3 *Type Problems in Object-Oriented Languages*

We begin our study of the type systems of object-oriented programming languages by first providing a critique of the type systems of existing statically typed object-oriented programming languages. The reason for providing such a critique is that our goal is not only to describe existing programming languages, but also to use a deep understanding of the object-oriented concepts in order to design better object-oriented languages. While the next few chapters will mainly focus on describing the types and semantics of existing languages, we hope this chapter will give the reader a better insight into the overall goals of this work and provide motivation for the later work on extending the expressiveness of object-oriented languages.

3.1 Type checking object-oriented languages is difficult

The features of object-oriented languages that provide added flexibility, like subtyping and inheritance, also create difficulties in type checking these languages. We provide here a brief overview of some of the typing difficulties that arise from them.

While subtyping is trivial for simple types, defining a correct notion of subtype for more complex types like record, function, and object types can be tricky. In particular, there has been great confusion over what is the proper subtyping rule for functions. Later we explain why the so-called "contravariant" subtyping rule for the types of parameters in function types is correct and why the "covariant" rule may lead to typing errors.

Adding new instance variables and methods to a subclass does not cause typing difficulties, but modifying existing methods may create problems. If the types of parameters or the return type of the modified method differ from those in the corresponding method of the superclass, it might cause type problems.

If the method `m` being modified was used in a second method `n` of the superclass, then changes in types in `m` may destroy the type correctness of `n` when it is inherited in the subclass. We will see later that considerations involving subtyping can be used to determine which changes in types are guaranteed to preserve type safety.

Another important type-related question that arises with subclasses is determining whether the type of an object generated from a subclass is always a subtype of the type of an object generated from the superclass. While most current object-oriented languages have type systems that ensure this is the case, we will see that it need not hold if the language contains certain features that provide for more flexible constructions of subclasses.

These and other features of object-oriented languages have made it difficult to create statically typed object-oriented programming languages that are both very expressive and type safe. The following enumerates some of the strengths and weaknesses of the type-checking systems of some of the more popular object-oriented languages (or the object-oriented portions of hybrid languages[1]).

- Some show little or no regard for static typing (*e.g.*, Smalltalk).
- Some have relatively inflexible static type systems, requiring type casts to overcome deficiencies of the type system. These type casts may be unchecked, as in C++ and Object Pascal [Tes85], or checked at run time, as in Java.
- Some provide mechanisms like "`typecase`" statements to allow the programmer to instruct the system to check for more refined types than can be determined by the type system (*e.g.*, Modula-3 [CDG$^+$88], Simula 67 [BDMN73], and Beta [KMMPN87]).
- Some allow "reverse" assignments from superclasses to subclasses, which require run-time checks (*e.g.*, Beta, Eiffel [Mey92]).
- Some require that parameters of methods overridden in subclasses have exactly the same types as in the superclasses (*e.g.*, C++, Java, Object Pascal, and Modula-3), resulting in less flexibility than would be desirable, while

1. We consider a hybrid language to be one that attempts to support multiple paradigms. C++ and Object Pascal are examples of languages that attempt to support both procedural and object-oriented styles of programming.

others allow *too* much flexibility in changing the types of parameters or instance variables, requiring extra run-time or link-time checks to catch the remaining type errors (*e.g.*, Eiffel and Beta).

Thus all of these languages either require programmers to program around deficiencies of the type system, require run-time type checking, or allow run-time type errors to occur. Thus, there appears to be a lot of room for improvement in moving toward a combination of better security and greater expressiveness in the type systems. In the next section we provide several examples showing problems with current static type systems.

3.2 Simple type systems are lacking in flexibility

Languages like Object Pascal, Modula-3, and C++ arose as object-oriented extensions of imperative programming languages. These languages, as well as Java, have relatively simple and straightforward type systems whose features are similar to those of the procedural languages from which they were derived. In these simple type systems, the programmer has little flexibility in redefining methods in subclasses. They require that a redefined method have exactly the same type as the original method in the superclass. Similarly the types of instance variables may not be changed in subclasses. We refer to type systems that restrict the types of methods and instance variables in subclasses to be identical to those in superclasses as *invariant* type systems.

