On the relationship between classes, objects, and data abstraction

Kathleen Fisher
AT&T Labs—Research, 180 Park Ave., Florham Park, NJ 07932
kfisher@research.att.com

John C. Mitchell *
Computer Science Department, Stanford University, Stanford, CA 94305
mitchell@cs.stanford.edu

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 class constructs of the form found in C++, Eiffel, and Java and object primitives of the form used in recent type-theoretic studies. One insight is that classes require both an extensible aggregate, to serve as the basis for inheritance, and a non-extensible form of object to support subtyping; typed object calculi without extensible objects or extensible records do not seem adequate for conventional class-based programming. We develop our analysis by comparing three approaches to class-based programming, 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 is simplest, using fewer primitive operations on objects, it does not seem to accurately provide several features of conventional class-based languages. In the latter two approaches, we give more comprehensive treatments of classes by combining prototypes with standard abstraction mechanisms. All three treatments of classes are based on typed translations into provably sound object calculi.

1. Introduction

There are several forms of object-oriented programming languages, including class-based languages such as C++ [20, 47], Eiffel [35], Java [6], Simula [7], and Smalltalk [26]; prototype-based languages such as Self [48] and Obliq [12]; and multi-method-based approaches such as CommonLisp [46] and Cecil [15]. This paper is concerned with the study of class-based languages and their relationship to prototype-based languages. In particular, we study ways to translate class-based object declarations, inheritance, and subtyping into languages that do not have classes. Since class-based encapsulation appears to be a separable concept that can be provided by a form of traditional data abstraction, our translations lead to a study of the relationships between three language constructs: classes, prototype-based object primitives, and traditional data abstraction of the form found in languages such as CLU [34, 32], Ada [49], and ML [36].

Our study begins with an overview of essential properties of class constructs, as they appear in conventional typed, class-based object-oriented languages. Focusing on core class features shared by C++ [20, 47], Eiffel [35], Java [6], and Simula [7], we develop criteria for comparing and evaluating different approaches to class-based programming. We then explain three such approaches using three translations of a skeletal class construct into more basic object calculi. The translations differ in two ways: the degree to which they respect important properties of classes and the complexity of the object operations used in the target calculus. Since these encodings are meant to provide some insight into the degree to which conventional object-oriented programming could be carried out in a statically-typed language without classes, we compare them using our design criteria for class constructs. We believe that the differences between the three translations provide useful insight into the suitability of object-based languages for carrying out traditional class-based programming.

To the best of our knowledge, the translations in this paper provide the first type soundness proofs for standard forms of class constructs, as a consequence of the type soundness of their target object calculi. The soundness proof for the first target calculus appears in [4]; the analysis of extensible object calculi is developed

* Supported in part by NSF Grants CCR-9303099 and CCR-9629754.

in full in [21]. 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 [22, 23] for discussions of type soundness proofs for extensible object calculi and [24] 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 book by Abadi and Cardelli [4].

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 class encodings. 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, 9, and 10 describe related work, outline future directions, 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. While purists from various language camps may differ on the value of certain aspects of each language, we believe it is important to understand the essential properties of core class concepts common to accepted object-oriented languages. Although Smalltalk or Beta enthusiasts may regard portions of C++ as awkward or ugly, or conversely, we believe that the rise in popularity of class-based object-oriented languages over the past three decades is ample evidence that these languages provide something of use to a large number of programmers and software designers. In the interest of promoting sound scientific discourse on the difference between various object-oriented programming languages and program development styles, it seems important to try to distill the main features of these object systems and understand their underlying language principles and properties. In carrying out the study reported in this paper and elsewhere, it is not our objective to promote any specific programming language or approach. Our goal is to analyze certain phenomena that are common to Eiffel, Java, C++, and related languages using the framework of type-theoretic analysis [38].

As a starting point, we first examine core aspects of class-based programming shared by these languages. The goal of this section is to explain our assumptions and establish a working terminology. In brief, a class declaration provides a mechanism for creating objects that all share a common 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. Additionally, classes 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 from it will share. Finally, in contemporary languages, class declarations explicitly determine the subtyping relation between object types. In the remainder of this section, we review each of these issues in turn.

2.1. *Defining Objects*

A class declaration provides a mechanism for creating objects. Typically, all objects created from a single class have the same representation, with the values of some data fields differing from object to object. 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.

class `Point` {

`x`	:	*int*
`setX`	:	$int \to Point$
`getX`	:	*int*
`move`	:	$int \to Point$
constructor `newPoint`	:	$int \to Point$

}

Each object constructed from this class has an `x` component, representing the location of the point. 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 created from a given class are usually shared. In Smalltalk for example, data, called *instance variables*, are stored within each object, while functions, called *methods*, are shared by all objects created from the same class. The distinction between components allocated per object and per class does not always fall between data and functions, but this division is a common one. In the `Point` example, the `x` component would be an instance variable in most languages, set to different values in different `Point` objects, while the `setX`, `getX`, and `move` methods 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. Constructor definition is an important part of class design since each object must be initialized in accordance with any intended object invariants. 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.

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:

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

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 redefine (or override) existing methods. 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. In most languages, the derived class constructor will need to call the base class constructor to initialize any private data associated with base class objects.

