The Development of Type Systems for Object-Oriented Languages

Kathleen Fisher *and John C. Mitchell †
Computer Science Department, Stanford University, Stanford, CA 94305
{kfisher,mitchell}@cs.stanford.edu

Abstract

This paper, which is partly tutorial in nature, summarizes some basic research goals in the study and development of typed object-oriented programming languages. These include both immediate repairs to problems with existing languages and the long-term development of more flexible and expressive, yet type-safe, approaches to program organization and design. The technical part of the paper is a summary and comparison of three object models from the literature. We conclude by discussing approaches to selected research problems, including changes in the type of a method from super class to sub class and the use of types that give information about the implementations as well as the interfaces of objects. Such implementation types seem essential for adequate typing of binary operations on objects, for example.

1 Introduction

A number of largely "theoretical" research efforts over the last five to ten years have developed and analyzed type systems for model object-oriented languages. This might seem strange to the object-oriented practitioner. Since the first object-oriented language, Simula [BDMN73], and the most widely-used language with object-oriented features, C++ [Str86, ES90], are both typed, it is natural to ask why new or different type systems are needed. One problem is that historically significant type systems, such as the Simula type system and the one developed by Borning and Ingalls for Smalltalk [BI82], have had significant type insecurities. This means that a well-typed program could, when executed, send a message to an object that has no method for responding to the message – a situation that the type system was intended to prevent. The second and less clear-cut problem is that there remain a large body of object-oriented programmers who consider the restrictions imposed by existing type systems too severe to allow true object-oriented programming. This is hard to find documented in the literature, but it seems to be a relatively common view of Smalltalk programmers toward C++ , for example. Finally, we hope that by identifying the essential features of object-oriented languages and studying their relationships, we will be able to design better programming languages in the future.

A basic insecurity in the original type system for Simula was the incorrect treatment of assignable locations and data structures. This can be illustrated using arrays. If `A` is a subclass of `B`, then an array of type `A` could be used in place of an array of type `B`. This makes sense if we are only reading values from the array. But if we are assigning to the array, then this allows a `B` object to be assigned to an array location designated to contain an `A` object, a breach of the type system that can easily be shown to result in a *method not understood* error at run-time. (Thanks to Alan Borning and Pavel Curtis for sending Simula programs demonstrating this bug; see [Cur90].) This problem is discussed in more detail in [Coo89b], where a much more subtle Eiffel bug is also described.

Some goals for future type systems can be described using an analogy, presented in the table below.

Conventional	Object-oriented
Lisp	Smalltalk
ML	??
C	C++

In the left column are three "conventional" languages. These are Lisp, which is an essentially pure untyped language for writing higher-order functional and imperative programs. At the bottom of the column is C, which is more widely used and more efficient, but suffers certain drawbacks of "machine-dependence" that any C programmer should be painfully aware of. In the middle is ML[Mil85, MTH90, MT91, Ulm94], which represents a typed alternative to Lisp, allowing many (but not all) Lisp-style programs to be statically typed. In comparison with C, ML offers a much richer set of type constructions and superior type security. However, it suffers some loss of efficiency because of garbage collection and the data representation used to support polymorphism. An approximate object-oriented analog of Lisp is Smalltalk, which is a pure, untyped language that is widely used, but sometimes considered too inefficient for certain tasks. To continue this inexact analogy, we can say that the C of object-oriented programming is C++ . The most promising research area in object-oriented programming seems to be to explore the middle-ground between Smalltalk and C++ . A specific research goal that may help focus the readers attention is to find an object-oriented analog of ML: a statically-typed, object-oriented language allowing the most common or most important Smalltalk idioms with greater flexibility than C++ .

* Supported in part by a Fannie and John Hertz Foundation Fellowship and NSF Grant CCR-9303099.

† Supported in part by NSF Grant CCR-9303099 and the Wallace F. and Lucille M. Davis Faculty Scholarship.

The same basic understanding that would be critical to the design of an "object-oriented ML" could also lead to improvements in existing typed languages, a type system for Smalltalk, or higher-level system design tools. Such tools could facilitate both modular design and static error detection in programs written in existing and future languages. (The reason that types are critical to modular system design is that the interfaces to system components are expressed using types.) Given the importance of both error detection and systematic program design, as well as the increasing use of object-oriented languages and ideas, object-oriented type systems seem an important and practical research direction.

This paper is a mildly revised and expanded version of a document intended to provide background and reference material for the second author's invited Sendai lecture, contemporaneous with the 1994 TACS conference [FM94]. Although some references were updated in 1995, it was not feasible to account for research completed after early 1994.

The main goal of the paper is to make the motivating problems clear to specialists and non-specialists, so that those with an interest in research in this area might find it possible to choose problems and non-specialists might develop an appreciation for this research area. The contents, by section, are:

- Brief overview of object-oriented concepts.
- Example comparing object-oriented approach with type-case statements.
- Type-theoretic preliminaries.
- Introduction to three theoretical object models.
- Recursive record model.
- Existential types model.
- An axiomatic model with object-based inheritance.
- Summary comparison chart
- Open problems and directions for future research

A general reference on type systems and objects is the collection [GM94].

2 What is Object-Oriented Programming?

"Object orientation" is both a language feature and a design methodology. The design methodology seems to have evolved from the simulation approach of Simula. A brief outline of this methodology is given in the following list, based on [Boo91], one of many current books on object-oriented design.

Methodology:

1. Identify the objects at a given level of abstraction.
2. Identify the semantics (intended behavior) of these objects.
3. Identify the relationships among the objects.
4. Implement these objects.

This methodology is an iterative process based on associating objects with components or concepts in a system. One reason the process is iterative is that one object is typically implemented using a number of "sub-objects," just as in top-down programming, a procedure is typically implemented by a number of finer-grained procedures. A difference is that both functionality and data representation may be refined in the design process. In the early examples illustrating top-down programming (see [Dij72]), the data structures were very simple and remained invariant under refinements of the program. This is not typical of more complex systems.

An important characteristic of objects is that they provide a uniform interface to all components of a system. In particular, an object can be as small as a single integer or as large as a file system or an output device. Regardless of its size, all interactions with an object occur via message send. This use of objects to hide implementation details and provide a "black box" capability is useful for the same reasons that data and procedural abstraction are useful.

There are two main flavors of object-oriented languages: class-based and object-based. In class-based languages, the implementation of an object is specified by its class. In such languages, objects are created by instantiating their classes. In object-based languages, objects are defined directly from other objects by adding new methods via *method addition* and replacing old methods via *method override*. Some important features of object-oriented languages are encapsulation, dynamic lookup, subtyping, and inheritance. These features are explained in the following subsections.

2.1 Encapsulation

Objects are used in most object-oriented programming languages to provide an encapsulation barrier. Typically, an object contains some private data that can be accessed only by its methods. Such encapsulation helps insure that programs can be written in a modular fashion and that the implementation of an object can be changed without forcing changes in the rest of the system. These benefits are the same as those realized by abstract data types (ADT's). However, objects have additional benefits not provided by ADT's when we wish to use related data abstractions in similar ways. We illustrate this point with the following example involving queues.

A typical language construct for defining an abstract data type is the ML `abstype` declaration, which we use in Appendix A to define queues. In this example, a queue is represented by a list. However, only the functions given in the declaration may access the list. This allows the invariant that list elements appear in first-in/first-out order to be maintained, regardless of how queues are used in any client program.

A drawback of the kind of abstract data types used in ML and other languages such as CLU [LSAS77, L^+81] and Ada [US 80] becomes apparent when we consider a program that uses both queues and priority queues. For example, suppose that we are simulating a system with several "wait queues", such as a bank or hospital. In a teller line or hospital billing department, customers are served on a first-come, first-served basis. However, in a hospital emergency room, patients are treated in an order that takes into account the severity of their injuries or ailments. Some aspects of this kind of "wait queue" are modeled by the abstract data type of priority queues, shown in Appendix A.

Note that the signature of priority queues is the same as for ordinary queues: both have the same number of operations, and each operation has the same type, except for the difference between the type names `pqueue` and `queue`. However, if we declare both queues and priority queues in the same scope, the second declarations of `is_empty, add, first, rest,` and `length` will hide the first. This will require us to rename them, say as `q_is_empty, q_add, q_first, q_rest, q_length` and `pq_is_empty, pq_add, pq_first, pq_rest, pq_length.`

In a hospital simulation program, for example, we might like to treat priority queues and ordinary queues uniformly. For example, we might wish to count the total number of people waiting in any line in the hospital. To do this, we would like to have a list of all the queues (both priority and ordinary) in the hospital and go down the list asking each queue for its length. But if the `length` operation is different for queues and priority queues, we have to decide whether to call `q_length` or `pq_length`, even though the correct operation is uniquely determined by the data. This shortcoming of ordinary abstract data types is eliminated by dynamic method lookup for objects.

2.2 Dynamic Lookup

In any programming language with objects, there is some syntax for invoking an operation associated with an object. In Smalltalk this is called "sending a message to an object," while the C++ terminology is to "call a member function of the object." To give a neutral syntax, we write

$$\texttt{receiver} \Leftarrow \texttt{operation}$$

for invoking `operation` on the object `receiver`. There are two main views for what this means (operationally):

1. Consider objects as tables that associate a method (function body) with each message (operation or function name).
2. Consider a message name as an identifier of some kind of "overloaded" function that is not "part" of the object.

In the first view, an object is a collection of methods (member functions) and data. When an object is sent a message at run-time, the appropriate method (function body) is invoked. As a result, sending the same message to different objects may result in the execution of different code. This is called *dynamic lookup,* or, variously, dynamic binding, dynamic dispatch, or runtime dispatch.

In the second view, a message is the name of a kind of "overloaded function," where overloading is resolved at run-time, instead of compile-time. The important characteristic is that sending the same message to different objects may result in execution of different code. In this view, it is possible to take more than the first argument into account. For example, if we write

$$\texttt{receiver} \Leftarrow \texttt{operation}(\texttt{arguments})$$

for invoking an operation with a list of arguments, then the actual code invoked may depend on the receiver alone (as in the first view above), or on the receiver and one or more arguments. When the selection of code depends only on the receiver, it is called *single dispatch;* when it also depends on one or more arguments, it is called *multiple dispatch.*

Multiple dispatch is used in Common Lisp [Ste84]. A theoretical study of multiple dispatch appears in [CGL92].

Although multiple dispatch is in some ways more general, there seems to be some loss of encapsulation. Specifically, in order to define a function on different kinds of arguments, the function must typically have access to the internal data of each function argument. The main problem seems to be that while a function or method belongs to a single object or class of objects in a single dispatch language, a function or method does not obviously belong to any particular object or class in the multiple dispatch approach. It is not clear that the loss of encapsulation is inherent to multiple dispatch, but current multiple dispatch systems do not seem to offer any reasonable encapsulation of private or local data for objects. Recent work addressing these issues appears in [CL94].

2.3 Objects and closures

An interesting and historically significant explanation of Simula objects is that they are simply activation records of procedures that are left on the "stack" after the procedure terminates [BDMN73]. More precisely, when a procedure is called, an activation record is allocated that contains space for the local variables of the procedure and pointers to locally-declared functions. In an ordinary Algol-like language with block structure, this record is popped off the run-time stack when the procedure returns. However, it is possible to instead return a pointer to the activation record, and abandon stack storage management. (In this case, the activation record should be deallocated when there are no active pointers to it.) The result is precisely a Simula object, as explained in [BDMN73]. Since a closure is a function (or other value) together with its environment (activation record), objects and closures are essentially the same thing. An interesting example in this regard is the T object system [RA82, AR88].

Based on this analysis, one might wonder why we are not concerned with "closure-oriented programming." More precisely, we might ask what object-oriented languages provide that languages with closures, such as Lisp, Scheme or ML, do not. The answer is that object-oriented languages provide subtyping and inheritance, which are not directly supported in either Lisp or ML. Another way of looking at the difference is that if there is only one class of objects in a program, the class could be replaced by a function that returns closures. The differences between closures and objects emerge only when a second class is defined from the first using inheritance, and/or objects of a second class are used in place of the first via subtyping.

In passing, we note that while Simula had subtyping and inheritance in the late 1960's, encapsulation was a later addition. More precisely, all of the local declarations that constitute an object were accessible through the "dot notation." This accessibility meant Simula could not guarantee desired invariants of its objects. For example, it could not insure that a list of elements in a queue object were kept in first-in/first-out order, since malicious programs could have directly accessed and changed the queue's local state.

2.4 Object types

There are two forms of type we might give to objects. The first is a type that simply gives interface information. The second is an interface plus some implementation constraints. In the first case, the elements of the type will be all objects that have the given interface. In the second case, it will only be those elements that also have a certain representation. Since the first is more basic, we begin by discussing interface types. The following example uses the syntax of Rapide, an experimental language designed for prototyping software and mixed software/hardware systems [BL90a, MMM91, KLM94, KLMM94].

```
type Point is interface
   x_val : Int;
   y_val : Int;
   distance : Point -> Int;
end interface;
```

Objects of type `Point` must have two integer methods, called `x_val` and `y_val`, and a function method called `distance`. A function computing the distance between two points requires only one argument, since the function belongs to a particular point and therefore may compute the distance between the point passed as an actual parameter and the particular point to which the function belongs. In other words, the intended use of the `distance` method of a point object p is to compute the distance between p and another point object q, by a call of the form $\mathtt{p} \Leftarrow \mathtt{distance(q)}$. Of course, since the interface gives only signature information, the `distance` method is not forced to compute the distance between two points. If we wish to specify that `distance` must compute distance, then a more expressive form of specification must be added to the interface.

One significant feature of this interface is that the type name `Point` appears in the interface itself. The simplest explanation is that `Point` is a recursively-defined type or, equivalently, a fixed-point of some function from types to types. In formalizing a calculus of objects and their types, we therefore introduce functions from types to types of a special form, which we call object interfaces. In particular, we use the syntax $\{\!|\mathtt{m}_1{:}\mathtt{A}_1, \ldots, \mathtt{m}_k{:}\mathtt{A}_k|\!\}$ for the interface type specifying methods $\mathtt{m}_1, \ldots, \mathtt{m}_k$ of types $\mathtt{A}_1, \ldots, \mathtt{A}_k$, respectively. Using this notation, we may write a function from types to interface types in the form

$$\lambda \mathtt{t} : \mathit{Type}.\{\!|\mathtt{m}_1 : \mathtt{A}_1, \ldots, \mathtt{m}_k : \mathtt{A}_k|\!\}$$

We will often omit the specification $:\mathit{Type}$, with the understanding that lower case letters $\mathtt{r}, \mathtt{s}, \mathtt{t}$, possibly with subscripts, primes, etc., designate types.