INVARIANT TYPE SYSTEM

Interestingly, in these invariant systems, when a method is inherited or redefined in the subclass, the programmer is often able to deduce more refined types for methods than the language allows to be written. For example the programmer may know that a certain method always returns an object of type `DType` even though the type system of the program restricts the programmer from writing that type as the return type because it does not allow changes to method types in subclasses. We present examples illustrating this below.

As mentioned earlier, we will find it helpful to keep the notions of class and type separate. The type represents only the public interface of the object (in our case the names and types of all of the methods, but not the instance variables), while the class includes names, types, and initial values for instance variables and names, types, and code for methods.

We will often use the convention in this book of writing `CType` for the type of objects generated by a class named `C` or `CClass`. In Chapter 2, we followed that convention in using `CellType` as the name of the type of objects generated by class `CellClass`.

3.2.1 The need to change return types in subclasses

In our first examples we show that it is useful to be able to modify the return types of methods when the methods are redefined in subclasses (and sometimes even when they are *not*).

In most pure object-oriented languages (*e.g.*, Eiffel, Java, and Smalltalk, as well as the languages we introduce later in this text), all objects are represented as references (*i.e.*, implicit pointers). Thus assignment results in sharing, rather than copying. In these languages, it is useful to have an operation that makes a new copy or clone of an object. A common way of supporting this is to provide a built-in `clone` method in the top-most class of the object hierarchy (called `Object` in Java), so that all other classes automatically inherit it. For the rest of this chapter we assume that our language has a top-most class named `Object`, and its type is `ObjectType`.

CLONE

A shallow copy is made by copying the values of instance variables and taking the same suite of methods as the original. If the instance variables hold references to other objects, only the references are copied, not the objects being referred to. Thus if this shallow clone method is applied to the head of a linked list, only the head node is copied while the rest of the list is shared between the new and old lists.

What should be the type of `clone`? When defined in the class `Object`, it seems apparent that it should return a value of type `ObjectType`. However, when this is inherited by a class `CellClass`, we would like it to return a value of type `CellType`. In the invariant type systems, the return type of `clone` remains `ObjectType`, even though the method actually returns a value that is a cell! That is, the semantics of the language does the correct thing, but the type system is not expressive enough to capture that. Instead the programmer must perform a type cast or other operation after the `clone` method has returned in order to allow the system to treat the value as having the proper type!

Often it is desirable to write a `deepClone` method that is based on `clone`. This is typically done by first writing code to send the `clone` message to `self` to make the shallow copy, and then writing code to clone all objects held in the instance variables of the original object.

Suppose we have a class `C` that includes a method `deepClone`, which returns an object of type `CType`. See Figure 3.1. Suppose we now define a subclass `SC` of `C` that includes a new method, `newMeth`, as well as a new instance variable, `newVar`, holding an object with type `newObjType`. We assume for simplicity that `newObjType` also supports a `deepClone` method.

We would like to redefine `deepClone` to clone the contents of this new instance variable after all of the code in the original `deepClone` method from `C` has been executed. (This is a quite common desire in real object-oriented languages as subclass methods cannot obtain access to private instance variables from the superclass.)

Unfortunately, the rules of the simple type systems require that `deepClone` for `SC` also return a `CType`, just as in `C`, even though it is obvious that it actually returns an object of type `SCType`. While this is not type-unsafe, it represents an unnecessary loss of information in the type system.

Suppose `anSC` is a variable of type `SCType`. If we write `(anSC ⇐ deepClone()) ⇐ newMeth()`, the type checker will complain, even though the object resulting from `anSC ⇐ deepClone()` has the method `newMeth`. The problem is that the type system is not aware of this!

In these circumstances, Object Pascal, C++, and Java programmers would normally be forced to perform a type cast to tell the compiler that the cloned object has the type `SCType`. In the case of Object Pascal and C++, the type cast is unchecked. In Java, it would be checked at run time. Modula-3 programmers would typically use a `typecase` statement that also performs a run-time check to get the same effect.

One could attempt working around these deficiencies in the static type system by making up a new name for the revised `deepClone` method (e.g., `SCdeepClone`). Unfortunately this would mean that inherited methods that included a message send of `deepClone` would call the old `deepClone` for `C` rather than the updated method from `SC` actually desired.

