of reference and function types under subtyping by considering the different roles played by *suppliers* and *receivers* of values. Any slot in a type expression that corresponds to a supplier of values must have subtyping behave covariantly (the same direction as the full type expression), while any slot corresponding to a receiver of values must have contravariant subtyping (the opposite direction).

Thus *l*-values of variables and parameters of functions, both of which are receivers of argument values, behave contravariantly with respect to subtyping. On the other hand, the *r*-values of variables and the results of functions, both of which are suppliers of values, behave covariantly. Because variables have both behaviors, any changes in type must be simultaneously contravariant and covariant. Hence subtypes of reference types must actually be equivalent.

5.1.4 Types of updatable records and arrays

The same analysis as for references can lead us to subtyping rules for updatable records and arrays. An updatable record should support operations of the form `r.l := e`, which results in a record whose `l` field is `e`, while the values of the other fields are unchanged. The simplest way to model this with the constructs introduced so far is to represent an updatable record as an immutable record, each of whose fields represents a reference.[2] Thus the fields represent locations whose values could be updated.

An updatable record with name and age fields would have type

```
PersonInfo = {| name: Ref String; age: Ref Integer |}
```

Thus if `mother` has type `PersonType`, then `mother.name` has type `Ref String`.

Combining the record and reference subtyping rules,

$$\{\!|\ \mathtt{l}_j\!:\mathtt{Ref}\ \mathtt{T}_j |\!\}_{1\le j\le n} <: \{\!|\ \mathtt{l}_i\!:\mathtt{Ref}\ \mathtt{U}_i |\!\}_{1\le i\le k}, \quad \textit{if } k \le n \textit{ and for all } 1 \le i \le k, \mathtt{T}_i \simeq \mathtt{U}_i.$$

2. In a real implementation, the locations of the fields would be calculated from the location of the beginning of the record and the size of each field. However this difference has no impact on the subtyping rules.

Thus the subtype has at least the fields of the supertype, but, because the fields can be updated, corresponding fields must have equivalent types. Thus adding fields results in a subtype, but no changes to the types of existing fields is allowed.

Arrays behave analogously to functions. Let

```
ROArray[IndexType] of T
```

denote a read-only array of elements of type `T` with subscripts in `IndexType`. This data type can be modeled by a function from `IndexType` to `T`, as one can think of accessing an array element, `A[i]`, as being similar to applying a function to that index and obtaining the value. As a result, the subtyping rules are similar to those of functions:

$$\texttt{ROArray[IndexType'] of S'} <: \texttt{ROArray[IndexType] of S,}$$
$$\textit{if } \texttt{S'} <: \texttt{S} \textit{ and } \texttt{IndexType} <: \texttt{IndexType'}$$

Intuitively, the index types of read-only arrays change contravariantly because, like function parameters, they are value receivers, while the types of elements of the arrays change covariantly because read-only arrays supply values of those types, just like function return types.

Of course, arrays in most programming languages allow individual components to be updated. We can model `Array[IndexType] of T` by a function from `IndexType` to `Ref T`. From function and reference subtyping rules it follows that

$$\texttt{Array [IndexType'] of S'} <: \texttt{Array [IndexType] of S}$$
$$\textit{if } \texttt{S'} \simeq \texttt{S} \textit{ and } \texttt{IndexType} <: \texttt{IndexType'}$$

As before, the index types of arrays change contravariantly, but now the types of elements of the arrays are invariant because arrays both supply and receive values of those types.

Java's [AGH99] type rules for array types are not statically type-safe. In Java the type of an array holding elements of type `T` is written `T[]`.[3] The subtyping rule for array types in Java is

$$\texttt{S'[]} <: \texttt{S[]}, \textit{ if } \texttt{S'} <: \texttt{S}.$$

The following Java class will illustrate the problems with this typing rule. Suppose `C` is a class with a subclass `CSub`, and suppose the method `methodOfCSubOnly()` is defined in class `CSub`, but was not available in `C`. Now define the class `BreakJava` below:

3. Java array types do not mention the type of subscripts because they are always integers.

```
class BreakJava{
   C v = new C();
   void arrayProb(C[] anArray){
      if (anArray.length > 0)
         anArray[0] = v;                      // ( 2 )
   }

   static void main(String[] args){
      BreakJava bj = new BreakJava();
      CSub paramArray = new CSub[10];
      bj.arrayProb(paramArray);               // ( 1 )
      paramArray[0].methodOfCSubOnly();       // ( 3 )
   }
}
```

The first two lines of the `main` method construct a new instance of the class `BreakJava` and create an array of `CSub` with 10 elements. The message send of `arrayProb` to `bj` at (1) will result in a type error.