Although it is tempting to consider `Point` as the least fixed-point of the type function

$$\lambda \mathtt{t}.\{\!|\mathtt{x\_val} : \mathtt{Int}, \mathtt{y\_val} : \mathtt{Int}, \mathtt{distance} : \mathtt{t} \rightarrow \mathtt{Int}|\!\}$$

we will take a more "abstract" approach in this paper. One reason is that this approach allows us to compare three theoretical models in a uniform way. Specifically, we assume some type functional

$$\mathbf{Object} : (\mathit{Type} \Rightarrow \mathit{Type}) \Rightarrow \mathit{Type}$$

and write

$$\mathtt{Point} \;=\; \mathbf{Object}(\lambda \mathtt{t}.\{\!|\mathtt{x\_val} : \mathtt{Int}, \mathtt{y\_val} : \mathtt{Int}, \mathtt{distance} : \mathtt{t} \rightarrow \mathtt{Int}|\!\})$$

We will often use **Object** as a binding operator and write $\mathbf{Object}\,\mathtt{t.A}$ for $\mathbf{Object}(\lambda\mathtt{t.A})$. This convention gives us the syntax

$$\mathbf{Object}\,\mathtt{t}.\{\!|\mathtt{m}_1 : \mathtt{A}_1, \ldots, \mathtt{m}_k : \mathtt{A}_k|\!\}$$

for an object interface. Objects that satisfy this interface will have methods $\mathtt{m}_1, \ldots, \mathtt{m}_k$ with types $\mathtt{A}_1, \ldots, \mathtt{A}_k$, respectively.

An alternate form of object type is an interface with some additional guarantees about the form of the implementation. We will see that this is important for objects with binary operations, for example. For these types, subtyping must take into account both interface subtyping and compatibility of implementations. Since the implementation of an object is intended to be hidden, the second form of type should not give any explicit information about the implementation. Instead, it appears that "implementation types" are properly treated as a form of partially-abstract types. This is a current research topic, with some of the basic ideas explained in [CW85, KLM94, PT93] using bounded existential types. We will return to this in the final sections of the paper.

2.5 Inheritance

Perhaps the most common confusion surrounding object-oriented programming is the difference between subtyping and inheritance. Inheritance is an implementation technique. For every object or class of objects defined using inheritance, there is an equivalent definition that does not use inheritance, obtained by expanding the definition so that inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating code, and that when one class is implemented by inheriting from another, changes to one affect the other. This has a significant and sometimes debated impact on maintenance and modification. One reason why subtyping and inheritance are confused is that some class mechanisms combine the two. A typical example is C++ , where `A` will be recognized by the compiler as a subtype of `B` only if `A` is defined as a subclass of `B`. However, this is an elective design decision; there seems to be no inherent reason for linking subtyping and inheritance in this way. Some interesting and relevant examples may be found in [Sny86, Coo92].

Using a neutral notation, we can illustrate inheritance of the form that appears in most object-oriented languages by a simple example. The two classes that appear in 1 define objects with private data `v` and public methods `f` and `g`. The class `B` is defined by reusing the declarations of `A`, but redefining the function `g`.

The simplest, but not most efficient, implementation of inheritance is to incorporate the relationship between classes explicitly in the run-time representation of objects. For the example classes `A` and `B` above, this is shown in Figure 2.

This figure shows data structures representing (i) the `A` class, with pointers to the `A` template and method dictionary, (ii) the `A` template, which gives the names of data associated with each `A` object, (iii) the `A` method dictionary, which gives the names of methods associated with each `A` object, (iv) the `B` class, with pointers to the `B` template, `B` method dictionary and base class `A` (v) the `B` template, (vi) the `B` method dictionary, and (vii) a `B` object `b`. Each class template shows the names and order

```
class A =
    private
        val v =  ...
    public
        fun f(x)  = ... g(...) ...
        fun g(y)  = ... f(...) ...
    end;

class B = extend A with
    public
        fun g(y)  = ...
    end;
```

Figure 1: Simple example illustrating inheritance.

of data for object of that class, while the method dictionary contains names and pointers to code for methods.

We can see how this works by tracing how the expression $\mathtt{b} \Leftarrow \mathtt{f}()$ is evaluated at run-time. The sequence of events is:

1) The method dictionary for `B` objects is found by looking in the `B` class.

2) The `B` method dictionary is searched for method name `f`.

3) Since `f` is not there, the method dictionary for base class `A` is searched.

4) The function `f` is found in the `A` method dictionary.

5) When the body of `f` refers to `g`, the search for the `g` method begins again with the `b` object, guaranteeing that the `g` function defined for class `B` is used.

This implementation may be optimized in several ways. The first is to cache recently-found methods. Another possibility is to expand the method tables of derived classes to include the method tables of their base classes. This expansion eliminates the upward search through the method dictionaries of more than one class. Since the dictionaries contain only pointers to functions, this duplication does not involve a prohibitive space overhead.

A more significant optimization may be made in typed languages such as C++ , where the set of possible messages to each object can be determined statically. If the method dictionaries, or *vtables* in C++ terminology, can be constructed so that all objects of a given type have all their methods in the same relative position in the vtable, then the offset of a method within the vtable can be computed at compile-time. This optimization, which is possible in C++ , reduces the cost of method lookup to a simple indirection without search, followed by ordinary function call. For more information, see [ES90, Section 10.7c].

2.6 Subtyping

The basic principle associated with subtyping is *substitutivity:* if `A` is a subtype of `B`, then any expression of type `A` may be used without type error in any context that requires an expression of type `B`. We will write "`A` <: `B`" to indicate that `A` is a subtype of `B`.

The primary advantage of subtyping is uniform operation over various types of data. For example, subtyping

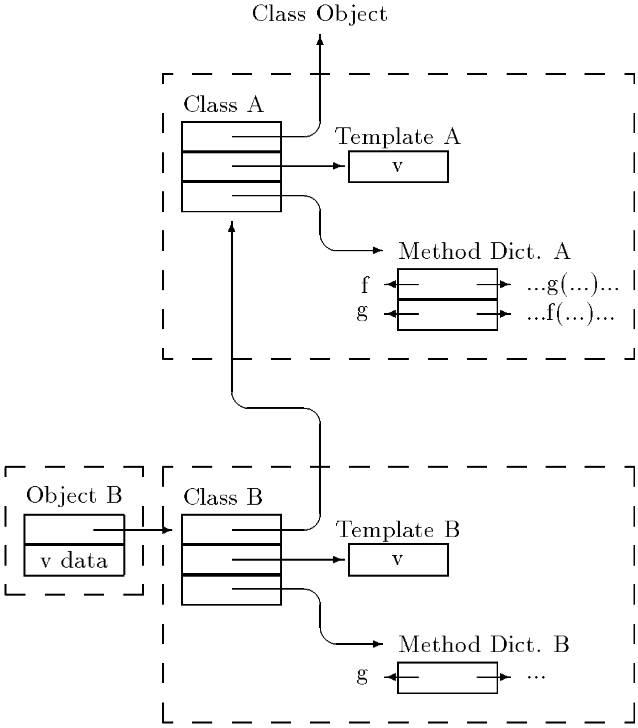

Figure 2: Smalltalk-style representation of `B` object inheriting from class `A`.

makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some type. Consider as an example a queue containing various bank accounts to be balanced. These accounts could be savings accounts, checking accounts, investment accounts, etc., but each is a subtype of `bank_account` so balancing is done in the same way for each. This uniform treatment is generally not possible in strongly typed languages without subtyping.

Another advantage of subtyping in an object-oriented language is to allow functionality to be added with minimal modification to the system. If objects of a type `B` have some behavior which roughly approximates a desired behavior, then we may want to replace objects of type `B` with objects of another type `A` that better approximates the desired behavior. In many cases, the type `A` will be a subtype of `B`. By designing the language so that substitutivity is allowed, one may add functionality in this way without any other modification to the original program.

If the type of an object is its interface, then subtyping for object types is "compatibility" or "conformance" of interfaces. More specifically, if one interface provides all of the components of another, with compatible types, then every object of the first type should be substitutable for the other. This restriction gives us subtyping of the form:

$$\mathbf{Object}\,\mathtt{t}.\{\!\!\{\mathtt{x}:\mathtt{Real},\ \mathtt{y}:\mathtt{Real},\ \mathtt{c}:\mathtt{Color}\}\!\!\} \\ \quad <:\ \mathbf{Object}\,\mathtt{t}.\{\!\!\{\mathtt{x}:\mathtt{Real},\ \mathtt{y}:\mathtt{Real}\}\!\!\}$$

We call this form of subtyping "width" subtyping. In addition, it is generally possible to restrict the type of one or more components to a subtype. This gives us subtyping of the following form, which we call "depth" subtyping:

$$\mathbf{Object}\,t.\ \{\!\!\{\mathtt{x}:\mathtt{Int},\ \mathtt{y}:\mathtt{Int}\}\!\!\} \\ \quad <:\ \mathbf{Object}\,\mathtt{t}.\{\!\!\{\mathtt{x}:\mathtt{Real},\ \mathtt{y}:\mathtt{Real}\}\!\!\}$$

if we assume that $\mathtt{Int} <: \mathtt{Real}$.

Combining these two, we have

$$\mathbf{Object}\,\mathtt{t}.\ \{\!\!\{\mathtt{x}:\mathtt{Int},\ \mathtt{y}:\mathtt{Int},\ \mathtt{c}:\mathtt{Color}\}\!\!\} \\ \quad <:\ \mathbf{Object}\,\mathtt{t}.\{\!\!\{\mathtt{x}:\mathtt{Real},\ \mathtt{y}:\mathtt{Real}\}\!\!\}$$

The situation is more subtle with object types of the form $\mathbf{Object}\,\mathtt{t}.\{\!\!\{\mathtt{m}_1:\mathtt{A}_1,\ldots,\mathtt{m}_k:\mathtt{A}_k\}\!\!\}$, where `t` appears in $\mathtt{A}_1,\ldots,\mathtt{A}_k$. This issue is discussed in Sections 6 through 8.

2.7 Method Specialization

It is relatively common for one or more methods of an object to take objects of the same type as parameters or return objects of the same type as results. For example, consider points with the following interface.

```
type Point is interface
   x    : Int
   move : Int -> Point
   eq   : Point -> Bool
end interface;
```

(For simplicity, we drop the `y` coordinate and work with one-dimensional points.) The `move` method of a point `p` returns a point. Similarly, the `eq` method takes as a parameter an object of point type.

When colored points are defined in terms of points, it is desirable that the types of the methods be specialized to return or use colored points instead of points. Otherwise, we effectively lose type information about the object we are dealing with whenever we send the `move` method, and we are restricted to using only point methods when comparing colored points for equality. If it is possible to inherit a `move` method defined for points in such a way that the resulting method on colored points has type $\mathtt{Int} \rightarrow \mathtt{Colored\_Point}$, then we say that *method specialization* occurs. This form of method specialization is called "mytype" specialization because the type that changes is the type of the object that contains the methods [Bru92, Bru93].

It is also meaningful to specialize types other than the type of the object itself when defining a derived class.

Method specialization is generally not provided in existing typed object-oriented languages, but it is common to take advantage of method specialization (in effect) in untyped object-oriented languages. Therefore, if we are to devise typed languages that support useful untyped programming idioms, we should devise type systems that support method specialization.

3 Example

In this section, we give an extended example, which is a portion of a program manipulating several kinds of geometric shapes. There are two versions of this program, one with classes and the other without. Without classes, we use records (or `struct`'s) to represent each shape. For each operation on shapes, we have a function that tests the type of shape passed as an argument and branches accordingly. We illustrate this using a C program, with each shape represented as a `struct` (analogous to Pascal or ML record). We will refer to this program, which appears in Appendix B as the "typecase" version, since each function is implemented by a case analysis on the types of shapes. For brevity, the only shapes are circles and rectangles.

We can see the advantage of object-oriented programming by rewriting the program so that each object has the shape-specific operations as methods. This version of the program appears in Appendix C.

Some observations:

- We can see the difference between the two program organizations in the following matrix. For each function, `center`, `move`, `rotate` and `print`, there is code for each kind of geometric shape, in this case `circle` and `rectangle`. Hence we get eight different pieces of code.

class	*function*			
	center	move	rotate	print
circle	c_center	c_move	c_rotate	c_print
rectangle	r_center	r_move	r_rotate	r_print

In the "typecase" version, these are arranged by column, while in the class-based program, they are arranged by row. Each arrangement could be considered to have some advantages when it comes to program maintenance and modification. In the object-oriented approach, adding a new shape is straightforward. The code detailing how the new shape should respond to the existing operations all goes in one place: the class definition. Adding a new operation is more complicated, since the appropriate code must be added to each class definition, which could be spread throughout the system. In the "typecase" version, the reverse situation is true. Adding a new operation is relatively easy, but adding a new shape is difficult.

- There is a loss of encapsulation in the typecase version, since the data manipulated by `rotate, print` and other functions has to be publicly accessible. In contrast, the object-oriented solution encapsulates the data together with the functions.

- The "typecase" version cannot be statically type checked in C. It could be in a language where there was a built-in "typecase" statement which tested the type of an object instead of a data field that might or not give the correct type. An example is the Simula `inspect` statement.

```
class A;
A class B;  /* B is a subclass of A */
ref (A) a;
inspect a
  when B do   /* some operation
                 on subclass B */
  otherwise   /* operations from
                 superclass a */
```

(See also Pascal variant records or ML datatypes, both a form of disjoint or tagged unions.)

This approach would require that every object be tagged with type, which is about the same amount of space overhead as making each into an object. (At least in comparison with the C++ optimization, where the overhead of treating something as an object is approximately one extra pointer per datum.)

- In the typecase version, "subtyping" is used in an ad hoc manner. We coded circle and rectangle so that they have a shared field in their first location. This is a hack to implement a tagged union that could be avoided in a language providing disjoint (as opposed to C unchecked) unions.

- The complexity of the two programs is roughly the same. In the "typecase" version, there is the space cost of an extra data field (the type tag) and the time cost, in each function, of branching according to type. In the "object" version, there is a hidden class or `vtbl` pointer in each object, requiring essentially the same space as a type tag. In the optimized C++ approach, there is one extra indirection in determining which method to invoke, which corresponds to the case analysis in the "typecase" version. A Smalltalk-like implementation would be less efficient in general, but for methods that are found immediately in the subclass method dictionary (or via caching), the run-time efficiency may be comparable.

4 Type-theoretic framework

The calculi used to explain and model objects are generally fragments of higher-order polymorphic lambda calculus, extended with subtyping and a few extra constructs. Some type binding operator is needed to represent object interface types. In the recursive record model, this is ordinary type recursion, while the existential model uses existential type quantification. Although several forms of records may be reduced to higher-order polymorphism, as demonstrated in [Car94], it is generally simpler to work with a basic calculus that has records with some form of record extension operation. In this section, we review by simple example many of the basic type-theoretic concepts that play a role in the study of objects. Some good references are [CW85] and the appendix of [PT94]). Due to space constraints, we will simply list the basic constructs by example or illustrative rules.

4.1 Subtyping

4.1.1 Cartesian Products

For products, we have coordinate-wise subtyping.

$$(\times\ <:)\qquad \frac{A_1 <: B_1, \ldots, A_k <: B_k}{A_1 \times \ldots \times A_k <: B_1 \times \ldots \times B_k}$$

4.1.2 Records

There are two sources of record subtyping, which we refer to as "width" (making the record wider) and "depth" (subtyping by component). The second one corresponds to the cartesian product rule above. These two may be combined to obtain the third rule. These rules appear in Figure 3

4.1.3 Functions

Since object methods are typically functions, function subtyping enters into object subtyping. The relevant typing rule is the following:

$$\frac{A <: B,\ C <: D}{B \rightarrow C <: A \rightarrow D}$$

Note that if we speak of the subtype relation as a partial- (or quasi-) order, this rule says that the function type constructor $\rightarrow$ is monotonic in its second argument, but anti-monotonic in its first argument. The anti-monotonicity is the source of many complications in typing algorithms and semantic constructions.