As this example illustrates, inheritance is a code-reuse mechanism. If we ignore typing considerations, then in principle, for every object or class of objects defined via inheritance, there is a functionally 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. Moreover, while it is operationally natural to think of inheritance as an abbreviation mechanism that could be eliminated by syntactic expansion of programs, this view of inheritance does not accurately reflect the use of classes in program development. In particular, separate classes are meant to be developed separately, with designers of one class oblivious (to some degree, at least) of the details involved in implementing another. Therefore, inheritance has a significant impact on program maintenance and modification.

A convenient feature of class-based inheritance is that classes are essentially static structures. Specifically, 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 the corresponding 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 [22, 24, 23], 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 the 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

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

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

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 private components inherited from the parent class. Hence there must be a way for a base class to provide initialization to derived classes. Typically, each derived-class constructor invokes a parent class constructor for this purpose.

2.4. The Types of Objects

In most statically typed, class-based languages, a class declaration defines a type. Each class name is regarded as a type name, and all objects created by constructors of the class are given this type. In general, types in different programming languages may convey different kinds of information about the values that have each type. For example, a type may completely determine the size of an object in memory in some languages but not in others. In all typed object-oriented languages, the type of an object gives the public *interface* of the object, consisting of the names of public operations (methods or member functions) and their types (the types of method arguments and return values). If the implementations of objects of a given type may differ arbitrarily, then the type of an object may only convey a public interface. However, in current class-based languages, it is possible for a type (associated with a class) to also guarantee certain *implementation* properties. For example, given a C++ object type, the order of function pointers in the virtual function table can be determined. This inference is possible because the objects of this type must all be created by either the class of the same name or some derived class, and derived classes are implemented so that this correspondence between virtual function tables is maintained. Although there are varying amounts of implementation information that might be conveyed as part of the type of an object (depending on the design and, possibly, implementation of the programming language), we find it useful to distinguish between object types that are meant to give the interface only and object types that may guarantee additional implementation properties.

Although this expository technique has certain drawbacks, we adopt an informal but suggestive terminology that distinguishes between two forms of object type: *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.*, [5, 9, 22, 42] and the earlier papers appearing in [27], but it is not used in current class-based languages. In contrast, implementation types are a form of abstract type. In fact, one of the main points of the present paper is that the types associated with class names in conventional object-oriented languages are actually the traditional, accepted notion of abstract data type, generalized in a natural way to allow subtyping. (General discussion of abstract data types may be found in [33], for example.) An implementation type may guarantee both a public interface and a portion of the implementation of an object. Although interface types offer increased flexibility, existing languages such as C++, Java, and Eiffel ([20, 6, 35]) use implementation types since the type of an object gives some information about the ways that the object could have been created.

The distinctions we hope to clarify using the terminology of interface and implementation types also rely on separating the concept of a class as a way of defining the implementation of objects from the type name associated with a class. In hopes of distinguishing between the implementation part of a class and the type defined by a class, we will write 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. One goal of the paper is to show how "interface types" and "implementation types" may be consistently combined in a single language and type system. This combination may be useful in practical program development. However, our initial objective is simply to show, in precise terms, how the object types commonly used in prior theoretical analyses differ from the form of object types that arise in conventional class-based languages.

2.4.1. 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.

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

An expression with this type is guaranteed to denote an object that has at least `getX` and `move` methods. Furthermore, when the `getX` message is sent, the result is an object with the same type as the receiver, *i.e.*, $Point_{Inter}$. Similarly, sending the `move` message with an integer parameter results in an object with the type of the receiver. Since the return type of these methods is *Point*, we must use some 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. The constructor function `newPoint` does not appear in the interface because constructors are not part of the objects they create.

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

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

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 could 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 semantically have type $Point_{Inter}$. There is no need to inherit from the `Point` class.

A significant advantage of interface types is the flexibility they provide. For example, in a programming language with interface types, we could 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 support the same public interface and therefore may be given the same type. This allows the two kinds of matrices to be used interchangeably in any program. A benefit is that library functions may be written using a matrix interface type, without concern for how matrices might be implemented in any program using the library.

2.4.2. Implementation Types

Informally, 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 type $ColorPoint_{Imp}$, 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, possibly with `x` occurring before `c` in the representation if the language requires it. This implementation constraint is *not* reflected in the public interface.

There are several reasons why a language designer (or user) may choose to associate implementation types with classes. These include 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 issue 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 [20, Section10.7c]. Such optimizations are not possible in an untyped language such as Smalltalk [26] 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. This is reflected in the fact that different Java bytecodes are used for accessing methods when the class or interface are known.

Behavioral Guarantees. A 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. Considering interface types only, 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 [10] for a detailed discussion of binary methods.

