The design of a class mechanism for MOBY

Kathleen Fisher
AT&T Labs, Research
kfisher@research.att.com

John Reppy
Bell Labs, Lucent Technologies
jhr@research.bell-labs.com

Abstract

Typical class-based languages, such as C++ and JAVA, provide complex class mechanisms but only weak module systems. In fact, classes in these languages incorporate many of the features found in richer module mechanisms. In this paper, we describe an alternative approach to designing a language that has both classes and modules. In our design, we rely on a rich ML-style module system to provide features such as visibility control and parameterization, while providing a minimal class mechanism that includes only those features needed to support inheritance. Programmers can then use the combination of modules and classes to implement the full range of class-based features and idioms. Our approach has the advantage that it provides a full-featured module system (useful in its own right), while keeping the class mechanism quite simple.

We have incorporated this design in MOBY, which is an ML-style language that supports class-based object-oriented programming. In this paper, we describe our design via a series of simple examples, show how various class-based features and idioms are realized in MOBY, compare our design with others, and sketch its formal semantics.

1 Introduction

Modules and classes are important programming mechanisms, both of which help programmers to structure large software systems and to develop reusable libraries. Despite this similarity of purpose, there are significant differences between these mechanisms, and it is desirable to include both in a programming language [Szy92, BPV98]. Currently, programmers must choose between class-based languages like C++ [Str97] and JAVA [GJS96], which provide rich class mechanisms but weak or non-existent module systems, and languages like MODULA-3 [CDG+89] and SML [MTHM97], which have strong module systems but weak or nonexistent class mechanisms. Unfortunately, there is a significant overlap between the features of typical class mechanisms and rich module systems, an overlap that leads to unreasonable complexity if one attempts a naïve combination. The contribution of this paper is the design of a simple class mechanism in the context of a language with a rich module system. Our design has no overlap between the class and module mechanisms, but still provides support for the rich, class-based programming style found in languages like C++ and JAVA.

Designers of such class-based languages have taken a "Swiss-army knife" approach, folding many distinct language features into their classes, including object types, subtyping, inheritance, name-space control, static class members, and scoping. We believe that much of this complexity results from the designers' attempts to compensate for the lack of an independent notion of subtyping and the lack of a full-featured module mechanism. For example, one of the primary uses of multiple inheritance in C++ is to introduce subtyping relationships [Str94]. Other language complications arise because of the lack of a rich module system. Static class members and singleton classes are both examples of classes serving as modules. Similarly, many aspects of the visibility annotations in C++ and JAVA duplicate the visibility control provided by modules.

These examples lead one to ask: what features are intrinsic to classes? We believe that the key feature provided by classes (and not by objects or modules) is *implementation inheritance*. A key aspect of this mechanism is that the binding of method dispatch is left until the time when an object is created, which means that a subclass can override a method defined by its superclass. Classes fill two rôles: they implement objects and they support inheritance. Thus classes have two kinds of clients: *object clients*, which use the class to create new objects, and *class clients*, which use the class as a base for deriving new subclasses. Since there are two kinds of clients, it is natural for a class mechanism to support two distinct views of a class's members, which we call the *object view* and the *class view*. C++ and JAVA make a

To appear in the *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*. Atlanta, Georgia, May 1-4, 1999.

similar distinction between *public* and *protected* class members. Another ramification of inheritance is the need for features that allow base classes to restrict how deriving classes may modify inherited behavior (*e.g.*, *final* methods in JAVA). Such mechanisms are necessary to support the establishment and maintenance of invariants in the presence of inheritance.

The class design that we describe in this paper provides these intrinsic features, but no more, keeping with our philosophy of designing simple independent mechanisms that can be combined to achieve more complex effects. We have adopted this approach in the design of MOBY, an ML-like language that supports class-based object-oriented programming.[1] Specifically, MOBY has three distinct mechanisms to cover the feature set of typical class-based languages:

- *Objects and object types.* Objects are run-time values that support dynamic dispatching. Each object is created by invoking a class *maker* (a.k.a. constructor) with the **new** keyword. An object type describes the interface of its objects, but does not specify their implementations. Subtyping on object types is structural.
- *Classes and class interfaces.* Classes in MOBY provide implementations of objects and support implementation inheritance, with mechanisms for imposing restrictions on subclasses. A class interface is the *type* of a class. In module signatures, class interfaces specify the object and class views of the class as seen outside the module. Thus we can use module signature matching to restrict both the object and class views of a class.
- *Modules and module signatures.* MOBY provides an ML-style module system [Mac84, Ler94, MTHM97]. Such modules structure the name space by grouping definitions, including other modules. We control the external view of the definitions in a module by constraining the module with a signature, which provides data and type abstraction. As MOBY supports subtyping, signatures may specify partially abstract types. We also allow parameterized modules.

In the next section, we briefly review aspects of MOBY necessary to understand our class design. In Section 3 we introduce the major features of our design via some simple examples. Section 4 explores the full range of visibility control that our design provides. In Section 5, we show how various features of typical class-based languages can be realized using combinations of MOBY objects, classes, and modules. We discuss related work in Section 6 and conclude with a summary. We have explored the formal foundations of our design in detail; we outline this work in Appendix B, with a focus on the interaction between classes and modules.

[1] The purpose of MOBY is to serve as a test-bed for experimenting with ideas in language design that might be applied to the design of ML2000. More information about MOBY can be found at `http://www.cs.bell-labs.com/~jhr/moby`.

2 Preliminaries

This section describes features of MOBY that are relevant to understanding the class mechanism described in the next section. In addition to these features, MOBY has constructs common to ML-style languages, including higher-order functions, datatypes, and parametric polymorphism.

2.1 Objects

An *object* in MOBY is a collection of fields (both mutable and immutable) and methods. The fields and methods of an object are called its *members*. There are three operations on objects: select a field, update a field, and invoke a method. An *object type* specifies which object members clients may use. Note that subtyping and member hiding enable an object to have different types in different contexts. Object types are declared and may be recursive, as in the following standard point example:

```
objtype Point {
  meth getX : Unit -> Int
  meth move : Int -> Point
}
```

A `Point` object has no visible fields and two visible methods: `getX`, which takes no arguments and returns an integer, and `move`, which takes an integer and returns a point. The intended semantics is that `getX` returns the point's current position, while `move` shifts the point by some amount and returns it. Because `move` returns the point, we can chain together method invocations. For example, assuming that `pt` is a point, the expression:

```
pt.move(1).getX()
```

moves `pt` by one and then returns its new position.