4.2 Recursive types

Recursive types are defined by the μ binding operator. In particular, the type that satisfies the recursive type equation $t = F[t]$ is denoted by $\mu t.F[t]$. The equational rule for recursive types is:

$$\mu t.A = [\mu t.A/t]A$$

where the notation $[\mu t.A/t]A$ denotes substituting expression $\mu t.A$ for all free occurrences of t in A, renaming bound variables as necessary to avoid capture. Note that this rule permits recursive types to be wound and unwound implicitly. (Such implicit winding is not always

$$(width) \qquad \{\ell_1 : A_1, \ldots, \ell_k : A_k, \ldots \ell_n : A_n\} <: \{\ell_1 : A_1, \ldots, \ell_k : A_k\}$$

$$(depth) \qquad \frac{A_1 <: B_1, \ldots, A_k <: B_k}{\{\ell_1 : A_1, \ldots, \ell_k : A_k\} <: \{\ell_1 : B_1, \ldots, \ell_k : B_k\}}$$

$$(both) \qquad \frac{A_1 <: B_1, \ldots, A_k <: B_k}{\{\ell_1 : A_1, \ldots, \ell_k : A_k, \ldots, \ell_n : A_n\} <: \{\ell_1 : B_1, \ldots, \ell_k : B_k\}}$$

Figure 3: Record subtyping rules

permitted, c.g. [AC94]) The subtyping rule for recursive types is given as follows:

$$\frac{s <: t \triangleright A <: B}{\mu s.A <: \mu t.B}$$

which says that $\mu s.A <: \mu t.B$ if we may show that $A <: B$ under the assumption that $s <: t$. For a more detailed discussion, see [AC91].

4.3 Polymorphism

4.3.1 Parametric polymorphism

To write functions that work for elements of any type, we can use type abstraction, as provided in higher-order lambda calculus. For example, the following function

$$id \ ::= \ \lambda t : T.\, \lambda x : t.\, x \ : \ \forall t.\, t \rightarrow t$$

can be thought of as the identity function on all types, since when it is applied to any type A, it returns the identity function on A.

4.3.2 Subtype polymorphism

Although parametric polymorphism is quite useful, it can only be used to write functions that work for any type. In object-oriented programming, we are interested in writing functions that work for any subtype of the type for which they were originally intended. One way to provide this flexibility is through the following rule, which permits us to promote the type of an element to any of its supertypes:

$$(subsum) \qquad \frac{p : A, \quad A <: B}{p : B}$$

This process is called subsumption. Using this rule, we can apply a function f written for elements of type A to elements of type B, if $A <: B$:

$$\frac{f : A \rightarrow C, \quad x : B, \quad B <: A}{f(x) : C}$$

Note that the static type of $f(x)$ does not depend on the type of x, with this mechanism, even though the actual run-time value of $f(x)$ may.

4.3.3 Bounded universal quantification

To write functions that work for any subtype of a particular type and that can specialize the result type depending on the their argument types, we consider bounded universal quantification. The basic rule is

$$(\forall\ bd) \qquad \frac{p : \forall t <: A.\, B, \quad C <: A}{p[C] : [C/t]B}$$

Here a side condition requires that A does not contain t free. (Contrast this condition with the rule for F-bounded universal quantification, given below). The subtyping rule for bounded universal quantification has the consequence:

$$\frac{f : \forall t <: A.\, t \rightarrow t, \quad x : B, \quad B <: A}{f[B](x) : B}$$

Here f is defined to work on any subtype of A, and the return type of f is specialized to B when it is used on elements of B. Unfortunately, as explained in [BL90b], the only element of type $\forall t <: A.\, t \rightarrow t$ is the identity function. One recent approach to solving this problem, that differs from the one discussed below, appears in [HP95].

4.3.4 F-Bounded universal quantification

To get more than identity functions, we need to be able to provide a more general bound. In particular, we will consider all types t that are subtypes of some type $F[t]$, where F is a type functional. F is used to specify the interface of objects for which the function is supposed to work. The basic rule is

$$\frac{p : \forall t <: F[t].\, B, \quad C <: F[C]}{p[C] : [C/t]B}$$

which has the consequence

$$\frac{\begin{array}{l} equal : \forall t <: \{\!|\mathtt{eq} : t \rightarrow \mathtt{Bool}|\!\}.\, t \times t \rightarrow \mathtt{Bool}, \\ x : \mathbf{Object}\, t.\{\!|\mathtt{eq} : t \rightarrow \mathtt{Bool}, \mathtt{inc} : \mathtt{Unit} \rightarrow t, \ \ldots|\!\} \end{array}}{\begin{array}{l} equal[\mathbf{Object}\, t.\{\!|\mathtt{eq} : t \rightarrow \mathtt{Bool}, \\ \qquad \mathtt{inc} : \mathtt{Unit} \rightarrow t, \ldots|\!\}](x, x) : \mathtt{Bool} \end{array}}$$

assuming

$$\begin{array}{l} \mathbf{Object}\, t.\{\!|\mathtt{eq} : t \rightarrow \mathtt{Bool}, \mathtt{inc} : \mathtt{Unit} \rightarrow t, \ \ldots|\!\} <: \\ \quad \{\!|\mathtt{eq} : \mathbf{Object}\, t.\{\!|\mathtt{eq} : t \rightarrow \mathtt{Bool}, \\ \qquad \mathtt{inc} : \mathtt{Unit} \rightarrow t, \ \ldots|\!\} \rightarrow \mathtt{Bool}|\!\} \end{array}$$

In this example, the type function

$$\lambda t.\{\!|\mathtt{eq} : t \rightarrow \mathtt{Bool}|\!\}$$

plays the role of F in the typing rule given above. This typing derivation holds in the recursive record model, for example, where $\mathbf{Object}\, t$ is $\mu\, t$.

4.3.5 Higher-Order bounded universal quantification

Another approach to providing the flexibility of F-Bounded universal quantification that requires slightly weaker assumptions uses higher-order bounded universal quantification. The basic rule is

$$\frac{p : \forall g <\!\!: F.\, B, \quad G <\!\!: F : Type \Rightarrow Type}{p[G] : [G/g]B}$$

In this rule, the notation $<\!\!:$ represents a subtyping relationship between type functions. This relationship is defined by extending the subtyping relation on types pointwise to type functions. This typing rule has the consequence

$$\frac{\begin{array}{l} equal : \forall g <\!\!: \lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}|\!\}. \\ \qquad (\mathbf{Object}\, t.g\, t) \times (\mathbf{Object}\, t.g\, t) \to \mathtt{Bool}, \\ x : \mathbf{Object}\, t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}, \mathsf{inc} : \mathtt{Unit} \to t, \ \ldots|\!\} \end{array}}{\begin{array}{l} equal[\lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}, \\ \qquad \mathsf{inc} : \mathtt{Unit} \to t, \ \ldots|\!\}](x, x) : \mathtt{Bool} \end{array}}$$

assuming

$$\lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}, \mathsf{inc} : \mathtt{Unit} \to t, \ \ldots|\!\} \quad <\!\!: \quad \lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}|\!\}$$

Here again, the type function

$$\lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}|\!\}$$

plays the role of type functional F in the typing rule above. Note that this F is the same as the one used above to type the *equal* expression using F-Bounded universal quantification. However, this typing holds more generally than the one given above because the assumption

$$\lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}, \mathsf{inc} : \mathtt{Unit} \to t, \ \ldots|\!\} \quad <\!\!: \quad \lambda t.\{\!|\mathsf{eq} : t \to \mathtt{Bool}|\!\}$$

does not require any recursion-like properties of $\mathbf{Object}$.

4.4 Existential quantification and data abstraction

While space considerations prohibit a full discussion of data abstraction and its applications, we will describe a general form of data type declaration that may be incorporated into any language with type variables, including the basic object models compared in this paper. For further information, the reader is referred to [LSAS77, MP88, Mor73, Rey83, Set89].

The declaration form

$$\begin{array}{l} \mathtt{abstype}\ t\ \mathtt{with}\ x_1 : \sigma_1, \ldots, x_k : \sigma_k \\ \qquad\qquad \mathtt{is}\ \langle \tau; M_1, \ldots, M_k \rangle\ \mathtt{in}\ N \end{array}$$

declares an abstract type t with "operations" $x_1, \ldots, x_k$ and implementation $\langle \tau; M_1, \ldots, M_k \rangle$. The scope of this declaration is N. For example, the expression

$$\begin{array}{ll} \mathtt{abstype}\ stream\ \mathtt{with} & \\ & s : stream, \\ & first : stream \to nat, \\ & rest : stream \to stream \\ \mathtt{is} & \\ & \langle \tau, M_1, M_2, M_3 \rangle \\ \mathtt{in} & \\ & N \end{array}$$

declares an abstract data type *stream* with distinguished element $s : stream$ and functions *first* and *rest* for operating on streams. Within the scope N of the declaration, the stream s and operations *first* and *rest* may be used to compute natural numbers or other results. However, the type of N may not be *stream,* since this type is local to the expression. In computational terms, the elements of the abstract type *stream* are represented by values of the type τ given in the implementation. Operations $s, first$ and $rest$ are implemented by expressions M_1, M_2 and M_3. Since the value of s must be a stream, the expression M_1 must have type τ, the type of values used to represent streams. Similarly, we must have $M_2 : \tau \to nat$ and $M_3 : \tau \to \tau$. Using cartesian products, we may put any abstract data type declaration in the form $\mathtt{abstype}\ t\ \mathtt{with}\ x : \sigma\ \mathtt{is}\ \langle \tau, M \rangle\ \mathtt{in}\ N$. For example, the *stream* declaration may be put in this form by combining the three operations s, *first* and *rest* into a single operation of type $stream \times (stream \to nat) \times (stream \to stream)$. There is no loss in doing so, since we may recover s, *first* and *rest* using projection functions.

Some useful flexibility is gained by considering abstract data type declarations and data type implementations separately. An implementation for the abstract type *stream* mentioned above consists of a type τ, used to represent *stream*'s, together with expressions for the specified stream s and the stream operations *first* and *rest.* If we want to describe implementations of streams in general, we might say that in any implementation, "there exists a type t with elements of types t, $t \to nat$, and $t \to t$." This description would give just enough information about an arbitrary implementation to determine that an `abstype` declaration makes sense, without giving any information about how streams are represented. This fits the general goals of data abstraction, as discussed in [Mor73], for example.

We may add abstract data type implementations and `abstype` declarations to a language with type variables as follows. The first step is to extend the syntax of type expressions

$$\sigma \ ::= \ \ldots \mid \exists t.\sigma$$

to include *existential types* of the form $\exists t.\sigma$.

Intuitively, each element of an existential type $\exists t.\sigma$ consists of a type τ and an element of $[\tau/t]\sigma$. Using products to combine s, *first*, and *rest*, an implementation of *stream* would have type $\exists t.[t \times (t \to nat) \times (t \to t)]$, for example.

Existential types were part of Girard's System F [Gir71, Gir72] (but not linked to abstract data types).

When we write abstract data type implementations apart from the declarations that use them, it is necessary to include some type information which might at first appear redundant. We will write an implementation in the form $\langle t = \tau, M : \sigma \rangle$, where $t = \tau$ binds t in the remainder of the expression. The reader may think of $\langle t = \tau, M : \sigma \rangle$ as a "pair" $\langle \tau, M \rangle$ in which access to the representation type τ has been restricted. The bound type variable t and the type expression σ serve to disambiguate the type of the expression. The type of a well-formed expression $\langle t = \tau, M : \sigma \rangle$ is $\exists t.\sigma$, according to the following rule.

$$(\exists\ \text{Intro}) \qquad \frac{e_1 : [\tau/t]\sigma}{\langle t = \tau, e_1 : \sigma \rangle : \exists t.\sigma}$$

Since a type σ is not uniquely determined by the form of a substitution instance $[\tau/t]\sigma$, the type of a simpler implementation form $\langle \tau, M \rangle$ would not be determined uniquely. The following rule for `abstype` declarations allows us to bind names to the type and value components of a data type implementation.

$$(\exists \text{ Elim}) \quad \frac{x : \tau \triangleright e_2 : \sigma, \ e_1 : \exists t.\tau}{(\texttt{abstype } t \texttt{ with } x : \tau \texttt{ is } e_1 \texttt{ in } e_2) : \sigma}$$

$$t \text{ not free in } \sigma$$

Informally, this rule binds type variable t and ordinary variable x to the type and value part of the implementation e_1, with scope e_2.

4.4.1 Bounded existential quantification

For each of the various forms of bounded universal quantification discussed in Section 4.1, there is a corresponding form for existential quantification. These ideas are used in the existential-type model of object-oriented programming to encode binary methods. These techniques are beyond the scope of this paper.

5 Theoretical object models

In the next three sections, we summarize three object models from the literature.

- *A record model.* This is an insightful encoding of objects as recursively-defined records. However, the treatment of inheritance is relatively complex, requiring some form of record concatenation. This model is largely due to Cook [Coo89a, CHC90], drawing on earlier work by Cardelli and others [Car88, CW85].
- *An existential-type model.* This is an encoding that treats objects as elements of existential type. The advantages are that this encoding makes the hiding of private "instance variables" or "members" explicit and renders type recursion non-essential. It also seems possible to model "protected" members, to use C++ terminology, which does not seem directly possible in other models. A disadvantage is the relatively complex treatment of binary methods. The model is due to Pierce and Turner [PT94, PT93], drawing on the earlier formulation of abstract data types using existential types [MP88]. (Some comments suggesting a similar approach appear in the concluding section of [Bru93].)
- *An axiomatic model of objects.* Unlike the other approaches, this is a direct model, giving explicit typing rules for operations on objects, instead of an encoding (or "compilation") into a typed calculus without objects. An interesting trade-off that appears is that object-based inheritance, as found in Self [US91, CU89] (see also [Lie86, Ste87]), seems to preclude subtyping. The calculus described here is from [Mit90, FHM94]; in a variant due to Abadi and Cardelli [Aba94, AC94], restrictions on the object operations yield a natural form of subtyping.

All of the models presented here are functional. Adding memory and side-effects seems to be orthogonal to object features, at least as far as has been investigated [AC95a, Bru93, Pie93].

In the recursive record and existential type models, inheritance is class-based, while the third model uses an object-based form of inheritance. The main difference is that in object-based forms, method override and method addition are operations on objects. In class-based languages, the set of methods is fixed at the time an object is created. A more recent "hybrid" model is developed in [Aba94, AC94], where run-time override is allowed, but run-time method addition is not. A relatively technical comparison between the first and third approaches appears in [Bru92]. The recursive record and existential models cannot model runtime method addition because the new methods cannot be added inside the recursion or inside the existential. Subtyping consequences of object-based inheritance are discussed in Section 8.7.

6 The recursive record model

We will use the term "record model" to refer to the representation of objects by records that was developed by William Cook and others [Coo87, Coo89a, CHC90] using concepts pioneered by Luca Cardelli [Car88, CW85]. The main idea is to represent an object by a record of access functions or methods. If we use a lambda calculus with records, then this approach provides a functional model of class-based, object-oriented languages. Some important points are that this is an encoding; it is not a direct presentation of typing rules or logical principles for objects. Furthermore, to account for inheritance in a type-correct way, we need a typed, polymorphic lambda calculus with type recursion and some form of extensible records. As presented in the literature, the approach requires lazy evaluation for the fixed points to work correctly. However, this does not seem to be a fundamental limitation; a variant should work properly under eager evaluation.