2.4.3. Interface vs. Implementation Types

In summary, interface types and implementation types have complementary strengths:

	Interface	Implementation
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 [5, 9, 22, 42, 27]. It is possible to associate both an interface and an implementation type with each class, since, as we will see in the next section, an implementation type is semantically a subtype of the interface type obtained by "forgetting" the implementation constraints. This use of subtyping allows both forms of object type to be used to advantage 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 used to manipulate A objects (at least without type errors). The 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 natural 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 [25, 21] 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 [21]. 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 suggest some of the ways that classes are used to structure code by providing reuse (subtyping and inheritance) and encapsulation (access restrictions, initialization code) mechanisms. Based on the discussion of the preceding sections, we believe that a class mechanism should provide at least 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, 2.2]. A class defines an extensible collection of object components. These components may include methods, data, local constants, specifications of communication protocols, *etc.* In current languages, these components must be *coherent*, in the sense that if a class is well-formed, 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 or some form of indication that an integer method must be defined before objects can be created. *Extensibility* is the basis for inheritance, allowing derived classes to reuse object components from their parents, possibly adding additional coherent components and replacing others. Inheritance is statically checked, insuring that the components of a derived classes are consistent. While it would be possible, in principle, to check consistency at object creation time, there are software engineering arguments against this practice.

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. The purpose of this restriction is to guarantee that the implementor of a class may change its private implementation without introducing errors into clients or derived classes. For this data abstraction principle to hold, access controls must apply uniformly to both the current class and all derived classes: if a method is private in a given class, it cannot become public or overridden in some derived class.

Guaranteed Initialization [Sections 2.1 and 2.2]. Classes provide code to initialize the components of objects when they are created. This practice is essential for establishing invariants. For example, in a class defining doubly-linked lists, an empty list might consist of a list node with links set appropriately. There are several ways that initialization interacts with inheritance and encapsulation. Part of class-based encapsulation is that a derived class cannot access private components of a parent class. Therefore, a derived class must invoke code supplied by the parent class to initialize private components defined in the parent class. A derived class should not (and indeed cannot) be responsible for initializing and establishing invariants for inherited components.

Explicit Type Hierarchy [Sections 2.4 and 2.5]. In conventional class-based languages, the subtype relation on the types defined by classes is explicitly declared. If the type of objects defined by class `A` is to be recognized as a subtype of the type of objects defined by class `B`, then this relationship must be declared explicitly in the program. In C++, subtyping arises only by inheritance. In Java, there is a possibility to use both inheritance (Java `extends`) and explicit declaration of subtyping between classes and interfaces (Java `implements`). In the terminology of Section 2.4, classes define implementation types. Since there are advantages to both interface and implementation types, it might be beneficial in the future to define languages with both forms of type that further consider implementation types to be subtypes of the corresponding interface types. A language with such a type structure would allow programmers to use implementation types where the extra information is useful (such as in the types of binary methods) and interface types where more generality is required (such as in the argument types of library functions).

Automatic Propagation of Base Class Changes. Classes are intended to support program extension, modification, and maintenance. In particular, incremental changes to a base class are automatically propagated to all derived classes. If any type errors are introduced in derived classes because of such changes, these errors are reported at compile or link time, according to the language. 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 when the revised program is compiled, not when objects are instantiated from the derived class at run time.

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.

class $\langle$*class-name*$\rangle$ { : $\langle$*superclass*$\rangle$ }
  constructor $\langle$*name*$\rangle$: $\langle$*type*$\rangle \rightarrow \langle$*class-name*$\rangle$;
  private $\{\langle$*name*$\rangle$: $\langle$*type*$\rangle;\}^*$
  public $\{\langle$*name*$\rangle$: $\langle$*type*$\rangle;\}^*$
end;

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 from Section 3 these classes may be written as follows:

class `Point`
  constructor `newPoint` : $int \rightarrow Point$;
  private `x` : int;
  public `setX` : $int \rightarrow Point$;
    `getX` : int;
end;

class `ColorPoint : Point`
  constructor `newCPoint` : $int \rightarrow color \rightarrow ColorPoint$;
  private `c` : $color$;
  public `setC` : $color \rightarrow ColorPoint$;
    `getC` : $color$;
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 [12], 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 describe its operations using 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

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

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. We will use a dot notation (`a.x`) to denote field selection and an arrow ($\mathtt{a} \Leftarrow \mathtt{m}$) for message sending. The distinction in how these components are accessed may be illustrated using the object

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

We have the value of `a.x` is 6 simply by extracting the value of `x`, and

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

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 [5, 2, 3, 37, 22, 23]. 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 $\mathtt{x} = \lambda(\mathtt{self})\, 6$. 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 often 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. We use the following equality to evaluate message sending:

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

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

$$\langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_i} = e_i, \ldots, \mathtt{m_k} = e_k \rangle \leftarrow \mathtt{m_i} = e'_i \quad = \quad \langle \mathtt{m_1} = e_1, \ldots, \mathtt{m_i} = e'_i, \ldots, \mathtt{m_k} = e_k \rangle$$

These object operations are shared by the calculi defined in [5, 2, 3, 37, 22, 23] and subsequent variants. Before discussing the related operation of object extension, it will be useful to introduce object types.

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 [22, 23, 5, 2], **obj** types are essentially a specialized form of recursive type, with "folding" combined with object formation and "unfolding" combined with method invocation.