The problem is that `paramArray`, an array of elements of type `CSub`, is passed to method `arrayProb` where an array of type `C` was expected. Because of this, the assignment in line (2) of `arrayProb` will result in the assignment of a value `v` from class `C` into an array which is supposed to hold values of type `CSub`. We know that it is illegal to assign a value of a superclass into a variable holding values of the subclass. In fact, if allowed this would result in a run-time error in line (3), which would be executed immediately after the method `arrayProb` finishes executing. Because the value `v` of class `C` was assigned to `paramArray[0]`, the message send of `methodOfCSubOnly()` would fail as elements of class `C` do not support that method.

The correct rule for arrays specified above implies that the message send in line (1) would result in a static type error because `CSub[]` fails to be a subtype of `C[]`.

While the Java designers used an incorrect rule for static checks of subtyping with arrays, they compensated for this by inserting extra dynamic checks. Thus Java would not indicate any type errors at compile time, but it would insert a dynamic check at line (2) because of the assignment to an array parameter. That dynamic check would fail during the execution of the message send `bj.arrayProb(paramArray)` from line (1). The message send at line (3) would never be reached at run time because an exception would have been raised due to the failure of the dynamic check at line (2).

Thus the Java designers compensate for not catching the type error *statically* by performing *dynamic* checks when an individual component of an array is assigned to. Why did they use this obviously faulty subtyping rule, when it results in having to add extra code to assignments to array parameters? While it is necessary for type safety, this extra code in compiled programs is problematic as it results both in increased size of programs and a slowdown in their execution.

One reason the Java designers might have included this faulty rule would be to allow generic sorts (and similar operations) to be written that could pass the static type checker. Java programmers can write sort methods that take elements of type `Comparable[ ]`, where `Comparable` is an interface supporting a method `compareTo` that returns a negative, zero, or positive `int` depending on whether the receiver is smaller than, equal to, or larger than the parameter. Java's unsafe subtyping rule for arrays allows any array of elements that implement `Comparable` to be passed to such sort methods, even though they are in theory vulnerable to the same errors as illustrated above.

However, the actual code written in these sort routines typically does not create a dynamic type error because it simply reorders elements of the array, rather than assigning brand new values. Thus one result of the decision to give up static type safety by including an "incorrect" subtyping rule for arrays is to make it easier for programmers to write more flexible programs.[4]

As we saw in Section 4.1, parametric polymorphism of the sort introduced in GJ would allow the creation of type-correct generic sorts without the need for this unsafe rule. Thus we can recapture static type safety and maintain expressiveness of the language by introducing a richer type system. We will see other examples of this trade-off in later chapters.

5.2 Object types

While most popular object-oriented languages determine subtyping of object types based on whether the corresponding classes are subclasses, this identification of subclass with subtype is not necessary. In this section we determine subtyping rules for objects that depend only on their public interfaces or object types.

4. The reason why this subtyping rule for arrays was included is apparently not as principled. An implementation hack for arrays resulted in a desire for this subtyping rule [Joy98].

The subtyping rules for object types follow from those of records and functions. From the outside, the only operation available on objects is message sending. As a result, object types behave like immutable records. The subtyping rule is:

$$\texttt{ObjectType}\ \{\!|\, \texttt{l}_j\colon \texttt{S}'_j \,|\!\}_{1\le j\le n} <: \texttt{ObjectType}\ \{\!|\, \texttt{l}_i\colon \texttt{S}_i \,|\!\}_{1\le i\le k},$$
$$\textit{if } k \le n \textit{ and for all } 1 \le i \le k, \texttt{S}'_i <: \texttt{S}_i.$$

Because the types $\texttt{S}'_i$ and $\texttt{S}_i$ are method types, they are functional types. Suppose $\texttt{S}'_i = \texttt{T}'_i \to \texttt{U}'_i$ and $\texttt{S}_i = \texttt{T}_i \to \texttt{U}_i$. Then, by the subtyping rule for function types, $\texttt{S}'_i <: \texttt{S}_i$ if both $\texttt{T}_i <: \texttt{T}'_i$ and $\texttt{U}'_i <: \texttt{U}_i$. That is, object types are subtypes if for every method in the supertype there is a method with the same name in the subtype such that the range types of corresponding methods vary covariantly, and the domain types vary contravariantly.

What is the relation between subclasses and subtypes? Most popular statically typed object-oriented languages allow no changes to method types in subclasses. This clearly implies that the object types generated by a subclass-superclass pair are in the subtype relation. We noted earlier that C++ allows covariant changes to result types in subclasses. By the above, this also results in subtypes.

As we saw in Section 4.2, Eiffel [Mey92] allows covariant changes to both parameter and result types of methods in subclasses. We exhibited an example there showing that this was not type-safe. The subtyping rule given here for object types explains this failure by making it clear that covariant changes to parameter types are *not* statically type safe. The language Sather [Omo91] allows contravariant changes to parameter types and covariant changes to return types in subclasses. Thus it is the most flexible in allowing changes to subclasses so that the resulting object types are in the subtype relation.

