On the relationship between classes, objects and data abstraction

Kathleen Fisher[1] and John C. Mitchell[2] *

[1] AT&T Research, 600 Mountain Ave., Murray Hill, NJ 07974
kfisher@research.att.com
[2] Computer Science Department, Stanford University, Stanford, CA 94305
mitchell@cs.stanford.edu

Abstract. While most object-oriented programming is done in class-based languages, the trend in theoretical study has been to develop formal systems that are object-based, without classes and often without explicit inheritance mechanisms. This paper studies the correspondence between object primitives and class constructs of the form found in C++, Eiffel, and Java. The main qualitative insight is that such classes require both an extensible aggregate, to serve as the basis for inheritance, and a non-extensible form of object to support subtyping. We compare three approaches to modeling classes, the first using records of object components called "premethods" and the latter two using an extensible form of object called a "prototype." While the first approach uses fewer primitive operations on objects, it does not provide several important features of class-based languages. In the latter two approaches, we overcome these deficiencies by combining prototypes with standard abstraction mechanisms. All three treatments of classes use translations into provably sound object calculi.

1 Introduction

There are several forms of object-oriented programming languages, including class-based languages such as C++ [ES90, Str94], Eiffel [Mey92], Java [AG96] Simula [BDMN73], and Smalltalk [GR83], prototype-based languages such as Self [US91] and Obliq [Car95], and multi-method-based approaches such as CommonLisp [Ste84] and Cecil [Cha95]. This paper is concerned with the study of class-based languages and the relationship between three language constructs: classes, prototype-based inheritance primitives, and traditional data abstraction of the form found in languages such as CLU [LSAS77, L$^+$81], Ada [US 80], and ML [MTH90]. We begin by identifying some essential properties of the class constructs that appear in class-based object-oriented languages. This analysis gives rise to evaluation criteria that we will later use to assess proposed class constructs. We then present three ways of representing classes in terms of basic operations on objects. These representations are evaluated using both our design

* Supported in part by NSF Grant CCR-9303099.

criteria and comparisons between the kinds of object operations involved. The differences between the three translations from classes into object operations provide some insight into the suitability of prototype-based object systems for carrying out traditional class-based programming. To the best of our knowledge, the translations also provide the first type soundness proofs for standard forms of class constructs.

The first treatment of classes is based on an encoding into record operations. A version of this encoding appears in [AC96c, AC96b], where it is used to show how a pure object calculus can be used to represent simple class constructs. In this model, each class has three parts: a type for the instances of the class, a constructor function to produce instances, and a record of object-part implementations. One deficiency of this model is that it is not possible to inherit initialization code. There are also limitations in the way that changes may be propagated through a class hierarchy. Finally, the translation uses structural subtyping on object types to account for the similarities between classes related via inheritance; consequently, it is not possible to explicitly control the inheritance hierarchy or the implementation of objects with any object type. An advantage of this approach is that this form of class may be expressed using a relatively straightforward and understandable object system, without cloning or object extension. While we believe the lack of constructor inheritance and the inability to control the class hierarchy are serious limitations (as elaborated in this paper), others may disagree [AC96b, RR96].

The second approach to classes, which addresses the deficiencies in the first approach, provides the central features found in Java, Eiffel, and C++. The fundamental improvements arise from the use of extensible objects to define extensible constructors and a form of extensible abstract datatype to provide extensible types with implementation guarantees. In the process, we gain explicit control over the subtyping hierarchy without invalidating ordinary structural subtyping. One appeal of this translation is that it clearly shows how classes may be viewed as an orthogonal combination of pure operations on objects (providing aggregation but no encapsulation) and data abstraction (providing encapsulation but no aggregation).

The third translation of classes into object operations is an elaboration of the second, allowing the type of base-class methods to be refined in derived classes. This ability to refine the types of methods has been called "method specialization" or "mytype specialization" in the literature [Mit90, BSv95]. Although method specialization complicates the typing issues considerably, we are again able to show that classes may be viewed as an orthogonal combination of object and data abstraction primitives. The third translation requires a target language with extensible objects, specific forms of higher-order polymorphism, and so-called variance analysis.

All of the translations yield type soundness results for their classes as a consequence of the type soundness of their target object calculi. The soundness proof for the first target calculus appears in [AC96b]; the analysis of extensible object calculi is developed in full in [Fis96]. For brevity and to present the major design issues as clearly as possible, we do not repeat the technical details in this paper. The interested reader may consult our previous publications [FHM94, FM95a] for discussions of type soundness proofs for extensible object calculi, and [FM95b] for a general discussion of type systems, object calculi, and object-oriented programming. An alternate view of many of these issues may be found in the excellent book by Abadi and Cardelli [AC96b].

The remainder of this paper is structured as follows. In the next section, we discuss the use of classes in traditional object-oriented programming, developing criteria for evaluating the properties of various class constructs. Section 3 gives the syntax of a sample class construct and two examples to be used in comparing different class mechanisms. In Section 4, we describe our notation and discuss the background material necessary to understand the first class encoding. Sections 5 to 7 describe and analyze three translations of the sample class syntax into specific formal calculi, each inducing a different set of restrictions on the use of objects and inheritance. Sections 8 and 9 describe related work and conclude.

2 Classes

In contemporary class-based object-oriented languages such as Eiffel, Java, and C++, classes are the main tool for program organization. Since program organization is a complex task, classes are designed to serve a number of purposes. In brief, a class declaration provides a mechanism for creating a family of objects, all sharing some or all of their implementation. Inheritance, which enables the reuse of object implementations, is structured along class lines, with derived classes incorporating some or all of the implementation of their base classes. Classes typically provide access control mechanisms, which allow programmers to hide implementation details. A class declaration also defines a type, namely, the type that all objects created by this class will share. Finally, in contemporary languages, class declarations have explicitly determined the subtyping relation between object types. We discuss each of these issues in turn.

2.1 Defining Objects

A class declaration provides a mechanism for creating a family of objects. Typically, all objects created from a single class will have the same representation, with the values of some data field differing from instance to instance. For example, the following class declaration, written in a Java-like pseudo-code, defines a family of `Point` objects, created (or *instantiated*) by calling the *constructor* function of the class. For simplicity, we illustrate the main ideas using one-dimensional points.

$$
\begin{array}{rcl}
\textbf{class}\ \texttt{Point}\ \{ & & \\
\texttt{x} & : & \mathit{int} \\
\texttt{setX} & : & \mathit{int} \rightarrow \mathit{Point} \\
\texttt{getX} & : & \mathit{int} \\
\texttt{move} & : & \mathit{int} \rightarrow \mathit{Point} \\
\textbf{constructor}\ \texttt{newPoint} & : & \mathit{int} \rightarrow \mathit{Point} \\
\} & &
\end{array}
$$

Each object constructed from this class has an `x` component, representing the point's geometric location. The `setX` and `getX` components write and read the `x` coordinate, respectively, while `move` changes the location by a given offset. For simplicity, we omit the implementations of these components. We will see the need for `setX` and `getX` when we discuss access controls in Section 2.3.

For efficiency, components that are the same for all objects instantiated from a given class are typically shared. In Smalltalk for example, data, called *instance variables*, are stored within each object, while functions, called *methods*, are shared by all objects instantiated from the same class. The distinction between components allocated per object and per class does not always fall between data and functions, but that division is a common one. In the `Point` example, the `x` component would probably be an instance variable, set to different values in different `Point` objects, while `setX`, `getX`, and `move` would be shared among all instances of the class.

Programmers create *instances* of the `Point` class (*i.e.*, `Point` objects) by calling the `Point` class constructor function `newPoint`, here distinguished by the keyword **constructor**. We again omit the implementing code. Since objects are created by calling constructor functions, we must be able to call the constructors of a class before any objects have been created. Hence constructor functions are not components of the objects they create. Despite this independence, constructor definition is an important part of class design since each object must be initialized in accordance with any intended object invariants.

2.2 Defining Related Objects Via Inheritance

Inheritance allows a new family of objects to be defined as an incremental modification to another. For example, we may use inheritance to define the class `ColorPoint` from `Point` as follows:

$$
\begin{array}{rcl}
\textbf{class}\ \texttt{ColorPoint : Point}\ \{ & & \\
\texttt{c} & : & \mathit{color} \\
\texttt{turnRed} & : & \mathit{ColorPoint} \\
\texttt{getC} & : & \mathit{color} \\
\textbf{constructor}\ \texttt{newColorPoint} & : & \mathit{int} \rightarrow \mathit{color} \rightarrow \mathit{ColorPoint} \\
\} & &
\end{array}
$$

The `ColorPoint` class *inherits* the `Point` `x`, `setX`, `getX`, and `move` methods and adds three new ones: `c`, `turnRed`, and `getC`. The `Point` class is said to be a *base*, *parent* or *super* class and `ColorPoint` a *derived*, *child* or *sub* class. In addition to adding components, derived classes may also modify existing methods during inheritance. For example, the `ColorPoint` class may redefine the `Point move` method so that `ColorPoint` objects change color whenever they are moved. The constructor function `newColorPoint` is responsible for creating fully initialized `ColorPoint` objects.

As this example illustrates, inheritance is a code-reuse mechanism. In principle, for every object or class of objects defined via inheritance, there is an equivalent definition that does not use inheritance, obtained by expanding the definition so that the inherited code is duplicated. The importance of inheritance is that it saves the effort of duplicating (or reading duplicated) code, and that when one class is implemented by inheriting from another, changes to the parent are automatically reflected in its descendants. This mechanism has a significant impact on program maintenance and modification.

A benefit of class-based inheritance is that classes are essentially static structures. Class-based inheritance may be implemented as a compile-time operation, greatly simplifying the static checking involved in compiling class declarations. In particular, the correctness of a derived class declaration depends on both the presence and absence of certain components in its base class. This information is available at compile time if the base class declaration is known. If we want to statically check *delegation*, a run-time form of inheritance based on individual objects instead of classes, then it is necessary to include both presence and absence of components as part of the static type system. This complicates subtyping, as discussed in [FHM94, FM95b, FM95a], for example.

2.3 Access Restrictions

Classes typically provide access controls, which are used to restrict the visibility of various object components. Some common visibility levels are *private*, for use only within a given class implementation; *protected*, for use only by derived classes; and *public*, for use by anyone. As an example, we may add the annotations **private**, **protected**, and **public** to the `Point` class above to obtain the class

$$
\begin{array}{llrcl}
\textbf{class}\ \texttt{Point}\ \{ & & & & \\
 & \textbf{private} & \texttt{x} & : & \mathit{int} \\
 & \textbf{protected} & \texttt{setX} & : & \mathit{int} \rightarrow \mathit{Point} \\
 & \textbf{public} & \texttt{getX} & : & \mathit{int} \\
 & & \texttt{move} & : & \mathit{int} \rightarrow \mathit{Point} \\
 & \textbf{constructor} & \texttt{newPoint} & : & \mathit{int} \rightarrow \mathit{Point} \\
 & \} & & &
\end{array}
$$

This modified class permits the `x` component to be used only within the `Point` class declaration, *i.e.*, within `setX`, `getX`, `move`, and `newPoint`. Similarly, `setX` is visible only to the implementation of the `Point` class and all classes derived from it, *e.g.*, the `ColorPoint` class defined above. We may similarly annotate the `ColorPoint` class to make its `c` component private.