4.3. 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 $\longleftrightarrow\!\!+$, 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, a crucial aspect of object types [22, 5]. 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 ones. For example, assuming that $ColorPoint <: Point$, then $\mathbf{obj\,.} \langle\!\langle \mathtt{x} : ColorPoint \rangle\!\rangle$ is a depth subtype of $\mathbf{obj\,.} \langle\!\langle \mathtt{x} : Point \rangle\!\rangle$. Throughout this paper, we will use the symbol $<:_w$ to specify subtyping relationships in which only width subtyping has been used.

Because of the conflict between subtyping and object-based inheritance, two different forms of object type are useful in constructing classes: (i), an extensible form for use within class declarations to support inheritance and (ii), 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 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* [50] 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.

The second source of difficulty arises from the need to have a type operation to track object extension and 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 track object extension, we have equipped our type system with a type operation to extend a given row with a new method-type pair. We use the notation $\langle\!\langle R \,|\, m : \tau \rangle\!\rangle$ to denote the row that results from extending row R with method m of type τ. We then add a kinding system to insure that rows do not contain multiple occurrences of the same method name. Intuitively, this kinding system works by tracking method absences. Each kind consists of a set of method names, M. If row R has kind M, then R is guaranteed not to contain any methods listed in kind M. To add method m to row R, we need to deduce via kinding rules that row R has kind M and $m \in M$. Recent work reveals that a kinding system that tracks potential method *presences* will also suffice [8].

Since we will use both object extension and method override for inheritance, we lump the inheritance operations of 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 the extensible, overridable objects, reserving types of the form $\mathbf{obj}.\langle\!\langle \mathtt{m}_1 : \tau_1, \ldots, \mathtt{m}_k : \tau_k \rangle\!\rangle$ for proper objects. It turns out that we may obtain proper objects from these extensible ones via subsumption [23]:

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

4.4. Comparing Object Calculi

One insight from the analysis in this paper is that the Abadi/Cardelli calculi, which include only non-extensible objects, and the Fisher/Mitchell calculi, which also allow object extension, 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 [31] addresses this issue.

4.5. Existential Types 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 [34, 39, 40, 44, 45]; the connection between existential types and data abstraction is elaborated in [39, 38].

The declaration form

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

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

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

declares an abstract datatype *stream* with distinguished element s:*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 involve *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, respectively. Since the value of s must be a stream, the expression M_1 must have type τ, the type 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 $\mathit{stream} \times (\mathit{stream} \to \mathit{nat}) \times (\mathit{stream} \to \mathit{stream})$. We may recover s, *first*, and *rest* using projection functions.

Some useful flexibility is gained by considering abstract datatype declarations and implementations separately. An implementation for the abstract type *stream* consists of a type τ, used to represent streams, and expressions for the specified stream s and the *stream* operations *first* and *rest.* To describe implementations of streams in general, we might say that in any implementation, "there exists a type t with operations of types t, $t \rightarrow nat$, and $t \rightarrow t$." 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. Thus, this construct fits the general goals of data abstraction, as discussed in [40], for example.

We may add abstract datatype implementations and `Abstype` declarations to a type system by encoding them 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 $[\tau/t]\sigma$, where $[\tau/t]\sigma$ denotes the type obtained by substituting τ for t in σ. Using products to combine s, *first*, and *rest*, an implementation of *stream* would have type $\exists t.[t \times (t \rightarrow nat) \times (t \rightarrow t)]$, 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 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 $\exists t.\sigma$.

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 has type τ 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.

4.5.1. Existential Types and Objects Existential types are useful in the context of modeling objects because they can hide private components [42, 41]. 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

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

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 [16, 17], 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* [37].

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 [5]; these ideas are also used in [43]. In this approach, a class is an object that contains a record of premethods and a constructor function used to package these premethods into objects. We illustrate some of the advantages and disadvantages of the record-of-premethods approach by example.

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 (i) the object type associated with the `Point` class:

$$\textbf{obj.}\langle\!\langle \texttt{x} : int\ ref,\ \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle,$$

(ii) the record of `Point` premethods `PointImpl,` and (iii) the `Point` constructor `newPoint.` For the sake of simplicity, we assume an imperative semantics, with `setX` changing the value of an assignable `x` field. This assumption allows us to use *unit* as the return type of `setX`, side-stepping some difficulties associated with *method specialization.* In brief, method specialization occurs when the types of methods are "improved" during inheritance to reflect their new, more specialized

```
type Point        = obj.⟪x : int ref, getX : int, setX : int → unit⟫;

 val PointImpl    : record
                           x : int ref;
                        getX : Point → int;
                        setX : Point → int → unit;
                    end;
                  = record
                           x = ref 0;
                        getX = λ(self) !(self.x);
                        setX = λ(self) λ(newX)  self.x:= newX;
                    end;

 fun newPoint     : int → Point
                  = λ(xi)let s  =  ⟨          x = PointImpl.x,
                                           getX = PointImpl.getX,
                                           setX = PointImpl.setX  ⟩
                         in
                            s ⇐ setX xi;
                            s
                         end


type ColorPoint   = obj.⟪ x : int ref, getX : int, setX : int → unit,
                          c : color ref, getC : color, setC : color → unit⟫

 val CPointImpl   : record
                           x : int ref;
                        getX : ColorPoint → int;
                        setX : ColorPoint → int → unit;
                           c : color ref;
                        getC : ColorPoint → color;
                        setC : ColorPoint → color → unit;
                    end;
                  = record
                           x = PointImpl.x;
                        getX = PointImpl.getX;
                        setX = PointImpl.setX;
                           c =  ref red;
                        getC = λ(self) !(self.c);
                        setC = λ(self) λ(newC) self.c: = newC;
                    end

 fun newCPoint    : int → color  → ColorPoint
                  = λ(xi)λ(ci)let s  =  ⟨     x = CPointImpl.x,
                                           getX = CPointImpl.getX,
                                           setX = CPointImpl.setX  ⟩
                                              c = CPointImpl.c  ⟩
                                           getC = CPointImpl.getC,
                                           setC = CPointImpl.setC  ⟩
                         in
                            s ⇐ setX xi;
                            s ⇐ setC ci;
                            s
                         end
```

