Interfaces for Strongly-Typed Object-Oriented Programming

Peter S. Canning, William R. Cook, Walter L. Hill, Walter G. Olthoff

Hewlett-Packard Laboratories
P.O.Box 10490, Palo Alto, CA 94303-0971

Abstract

This paper develops a system of explicit interfaces for object-oriented programming. The system provides the benefits of module interfaces found in languages like Ada and Modula-2 while preserving the expressiveness that gives untyped object-oriented languages like Smalltalk their flexibility. Interfaces are interpreted as polymorphic types to make the system sufficiently powerful. We use interfaces to analyze the properties of inheritance, and identify three distinct kinds of inheritance in object-oriented programming, corresponding to objects, classes, and interfaces, respectively. Object interfaces clarify the distinction between interface containment and inheritance and give insight into limitations caused by equating the notions of type and class in many typed object-oriented programming languages. Interfaces also have practical consequences for design, specification, and maintenance of object-oriented systems.

1. Introduction

An important contribution of languages like Ada, Mesa, and Modula-2 is the explicit separation of the interface and implementation of program modules. The interface provides a boundary between the implementations of an abstraction and its clients. It limits the amount of implementation detail visible to clients. It also specifies the functionality that implementations must provide.

In this paper we develop a system of explicit interfaces for object-oriented programming. The primary goal for the system of object interfaces presented here is to allow interface compatibility to be checked at compile-time (eliminating the possibility of certain run-time errors) while preserving the power and flexibility of untyped object-oriented languages like Smalltalk. Interpreting interfaces as types in a polymorphic lambda calculus is our main technique for moving towards that goal. We have implemented a type-checker for a typed programming language and are using it to test the feasibility and value of interfaces for object-oriented programming.

Explicit interfaces are useful for clarifying the role of inheritance in object-oriented programming languages. Using recent work on the semantics of inheritance, we describe three distinct kinds of inheritance in object-oriented programming, corresponding respectively, to objects, classes, and interfaces. Interface inheritance is important in reinforcing the difference between interface compatibility and implementation inheritance. It also gives insight into the limitations of typed object-oriented languages that equate the notions of type and class.

In a more practical vein, this work addresses important needs and opportunities in object-oriented software development. At Hewlett-Packard, large-scale object-oriented software development has suggested the need for the benefits that interfaces provide. Object interfaces provide a useful framework for specifying objects when designing a system. Checking them should yield significant gains in productivity and quality in object-oriented software development. Interfaces may also be the basis of new tools for organizing object-oriented systems.

In Section 2, we review the significance of interfaces in large-scale software development. Section 3 develops the notion of object interface in contrast with object implementation and shows the essential role played by polymorphic types. Section 4 discusses various forms of inheritance in object-oriented programming, and the difference between the interface and implementation hierarchies in typed object-oriented programming. Section 5 briefly compares this work with other approaches to typed object-oriented programming.

2. Interfaces

> "The most important recent development in programming languages is the introduction of an explicit notion of *interface* to stand between the implementation of an abstraction and its clients."
>
> Burstall and Lampson, 1984

When analyzing and designing software systems, interfaces enforce conceptual discipline and provide modular specifications and documentation. Checking interfaces may detect conceptual as well as superficial errors. Interfaces are also used in code to declare or annotate variable or operation names, making programs easier to understand and maintain. They provide for generic modules and allow categorizing implementations, supporting reuse and library administration. In all these forms, interfaces include the idea of a contractual agreement between providers and users of system components, and what it means for the terms of the agreement to be satisfied. On the other hand, the contract is given in an abstract form that deliberately disregards details of how to accomplish its provisions.

Ideally, interfaces should contain a formal description of the behavior of operations, for example as a logical theory. Module connections would then be validated by theorem proving. There is much work in program specification and validation that is motivated by this model [EM85, GHW85]. However, this approach requires mechanical theorem proving support which is currently an obstacle for including it in practical programming languages. Moreover, the application of specification and validation techniques to object-oriented programming is not yet well understood.

Interfaces have been used in a variety of ways in traditional software development. During design, interfaces are typically semi-formal descriptions of how an abstraction interacts with the rest of a system. Together with appropriate naming and documentation conventions, they provide partial specifications. In the implementation phase, interfaces usually describe the names and the parameter and result types of operations supplied by a module as in Ada, Mesa, and Modula-2. In Ada, for example, one distinguishes between package specification and package body. The package specification presents an interface as a list of type, procedure and other declarations. Explicit interfaces are important in realizing data abstraction, data encapsulation and separate compilation.

In Ada, Mesa, and Modula-2, the module interfaces can be interpreted as types [BL84]. While the object interfaces discussed below are structurally different, they admit an analogous type-theoretic interpretation. In all these cases, interface checking reduces to type-checking, for which there is an existing technology. Interfaces consisting only of operation names and types are weaker than complete behavioral specifications. Nevertheless, this compromise has proven successful in practice, both in terms of expressiveness, user acceptance and automated support from compilers and programming environments.