Visibility levels help insure that separate blocks of code have no hidden dependencies and hence may be separately maintained. For example, if the designers of the `Point` class decide to change its underlying representation (say to a polar representation), they are free to modify private components without fear of breaking either client code or derived classes. Similarly, they may change protected components without worrying about breaking client code; however, in this case, they may have to fix derived classes. Throughout this paper, we will use the term "client code" for code restricted to public-level access and "derived classes" for code granted protected-level access.

There is a subtle interaction between visibility and constructors, which we will explain by example. Consider the `ColorPoint` constructor `newColorPoint`. To create an initialized `ColorPoint` object, `newColorPoint` must initialize the inherited portion of the object, in particular, the `x` component. However, the `ColorPoint` class has no access to its parent's private components. Hence there must be a way for base classes to provide initialization code for use within the constructors of derived classes. Typically, derived-class constructors invoke their parent's constructor to initialize these portions.

2.4 Defining Object Types

The fourth function of a class declaration is to define an object type. Each class associates a type with the objects it creates; this type will be the return type of the class constructor functions. In principle, there are at least two forms of object type that a language may use: *interface* types and *implementation* types. As the name suggests, interface types constrain only the interface of objects, specifying the set of messages each object must understand and the types of these messages. This form of object type has been extensively studied in the theoretical literature, *e.g.*, [AC96c, Bru93, FHM94, PT94] and the earlier papers appearing in [GM94], but it is not used in current class-based languages. In contrast, implementation types are a form of abstract type. An implementation type specifies both the interface and a portion of the implementation of an object. Although interface types offer increased flexibility, existing languages such as C++, Java, and Eiffel ([ES90, AG96, Mey92]) typically use implementation types by identifying the type of each object type with the class that created the object. To avoid confusion, we will distinguish between classes and the object types they define by writing class names in teletype font (`ClassName`) and object types in italics ($TypeName$). We discuss interface and implementation types in more detail in the remainder of this section.

Interface types An interface type specifies a list of operations, generally as method names and return types. If an object is declared to implement all of the operations with the appropriate types, it is considered an element of an interface type. For example, objects instantiated from the `Point` class have the following interface type, written using a notation defined in Section 4.2.

$$Point_{Inter} \stackrel{def}{=} \mathbf{obj}\, u.\langle\!\langle \mathtt{getX} : int \rightarrow u,\ \mathtt{move} : int \rightarrow u \rangle\!\rangle$$

An expression with this type is guaranteed to be an object that has at least `getX` and `move` methods. Furthermore, when either of these messages is sent with an integer parameter to such an object, the result is an object with the same type as the receiver, *i.e.*, $Point_{Inter}$. In more detail, to reflect the fact that the `Point` class is recursive (the return type of `getX` and `move` methods is *Point*), we use a form of recursive type for the `Point` class interface type. To that end, we have replaced the *Point* return type of the `getX` and `move` methods with a type variable u, which is bound by the keyword **obj** to the type $Point_{Inter}$. We omit the `x` and `setX` methods from the interface type because they are not public components and are therefore not visible to clients of the class. We also omit the constructor function `newPoint` because constructors are not part of the objects they create.

Similarly, we may write the interface type for the access-controlled version of the `ColorPoint` class as follows:

$$\begin{aligned} ColorPoint_{Inter} \stackrel{def}{=} \mathbf{obj}\, u.\langle\!\langle\ & \mathtt{getX} : int \to u,\ \mathtt{move} : int \to u, \\ & \mathtt{turnRed} : u,\ \mathtt{getC} : color \rangle\!\rangle \end{aligned}$$

Because interface types specify only the names and types of operations, objects with the same interface may have significantly different internal representations. In particular, objects from many different classes may all be given the same interface type. For example, objects instantiated from any class with public `getX` and `move` methods of the appropriate types will have type $Point_{Inter}$, not just objects instantiated from the `Point` class.

A significant advantage of interface types is the flexibility they provide. For example, using interface types we can define a single type of matrix object and then represent dense matrices with one form of object and sparse matrices with another. Both matrix representations may be given the same type and therefore be used interchangeably in any program. This kind of flexibility is particularly useful when we write library operations on matrices without assuming any particular implementation. Such library functions may be written using a standard interface type without concern for how matrices might be implemented in later (or earlier) development of a software system.

Implementation Types Intuitively, implementation types are in a one-to-one correspondence with classes. More specifically, an implementation type is a form of object type that guarantees both a specific public interface and some aspects of the implementation of objects, in a fashion similar to abstract datatypes. For example, $ColorPoint_{Imp}$, the implementation type associated with the `ColorPoint` class, imposes two requirements: if an object has $ColorPoint_{Imp}$, then it must (i) support the public interface $ColorPoint_{Inter}$ and (ii) have the private and protected components defined in the `ColorPoint` class. Thus $ColorPoint_{Imp}$ objects must have private `x` and `c` components, with `x` occurring before `c` in the representation if the language requires it. This implementation constraint that is *not* reflected in the public interface.

The extra information supplied by an implementation type is useful for several reasons, including optimizing component lookup, guaranteeing behavioral similarity, and allowing class-level protection mechanisms. We discuss these issues in more detail in the following paragraphs.

Optimizing Lookup. This point may be explained by example. If we know that all *Point* objects inherit a specific representation of `x` and `y` coordinates, then a program may be optimized to take advantage of this static guarantee. The usual implementations of C++, for example, use type information to statically calculate the offset of member fields relative to the starting address of the object. A similar calculation is used to find the offset of methods in the virtual function table at compile time [ES90, Section10.7c]. (Virtual function tables store the methods associated with all objects of a given class.) Such optimizations are not possible in an untyped language such as Smalltalk [GR83] and would not be possible in a typed language where objects of a single type could have arbitrarily dissimilar implementations. In particular, the Java interface mechanism allows us to specify that several classes that do not have similar implementations all satisfy a common interface. When all that is known about an object (say, passed as an actual parameter to a function) is an interface it supports, then basic facts about the implementation of the object are not known at compile time.

Behavioral Guarantees. A second, more methodological reason that programmers may be interested in implementation types is that they convey greater behavioral guarantees. For example, since the only non-trivial way to create a new object with a given implementation type is to call the constructor function for the associated class, we can be sure that an object with a given implementation type has been properly initialized and satisfies the expected invariants for its class.

Class-Level Protection. A more subtle reason to use types that restrict the implementations of objects has to do with the implementation of binary operations. In an object-oriented context, a binary operation on type A is realized as a method that requires another A object as a parameter. In a language where all objects of type A share some common representation, it is possible for an A method to safely access part of the private internal representation of another A object. A simple example arises with *Set* objects that have only a membership test and a union operation in their public interfaces. With interface types, some objects of type *Set* might be represented internally using bit vectors, while others might use linked lists. In this case, there is no type-safe way to implement union, since no single operation will access both a bit vector and a linked list correctly. With only interface types, it is necessary to extend the public interface of both kinds of sets with some sort of "`elementsAsList`" method to make this operation possible. In contrast, if the type of an object conveys implementation information, then a less flexible but type-safe implementation of set union is possible without polluting the public interfaces. In this case, all *Set* objects would have one representation and a union operation could be implemented by taking advantage of this uniformity. See [BCC+96] for a detailed discussion of binary methods.

Interface vs. Implementation Types In summary, interface types and implementation types have complementary strengths:

	Implementation types	Interface types
Flexibility	-	+
Efficiency	+	-
Behavioral Guarantees	+	-
Binary Methods	+	-

Current languages use implementation types, essentially identifying classes and their associated object types, but interface types have been widely studied in the theoretical literature [AC96c, Bru93, FHM94, PT94, GM94]. It is possible to associate both an interface and an implementation type with each classes, since, as we will see in the next section, each implementation type is typically a subtype of the interface type obtained by "forgetting" its implementation constraints. This technique combines the benefits of both forms of object type to be realized in a single language. To some extent, the Java class and interface mechanisms provide this combination of features, although since it is necessary to specifically declare membership in each Java interface, Java interfaces are actually a form of abstract data type in which the implementation constraint is vacuous, not a pure interface type of the form we discuss in this paper.

2.5 Defining a Subtyping Hierarchy

Like inheritance, *subtyping* is a code reuse mechanism. Instead of supporting the reuse of object definitions, however, subtyping supports the reuse of client code. 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. To see the importance of this principle, suppose we have written a large body of code to manipulate B objects. If A is a subtype of B, then it follows from the basic substitutivity principle of subtyping that the same code may also be able used to manipulate A objects (at least without type errors). The main significance of subtyping is that it permits uniform operations over various types of data that share some basic structure. Subtyping therefore makes it possible to have heterogeneous data structures containing objects that belong to different subtypes of some common base type. We write "$A <: B$" to indicate that A is a subtype of B.

It is important to keep in mind that subtyping is a relation on types, not classes. This distinction is easily understood in languages where objects are either untyped (in which case subtyping is an external notion that can be used to describe the differences between interfaces of objects) or where objects are created individually, without using classes. However, in class-based languages, as we have already noted, it is typical to associate a type with each class. This may either be an interface type or the form of abstract type we call an implementation type. The standard subtype relation on interface types is a form of structural subtyping, while the standard form of subtyping on implementation types requires some further correspondence between hidden implementation constraints. The only practical way to guarantee subtyping between implementation types, without violating principles of data abstraction by exposing hidden components of objects, is to make the subtyping hierarchy a subset of the inheritance hierarchy, *i.e.*, if

$$ColorPoint_{Imp} <: Point_{Imp}$$

then `ColorPoint` must inherit from `Point`. See [FM96, Fis96] for more details. Thus with implementation types, the class construct plays a substantial role in defining the subtyping hierarchy.

Since the implementation subtyping and inheritance hierarchies are closely related, providing one mechanism (class hierarchies) to declare both the subtyping and inheritance relations can be a significant conceptual simplification. This simplification does not mean, however, that the two hierarchies are necessarily the same. For example, C++ private inheritance allows two classes to be related in the inheritance hierarchy without inducing an associated subtype relationship. As mentioned briefly earlier, Java *interfaces* allow implementation types to have a common supertype, without sharing implementations.

With interface types, the subtype relation may be inferred by comparing the conditions imposed by each interface. For example,

$$ColorPoint_{Inter} <: Point_{Inter}$$

may be inferred simply by looking at the structure of these two types. While classes are fundamental to subtyping between implementation types, interface types allow subtyping and inheritance to be unrelated.

There is a third language design possibility, arising because implementation types are subtypes of interface types [Fis96]. In this scenario, each class has an associated implementation type that is a subtype of the interface type obtained by "forgetting" the extra implementation constraints. For example,

$$Point_{Imp} <: Point_{Inter}$$

The class hierarchy is responsible for determining subtyping relationships between implementation types, while subtype inference rules are used to infer relationships between interface types. Hence in this case, the class hierarchy defines the "bottom" portion of the subtyping hierarchy, while structural subtyping rules define the "top."