For example, suppose `C` includes a method `m` invoking `deepClone` as follows

```
function m(): Void is
{ ...
   self ⇐ deepClone()
   ...
}
```

Also suppose class `SC` is defined as a subclass of `C` as in Figure 3.1, except that it adds a new method `SCdeepClone` that returns a value of type `SCType`. If `sc` has type `SCType`, then `sc ⇐ SCdeepClone()` will certainly result in the newly defined clone method being called. However the execu-

```
class C {
   ...

   function deepClone(): CType is
   { self ⇐ clone(); ... }
}

class SC inherits C modifies deepClone {
   newVar:newObjType := nil;

   function newMeth(): Void is
   { ... }

   function setNewVar(newVarVal:newObjType): Void is
   { self.newVar := newVarVal }

   function deepClone(): SCType is
                  // illegal return type change!
                  // Must return CType instead
   var      // local variable declaration
      newClone:SCType := nil;
   {
      newClone := super ⇐ deepClone();
                  // (*) another problem
      newClone ⇐ setNewVar(newVar ⇐ deepClone());
      return newClone
   }
}
```

Figure 3.1 Typing `deepClone` methods in subclasses.

tion of `sc` ⇐ `m()` will result in the execution of the method `deepClone` from the superclass rather than the newly defined `SCdeepClone` that was likely intended. As a result of calling only the old method, the value in the new instance variable, `newVar`, will not be cloned, possibly causing problems later in the program.

Thus the restriction on changing types of methods in subclasses gets in the way of the programmer, even though the run-time system does the right thing. Not surprisingly, we have similar problems even writing down the type of the built-in (shallow) `clone`. In Java, `clone` is simply given a type indicating that it returns an element of the top class, `Object`. The result must then be cast to the appropriate type before anything substantial may be done with it.

In newer versions of C++, it is possible to specialize the return type of methods in subclasses. Thus method `deepClone` in `SC` could be specified to return a value of type `SCType` rather than `CType`. Unfortunately this does not solve all of our problems. First note that the right side of the assignment on line (*) of Figure 3.1 returns a value of type `CType`. Because the type of the variable on the left side is a subtype of `CType`, the assignment is illegal. (A value of the subtype can masquerade as a supertype, not the reverse!) Thus a type cast would have to be inserted to make the assignment legal.

Moreover, suppose class `SC` has a subclass `SSC` that adds new methods, but no new instance variables. As a result, there is no need to override method `deepClone`. However, if it is not overridden then it will continue to return type `SCType` rather than the desired `SSCType`. To get the types right, the programmer would have to override `deepClone` solely to cast the return type of the call of the superclass to the new type.

Thus, allowing covariant changes to the types of methods in subclasses would be helpful, but it would be even more helpful if some way could be found to have them change automatically. In the next section we show that it would also be convenient to be able to change the types of method parameters in subclasses.

3.2.2 Problems with binary methods

Our next class of examples is one that arises surprisingly frequently in practice. The particular typing problem arises in connection with what are often called *binary methods*. Binary methods are methods that have a parameter whose type is intended to be the same as the receiver of the message. Methods involving comparisons, such as `eq`, `lt`, and `gt`, or other familiar binary operations or relations are common examples of such methods (*e.g.*, `someElt` $\Leftarrow$ `lt(otherElt)`). In procedural languages, these methods would be written as functions that take two parameters (hence the "binary"). However, they are written with single parameters in object-oriented languages because the receiver of the message plays the role of the other pa-

BINARY METHODS

rameter. Other compelling examples arise in constructing linked structures (*e.g.*, `node` $\Leftarrow$ `setNext(newNext)`).

While it is not difficult to define binary methods when specifying classes from scratch, it is much more problematic for subclasses. Suppose we define class `C` below with method `equals`:

```
class C {
   ...
   function equals(other:CType): Boolean is
   { ... }
   ...
}
```

In the example we use our convention that `CType` is the type of objects generated from class `C`. If `o` is generated from class `C`, the signature of `equals` requires that the parameter `o’` in `o.equals(o’)` have type `CType` for the message send to be well-typed.