Continuing the standard example, we might define the type of color point objects as follows:

```
objtype CPoint extends Point with {
  meth getC  : Unit -> Color
  meth shade : Color -> CPoint
}
```

This declaration defines the `CPoint` type to be an object type that includes all of the members of `Point` (with their given types), plus two new methods: `getC` and `shade`. It is shorthand for the following declaration:

```
objtype CPoint {
  meth getX  : Unit -> Int
  meth move  : Int -> Point
  meth getC  : Unit -> Color
  meth shade : Color -> CPoint
}
```

Notice that if `cpt` is a `CPoint` object, the expression

```
cpt.move(1).shade(Red)
```

is not well typed, since `move` returns a `Point` object. While it is possible to design a type system that can specialize the return type of `move` in the definition of `CPoint`, such systems cannot coexist with other features of our design (we discuss these issues in Section 6). In general, the restricted type of `move` is not a problem, since we can write

```
cpt.move(1); cpt.shade(Red)
```

to get the same effect.

2.2 Typing

Although object types in MOBY must be named in **objtype** declarations, subtyping and type equality on object types are *structural*. Structural subtyping is a form of subtyping determined solely by the structure of the types and not by any declared relationships. Structural subtyping for object types has several variants. *Width subtyping* occurs between object types A and B if A has all of B's members and possibly more. *Depth subtyping* allows the types of methods and immutable fields in A to be subtypes of the corresponding members in B [FM95, AC96]. We refer to the combination of width and depth subtyping as *full subtyping*. Soundness considerations prevent depth subtyping on mutable fields [Coo89]. We consider the order of members in object types unimportant and extend subtyping to function and tuple types in the standard way.

An object type in MOBY plays a rôle similar to an *interface* in JAVA, but object types are more flexible because they use structural instead of by-name subtyping. Because MOBY object types specify only the interface of an object and because they support full subtyping, the compiler cannot know an object's layout in general. Consequently, method and field operations are more expensive than in languages that tie object types to implementations. We believe, however, that it is important to have the full flexibility provided by structural subtyping, and so we are willing to accept this cost.

Another consequence of supporting subtyping (as opposed to *row polymorphism* [Rém94]) is that ML-style type inference is not possible. In the current design, MOBY requires explicit argument and return type annotations on function (and method) definitions, but it may be possible to incorporate a technique like *local type inference* [PT98] to reduce the amount of programmer supplied type information.

Most object-oriented languages support some form of mechanism to coerce an object to a subtype (a so-called *downward coercion*). The primary disadvantage of such mechanisms is that they weaken the effectiveness of static typechecking, since these coercions must be checked dynamically. For a language like MOBY that relies on type abstraction to control visibility, there is the additional problem that downward coercions can be used to break abstractions. For this reason, MOBY does not provide a general downward coercion mechanism. Instead, it provides a *hierarchically extensible* datatype mechanism that allows programmers to get the effect of dynamically checked coercions using ML-style pattern matching. This mechanism is taken from OML [RR96b], but has been decoupled from object types. It can also be used to encode binary methods and multimethod dispatch (see [RR96b] for examples).

2.3 Modules

Modules in MOBY are similar to those of other ML-style languages [Mac84, Ler94, MTHM97]. Each module has a *signature*, which controls the visibility of definitions outside the module. Modules may be nested and parameterized by other modules. Since MOBY has subtyping, we allow partial type revelations in signatures, which have the form:

type *type-name* **<:** *type*

For example, the following module collects together the definitions of points and color points from above, but it is constrained by a signature that reveals only that the color point type is a subtype of `Point`:

```
module PointTypes : {
  // the signature of PointTypes
  objtype Point {
      meth getX : Unit -> Int
      meth move : Int -> Point
    }
  type CPoint <: Point
} {
  ... the body of PointTypes ...
}
```

Of course, modules can contain other kinds of definitions besides types, including values, exceptions, classes, and even other modules. Components of modules are referenced by the standard *dot* notation; for example `PointTypes.Point` refers to the point object type defined in the `PointTypes` module.

3 MOBY classes

In this section, we introduce the major features of our class mechanism via a series of simple examples.

3.1 Class declarations

In MOBY, classes describe the implementations of objects. For example, the following class implements points:

```
class PointClass {
  field x : var Int
  public meth getX () : Int { self.x }
  public meth move (dx : Int) : typeof(PointClass)
  {
      self.x := self.getX() + dx;
      self
  }
  public maker point (x0 : Int) {
      field x = x0
  }
}
```

The body of this class contains declarations for the fields, methods, and makers that implement the class (collectively, these are known as the class *members*). `PointClass` has a mutable field named `x` (keyword **var** denotes a mutable field), methods `getX` and `move`, and a maker named `point`. Inside the method implementations, the reserved identifier **self** names the host object.

Makers are special functions used to create objects.[2] Unlike many class-based languages, we do not use the class name for makers, but instead allow programmers to specify maker names. In general, a class may contain any number of makers. The MOBY typing rules require that each maker initialize all the fields defined in its class. In this case, there is only the `x` field. An object client of a class creates an object by using the **new** operator to invoke one of the class's makers with an appropriate argument. For example:

```
new point(0)
```

returns a new `Point` object located at the origin. Note that the maker name `point` is available without qualification because our minimal class mechanism does not define any form of name-space structure — that rôle is left to modules.

Each class member declaration may be annotated with the **public** keyword, which means that it is visible to object clients. The public fields and methods of a class can be viewed as defining an object type; MOBY's **typeof** keyword provides a way to refer to this type. For example, the notation "**typeof**(`PointClass`)" in the above example denotes the object type that is synthesized from the public fields and methods of the `PointClass`. This type is equal to the `Point` type defined in the previous section.

Non-public members (*e.g.*, the `x` field) are visible to class clients, but not to object clients. Such fields and methods are not part of the type of objects created from the class, but are part of the object type assigned to **self** in the implementation of the class. For example, the type of **self** inside the `PointClass` is equal to the type:

```
objtype xPoint extends Point with {
    field x : var Int
}
```

which is a subtype of `Point`. Non-public makers cannot be used to create objects (even in the body of a subclass).