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

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 the record-of-premethods approach can support method specialization, significantly more complex types are required to do so [4]. (The reason the `x` component of the `PointImpl` record does not take *Point* as an argument is because it is a field, not a method.)

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 $ColorPoint <: Point$, and therefore, by the contravariance of function types, we have $Point \rightarrow \tau <: ColorPoint \rightarrow \tau$ 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, `newPoint`, because objects are not extensible in this model. Thus, we cannot use a `Point` object to create a `ColorPoint` object (except trivially).

5.2. Adding Protection

In the basic record-of-premethods encoding shown in Figure 1, the `x` and `c` fields of a point or color point object are not private; they are just as accessible as the public methods. There are three ways of incorporating private fields into the record-of-premethods approach, each with advantages and disadvantages. The first approach uses subtyping to restrict the return type of constructor functions to reflect only public methods. The second uses a variation of the existential-type object model. The third uses closures to create private fields whenever a constructor is called.

5.2.1. Protection Via Constructor Return Type

In this modification of the basic encoding, illustrated in Figure 2, there are two object types associated with the `Point` class. Type *Point* lists the public components, while $Point_{Priv}$ 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 since its return type is *Point* instead of $Point_{Priv}$. This change of type is allowed because $Point_{Priv}$ is a subtype of *Point* (in symbols, $Point_{Priv} <: Point$).

Although this use of subtyping hides the `x` component from clients who invoke the `newPoint` constructor function, it fails to hide the `x` premethod in the record used by derived classes. 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 part of the class. A second encapsulation problem stems from the fact that the `ColorPoint` class must explicitly extract the private components of `PointImpl` to inherit them. In summary, this modification using constructor subtyping provides private methods and fields, but derived classes must be aware of the private components of parent classes.

5.2.2. A Refinement

Martin Abadi has suggested a refinement to the record-of-premethods approach that uses a form of existential type to hide the types of private components [1]. (A similar idea appears in [43].) Figure 3 illustrates the idea using the declarations associated with the `Point` class.

In this refinement, the `x` premethod is still contained in the `PointImpl` record, but the type of `x` is hidden by an existential binding of its type. Since the other premethods of the `Point` class are within the scope of this existential binding, they can use the `x` premethod in a non-trivial way. However, other functions cannot perform any operations on `x` that are specific to the type of `x`. The main differences between this encoding and the subtyping approach illustrated in Figure 2 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 $Point_{Priv}(R)$, the private type instantiated to the hidden representation type R,
- implementing `PointImpl` as an element of existential type and binding the representation type variable R to the selected representation type, $int\ ref$, and
- moving the code for the constructor function into `PointImpl` so that it will be within the scope of the hidden representation type.

This use of existential types hides the private `x` component from general use: since the `x` premethod has an unknown type, clients cannot access or manipulate it in any non-trivial way. However, to use existential types, it is necessary to complicate the object calculus with existential types.

A remaining problem is that derived classes must explicitly name the private components they wish to inherit. For private fields, there is at least a partial solution to this naming problem [1]. For example, each class could have a special component named "`private_fields`" that stores a record of all the private fields. Instead of naming each of the fields individually, then, a derived class can simply refer to `private_fields`. At the price of this one name being globally known, we can insure that the actual private names do not leak out.

```
type Point      = obj.《getX : int, setX : int → unit》;
type Point_Priv = obj.《x : int ref, getX : int, setX : int → unit》;

val PointImpl   : record
                       x : int ref;
                    getX : Point_Priv → int;
                    setX : Point_Priv → int → unit;
                  end;
                = record
                       x = ref 0;
                    getX = λ(self) !(self.x);
                    setX = λ(self) λ(newX) self.x := newX;
                  end;

fun newPoint    : int → Point
                = λ(xi)let s = 〈    x = PointImpl.x,
                                 getX = PointImpl.getX,
                                 setX = PointImpl.setX 〉
                       in
                          s ⇐ setX xi;
                          s
                       end
```

FIG. 2. Declarations for the `Point` class with modifications to support private components via restricting the return type of `Point`'s constructor function.

