6 *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.

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`.

This definition was made more concrete in Chapter 5 by including a *Subsumption* rule among our type-checking rules for $\mathcal{SOOL}$. Recall that the *Subsumption* rule states that if `S` <: `T` and expression `e` has type `S` then `e` also has type `T`. This is in contrast to languages without subtyping in which most expressions can be assigned a unique type. The subsumption rule thus provides a mechanism for allowing an expression of a subtype to masquerade as an element of a supertype.

The support for subtyping provides added flexibility in constructing legal expressions of a language. For instance, let `x` be a variable holding values of type `T`. If `e` is an expression of type `T`, then `x := e` is a legal assignment statement. If `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, these mechanisms are supported by representing objects as implicit references, and interpreting assignment and passing parameters as binding new names to existing objects, i.e., as ways of creating sharing.

How can we determine when one type is a subtype of another? A careful technical discussion 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 first present intuitive arguments for determining when one type is a subtype of another. In the last section of this chapter we sketch the basic ideas behind a model verifying the subtyping rules.

Because the typing of message sends has similarities to typing records of functions, we begin with examining the simpler cases of record, function, and array types now, holding off on object types until later. We also include a discussion of references (*i.e.*, types of variables) here, in order to prepare for the later discussion of instance variables in objects. The subtyping rules in the rest of this section are based on those given by Cardelli [Car88].

6.1 Subtyping for non-object types

Rather than plunging directly into the study of subtyping for object types, we instead take a step back and examine how subtyping might apply to types in more traditional languages.

6.1.1 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. No operations are available to update particular fields of these records. While one can create records as a whole, the only operations which may be applied to record values are to extract the values of particular fields. We discuss in section 6.1.3 the impact of allowing updatable fields.

As discussed earlier, records associate values to particular labels. The type of a record specifies the type of the value corresponding to each label. We can define the record type

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

An example of an element of type `SandwichType` is

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

Because these records are immutable, the only operations available on `s` are the extraction of the `bread` and `filling` fields: `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 = white;
                 filling = cheddar; sauce = mustard; |}
```

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

In order for elements of `CheeseSandwichType` to 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`, and the result of extracting the `filling` field from a record of type `CheeseSandwichType` is of type `CheeseType`, which, by the assumption above, can masquerade as a value of type `FoodType`. Thus `CheeseSandwichType` is a subtype of `SandwichType`.

Figure 6.1 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 as a record of type {| `m: S; n: T; p: U;` |}. For this masquerade to be successful, the value of, for example, the `m` field of `r'` must be able to masquerade as a value of type `S`. Again, notice that the subtype may have more labeled fields 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 also subtypes. We write this more formally as follows. Let $\{\!| \, \mathtt{l}_i : \mathtt{T}_i \, |\!\}_{1 \leq i \leq n}$ represent the type of a record with labels $\mathtt{l}_i$ of type $\mathtt{T}_i$ for $1 \leq i \leq n$. Then,

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

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

Again, this rule is appropriate for record values in which the only operations available are extracting labeled fields. Later we discuss how the subtyping rule would change if operations are available to update the fields.

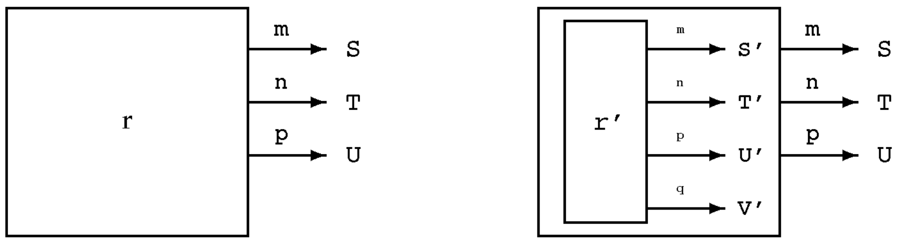

Figure 6.1 A record `r`: {| `m`: `S`; `n`: `T`; `p`: `U`; |}, and record `r'`: {| `m`: `S'`; `n`: `T'`; `p`: `U'`; `q`: `V'`; |} masquerading as an element of type {| `m`: `S`; `n`: `T`; `p`: `U`; |}.

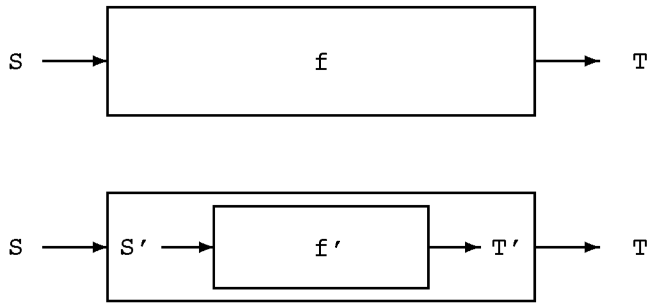

Figure 6.2 A function `f`: `S` → `T`, and `f'`: `S'` → `T'` masquerading as having type `S` → `T`.

6.1.2 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 an element, `f'`, of 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 6.2. Because `f'` is defined to accept arguments of type `S'`, `f'` can be applied to an argument, `s`, of type `S` if `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 can be treated as an element of type `T`. Summarizing,

$$(\texttt{S'} \to \texttt{T'}) <: (\texttt{S} \to \texttt{T}) \textit{ iff } \texttt{S} <: \texttt{S'} \textit{ and } \texttt{T'} <: \texttt{T}$$

Again, assuming that `CheeseSandwichType` <: `SandwichType`, we get that