In addition to implementing objects, classes provide a vehicle for code reuse via inheritance. For example, we can implement color points by inheriting from `PointClass`:

```
class CPointClass {
  inherits PointClass
  field c : var Color
  public meth move (dx : Int) : Point {
    super.move(2*dx)
  }
  public meth getC () : Color { self.c }
  public meth shade (dc : Color) : CPoint {
    self.c := self.getC().blend(dc);
    self
  }
  public maker cpoint (x0 : Int, c0 : Color) {
    super point(x0);
    field c = c0
  }
}
```

In this case, we call `PointClass` the *superclass* (or *base* class), and `CPointClass` the *subclass* (or *derived* class). There are three things to note about the `CPointClass` code. First, the **inherits** clause specifies that the fields and methods of `PointClass` are inherited by `CPointClass`. Second, we *override* the `move` method to make color points speedy. The use of the reserved identifier **super** in the body of the `move` method statically binds the call of `move` to the `PointClass` implementation. Third, the `cpoint` maker invokes the `point` maker of its parent class before initializing the `c` field. Since non-public makers are visible to subclasses, they may be used for superclass initialization.

3.2 Class interfaces

An *interface* is the type (or signature) of a class; it is used when specifying a class in a module's signature.[3] For example, the following module encapsulates the implementation of `PointClass`:

```
module Pt : {
  class PointClass : {
    public meth getX : Unit -> Int
    public meth move : Int -> Point
    public maker point of Int
  }
} { ... implementation of PointClass ... }
```

In this example, the `Pt` module is constrained by a signature that specifies an interface for `PointClass`. This interface indicates that `PointClass` has public methods `getX` and `move`, and a public maker `point` that takes an integer argument. Matching the `PointClass` with this interface has the effect of hiding the `x` field (*i.e.*, `x` is visible to deriving classes inside the `Pt` module but not outside of it). Once a field or method is hidden in this way, subclasses are free to define new members with the same name. Note, however, that the original member is still accessible from the superclass's methods (*e.g.*, `getX`), even if its name is reused. We discuss the technical implications of such private names in Section B.

A class interface defines both the object and class views of a class as seen outside the module. In the above example, these views are identical, but often the class view will contain more members. For example, we might encapsulate the `CPointClass` as follows:

```
module CPt : {
  class CPointClass : {
    public meth getX  : Unit -> Int
    public meth move  : Int -> Point
    public final meth getC  : Unit -> Color
    public meth shade : Color -> CPoint
    public maker cpoint of (Int * Color)
    field c : var Color
  }
} { ... implementation of CPointClass ... }
```

In this example, we make the `c` field visible to class clients that live outside the `CPt` module by including the field in the interface but omitting the **public** label. We have also

[2] We use the term *maker* (borrowed from Theta [Pro95]), instead of the more standard *constructor*, to avoid confusion with data and type constructors.

[3] MOBY's notion of an interface should not be confused with that of JAVA. Interfaces in JAVA play a rôle that is related more closely to MOBY's object types.

added the restriction that the `getC` method cannot be overridden by subclasses by annotating it with the **final** keyword. As will be seen below, MOBY also has an **abstract** annotation for methods that *must* be overridden by a subclass.

In general, a class interface may differ from the class it specifies in the following ways:

- It may omit members that are in the implementation (*e.g.*, the `x` field in `Pt.PointClass`).
- It may omit the **public** annotation from a member that is public in the implementation.
- It may add a **final** or **abstract** annotation to a method (*e.g.*, the `getC` method in `CPt.ColorPointClass`).
- It may specify a more specific type for the argument of a maker function.

An important property of our design is that type checking a class definition depends only on the class interface of its direct superclass. Consequently, the internal class hierarchy and private class members of a class library may be changed without requiring any changes in the source of clients. This property fits with the goal of supporting separate compilation for modules based only on the interfaces of the module's antecedents [Ler94].

3.3 Object construction and invariants

An important feature of class-based languages is support for the establishment and maintenance of object and class-level invariants.[4] These guarantees are especially important to base-class implementors who want to restrict how their classes may be extended. Providing such support motivates several features of the MOBY class mechanism design.

Object-level invariants are established when an object is initialized by its maker. We require that all fields defined in a class be initialized by each maker in the class and that subclasses always invoke a superclass maker. These requirements guarantee that fields are always initialized before an object is used. The author of a class can ensure that its invariants are maintained by hiding the mutable state from subclasses (*i.e.*, making the state private) and by declaring methods to be **final**.

Unlike many class-based languages, we do not allow access to an object's methods (via **self**) inside a maker. This restriction avoids the complication of a superclass maker invoking a subclass method before the subclass's fields have been initialized.[5] The main disadvantage of not being able to reference **self** inside makers is that class-level invariants cannot be maintained when new objects are created, since they require access to the new object. MOBY allows classes to include an **initially** clause, which specifies an expression to be evaluated each time an object is created. This expression may refer to **self**. The **initially** clauses are executed in the same order as makers — superclass and then subclass. Thus, the act of creating a new object involves first computing the initial values of its fields by invoking maker functions, then creating the object and passing it to the **initially** clauses, and finally returning the new object to the context that invoked **new**. Appendix A gives an example that uses this feature.

3.4 Advanced features

The last example of this section is an idealized graphics application that illustrates the interactions between classes and modules. This example uses objects to represent the graphical elements of a picture, called *glyphs*. We start by defining a module signature that specifies a base class for implementing glyphs.

```
signature BASE_GLYPH {
  class Glyph : {
    public final meth draw : Point -> Unit
    abstract meth drawGlyph : Drawable -> Unit
    maker mk of Unit
  }
}
```

where the type `Drawable` is the type of an object that represents a drawing surface. Note that the `drawGlyph` method in the `BaseGlyph` class is marked as **abstract**. This annotation means that `BaseGlyph` does not provide an implementation of `drawGlyph`; its subclasses must provide the implementation. A class that contains an **abstract** method is known as an *abstract* base class. Instantiating objects from such classes is nonsensical. Consequently, we do not permit abstract base classes to have makers in their object views. Also, the class interface for `BaseGlyph` marks the `draw` method as **final**, which means that derived classes cannot override or hide it. Although no implementation is given here, the intuition is that the `draw` method sets up any conditions necessary for drawing, perhaps translates the relevant coordinate system to the origin, *etc.*, and then invokes the `drawGlyph` method provided by the derived class to do the actual drawing. This design illustrates *method factoring*: drawing code common to all glyphs is *factored* into the `draw` method supplied by the base class.

Since we might want to support different kinds of drawables (*e.g.*, computer screens and postscript), we collect the subclasses of `BaseGlyph` into a module that is parameterized by the `BASE_GLYPH` signature:

```
module Glyphs (G : BASE_GLYPH) {
  class LineGlyph {
    inherits G.BaseGlyph
    field p1 : Point
    field p2 : Point
    meth drawGlyph (d : Drawable) : Unit {
      d.drawLine(p1, p2)
    }
```

[4] Object-level invariants are properties of the state of a given object, whereas class-level invariants are properties of a collection of objects.