3. Objects and Their Interfaces

3.1. Introduction

In this section we develop the notion of separating interface from implementation in object-oriented programming. Our language model for object-oriented programming is patterned after Smalltalk-80 [GR83]. Object-oriented systems are characterized by *objects* which group together data and the operations for manipulating that data. The operations, called *methods*, can be invoked only by sending *messages* to the object. Sending a message names the operation and supplies necessary arguments, but does not determine how the operation is implemented. The target of the message (the *receiver*) determines what action takes place when the message is received. Since the data (*instance variables*) in an object can be accessed only by the methods of that object, and methods can be invoked only by sending messages, objects are encapsulated data abstractions.

Our goal is to provide a language with the flexibility of Smalltalk, but with the additional structure and protection that comes from making interfaces explicit, and verifying the safety of message sends at compile-time. The rest of this section describes object implementations and then object interfaces. The basic ideas in this section have appeared in previous work, as discussed in Section 5. In our treatment of recursive and polymorphic interfaces, we have attempted greater consistency with object-oriented programming.

3.2. Object Implementations

As described above, an object is a collection of instance variables and methods. Each method provides a concrete implementation of some abstract operation. A method can implement an operation by manipulating the instance variables, giving them new values or sending messages to them, or by sending messages to `self`. Figure 1 shows an example of a definition of an object

```
object origin implements Point
  variable rho:Real := 0.0
  variable theta:Real := 0.0

  method x() returns Real
    return rho * sine(theta)

  method y() returns Real
    return rho * cosine(theta)

  method equal(p:Point)
              returns Boolean
    return (self.x() = p.x()) and
           (self.y() = p.y())
  ...
```

Figure 1

which uses polar coordinates to represent a point in the plane.

This example illustrates a common characteristic of object implementations, namely their recursive structure. The recursive references are indicated by occurrences of `self` in a method. The recursion is not obvious in the textual form because `self` is a pseudo-variable that is implicitly declared. The graphical representation of Figure 2 should make the recursion apparent.

When the `equal` method is invoked, the pseudo-variable `self` is bound to the receiver of the message (i.e. the object `origin`). In this way, when the `equal` method sends the `x` message to `self`, the appropriate `x` method is invoked.

In our system as in Smalltalk, an object may be an *instance* of a *class*. A class is a pattern that can be used to create many objects with common structure. Specifically, a class describes what instance variables each instance will have, the code for each method, and how to create instances. Figure 3 shows a class for objects similar to the object from Figure 1.

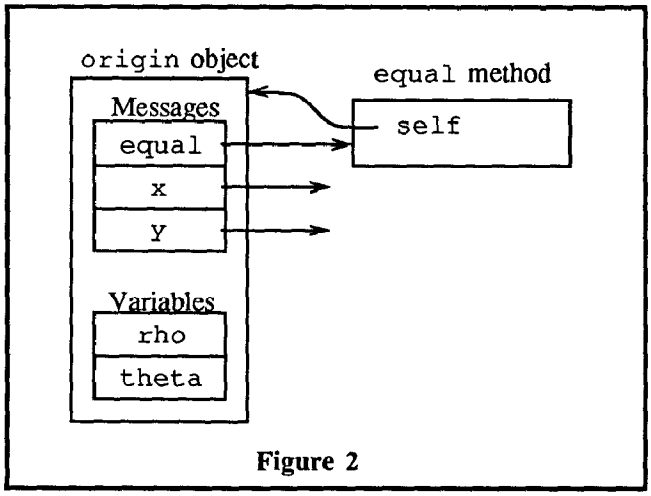

Figure 2

```
class polar_point(x:Real, y:Real)
  implements Point
  variable rho:Real :=
                sqrt((x*x)+(y*y))
  variable theta:Real :=
                arctangent(y/x)

  method x() returns Real
    return rho * sine(theta)

  method y() returns Real
    return rho * cosine(theta)

  method move(dx:Real, dy:Real)
              returns Point
    return
      new myclass(self.x()+dx,
                  self.y()+dy)

  method equal(p:Point)
              returns Boolean
    return (self.x() = p.x()) and
           (self.y() = p.y())
```

Figure 3

Instances of class `polar_point` will share the same structure and methods, but can have different values for their instance variables. The objects that are created from `polar_point` have the same recursive structure that the object `origin` did. Figure 3 illustrates another form of recursion that occurs in classes. This recursion is indicated by occurrences of the pseudo-variable `myclass` which is used to create an object of the same class as `self` (`myclass` has the same meaning as the expression `self class` in Smalltalk). As with object recursion, the graphical presentation of Figure 4 should clarify this structure. In this example, the recursive reference is to class `polar_point`.