```
(Integer → CheeseSandwichType)
                           <:  (Integer → SandwichType)
```

but

```
(SandwichType → Integer)
                   <:  (CheeseSandwichType → Integer)
```

Note that in the latter case, if `f'`: `SandwichType` → `Integer`, then `f'` can be applied to an expression of type `CheeseSandwichType`, since that expression can masquerade as being of type `SandwichType`. The reverse is not true, since if `f`: `CheeseSandwichType` → `Integer`, it may not be possible to apply `f` to an argument of type `SandwichType`, as the body of `f` may apply an operation that is only defined for expressions of type `CheeseSandwichType`. For example, if the body of `f` attempts to access the `sauce` field of the parameter, an execution error would arise if the actual parameter was of type `SandwichType` and not `CheeseSandwichType`.

As in our language $\mathcal{SOOL}$, 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` $\rightarrow$ `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` $\rightarrow$ `void` can masquerade as an element of type `CheeseType` $\rightarrow$ `void`.

6.1.3 Types of variables

Variables holding values of type `T` have very different properties than values of type `T`. Variables holding values of type `T` (*i.e.*, values of type `Ref T`) may be the targets of assignments, while values of type `T` may be the sources (right hand sides) of such assignments.

If `fv` is a variable holding values of type `FoodType` (*i.e.*, `fv` has type `Ref FoodType`), and `apple` is a value of type `FoodType`, then the statement `fv := apple;` is a type-correct statement, whereas `apple := fv` is clearly not correct. In this section we show that the types of variables have only trivial subtypes.

Suppose that `CheeseType` $<:$ `FoodType` and `cheddar` is a value of type `CheeseType`. Then `fv := cheddar;` is 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 holding values of type `CheeseType`. We noted above that `fv := apple` is fine, but if we attempt to replace `fv` by `cv` in the assignment statement to obtain `cv := apple`, then we obtain a type error. For instance there may be an operation `melt` which 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`, which 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.

In Chapter 5 we defined type `Ref T` to be the type of a variable which holds values of type `T`. As we saw 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, `Ref CheeseType` is not a subtype of `Ref FoodType`, even though `CheeseType` $<:$ `FoodType`.

Suppose we have a variable `x'` which holds values of type `S'` (*i.e.*, `x'` is an expression of type `Ref S'`), that we wish to have masquerade as a variable holding values of type `S`. See Figure 6.3. As indicated earlier, a variable holding values of type `S` has two values: an *l-value* and an *r-value*. The l-value is the location corresponding to the variable, while the r-value is the value of type `S` actually stored there. In Chapter 5 we introduced the notation `val x` for the value stored in `x`. For example, if `n` is an integer variable, the assignment `n := n + 1` in our language with abbreviations stands for the expanded assignment `n := val n + 1`. Thus in the formal language, two operations are applicable to variables, assignment statements 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 first of the two operations in the figure is represented by the arrow labeled "*val*" coming out of the variable (because it supplies a value). Recall that if `x` is a variable holding values of type `S` (*i.e.*, is a value of type `Ref S`), then `val x` returns a value of type `S`.

By the definition of subtype, in order 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-hand 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-hand side of the assignment is a value of type `S`, we must ensure that `S` <: `S'`. Again this should be clear from the right-hand 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

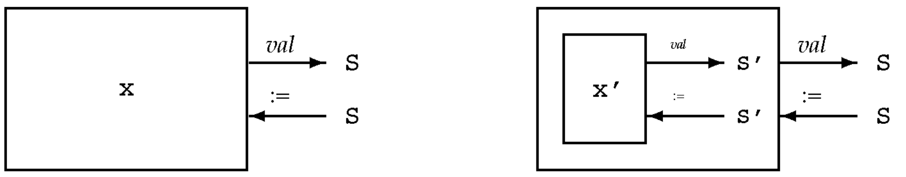

Figure 6.3 A variable `x`: `Ref` S, and `x'`: `Ref` S′ masquerading as having type `Ref S`.

only if `S` <: `S'`. It follows that there are no non-trivial subtypes of variable (reference) types. Thus,

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

where `S'` ≃ `S` abbreviates `S'` <: `S` and `S` <: `S'`. We can think of ≃ 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.

6.1.4 Types of updatable records and arrays

This 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 other fields have the same values as originally. The simplest way to model this with the constructs introduced so far is to represent an updatable record as a record, each of whose fields represents a reference.[1]

An updatable record with name and age fields would have type

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

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

Combining the record and reference subtyping rules, it follows that:

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

Thus the subtype has at least the fields of the supertype, but, because the fields can be updated, corresponding fields must have equivalent types.

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`. Thus

$$\mathtt{ROArray\ [IndexType']\ of\ S'} <: \mathtt{ROArray\ [IndexType]\ of\ S} \quad \textit{iff}\ \mathtt{S'} <: \mathtt{S}\ \textit{and}\ \mathtt{IndexType} <: \mathtt{IndexType'}$$

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

$$\mathtt{Array\ [IndexType']\ of\ S'} <: \mathtt{Array\ [IndexType]\ of\ S} \quad \textit{iff}\ \mathtt{S'} \simeq \mathtt{S}\ \textit{and}\ \mathtt{IndexType} <: \mathtt{IndexType'}$$

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 [AG96] 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[ ]`.[2] The subtyping rule for array types in Java is `S'[ ]` $<:$ `S[ ]` *iff* `S'` $<:$ `S`.

1. 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.
2. Java array types do not include the type of subscripts.

The following Java class illustrates the problems with this typing rule:

```
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 *)
   }
}
```

Suppose class `CSub` extends class `C` by adding a new method `methodOfC-SubOnly`. Then the message send of `arrayProb` to `bj` at (* 1 *) will result in a type error because an element, `v`, of type `C` should not be assignable to an element of an array of type `CSub[ ]` at (* 1 *). If that assignment were allowed, then the message send of `methodOfCSubOnly` would fail because the current value of `paramArray[0]` would be of type `C`, and hence not understand that message at (* 3 *).

If Java used the rule suggested above, a static type error would arise at statement (* 1 *). Instead, 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)` at line (* 1 *). Thus the message send at line (* 3 *) would never be executed at run-time.

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? This extra code in compiled programs results in increased size of programs and a slowdown in the execution of programs.

One reason Java might have included this faulty rule would be to allow generic sorts (and similar operations) to be written which could pass the static type checker. Java programmers can write sort routines which take elements of type `Comparable[ ]`, where `Comparable` is an interface supporting a method `compareTo` which returns a negative, zero, or positive `int` depending on whether the receiver is small than, equal to, or larger than the parameter. Java's unsafe subtyping rule for arrays allows any array of elements which implement `Comparable` to be passed in to such sorts, 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. 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.[3]

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 later in this text.

6.2 Object types

While most popular object-oriented language 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 which depend only on their public interface or object type.

The subtyping rules for object types follow from those of records and references. 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:

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

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, $\texttt{S}'_i <: \texttt{S}_i$ iff $\texttt{T}_i <: \texttt{T}'_i$ and $\texttt{U}'_i <: \texttt{U}_i$. That is, object types are subtypes iff 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.

3. 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 [?].

What is the relation between subclasses and subtypes? Most popular object-oriented languages, like the language $\mathcal{SOOL}$ introduced in Chapter 5, 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.

Eiffel [Mey92] allows covariant changes to both parameter and result types of methods in subclasses. This is *not* statically type safe. While proposals have been made to add a new type-checking phase at link time (a *system validity check*[Mey89]), and more recently a proposal to add checks with an incremental compiler (the *no polymorphic cat-calls* proposal[Mey95]), current Eiffel compilers generate code that can be made to crash because of type errors at run-time. 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 kind of restrictions must we have on changing types of methods in subclasses if we don't care whether subclasses generate subtypes. We examine that question in Chapter 7.

6.3 Subtyping for class types