[5] C++ addresses this problem by changing the semantics of method dispatch inside constructors, while JAVA relies on the property that all fields are initialized to some type-specific value prior to executing the constructor.

```
    maker line (p1 : Point, p2 : Point) {
      super mk();
      field p1 = p1; field p2 = p2
    }
  }
  ... implementations of other glyph classes ...
}
```

In this code fragment, LineGlyph is a subclass of G.BaseGlyph and provides the implementation of the drawGlyph method for drawing lines. This example illustrates both module and class-based code reuse. We use a parameterized module to abstract over the base class of LineGlyph, which allows multiple class hierarchies to be defined by applying Glyphs to different base classes; we use inheritance to factor out code common to all glyphs.

4 Visibility Control

As we have described, classes have two views: object and class, while modules define two scopes: internal and external. By enclosing a class inside a module, we can produce four distinct *visibility contexts*: *internal-class view*, *internal-object view*, *external-class view*, and *external-object view*. Further refinement of these contexts may be achieved by nesting modules. In this section, we construct two synthetic examples to demonstrate many of these contexts. Because of the flexibility afforded by the combination of classes and modules, these examples are necessarily somewhat complicated. It is important to understand that this complexity arises from the combination of orthogonal features, the formal description of which can be captured in a single definition (see Definition 3 in Appendix B).

In these examples, we assume that we have two types T1 and T2, with T2 a subtype of T1, and aT1 a value of type T1. The following code fragment defines the module M1:

```
module M1 : {
  class C1 : {
    public meth m1 : Int -> Int
           meth m2 : Int -> Int
    public maker mk1 of T2
           maker mk2 of T2
  }
  val obj : typeof(C1)
} {
  class C1 {
    public meth m1 (x : Int) : Int { ... }
           meth m2 (x : Int) : Int { ... }
    public meth m3 (x : Int) : Int { ... }
           meth m4 (x : Int) : Int { ... }

    public maker mk1 (t1 : T1) { ... }
           maker mk2 (t1 : T1) { ... }
    public maker mk3 (t1 : T1) { ... }
           maker mk4 (t1 : T1) { ... }
  }
  val obj = new mk3(aT1);
  ...
}
```

The module M1 contains a declaration of the class C1, which provides methods m1, m2, m3, and m4 and makers mk1, mk2, mk3 and mk4. The module also contains a binding of obj to an object created from C1. The signature for module M1 reveals that class C1 is in the module, but hides aspects of C1 by providing a limited class interface. The signature exposes the value obj, but reveals only that obj has (external) type **typeof**(C1), a type which contains only method m1.

Figure 1 summarizes the visibility of C1's members in the four visibility contexts defined by the example. In each table, the "Internal" and "External" rows represent views of the method from inside and outside M1, respectively, while the "Class view" and "Object view" columns denote the class and object views. Each entry in a table gives the type of the member in that context; the entry "Not visible" indicates that the member is not accessible.

There are a few points about this table that deserve further discussion. The method m1 serves as a base-line; it is visible everywhere with type Int -> Int. The methods m2, m3, and m4 illustrate the kinds of visibility control found in JAVA (and to some extent, C++). The m2 method is analogous to a *protected* method in C++; it is visible to class clients, but not to object clients. The m3 method has JAVA's default (or package) visibility; it is visible to both object and class clients inside M1, but hidden from both outside the module. Lastly, the m4 method is close to a *private* method; it is visible to class clients inside M1, but completely hidden from all clients outside the module.

This example also illustrates that we have control over the visibility of maker functions. The maker mk1 takes an argument of type T1, which accounts for its type in the internal views. Outside the module M1, however, it has an interface that requires an argument of type T2. Note that since we are talking about the *arguments* to a maker, the subtyping relationship is contravariant. Maker mk2 illustrates that we can prevent makers from being used by object clients, while mk3 shows that we can give makers package scope. Finally, maker mk4 can only be used by class clients within the module.

Module M1 shows how the combination of module signature matching and our class mechanism provides fine-grained control over class-member visibility. The following code fragment defines a module M2 that includes a subclass of C1:

```
module M2 : {
    ...
} {
  class C2 {
    inherits M1.C1
           meth m1 : Int -> Int { ... }
    public meth m2 : Int -> Int { ... }
           meth m3 (s : String) : String { ... }
    public maker mk () { ... }
  }
}
```

This example illustrates that the deriving class (C2) is allowed full control over the visibility of the methods that it inherits from C1. In particular, C2 hides the "public" m1 method it inherited, while revealing the "protected" m2 method. In addition, class C2 adds a new method named m3. This method has no connection to the m3 method defined in

`m1`	Class view	Object view
Internal	`Int -> Int`	`Int -> Int`
External	`Int -> Int`	`Int -> Int`

`m2`	Class view	Object view
Internal	`Int -> Int`	Not visible
External	`Int -> Int`	Not visible

`m3`	Class view	Object view
Internal	`Int -> Int`	`Int -> Int`
External	Not visible	Not visible

`m4`	Class view	Object view
Internal	`Int -> Int`	Not visible
External	Not visible	Not visible

`mk1`	Class view	Object view
Internal	**of** `T1`	**of** `T1`
External	**of** `T2`	**of** `T2`

`mk2`	Class view	Object view
Internal	**of** `T1`	Not visible
External	**of** `T2`	Not visible

`mk3`	Class view	Object view
Internal	**of** `T1`	**of** `T1`
External	Not visible	Not visible

`mk4`	Class view	Object view
Internal	**of** `T1`	Not visible
External	Not visible	Not visible

Figure 1: Visibility of class `C1`'s members

`C1`, which was not made visible outside the module `M1`. This example also shows that the type of objects implemented by a subclass is independent of the type of objects implemented by its superclass. In particular, the type **`typeof`**(`C2`) is neither a subtype nor a supertype of **`typeof`**(`C1`). This independence exemplifies the slogan that "*inheritance is not subtyping*" [CHC90]. Note, however, that the subclass's class-view type is always a subtype of the superclass's class-view type (*i.e.*, the type of **`self`** is a subtype of the type of **`super`**). This typing relationship is necessary for the type soundness of inheritance.