2.6 Evaluation Criteria

Sections 2.1 through 2.5 reveal that classes play a fundamental role in structuring code, providing both reuse (subtyping and inheritance) and code separation (access restrictions, initialization code) mechanisms. Since the goal of incorporating code reuse and code separation mechanisms into a programming language is to support large-scale programming, any class construct should be evaluated with large-scale programming in mind. Based on this consideration and the discussions in the preceding sections, we believe that a class mechanism should provide the features outlined in the following paragraphs. Where appropriate, we annotate these paragraphs with the sections that contain relevant discussion.

Coherent, Extensible Collection [Sections 2.1 and 2.2]. A class construct should provide an extensible collection of object components. Such components may include methods, data, local constants, specifications of communication protocols, *etc.* These parts should be *coherent*, in the sense that if a class is well-formed, then it should be guaranteed that all objects instantiated from the class will themselves be well-formed. For example, if one of the methods of a class requires an integer `f` method, then the class must contain an integer `f` method. The consistency of the components should be checked when the class is formed, not when objects are instantiated from the class, a timeliness which facilitates debugging. The *extensibility* of the collection means that derived classes may reuse object components from their parents, possibly adding additional coherent components and replacing others. This inheritance should be statically checked as well, insuring that derived classes are also well-formed.

Guaranteed Initialization [Sections 2.1 and 2.2]. Classes should provide code to initialize their collection of components when objects are instantiated. A class-based language should insure that this initialization code is executed both for objects instantiated directly from a given class and for those instantiated from its descendants. Running this initialization code is essential for establishing invariants for the components, particularly in light of the private level of visibility. Since derived classes cannot see private components and are not supposed to be aware of their parents' implementation details, they should not (and indeed cannot) be responsible for initializing and setting up invariants for inherited components.

Enforced Access Restrictions [Section 2.3]. A class construct should allow programmers to specify a level of visibility for each of the components defined in a class. These specifications should be enforced by the language. For example, if a programmer specifies that a given method is private, the type system should insure that only the implementation of the given class may access the method in any way. This restriction guarantees that the implementor of a class may change its private implementation without fear of introducing type errors into clients or descendants of the class. Access controls should clearly indicate the visibility of each of the parts for both the current class and all derived classes, so if a method is private in a given class, that method is guaranteed to be private in all derived classes as well.

Explicit Type Hierarchy [Sections 2.4 and 2.5]. Another important point of comparison for class constructs is whether or not they provide implementation types and explicit control over a portion of the subtyping hierarchy. Because implementation types are subtypes of interface types, classes that provide implementation types give programmers more flexibility. They may use implementation types where the extra information is useful and interface types where more generality is required.

Automatic Propagation of Base Class Changes. Because classes are intended to support large-scale programs and because such programs are inevitably modified, classes should support code maintenance. In particular, incremental changes to a base class should automatically be propagated to all derived classes. If any type errors are introduced in derived classes because of such changes, these errors should be reported immediately. For example, if a programmer adds a new method to a parent class, that method should then be automatically added to all derived classes. If the new method conflicts with a method defined in a descendent, the conflict should be reported immediately, not when objects are instantiated from the given derived class.

3 Class Construct

In this brief section, we give the syntax of the class construct we will translate into formal calculi in Sections 5 to 7.

$$
\begin{array}{llll}
\textbf{class}\ \langle \textit{class-name} \rangle & \{ & : & \langle \textit{superclass} \ \rangle \quad \} \\
\quad \textbf{constructor} & \langle \textit{name} \rangle & : & \langle \textit{type} \rangle \rightarrow \langle \textit{class-name} \rangle\,; \\
\quad \textbf{private} & \{\langle \textit{name} \rangle & : & \langle \textit{type} \rangle ; \}^{*} \\
\quad \textbf{public} & \{\langle \textit{name} \rangle & : & \langle \textit{type} \rangle ; \}^{*} \\
\textbf{end;} & & &
\end{array}
$$

This declaration of class *class-name* may specify a parent class *superclass*, gives a constructor function returning instances of the class with the type *class-name* (using the pun between class names and object types) and lists public and private methods. For the sake of simplicity, we leave the implementing code implicit, specifying an implementation in each object calculus, and omit the **protected** level of visibility. We also restrict our attention to the special case of exactly one constructor per class.

3.1 `Point` and `ColorPoint` Examples

To simplify the exposition in the following sections, we will use the familiar example of points and color points, slightly streamlined from the versions described in Section 2. Using the syntax above these classes may be written as follows:

$$
\begin{array}{lrl}
\textbf{class}\ \texttt{Point} & & \\
\quad \textbf{constructor} & \texttt{newPoint} & : \textit{int} \rightarrow \textit{Point}; \\
\quad \textbf{private} & \texttt{x} & : \textit{int}; \\
\quad \textbf{public} & \texttt{setX} & : \textit{int} \rightarrow \textit{Point}; \\
 & \texttt{getX} & : \textit{int}; \\
\textbf{end;} & &
\end{array}
$$

class `ColorPoint` : `Point`

$$
\begin{array}{llrl}
 & \textbf{constructor} & \texttt{newCPoint} : & \mathit{color} \to \mathit{int} \to \mathit{ColorPoint}; \\
 & \textbf{private} & \texttt{c} : & \mathit{color}; \\
 & \textbf{public} & \texttt{setC} : & \mathit{color} \to \mathit{ColorPoint}; \\
 & & \texttt{getC} : & \mathit{color};
\end{array}
$$

end;

Intuitively, the `set` methods are used to assign to private location or color data while the `get` methods read these private values. The constructor functions `newPoint` and `newCPoint` return new `Point` and `ColorPoint` objects, respectively, initialized to store the values passed to them as parameters.

4 Preliminaries: Objects, Object Types and Existential Types

Before proceeding to the first translation, we introduce some background material, including basic object calculi terminology and notation, the object types we will use, and a form of existential type that will be useful in hiding private components.

4.1 Objects, Fields and Methods

Many of the basic object operations that we will need may be presented using Cardelli's Obliq [Car95], a simple dynamically-typed language for distributed programming that has been used for a variety of purposes, including graphics and web "oblets." Obliq has a pure object system which is not based on classes. Although we refer to Obliq as an example of a "real" programming language based on pure objects without classes, we will use a simplified lambda-calculus-style notation that has fewer syntactic distinctions than Obliq, but retains the same essential expressiveness.

An Obliq object may be written in the form

$$\{\mathtt{m_1} = e_1, \ldots, \mathtt{m_k} = e_k\}$$

where each e_i defines a *component* of the object. For clarity, Obliq distinguishes between two forms of components, *data fields* and *methods.* The distinction is not based on the types of components, since functions are considered a particular kind of data. Instead, the distinction is based on how the component is accessed. Specifically, simple selection, as in records, is used for fields, while method invocation is used for methods. The distinction may be illustrated using the object

$$\mathtt{a} \stackrel{def}{=} \{\mathtt{x} = 6,\ \mathtt{setX} = \lambda\,(\mathtt{self}, \mathtt{n})\,\mathtt{self.x} := \mathtt{n}\}$$

We have $\mathtt{a.x} = 6$ simply by extracting the value of `x`, and

$$\mathtt{a.setX(8)} \;=\; \{\mathtt{x} = 8,\ \mathtt{setX} = \lambda\,(\mathtt{self}, \mathtt{n})\,\mathtt{self.x} := \mathtt{n}\}$$

by a function call that involves passing the receiver object `a` to the `setX` method as the first actual parameter. The important point is that method invocation is considered an atomic operation; if a component is a method, it is not possible to extract the method and apply it (as a function) to any object other than the object to which it belongs. An object with only fields may be called a *record.*

Several formal calculi that provide Obliq operations, and more, have been developed [AC96c, AC95, AC96a, Mit90, FHM94, FM95a]. As a foundational point, it is possible to eliminate the distinction between fields and methods since any field may be represented as a method which does not use the `self` argument. For example, we could replace `x = 6` by `x =` λ`(self) 6`. The only differences are that method invocation is less efficient than field selection in practice and that objects of a given class typically have different data but the same methods. This second distinction allows objects to share method code and method lookup tables.

For syntactic simplicity, we will assume every object component is a method and write objects in the form

$$\langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_k} = e_k \rangle$$

where each expression $e_1, \ldots, e_k$ is assumed to be a function that expects its host object as its first actual parameter when invoked. Using the symbol $\Leftarrow$ for method invocation, we have

$$\langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_k} = e_k \rangle \Leftarrow \mathtt{m_i} \quad = \quad e_i \langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_k} = e_k \rangle$$

A useful operation on objects is to *override* a method, creating a new object with a different value at that method. Using the symbol $\leftarrow$ for override, we have

$$\begin{aligned}\langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_i} = e_i, \ldots, \mathtt{m_k} = e_k \rangle \leftarrow \mathtt{m_i} = e_i' \quad &= \\ &\langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_i} = e_i', \ldots, \mathtt{m_k} = e_k \rangle\end{aligned}$$

These object operations are shared by the calculi defined in [AC96c, AC95, AC96a, Mit90, FHM94, FM95a] and subsequent variants.

4.2 Object Types

We will use the type expression

$$\mathbf{obj}\, u\,\mathbf{.}\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_k} : \tau_k \rangle\!\rangle$$

to type objects that have methods $\mathtt{m_1}, \ldots, \mathtt{m_k}$ with associated types $\tau_1, \ldots, \tau_k$. In this expression, the bound type variable u may appear in the type of a method, in which case it refers to the type of the host object. For example, if method `setX` is invoked on an object of type $\mathbf{obj}\, u\,\mathbf{.}\langle\!\langle \mathtt{x} : int, \mathtt{setX} : int \rightarrow u \rangle\!\rangle$, the result is a function which, given an integer, returns an object of type $\mathbf{obj}\, u\,\mathbf{.}\langle\!\langle \mathtt{x} : int, \mathtt{setX} : int \rightarrow u \rangle\!\rangle$. When u does not appear in the types $\tau_1, \ldots, \tau_k$, we will use the shorthand

$$\mathbf{obj.}\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_k} : \tau_k \rangle\!\rangle$$

As described elsewhere [FHM94, FM95a, AC96c, AC95] **obj** types are essentially a specialized form of recursive type, with "folding" combined with object formation and "unfolding" combined with method invocation.

4.3 Existential quantification and data abstraction

Since we will use existential types in several places, we review a general form of datatype declaration based on existential types. For further information on data abstraction, the reader is referred to [LSAS77, MP88, Mor73, Rey83, Set89]; the connection between existential types and data abstraction is elaborated in [MP88, Mit93].

The declaration form

$$\texttt{Abstype } t \texttt{ with } x_1 : \sigma_1, \ldots, x_k : \sigma_k \texttt{ is } \langle \tau; M_1, \ldots, M_k \rangle \texttt{ in } N$$

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

$$\begin{array}{lll} \texttt{Abstype} & \multicolumn{2}{l}{\mathit{stream}\ \texttt{with}} \\ & & s : \mathit{stream}, \\ & & \mathit{first} : \mathit{stream} \to \mathit{nat}, \\ & & \mathit{rest} : \mathit{stream} \to \mathit{stream} \\ & \texttt{is} & \\ & & \langle \tau, M_1, M_2, M_3 \rangle \\ & \texttt{in} & \\ & & N \end{array}$$

declares an abstract datatype *stream* with distinguished element $s : \mathit{stream}$ 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 $M_2 : \tau \to \mathit{nat}$ and $M_3 : \tau \to \tau$. Using cartesian products, we may put any abstract datatype declaration in the form $\texttt{Abstype } t \texttt{ with } x{:}\sigma \texttt{ is } \langle \tau, M \rangle \texttt{ in } N$. 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 $\mathit{stream} \times (\mathit{stream} \to \mathit{nat}) \times (\mathit{stream} \to \mathit{stream})$. 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 datatype declarations and datatype 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, $t \to \mathit{nat}$, 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 datatype implementations and `Abstype` declarations to a type system using *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.