However, we have problems when we define a subclass `SC`:

```
class SC inherits C modifies equals {
   ...
   function equals(other:CType): Boolean is
       // Want parameter type to be SCType instead!
   { super ⇐ equals(other);
      ...  // Can’t access SC-only features in other
   }

   ...
}
```

Because we are not allowed to change the type of parameters in subclasses, we cannot change the type of `CType` to `SCType`, even though that may be what is desired here.

Similarly to our previous example with `SCdeepClone`, changing the name of the method to `newEquals` does not help. If there are occurrences of `equals` in the inherited methods of the superclass, we would like the new method body to be called, but instead the old would be invoked. Overloading the name `equals` does not help either. Overloading is resolved statically rather than at run time, so the problem of calling the wrong method is no different than using the new name above.

Programmers using languages with invariant type systems sometimes use the following trick of overriding equals in the subclass with a body that casts the argument to the desired type before it is used.

```
class SC′ inherits C modifies equals {
   ...
   function equals(other:CType): Boolean is
   var
     otherSC:SCType := nil;
   { otherSC := (SCType)other;    // type cast!
     ...
     return super ⇐ equals(other) & ... }

   ...
}
```

The expression `(SCType)other` represents casting the expression `other` to type `SCType`. However, these casts can fail at run time. This technique requires the programmer to be quite disciplined in adding casts to all overridden versions of binary methods. It also suffers from the twin disadvantages of adding run-time checks to each execution of a method, as well as requiring the programmer to handle the situation where the cast would fail. Clearly it would be a significant advantage both in programming and execution time to be able to check such calls statically.

A second example of the problems with binary methods arises from linked structures. Figure 3.2 contains a definition of the class, `Node`, which generates objects of type `NodeType` that form nodes for a singly linked list of integers. In the class there is one instance variable, `value`, for the value stored in the node, and another, `next`, to indicate the successor node. There are methods `getValue` and `setValue` to get and set the values stored in the node, and methods `getNext` and `setNext` to get and set the successor of the node.

Notice that method `getNext` returns a value of type `NodeType`, the type of object generated by class `Node`, while the `setNext` method takes a parameter of type `NodeType`. Thus the type `NodeType` is recursively defined. It is not uncommon to have such recursively-defined types in object-oriented languages.

Suppose we now wish to define a subclass of `Node`, `DoubleNode`, which implements doubly linked nodes, while reusing as much as possible the code for methods in `Node`. Figure 3.3 contains an attempt at defining such a

```
NodeType = ObjectType {
   getValue: Void → Integer;
   setValue: Integer → Void;
   getNext: Void → NodeType;
   setNext: NodeType → Void
}

class Node {
   value: Integer := 0;
   next: NodeType := nil;

   function getValue(): Integer is
   { return self.value }

   function setValue(newValue: Integer): Void is
   { self.value := newValue }

   function getNext(): NodeType is
   { return self.next }

   function setNext(newNext: NodeType): Void is
   { self.next := newNext }
}
```

Figure 3.2 Node class.

subclass, `DoubleNode`. `DoubleNode` adds to `Node` an additional instance variable, `previous`, as well as new methods to retrieve and set the `previous` node. If `DoubleNodeType` is the type of objects generated from the class, then we will want both the `next` and `previous` instance variables to have type `DoubleNodeType`. Similarly, the methods that get and set `next` or `previous` nodes should take parameters or return values of type `DoubleNodeType` rather than `NodeType`. This is particularly important because we do not want to allow the attachment of a singly linked node to a doubly linked node.[2]

2. The method `setPrev` is not really intended for public use because it only sets one of the two links. Normally it would be given a designation that would indicate this. Because we have not

```
DoubleNodeType = ObjectType {
   getValue: Void → Integer;
   setValue: Integer → Void;
   getNext: Void → NodeType;
   setNext: DoubleNodeType → Void;
   getPrev: Void → DoubleNodeType;
   setPrev: DoubleNodeType → Void
}

class DoubleNode inherits Node modifies setNext {
   previous: DoubleNodeType := nil;

   function getPrev(): DoubleNodeType is
   { return self.previous }

   function setPrev(newPrev: DoubleNodeType): Void is
   { self.previous := newPrev }

   function setNext(newNext: DoubleNodeType): Void is
            // error - illegal change to parameter type
   { super ⇐ setNext(newNext);
     newNext ⇐ setPrev(self) }
}
```

Figure 3.3 Doubly linked node class — with errors.