5 Class-based programming

We have shown how the combination of classes and modules in MOBY provides the full power of the visibility annotations found in C++ and JAVA. In this section, we further demonstrate that this combination provides an expressive class-based programming model by describing how common class-based features and idioms may be realized in MOBY.

- *Static class members.* MOBY relies on the module system to support global definitions.
- *C++ friends and* JAVA*'s package scope.* Objects provide data abstraction, but it is often useful to break that abstraction in controlled ways. For this purpose, C++ allows a class to declare that other classes and functions may access its internal representation. Likewise, JAVA has the notion of *package* scope: classes defined inside the same package have access to each other's protected members. We can use the combination of modules and partial type revelations to implement friends in a way that is equivalent to JAVA's package scope [PT93, KLM94, RR96b]. An example of this technique can be found in Appendix A.
- *C++'s private inheritance.* In C++, a class can inherit the implementation of a superclass without inheriting its interface. MOBY's inheritance mechanism provides this functionality as well, as the object type implemented by a subclass is independent of the object type implemented by its superclass.
- JAVA*'s interfaces and C++'s multiple inheritance with abstract base classes.* The main use of multiple inheritance in C++ is to provide a richer subtyping hierarchy; JAVA instead introduced *interfaces*, which have subtyping relationships that are independent of the class hierarchy. MOBY provides such subtyping relations via structural subtyping rules.
- *Final classes.* A final class is one that cannot be extended; it can be used only for creating objects. Because MOBY object types are independent of classes, we can achieve the effect of a final class by hiding the class in a module and exporting functions that create its objects by invoking the appropriate makers.
- *Inner and anonymous classes.* JAVA 1.1 [AG98] allows class definitions to be nested inside other classes. In keeping with their Swiss-army knife tradition, JAVA has three different flavors of inner class, each of which corresponds to a different mechanism in MOBY. Anonymous inner classes play the rôle of first-class functions, which MOBY already has. Static inner classes allow the enclosing class to be used to collect together several class definitions; modules fill this rôle in MOBY. To get the effect of non-static named inner classes, MOBY can use top-level classes that take their

logically enclosing object as an extra parameter to their makers. We can then group the "inner" and "outer" classes together in a module to control the visibility of class members. While the result of this transformation may be complicated, it mirrors the underlying complexity of the inner-class mechanism.[6] We do not believe that this mechanism is useful enough to warrant supporting it directly in the language.

6 Related work

6.1 Encoding classes

A number of researchers have explored encoding classes via objects and modules. We believe that these studies provide evidence that a primitive class mechanism is a good idea. Encoding classes using collections of pre-methods, as described in [AC96] and [RR96a], requires substantial programmer overhead. Furthermore, compiling such encodings into efficient object representations with shared method suites is likely to be difficult. Vouillon has proposed unifying classes and modules by extending an ML-like module mechanism with method and inheritance declarations [Vou98].[7] His proposal includes weakening the phase distinction and making modules dynamic. The fact that his encoding of classes uses little of the existing module mechanism and requires extensive changes is evidence that classes and modules are inherently different.

6.2 MyType

Some statically typed object-oriented languages provide a kind of *open recursion* in the type of **`self`** in methods [BSv95, FM95, AC96], typically by using a special type variable, called *mytype* (or *selftype*), for the type of **`self`**. For example, we can declare the `move` method in `PointClass` to return *mytype*, which would mean that moving a `CPoint` would return a `CPoint`. In addition to allowing the return type of inherited methods like `move` to be specialized, this mechanism also allows statically typable *binary methods* [BCC$^+$96].

Unfortunately, ensuring the soundness of *mytype* requires sacrificing flexibility in control of class member visibility. Specifically, if a method is public in a class that has *mytype* in its interface, that method cannot be hidden from either class clients or object clients of a deriving class — to do so would render the use of *mytype* unsound. In MOBY, we have chosen not to provide this form of open recursion in the type of **`self`** and instead provide complete control over class-member visibility.

[6] The JAVA Virtual Machine was not modified to support inner classes, so JAVA compilers must do a similar transformation [Jav97].

[7] Stone and Riecke explored a similar design in the summer of 1997; discussions with them at the time led us to develop our design.

6.3 Other language designs

There are a number of languages that have both modules and classes. Of these, OCAML [RV98, Ler98] is the most closely related to MOBY. OCAML is a dialect of ML that provides both a rich module system and support for class-based, object-oriented programming. Although the module systems of the two languages are essentially equivalent, the languages differ significantly in their object types and class constructs. The main differences stem from a fundamental design choice: OCAML's design emphasizes the expressiveness of the object-type system, whereas MOBY's design emphasizes the expressiveness of the class system. OCAML uses *row polymorphism* [Rém94] to support extensible object types, whereas MOBY uses structural subtyping. OCAML's approach has two advantages: it is backward compatible with ML's Hindley-Milner type inference, and one can get the effect of *mytype* by the use of recursive type variables. The cost of providing *mytype* is that OCAML's classes are less flexible than MOBY's. Specifically, if a method is public in an OCAML class, it cannot be hidden from either class clients or object clients of a deriving class, as required for the soundness of *mytype*. In addition, if a method is hidden from object clients in a class, it cannot be made public by a deriving class. The former restriction means that the class hierarchy determines an object-type hierarchy. These restrictions, along with the fact that OCAML does not have public fields, limit one's ability to use the module system to implement friends via package scope in two ways. First, it is not possible to reveal inherited protected methods or fields to the friend functions, and any method that a class reveals to its friends must also be revealed to any client outside the module (assuming that the class is exported).

Another design choice is support for *functional* objects: OCAML provides functional update (essentially a *cloning* operation) and *mytype* to support them, whereas MOBY does not. Again, this choice reflects the difference in focus of the two designs. While functional objects provide additional flexibility in the object system, they cause problems for the class system. Specifically, the ability to copy an object via functional update undermines class-level invariants.

MOBY has a number of similarities to the language THETA [Pro95]. Like MOBY, object types and classes are separate notions in THETA. Furthermore, THETA also allows independent control of the object view and class view of a class via *provides* and *hiding* clauses in class definitions. Object construction in THETA involves a top-down pass that can only initialize fields followed by a bottom-up pass that can access the created object. This second pass plays a rôle similar to MOBY's **`initially`** clause, but each maker in a class may specify a different action. The main difference between MOBY and THETA is in the module system: THETA has a weak notion of signatures that specifies the names but not the types of exported components and does not support nested or parameterized modules.