We will write an implementation of an abstract type 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 $\exists t.\sigma$. For notational simplicity, we write $\langle t = \tau, M\rangle$ instead of $\langle t = \tau, M : \sigma\rangle$ when the type σ is clear from context.

The `Abstype` declaration form allows us to bind names to the type and value components of a datatype implementation. More specifically, if expression e_1 has type $\exists t.\tau$ and, under the assumption that $x{:}\tau$ the expression e_2 has type σ, then

$$\texttt{Abstype}\ t\ \texttt{with}\ x : \tau\ \texttt{is}\ e_1\ \texttt{in}\ e_2\ :\ \sigma$$

Informally, this rule binds type variable t and ordinary variable x to the type and value part of the implementation e_1, respectively, with scope e_2. A technical-sounding condition is that in $\texttt{Abstype}\ t\ \texttt{with}\ x : \tau\ \texttt{is}\ e_1\ \texttt{in}\ e_2\ :\ \sigma$, the type variable t cannot occur free in the type σ of the entire expression. This is actually a straightforward scoping condition: the bound variable t should not be allowed to escape from its scope.

Existential types and objects Existential types are useful in the context of modeling objects because they can hide private components [PT94, PT93]. To develop a feel for this idea, we briefly explain how point objects may be modeled as elements of existential type. The type of an object with private `x` and public `getX` and `setX` components could be written as follows

$$
\begin{array}{l}
Point \stackrel{def}{=} \exists\, Rep.\ record \\
\qquad\qquad\qquad\qquad \texttt{state} \ :\ Rep; \\
\qquad\qquad\qquad\qquad \texttt{methods} :\ record \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{getX} : Rep \to int; \\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{setX} : Rep \to int \to unit; \\
\qquad\qquad\qquad\qquad\qquad\quad end; \\
\qquad\qquad\quad end;
\end{array}
$$

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. The types of an object's methods are written in terms of the type of the hidden state.

Now that we have defined the operations and types that we need, we may proceed with the first class encoding, translating the class construct we saw in Section 3 into a simple object calculus.

5 Record-of-Premethods Model

In Cook's important early work on the foundations of object-oriented languages [Coo87, Coo89], each class is represented by a function, called an *object generator.* The fixed point of a generator is a record that represents an object, with recursion used to resolve references to *self.* The reason for object generators is that inheritance cannot work directly on objects that encapsulate (or hide) the dependence of one method on another. However, inheritance may be formulated as an ordinary operation on method bodies if we explicitly treat methods as functions of *self* [Mit90].

In the context of object calculi, it seems natural to define inheritance using premethods, functions that are written with the intent of becoming object methods, but which are not yet installed in any object. Like Cook's generators, premethods are functions that explicitly depend on the "object itself," typically assumed to be the first parameter to the function. Following this idea, Abadi and Cardelli have proposed encoding classes in a pure object system using records of premethods [AC96c]. In this model, a class is an object that contains a record of premethods and a constructor function used to package these premethods into objects. We illustrate this proposal by example in the next subsection. A similar approach is used in [RR96].

5.1 Example Classes

Figure 1 shows the six declarations that result from translating the classes given in Section 3 into a simple object calculus along the lines outlined above. The first three declarations give (1) the object type associated with the `Point` class:

$$\mathbf{obj.}\langle\!\langle \mathtt{x} : \mathit{int}, \mathtt{getX} : \mathit{int}, \mathtt{setX} : \mathit{int} \rightarrow \mathit{unit} \rangle\!\rangle,$$

(2) the record of `Point` premethods `PointImpl,` and (3) the `Point` constructor `newPoint.` For the sake of simplicity, we assume we are working with an imperative semantics and `setX` works by changing the value of `x`. This assumption allows us to make *unit* the return type of `setX`, side-stepping the problem of *method specialization*. Method specialization occurs when the types of methods are "improved" during inheritance to reflect their new, more specialized host object. For example, if the type of `setX` were *Point* in `Point` objects and *ColorPoint* in `ColorPoint` objects, we would say the type of `setX` was *specialized* during inheritance. Although this general approach can support method specialization, significantly more complex types are required to do so [AC96b].

The next three declarations give the corresponding representation for the `ColorPoint` class. Note that the implementation `CPointImpl` of `ColorPoint` is defined from the implementation `PointImpl` of `Point`. This reuse is type correct because $\mathit{ColorPoint} <: \mathit{Point}$, and therefore, by the contravariance of function types, we have $\mathit{Point} \rightarrow \tau <: \mathit{ColorPoint} \rightarrow \tau$ for each type τ. However, this dependency appears to be the extent to which inheritance can be realized. It is not possible to reuse the `Point` constructor because objects are not extensible in this model. Thus, we cannot use a `Point` object to create a `ColorPoint` object (except trivially).

$$
\begin{array}{lll}
\texttt{type}\ \mathit{Point} & = & \mathbf{obj.}\langle\!\langle \texttt{x} : \mathit{int},\ \texttt{getX} : \mathit{int},\ \texttt{setX} : \mathit{int} \rightarrow \mathit{unit} \rangle\!\rangle; \\
\texttt{val PointImpl} & : & \mathit{record} \\
 & & \quad\quad \texttt{x} : \mathit{Point} \rightarrow \mathit{int}; \\
 & & \quad \texttt{getX} : \mathit{Point} \rightarrow \mathit{int}; \\
 & & \quad \texttt{setX} : \mathit{Point} \rightarrow \mathit{int} \rightarrow \mathit{unit}; \\
 & & \mathit{end}; \\
 & = & \texttt{record} \\
 & & \quad\quad \texttt{x} = \lambda(\texttt{self})\ \texttt{0}; \\
 & & \quad \texttt{getX} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{x}; \\
 & & \quad \texttt{setX} = \lambda(\texttt{self})\, \lambda(\texttt{newX})\ \texttt{self.x} := \texttt{newX}; \\
 & & \texttt{end}; \\
\texttt{fun newPoint} & : & \mathit{int} \rightarrow \mathit{Point} \\
 & = & \lambda\, \texttt{xi} : \mathit{int}. \langle \quad \texttt{x} = \texttt{PointImpl.x}, \\
 & & \quad\quad\quad \texttt{getX} = \texttt{PointImpl.getX}, \\
 & & \quad\quad\quad \texttt{setX} = \texttt{PointImpl.setX}\ \rangle \Leftarrow \texttt{setX xi}
\end{array}
$$

$$
\begin{array}{lll}
\texttt{type}\ \mathit{ColorPoint} & = & \mathbf{obj.}\langle\!\langle\ \texttt{x} : \mathit{int},\ \texttt{getX} : \mathit{int},\ \texttt{setX} : \mathit{int} \rightarrow \mathit{unit}, \\
 & & \quad\quad \texttt{c} : \mathit{color},\ \texttt{getC} : \mathit{color},\ \texttt{setC} : \mathit{color} \rightarrow \mathit{unit} \rangle\!\rangle \\
\texttt{val CPointImpl} & : & \mathit{record} \\
 & & \quad\quad \texttt{x} : \mathit{ColorPoint} \rightarrow \mathit{int}; \\
 & & \quad \texttt{getX} : \mathit{ColorPoint} \rightarrow \mathit{int}; \\
 & & \quad \texttt{setX} : \mathit{ColorPoint} \rightarrow \mathit{int} \rightarrow \mathit{unit}; \\
 & & \quad\quad \texttt{c} : \mathit{ColorPoint} \rightarrow \mathit{color}; \\
 & & \quad \texttt{getC} : \mathit{ColorPoint} \rightarrow \mathit{color}; \\
 & & \quad \texttt{setC} : \mathit{ColorPoint} \rightarrow \mathit{color} \rightarrow \mathit{unit}; \\
 & & \mathit{end}; \\
 & = & \texttt{record} \\
 & & \quad\quad \texttt{x} = \texttt{PointImpl.x}; \\
 & & \quad \texttt{getX} = \texttt{PointImpl.getX}; \\
 & & \quad \texttt{setX} = \texttt{PointImpl.setX}; \\
 & & \quad\quad \texttt{c} = \lambda(\texttt{self})\ \texttt{red}; \\
 & & \quad \texttt{getC} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{c}; \\
 & & \quad \texttt{setC} = \lambda(\texttt{self})\, \lambda(\texttt{newC})\, \texttt{self.c} := \texttt{newC}; \\
 & & \texttt{end} \\
\texttt{fun newColorPoint} & : & \mathit{int} \rightarrow \mathit{color} \rightarrow \mathit{ColorPoint} \\
 & = & \lambda\, \texttt{xi} : \mathit{int}.\, \lambda\, \texttt{ci} : \mathit{color}. \\
 & & (\langle \texttt{getX} = \texttt{CImpl.getX},\ \ldots\ \rangle \Leftarrow \texttt{setX xi}) \Leftarrow \texttt{setC ci}
\end{array}
$$

Fig. 1. Type and constructor declarations for `Point` and `ColorPoint` classes.

5.2 Adding Protection

Notice that in the basic record-of-premethods encoding, the `x` and `c` components of an object are not private; they are just as accessible as the public methods. There are three ways of incorporating private fields into this approach, each with advantages and disadvantages. The first approach uses closures to create private fields whenever a constructor is called. The second uses subtyping to restrict the return type of constructor functions to reflect only public methods. The third uses a variation of the existential-type object model.

5.3 Protection Via Closures

$$
\begin{array}{lll}
\texttt{type}\ \mathit{Point} & = & \mathbf{obj.}\langle\!\langle \texttt{getX} : \mathit{int}, \texttt{setX} : \mathit{int} \rightarrow \mathit{unit} \rangle\!\rangle; \\
\texttt{val PointImpl} & : & \mathit{record} \\
 & & \quad \texttt{getX} : \mathit{int} \rightarrow \mathit{Point} \rightarrow \mathit{int}; \\
 & & \quad \texttt{setX} : \mathit{int} \rightarrow \mathit{Point} \rightarrow \mathit{int} \rightarrow \mathit{unit}; \\
 & & \mathit{end}; \\
 & = & \texttt{record} \\
 & & \quad \texttt{getX} = \lambda(\texttt{x})\, \lambda(\texttt{self})\ \texttt{x}; \\
 & & \quad \texttt{setX} = \lambda(\texttt{x})\, \lambda(\texttt{self})\, \lambda(\texttt{newX})\, \texttt{x} := \texttt{newX}; \\
 & & \texttt{end}; \\
\texttt{fun newPoint} & : & \mathit{int} \rightarrow \mathit{Point} \\
 & = & \lambda\, \texttt{xi} : \mathit{int}.\texttt{let x = xi in} \\
 & & \qquad \langle\, \texttt{getX} = \texttt{PointImpl.getX x}, \\
 & & \qquad\ \ \texttt{setX} = \texttt{PointImpl.setX x}\ \rangle
\end{array}
$$

Fig. 2. Declarations for the `Point` class with modifications to support closure-style private fields.