6.1 The Objects

Objects are modeled as recursively-defined records. The components of the record represent the methods of the object. Since each method may need to access the other methods of the object, there is a special variable available to the methods that represents the object itself. This variable is frequently called `self` to reinforce the intuition that it refers to the object itself. The mutual recursion of object methods is the motivation for using recursively-defined records. Objects are formed by taking the fixed point of an object-definition function, which is a function of the following form:

$$\texttt{object\_definition} \stackrel{def}{=} \lambda self.\{m_1 = e_1, \ldots, m_n = e_n\}$$

The record labels $\texttt{m}_1 \ldots \texttt{m}_n$ are the names of the methods of the desired object, and the expressions $\texttt{e}_1 \ldots \texttt{e}_n$ are the method implementations. The abstracted variable `self` is the variable that the methods refer to in order to access the other methods of the object. The object described by this definition is created by taking its fixed point:

$$\texttt{object} \stackrel{def}{=} \texttt{Y(object\_definition)}$$

If we want to create a one-dimensional point object with x, move, and eq methods, the object-definition function could be:

$$\begin{aligned}&\texttt{point\_definition} \stackrel{def}{=} \\ &\quad \lambda\texttt{self}.\{\ \ \texttt{x} = 3, \\ &\qquad\qquad \texttt{move} = \lambda\texttt{dx}.\{\ldots \texttt{self.x} + \texttt{dx}\ldots\}, \\ &\qquad\qquad \texttt{eq} = \lambda\texttt{other}.(\texttt{equal other.x self.x})\ \}\end{aligned}$$

We don't yet have enough machinery to write the move method. (See Section 6.5 below.) Point objects are created by taking the fixed point of this function:

$$\texttt{pt} \quad \stackrel{def}{=} \quad \texttt{Y}(\texttt{point\_definition})$$

6.2 Message Send

Message send in this model is very straightforward: it is simply record field selection. The self variable is already bound to the object via the fixed-point operation.

6.3 Object Types

Since objects are recursively-defined records, object types are record types. These are typically recursively-defined types since an argument or return value of a method may be an object of the same type. In an object-definition function, if the type of the abstracted variable self is assumed to be t, then the type of each method can be written in terms of t. This correspondence can be written as a type function F from the type of the abstracted variable to the type of the resultant method record:

$$\texttt{F}[\texttt{t}] = \{\!\!\{\texttt{m}_1 : \sigma_1, \ldots, \texttt{m}_n : \sigma_n\}\!\!\}$$

where σ_i is the type of the ith method, $\texttt{m}_i$, and t may appear in σ_i. Such a function can be thought of as an object interface, since it specifies the types of the object's methods. Occurrences of t in the method types will refer to the type of the object itself after the fixed point is taken. In particular, if the type of an object-definition function f corresponds to such an F, then the type of objects created from f is $\mu\texttt{t}.\texttt{F}[\texttt{t}]$.

The type function that corresponds to the point_definition example above is:

$$\texttt{P}[\texttt{t}] \quad = \quad \{\!\!\{\texttt{x} : \texttt{Int},\ \texttt{move} : \texttt{Int} \to \texttt{t},\ \texttt{eq} : \texttt{t} \to \texttt{Bool}\}\!\!\}$$

and the type of pt is $\mu\texttt{t}.\texttt{P}[\texttt{t}]$. Written out, this type is $\mu\texttt{t}.\{\!\!\{\texttt{x} : \texttt{Int},\ \texttt{move} : \texttt{Int} \to \texttt{t},\ \texttt{eq} : \texttt{t} \to \texttt{Bool}\}\!\!\}$.

Although appealing in its simplicity, this typing is inadequate when we try to add inheritance. Intuitively, the problem is that the methods we define for the current object should be useful in future extensions to it. This possibility means that we do not know the exact type of the object for which we are defining methods. To address this lack of precise knowledge, we add a type parameter to each object-definition function. This parameter represents the type of object that the method definitions are being used in. In typing the methods for the current object, we do need to know something about the type of object we are creating. In particular, we need to know that the self variable contains at least the methods we are defining, and that these methods have the appropriate types. To this end, we use F-bounded quantification to restrict the types that can be passed as the type of the current object. The F's that we use are the familiar type functions from above. Thus the generalized form of object-definition function is now:

$$\lambda\texttt{t} <: \texttt{F}[\texttt{t}].\ \lambda\texttt{self} : \texttt{t}.\{\texttt{m}_1 = \texttt{e}_1, \ldots, \texttt{m}_n = \texttt{e}_n\}$$

where F is as above. This expression has type:

$$\forall\texttt{t} <: \texttt{F}[\texttt{t}].\texttt{t} \to \texttt{F}[\texttt{t}]$$

The object-definition function for points, with this refined typing, is:

$$\begin{aligned}&\texttt{point\_definition} \stackrel{def}{=} \\ &\quad \lambda\texttt{t} <: \texttt{P}[\texttt{t}].\ \lambda\texttt{self} : \texttt{t}. \\ &\quad \{\texttt{x} = 3, \\ &\quad\ \ \texttt{move} = \lambda\texttt{dx} : \texttt{Int}.\{\ldots\texttt{self.x} + \texttt{dx}\ldots\}, \\ &\quad\ \ \texttt{eq} = \lambda\texttt{other} : \texttt{t}.(\texttt{equal other.x self.x})\ \}\end{aligned}$$

which has type

$$\forall\texttt{t} \leq \texttt{P}[\texttt{t}].\ \texttt{t} \to \texttt{P}[\texttt{t}],$$

assuming that the implementation of move has the appropriate type.

To create objects, we now need to supply the type of object we are creating. Keeping with our earlier notational convention, let $\mathbf{Object}\,\texttt{F} = \mu\texttt{t}.\texttt{F}[\texttt{t}]$ for any $\texttt{F} : \mathit{Type} \Rightarrow \mathit{Type}$. Then the expression

$$\texttt{pt} \quad \stackrel{def}{=} \quad \texttt{Y}(\texttt{point\_definition}[\mathbf{Object}\,\texttt{P}])$$

creates a point object pt with type $\texttt{Point} \stackrel{def}{=} \mathbf{Object}\,\texttt{P}$.

6.4 Inheritance

Inheritance is modeled via a simple form of record concatenation. Basically, in the object-definition function for the derived class, the record of methods from the base class is concatenated with the record of methods to be added. This concatenation is achieved via the with record combination operator, which is typed using the following introduction rule:

$$\frac{\begin{array}{ll}\texttt{e}_1 : \{\!\!\{\texttt{m}_1 : \sigma_1, \ldots, \texttt{m}_j : \sigma_j, \texttt{m}_{j+1} : \tau_1, \ldots \texttt{m}_k : \tau_{k-j}\}\!\!\} & \\ \texttt{e}_2 : \{\!\!\{\texttt{m}_{j+1} : \sigma_{j+1}, \ldots, \texttt{m}_n : \sigma_n\}\!\!\} & (\texttt{n} \geq \texttt{k})\end{array}}{\texttt{e}_1\ \texttt{with}\ \texttt{e}_2 : \{\!\!\{\texttt{m}_1 : \sigma_1, \ldots, \texttt{m}_n : \sigma_n\}\!\!\}}$$

If the two records $\texttt{e}_1$ and $\texttt{e}_2$ have common fields, the conflict is resolved by taking the value and type from $\texttt{e}_2$. Since objects are recursive records, this concatenation operator seems to present a problem. If we change the type of some method m in $\texttt{e}_1$ via concatenation, the methods of $\texttt{e}_1$ that depended on the type of m could now break. As we will see, the records that need to be concatenated for this encoding of inheritance to work do not have this problem. (This is our observation, not one discussed in [CHC90].) There is a corresponding record type concatenation operator $+$, defined analogously.

Using the with operator, we can define an object-definition function for a colored point from point_definition as follows:

$$
\begin{array}{l}
\texttt{cpoint\_definition} \stackrel{def}{=} \\
\lambda \texttt{t} <: \texttt{C}[\texttt{t}].\, \lambda \texttt{self} : \texttt{t}. \\
\quad \texttt{point\_definition}[\texttt{t}](\texttt{self})\ \texttt{with} \\
\quad\quad \{\texttt{myColor} = \texttt{blue}, \\
\quad\quad\ \ \texttt{eq} = \lambda \texttt{other} : \texttt{t}. \\
\quad\quad\quad\quad (\texttt{equal self.myColor other.myColor} \\
\quad\quad\quad\quad\quad \texttt{and equal self.x other.x}) \\
\quad\quad \}
\end{array}
$$

where

$$
\begin{array}{lcl}
\texttt{C}[\texttt{t}] & \stackrel{def}{=} & \{\!|\texttt{x} : \texttt{Int}, \texttt{move} : \texttt{Int} \to \texttt{t}, \\
& & \ \texttt{eq} : \texttt{t} \to \texttt{Bool}, \texttt{myColor} : \texttt{Color}|\!\}
\end{array}
$$

The type of colored points is:

$$
\begin{array}{lcl}
\texttt{CPoint} & \stackrel{def}{=} & \textbf{Object}\,\texttt{C} \\
& = & \mu \texttt{t}.\{\!|\texttt{x} : \texttt{Int}, \texttt{move} : \texttt{Int} \to \texttt{t}, \\
& & \quad \texttt{eq} : \texttt{t} \to \texttt{Bool}, \texttt{myColor} : \texttt{Color}|\!\}
\end{array}
$$

To guarantee that the type application `point_definition[t]` is well typed, we need $\texttt{C}[\texttt{t}] <: \texttt{P}[\texttt{t}]$. This requirement prevents us from redefining the type of the `x` method to anything other than a subtype of `Int`. This is exactly what we need to guarantee that the addition in the `move` method defined for points works correctly when it is used in colored points. There is a slight bug in [CHC90] related to whether or not $\texttt{C}[\texttt{t}] <: \texttt{P}[\texttt{t}]$. See Section 6.8 for a discussion of this problem.

6.5 Inheritance with Classes

Each of these object-definition functions can create only a single object. This lack of generality is a problem. We cannot write the `move` method above, because we cannot create a point in a different location. To remedy this problem, we need the notion of a "class." Intuitively, a class is a function that creates an object-definition function from instantiation parameters. A class for points would take an `x`-coordinate as a parameter and return a point object-definition function that creates points at that location. A complication arises, however, from the fact that objects defined from a class may need to use that class to create new instances of the class. For example, point objects need to be able to create new point objects to implement the `move` method. This requirement means that classes, like objects, are recursive. A class is converted into an object-definition function by applying it to its instantiation parameters and then taking a fixed point.

A point class could be defined as follows:

$$
\begin{array}{l}
\texttt{point\_class} \stackrel{def}{=} \\
\quad \lambda \texttt{t} <: \texttt{P}[\texttt{t}].\, \lambda \texttt{myclass} : \texttt{Int} \to (\texttt{t} \to \texttt{t}). \\
\quad\quad \lambda \texttt{init\_x} : \texttt{Int}.\, \lambda \texttt{self} : \texttt{t}. \\
\quad\quad\ \{\texttt{x} = \texttt{init\_x}, \\
\quad\quad\ \ \texttt{move} = \lambda \texttt{dx} : \texttt{Int}.\, \texttt{Y}(\texttt{myclass}(\texttt{self.x} + \texttt{dx})), \\
\quad\quad\ \ \texttt{eq} = \lambda \texttt{other} : \texttt{t}.(\texttt{equal other.x self.x})\ \}
\end{array}
$$

The following expression creates a point at location 2:

$$
\texttt{pt} \stackrel{def}{=} \texttt{Y}(\texttt{Y}(\texttt{point\_class}[\textbf{Object}\,\texttt{P}])(2))
$$

We can define a color point class function from `point_class` in a manner similar to above. The only difficulty arises from the fact that colored points have an additional instantiation parameter: their color. To use the `point_class` definition, we need to supply conversions between the two types of instantiation parameters:

$$
\begin{array}{l}
\texttt{cpoint\_class} \stackrel{def}{=} \\
\quad \lambda \texttt{t} <: \texttt{C}[\texttt{t}].\, \lambda \texttt{myclass} : \texttt{Int} \times \texttt{Color} \to (\texttt{t} \to \texttt{t}). \\
\quad \lambda \langle \texttt{init\_x}, \texttt{init\_color} \rangle : \texttt{Int} \times \texttt{Color}.\, \lambda \texttt{self} : \texttt{t}. \\
\quad\ \texttt{inherited\_methods with} \\
\quad\quad \{\texttt{myColor} = \texttt{init\_color}, \\
\quad\quad\ \texttt{eq} = \lambda \texttt{other} : \texttt{t}. \\
\quad\quad\quad (\texttt{equal other.x self.x and} \\
\quad\quad\quad\quad \texttt{equal other.myColor self.myColor})\}
\end{array}
$$

where

$$
\begin{array}{l}
\texttt{inherited\_methods} \stackrel{def}{=} \\
\quad \texttt{point\_class}[\texttt{t}](\lambda \texttt{init\_x} : \texttt{Int}. \\
\quad\quad \texttt{myclass}\langle \texttt{init\_x}, \texttt{self.myColor}\rangle)\, \texttt{init\_x self}
\end{array}
$$

The expression

$$
\texttt{cpt} \stackrel{def}{=} \texttt{Y}(\texttt{Y}(\texttt{cpoint\_class}[\textbf{Object}\,\texttt{C}])(\langle 2, \texttt{blue}\rangle))
$$

creates a blue color point at location 2.

Note that this system's notion of a class is very different from its notion of an object type. This differs from existing typed object-oriented languages, which have unified the two concepts.

6.6 Dynamic Lookup

We get dynamic method lookup in this system because each object carries its methods with it, and because each method refers to the other methods of the object via an indirection through the `self` variable. Since this variable is only bound when an object is created, each method uses the last version of the object's other methods. Earlier definitions in the record of methods are eliminated via the record concatenation operator `with`.

6.7 Encapsulation

This model provides minimal support for encapsulation. It insures that a method is an integral part of the object that contains it: a method cannot be extracted from its object and used on some other object, for example. However, there is no mechanism for encapsulating local state or making certain methods accessible only to derived classes. Some work would be required to add such forms of protection, particularly if we wanted to model various levels of encapsulation, such as C++ 's notions of public, protected and private object members.

6.8 Subtyping

In the recursive record model, subtyping and inheritance are completely separate. The inheritance hierarchy is defined by the programmer as classes are written. The subtype relation is inferred from types according to subtyping rules. The only non-standard rule in this system is the one for record subtyping, which prevents a record type from being a subtype of a record with fewer methods. In the terminology of Section 4.1, we have subtyping in "depth" but not in "width." The reason is that only "depth" preserves the list of methods, which is needed for the soundness of the record combination operator `with`. It is overly restrictive, however, because it prevents the encoding above of points and colored points. Consider that we required $\mathtt{C[t]} <: \mathtt{P[t]}$ to define colored points from points. This relation between `C[t]` and `P[t]` does not hold under the restrictive record subtyping rule given above because `C[t]` has extra fields. A more precise record combination operator is probably required to allow the general record subtyping inference rule and to fix the above bug.