MODULA-3 [CDG+89] and LOOM [BFP97] are two other languages that combine a class construct with a module system. In MODULA-3, both the module system and the support for object-oriented programming are weaker than what we propose. For example, its module system disallows nested modules and its class construct unifies the subtyping and inheritance hierarchies. LOOM is a class-based object-oriented language based on matching, a relation similar to row-polymorphism. Recently, the designers of LOOM proposed a module system for their language that supports an *open* operation in addition to the standard *import* operation [BPV98]. The distinction between importing and opening a module in LOOM is similar to our distinction between the object and class views of a class. A client that imports a module sees only the *public* components of the module as specified in the module interface, whereas a client that opens a module sees the entire contents of the module, which makes the classes hidden within the module available for inheritance. This feature means that LOOM modules do not provide the abstraction that one normally expects from modules, since any client of a module can peek inside by opening it.

7 Conclusions

We have presented the design of a simple class mechanism and have shown that it works in concert with the module system to provide a rich, class-based programming experience. We have also related our proposal to other designs. While space does not permit a formal treatment of MOBY's design here, we outline our work on this topic in Appendix B. We are currently writing a compiler for MOBY, which we hope to release by mid-1999.

Our design philosophy has been to identify the fundamental properties of classes and then to design a mechanism that supports them. While the formal semantics of our class mechanism has provided a guide to the design of the surface language features, we have also used our expectations about programming in MOBY to improve the design. For example, our original design supported both width and depth subtyping between the class and object views of a class (*e.g.*, the type of a method in the class view was allowed to be a subtype of its type in the object view). While this flexibility naturally falls out of the semantics, supporting it in MOBY required cumbersome syntax, and we were unable to find any motivating examples for such flexibility. Therefore, we decided that a more natural syntax was more important than the more natural semantics, and so we removed the feature.

Table 1 compares the way that JAVA and MOBY use their package/module and class mechanisms to support various language features (a similar comparison might be made with C++'s namespaces and classes). This table illustrates three important points:

- Comparing JAVA's class column with MOBY's module column shows that there is significant overlap between traditional class mechanisms and rich module systems. Perhaps this overlap explains why typical class-based languages tend to have weak or non-existent module systems.
- Comparing the two MOBY columns shows that there is *no* overlap between MOBY's module and class mechanisms.
- Lastly, we note that there are several features provided by the MOBY module mechanism that are missing from JAVA: type abstraction, partial type abstraction, and signature matching.

While it is clear that our class mechanism is substantially simpler than JAVA's, one might argue that the total complexity of our object, class, and module systems outweighs the total complexity of JAVA's interface, class, and package mechanisms. Such comparisons are hard to make, and we are not convinced that MOBY is more complicated when viewed as a whole. But even so, such an argument overlooks the fact that a rich module system, like MOBY's, is valuable in its own right as a mechanism for supporting large-scale software development and reusable libraries. We expect that MOBY's combination of a rich module system with support for traditional class-based programming will enable new and interesting programming techniques.

Acknowledgments

Work on the foundations and design of ML2000 provide the basis for our work on MOBY, and discussions with the group about the design have been helpful. Bob Harper and David MacQueen have provided guidance on the semantics of modules. David MacQueen and Jon Riecke have also been involved in the MOBY design effort. Comments from Mary Fernández, David MacQueen, Greg Morrisett, Didier Rémy, Jon Riecke, and Phil Wadler on drafts of this paper helped improve the presentation.

References

[AC93] Amadio, R. M. and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, **15**(4), September 1993, pp. 575–631.

[AC96] Abadi, M. and L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.

[AG98] Arnold, K. and J. Gosling. *The Java Programming Language*. Addison-Wesley, Reading, MA, 2nd edition, 1998.

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

[BFP97] Bruce, K. B., A. Fiech, and L. Petersen. Subtyping is not a good "match" for object-oriented languages. In *ECOOP'97*, vol. 1241 of *LNCS*, New York, NY, 1997. Springer-Verlag, pp. 104–127.

Table 1: A comparison of JAVA and MOBY

Feature	JAVA Packages	JAVA Classes	MOBY Modules	MOBY Classes
Friends via package scope	√		√	
Protected scope		√		√
Private scope		√	√	
Name spaces — providing a hierarchical structure to the name space.	√	√	√	
Signature matching — constraining the exports of a module.			√	
Data abstraction — hiding state.		√	√	
Type abstraction — revealing the name of a type, but hiding its representation.			√	
Partial type abstraction — revealing the name of a type and a super-type bound, but not its representation.			√	
Parameterization — parameterizing definitions by definitions.			√	
Import	√		√	
Implementation inheritance		√		√
Value/type definitions		√	√	
Method/field definitions		√		√
Object construction		√		√

[BPV98] Bruce, K. B., L. Petersen, and J. Vanderwaart. Modules in LOOM: Classes are not enough. Available from `http://www.cs.williams.edu/~kim`, April 1998.

[BSv95] Bruce, K., A. Schuett, and R. van Gent. PolyTOIL: A type-safe polymorphic object-oriented language. In *ECOOP'95*, vol. 952 of *LNCS*, New York, NY, 1995. Springer-Verlag, pp. 26–51.

[CDG+89] Cardelli, L., J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). *Technical Report 52*, Digital System Research Center, Palo Alto, CA, November 1989.

[CHC90] Cook, W. R., W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *POPL'90*, January 1990, pp. 125–135.

[Coo89] Cook, W. R. A proposal for making Eiffel type-safe. In *ECOOP'89*, 1989, pp. 57–72.

[Fis96] Fisher, K. *Type Systems for Object-oriented Programming Langauges*. Ph.D. dissertation, Stanford University, August 1996.

[FM95] Fisher, K. and J. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, **1**(3), 1995, pp. 189–220.

[FR99] Fisher, K. and J. Reppy. Foundations for MOBY classes. *Technical Memorandum*, Bell Labs, Lucent Technologies, Murray Hill, NJ, February 1999.

[GJS96] Gosling, J., B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[HL94] Harper, R. and M. Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *POPL'94*, January 1994, pp. 123–137.

[HS97] Harper, R. and C. Stone. An interpretation of Standard ML in type theory. *Technical Report CMU-CS-97-147*, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, June 1997.

[Jav97] Inner classes specification, February 1997. Available from `http://java.sun.com/products/JDK/1.1`.

[KLM94] Katiyar, D., D. Luckham, and J. Mitchell. A type system for prototyping languages. In *POPL'94*, January 1994, pp. 138–161.