Figure 2 shows a version of the `Point` class, modified to incorporate closure-style private fields. The necessary changes include:

- removing the `x` field from type *Point* to indicate that `x` is not publicly available,
- removing the `x` field from `PointImpl`, since the `x` field will be allocated within the constructor function on a per object basis,
- rewriting the remaining premethods (and their types) to take an extra parameter, a placeholder for the `x` field that will be supplied whenever a `Point` object is created, and
- modifying the constructor function to allocate the private `x` field and pass it to each of the premethods during their installation in the new object.

This mechanism succeeds in hiding the `x` field from general clients; only the `setX` and `getX` methods of a particular object may access its `x` field. The approach does not succeed, however, at hiding private fields from derived classes. Notice that because inheritance manipulates records of premethods instead of constructors, the private `x` field is not inherited by the `ColorPoint` class. Instead, the constructor for the derived class `ColorPoint` has to allocate its own `x` field for the inherited methods, illustrating the fact that the supposedly private `x` component has been exposed to derived classes. This exposure of private `x` is also reflected in the new types for the `setX` and `getX` premethods. Another problem with this approach is its lack of support for private *methods*. Because methods need to reside within objects to allow access to the *self* parameter, they cannot be stored in closures.

5.4 Protection Via Restricting Constructor Return Type

In this version, which appears in Figure 3, there are two object types associated with the `Point` class. Type *Point* lists the public components, while $Point_{Priv}$

$$
\begin{array}{lll}
\texttt{type}\ Point & = \mathbf{obj.}\langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle; \\
\texttt{type}\ Point_{Priv} & = \mathbf{obj.}\langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle; \\
\texttt{val PointImpl} & : record \\
 & \qquad\quad \texttt{x} : Point_{Priv} \rightarrow int; \\
 & \qquad \texttt{getX} : Point_{Priv} \rightarrow int; \\
 & \qquad \texttt{setX} : Point_{Priv} \rightarrow int \rightarrow unit; \\
 & \quad end; \\
 & = \texttt{record} \\
 & \qquad\quad \texttt{x} = \lambda(\texttt{self})\ \texttt{0}; \\
 & \qquad \texttt{getX} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{x}; \\
 & \qquad \texttt{setX} = \lambda(\texttt{self})\,\lambda(\texttt{newX})\ \texttt{self.x} := \texttt{newX}; \\
 & \quad \texttt{end}; \\
\texttt{fun newPoint} & : int \rightarrow Point \\
 & = \lambda\ \texttt{xi} : int.\langle \quad \texttt{x} = \texttt{PointImpl.x}, \\
 & \qquad\qquad \texttt{getX} = \texttt{PointImpl.getX}, \\
 & \qquad\qquad \texttt{setX} = \texttt{PointImpl.setX}\ \rangle \Leftarrow \texttt{setX xi}
\end{array}
$$

Fig. 3. Declarations for the `Point` class with modifications to support private components via restricting the return type of `Point`'s constructor function.

lists the public and the private components. The premethods assume the *self* object has the private as well as the public components, but the constructor function exposes only the public methods by making its return type *Point* instead of $Point_{Priv}$. This technique is type correct because $Point_{Priv}$ is a subtype of *Point* ($Point_{Priv}$ <: *Point*).

Although this approach succeeds in hiding the `x` component from clients who invoke the `newPoint` constructor function, it fails to hide the `x` premethod. Consequently, if the `Point` class designers wish to change their private representation, they must worry about breaking client code that has come to rely on a supposedly private premethod. A second encapsulation failure stems from the fact that the `ColorPoint` class must explicitly extract the private components of `PointImpl` to inherit them. Hence in this approach, derived classes must again be aware of the private components of their parents. On the positive side, this technique does support private methods in addition to private fields.

5.5 A Refinement

In response to our criticism, Martin Abadi has suggested a refinement to the record-of-premethods approach that involves using a form of existential type to hide the types of private components [Aba96]. Figure 4 shows the necessary declarations for the `Point` class. In this refinement, the `x` premethod is still

$$
\begin{array}{lll}
\texttt{type}\ Point & = & \mathbf{obj.}\langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle; \\
\texttt{type}\ Point_{Priv}(R) & = & \mathbf{obj.}\langle\!\langle \texttt{x} : R,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle; \\
\texttt{val PointImpl} & : & \exists R.record \\
 & & \quad \texttt{x} : Point_{Priv}(R) \rightarrow int; \\
 & & \quad \texttt{getX} : Point_{Priv}(R) \rightarrow int; \\
 & & \quad \texttt{setX} : Point_{Priv}(R) \rightarrow int \rightarrow unit; \\
 & & \quad \texttt{newP} : int \rightarrow Point; \\
 & & \quad end; \\
 & = & \langle\ \texttt{R} = int, \\
 & & \quad \texttt{record} \\
 & & \quad\quad \texttt{x} = \lambda(\texttt{self})\ \texttt{0}; \\
 & & \quad\quad \texttt{getX} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{x}; \\
 & & \quad\quad \texttt{setX} = \lambda(\texttt{self})\,\lambda(\texttt{newX})\,\texttt{self.x}{:=}\,\texttt{newX}; \\
 & & \quad\quad \texttt{newP} = \lambda\,\texttt{xi} : int.\langle\ \texttt{x} = \texttt{PointImpl.x}, \\
 & & \quad\quad\quad\quad \texttt{getX} = \texttt{PointImpl.getX}, \\
 & & \quad\quad\quad\quad \texttt{setX} = \texttt{PointImpl.setX}\ \rangle \Leftarrow \texttt{setX xi} \\
 & & \quad \texttt{end}\rangle; \\
\texttt{fun newPoint} & : & int \rightarrow Point \\
 & = & \lambda\,\texttt{xi} : int.\,\texttt{PointImpl.newP xi}
\end{array}
$$

Fig. 4. Declarations for the `Point` class with modifications to support private components via restricting the return type of `Point`'s constructor function and hiding the types of private methods via data abstraction.

exposed in `PointImpl`, but its type is hidden by an existential type. Since only the other premethods of the `Point` class are within the scope of this existential, only they can see `x`'s true type; hence only they can use the `x` premethod in any non-trivial way. The main changes from Figure 3 include

– parameterizing $Point_{Priv}$ by the representation type for `x`,
– giving `PointImpl` existential type, to hide the private representation for `x`,
– changing the argument type for each of the premethods to be the private type $Point_{Priv}(R)$ instantiated to the hidden representation type,
– implementing `PointImpl` as an element of existential type and binding the representation type variable R to the selected representation type, *int*, and
– moving the code for the constructor function into `PointImpl` so that it will be within the scope of the hidden representation type.

This approach succeeds in hiding the private `x` component from general use. Furthermore, since the `x` premethod has an unknown type, clients cannot usefully access it, despite its public availability, because its hidden type means it can only be used in trivial ways. Derived classes still must explicitly name the private component to inherit it, which is somewhat problematic. For private fields, there is at least a partial solution to this naming problem [Aba96]. In particular, a convention could be adopted in which each class has a special component named "`private_fields`" that stores a record of all the private fields. Instead of having to name each of the fields individually, the derived class can just name `private_fields`. At the price of this one name being globally known, we can insure that the actual private names do not leak out.

Since this approach is the most successful at modeling the private level of visibility, we will adopt this version as the "record-of-premethods" class interpretation for the purposes of the analysis in the next section.

5.6 Evaluation

The primary advantage of the record-of-premethods encoding of classes is that it does not require a complicated form of object. All that is needed is a way of forming an object from a list of component definitions. However, looking back at the list of desiderata in Section 2.6, it is clear that this approach has some serious drawbacks. We discuss each of the criteria in order.

Coherent, Extensible Collection. The combination of a record of premethods and a constructor function may be thought of as a coherent, extensible collection. Because premethods are simply fields in a record, nothing requires that they be coherent until a constructor function is supplied. Since the constructor function installs the premethods into an object, however, the fact that a given constructor is typable implies that the premethods it uses are coherent. Notice, however, that nothing requires a given constructor to mention all of the premethods in a given premethod record.

Guaranteed Initialization. In this model, class designers *can* guarantee that objects instantiated directly from their class are initialized if they insure that each of the classes that they inherit from behave properly; however, they *cannot* guarantee that classes deriving from theirs will inherit properly initialized components. The problem arises from the fact that inheritance manipulates records of premethods instead of constructor functions. Because inherited private components have unknown type, derived classes cannot initialize them directly. Constructor functions return non-extensible objects, so they cannot be used for this initialization either, except trivially. Instead, parent classes must supply separate initialization functions for this purpose. Nothing requires derived classes to invoke the supplied initialization code, however. Hence the designers of parent classes cannot assume that derived classes will always properly initialize the components they inherit. Furthermore, if a class (or one of its parents) does not initialize its inherited components properly, then the objects instantiated from that class may not be properly initialized either.

Enforced Access Restrictions. As we saw above, the final version of the record-of-premethod approach largely succeeds in providing enforced access restrictions, at the cost of complicating the object primitives with existential types. The only (small) leak is the name of private methods. A subtle point is that premethods are available to derived classes, not just methods. In other words, a derived class has access to a record of functions, and these functions may be used in any way. They need not be used as methods of an object, but could be used to define functions or could be tested, for example to determine whether a base class method depends on another methods of the object.

Explicit Type Hierarchy. Another issue that we believe is equally significant is the lack of explicit control over the subtyping hierarchy. In many existing class-based languages, it is possible to restrict the subtypes of an object type to classes that inherit all or part of the class implementation. As we discussed in Section 2.4, this restriction may be useful for optimizing operations on objects, allowing object access in binary methods and guaranteeing semantic consistency beyond type considerations (some discussion of these points appears in [KLM94]). A special case of this capability is the ability to define *final* classes, as recognized in work on Rapide [KLM94] and incorporated (presumably independently) as a language feature in Java. This ability is lacking in the record-of-premethods approach since any object whose type is a structural subtype can be used as an object of that type.

Automatic Propagation of Base Class Changes. Because derived class constructors must explicitly name the components that they wish to inherit, the record-of-premethods approach does not automatically propagate base class changes. In particular, if a derived class `D` is defined from a base class `B` in Java or related languages, then adding a method to `B` will result in an additional method of `D`, and similarly for every other class derived from `B` (and there may be many). With the approach shown here, derived class constructors must be explicitly rewritten each time their base classes change. Since object-oriented programs are typically quite large and maintenance may be distributed across many people, the person who maintains a base class may fail to inform those maintaining its derived classes of its change, causing unpredictable errors. There is no mechanism in the language to detect such errors automatically. Another way of stating this point is to say that in this approach, the translation from classes to object operations is not local – the definition of the constructor of a class depends on all of the definitions of classes from which components are inherited.

While reasonable people may disagree about the potential for records of premethods in large-scale object-oriented system design, it seems clear to us that this approach involves a significant loss of language support, in favor of very simple object primitives. While we readily admit that simplicity is a virtue, it also seems worthwhile to carefully examine the features that are lost.