3.3. Object Interfaces

As described earlier, an interface provides the information necessary to use a module. In object-oriented systems, the only way to use an object is to send messages to it, so the interface to an object is a description of the messages the object understands. These collections of messages are known as protocols [GR83]. In our system we define an object interface to be such a col-

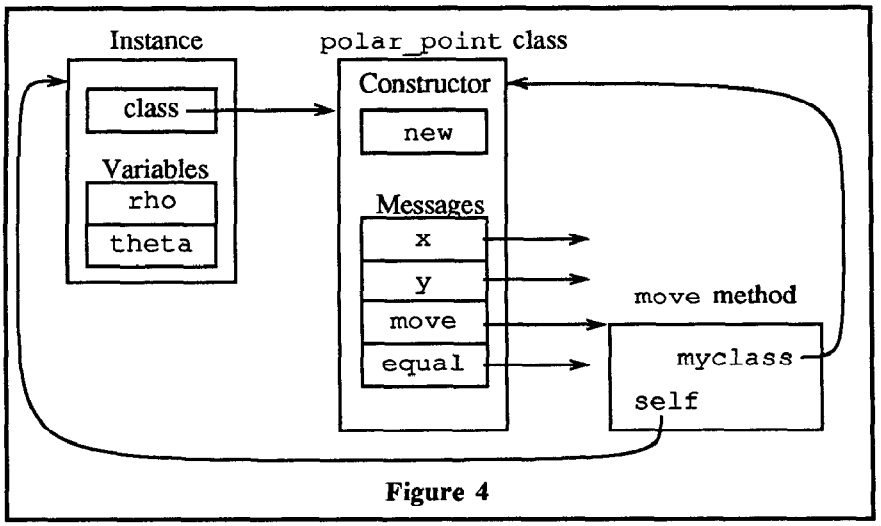

Figure 4

lection, where each message is given with the interfaces for the parameters it requires and for the result it returns. As an example, the interface for the planar point objects described above is shown in Figure 5.

Like object implementations, object interfaces have some interesting structural aspects which are illustrated by this example.

1. Interfaces are constructed from other interfaces. The `Point` interface makes use of the `Real` and `Boolean` interfaces. The `Boolean` interface would contain messages like `not`, `and`, and `or`.
2. Interfaces are often recursively defined. There are two places this recursion appears, in a parameter interface of a message (e.g. `equal`), and in the result interface of a message (e.g. `move`).
3. Interfaces do not contain information about the internal representation of the object. The `x` and `y` messages in the `Point` interface provide information about an object, but they do not imply that the object contains x and y instance variables.

In other programming paradigms, interfaces are used to ensure certain correctness properties of programs. In object-oriented programming, the correctness property is that messages sends are safe. This means that a run-time message error will never occur. Safety is guaranteed by compile-time type-checking which uses the declarations of instance variables and formal parameters (e.g. `p:Point` in class `polar_point`) to ensure that variables always refer to objects that understand all the messages that are sent to them.

A significant part of the flexibility of untyped object-oriented programming is the potential of using any object that can handle a given set of messages (*satisfies* an interface) in a context that sends just those messages (*requires* that interface) independent of how the messages are implemented inside the object. This provides a form of polymorphism that allows multiple implementations of an interface to interact within the same program (in contrast with other programming paradigms). For example, a function that requires a parameter satisfying the `Point` interface and computes its distance from the origin, works equally well on objects that internally use polar coordinates and on objects that use rectangular coordinates as long as both objects understand all the messages in the `Point` interface.

```
interface Point
    x() returns Real
    y() returns Real
    move(Real, Real) returns Point
    equal(Point) returns Boolean
```

Figure 5

On closer examination, we see that this function will also work with objects that understand other messages in addition to those in the `Point` interface. In its most general form, this idea, called *interface containment*[1], provides flexibility in the use of code (and objects). Informally, we say `Big` *contains* `Small` if an object satisfying interface `Big` can be used in a context requiring interface `Small`. More precisely, `Big` contains `Small` if it contains all the messages that `Small` does, and for each of these common messages, each parameter interface for `Small` contains the corresponding parameter interface for `Big` while the result interface for `Big` contains the result interface for `Small`. The important thing to notice here is that the condition on parameter interfaces is reversed.

While this reversal looks counter-intuitive, the example in Figure 6 proves that the non-reversed condition (i.e. claiming `Actual_Parameter` contains `Formal_Parameter`) is incorrect. The function `test` requires an argument satisfying `Formal_Parameter`, since it sends the message `msg` to its formal parameter.

[1] The properties of interface containment, or *subtype polymorphism*, which allows writing programs that work on values that have types that are arbitrary subtypes of a particular type, are well known[Car84].

Calling `test` with an actual parameter `obj` that satisfies `Actual_Parameter` will do `p.msg(s)`. This invokes the `msg` method from `obj` which expects a parameter satisfying `Big`, but passes it `s` which only satisfies `Small`. This is clearly incorrect as the `msg` method could send `s` a message that it doesn't understand (i.e. a message in `Big` but not in `Small`) resulting in a run-time error.

```
interface Formal_Parameter
    msg(Small) returns Boolean

interface Actual_Parameter
    msg(Big) returns Boolean

function test(p:Formal_Parameter)
    variable s:Small = ...
  if p.msg(s) then ...

variable obj:Actual_Parameter=...

test(obj)
```

Figure 6

Since many object interfaces are recursive, it is important to understand what interface containment means for recursively defined interfaces. As is typical for recursive structures, induction is used to determine whether two recursive interfaces are in a containment relation. A recursive interface `Big` contains another recursive interface `Small` if assuming `Big` contains `Small` implies that the corresponding message parameter and result interfaces in `Big` and `Small` are in the required containment relations.