In this variant of the record-of-premethods model, class designers can guarantee that objects are initialized properly. Intuitively, this guarantee is possible because the only way to build a `Point` object is to call the `newPoint` function, and the `Point` class designers can insure that they properly initialize all of their ancestor classes. Unfortunately, this process works only for the specific class where a constructor is defined; the initialization guarantees are not preserved under inheritance. The problem arises from the fact that inheritance is based on records of premethods instead of constructor functions. Because inherited private components have unknown type, derived classes cannot initialize them directly. Since constructor functions return non-extensible objects, they cannot be used for this initialization either, except trivially. Instead, parent classes must supply separate initialization functions for this purpose. However, nothing requires derived classes to invoke the supplied initialization code. Hence the designers of parent classes cannot assume that derived classes will always properly initialize the components they inherit. 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.

A further weakness of this technique, pointed out to us by an anonymous referee, is that under call-by-name semantics, this approach to protection can be subverted. Briefly, if we define an identity function $\mathtt{Id} : Point_{Priv}(R) \rightarrow Point_{Priv}(R)$, then the fixed point `Y(Id)` has type $Point_{Priv}(R)$. If the premethod `getX` is applied to this value but does not depend on it, `getX` will return its supposedly hidden data. For example, if the `getX` premethod has the form $\lambda(\mathtt{self})3$ and it is applied to `Y(Id)`, it will return its "private" data 3. Under call-by-value semantics, the term `Y(Id)` will diverge, foiling this particular attack. However, there may be related problems using the call-by-value fixed-point operator.

5.2.3. Protection Via Closures Figure 4 shows a version of the `Point` class, modified to incorporate closure-style private fields. The necessary changes to the `Point` class in Figure 1 include:

- removing the `x` field from type *Point* to indicate that `x` is not publicly available,
- making `PointImpl` into a function that allocates storage for private field `x` before creating the record of premethods,
- rewriting the remaining premethods to reference the `let`-allocated variable `x`,
- modifying the constructor function `newPoint` to first call `PointImpl` with `xi` to create a record of premethods `meth` referencing private field `x`, and
- installing the premethods in `meth` into the final object.

This approach succeeds in hiding the private components of parent classes and in insuring that these components are properly initialized by all derived classes. One drawback is that it supports only private fields; private methods cannot be installed into objects without violating encapsulation. A further drawback is that

```
type Point          = obj.⟪getX : int, setX : int → unit⟫;
type Point_Priv(R)  = obj.⟪x : R, getX : int, setX : int → unit⟫;

 val PointImpl      : ∃R.record
                              x : int ref;
                           getX : Point_Priv(R) → int;
                           setX : Point_Priv(R) → int → unit;
                           newP : int → Point;
                           end;
                    = ⟨ R  =  int ref,
                        record
                              x = ref 0;
                           getX = λ(self) !(self.x);
                           setX = λ(self) λ(newX) self.x := newX;
                           newP = λ(xi)let s  =  ⟨      x = PointImpl.x,
                                                     getX = PointImpl.getX,
                                                     setX = PointImpl.setX  ⟩
                                       in
                                          s ⇐ setX xi;
                                          s
                                       end
                        end⟩;

 fun newPoint       : int → Point
                    = λ(xi)PointImpl.newP xi
```

FIG. 3. 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 components via data abstraction.

because each ancestor class of an object potentially requires a closure, it seems difficult to implement these objects efficiently, *i.e.*, in such a way that they may share method suites.

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. Most of the analysis is applicable to all three protection schemes.

5.3. 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 the record-of-premethods model, the code to initialize private instance variables is guaranteed to run if any of the associated premethods is installed into an object. However, constructor functions may not be usefully reused in derived classes. A consequence is that if a class designer puts additional initialization code into a class constructor, there is no guarantee that this code will be executed for all derived classes. (In fact, it is guaranteed not to be.) There are several program-development scenarios where this lack would be a serious problem. For example, additional initialization code is useful if class designers wish to use methods of an object to perform initialization. These methods are not available until an object that contains them has been constructed, and hence they cannot be used in the premethod function. Another scenario that arises is that class designers may wish to perform some kind of bookkeeping whenever objects are instantiated from a class or its descendants. Such bookkeeping is common in database applications, for example. To achieve it, programmers need a place to put code that will execute whenever an object is instantiated. With the program structure associated with the record-of-premethods approach, however, there is

$$
\begin{array}{llll}
\texttt{type}\ \mathit{Point} & = & \multicolumn{2}{l}{\mathbf{obj.}\langle\!\langle \texttt{getX} : \mathit{int},\ \texttt{setX} : \mathit{int} \to \mathit{unit} \rangle\!\rangle;} \\
\texttt{val PointImpl} & : & \mathit{int} \to & \mathit{record} \\
 & & & \quad \texttt{getX} : \mathit{Point} \to \mathit{int}; \\
 & & & \quad \texttt{setX} : \mathit{Point} \to \mathit{int} \to \mathit{unit}; \\
 & & & \mathit{end}; \\
 & = & \lambda(\texttt{xi}) & \texttt{let x} = \texttt{ref xi in} \\
 & & & \quad \texttt{record} \\
 & & & \qquad \texttt{getX} = \lambda(\texttt{self})\,!\texttt{x}; \\
 & & & \qquad \texttt{setX} = \lambda(\texttt{self})\,\lambda(\texttt{newX})\,\texttt{x:} = \texttt{newX}; \\
 & & & \quad \texttt{end}; \\
 & & & \texttt{end}; \\
\texttt{fun newPoint} & : & \mathit{int} \to \mathit{Point} & \\
 & = & \lambda(\texttt{xi}) & \texttt{let meth} = \texttt{PointImpl xi in} \\
 & & & \langle\, \texttt{getX} = \texttt{meth.getX}, \\
 & & & \quad \texttt{setX} = \texttt{meth.setX}\ \rangle
\end{array}
$$

FIG. 4. Declarations for the `Point` class with modifications to support closure-style private fields.

no appropriate place: no base class constructor function will be called for derived classes, and a premethod function may be called without creating an object.