While our focus in this section has been on subtyping, a related interesting question is what, if any, restrictions must be placed on changing types of methods in subclasses, even if we don't care whether subclasses generate subtypes. We examine that question in Chapter 6.

5.3 Subtyping for class types

We haven't yet introduced the notion of class types as ways of categorizing classes (just as object types categorize objects). We will do that carefully later. However, it is evident that a class type should include information on the types of instance variables and methods. The reason is that to determine whether we can extend a class with new methods or instance variables, we need to know what methods and instance variables already exist there. If a type of a class is to give us sufficient information about a class to determine whether or not a particular extension is legal, it will need to include that information about methods and instance variables.

Let us use the notation `ClassType(IV, M)` for the type of a class whose instance variables have names and types given by the labels and types of record type `IV`, and whose methods have names and types given by the record type `M`. If all instance variables are invisible from outside of an object generated by a class, then the type of objects generated from a class with type `ClassType(IV, M)` will be `ObjectType M`.

Because most object-oriented languages use class names as types, this notation for class types may look unusual to the reader. We emphasize again that objects have types of the form `ObjectType M`, while classes will now have types of the form `ClassType(IV, M)`.

We can ask whether one class type can be a subtype of another. As before, to determine whether one class type can be a subtype of another we must consider what operations are available on classes. There are only two: creating new objects and extending classes to form subclasses. We will see that in our system there can be no non-trivial subtypes of class types exist, because of the difficulty of masquerading in both of these contexts.

Suppose class `C′` of type `ClassType(IV′, M′)` is attempting to masquerade as having type `ClassType(IV, M)`. Let us see what constraints on `IV′`, `IV`, `M′`, and `M` follow from this assumption.

Evaluating `new C′` will generate an object of type `ObjectType M′`. If `C′` is to successfully masquerade as an element of type `ClassType(IV, M)` then the type of the expression `new C′`, `ObjectType M′`, must be a subtype of `ObjectType M`. Thus we need `M′` $<:$ `M`.

Suppose a subclass `SC` is defined by inheritance from `C`:

$$\texttt{class SC inherits C modifies } \texttt{l}_{i_1}, \ldots, \texttt{l}_{i_m}\{\ldots\}$$

so that `SC` is well-typed with type `ClassType(`$\texttt{IV}_{sub}$,$\texttt{M}_{sub}$`)` when `C` has type `ClassType(IV,M)`. If the type of `C′` is a subtype of `ClassType(IV,M)`, then `SC` should be well-typed if `C` is replaced by the masquerading `C′`. However, any method `m` of `C` could have been overridden in `SC` with a method of the same type. Because this override must still be legal in the subclass built from `C′`, all methods in `M` must have the same type in `M′` (as otherwise the override would have been illegal).

Similarly, `M′` could have no more methods than `M`. If `M′` had an extra method, m', we could define a subclass of `C` with an added method m' with an incompatible type from that of m' in `M'`. If we attempt to define a similar subclass from `C'`, we would get a type error in defining the subclass (presuming that there are any restrictions at all on changing types in subclasses).

Thus if `ClassType(IV',M')` $<:$ `ClassType(IV,M)` then we must have `M'` $\simeq$ `M`. Similar arguments on instance variables can be used to show that `IV'` $\simeq$ `IV`. Thus there can be no non-trivial subtypes of class types:

$$\texttt{ClassType}(\texttt{IV'},\texttt{M'}) <: \texttt{ClassType}(\texttt{IV},\texttt{M}),\ \textit{if}\ \texttt{IV'} \simeq \texttt{IV}\ \textit{and}\ \texttt{M'} \simeq \texttt{M}$$

The language discussed so far in this text has no access qualifiers like Java and C++'s private, protected, and public. We will discuss these qualifiers in Section 14.4, where we introduce the names `secret`, `hidden`, and `visible` for access qualifiers whose meanings are similar to Java and C++'s. `Secret` features are not visible outside of the class. That is, they are not visible to subclasses or other objects. Our default for instance variables is that they are `hidden`. This means that they are accessible to subclasses, but not to other objects. In Section 14.4 we assume that class types should not mention `secret` features (i.e., Java's private features). Thus two classes whose `visible` and `hidden` feature names and signatures are the same have the same class type. With this understanding of class types, the claim that there are no non-trivial subtypes for class types remains true.

5.4 Summary

In this chapter, we provided a relatively careful, though informal, analysis of subtyping. The subtyping rules for immutable record types included both *breadth* and *depth* subtyping. That is, a subtype of a record type could include extra labeled fields (breadth) or could replace the type of one of the existing labeled fields by a subtype (depth subtyping).