[Ler94] Leroy, X. Manifest types, modules, and separate compilation. In *POPL'94*, January 1994, pp. 109–122.

[Ler96] Leroy, X. A syntactic theory of type generativity and sharing. *JFP*, **6**(5), September 1996, pp. 1–32.

[Ler98] Leroy, X. *The Objective Caml System (release 2.00)*, August 1998. Available from `http://pauillac.inria.fr/caml`.

[Mac84] MacQueen, D. Modules for Standard ML. In *LFP'84*, August 1984, pp. 198–207.

[MTHM97] Milner, R., M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997.

[Pro95] Programming Methodology Group, MIT LCS, Cambridge, MA. *Theta Reference Manual (Preliminary Version)*, February 1995. Available from `http://www.pmg.lcs.mit.edu/Theta.html`.

[PT93] Pierce, B. C. 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.

[PT98] Pierce, B. C. and D. N. Turner. Local type inference. In *POPL'98*, January 1998, pp. 252–265.

[Rém94] Rémy, D. Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell (eds.), *Theoretical Aspects of Object-oriented Programming*, pp. 67–95. The MIT Press, Cambridge, MA, 1994.

[RR96a] Reppy, J. H. and J. G. Riecke. Classes in Object ML via modules. In *FOOL3*, July 1996.

[RR96b] Reppy, J. H. and J. G. Riecke. Simple objects for SML. In *PLDI'96*, May 1996, pp. 171–180.

[RS98] Riecke, J. G. and C. Stone. Privacy via subsumption. In *FOOL5*, January 1998. A longer version will appear in TAPOS.

[RV98] Rémy, D. and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *TAPOS*, **4**, 1998, pp. 27–50.

[Str94] Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley, Reading, MA, 1994.

[Str97] Stroustrup, B. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 3rd edition, 1997.

[Szy92] Szyperski, C. A. Import is not inheritance; why we need both: Modules and classes. In *ECOOP'92*, 1992, pp. 19–32.

[Vou98] Vouillon, J. Using modules as classes. In *FOOL5*, January 1998.

A Example: A graph module

This example shows an implementation of undirected graphs in MOBY. It illustrates the use of partial type abstraction to implement friends and the use of an **initially** clause. The interface is described in the following signature:

```
signature GRAPH {
  // public operations on nodes
    objtype NodeOps {
      field name : String
      meth listEdges : Unit -> List(Edge)
    }
  // public operations on edges
    objtype EdgeOps {
      meth nodes : Unit -> (Node * Node)
    }
  // the actual node and edge types are richer
    type Node <: NodeOps
    type Edge <: EdgeOps
  // functions to create nodes and edges
    val mkNode : String -> Node
    val mkEdge : (Node * Node) -> Edge
}
```

Note that the Node and Edge types are partially abstract; clients of this signature can use the NodeOps and EdgeOps interfaces, but do not have access to their full representations. The implementation follows:

```
module Graph : GRAPH {
  // public operations on nodes
  objtype NodeOps {
      field name : String
      meth listEdges : Unit -> List(Edge)
    }
  // public operations on edges
  objtype EdgeOps {
      meth nodes : Unit -> (Node * Node)
    }

  class NodeCls {
    public meth listEdges () : List(Edges) {
      self.edges
    }
    public meth addEdge (e : Edge) : Unit {
      self.edges := e :: self.edges
    }
    public maker node (s : String) {
      field edges = Nil; field name = s
    }
  }

  class EdgeCls {
    public meth nodes () : Unit {
      (self.n1, self.n2)
    }
    public maker edge (n1 : Node, n2 : Node) {
      field n1 = n1; field n2 = n2
    }
    // Once an edge is created, we need to add
    // it to its node's edge lists.
    initially {
      self.n1.addEdge(self);
      self.n2.addEdge(self)
    }
  }

  type Node = typeof(NodeCls)
  type Edge = typeof(EdgeCls)

  fun mkNode (s : String) : Node { new node(s) }
  fun mkEdge (n1 : Node, n2 : Node) : Edge {
    new edge(n1, n2)
  }

} // end Graph
```

Note that the **initially** clause in EdgeCls uses the addEdge method of the Node object type, and that this method is not available outside the Graph module. This example illustrates the use of *package* scope to allow friends access to private members. The classes in this example are also *final*, since they are not exported from the Graph module and thus cannot be further extended.

B MINIMOBY

In this appendix, we give an overview of the formal semantics of a substantial subset of MOBY, called MINIMOBY, which includes almost all of the features of MOBY's object, class, and module mechanisms. Following the style of Harper and Stone's SML semantics [HS97], we define the semantics of MINIMOBY via a translation into an internal language (IL).

There are two important technical issues that our IL addresses: the class/object distinction and the handling of private names. We address the former by adopting Fisher's technique of defining two kinds of objects [Fis96]: *extensible objects*, which are used to model classes, and *fixed objects*, which are used to model object values. The extensible objects support method override and extension but not dispatch, while fixed objects support method dispatch but not override or extension. We address the issues raised by private members by using Riecke and Stone's technique of adding dictionaries to objects to map method names to internal slot numbers [RS98]. Members are hidden by removing their name from the domain of the dictionary. A more detailed description of our IL, including subject reduction and soundness results, can be found in a technical report [FR99].

In the remainder of this appendix, we focus on the interaction between modules and classes. A number of different approaches to specifying the semantics of ML-style modules exist [MTHM97, Ler96, HL94, HS97]. The key technical issue that shapes these semantics is accounting for type identity and abstraction. For the discussion here, however, we ignore these issues and focus on the formal characterization of *class interfaces*.

One of the key ideas in the semantics of ML-style modules is the notion of *enrichment*, which models the mechanism of thinning a module's interface via signature matching. Informally, we say that a signature *enriches* another (written $\succ$) if the first has more components with richer structure than the second.[8] It is precisely in the extension of enrichment to cover class interfaces that the static semantics of classes and modules interact. Although the full definition of signature matching involves the *instantiation* of bound types, type instantiation has no impact on the semantics of classes, and we omit further discussion of it.

B.1 Semantic objects

Before discussing the semantics, we need a brief digression to describe some of the semantic objects we use. We assume the existence of the usual kinds of identifiers, including object-field IDs (FIELDID), object-method IDs (METHID), and maker IDs (MKID). We use LABID = FIELDID $\cup$ METHID to denote the set of object label IDs. Figure 2 lists the definitions that are related to typing classes. The set of types (TYPE) includes base types, type names (used to represent abstract types), function types, and recursive object types. A proto-object type (PROTOOBJ) is a finite map from labels to a pair of an object member kind and its type. We use these maps to represent the object and class-view types when typing class bodies and specifications. The notation $\mathrm{Labels}(\theta)$ denotes the domain of θ and the notation $\mathrm{Methods}(\theta)$ denotes the method labels in the domain of θ.