No non-trival subtypes of class types exist. The possibility of defining subclasses and using inheritance is the main impediment to formulating subtypes. Suppose class `C'` of type `ClassType(IV',M')` is masquerading as having type `ClassType(IV,M)`. Let us see what constraints on `IV'`, `IV`, `M'`, and `M` follow from this assumption. Of course `new C'` generates an object of type `ObjectType M'`. If `C'` is successfully masquerading as an element of type `ClassType(IV,M)` then `new C'` must have a type which is a subtype of `ObjectType M`. Thus we need `M'` $<:$ `M`.

Suppose the subclass given by `class inherits C modifies` $\mathtt{l}_{i_1}, \ldots, \mathtt{l}_{i_m}$ `(IV", M")` is well-typed when `C` has type `ClassType(IV,M)`. Therefore it should be well-typed if `C` is replaced by `C'`. However any method `m` of `C` could have been overridden with a method of the same type. Because this 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` as if `M'` had an extra method, we could define a subclass of `C` with an added method with an incomparable type from that in `M'`. Then if we attempted to define a similar subclass from `C'`, we would get a type error in defining the subclass. Thus if type `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 are no non-trivial subtypes of class types.

6.4 Summary

In this chapter, we provided a relatively careful analysis of subtyping. The subtyping rules for 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 famous covariance-contravariance controversy with 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 which 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 objects of these types can both be used as sources of values (*e.g.*, using the `val` construct in $\mathcal{SOOL}$) and as receivers of values in assignment statements.

Subtyping for object types followed naturally from the rules for records and functions. A subtype of an object type can add new methods (width subtyping again) or replace the type of an existing method with a subtype (depth subtyping). By the subtyping rules on function spaces, 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.

We summarize the subtyping rules for $\mathcal{SOOL}$ types in Figure 6.4. For simplicity we presume that there are no subtype relations involving type con-

Reflexive <:
$$\mathcal{C} \vdash \mathtt{T} <: \mathtt{T}$$

Trans <:
$$\frac{\mathcal{C} \vdash \mathtt{S} <: \mathtt{T} \qquad \mathcal{C} \vdash \mathtt{T} <: \mathtt{U}}{\mathcal{C} \vdash \mathtt{S} <: \mathtt{U}}$$

Record <:
$$\frac{\mathcal{C} \vdash \mathtt{T}_i <: \mathtt{U}_i \textit{ for all } 1 \leq i \leq k, \textit{ and } k \leq n}{\mathcal{C} \vdash \{\!|\, \mathtt{l}_j \colon \mathtt{T}_j \,|\!\}_{1 \leq j \leq n} <: \{\!|\, \mathtt{l}_i \colon \mathtt{U}_i \,|\!\}_{1 \leq i \leq k}}$$

Func <:
$$\frac{\mathcal{C} \vdash \mathtt{S}_i <: \mathtt{S}'_i \textit{ for } 1 \leq i \leq n, \qquad \mathcal{C} \vdash \mathtt{T'} <: \mathtt{T}}{\mathcal{C} \vdash (\mathtt{S}'_1 \times \ldots \mathtt{S}'_n \rightarrow \mathtt{T'}) <: (\mathtt{S}_1 \times \ldots \mathtt{S}_n \rightarrow \mathtt{T})}$$

Obj <:
$$\frac{\mathcal{C} \vdash \mathtt{M'} <: \mathtt{M}}{\mathcal{C} \vdash \mathtt{ObjectType}\ \mathtt{M'} <: \mathtt{ObjectType}\ \mathtt{M}}$$

Type Abbrev <:
$$\frac{\mathcal{C} \vdash \mathcal{C}(\mathtt{T'}) <: \mathcal{C}(\mathtt{T})}{\mathcal{C} \vdash \mathtt{T'} <: \mathtt{T}}$$

Figure 6.4 Subtyping rules for $\mathcal{SOOL}$

stants. 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. We do not include separate rules for reference types, class types, or visible object types as they only have trivial subtypes. The rule *Type Abbrev* <: takes into account the type definitions in $\mathcal{C}$, by allowing type expressions to be in the subtype relation if the subtype relation may be proved after replacing type variables by their definitions.

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.