The record subtyping rule does permit the specialization of record field types. In contrast, such subtyping is unsound in the presence of method override, as is discussed in Section 8.7.

6.9 Method Specialization

Although not discussed in [CHC90], this model allows even more method specialization than the "mytype" specialization described in Section 2.7. Since the only restriction on the record of methods for the derived class is that its type be a subtype of the type of the base method record, the types of individual record components can be specialized to subtypes. For example, a method that returns an integer in the base class can be specialized to a method that returns a positive integer in the derived class.

6.10 Binary Methods

Binary methods are those that take as a parameter an object of the same type as their containing object. When a class that defines binary methods is inherited from, objects created from the resulting class definition are not subtypes of objects created from the original class. This is because the "mytype" parameter appears contravariantly. In the example above, the type of colored points is not a subtype of the type of points because of the `eq` method. This means that colored points cannot be passed to functions that expect points. However, there is still a great deal of similarity between points and colored points. We can write F-bounded functions that take as a first parameter the type of the object for which they are expected to work. If such a function is defined for all types that are subtypes of P[t], then the function will work for colored points if we pass as the first parameter **Object** `C` because $\mathtt{C[t]} <: \mathtt{P[t]}$. This mechanism prevents colored points and points from being used interchangeably, but allows significant code reuse. Recent work in this direction appears in [BSv95, AC95b]

7 The Existential Model

This approach provides a functional model of class-based object-oriented languages based on an encoding into $F^{\omega}_{\leq}$, an explicitly typed, polymorphic lambda calculus with subtyping. This summary is based on [PT94]. See also [PT93].

7.1 Objects and their types

In this system, objects are modeled as elements of existential type. For example, the type of our point object would be as follows:

$$
\begin{array}{l}
\mathtt{Point} \overset{def}{=} \\
\quad \exists \mathrm{Rep}. \ \{\!\!\{ \ \mathtt{state} : \mathtt{Rep}, \\
\qquad\qquad \mathtt{methods} : \{\!\!\{ \mathtt{getX} : \mathtt{Rep} \to \mathtt{Int}, \\
\qquad\qquad\qquad \mathtt{move} : \mathtt{Rep} \to \mathtt{Int} \to \mathtt{Rep}, \\
\qquad\qquad\qquad \mathtt{eq} : \mathtt{Rep} \to ?? \to \mathtt{Bool} \ \}\!\!\} \\
\qquad\qquad \}\!\!\}
\end{array}
$$

Each object has an internal `state` component and a record of methods that operate on that state. The type of the internal state is hidden by the existential quantifier, so external functions cannot operate on it. This hiding provides an encapsulation barrier that models the encapsulation found in many existing object-oriented languages. The types of an object's methods are written in terms of the type of the hidden state. The reasons for the question marks (??) in the type for the `eq`-method will be discussed later. Interface specification functions are type operators that map the internal representation type of an object to a record of method types. It is useful to write the type of an object in terms of such specification functions, for reasons that we will see later. The specification function for the example above is:

$$
\begin{array}{lll}
\mathtt{PointM} & \overset{def}{=} & \lambda \mathrm{Rep}.\{\!\!\{ \ \mathtt{getX} : \mathtt{Rep} \to \mathtt{Int}, \\
& & \qquad \mathtt{move} : \mathtt{Rep} \to \mathtt{Int} \to \mathtt{Rep}, \\
& & \qquad \mathtt{eq} : \mathtt{Rep} \to ?? \to \mathtt{Bool} \\
& & \quad \}\!\!\} : \mathit{Type} \Rightarrow \mathit{Type}
\end{array}
$$

The kind $\mathit{Type} \Rightarrow \mathit{Type}$ indicates that `PointM` is a function from types to types. Note that the `PointM` function is very similar to the F-bound `P` that we saw for points in the previous section. Their roles are roughly analogous.

Using the `PointM` function, we can write the type of the point object above as:

$$
\begin{array}{l}
\mathtt{Point} \overset{def}{=} \\
\qquad \exists \mathrm{Rep}.\{\!\!\{ \mathtt{state} : \mathtt{Rep}, \ \mathtt{methods} : \mathtt{PointM}\,[\mathrm{Rep}] \}\!\!\}
\end{array}
$$

Since every object type has this form, we can get a general object type constructor by abstracting away the `PointM` function:

$$
\begin{array}{l}
\mathtt{Object} \overset{def}{=} \\
\quad \lambda M : \mathit{Type} \Rightarrow \mathit{Type}. \exists \mathrm{Rep}. \\
\qquad \{\!\!\{ \mathtt{state} : \mathtt{Rep}, \ \mathtt{methods} : \mathtt{M}\,[\mathrm{Rep}] \}\!\!\}
\end{array}
$$

This use of `Object` fits the general pattern of the **Object** type functional discussed in Section 2.4. Objects are created via the existential introduction rule. They have the form $\langle \mathtt{R}, \mathtt{r} \rangle : \mathtt{T}$. The `R` is a witness type for the existential; it provides the type of the object's state component in `r`, the implementation of the object. The annotation `T` is the external type of the object. For example, an object with type `Point` might be implemented as follows:

$$\begin{array}{l}
\texttt{p1} \stackrel{def}{=} \\
\quad \langle \{\!|\texttt{x}:\texttt{Int}|\!\}, \\
\quad\ \{\texttt{state} = \{\texttt{x} = 3\}, \\
\quad\ \ \texttt{methods} = \{\texttt{getX} = \lambda\texttt{s}:\{\!|\texttt{x}:\texttt{Int}|\!\}.(\texttt{s.x}), \\
\qquad\qquad\qquad \texttt{move} = \lambda\texttt{s}:\{\!|\texttt{x}:\texttt{Int}|\!\}. \\
\qquad\qquad\qquad\qquad \lambda\texttt{i}:\texttt{Int}.\{\texttt{x} = \texttt{s.x} + \texttt{i}\}, \\
\qquad\qquad\qquad \texttt{eq} = \lambda\texttt{s}:\{\!|\texttt{x}:\texttt{Int}|\!\}.(??) \\
\quad \}\}\rangle : \texttt{Point};
\end{array}$$

In this example, $\{\!|\texttt{x}:\texttt{Int}|\!\}$ is the type of the internal state component of `p1`. This point is initially positioned at location 3, and the `methods` record defines the `getX`, `move`, and `eq` methods. The implementation of binary methods in this model is beyond the scope of this paper, so the definition of the `eq` method is omitted here. The problem is discussed in Section 7.8.

Unlike existing languages that unify the concepts of object implementation (i.e., class) and object type, this system allows multiple object implementations of the same object type. Another implementation for points might be:

$$\begin{array}{l}
\texttt{p2} \stackrel{def}{=} \\
\quad \langle \{\!|\texttt{x}:\texttt{Int}, \texttt{other}:\texttt{Int}|\!\}, \\
\quad\ \{\texttt{state} = \{\texttt{x} = 3, \texttt{other} = 20\}, \\
\quad\ \ \texttt{methods} = \\
\quad\ \ \ \{\texttt{getX} = \lambda\texttt{s}:\{\!|\texttt{x}:\texttt{Int}, \texttt{other}:\texttt{Int}|\!\}.(\texttt{s.x}), \\
\quad\ \ \ \ \texttt{move} = \lambda\texttt{s}:\{\!|\texttt{x}:\texttt{Int}, \texttt{other}:\texttt{Int}|\!\}. \\
\qquad\qquad\qquad \lambda\texttt{i}:\texttt{Int}.\{\texttt{x} = \texttt{s.x} + \texttt{i}, \\
\qquad\qquad\qquad\qquad \texttt{other} = \texttt{s.other}\}, \\
\qquad\ \ \texttt{eq} = \lambda\texttt{s}:\{\!|\texttt{x}:\texttt{Int}, \texttt{other}:\texttt{Int}|\!\}.(??)\ \} \\
\quad\ \ \} \\
\quad \rangle : \texttt{Point};
\end{array}$$

7.2 Message Send

Sending messages is more complicated in this system than in the recursive record model. Here, message send is modeled by polymorphic functions that take an object as a parameter, open it, apply the appropriate methods from its `methods` component, and then repackage it. Such functions are polymorphic to insure that they will work on all objects derived from the one for which they were originally intended. The form of polymorphism used is higher order bounded universal quantification, which is often interchangeable with the F-bounded polymorphism used in the recursive record model. Here, the bound is the interface specification function that we originally used to give the object's type. As an example, the following function is used to send the message `move` to points and any objects derived from points:

$$\begin{array}{l}
\texttt{Point'move} \stackrel{def}{=} \\
\quad \lambda\texttt{M} <: \texttt{PointM}.\ \lambda\texttt{p}:\texttt{Object M}. \\
\quad \texttt{open p as } \langle\texttt{Rep},\texttt{r}\rangle \texttt{ in} \\
\quad\ \ \lambda\texttt{dx}:\texttt{Int}.\langle\texttt{Rep}, \\
\qquad\qquad \{\texttt{state} = \\
\qquad\qquad\ \ \texttt{r.methods.move(r.state)dx}, \\
\qquad\qquad\ \texttt{methods} = \texttt{r.methods}\} \\
\qquad\qquad \rangle : \texttt{Object M} \\
\quad \texttt{end};
\end{array}$$

The type of `Point'move` is:

$$\forall \texttt{t} <: \texttt{PointM}.\texttt{Object M} \rightarrow \texttt{Int} \rightarrow \texttt{ObjectM}$$

When applied to the type `Point` and a point `p`, this function first opens `p` by binding `Rep` to `p`'s representation type and `r` to `p`'s implementation of a point. It then returns a function that takes a displacement as a parameter. This function creates a new point object in the new location by repackaging the old `Rep` type with the modified state and the methods of the original point. The expression:

$$\texttt{Point'move [Object PointM] p1 2}$$

returns a point exactly like `p1` except at location 5.

Since the interface specification function for our colored points,

$$\begin{array}{lll}
\texttt{CPointM} & \stackrel{def}{=} & \lambda\texttt{Rep}.\{\!|\ \texttt{getX}:\texttt{Rep} \rightarrow \texttt{Int}, \\
& & \qquad\quad \texttt{getCol}:\texttt{Rep} \rightarrow \texttt{Color} \\
& & \qquad\quad \texttt{move}:\texttt{Rep} \rightarrow \texttt{Int} \rightarrow \texttt{Rep}, \\
& & \qquad\quad \texttt{eq}:\texttt{Rep} \rightarrow ?? \rightarrow \texttt{Bool} \\
& & \qquad\quad |\!\} : \mathit{Type} \Rightarrow \mathit{Type}
\end{array}$$

is a higher-order subtype of `PointM`, the message sending function `Point'move` will work on colored points as well as points. In particular, the function

$$\begin{array}{l}
\texttt{Point'move[Object CPointM]}: \\
\quad \texttt{Object CPointM} \rightarrow \texttt{Int} \rightarrow \texttt{Object CPointM}
\end{array}$$

sends messages to colored points.

7.3 Classes and Inheritance

In this system, classes are functions that can either be instantiated to create objects or extended to create new classes. Class extension is how inheritance is modeled in the existential system. As in the recursive record model, classes and types are distinct notions. The objects created from a particular class all have the same type, but not all objects of a particular type must have been created from the same class. A class fixes the representation to be used by the objects it defines. However, future classes can modify this representation, so each of the methods defined in a class must be polymorphic in the final representation type. We also need to be able to convert between the final representation type and the representation type used in the current class. To this end, each class requires a `get` function to convert from the final representation to the current one and a `put` function to map the current representation to the final one. Because the current representation may contain less information than the final one, the `put` function requires an additional parameter that provides a default final representation to supply any missing information.

Another consideration in the encoding of classes is that the methods of an object may refer to the object's other methods. Since these methods may be redefined in later classes, such methods must be accessed via an indirection to guarantee that their latest definitions are used. To encode this dynamic lookup behavior, methods refer to other methods through a `self` parameter, which must be supplied to the class function when creating an object. This `self` parameter contains the methods defined in the object being instantiated.

As an example, the `pointClass` function defined below may be instantiated to create our familiar point objects.

It fixes the representation type for its points to be PointR.

$$\texttt{PointR} \stackrel{def}{=} \{\!|\texttt{x} : \texttt{Int}|\!\}$$

$$\begin{array}{l} \texttt{pointClass} \stackrel{def}{=} \\ \quad \lambda\texttt{FinalR}.\, \lambda\texttt{get} : \texttt{FinalR} \to \texttt{PointR}. \\ \quad \lambda\texttt{put} : \texttt{FinalR} \to \texttt{PointR} \to \texttt{FinalR}. \\ \quad \lambda\texttt{self} : \texttt{PointM}\,[\texttt{FinalR}]. \\ \qquad \{\texttt{getX} = \lambda\texttt{s} : \texttt{FinalR}.((\texttt{get s}).\texttt{x}), \\ \qquad\ \texttt{move} = \lambda\texttt{s} : \texttt{FinalR}.\, \lambda\texttt{i} : \texttt{Int}. \\ \qquad\qquad\qquad \texttt{put s}\{\texttt{x} = (\texttt{self.getX} + \texttt{i})\}, \\ \qquad\quad \texttt{eq} = \lambda\texttt{s} : \texttt{FinalR}.(??)\} \end{array}$$

Note that the `move` function uses the `self` parameter to access the object's `getX` method. The definition of `eq` is subtle; see Section 7.8.

The following expression creates a point object from the `pointClass` function:

$$\begin{array}{l} \texttt{p} \stackrel{def}{=} \\ \quad \langle \texttt{PointR}, \\ \qquad \{\texttt{state} = \{\texttt{x} = 3\}, \\ \qquad\ \texttt{methods} = \texttt{rec}\,[\texttt{PointM}\,\texttt{PointR}] \\ \qquad\qquad\qquad\qquad \texttt{pointClass}\,\texttt{PointR} \\ \qquad\qquad\qquad\qquad (\lambda\texttt{s} : \texttt{PointR}.\,\texttt{s}) \\ \qquad\qquad\qquad\qquad \lambda\texttt{s} : \texttt{PointR}.\, \lambda\texttt{s}' : \texttt{PointR}.\,\texttt{s}'\} \\ \quad \rangle : \texttt{Object}\,\texttt{PointM} \end{array}$$

The representation used in the class function is packaged with an initial value for the state and the methods for the object. These methods are obtained by taking the appropriate fixed point (via `rec`)of the `pointClass` function applied to the final representation type (which is just `PointR` in this case) and `get` and `put` functions. Because the final representation type is the same as the representation type used in the `pointClass` function, the `get` function is just the identity and the `put` function simply returns its second argument.

It is relatively straightforward in $F^{\omega}_{\leq}$ to create a derived class function from a base class function and the incremental changes. However, the details are beyond the scope of this paper.

7.4 Dynamic Lookup

We get dynamic method lookup in this system because each object carries its methods with it, and because each method refers to the other methods of the object via an indirection through the `self` variable. Since this variable is only bound to the appropriate record of methods when an object is created, each method uses the last version of the object's other methods.

7.5 Encapsulation

Because the basic mechanism of this system is existentials, it provides rich possibilities for modeling encapsulation. In the encoding presented here, access to instance variables is limited to the methods of an object and the methods of derived objects. This degree of access models protected instance variables in C++. In a slightly different encoding of classes and inheritance, it is possible to model C++ private instance variables as well.