Enforced Access Restrictions. The record-of-premethods approach succeeds in hiding private instance variables, with little additional language complexity. It cannot handle private methods, unfortunately, as there is no way to install them into objects at instantiation time. There is a slightly awkward workaround that can be used if needed, however. Specifically, private functions that expect a "self" parameter may be allocated and explicitly passed the self object at run-time to mimic private methods.

Explicit Type 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 2.4.2, this restriction may be useful for optimizing operations on objects, allowing access to argument objects in binary methods, and guaranteeing semantic consistency beyond type considerations (some discussion of these points appears in [30]). A special case of this capability is the ability to define *final* classes, as recognized in work on Rapide [30] 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 of another type τ can be used as an object of type τ.

Automatic Propagation of Base Class Changes. Because derived class constructors must explicitly name the methods that they wish to inherit, the record-of-premethods approach does not automatically propagate base class method 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 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 seems that several important and desirable features are lost.

6. Classes = Prototypes + 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 initialize properly any inherited components. It also guarantees that the designers of the `Point` class have the opportunity to update any bookkeeping information they may be keeping about instantiations of `Point` objects. 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 introduce a refined form of existential type to provide the necessary data abstraction. We then combine the extensible objects from Section 4.3 with this abstraction mechanism to produce the desired class construct.

6.1. Data Abstraction

To provide the encapsulation needed for classes, we introduce a fairly standard form of bounded existential quantification [14, 39], adapted to rows instead of types. Bounded existential types (rows) are similar to the existential types we described in Section 4.5, except that they provide a supertype (superrow) bound for the hidden type (row), revealing partial information about it. In more detail, a declaration of the form

$$\texttt{Abstype}\ p <:_w P :: M\ \texttt{with}\ Ops\ \texttt{is}\ Impl\ \texttt{in}\ Client$$

introduces a new row name, p, with upper bound P and kind M. 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. When we encode our running `Point` example, P will have the form $\langle\!\langle \texttt{getX} : int,\ \texttt{setX} : int \rightarrow unit \rangle\!\rangle$, indicating that `Point` objects have public `getX` and `setX` methods. Kind M lists the method names that derived classes may add to p. In our example, M will be $\{\texttt{c}, \texttt{getC}, \texttt{setC}\}$. 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. When this construct is used, parts of *Impl* are bound to p and Ops in *Client*, allowing *Client* to use *Impl* as specified by P, M, and the type annotations in 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

$$\{\!\!\{ p <:_w P :: M = P_{priv},\ ConsImp \}\!\!\}$$

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 M describes how the class may be extended via inheritance. Thus P and M 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.

6.2. Example Classes

Using extensible objects and the abstract datatype mechanism from Section 6.1, 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. The class interpretation presented in the next section provides enough machinery to allow 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 = Prototypes + Data Abstraction" model without support for method specialization. Recent work establishes that this target calculus is type sound [8]. The most relevant published soundness proof for an imperative version of the underlying object calculus appears in [18], which also includes support for concurrency.

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

where

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

and

$$
\begin{array}{lcl}
\texttt{Imp}_p & \stackrel{def}{=} & \lambda(\texttt{xi})\langle\ \ \texttt{x} = \texttt{ref xi}; \\
& & \quad\quad \texttt{getX} = \lambda(\texttt{self})\ !(\texttt{self.x}); \\
& & \quad\quad \texttt{setX} = \lambda(\texttt{self})\ \lambda(\texttt{newX})\ \texttt{self.x} := \texttt{newX}\rangle;
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{Imp}_{cp} & \stackrel{def}{=} & \lambda(\texttt{xi})\lambda(\texttt{ci}) \\
& & \quad \langle\texttt{newPoint xi} \longleftrightarrow\ \ \texttt{c} = \texttt{ref ci}; \\
& & \quad\quad\quad \longleftrightarrow \texttt{getC} = \lambda(\texttt{self})\ !(\texttt{self.c}); \\
& & \quad\quad\quad \longleftrightarrow \texttt{setC} = \lambda(\texttt{self})\ \lambda(\texttt{newC}) \\
& & \quad\quad\quad\quad\quad \texttt{self.c} := \texttt{newC}\rangle;
\end{array}
$$

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

The two abstract type declarations in Figure 5 define the `Point` and `ColorPoint` classes, in that order. The `Point Abstype` expression 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 $\{\texttt{c}, \texttt{getC}, \texttt{setC}\}$ 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 **pro.**p. The constructor returns a **pro** type 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* after [14] 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 $\texttt{Imp}_p$, defined towards the bottom of Figure 5, gives the implementation of the constructor `newPoint`. The type system insures that $\texttt{Imp}_p$'s type is **pro.**p with p replaced by `Point`'s private interface, P_{priv}. The `Abstype` construct binds row variable p to P_{priv} and variable `newPoint` to $\texttt{Imp}_p$ within the scope of its `in` clause.