Unfortunately, in the simple type system described here, we have no way of changing these types, either automatically or manually, in the subclass. The class `DoubleNode` defined in the figure illegally changes the parameter type of method `setNext` in a covariant way. It is also troublesome that method `getNext` returns type `NodeType`, while `setNext` takes a parameter of type `DoubleNodeType`.

Suppose we create `LglDbleNode` as a legal subclass of `Node`. We might write it as shown in Figure 3.4. Problems become apparent when overriding the `setNext` method. We cannot send a `setPrev` message to the bare parameter `newNext`, since its declared type is `NodeType` rather than `LglD-`

yet introduced such features, we will ignore that minor point here.

bleNodeType. As a result the programmer must insert a cast to tell the type checker to treat it as though it has type LglDbleNodeType.

However, if a programmer sends setNext to an object generated from LglDbleNode with a parameter that is generated from Node, it will not be picked up statically as an error. Instead the cast will fail at run time.

Even if a variable dn has type LglDbleNodeType, the evaluation of

```
(dn ⇐ getNext()) ⇐ getPrev()
```

will generate a static type error, because the type checker can only predict that the results of dn ⇐ getNext() will be of type NodeType, not the more accurate LglDbleNodeType. Thus, even if the programmer has created a list, all of whose nodes are of type LglDbleNodeType, the programmer will still be required to write type casts to get the type checker to accept the program.

To get the desired types for the instance variable next and the methods getNext and setNext, we would instead have to define DoubleNode independently of Node, even though much of the code is identical. This is clearly undesirable.

We do not bother to show the code for an independently defined class, IndDoubleNode, with the same behavior as DoubleNode. We leave it as an exercise for the reader to write it and notice how similar the code is to that of Node. However, we show the corresponding object type, IndDoubleNodeType, in Figure 3.5. Even such an independent definition has problems. While we have not yet discussed the rules for determining when object types are in the subtype relation, the code in Figure 3.5 can be used to show that the resulting type, IndDoubleNodeType, cannot be a subtype of NodeType.

The function breakit in the figure is well-typed, as setNext takes a parameter of type NodeType, and the expression new Node as the actual parameter creates a value of type NodeType.

Suppose IndDoubleNodeType were a subtype of NodeType. If so, and if dn was a value generated from IndDoubleNode, then breakit(dn) would be well-typed, as values of type IndDoubleNodeType could masquerade as elements of type NodeType. However, that is not the case!

The execution of node ⇐ setNext(newNode) in the body of breakit would result in the message send of setPrev to the parameter newNext. But newNext holds a value that is generated from Node. This would result in a run-time type error as elements generated from Node have no setPrev method. This shows that object type IndDoubleNodeType could not be

```
LglDbleNodeType = ObjectType {
   getValue: Void → Integer;
   setValue: Integer → Void;
   getNext: Void → NodeType;
   setNext: NodeType → Void;
   getPrev: Void → LglDbleNodeType;
   setPrev: LglDbleNodeType → Void
}

class LglDbleNode inherits Node modifies setNext {
   previous: LglDbleNodeType := nil;

   function getPrev(): LglDbleNodeType is
   { return self.previous }

   function setPrev(newPrev: LglDbleNodeType): Void is
   { self.previous := newPrev }

   function setNext(newNext: NodeType): Void is
   { super ⇐ setNext(newNext);
     ((LglDbleNodeType)newNext) ⇐ setPrev(self) }
     // cast necessary to recognize setPrev
}
```

Figure 3.4 Legal doubly linked node class — with cast.

a subtype of `NodeType`, as elements of type `IndDoubleNodeType` cannot safely masquerade as elements of type `NodeType`.

This example is particularly troubling in that it seems to be tailor-made for the use of inheritance, but it (i) cannot be written correctly with an invariant type system, and (ii) would result in a type error if the type of objects generated from the desired subclass were a subtype of the type of objects generated from the original class. These problems are not special to the `Node` example, but arise with all binary methods because of the desire for a covariant change in the parameter type of binary methods.