7.6 Subtyping

As in the recursive record model, the subtype and inheritance relations are separated. The inheritance hierarchy is defined by the programmer as classes are written, and the subtype relation is inferred from types according to subtyping rules. Unlike the recursive record model, we have subtyping in both width and depth (as defined by inference rules in Section 4.1).

7.7 Method Specialization

Although not discussed in [PT94], this encoding supports "mytype" method specialization because the message sending functions are polymorphic in the interface specification type operator. In particular, the `move` method for points will return a colored point if the code is invoked from a colored point instead of from a point. The model does not seem to directly support specializing other types appearing in the interface specification. If it were possible to extract field types from a type of the form `Object M`, such specializations would be possible.

7.8 Binary Methods

Because of its notions of encapsulation and the fact that its types do not fix implementation details, this model has difficulty modeling binary methods, such as the `eq` method whose body we omitted above. In single-dispatch object-oriented languages, the first parameter to a "binary" method is used to select which object's code is going to be executed. The second object is then sent as a parameter to the selected method. To implement such methods efficiently, the first object usually requires access to the representation of the second. Because different objects of the same type can have completely independent internal representations in this system, object types do not provide enough information to allow such operations to be written. [PT93] presents a generalization of this encoding based on partially abstract types that permits the encoding of binary methods.

8 An axiomatic approach

To demonstrate a third alternative, we briefly summarize our own system [Mit90, FHM94]. This is an axiomatic model of a language that supports object-based inheritance. In this system, objects are derived from other objects by adding new methods via method addition or replacing old methods with new ones via method override. This eliminates the need for a separate "class" construct for creating objects that share a common set of methods.

There are two parts to this system, an untyped language for defining objects and functions, and a type system for assigning types to terms.

8.1 The Objects

The untyped object calculus is the result of adding four new object-related syntactic forms to untyped lambda calculus:

$\{\}$	the empty object
$e \Leftarrow m$	send message m to object e
$\{e_1 \leftarrow\!\!+\; m{=}e_2\}$	extend object e_1 with new method m having body e_2
$\{e_1 \leftarrow m{=}e_2\}$	replace e_1's method body for m by e_2

We consider $\{e_1 \leftarrow\!\!+\; m{=}e_2\}$ meaningful only if e_1 denotes an object that does not have an m method, and $\{e_1 \leftarrow m{=}e_2\}$ meaningful only if e_1 denotes an object that already has an m method. These conditions will be enforced by the type system. If a method is new, then no other method in the object could have referred to it, so it may have any type. On the other hand, if a method is being replaced, then we must be careful not to violate any typing assumptions in other methods that refer to it.

To simplify notation, we write $\{m_1 = e_1, \ldots, m_k = e_k\}$ for $\{\ldots\{\{\} \leftarrow\!\!+\; m_1 = e_1\}\ldots \leftarrow\!\!+\; m_k = e_k\}$, where $m_1, \ldots, m_k$ are distinct method names. While we used $\langle\ldots\rangle$ to designate objects in [FHM94], we use $\{\ldots\}$ here for consistency with the other systems. Note, however, that here $\{\ldots\}$ indicates some form of object, not a record.

In this system, we may encode our one-dimensional points as follows:

$$
\begin{array}{l}
\mathtt{p} \stackrel{def}{=} \\
\quad \{\mathtt{x} = \lambda\mathtt{self}.\,3, \\
\quad\ \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}. \\
\qquad \{\mathtt{self} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{self} \Leftarrow \mathtt{x}) + \mathtt{dx}\}, \\
\quad\ \mathtt{eq} = \lambda\mathtt{self}.\,\lambda\mathtt{other}. \\
\qquad \mathtt{equal}(\mathtt{self} \Leftarrow \mathtt{x})(\mathtt{other} \Leftarrow \mathtt{x}) \\
\quad \}
\end{array}
$$

8.2 Message Send

Sending a message m_i to an object containing such a method is modeled by extracting the corresponding method body e_i from the object and applying it to the object itself:

$$\{m_1 = e_1, \ldots, m_k = e_k\} \Leftarrow m_i \stackrel{eval}{\longrightarrow} e_i\ \{m_1 = e_1, \ldots, m_k = e_k\}$$

For example, sending the message move with a displacement of 2 to p, we get:

$$
\begin{array}{rcl}
\mathtt{p} \Leftarrow \mathtt{move}\,2 & = & (\lambda\mathtt{self}.\,\lambda\mathtt{dx}.\{\ \ldots\ \})\ \mathtt{p}\ \ 2 \\
& = & \{\mathtt{p} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{p} \Leftarrow \mathtt{x}) + 2\} \\
& = & \{\mathtt{p} \leftarrow \mathtt{x} = \lambda\mathtt{s}.\,3 + 2\} \\
& = & \{\mathtt{p} \leftarrow \mathtt{x} = \lambda\mathtt{self}.\,5\}
\end{array}
$$

Using a sound rule for object equality,

$$\{\{m_1{=}e_1, \ldots, m_k{=}e_k\} \leftarrow m_i{=}e_i'\} \ = \ \{m_1{=}e_1, \ldots, m_i{=}e_i', \ldots, m_k{=}e_k\}$$

we may reach the conclusion

$$
\begin{array}{l}
\mathtt{p} \Leftarrow \mathtt{move}\,2 \ = \\
\quad \{\mathtt{x} = \lambda\mathtt{self}.\,5, \\
\quad\ \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\{\ldots\} \\
\quad\ \mathtt{eq} = \lambda\mathtt{self}.\,\lambda\mathtt{other}. \\
\qquad\qquad \mathtt{equal}\ (\mathtt{self} \Leftarrow \mathtt{x})(\mathtt{other} \Leftarrow \mathtt{x}) \\
\quad \}
\end{array}
$$

showing that the result of sending a move message with an integer parameter, is an object identical to p, but with an updated x-coordinate.

8.3 Inheritance

In this system, the inheritance mechanism is very simple, although its typing is somewhat complex. To define colored points from points, we simply need to add a color method to our point object p and then redefine the eq method to test the color fields for equality as well as the x-coordinates.

$$
\begin{array}{l}
\mathtt{cp} \stackrel{def}{=} \\
\quad \{\{p \leftarrow\!\!+\; \mathtt{myColor} = \lambda\mathtt{self}.\,\mathtt{blue}\} \\
\qquad \leftarrow \mathtt{eq} = \lambda\mathtt{self}.\,\lambda\mathtt{other}. \\
\qquad\qquad (\mathtt{equal}\,(\mathtt{self} \Leftarrow \mathtt{x})\ (\mathtt{other} \Leftarrow \mathtt{x})\ \ \mathtt{and} \\
\qquad\qquad\quad \mathtt{equal}\,(\mathtt{self} \Leftarrow \mathtt{myColor}) \\
\qquad\qquad\qquad (\mathtt{other} \Leftarrow \mathtt{myColor}))\}
\end{array}
$$

If we send the move message to cp with the same parameter, we may calculate the resulting object in exactly the same way as before:

$$
\begin{array}{rcl}
\mathtt{cp} \Leftarrow \mathtt{move}\,2 & = & (\lambda\mathtt{self}.\,\lambda\mathtt{dx}.\{\ \ldots\ \})\ \mathtt{cp}\ \ 2 \\
& = & \{\mathtt{cp} \leftarrow \mathtt{x} = \lambda\mathtt{s}.(\mathtt{cp} \Leftarrow \mathtt{x}) + 2\} \\
& = & \ldots \\
& = & \{\mathtt{cp} \leftarrow \mathtt{x} = \lambda\mathtt{self}.\,5\}
\end{array}
$$

with the final conclusion that

$$
\begin{array}{l}
\mathtt{cp} \Leftarrow \mathtt{move}\,2 \ = \\
\{\,\mathtt{x} = \lambda\mathtt{self}.\,5, \\
\ \ \mathtt{move} = \lambda\mathtt{self}.\,\lambda\mathtt{dx}.\{\ldots\}, \\
\ \ \mathtt{eq} = \lambda\mathtt{self}.\,\lambda\mathtt{other}. \\
\qquad \mathtt{equal}\,(\mathtt{self} \Leftarrow \mathtt{x})(\mathtt{other} \Leftarrow \mathtt{x})\ \ \mathtt{and} \\
\qquad \mathtt{equal}\,(\mathtt{self} \Leftarrow \mathtt{myColor})(\mathtt{other} \Leftarrow \mathtt{myColor}), \\
\ \ \mathtt{myColor} = \lambda\mathtt{self}.\,\mathtt{blue} \\
\}
\end{array}
$$

The important feature of this computation is that the color of the resulting colored point is the same as the original one. While move was defined originally for points, the method body performs the correct computation when the method is inherited by an object with additional methods. This gives us the "mytype" method specialization discussed in Section 2.7.

8.4 Object Types

In the original presentation of this system, the type of an object was called a *class type.* We use **Object** here instead, to maintain uniformity with the rest of this paper. The type

$$\mathbf{Object}\,\mathtt{t}.\{\!|m_1 : \tau_1, \ldots, m_k : \tau_k|\!\}$$

is a type t with the property that any element x of this type is an object such that for $1 \leq i \leq k$, the result of $x \Leftarrow m_i$ is a value of type τ_i. A significant aspect of this type is that the bound type variable t may appear in the types $\tau_1, \ldots, \tau_k$. Thus, when we say $x \Leftarrow m_i$ will have type τ_i, we mean type τ_i with any free occurrences of t in τ_i referring to the type $\mathbf{Object}\,\mathtt{t}.\{\!|m_1 : \tau_1, \ldots, m_k : \tau_k|\!\}$ itself. Thus, $\mathbf{Object}\,\mathtt{t}.\{\!|\ldots|\!\}$ is a special form of recursively-defined type.

The typing rule for message send has the form

$$\frac{e : \mathbf{Object}\,\mathtt{t}.\{\!|\ \ldots\ m : \tau|\!\}}{e \Leftarrow m : [\mathbf{Object}\,\mathtt{t}.\{\!|\ \ldots\ m : \tau|\!\}/t]\tau}$$

where the substitution for $\mathtt{t}$ in τ reflects the recursive nature of object types. This rule may be used to give the point $\mathtt{p}$ with $\mathtt{x}$, $\mathtt{move}$ and $\mathtt{eq}$ methods type

$$\mathbf{Object}\,\mathtt{t}.\{\!|\, \mathtt{x}:\mathtt{Int},\ \mathtt{move}:\mathtt{Int}\rightarrow\mathtt{t},\ \mathtt{eq}:\mathtt{t}\rightarrow\mathtt{Bool}\,|\!\}$$

since $\mathtt{p} \Leftarrow \mathtt{x}$ returns an integer, $\mathtt{p} \Leftarrow \mathtt{move}\ \mathtt{n}$ has the same type as $\mathtt{p}$, and when $\mathtt{eq}$ is passed an object with the same type as $\mathtt{p}$, it returns a boolean.

A subtle but very important aspect of the type system is that when an object is extended with an additional method, the syntactic type of each method does not change. For example, when we extended $\mathtt{p}$ with a color to obtain $\mathtt{cp}$, we obtain an object with type

$$\begin{array}{l}\mathbf{Object}\,\mathtt{t}.\{\!|\, \mathtt{x} : \mathtt{Int},\ \mathtt{move} : \mathtt{Int}\rightarrow\mathtt{t}, \\ \qquad\qquad \mathtt{eq} : \mathtt{t}\rightarrow\mathtt{Bool},\ \mathtt{myColor} : \mathtt{Color}\,|\!\}\end{array}$$

The important change to notice here is that although the syntactic type of $\mathtt{move}$ is still $\mathtt{Int}\rightarrow\mathtt{t}$, the meaning of the variable $\mathtt{t}$ has changed. Instead of referring to the type of $\mathtt{p}$ as it did originally, it now refers to the type of $\mathtt{cp}$. This is the "mytype" method specialization discussed in Section 2.7. For this kind of reinterpretation of type variables to be sound, the typing rule for object extension must insure that every possible type for a new method will be correct. This is done through a form of implicit higher-order polymorphism.

8.5 Dynamic Lookup

We get dynamic method lookup in this system because each object carries its methods with it, and because each method refers to the other methods of the object via an indirection through the `self` variable. Since this variable is bound to its object when messages are sent to the object, methods are guaranteed to get the latest version of the other methods.

8.6 Encapsulation

As in the recursive record model, this model provides minimal encapuslation. Methods cannot be extracted from their objects, but no support is provided for encapsulating local state or limiting access to existing methods when an object is extended. Some work would be required to add such protection, particularly if we wanted to model various levels of encapsulation, such as those provided by C++.

8.7 Subtyping

In languages that support pure object-based inheritance, no subtyping is possible. We did not realize this in writing [FHM94]. Consider the intuitive definition of a subtype: $\mathtt{A}$ is a subtype of $\mathtt{B}$ if we may use an object of type $\mathtt{A}$ in any context expecting an object of type $\mathtt{B}$. If objects of type $\mathtt{A}$ are to be used as $\mathtt{B}$'s, then $\mathtt{A}$-objects must have all of the methods of $\mathtt{B}$-objects. Because method addition is a legal operation on objects in object-based languages, objects with extra methods cannot be used in some contexts where an object with fewer methods may. As an example, a colored point object cannot be used in a context that will add color, but a point object can. For $\mathtt{A}$ to be a subtype of $\mathtt{B}$ then, $\mathtt{A}$'s must contain exactly the same methods as $\mathtt{B}$'s. It is not even possible to specialize the types of methods that appear in both $\mathtt{A}$'s and $\mathtt{B}$'s, since deep record/object subtyping is unsound when method override is a legal operation on objects. This observation is made in [AC94]. The following example, discussed in [AC94], illustrates the problem.

Consider the object types $\mathtt{A}$ and $\mathtt{B}$:

$$\begin{array}{lcl}\mathtt{B} & \stackrel{def}{=} & \mathbf{Object}\,\mathtt{t}.\{\!|\mathtt{x}:\mathtt{Int},\ \mathtt{y}:\mathtt{Real}|\!\} \\ \mathtt{A} & \stackrel{def}{=} & \mathbf{Object}\,\mathtt{t}.\{\!|\mathtt{x}:\mathtt{PosInt},\ \mathtt{y}:\mathtt{Real}|\!\}\end{array}$$

If we allow deep record/object subtyping, $\mathtt{A} <: \mathtt{B}$ since $\mathtt{PosInt} <: \mathtt{Int}$. Now consider an object $\mathtt{a}$ defined as follows:

$$\mathtt{a} \stackrel{def}{=} \{\mathtt{x} = 1,\ \mathtt{y} = \lambda\mathtt{self}.\,\mathtt{ln}(\mathtt{self} \Leftarrow \mathtt{x})\}$$

We can see that $\mathtt{a}$ has type $\mathtt{A}$. By the subsumption rule, we may consider $\mathtt{a}$ to have type $\mathtt{B}$. With this typing, the expression $\{\mathtt{a} \leftarrow \mathtt{x} = -1\}$ is legal. But then sending the message $\mathtt{y}$ to $\mathtt{a}$ produces a run-time type error.