Unfortunately the reversal of the interface containment relation between message parameter interfaces combined with the recursive structure of object interfaces leads to a problem in using interface containment in object-oriented programming. Many pairs of object interfaces that appear intuitively to be in a containment relationship are not. The interface to colored planar points in Figure 7 provides a good illustration.

The `ColorPoint` interface does not contain the `Point` interface because the `equal` message in the `ColorPoint` interface requires a `ColorPoint` parameter and that contradicts the constraint that message parameter interfaces be in a reverse containment relation.

Interface containment with structured interfaces is sufficient to type-check many object-oriented programs, but is too restrictive for the recursive interfaces that are appropriate for many Smalltalk classes. Section 4 describes another relation between interfaces, which solves this problem.

```
interface ColorPoint
  x() returns Real
  y() returns Real
  color() returns Color
  move(Real,Real) returns ColorPoint
  equal(ColorPoint) returns Boolean
```

Figure 7

3.4. Parametric Polymorphism

In addition to subtype polymorphism, there is another kind of polymorphism known as parametric polymorphism [CW85] which provides the ability to write programs admitting a form of type parameterization. Due in part to their lack (or limited form) of parametric polymorphism, most strongly-typed object-oriented programming languages have sacrificed flexibility when compared with untyped object-oriented languages [BHJ87, Str86].

```
class stack[T] implements Stack[T]
  variable elts:List[T] :=
                new List[T]

  method push(x:T) returns nothing
    elts := elts.prepend(x)

  method pop() returns T
    variable tmp:T := elts.head()
    elts := elts.tail()
    return tmp

  method top() returns T
    return elts.head()

  method empty?() returns Boolean
    return elts.empty?()
```

Figure 8

One use of parametric polymorphism is for building generic (or parameterized) modules. A simple object-oriented example is the stack class of Figure 8.

Clearly, nothing in class `stack` depends on what type of elements are stored in the stack. It is important to be able to provide one stack class that can be reused for elements satisfying different interfaces and to be able to type-check the class. For example, the result of sending `top` to a stack of windows must satisfy the

Window interface. The major idea introduced by parametric polymorphism is that of an *interface variable*. In class stack, T is an interface variable representing the interface of the objects stored on the stack. The introduction of T makes it possible to declare variables with interface T and define methods that take arguments or return results with interface T, and still type-check the class definition.

Just as it is possible to have parameterized classes, it is also possible to have parameterized interfaces. Class stack claims that it implements the parameterized interface Stack[T] shown in Figure 9. Interface Stack uses the interface variable T to express the fact that it is independent of the interface of the elements stored in the stack.

```
interface Stack[T]
    push(T) returns nothing
    pop() returns T
    top() returns T
    empty?() returns Boolean
```

Figure 9

For the stack class, T represents an arbitrary interface. Sometimes it is necessary to restrict the interface variable to interfaces that contain certain messages. Combining the ideas of interface variables and interface containment into a notion called bounded parametric polymorphism [CW85, CCH89] accomplishes this restriction. Again this can be most easily explained using an example (Figure 10).

The key point is that the member? method sends the equal message to a variable with interface T, so it would not type-check without restricting T to the interfaces that contain an equal message. This restriction also prevents the creation of sets of elements that do not have an equal operation.

The forms of parametric polymorphism shown so far are not specific to object-oriented programming. Similar functionality is available, for example, in ML [HMT88] and Ada [DOD83]. The following form for methods however, is specific to and needed for object-oriented programming. It requires a type system strictly more powerful than those of most languages[2].

[2]The first-order polymorphism found in ML [HMT88] will not handle this situation since the quantified type variable does not occur at the outermost level. The interfaces described here are interpreted in an extension of the higher-order system of Girard-Reynolds [Gir72, Rey74] , which is sufficient, but more complex[CHO88].

```
class set[T] implements Set[T]
  where T contains
          equal(T) returns Boolean
    ...
  method member?(x:T) returns Boolean
    variable y:T
        ...
    if x.equal(y) then return TRUE
        ...
```

Figure 10

Suppose we want to add a method called select_nearest to class polar_point that takes a set of objects satisfying the Point interface, and returns the element of the set that is closest to the receiver. An obvious way to add select_nearest to the Point interface would be:

```
select_nearest(Set[Point])
                returns Point
```

This is not really what we want. If we send select_nearest to an object along with a set of objects satisfying the ColorPoint[3] interface, the type-checker has been told that we get back an object satisfying *only* the Point interface (i.e. we lose the information that we selected an element from a set of ColorPoint objects). The solution is to use bounded parametric polymorphism again, this time on the select_nearest method rather than on a class:

```
select_nearest[T contains Point]
                (s:Set[T]) returns T
```

This form of bounded parametric polymorphism restricts T to interfaces containing the Point interface. It also ensures that the type-checker will know that select_nearest returns a object that satisfies the appropriate interface. Bounded polymorphism as presented here solves the problem for interfaces containing Point. The full solution, which also handles interfaces like ColorPoint that are closely related but do not contain Point, requires a generalization of bounded polymorphism as described in [CCH89]. It is based on the idea of interface inheritance developed in Section 4.4.