The structure of the `ColorPoint Abstype` declaration is similar. There are several aspects of this declaration that are worth mentioning however. In particular, notice that the `ColorPoint` constructor, implemented via the $\texttt{Imp}_{cp}$ 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 (**obj.**cp) would not be a subtype of `Point`'s object type (**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.

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 [23].

6.3. *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: (i), an object system that supports inheritance and message sending and (ii), 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 non-trivial 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 explicitly list `Point` as its parent by including p in CP_{pub}, the public interface for `ColorPoint` objects. Additionally, when we combine encapsulation 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. 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 = Prototypes + 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 [13, 28]. 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. A variant of the kinding system used here that tracks method *presences* instead of absences relieves parent classes of the need to make such guesses. Instead of listing the methods that descendants may add, a class lists the methods it uses. This technique has the drawback, reflected in existing languages like C++, of exposing private names. This weakness is much less significant than requiring parents to guess what their descendants will add, however. Recent work shows that kinding systems based on method presences are sound [8]. We believe that some form of method-renaming and α-binding for method names may solve the problem of leaking private names. Work in this area is in progress.

A problem that we do know how to solve 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 $\lambda t.\langle\!\langle \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_k} : \tau_k \rangle\!\rangle$ or $\lambda t.\langle\!\langle R \,|\, \mathtt{m_1} : \tau_1, \ldots, \mathtt{m_k} : \tau_k \rangle\!\rangle$, where R is any row.

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 $T^+ \to (M; \emptyset)$, 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 [21]. 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 [13, 28]. We leave these tasks to future work.

8. Other Approaches

In this section, we briefly outline other approaches people have taken to modeling classes.

8.1. Existential Model

In [42], 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 [41].

In [29], 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

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

where

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

and

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

$$
\begin{array}{lll}
\texttt{Imp}_{cp} & \stackrel{def}{=} & \lambda(\texttt{xi})\lambda(\texttt{ci}) \\
& & \quad \langle\,\texttt{newPoint xi} \leftarrow\!\!+\ \ \texttt{c} = \lambda(\texttt{self})\ \texttt{red}; \\
& & \quad\quad \leftarrow\!\!+\ \texttt{getC} = \lambda(\texttt{self})\ \texttt{self} \Leftarrow \texttt{c}; \\
& & \quad\quad \leftarrow\!\!+\ \texttt{setC} = \lambda(\texttt{self})\ \lambda(\texttt{newC}) \\
& & \quad\quad\quad\quad \texttt{self} \leftarrow \lambda(\texttt{s})\texttt{newC}\rangle \Leftarrow \texttt{setC ic};
\end{array}
$$

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

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 [9], Bruce describes TOOPL, a functional object-oriented language. PolyTOIL, presented in [11], 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. An advantage of this approach, with respect to the one outlined in this paper, is that parent classes do not need to predict what methods their descendants will want to add.

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, [19]. 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. Future Work

As pointed out earlier, there are several problems with our current formulation of classes as a combination of prototypes and data abstraction. The most serious of these problems is the current need for negative information, essentially requiring parent classes to "guess" what methods their descendants will want to add. A variant of the kinding system used here that tracks method presences, instead of absences, solves this problem. Work in this area is in progress, and for a simple object calculus with existential types but not mytype, has been successful [8]. With this modification, parent classes no longer have to guess the names of the methods their descendants may require, but they do have to expose their private names, an undesirable leak of private information. To counter this problem, we intend to add a form alpha-binding for method names to make private names truly private. Another, less significant problem with the current formulation is that derived classes must be declared within the scope of their parents. We believe we can address this problem by replacing our block-structured encapsulation construct with modules and a dot notation, in a fashion similar to that done in [13, 28].

10. 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 [14].

Acknowledgments

We would like to thank Jon Riecke, John Reppy, and the anonymous referees who contributed many useful suggestions for improvement of a draft of this paper.

References

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

[2] 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.

[3] 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.

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

[5] 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.

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

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

[8] V. Bono and K. Fisher. A first-order, extensible-object calculus with support for classes. Unpublished, 1997.

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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

[15] C. Chambers. The Cecil language: Specification and rationale. Available from http://www.cs.washington.edu/research/projects/cecil, December 1995.

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

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

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

[19] 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.

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

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

[22] 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.

[23] 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.

[24] 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.

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

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

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

[28] 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.

[29] M. Hofmann and B.C. 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.

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

[31] 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. A portion of this work appears in ECOOP '97 Proceedings, LNCS 1241.

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

[33] B. Liskov and J. Guttag. *Abstraction and Specification in Software Development.* MIT Press, 1986.

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

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

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

[37] 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.

[38] J.C. Mitchell. *Foundations for Programming Languages.* MIT Press, 1996.

[39] 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.

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

[41] 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.

[42] 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".

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

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

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

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

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

[48] 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.

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

[50] 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.