6 Classes as Extensible Objects + Abstraction

Extensible objects provide a rich alternative to generators or premethods. They obviate the need for premethods, since collections of methods that are already installed in objects may be extended. Because of this fact, we may impose static constraints on the ways in which one method may be combined with others. For example, if an object contains two mutually recursive methods, then we cannot replace one with another of a different type. In contrast, in the record-of-premethods approach, it is possible to form a record of premethods without a "covering" constructor that checks to be sure that all of the premethods are coherent. A second advantage of extensible objects is that class constructors and initialization code can be inherited, *i.e.*, reused in derived classes. For example, to create `ColorPoint` objects, we may invoke the `Point` class constructor and add color methods to the resulting extensible object. This process guarantees that the `Point` class has the opportunity to properly initialize the inherited components. Another advantage arises with private (or protected) methods. In the extensible-object formulation, methods always remain within an object, even when it is extended. These hidden methods exist in all future extensions, but they can only be accessed by methods that were defined before the method became hidden. Furthermore, these private methods need not be explicitly manipulated by derived class constructors to insure that they are treated properly.

In this section and the next, we show how to obtain classes by combining extensible objects and a form of data abstraction. As a first step, we describe a suitable object calculus and briefly discuss the type system necessary to support it. We then introduce a refined form of existential type to provide the necessary data abstraction. Finally, we put these two mechanisms together to get the desired class construct.

6.1 Extensible Objects

Extensible objects support the operation of *object extension* in addition to method invocation and method override. Object extension, which we denote by the symbol $\longleftarrow\!\!+$, adds new components to objects. Although it seems appealing to work in a calculus with a single form of object that is both overridable and extensible, there is unfortunately a fundamental conflict between these operations and subtyping, another crucial aspect of object types. In particular, we lose *width* *subtyping* with extension and *depth subtyping* with override, as illustrated in the following table.

Object Inheritance Operations	Sound Subtyping
none	width and depth
override	width only
extension	depth only
override and extension	none

Width subtyping implies that longer object types (ones with more methods) are subtypes of narrower ones. Depth subtyping is the orthogonal notion; it specifies that object types with specialized components are subtypes of object types with more general components. For example, assuming that $ColorPoint <: Point$, then $\mathbf{obj}.\langle\!\langle \mathtt{x} : ColorPoint \rangle\!\rangle$ is a depth subtype of $\mathbf{obj}.\langle\!\langle \mathtt{x} : Point \rangle\!\rangle$. The conflict between subtyping and object-based inheritance is described in [FM95b].

Because of this conflict, we need two forms of object type to construct classes properly: first, an extensible form for use within class declarations to support inheritance and second, a non-extensible form that provides rich subtyping for use as the objects instantiated from classes. Ideally, there would be an easy way to obtain the non-extensible form from the extensible one. This conversion process should occur whenever programmers ask for instances of the class. We shall refer to extensible objects as *prototypes* and non-extensible ones as proper objects.

Type checking prototypes is significantly more challenging than type checking proper objects. Intuitively, the difficulties arise from two sources. The first is that we need to insure that a given method name is *not* used in a prototype before we add a new method with that name. This guarantee insures that the new method body cannot violate any typing assumptions the old prototype may have made for methods of that name. To provide this kind of information, we have equipped our type system with a kinding system that tracks method *absences*. In principle, a type system that tracks potential method *presences* may also suffice.

The second source of difficulty arises from the need to insure that each method body works correctly for all objects, extensible and otherwise, that it may eventually live in. Since we may extend prototypes, we must show that a candidate method has the necessary type for all possible extensions of its original host. To express this form of polymorphism, we need to be able to say formally "all future extensions" of a prototype. To that end, we introduce a version of Wand's *rows* [Wan87] into our type system. Intuitively, a *row* is a list of methods and associated types; it has the form $\langle\!\langle \mathtt{m_1}{:}\tau_1, \ldots, \mathtt{m_k}{:}\tau_k \rangle\!\rangle$. *Row variables* range over row expressions. Such variables, when implicitly universally quantified, stand for any list of method-type pairs. When we bound such a row variable with a "supertype" listing the methods of a prototype, we obtain (a representation of) the collection of all possible future extensions of that prototype.

Since we will use both object extension and method override for inheritance, we lump the two inheritance operations, method override and object extension, together in this paper. We will use types of the form $\mathbf{pro}.\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_k} : \tau_k \rangle\!\rangle$

to type our extensible, overridable objects. It turns out that we may obtain non-extensible objects from these extensible ones via subsumption [FM95a]:

$$\mathbf{pro.}\langle\!\langle \mathtt{m}_1 : \tau_1, \ldots, \mathtt{m}_k : \tau_k \rangle\!\rangle <: \mathbf{obj.}\langle\!\langle \mathtt{m}_1 : \tau_1, \ldots, \mathtt{m}_k : \tau_k \rangle\!\rangle$$

6.2 Data Abstraction

To provide the encapsulation needed for classes, we introduce a fairly standard form of bounded existential quantification [CW85, MP88]. Bounded existential types are similar to the existential types we described in Section 4.3, except that they provide a supertype bound for the hidden type, revealing partial information about it. In more detail, a declaration of the form

$$\mathtt{Abstype}\ p <:_w P :: \kappa\ \mathtt{with}\ \mathit{Ops}\ \mathtt{is}\ \mathit{Impl}\ \mathtt{in}\ \mathit{Client}$$

introduces a new row name, p, with upper bound P and kind κ. When we use this construct to model classes, p will play the role of a class name. Intuitively, the upper bound P lists the methods from p that clients will be able to use, while the kind information indicates how derived classes may extend p. The functions listed in *Ops* are the only operations that can access the private representation of p. Typically, class constructors and friend functions appear in this list. *Impl* gives the actual implementation of the abstraction, which *Client* uses as allowed by P, κ, and *Ops*.

Typically, a class implementation specifies an object layout and a set of method bodies (code for the methods of its objects). In our approach, object layouts are given by row expressions and method bodies as parts of constructor functions. Formally, the implementation of each class (*Impl*, above) is given by an expression of the form

$$\{\!|p <:_w P :: \kappa = P_{priv},\ \mathit{ConsImp}|\!\}$$

As above, the row variable p may be thought of as the name of the class implemented by this expression. Row P lists the class's publicly supported methods and their types. Kind κ describes how the class may be extended via inheritance. Thus P and κ essentially form the interface of the class implementation construct. The remaining components, P_{priv} and *ConsImp*, give the actual implementation of the class. P_{priv} describes the "layout" of the class's instances by listing the names of all of the methods defined within the class and specifying their types. The constructor function *ConsImp* provides the method bodies and initializes instantiated objects as necessary.

Following general principles of data abstraction, method bodies may rely on aspects of their host's representation that are hidden from other parts of the program. This fact is reflected in the above construct in that the constructor function *ConsImp* is within the scope of the private interface P_{priv}, and hence it may use private methods omitted from P. Our framework allows any number of constructor functions or other "non-virtual" operations to be provided in the *ConsImp* slot. However, for simplicity, we discuss only the special case of one constructor per class.

6.3 Example Classes

$$
\begin{array}{l}
\texttt{Abstype}\ p <:_w P_{pub} :: \{\texttt{c}, \texttt{getC}, \texttt{setC}\} \\
\quad \texttt{with}\ \texttt{newPoint} : int \rightarrow \boldsymbol{pro.}\,p \\
\quad\quad \texttt{is}\ \{\!\!|\, p <:_w P_{pub} :: \{\texttt{c}, \texttt{getC}, \texttt{setC}\} = P_{priv},\ \texttt{Imp}_p \,|\!\!\} \\
\quad\quad \texttt{in} \\
\quad\quad\quad \texttt{Abstype}\ cp <:_w CP_{pub} :: \emptyset \\
\quad\quad\quad\quad \texttt{with}\ \texttt{newCPoint} : int \rightarrow color \rightarrow \boldsymbol{pro.}\,cp \\
\quad\quad\quad\quad\quad \texttt{is}\ \{\!\!|\, cp <:_w CP_{pub} :: \emptyset = CP_{priv},\ \texttt{Imp}_{cp} \,|\!\!\} \\
\quad\quad\quad\quad\quad \texttt{in} \\
\quad\quad\quad\quad\quad\quad \texttt{let}\ \ \texttt{newPoint} : int \rightarrow \boldsymbol{obj.}\,p = \texttt{newPoint} \\
\quad\quad\quad\quad\quad\quad \texttt{let}\ \ \texttt{newCPoint} : int \rightarrow color \rightarrow \boldsymbol{obj.}\,cp = \texttt{newCPoint} \\
\quad\quad\quad\quad\quad\quad \texttt{in}\ \ \ \langle \texttt{Program} \rangle
\end{array}
$$

where

$$
\begin{array}{lcl}
P_{pub} & \stackrel{def}{=} & \langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle \\
P_{priv} & \stackrel{def}{=} & \langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle \\
CP_{pub} & \stackrel{def}{=} & \langle\!\langle p \,|\, \texttt{getC} : color,\ \texttt{setC} : color \rightarrow unit \rangle\!\rangle \\
CP_{priv} & \stackrel{def}{=} & \langle\!\langle p \,|\, \texttt{c} : color,\ \texttt{getC} : color,\ \texttt{setC} : color \rightarrow unit \rangle\!\rangle
\end{array}
$$

and

$$
\begin{array}{l}
\texttt{Imp}_p \stackrel{def}{=} \lambda\, \texttt{xi} : int. \langle \quad \texttt{x} = \lambda(\texttt{self})\ \ \texttt{0}; \\
\qquad\qquad\qquad\qquad \texttt{getX} = \lambda(\texttt{self})\ \ \texttt{self} \Leftarrow \texttt{x}; \\
\qquad\qquad\qquad\qquad \texttt{setX} = \lambda(\texttt{self})\ \ \lambda(\texttt{newX})\ \ \texttt{self.x:=newX} \rangle \Leftarrow \texttt{setX}\ \ \texttt{xi};
\end{array}
$$

$$
\begin{array}{l}
\texttt{Imp}_{cp} \stackrel{def}{=} \lambda\, \texttt{xi} : int. \lambda\, \texttt{ci} : color. \\
\qquad\qquad\qquad \langle\, \texttt{Imp}_p\ \ \texttt{xi}\ \ \longleftarrow\!\!\!+ \quad \texttt{c} = \lambda(\texttt{self})\ \ \texttt{red}; \\
\qquad\qquad\qquad\qquad\qquad\quad \longleftarrow\!\!\!+\ \texttt{getC} = \lambda(\texttt{self})\ \ \texttt{self} \Leftarrow \texttt{c}; \\
\qquad\qquad\qquad\qquad\qquad\quad \longleftarrow\!\!\!+\ \texttt{setC} = \lambda(\texttt{self})\ \ \lambda(\texttt{newC}) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{self.c:=newC} \rangle \Leftarrow \texttt{setC}\ \texttt{ic};
\end{array}
$$

Fig. 5. Nested abstract datatypes for `Point` and `ColorPoint` classes.

Using extensible objects and the abstract datatype mechanism from Section 6.2, we may encode our example class hierarchy as shown in Figure 5. As in the previous section, we work with an imperative version of the underlying object calculus to avoid complications arising from method specialization[3]. The class interpretation presented in the next section provides enough machinery to allow

[3] The most relevant soundness proof for an imperative version of the underlying object calculus appears in [DF96], which also includes support for concurrency.

us to address the method specialization problem directly; it does not require an imperative semantics. However, as this additional machinery is somewhat complex, we first present the basic ideas behind the "Classes = Extensible Objects + Data Abstraction" model without support for method specialization.