[3] The perceptive reader will recall that ColorPoint does not contain Point. For the moment assume that it does.

```
object instrumented
  implements Point
  inherits origin
    variable x_count:Integer := 0
    variable y_count:Integer := 0

    method x() returns Real
      x_count := x_count + 1
      return origin.x()

  ...
```

Figure 11

4. Inheritance

4.1. Introduction

This section discusses inheritance in object-oriented languages and its relation to interfaces. The inheritance mechanism used here provides a unified framework that describes the traditional forms of delegation and class inheritance, and exposes new connections with interfaces. Interfaces have their own form of inheritance which is important for understanding the interfaces arising from delegation [Lie86] and class inheritance [GR83], and explaining the difference between inheritance and interface containment.

As recent work has shown, inheritance is intimately connected to self-reference [Coo89a, CP89, Red88]. In these models, inheritance is defined as a mechanism for deriving modified versions of recursive structures. The characteristic pattern of inheritance is that self-reference in the inherited structure is changed to refer to the modified definitions. One result of this interpretation is that *every recursive construct is an opportunity for inheritance*. Recalling the three forms of self-reference (object, class, and interface) presented in the last section, this principle gives three corresponding kinds of inheritance. The first two of these correspond respectively to notions of delegation and class inheritance. The notion of interface inheritance is new; it plays a crucial role in representing the interfaces for objects obtained from the two other kinds of inheritance. The examples that follow serve to illustrate this relationship between recursion and inheritance and to apply it to strongly-typed object-oriented programming.

4.2. Object Inheritance

Object inheritance is the form of inheritance associated with object recursion that was illustrated in Figure 2. It is used to create a new object from an existing object (called a *prototype*) by describing how the new object differs from the prototype. These differences are expressed in terms of adding or replacing methods, and adding instance variables. Methods that are not replaced are inherited from the prototype. Figure 11 demonstrates object inheritance by creating an instrumented planar point object (Figure 1), that counts how many times the `x` and `y` messages are sent to it.

This object shows the change in recursive reference that is characteristic of inheritance. When the `equal` message is sent to `instrumented`, no `equal` method is found so the `equal` method defined by its prototype (object `origin`) is invoked. However, when the `equal` method then sends the `x` message to `self` (i.e. `self.x()`), the `x` method defined for `instrumented` is invoked[4]. Thus the result of object inheritance is that the instrumented `x` method is invoked even by message sends from methods originally defined for object `origin`. Reinterpreting `self` in the context of `instrumented` accomplishes this result as illustrated graphically in Figure 12.

An object's interface includes all the messages from its prototype's interface. In particular, an object can handle all the messages that its prototype sends to `self`. Intuitively this suggests that an object's interface contains the interface of its prototype. While this is often true, it does not hold in general as discussed below.

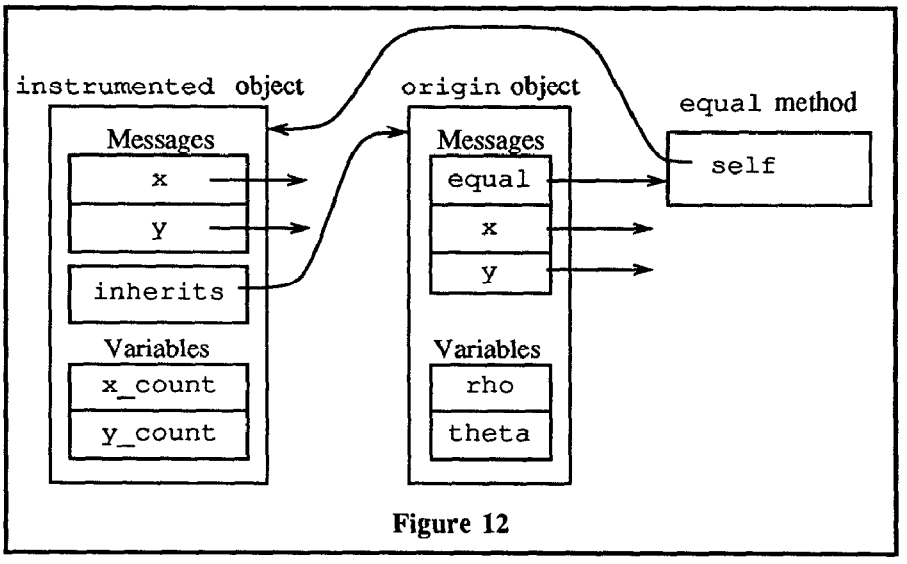

Figure 12

4.3. Class Inheritance

Class inheritance is the form of inheritance associated with the recursion in class definitions described in Figure 4. It is essentially the form of inheritance found in most object-oriented languages. It is used to create a new class, called the *child class* or *subclass*, from an existing *parent class* by describing how the instances of the child class differ from those of the parent class. As with object inheritance, class inheritance provides for adding and redefining methods, and adding instance variables In addition, class inheritance takes into account recursion in the pseudo-variable `myclass`. Figure 13 illustrates creating a class of colored points that inherits from class `polar_point` (Figure 3).