Because of this complete elimination of subtyping for pure object-based languages, the system described in [AC94] does not permit object extension as a run-time operation, instead supporting width subtyping on object types. The paper [FM95] addresses this lack of subtyping by introducing a "sealing" mechanism that converts extensible objects into non-extensible ones. Subtyping in both width and depth is supported for these non-extensible objects.

8.8 Method Specialization

As discussed above, this model supports "mytype" method specialization for the type of the containing object. It does not permit the types of methods to be specialized to subtypes.

8.9 Binary Methods

Since this system does not support subtyping, binary methods can only be applied to objects of the same type. Furthermore, since all object behavior in this system is captured by interfaces, two objects of the same type are interchangeable. There is no hidden state that could be different in two implementations of the same object type. Hence there is no problem defining binary methods in this system.

9 Summary table for three approaches

We summarize the main characteristics of the three theoretical models in Figure 9

The third column here refers to the model given in [FHM94]. In the related model of [AC94], subtyping is made possible by prohibiting the use of $\longleftrightarrow$ on objects. In [FM95], subtyping is made possible by restricting the use of $\longleftrightarrow$ and $\leftarrow$. With subtyping, a limited form of encapsulation may be achieved by promoting an object to a supertype that lists fewer methods. However, once we have changed the static type of an object in this way, we cannot override an existing method with a new one that refers to the hidden methods.

	rec. record	exis. type	ax. object
object type	$\mu t.Ft$	$\exists\, t.\{rep : t, ops : Ft\}$	$\mathbf{Object}\, t.Ft$
object rep.	rec. record	$\langle R, r\rangle : \exists t \ldots$	$\{m_1 = e_1, \ldots, m_k = e_k\}$
message send	field selection	‘`function`	$\Leftarrow$ operation
deleg./class	class	class	delegation
inheritance	record concat.	higher-order poly	$\leftarrow\!+$, $\leftarrow$ operations
dyn. lookup	yes	yes	yes
encapsulation	no protected or private members	protected, private	no protected or private members
subtyping	depth, not width	depth, width	not possible
method spec.	mytype and other types	mytype, others may be possible	mytype only
binary methods	direct	via partially abstract types	direct

Figure 4: Tabular comparison of three theoretical approaches.

10 Open Problems

There are several general problems whose solutions could lead to improvements in practical languages. In addition, the basic problems that are usually addressed by theoretical analysis are largely open. For example, it would be useful to devise understandable proof principles that could be used formally or informally to reason about objects. A specific challenge is to prove equations between objects. This is at least as hard as proving the equivalence of data representations, as shown by a very simple argument: given two implementations of stacks, for example, we can define two objects that are equivalent to each other iff the two implementations of stacks are equivalent.

We list some of the salient language design issues below, discussing a few in more detail in the remainder of this section.

- *Method specialization.* We have discussed method specialization and shown how some of the theoretical models allow it. The only implemented typed language to support a form of `mytype` seems to be Eiffel [Mey92]. However, as shown in [Coo89b], the original Eiffel approach is unsound. It would be possible to allow covariant uses of `mytype`, or something equivalent, in C++ , and to some extent this can already be accomplished using templates. For contravariant occurrences, it seems necessary to separate subtyping and inheritance. The language design trade-offs here remain to be investigated, particularly if some form of "implementation" types are used.

- *Multiple dispatch.* There are some cases in which it would be useful to incorporate multiple dispatch into single-dispatch languages, without sacrificing encapsulation. One motivating example is discussed below.

- *Mixin problem.* This problem is discussed below.

- *Class preconditions.* In C++ templates, a class may be parameterized by a class name. However, there is no way to impose the precondition that the actual class parameter must have certain operations. A solution is to use F-bounded or higher-order bounded quantification. However, this has not been incorporated into implemented object-oriented languages other than the experimental language Rapide [BL90a, MMM91, KLM94]. It would be a useful empirical study to determine whether forms of bounded quantification are sufficient in practice.

- *Modeling of "implementation types".* While most of the implemented languages use a form of "implementation" types, theoretical models tend to use "interface" types. It would be useful to develop a theoretical model of some form of implementation types, verify that it allows useful typing of binary operations, and explore whether the separation of subtyping from inheritance can be carried out cleanly with implementation types. We have started to explore these issues, but have not completed our study.

- *View problem.* A general problem with object-oriented programs is that the names of methods are critical to the use of objects. If we want to print every object in a queue, for example, this is easy if every object has a print method called `print`, but difficult if the names are `print`, `display`, `show` and `draw`. One approach to this problem is to use C++ member pointers, which do not seem to be widely implemented. It is worthwhile to incorporate something along these lines into a theoretical model and to develop programming strategies for writing library routines that would be applicable to objects with varying method names.

10.0.1 Method specialization

In some cases, method specialization seems to be a typing problem: it is clear how a "specialized" method should behave, and it is simply a typing problem to determine how to express this functionality properly. However, this is not always the case. Consider a class of points with equality. If we define a subclass of colored points and wish this to be a subtype, how should we define equality? There are four cases to consider, listed in the table below. A reasonable approach would be to compare `x,y` coordinates whenever one point has no color, and `x,y`, and `color` when both are colored points.

equality	point	colored point
point	`x,y`	`x,y`
colored point	`x,y`	`x,y,color`

It seems that we need to be able to give four cases separately. The class-based program organization does not seem to support this, without using the "typecase" approach it is intended to replace. Multiple-dispatch approaches, such as the Common Lisp Object System (CLOS) work well in this case, as does the "dynamic overloading" model of [CGL92]. But these have drawbacks with respect to encapsulation. Similar situations arise for other binary operations, such as addition on numeric objects.

10.0.2 Mixin problem

The term "mixin" comes from the Lisp Machine Flavors system, which took the name from a menu option offered at a local ice cream store. The main idea of a "mixin" is a class that is useful only for inheritance. More specifically, a class whose methods assume the existence of other methods that they do not define is a mixin. An illustrative example is a scroll-bar mixin, which can be combined via multiple inheritance with any window class to give a scrollable window class. The typing problem associated with this can be solved via parameterization, as described (for interfaces only) in [KLM94, KLMM94]. The C++ program in Appendix D shows how the problem may be solved using templates.

10.0.3 Implementation types

The general problem with typing binary operations in a system with only interface types is described in [MMM91, KLM94, KLMM94]. Briefly, the problem is that if a type gives only the interface to objects, then a single interface can have two implementations using completely different representations for objects. For example, we could have two "set" classes, one using linked lists and the other bit vectors (boolean arrays). In this case, it is difficult to define set union in a type-safe way, since it is necessary to access the internal representations of both sets involved, and these may be different. The approach proposed in [MMM91], which resembles the C++ `friend` concept, seems problematic in an interface-type context. An approach using implementation types is illustrated in Appendix E using C++ `protected` members. However, this program fails to pass the C++ type checker. We can see no reason why the type checker could not be modified to accept this program. Some progress with formalizing forms of "implementation types" is reported in [PT93, KLM94, KLMM94].

Acknowledgements: Thanks to Brian Freyburger and Steve Fisher for several insightful discussions of C++ , to Andy Hung for the drawing in Section 2.5 and to Luca Cardelli for comments on a draft of this paper.

References

[Aba94] M. Abadi. Baby Modula-3 and a theory of objects. *J. Functional Programming*, 4(2):249–283, April 1994. Also appeared as SRC Research Report 95.

[AC91] R.M. Amadio and L. Cardelli. Subtyping recursive types. In *Proc. ACM Symp. Principles of Programming Languages*, pages 104–118, 1991.

[AC94] M. Abadi and L. Cardelli. A theory of primitive objects: untyped and first-order systems. In *Proc. Theor. Aspects of Computer Software*, pages 296–320, Sendai, Japan, 1994. Expanded version to appear in *Information and Computation*.

[AC95a] M. Abadi and L. Cardelli. An imperative object calculus. In *TAPSOFT'95:Theory and Practise of Software Development*, pages 471–485. Springer-Verlag LNCS 915, 1995.

[AC95b] M. Abadi and L. Cardelli. On subtyping and matching. In *Proc 9th European Conference on Object-Oriented Programming*, pages 145–167, Aarhus, Denmark, 1995. Springer LNCS 952.

[AR88] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 277–288, July 1988.

[BDMN73] G.M. Birtwistle, O.-J. Dahl, B. Myhrhaug, and K. Nygaard. *Simula Begin*. Studentlitteratur, Box 1717, S-222 01 Lund, Sweden; Auerbach, Philadelphia, 1973.

[BI82] A.H. Borning and D.H. Ingalls. A type declaration and inference system for Smalltalk. In *ACM Symp. Principles of Programming Languages*, pages 133–141, 1982.

[BL90a] F. Belz and D.C. Luckham. A new approach to prototyping Ada-based hardware/software systems. In *Proc. ACM Tri-Ada'90 Conference*, December 1990.

[BL90b] K. Bruce and G. Longo. A modest model of records, inheritance and bounded quantification. *Information and Computation*, 87(1/2):196–240, 1990.

[Boo91] G. Booch. *Object-Oriented Design with Applications*. Benjamin Cummings, Redwood City, CA, 1991.

[Bru92] K. Bruce. The equivalence of two semantic definitions of inheritance in object-oriented languages. In *Proc. Mathematical Foundations of Programming Language Semantics*, pages 102–124, Berlin, 1992. Springer LNCS 598.

[Bru93] K. Bruce. Safe type checking in a statically-typed object-oriented programming language. In *Proc 20th ACM Symp. Principles of Programming Languages*, pages 285–298, 1993.

[BSv95] K. Bruce, A. Schuett, and R. van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *Proc 9th European Conference on Object-Oriented Programming*, pages 26–51, Aarhus, Denmark, 1995. Springer LNCS 952.

[Car88] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988. Special issue devoted to *Symp. on Semantics of Data Types,* Sophia-Antipolis (France), 1984.

[Car94] L. Cardelli. Extensible records in a pure calculus of subtyping. In C.A. Gunter and J.C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming*, pages 373–426. MIT Press, 1994. Also appeared as DEC Systems Research Center Technical Report 81, 1992.

[CGL92] G. Castagna, G. Ghelli, and G. Longo. A calculus for overloaded functions with subtyping. In *1992 ACM Conf. Lisp and Functional Programming*, pages 182–192, 1992.

[CHC90] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 125–135, January 1990.

[CL94] C. Chambers and G.T. Leavens. Typechecking and modules for multi-methods. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, October 1994.

[Coo87] W. Cook. A self-ish model of inheritance. Manuscript, 1987.

[Coo89a] W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[Coo89b] W.R. Cook. A proposal for making Eiffel type-safe. In *European Conf. on Object-Oriented Programming*, pages 57–72, 1989.

[Coo92] W.R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conf. Object-oriented Programming: Systems, Languages and Applications*, pages 1–15, 1992.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for Self, a dynamically-typed object-oriented programming language. In *SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 146–160, 1989.

[Cur90] Pavel Curtis. Constrained quantification in polymorphic type analysis. Technical report, Xerox PARC technical report CSL-90-1, February 1990. Cornell University PhD Thesis.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[Dij72] E.W. Dijkstra. Notes on structured programming. In O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, editors, *Structured Programming*. Academic Press, 1972.

[ES90] M. Ellis and B. Stroustrop. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[FHM94] K. Fisher, F. Honsell, and J.C. Mitchell. A lambda calculus of objects and method specialization. *Nordic J. Computing (*formerly *BIT)*, 1:3–37, 1994. Preliminary version appeared in *Proc. IEEE Symp. on Logic in Computer Science,* 1993, 26–38.

[FM94] K. Fisher and J.C. Mitchell. Notes on typed object-oriented programming. In *Proc. Theoretical Aspects of Computer Software*, pages 844–885. Springer LNCS 789, 1994.

[FM95] K. Fisher and J.C. Mitchell. A delegation-based object calculus with subtyping. In *Proc. 10th Int'l Conf. Fundamentals of Computation Theory (FCT'95)*, pages 42–61. Springer LNCS 965, 1995.

[Gir71] J.-Y. Girard. Une extension de l'interpretation de Gödel à l'analyse, et son application à l'élimination des coupures dans l'analyse et la théorie des types. In J.E. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 63–92. North-Holland, Amsterdam, 1971.

[Gir72] J.-Y. Girard. Interpretation fonctionelle et elimination des coupures de l'arithmetique d'ordre superieur. These D'Etat, Universite Paris VII, 1972.

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical aspects of object-oriented programming*. MIT Press, Cambridge, MA, 1994.

[HP95] M. Hofmann and B. Pierce. Positive subtyping. In *Proc. 22nd ACM Symp. on Principles of Programming Languages*, pages 186–197, San Fransisco, January 1995. To appear in *Information and Computation*.

[KLM94] Dinesh Katiyar, David Luckham, and John Mitchell. A type system for prototyping languages. In *Proc. 21-st ACM Symp. on Principles of Programming Languages*, 1994.

[KLMM94] Dinesh Katiyar, David Luckham, S. Meldal, and John Mitchell. Polymorphism and subtyping in interfaces. In *ACM Workshop on Interface Definition Languages*, 1994.

[L+81] B. Liskov et al. *CLU Reference Manual*. Springer LNCS 114, Berlin, 1981.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications*, pages 214–223, October 1986.

[LSAS77] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in clu. *Comm. ACM*, 20:564–576, 1977.

[Mey92] B. Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[Mil85] R. Milner. The Standard ML core language. *Polymorphism*, 2(2), 1985. 28 pages. An earlier version appeared in Proc. 1984 ACM Symp. on Lisp and Functional Programming.

[Mit90] J.C. Mitchell. Toward a typed foundation for method specialization and inheritance. In *Proc. 17th ACM Symp. on Principles of Programming Languages*, pages 109–124, January 1990.

[MMM91] J.C. Mitchell, S. Meldal, and N. Madhav. An extension of Standard ML modules with subtyping and inheritance. In *Proc. 18th ACM Symp. on Principles of Programming Languages*, pages 270–278, January 1991.

[Mor73] J.H. Morris. Types are not sets. In 1^{st} *ACM Symp. on Principles of Programming Languages*, pages 120–124, 1973.

[MP88] J.C. Mitchell and G.D. Plotkin. Abstract types have existential types. *ACM Trans. on Programming Languages and Systems*, 10(3):470–502, 1988. Preliminary version appeared in *Proc. 12th ACM Symp. on Principles of Programming Languages,* 1985.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Pie93] Benjamin C. Pierce. Mutable objects. Draft report; available electronically, May 1993.

[PT93] Benjamin C. Pierce and David N. Turner. Statically typed friendly functions via partially abstract types. Technical Report ECS-LFCS-93-256, University of Edinburgh, LFCS, April 1993. Also available as INRIA-Rocquencourt Rapport de Recherche No. 1899.

[PT94] Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in Principles of Programming Languages, 1993, under the title "Object-Oriented Programming Without Recursive Types".

[RA82] J. Rees and N. Adams. T, a dialect of Lisp, or lambda: the ultimate software tool. In *Proc. ACM Symp. Lisp and Functional Programming Languages*, pages 114–122, August 1982.

[Rey83] J.C. Reynolds. Types, abstraction, and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, Amsterdam, 1983.

[Set89] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, 1989.

[Sny86] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. ACM Symp. on Object-Oriented Programming Systems, Languages, and Applications*, pages 38–46, October 1986.

[Ste84] G.L. Steele. *Common Lisp: The language*. Digital Press, 1984.