We addressed the issue of covariance versus contravariance changes in creating subtypes of function types. We discovered that to avoid problems, only covariant changes were allowed to return types and only contravariant changes were allowed to domain types in subtyping function types. Most languages allow no changes to either domain or range types in subtyping function types, though some allow covariant changes in range types. There do not seem to be compelling examples where contravariant changes in domain types are useful.

We emphasize that the rules provided above can be proved mathematically to be safe. Languages that allow covariant changes to both range and domain types (like Eiffel) are not statically type-safe. They either sacrifice type safety altogether or require link or run-time checks to regain type safety.

Reference types (types of variables) allowed no subtyping because elements of these types can both be used as sources of values (*e.g.*, using the `val` construct) and as receivers of values in assignment statements. Subtyping for mutable records and arrays followed naturally from the rules for immutable records, functions, and references. Mutable records allow only breadth subtyping, while arrays only allow contravariant changes to the index types.

Subtyping for object types followed naturally from the rules for immutable records and functions. A subtype of an object type can add new methods (breadth subtyping again) or replace the type of an existing method with a subtype (depth subtyping). By the subtyping rules on function types, one may make contravariant changes to the domain type of the method and covariant changes to the return type. Because instance variables (or hidden methods) do not show up in the public interface of objects, they have no impact on subtyping.

There is no non-trivial subtyping for class types, because of the possibility of conflicts in extending class definitions using inheritance.

We summarize the subtyping rules discussed in this chapter in Figure 5.4. For simplicity we presume that there are no subtype relations involving type constants. (That is, we do not allow `Integer` $<:$ `Real`, for example.)

We have also generalized the subtyping rule for function types to include functions with more than one argument. The domain of a function with multiple arguments is represented as a product or tuple type.

In the next chapter we address the impact of our rules for subtyping on the allowed changes to types of methods and instance variables in defining subclasses.

Rec $_{<:}$

$$\{\!|\, \texttt{l}_j\!:\texttt{T}_j |\!\}_{1\le j\le n} <: \{\!|\, \texttt{l}_i\!:\texttt{U}_i |\!\}_{1\le i\le k},$$
$$\textit{if } k \le n \textit{ and for all } 1 \le i \le k, \texttt{T}_i <: \texttt{U}_i.$$

Fcn $_{<:}$

$$(\texttt{S}'_1 \times \ldots \times \texttt{S}'_n \to \texttt{T'}) <: (\texttt{S}_1 \times \ldots \times \texttt{S}_n \to \texttt{T}),$$
$$\textit{if } \texttt{S}_i <: \texttt{S}'_i \textit{ for } 1 \le i \le n, \textit{ and } \texttt{T'} <: \texttt{T}.$$

Ref $_{<:}$

$$\texttt{Ref S'} <: \texttt{Ref S}, \textit{ if } \texttt{S'} \simeq \texttt{S}.$$

Rec $_{<:}$

$$\{\!|\, \texttt{l}_j\!:\texttt{Ref T}_j |\!\}_{1\le j\le n} <: \{\!|\, \texttt{l}_i\!:\texttt{Ref U}_i |\!\}_{1\le i\le k},$$
$$\textit{if } k \le n \textit{ and for all } 1 \le i \le k, \texttt{T}_i \simeq \texttt{U}_i.$$

Read-only Array $_{<:}$

$$\texttt{ROArray[IndexType'] of S'} <:$$
$$\texttt{ROArray[IndexType] of S},$$
$$\textit{if } \texttt{S'} <: \texttt{S} \textit{ and } \texttt{IndexType} <: \texttt{IndexType'}.$$

Array $_{<:}$

$$\texttt{Array[IndexType'] of S'} <: \texttt{Array[IndexType] of S},$$
$$\textit{if } \texttt{S'} \simeq \texttt{S} \textit{ and } \texttt{IndexType} <: \texttt{IndexType'}.$$

Object $_{<:}$

$$\texttt{ObjectType } \{\!|\texttt{l}_j\!:\texttt{S}'_j|\!\}_{1\le j\le n} <: \texttt{ObjectType } \{\!|\texttt{l}_i\!:\texttt{S}_i|\!\}_{1\le i\le k},$$
$$\textit{if } k \le n \textit{ and for all } 1 \le i \le k, \texttt{S}'_i <: \texttt{S}_i.$$

Class $_{<:}$

$$\texttt{ClassType(IV',M')} <: \texttt{ClassType(IV,M)},$$
$$\textit{if } \texttt{IV'} \simeq \texttt{IV} \textit{ and } \texttt{M'} \simeq \texttt{M}$$

Figure 5.4 Summary of subtyping rules.