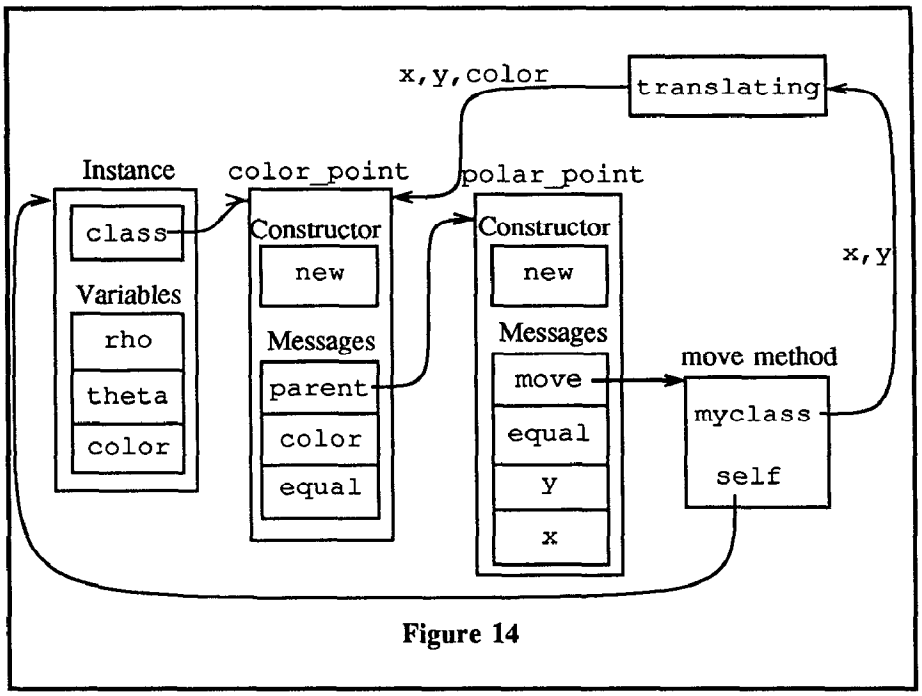

Figure 14

```
class color_point(a:Real,
                  b:Real,
                  c:Color)
  implements ColorPoint
  inherits polar_point(a,b)
    translating new myclass(x,y)
    to new myclass(x,
                   y,
                   self.color())

  variable color:Color = c

  method color() returns Color
    return color

  method equal(p:ColorPoint)
               returns Boolean
    return (self.x() = p.x()) and
           (self.y() = p.y()) and
           (self.color() =p.color())
```

Figure 13

Class inheritance reinterprets `self` in the context of the new class just as object inheritance did. In addition, it also reinterprets the pseudo-variable `myclass`. For example, class `color_point` inherits the `move` method from `polar_point`, which contains:

```
return new myclass(self.x() + dx,
                   self.y() + dy)
```

Using class inheritance, **new** `myclass` is reinterpreted as creating instances of class `color_point`. However, since instantiating class `color_point` takes an extra color argument in addition to the initialization arguments for class `polar_point`, the inheritance mechanism must provide a translation to create instances of class `color_point` using the initialization parameters provided by class `polar_point`. This translation is provided in the **inherits** clause in Figure 13. Using this translation, the class inheritance mechanism ensures that when a colored point is moved, it retains its color. Figure 14, shows the resulting system of classes and objects with the new connections for `myclass` and `self` in the inherited `move` method.

The translation mechanism described here, which is required for general class inheritance, goes beyond the capabilities of current object-oriented languages [CCH, Coo89a]. While most object-oriented languages have a notion of class inheritance, there are considerable differences in the treatment of `myclass`. In Eiffel [Mey87], creating values of type **like** `Current` corresponds to calling **new** `myclass`, while in Smalltalk, one invokes `self class new`. In some languages, including

[4] In this example, one might not have wanted calls to `equal` to increment the counters. This effect could be provided by extending the usual notion of inheritance to distinguish *loose* and *tight* references to `self`. Devising language mechanisms to express and apply such alternatives is an interesting problem which we are currently investigating.

C++, there is no corresponding mechanism[5]; classes are referred to only by explicit name.

When comparing interfaces for the instances of parent and child classes, the same considerations apply as were discussed for object inheritance above. Class inheritance, however, introduces additional connections between the recursive structures of classes and interfaces. Notice, for example, that the interface to the `move` method for a `polar_point` object is:

```
move(Real,Real) returns Point
```

while for a `color_point` it is:

```
move(Real,Real) returns ColorPoint
```

Methods that return `self` in object or class inheritance, or **`new`** `myclass` in class inheritance, impose recursion in the interfaces. As a result, the following notion of interface inheritance plays a central role in describing the interfaces of objects arising from object and class inheritance.