In order, the two abstract type declarations in Figure 5 define the `Point` and `ColorPoint` classes. The `Abstype` clause of the `Point` class declaration introduces row variable p, which formally serves as the "name" of the `Point` class. Row p is bounded by row P_{pub}, `Point`'s public interface. This row, defined in Figure 5, lists the methods available to clients and descendants of the `Point` class, namely `getX` and `setX`. The kind $\{\mathtt{c}, \mathtt{getC}, \mathtt{setC}\}$ indicates that descendant classes, *e.g.*, `ColorPoint`, may add methods with these names.

The `with` clause of the declaration specifies that the constructor for the `Point` class, named `newPoint`, will return expressions with **pro** type, specifically with type $\mathbf{pro}.p$, so that descendant classes may define their objects via extension from the `Point` class constructor. The row variable p in this return type signals that objects returned by the constructor are defined in the `Point` class. We call such types *partially abstract* because a portion of their structure is hidden.

The `is` clause of the abstraction describes the implementation of the `Point` class. Within this implementation, the row P_{priv} describes the full representation of `Point` class objects by listing all of the methods defined in the class and giving their associated types. The expression $\mathtt{Imp}_p$, defined towards the bottom of Figure 5, gives the implementation of the constructor `newPoint`. The type system insures that $\mathtt{Imp}_p$'s type is $\mathbf{pro}.p$ with p replaced by `Point`'s private interface, P_{priv}.

The structure of the `ColorPoint` class is similar. There are several aspects of this declaration that are worth mentioning however. In particular, notice that the `ColorPoint` constructor, implemented via the $\mathtt{Imp}_{cp}$ function defined at the bottom of Figure 5, invokes `newPoint` to inherit `Point`'s implementation and initialization code. Notice also that the `ColorPoint` class publicly names the `Point` class as its parent by listing p in its public interface CP_{pub}. Without thus naming its parent, the `ColorPoint` object type ($\mathbf{obj}.cp$) would not be a subtype of `Point`'s object type ($\mathbf{obj}.p$) because the only way to be a subtype of a partially abstract type is to extend that type. The row variable p in CP_{pub} signals that `ColorPoint` objects were formed via extension from `Point` prototypes, and hence their partially abstract types may be related via subtyping. This fact is the theoretical counterpart to the observation we made in Section 2 that the only way for implementation types to be related via subtyping is for them to be related via inheritance.

Since the `ColorPoint` class could have omitted p from CP_{pub}, this formulation of classes gives programmers explicit control over whether or not they wish a derived class's implementation object type to be a subtype of its parent's. As described in Section 2, such explicit control is useful for insuring that all objects of a certain type satisfy particular invariants (by guaranteeing they share a particular implementation). It is also useful for improving the efficiency of message sending and for typing binary methods. Additionally, when we combine this encapsulation mechanism with constructors that return the non-extensible form of object, we get *final* classes, classes that have no descendants. Such classes are used in Java as a security measure and in optimizing compilers to speed message sending.

Finally, the two `let` declarations in Figure 5 give constructors for the two classes that return proper objects, instead of prototypes. Technically, this shadowing is type correct because **pro** types are subtypes of **obj** types [FM95a].

6.4 Evaluation

The example in this section illustrates that a principled way to think about class-based object-oriented languages is as the combination of two orthogonal components: one, an object system that supports inheritance and message sending and two, an encapsulation mechanism that provides hiding. Referring to the class-evaluation checklist in Section 2.6, we can see that this approach successfully addresses each of the points listed there.

Coherent, Extensible Collection. The type checking of our object calculus insures that the methods defined in each class are compatible with each other: the methods are typed at class-creation time, not at object-instantiation time.

Guaranteed Initialization. The data abstraction mechanism provided by this framework allows each class to control the initialization of its objects by insuring that the only way to get an object from a particular class is to call its constructor. Hence class-writers can be sure that any invariants necessary for their objects will hold for all non-trivial expressions with the object type defined in their class. For example, the only way to get a nontrivial expression with type $\mathbf{obj}.p$ is to call the constructor for the `Point` class, insuring that such an expression has been properly initialized. This property holds with respect to derived classes as well, since the only way for the `ColorPoint` class to inherit from `Point` is to invoke the `Point` class constructor, again guaranteeing that `Point`'s invariants will hold.

Enforced Access Restrictions. The mechanism presented here also allows programmers to explicitly state which methods are public and which are private. Private methods are guaranteed to be used only within the class implementation itself; hence they may be changed with great flexibility.

Explicit Type Hierarchy. An additional strength of this model is that it provides explicit control over the bottom portion of the subtyping hierarchy, as discussed in Section 2.5. In particular, object types that include existentially bound row variables correspond to implementation types; the row variable signals the class which defined the type, putting such types in one-to-one correspondence with classes. The only subtyping that holds between these types is that declared via the inheritance hierarchy, *e.g.*, for $\mathbf{obj}.cp$ to be a subtype of $\mathbf{obj}.p$, the `ColorPoint` class had to inherit from `Point`. Furthermore, to enable this subtyping relationship, the `ColorPoint` class had to explicit list `Point` as its parent by including p in CP_{pub}, the public interface for `ColorPoint` objects. Thus the bottom portion of the inheritance hierarchy is explicitly under programmer control. In contrast, object types without row variables correspond to interface types. For these types, subtyping is determined by inference rules that examine their structure. Finally, since we have subtyping relationships between these two kinds of types, *e.g.*,

$$\mathbf{obj}.p <: \mathbf{obj}.\langle\!\langle \mathtt{getX} : int, \mathtt{setX} : int \rightarrow unit \rangle\!\rangle$$

we get the rich, hybrid subtyping hierarchy described in Section 2.5. In particular, we get the benefits of implementation types for the bottom portion of the hierarchy, while maintaining the flexibility of interface types in the top portion.

Automatic Propagation of Base Class Changes. The class structure described in this section provides explicit support for inheritance. In particular, when a derived class inherits from its parent, it must do so by calling its parent's constructor since the full representation of the parent is hidden. Hence changes to the parent class are automatically reflected in all derived classes. If such a change causes a type error in a derived class, that error is detected as soon as the derived class is rechecked.

Problems with this approach. A weakness of the current formulation of our idea that "Classes = Objects + Data Abstraction" is that subclasses must be declared within the scope of their parent classes. We believe this shortcoming may be overcome by replacing our block-structured construct with modules and a dot notation in the style of [CL90, HL94]. A further weakness of our interpretation is the need for negative information. In effect, parent classes must currently "guess" what methods their descendants will want to add. We believe a variant of the kinding system used here that tracks method *presences* instead of absences combined with some form of α-binding to hide private method names may solve this problem. However, we leave these tasks to future work.

A problem that we do know how to solve, however, is that of finding a way for method types to refer to the type of their host object. Such a type, frequently called "mytype," is useful both in automatically specializing method types during inheritance and in supporting binary methods. As an example of the first of these roles, we would like the `setX` and `setC` methods from our example to have return types indicating that the expressions they return have the same type as the object to which they were sent. (Recall that we adopted an imperative semantics earlier to side-step this problem.) Because such types change during inheritance *e.g.*, `setX` returns *Point* objects when it is sent to *Point*s and *ColorPoint* objects when sent to *ColorPoint*s, the ability to name "mytype" is said to support *method specialization.* It also allows us to write binary methods. In the next section, we present a framework that combines the strengths (and the above weaknesses) of the current system with additional support for method specialization and binary methods.

7 Classes with "Mytype" Specialization

To support binary methods and method specialization, we need to have a way for method types to refer to the type of their host object. To that end, we use type variables and recall that both **pro** and **obj** are type binding operators. In particular, when a type variable u appears within one of the types τ_i in

$$\mathbf{pro}\, u.\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_n} : \tau_n \rangle\!\rangle$$

u refers to the whole type (*i.e.*, to $\mathbf{pro}\, u.\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_n} : \tau_n \rangle\!\rangle$), and similarly for **obj** types.

The row variables that we used to express unknown lists of methods in the previous section must now be parametric in the type of the host object. Hence in this section, row variables will range over *row functions*, functions from the type of the self object to the list of method-type pairs it supports. Row functions are written using the notation $\lambda t.\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_k} : \tau_k \rangle\!\rangle$.

The final change necessary is that we need to be able to infer subtyping relationships between types of the form $\mathbf{obj}\, u.cp\, u$ and $\mathbf{obj}\, u.p\, u$. Now simply knowing that the *cp* row variable must stand for a row that extends whatever row p represents is not sufficient to insure the corresponding types are subtypes, because of the possible contravariance of the u type variable in the type $\mathbf{obj}\, u.p\, u$. Nor can we simply look at the type $\mathbf{obj}\, u.p\, u$ to determine that u appears covariantly, since the type contains only type and row variables. Instead, we need to add a variance analysis system to our kind system to keep track of the variance of type variables that appear in our type and row expressions. For example, if we may give a row function R kind $T^+ \rightarrow (M; \emptyset)$, then the variance annotation $+$ on the T indicates that R's argument type appears covariantly. The $\emptyset$ in the kind indicates that R contains no free type variables.

7.1 Example Classes with "Mytype"

Figure 6 shows our simple class hierarchy written in the full system. This version has the same structure as the one in the previous section, differing only in the addition of the "mytype" type variables and the variance annotations. Because of the richer type system, we may write the constructors for these two classes without using an imperative semantics, and we could write binary methods such as `eq` methods for the two classes.