We will later find a way to add expressiveness to languages in order to allow us to write `DoubleNode` as a subclass of `Node`. We will then need to

```
IndDoubleNodeType = ObjectType {
   getValue: Void → Integer;
   setValue: Integer → Void;
   getNext: Void → IndDoubleNodeType;
   setNext: IndDoubleNodeType → Void;
   getPrev: Void → IndDoubleNodeType;
   setPrev: IndDoubleNodeType → Void
}

function breakit(node: NodeType): Void is
{ node ⇐ setNext(new Node) }

var
   n: NodeType
   dn: IndDoubleNodeType
{
   n := new Node;
   dn := new IndDoubleNode;
   breakit(n);      // No problem
   breakit(dn)     // Run-time error here!
}
```

Figure 3.5 Example showing why `IndDoubleNodeType` cannot be a subtype of `NodeType`.

ensure that the resulting object types are *not* subtypes in order to avoid the second problem.

invariant type systems do have the desirable property that subclasses generate subtypes, but it may be worth losing subtyping in some circumstances if it makes it significantly easier to define desirable subclasses. We will evaluate these trade-offs when we examine the more flexible type system of Eiffel in the next chapter.

3.2.3 Other typing problems

In both of the examples above, the difficulties arose from an attempt to keep return or parameter types the same as those of the object being defined. While this is an extremely important special case, there are other examples

```
class CircleClass {
   center:PointType := nil;
   ...
   function getCenter(): PointType is
   { return self.center }
   ...
}

class ColorCircleClass inherits Circle
                       modifies getCenter {
   color:ColorType := black;
   ...
   function getCenter(): ColorPtType is { ... }
                     // illegal type change in subclass!
   ...
}
```

Figure 3.6 Circle and color circle classes.

where it is desirable to change a type in a subclass in a covariant way. In these examples, the type to be changed may have no relation to the type of objects generated by the classes being defined. Many examples of this phenomenon arise when we have objects with other objects as components.

Figure 3.6 contains the definition of a class, `CircleClass`, with a `getCenter` method that returns a point. If we define a subclass that represents a color circle, it would be reasonable to wish to redefine `getCenter` to return a color point, as in the code in the figure. This would be illegal by the rules on method types in these invariant object-oriented languages. We would like to have a typing system that allows such changes, as long as they are type safe.

Again, however, even if we allowed a change in the return type of `getCenter` in the subclass, we still have the problem that we cannot change the type of the instance variable `center`. If `center` is still of type `PointType`, we will either need to add a type cast (which may fail) to the body of `getCenter` or we may have to change the body to build a new color point from the `color` and `center` instance variables. The latter involves a lot of work each time `getCenter` is called, and is probably not worth the effort compared to separately returning the values of `center` (as a point) and `color`.

Finally, there likely will be a method `setCenter` in `CircleClass` that takes a parameter of type `PointType`. Even C++ will not allow changing the types of parameters of methods in subclasses, so `setCenter` in `Color-CircleClass` must accept parameters of type `PointType`. Thus if we wish `center` to have type `ColorPtType` in the subclass, we will have to add a dynamic check before assigning the value or live with the possibility that the value could fail to be a color point.

3.3 Summary of typing problems

In this chapter we illustrated several problems with invariant type systems. In each case the difficulty arose from a desire to change the types of methods that are modified in subclasses. However no changes to the types of methods are allowed in invariant type systems.

In order to help alleviate the rigidity in these simple systems, C++ allows changes to the return types of methods in subclasses. This provided some help with overriding `deepClone` and `getCenter` methods, but as we saw above, we are still left with significant problems to be overcome. Moreover, it provides no help in taking care of problems arising in examples involving binary methods, in particular those involved with redefining the type of the instance variable `next` and the parameter type of the method `setNext` in the `DoubleNode` subclass. In Chapter 16 we present a detailed design for an extension to invariant type systems that, when combined with parametric polymorphism, provides one possible solution to all of these problems.

Given these examples, readers may be wondering why statically typed object-oriented languages are so restrictive in not allowing covariant changes to the types of instance variables or the parameters of methods in subclasses. We put off this discussion until after we have discussed the rules for subtyping in chapter 5. For now we content ourselves with the explanation that we are simply examining the restrictions in the most popular existing statically typed object-oriented languages. In the next chapter we examine some more expressive type systems for object-oriented languages that will help overcome these difficulties.