4.4. Interface Inheritance

Interface inheritance is the form of inheritance associated with the recursive definition of object interfaces described in Section 3.3. Interfaces are typically modified by adding new messages or by changing the parameter or result interfaces of a message. Interface inheritance allows this change to take place so that every recursive reference is changed as well. As an example, the interface `ColorPoint` can be viewed as inheriting interface `Point`, because a message is added and occurrences of `Point` are changed to `ColorPoint`. Abstractly, this mechanism is the same as the change of `self` in object or class inheritance.

Interface inheritance characterizes a collection of interfaces that have similar recursive structure. All interfaces that inherit from `Point` will have the form:

```
interface X
    ...
    move(Real,Real) returns X
    equal(X) returns Boolean
    ...
```

This collection is different from the interfaces that are related by containment. For example, `ColorPoint` does not contain `Point`, even though it is derivable by inheritance.

[5] The effect can be simulated (although typing can still be a problem) by factoring all calls to the constructor through an ordinary message that must be manually redefined in each subclass.

Interface inheritance provides exactly the form of interface change that is required to express the effect of object and class inheritance. Specifically, `color_point` is defined by inheriting `polar_point`, and the interface `ColorPoint` is produced by interface inheritance from `Point`. Since `ColorPoint` does not contain `Point`, this shows that the class inheritance hierarchy and the interface containment hierarchy are distinct.

The Eiffel language [Mey87] provides an implicit form of interface inheritance that occurs when the type **`like`** `Current` is used to type a method. If this method is defined by a class `P` then **`like`** `Current` is bound to `P` for instances of class `P`. But if the method is inherited by a class `C`, then **`like`** `Current` represents `C`. However, the Eiffel type system in its current form is unsound [Coo89] because it assumes that instances of sub-classes always satisfy the interfaces of their superclasses, which we have shown not to be the case in general.

A more detailed analysis of the properties of object, class, and interface inheritance, and the relations between them, is currently in preparation[CCH].

5. Related Work

There have been several attempts to introduce interfaces into object-oriented programming. Efforts within the object-oriented language community include the work of Borning and Ingalls [BI82], the Emerald group [BHJ87], and Johnson et al. [JGZ88], all of whom have a notion of interface as message protocol. [BI82] and [JGZ88] describe type systems for Smalltalk with types based both on classes and protocols. The [BI82] work is less formal and does not distinguish interface containment from inheritance; nevertheless, it captures much of the essential structure of object interfaces. The more recent [JGZ88] work is interesting for its mixing of class-based and interface types, and has a limited form of polymorphism as found in ML. Emerald introduces interfaces to provide uniformity in a distributed object system. However, the interface system supports only limited parametric polymorphism and there is no notion of inheritance. Similarly, Modula-3 [CDK89] has notions of object and interface, but without polymorphism.

In most typed object-oriented languages, including Simula, C++, and Eiffel, the notion of type is different from ours. In those languages, classes are types, so that a type gives information about how objects are implemented. This improves performance and also provides the benefits of static type-checking, but at a cost

in flexibility, which is significant for system development. In addition to collapsing the distinction between interface and implementation, the notions of interface containment and inheritance are confused[Coo89]. The result is that programs that would be correct in Smalltalk cannot be type-checked in these languages.

From another direction, object-oriented programming has been studied in the typed functional programming community [Car84, Car86, CW85, CM88, JM88, BL88, Wan89]. While many features of our system were derived empirically, its basic structure is generally consistent with that earlier work. Our treatment of inheritance and polymorphism, however, gives a more faithful representation of conventional object-oriented programming than other type-theoretic work [CHO88, CCH89, CCH]. Our treatment of subtyping is taken from Cardelli [Car88]. The paper [BCG89] gives a semantics for a language which shares most of the properties of our object interfaces.

The structure of object interfaces is different from interfaces in Mesa and Modula-2 modules or Ada packages. For example, with an Ada package providing complex numbers, the interface is used to check that the *package* is used correctly, but it is not an interface to individual complex numbers provided by the package. In contrast, object interfaces are interfaces for individual data values (objects). This reflects a difference in the way object-oriented programming distributes data and procedural information.

Interfaces, and especially the notion of separate containment and inheritance hierarchies, suggest new ways of organizing object-oriented systems. Tools for managing such systems, such as the Smalltalk system browser [GR83] which relies on the class hierarchy, can be extended. Not only are there new hierarchies available, there are also important relations between them which can add power to these new tools. Interface checking makes it possible to maintain relations dynamically between implementations and the interfaces that they satisfy. A software library can then organize classes using both interfaces and inheritance to classify them. The development of interface-based tools is an active area of applied research at Hewlett-Packard Laboratories.

6. Conclusion

Interfaces for object-oriented programming are a valuable addition to the paradigm. By using the interpretation of interface as a type that formalizes the notion of an object protocol it is possible to develop a rich system of interfaces providing strong typing without compromising the programming model of untyped Smalltalk.

Analogous to Ada's package interfaces, explicit object interfaces have many uses which bear on quality and productivity in object-oriented software development. These include compatibility checking, system design and documentation, and software reuse. Interfaces are also the basis of the design of new tools for object-oriented software development.