In the discussion below, we use subtyping judgements of the form $\Gamma \vdash \tau <: \tau'$, where Γ is an *environment*. The rules for these judgements are the standard ones [FM95]; the most important detail is that we use the so-called "Amber rule" for subtyping recursive object types [AC93]. We also use width subtyping judgements ($\Gamma \vdash \tau <:_w \tau'$). The rule for width subtyping of recursive object types is simpler than the rule for full object subtyping, since the types of common members must be equal. We extend the subtyping relationships to proto-object types in the obvious way. Full details can be found in [FR99].

[8] Technically, this relationship is defined between environments, which are the semantic objects that represent the "types" of modules.

B.2 Class interfaces

The most important concept in the semantics of MINIMOBY classes is the formalization of class interfaces. We represent class interfaces in our static semantics by a 4-tuple of the form $\mathcal{I} = (\theta_{\mathcal{I}}, \mathcal{K}_{\mathcal{I}}, \mathcal{P}_{\mathcal{I}}, \mathcal{A}_{\mathcal{I}})$. Intuitively, $\theta_{\mathcal{I}}$ specifies the class-view type of objects created from the class, while $\mathcal{K}_{\mathcal{I}}$ describes the class-view of the maker functions (it maps maker IDs to their domain types). Restricting these two finite maps by the set of public members ($\mathcal{P}_{\mathcal{I}}$) defines the object-view of the class. Lastly, the finite map $\mathcal{A}_{\mathcal{I}}$ associates **`final`** and **`abstract`** annotations with some of the class's methods.

Using this formalization of class interfaces, we may specify the technical conditions under which one class interface enriches another. We start by defining the notion of a *well-formed* class interface.

Definition 1 A class interface $(\theta_{\mathcal{I}}, \mathcal{K}_{\mathcal{I}}, \mathcal{P}_{\mathcal{I}}, \mathcal{A}_{\mathcal{I}})$ is *well-formed* with respect to an environment Γ if the types in $\theta_{\mathcal{I}}$ and $\mathcal{K}_{\mathcal{I}}$ are well-formed, and if it satisfies the following properties:

- $\mathcal{P}_{\mathcal{I}} \subseteq \mathrm{Labels}(\theta_{\mathcal{I}}) \cup \mathrm{dom}(\mathcal{K}_{\mathcal{I}})$ and
- $\mathrm{dom}(\mathcal{A}_{\mathcal{I}}) \subseteq \mathrm{Methods}(\theta_{\mathcal{I}})$

MOBY's syntax enforces these well-formedness properties.

The most important relation on class interfaces is enrichment.

Definition 2 A well-formed class interface

$$\mathcal{I} = (\theta_{\mathcal{I}}, \mathcal{K}_{\mathcal{I}}, \mathcal{P}_{\mathcal{I}}, \mathcal{A}_{\mathcal{I}})$$

enriches a well-formed interface

$$\mathcal{I}' = (\theta_{\mathcal{I}'}, \mathcal{K}_{\mathcal{I}'}, \mathcal{P}_{\mathcal{I}'}, \mathcal{A}_{\mathcal{I}'})$$

in the environment Γ, written $\Gamma \vdash \mathcal{I} \succ \mathcal{I}'$, if the following properties hold:

- $\Gamma \vdash \theta_{\mathcal{I}} <:_w \theta_{\mathcal{I}'}$,
- $\mathrm{dom}(\mathcal{K}') \subseteq \mathrm{dom}(\mathcal{K})$,
- for all $mk \in \mathrm{dom}(\mathcal{K}')$, $\Gamma \vdash \mathcal{K}'(mk) <: \mathcal{K}(mk)$,
- $\mathcal{P}_{\mathcal{I}'} \subseteq \mathcal{P}_{\mathcal{I}}$,
- $\mathrm{dom}(\mathcal{A}_{\mathcal{I}}) \subseteq \mathrm{dom}(\mathcal{A}_{\mathcal{I}'})$, and
- for all $m \in \mathrm{dom}(\mathcal{A}_{\mathcal{I}})$, $\mathcal{A}_{\mathcal{I}}(m) = \mathcal{A}_{\mathcal{I}'}(m)$.

$$
\begin{array}{rcl}
\tau & \in & \textsc{Type} = \textsc{BaseTy} \cup \textsc{TyName} \cup \textsc{FunTy} \cup \textsc{ObjTy} \\
\theta & \in & \textsc{ProtoObj} = \textsc{LabId} \overset{\text{fin}}{\to} (\{\mathbf{\texttt{meth}}, \mathbf{\texttt{val}}, \mathbf{\texttt{var}}\} \times \textsc{Type}) \\
\mathcal{K} & \in & \textsc{MkEnv} = \textsc{MkId} \overset{\text{fin}}{\to} \textsc{Type} \\
\mathcal{P} & \in & \textsc{PubMembs} = \mathrm{Fin}(\textsc{LabId} \cup \textsc{MkId}) \\
\mathcal{A} & \in & \textsc{MethAn} = \textsc{MethId} \overset{\text{fin}}{\to} \{\mathbf{\texttt{final}}, \mathbf{\texttt{abstract}}\} \\
\mathcal{I} & \in & \textsc{Interface} = (\textsc{ProtoObj} \times \textsc{MkEnv} \times \textsc{PubMembs} \times \textsc{MethAn})
\end{array}
$$

Figure 2: Semantic definitions for MINIMOBY classes

As in the case of modules, enrichment captures the relationship between implementations and interfaces (where one should think of $\mathcal{I}$ as an implementation and $\mathcal{I}'$ as an interface). The first three conditions impose the requirement that the interface's members and makers be a restriction of the implementation. Note that since the maker environment maps names to domain types, the subtyping on maker types is contravariant. The fourth property ensures that interfaces do not make members and makers public that are not public in the implementation. The last two properties ensure that final and abstract annotations in the implementation are not lost in the interface. Note that an interface is free to add annotations that are not present in the implementation.

This notion of class interface enrichment can be used to extend the standard definition of an environment Γ enriching an environment Γ' found in the semantics of ML modules. We require that for each class C defined in Γ', C is also defined in Γ and that the interface given to C by Γ enriches that given to C by Γ'.