7.2 Evaluation

The type system used to type check the above example is presented in full and proved type sound in [Fis96]. Because this version has the same class structure as in the previous section, it provides the same rich support for classes. Also as before, it provides a rich subtyping hierarchy, the bottom portion of which is under explicit programmer control. In addition, the system described here supports method specialization and binary methods. The remaining problems to be solved are to change the kinding system to track positive, instead of negative, information about method existence and to replace our block-structured abstraction mechanism with modules and a dot notation in the style of [CL90, HL94]. We leave these tasks to future work.

$$
\begin{array}{l}
\texttt{Abstype}\ p <:_w P_{pub} :: T^+ \to (\{\texttt{c}, \texttt{getC}, \texttt{setC}\}; \emptyset)\\
\quad \texttt{with newPoint} : int \to \textbf{pro}\, u\,\textbf{.}\, p\, u\\
\quad\quad \texttt{is}\ \{\!| p <:_w P_{pub} :: (T^+ \to (\{\texttt{c}, \texttt{getC}, \texttt{setC}\}; \emptyset)) = P_{priv},\ \texttt{Imp}_p |\!\}\\
\quad\quad \texttt{in}\\
\quad\quad\quad \texttt{Abstype}\ cp <:_w CP_{pub} :: T^+ \to (\emptyset; \emptyset)\\
\quad\quad\quad\quad \texttt{with newCPoint} : int \to color \to \textbf{pro}\, u\,\textbf{.}\, cp\, u\\
\quad\quad\quad\quad\quad \texttt{is}\ \{\!| cp <:_w CP_{pub} :: (T^+ \to (\emptyset; \emptyset)) = CP_{priv},\ \texttt{Imp}_{cp} |\!\}\\
\quad\quad\quad\quad\quad \texttt{in}\\
\quad\quad\quad\quad\quad\quad \texttt{let}\quad \texttt{newPoint} : int \to \boldsymbol{obj}\, u\,\textbf{.}\, p\, u = \texttt{newPoint}\\
\quad\quad\quad\quad\quad\quad \texttt{let}\ \ \texttt{newCPoint} : int \to color \to \boldsymbol{obj}\, u\,\textbf{.}\, cp\, u = \texttt{newCPoint}\\
\quad\quad\quad\quad\quad\quad \texttt{in}\quad \langle \texttt{Program} \rangle
\end{array}
$$

where

$$
\begin{array}{ll}
P_{pub} & = \lambda u.\langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \to u \rangle\!\rangle\\
P_{priv} & = \lambda u.\langle\!\langle \texttt{x} : int,\ \texttt{getX} : int,\ \texttt{setX} : int \to u \rangle\!\rangle\\
CP_{pub} & = \lambda u.\langle\!\langle p\, u \mid \texttt{getC} : color,\ \texttt{setC} : color \to u \rangle\!\rangle\\
CP_{priv} & = \lambda u.\langle\!\langle p\, u \mid \texttt{c} : color,\ \texttt{getC} : color,\ \texttt{setC} : color \to u \rangle\!\rangle
\end{array}
$$

and

$$
\begin{array}{lll}
\texttt{Imp}_p \stackrel{def}{=} \lambda\, \texttt{xi} : int.\langle & \texttt{x} = \lambda(\texttt{self})\ \texttt{0};\\
& \texttt{getX} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{x};\\
& \texttt{setX} = \lambda(\texttt{self})\ \lambda(\texttt{newX})\ \texttt{self} \leftarrow \lambda(\texttt{s})\texttt{newX}\rangle \Leftarrow \texttt{setX xi};
\end{array}
$$

$$
\begin{array}{l}
\texttt{Imp}_{cp} \stackrel{def}{=} \lambda\, \texttt{xi} : int.\lambda\, \texttt{ci} : color.\\
\qquad\qquad \langle\, \texttt{Imp}_p\ \texttt{xi} \ \longleftrightarrow\!+\ \ \texttt{c} = \lambda(\texttt{self})\ \texttt{red};\\
\qquad\qquad\qquad\qquad\quad \longleftrightarrow\!+\ \texttt{getC} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{c};\\
\qquad\qquad\qquad\qquad\quad \longleftrightarrow\!+\ \texttt{setC} = \lambda(\texttt{self})\ \lambda(\texttt{newC})\\
\qquad\qquad\qquad\qquad\qquad\qquad \texttt{self} \leftarrow \lambda(\texttt{s})\texttt{newC}\rangle \Leftarrow \texttt{setC ic};
\end{array}
$$

Fig. 6. `Point` and `ColorPoint` classes encoded as nested abstract datatypes with support for method specialization.

8 Other Approaches

8.1 Existential Model

In [PT94], Pierce and Turner interpret classes as object-generating functions. In their interpretation, inheritance is interpreted as modifications to the object-generating functions that model classes. This encoding is somewhat cumbersome, since it requires programmers to explicitly manipulate `get` and `put` functions, which intuitively convert between the hidden state of parent class objects and derived class objects. Also, because this model provides protection at the object-level, as opposed to the class-level, binary methods require extra machinery. One such solution appears in [PT93].

In [HP96], Hofmann and Pierce introduce a refined version of $F_{<:}$ that permits only positive subtyping. With this restriction, `get` and `put` functions are both guaranteed to exist and hence may be handled in a more automatic fashion in class encodings. A disadvantage of this approach is that no subtyping is possible between existential types because the `put` functions for existential types are not meaningful. Since objects have existential type, this model does not currently support subtyping between object types; some form of explicit coercions are necessary. Extensions that combine positive subtyping with normal subtyping, addressing this lack of object subtyping, seem possible.

8.2 Direct Models

Kim Bruce has developed a family of type-safe formal languages that model classes directly instead of via an interpretation as the combination of more basic primitives. In [Bru93], Bruce describes TOOPL, a functional object-oriented language. PolyTOIL, presented in [BSv95], incorporates imperative features and introduces the notion of *matching*, a relationship between object types that holds whenever the first is an extension of the second, regardless of the variance of the "mytype" type variable. In these languages, the type of an object reflects only its public interface; it cannot convey implementation information.

Scott Smith and the Hopkins Object Group have designed a type-safe class-based object-oriented language with a rich feature set called I-LOOP, [EST95]. Their type system is based on polymorphic recursively constrained types, for which they have a sound type inferencing algorithm. The main advantage of this approach is the extreme flexibility afforded by recursively constrained types. Currently, the main problem is that it returns large, difficult-to-read types. Some form of simplification may be required. Work in this area is in progress.

9 Conclusion

Pure object systems are simpler than class-based systems; however, most of the empirical evidence for the utility of object-oriented languages is based on class-based languages, and certain features of classes seem critical to this success. These features include:

- Static checking that guarantees that the object "parts" defined in a class are consistent with each other (for example, all premethods must have consistent assumptions about the type of "self").
- Control over the initialization of objects (both in a given class and in all of its descendants). This control is essential for establishing inheritable invariants, for example.
- The ability to specify which methods and fields of a given class are private and which are public. These specifications should be meaningful both within the class itself and within its descendants.
- Explicit control over a portion of the subtyping hierarchy.

– Preservation of the relationships between classes when private or public components are added. This property implies that when changes are made to a parent class, those changes are automatically reflected in its descendants.

We believe it is important to evaluate pure object systems in light of their ability to support class-based programming styles. The fundamental constructs that seem necessary are (i) an extensible form of object, or perhaps pre-object, to support inheritance, (ii) a non-extensible form of object that can be created easily from an extensible form, and (iii) subtyping on the types of proper objects. Given structural subtyping on object types, we have shown in this paper how to gain more precise control over the class hierarchy using standard data abstraction, enhanced with subtype constraints as described in [CW85].

A further insight from the analysis in this paper is that the Abadi/Cardelli and Fisher/Mitchell calculi are incomparable. The Fisher/Mitchell calculi were originally formulated with method specialization in mind, which has led to typing rules for object formation that enforce a high degree of polymorphism. This polymorphism is necessary to make each method that is added to a prototype suitable for inclusion in any future extension of that prototype. However, it precludes the formation of certain objects that would behave sensibly only if not extended. The record-of-premethod class objects in Section 5 are an instance of this phenomenon. In its current form, the Fisher/Mitchell calculi cannot type these objects. Explicit universal quantification over rows must be added to remedy this deficiency. On the other hand, the Abadi/Cardelli family of calculi do not provide any form of extensible object. This lack makes it difficult to support class-based programming, for the reasons enumerated in Section 5. A key technical issue for further exploration is to devise a system with two forms of objects: an extensible form (similar to our prototypes) for representing class implementations and second form that supports method override and width subtyping (like the Abadi/Cardelli objects) to behave as proper objects, with a natural form of conversion between the first and the second forms. We do not see any fundamental impediment to completing a study of this additional part of the design space. Recent work by Luigi Liquori [Liq96] addresses this issue.

References

[Aba96] M. Abadi. A few thoughts on objects (ML2000). Private Email, August 1996.

[AC95] M. Abadi and L. Cardelli. A theory of primitive objects: Second-order systems. *Science of Computer Programming*, 25(2-3):81–116, December 1995. A preliminary version appeared in the 1994 Proc. of European Symposium on Programming.

[AC96a] M. Abadi and L. Cardelli. An imperative object calculus. *Theory and Practice of Object Systems*, 1(3):151–166, 1996. Earlier version appeared in TAPSOFT '95 proceedings.

[AC96b] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer, 1996.

[AC96c] M. Abadi and L. Cardelli. A theory of primitive objects: Untyped and first-order systems. *Information and Computation*, 125(2):78–102, 1996. Earlier version appeared in TACS '94 proceedings, LNCS 789.

[AG96] K. Arnold and J. Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[BCC+96] K. Bruce, L. Cardelli, G. Castagna, The Hopkins Object Group, G. Leavens, and B. Pierce. On binary methods. *Theory and Practice of Object Systems*, 1(3):221–242, 1996.

[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.

[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.

[Car95] L. Cardelli. A language with distributed scope. *Computing Systems*, 8(1):27–59, January 1995.

[Cha95] C. Chambers. The Cecil language: Specification and rationale. Available electronically from http://www.cs.washington.edu/research/projects/cecil/www/Papers/cecil-spec.html, December 1995.

[CL90] L. Cardelli and X. Leroy. *Abstract types and the dot notation*, pages 479–504. IFIP State of the Art Reports. North Holland, March 1990. Also appeared as SRC Research Report 56.

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

[Coo89] W.R. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

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

[DF96] P. DiBlasio and K. Fisher. A concurrent object calculus. In *CONCUR '96 Proc.*, pages 655–670, Pisa, 1996. Springer LNCS 1119.

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

[EST95] J. Eifrig, S. Smith, and V. Trifonov. Sound polymorphic type inference for objects. In *ACM Conf. Object-Oriented Programming: Systems, Languages and Applications*, pages 169–184, October 1995.

[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.

[Fis96] K. Fisher. *Type Systems for object-oriented programming languages*. PhD thesis, Stanford University, 1996.

[FM95a] 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.

[FM95b] K. Fisher and J.C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220,

1995. Preliminary version appeared in TACS '94 proceedings.

[FM96] K. Fisher and J.C. Mitchell. Classes = Objects + Data Abstraction. Technical Report STAN-CS-TN-96-31, Stanford University, 1996

[GM94] C.A. Gunter and J.C. Mitchell, editors. *Theoretical Aspects of Object-Oriented Programming*. MIT Press, Cambridge, MA, 1994.

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

[HL94] R. Harper and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Proc. 21st ACM Symp. on Principles of Programming Languages*, 1994.

[HP96] M. Hofmann and B. Pierce. Positive subtyping. *Information and Computation*, 126(1):11–33, 1996. Preliminary version appeared in *Proc. 22nd ACM Symp. on Principles of Programming Languages*, 1995.

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

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

[Liq96] L. Liquori. An extended theory of primitive objects: First and second order systems. Technical Report CS-23-96, Dipartimento di Informatica, Universitá di Torino, 1996.

[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.

[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.

[Mit93] J.C. Mitchell. *Introduction to Programming Language Theory*. MIT Press, to be published 1993.

[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.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[PT93] B.C. Pierce and D.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] B.C. Pierce and D.N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–248, 1994. Preliminary version appeared in *Proc. 20th ACM Symp. on Principles of Programming Languages*, 1993, under the title "Object-oriented programming without recursive types".

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

[RR96] J.H. Reppy and J.G. Riecke. Classes in Object ML via modules, 1996. Presented at FOOL3 workshop.

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

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

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

[Str94] B. Stroustrup. *The Design and Evolution of C^{++}*. Addison-Wesley, 1994. Chapter 3: The birth of C^{++}.

[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.

[Wan87] M. Wand. Complete type inference for simple objects. In *Proc. IEEE Symp. on Logic in Computer Science*, pages 37–44, 1987. Corrigendum in *Proc. IEEE Symp. on Logic in Computer Science,* page 132, 1988.