Interfaces help to clarify the role of inheritance in object-oriented programming, and to distinguish between interface containment and implementation inheritance. There are different forms of inheritance corresponding to objects, classes, and interfaces. In particular, delegation and class inheritance are given a unified treatment. Understanding their respective roles, along with the roles of interface containment and parametric polymorphism, leads to improved design of systems and languages for object-oriented programming.

7. Acknowledgments

We would like to thank John Mitchell for helping us with our type system and its semantics, and for bringing related work to our attention. We are grateful to Alan Snyder for enabling and encouraging us to pursue this work, and generously helping us along the way.

8. Bibliography

[BCG89] V. Breazu-Tannen, T. Coquand, C. Gunter, and A. Scedrov. "Inheritance and explicit coercion". In *Logic in Computer Science*, 1989.

[BHJ87] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and abstract types in Emerald". *IEEE Transactions on Software Engineering*, SE-13(1), 1987.

[BI82] A. H. Borning and D. H. Ingalls. "A type declaration and inference system for Smalltalk". In *ACM Symposium on Principles of Programming Languages*, 1982.

[BL84] R. Burstall, B. Lampson. "A kernel language for abstract data types and modules". In *Semantics of Data Types*, LNCS 173. Springer-Verlag, 1984.

[BL88] K. Bruce and G. Longo. "A modest model of records, inheritance and bounded quantification". In *Logic in Computer Science*, 1988.

[Car84] L. Cardelli. "A semantics of multiple inheritance". In *Semantics of Data Types*, LNCS 173. Springer-Verlag, 1984.

[Car86] L. Cardelli. "Amber". In *Combinators and Functional Programming Languages*, LNCS 242. Springer-Verlag, 1986.

[Car88] L. Cardelli. "Structural subtyping and the notion of power type". In *ACM Symposium on Principles of Programming Languages*, 1988.

[CCH] W. Cook, P. Canning, and W. Hill. "Inheritance in strongly-typed object-oriented programming". In preparation.

[CCH89] P. Canning, W. Cook, W. Hill, W. Olthoff, and J. Mitchell. "F-Bounded quantification for object-oriented programming". In *Functional Programming Languages and Computer Architectures*, 1989.

[CDK89] L. Cardelli, J. Donahue, B. Kaslow, and G. Nelson. "The Modula-3 type system". In *ACM Symposium on Principles of Programming Languages*, 1989.

[CHO88] P. Canning, W. Hill, and W. Olthoff. "Towards a kernel language for object-oriented programming". Technical Report STL-88-21, Hewlett-Packard Laboratories, 1988.

[CM88] L. Cardelli and J. Mitchell. "Semantic methods for object-oriented languages", part 2. OOPSLA'88 Tutorial on Semantics of Inheritance.

[Coo89] W. R. Cook. "A proposal for making Eiffel type-safe". In *European Conference on Object-Oriented Programming*, 1989.

[Coo89a] W. R. Cook. "A denotational semantics of inheritance". Ph. D. Thesis, Brown University, 1989.

[CP89] W. R. Cook and J. Palsberg. "A denotational model of inheritance and its correctness". In *Object-Oriented Programming Systems, Languages, and Applications*, 1989 .

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

[DOD83] U. S. Department of Defense. *Reference Manual for the Ada Programming Language*. 1983.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1*. Springer-Verlag, 1985.

[Gir72] J-Y. Girard. "Interpretation fonctionelle et elimination des coupres de l'arithmetique d'order superieur". These d'Etat, Paris, 1972.

[GR83] A. Goldberg and D. Robson. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, 1983.

[GHW85] J. Guttag, J. Horning, J. Wing. "Larch in five easy pieces". Technical Report 5, Digital Equipment Corp. - Systems Research Center, 1985.

[HMT88] R. Harper, R. Milner, M. Tofte. "The Definition of Standard ML, Version 2". LFCS Report, University of Edinburgh, 1988.

[JGZ88] R. Johnson, J. Graver, and L. Zurawski. "TS: An optimizing compiler for Smalltalk". In *Object-Oriented Programming Systems, Languages, and Applications*, 1988.

[JM88] L. Jategaonkar and J. Mitchell. "ML with extended pattern matching and subtypes". In *Lisp and Functional Programming*, 1988.

[Lie86] H. Lieberman. "Using prototypical objects to implement shared behavior in an object-oriented System". In *Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[Mey87] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1987.

[Red88] U. Reddy. "Objects as closures: abstract semantics of object oriented languages". In *Lisp and Functional Programming*, 1988.

[Rey74] J. Reynolds. "Towards a theory of type structure". In J. Loeckx, ed., *Conference on Programming*. Springer-Verlag, New York, 1974.

[SCB86] C. Schaffert, T. Cooper, B. Bullis. "An introduction to Trellis/Owl". *Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[Sny86] A. Snyder. "Encapsulation and inheritance in object-oriented programming languages". In *Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[Str86] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1986.

[Wan89] M. Wand. "Type inference for record concatenation and inheritance". In *Logic in Computer Science*, 1989.