[Ste87] L.A. Stein. Delegation is inheritance. In *Proc. ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 138–146, 1987.

[Str86] B. Stroustrop. *The C^{++} Programming Language*. Addison-Wesley, 1986.

[Ulm94] J.D. Ulman. *Elements of ML programming*. Prentice Hall, 1994.

[US 80] US Dept. of Defense. *Reference Manual for the Ada Programming Language*. GPO 008-000-00354-8, 1980.

[US91] D. Ungar and R.B. Smith. Self: The power of simplicity. *Lisp and Symbolic Computation*, 4(3):187–206, 1991. Preliminary version appeared in *Proc. ACM Symp. on Object-Oriented Programming: Systems, Languages, and Applications,* 1987, 227-241.

A Queue and Priority Queue

```
exception Empty;

abstype queue = Q of int list
with
     fun mk_Queue()     = Q(nil)
     and is_empty(Q(l)) = l=nil
     and add(x,Q(l))    = Q(l @ [x])
     and first (Q(nil))  = raise Empty
     |   first (Q(x::l)) = x
     and rest (Q(nil))   = raise Empty
     |   rest (Q(x::l))  = Q(l)
     and length (Q(nil)) = 0
     |   length (Q(x::l))= 1 + length (Q(l))
end;

abstype pqueue = Q of int list
with
     fun mk_PQueue()     = Q(nil)
     and is_empty(Q(l)) = l=nil
     and add(x,Q(l))   =
         let fun insert(x,nil) = [x:int]
         |   insert(x,y::l) = if x<y then x::y::l else y::insert(x,l)
         in Q(insert(x,l)) end
     and first (Q(nil))   = raise Empty
     |   first (Q(x::l))  = x
     and rest (Q(nil))   = raise Empty
     |   rest (Q(x::l))  = Q(l)
     and length (Q(nil)) = 0
     |   length (Q(x::l))= 1 + length (Q(l))
end;
```

B Shape program: Typecase version

```
#include <stdio.h>
#include <stdlib.h>


   /*
    * We use the following enumeration type to ``tag´´ shapes.
    * The first field of each shape struct stores what particular
    * kind of shape it is.
    */
enum ShapeTag {Circle, Rectangle};


   /*
    * The following struct Pt and functions newPt and copyPt are
    * used in the implementations of the Circle and Rectangle
    * shapes below.
    */
struct Pt {
   float x;
   float y;
};

struct Pt* newPt(float xval, float yval) {
   struct Pt* p = (struct Pt *)malloc(sizeof(struct Pt));
   p->x = xval;
   p->y = yval;
   return p;
};

struct Pt* copyPt(struct Pt* p) {
   struct Pt* q = (struct Pt *)malloc(sizeof(struct Pt));
   q->x = p->x;
   q->y = p->y;
   return q;
};
```

```
   /*
    * The Shape struct provides a flag that is used to get some static
    * type checking in the operation functions (center, move, rotate,
    * and print) below.
    */
struct Shape {
   enum ShapeTag tag;
};


   /*
    * The following Circle struct is our representation of a circle.
    * The first field is a type tag to indicate that this struct
    * represents a circle.  The second field stores the circle´s
    * center point and the third field holds its radius.
    */
struct Circle {
   enum ShapeTag  tag;
   struct Pt*     center;
   float          radius;
};


   /*
    * The function newCircle creates a Circle struct from a given
    * center point and radius.  It sets the type tag to ``Circle.´´
    */
struct Circle* newCircle(struct Pt* cp, float r) {
   struct Circle* c = (struct Circle*)malloc(sizeof(struct Circle));
   c->center=copyPt(cp);
   c->radius=r;
   c->tag=Circle;
   return c;
};


   /*
    * The function deleteCircle frees resources used by a Circle.
    */
void deleteCircle(struct Circle* c) {
   free (c->center);
   free (c);
};


   /*
    * The following Rectangle struct is our representation of a rectangle.
    * The first field is a type tag to indicate that this struct
    * represents a rectangle.  The next two fields store the rectangles
    * top-left and bottom-right corner points.
    */
struct Rectangle {
   enum ShapeTag  tag;
   struct Pt*     topleft;
   struct Pt*     botright;
};


   /*
    * The function newRectangle creates a rectangle in the location
    * specified by parameters tl and br.  It sets the type tag to
    * ``Rectangle.´´
    */
struct Rectangle* newRectangle(struct Pt* tl, struct Pt* br)  {
   struct Rectangle* r = (struct Rectangle*)malloc(sizeof(struct Rectangle));
   r->topleft=copyPt(tl);
   r->botright=copyPt(br);
   r->tag=Rectangle;
   return r;
};
```

```
   /*
    * The function deleteRectangle frees resources used by a Rectangle.
    */
void deleteRectangle(struct Rectangle* r) {
   free (r->topleft);
   free (r->botright);
   free (r);
};


   /*
    * The center function returns the center point of whatever shape
    * it is passed.  Because the computation depends on whether the
    * shape is a Circle or a Rectangle, the function consists of a
    * switch statement that branches according to the type tag stored
    * in the shape s.  If the tag is Circle, for instance, we know
    * the parameter is really a circle struct and hence that it has
    * a ``center'' component which we can return.  Note that we need
    * to insert a typecast to instruct the compiler that we have a
    * circle and not just a shape.  Note also that this program
    * organization assumes that the type tags in the struct are
    * set correctly.  If some programmer incorrectly modifies a type tag
    * field, the program will no longer work and the problem cannot
    * be detected at compile time because of the typecasts.
    */
struct Pt* center (struct Shape* s) {
   switch (s->tag) {
    case Circle: {
       struct Circle* c = (struct Circle*) s;
       return copyPt(c->center);
    };
    case Rectangle: {
       struct Rectangle* r = (struct Rectangle*) s;
       return newPt((r->botright->x - r->topleft->x)/2,
                    (r->botright->x - r->topleft->x)/2);
    };
   };
};


   /*
    * The move function receives a Shape parameter s and moves it
    * dx units in the x-direction and dy units in the y-direction.
    * Because the code to move a Shape depends on the kind of shape,
    * this function inspects the Shape's type tag field within a switch
    * statement.  Within the individual cases, typecasts are used to
    * convert the generic shape parameter to a Circle or Rectangle as
    * appropriate.
    */
void move (struct Shape* s,float dx, float dy) {
   switch (s->tag) {
    case Circle: {
         struct Circle* c = (struct Circle*) s;
         c->center->x    += dx;
         c->center->y    += dy;
      };
      break;
    case Rectangle: {
       struct Rectangle* r = (struct Rectangle*) s;
       r->topleft->x      += dx;
       r->topleft->y      += dy;
       r->botright->x     += dx;
       r->botright->y     += dy;
    };
   };
};


   /*
    * The rotate function rotates the shape s ninety degrees.  Like
    * the center and move functions, this code uses a switch statement
    * that checks the type of shape being manipulated.
```

```
   */
void rotate (struct Shape* s) {
   switch (s->tag) {
    case Circle:
         /* Rotating a circle is not a very interesting operation! */
      break;
    case Rectangle: {
         struct Rectangle* r = (struct Rectangle*)s;
         float d = ((r->botright->x - r->topleft->x) -
                    (r->topleft->y - r->botright->y))/2.0;
         r->topleft->x  += d;
         r->topleft->y  += d;
         r->botright->x -= d;
         r->botright->y -= d;
      };
      break;
   };
};


   /*
    * The print function outputs a description of its Shape parameter.
    * This function again selects its processing based on the type tag
    * stored in the Shape struct.
    */
void print (struct Shape* s) {
   switch (s->tag) {
    case Circle: {
         struct Circle* c = (struct Circle*) s;
         printf("circle at  %.1f  %.1f  radius %.1f \n",
                c->center->x, c->center->y, c->radius);
      };
      break;
    case Rectangle: {
         struct Rectangle* r = (struct Rectangle*) s;
         printf("rectangle at  %.1f  %.1f   %.1f  %.1f \n",
                r->topleft->x, r->topleft->y,
                r->botright->x, r->botright->y);
      };
      break;
   };
};


   /*
    * The body of this program just tests some of the above functions.
    */
void main() {
   struct Pt* origin = newPt(0,0);
   struct Pt* p1     = newPt(0,2);
   struct Pt* p2     = newPt(4,6);

   struct Shape* s1  = (struct Shape*)newCircle(origin,2);
   struct Shape* s2  = (struct Shape*)newRectangle(p1,p2);

   print(s1);
   print(s2);

   rotate(s1);
   rotate(s2);

   move(s1,1,1);
   move(s2,1,1);

   print(s1);
   print(s2);

   deleteCircle((struct Circle*)s1);
   deleteRectangle((struct Rectangle*)s2);

   free(origin);
   free(p1);
```

```
   free(p2);
};
```

C Shape program: Object-oriented version

```
#include <stdio.h>

   // (The following is a running C++ program, but it does not represent
   // an ideal C++ implementation.  The code has been kept simple so
   // that it can be understood by readers who are not well-versed in C++).


   // The following class Pt is used by the shape objects below.  Since
   // Pt is a class in this version of the program, the ``newPt´´ and
   // ``copyPt´´ functions may be implemented as class member functions.
   // For readability, we have in-lined the function definitions and
   // named both of these functions ``Pt´´; these overloaded functions
   // are differentiated by the types of their arguments.
class Pt {
 public:
   Pt(float xval, float yval) {
      x = xval;
      y=yval;
   };

   Pt(Pt* p) {
      x = p->x;
      y = p->y;
   };

   float x;
   float y;
};


   // Class shape is an example of a ``pure abstract base class,´´
   // which means that it exists solely to provide an interface to
   // classes derived from it.  Since it provides no implementations
   // for the methods center, move, rotate, and print, no ``shape´´
   // objects can be created.  Instead, we use this class as a base
   // class.  Our circle and rectangle shapes will be derived from
   // it.  This class is useful because it allows us to write
   // functions that expect ``shape´´ objects as arguments.  Since
   // our circles and rectangles are subtypes of shape, we may pass
   // them to such functions in a type-safe way.
class Shape {
public:
  virtual Pt* center()=0;
  virtual void move(float dx, float dy)=0;
  virtual void rotate()=0;
  virtual void print()=0;
};


   // Class Circle consolidates the center, move, rotate, and print
   // functions for circles.  It also contains the object constructor
   // ``Circle,´´ corresponding to the function ``newCircle´´ and the
   // object destructor ``~Circle, corresponding to the function
   // ``deleteCircle´´ from the typecase version.  Note that the
   // compiler guarantees that the Circle´s methods are only called on
   // objects of type Circle.  The programmer does not need to keep an
   // explicit tag field in the object.
class Circle : public Shape {
 public:
   Circle(Pt* cn, float r) {
      center_ = new Pt(cn);
      radius_ = r;
   };

   virtual ~Circle() {
      delete center_;
   };

   virtual Pt* center() {
      return new Pt(center_);
```

```
    };

    void move(float dx, float dy) {
       center_->x += dx;
       center_->y += dy;
    };

    void rotate() {
          /* Rotating a circle is not a very interesting operation! */
    };

    void print() {
       printf("circle at  %.1f  %.1f  radius %.1f \n",
              center_->x, center_->y, radius_);
    };

  private:
    Pt*   center_;
    float radius_;
};


    // Class Rectangle consolidates the center, move, rotate, and print
    // functions for rectangles.  It also contains the object constructor
    // ``Rectangle,'' corresponding to the function ``newRectangle'' and the
    // object destructor ``~Rectangle, corresponding to the function
    // ``deleteRectangle'' from the typecase version.  Note that the
    // compiler guarantees that the Rectangle's methods are only called on
    // objects of type Rectangle.  The programmer does not need to keep an
    // explicit tag field in the object.
class Rectangle : public Shape {
  public:
    Rectangle(Pt* tl, Pt* br) {
       topleft_  = new Pt(tl);
       botright_ = new Pt(br);
    };

    virtual ~Rectangle() {
       delete topleft_;
       delete botright_;
    };

    Pt* center() {
       return new Pt((botright_->x - topleft_->x)/2,
                     (botright_->x - topleft_->x)/2);
    };

    void move(float dx,float dy) {
       topleft_->x  += dx;
       topleft_->y  += dy;
       botright_->x += dx;
       botright_->y += dy;
    };

    void rotate() {
       float d = ((botright_->x - topleft_->x) -
                  (topleft_->y - botright_->y))/2.0;
       topleft_->x  += d;
       topleft_->y  += d;
       botright_->x -= d;
       botright_->y -= d;
    };

    void print () {
       printf("rectangle coordinates  %.1f  %.1f   %.1f  %.1f \n",
              topleft_->x, topleft_->y,
              botright_->x, botright_->y);
    };

  private:
    Pt* topleft_;
    Pt* botright_;
```

```
};


   /*
    * The body of this program just tests some of the above functions.
    */
void main() {
   Pt* origin = new Pt(0,0);
   Pt* p1     = new Pt(0,2);
   Pt* p2     = new Pt(4,6);

   Shape* s1 = new Circle(origin, 2 );
   Shape* s2 = new Rectangle(p1, p2);

   s1->print();
   s2->print();

   s1->rotate();
   s2->rotate();

   s1->move(1,1);
   s2->move(1,1);

   s1->print();
   s2->print();

   delete s1;
   delete s2;

   delete origin;
   delete p1;
   delete p2;
}
```

D Specializing comparison operations via templates

```
template <class T> class compare
{
public:
  virtual int operator==(const T&) const =0;
  virtual int operator< (const T&) const =0;

  int operator !=(const T& x) const {return !operator ==(x);}
  int operator >=(const T& x) const {return !operator < (x);}
  int operator <=(const T& x) const {return  operator ==(x) ||
operator <(x); }
  int operator > (const T& x) const {return !operator <=(x);}
};

class Int : public compare<Int>
{
  int val;
public:
  Int(int n)    {val=n;};

  int is_zero() const {if (val==0) return 1; else return 0;}
  void incr()    {val++;};
  void decr()    {val--;};

  virtual int operator== (const Int& x) const
         {if (val == x.val) return 1; else return 0;}
  virtual int operator<  (const Int& x) const
         {if (val < x.val) return 1; else return 0;}
};
```

E Multiple implementations of sets with union

```
struct node {
    node* next;
    int   data;
};
```

```
/*-------------------- virtual base class --------------------------*/
class set {
protected:
  virtual node* commonrep()=0;
public:
  virtual int ismember(int)=0;
  virtual void insert(int)=0;
  virtual void merge(set*)=0;
  virtual void print()=0;
};
/*------------------- two derived classes --------------------------*/

class linkedset : public set {
  node* rep;
protected:
  virtual node* commonrep();
public:
  linkedset();
  virtual int ismember(int key);
  virtual void insert(int k);
  virtual void merge(set* s);
  virtual void print();
};
class arrayset : public set {
  int member[100];
protected:
  node* commonrep();
public:
  arrayset();
  virtual int ismember(int key);
  virtual void insert(int k);
  virtual void merge(set* s);
  virtual void print();
};
/*--------------- member functions for set union  ---------------------*/

void arrayset::merge(set* s){       /* actual parameter could be    */
      node* add = s->commonrep();  /*  arrayset or linkedset       */
      node* n;
      for (n=add;n=n->next;n=0) {
          int k=n->data;
          member[k]=1;
          }
}
```