Object-Oriented Programming in Scheme with First-Class Modules and Operator-Based Inheritance

Guruduth Banavar*
Gary Lindstrom
Department of Computer Science
University of Utah, Salt Lake City, UT 84112

Abstract

We characterize object-oriented programming as structuring and manipulating a uniform space of first-class values representing *modules*, a distillation of the notion of classes. Operators over modules individually achieve effects such as encapsulation, sharing, and static binding. A variety of idioms of O-O programming find convenient expression within this model, including several forms of single and multiple inheritance, abstract classes, class variables, inheritance hierarchy combination, and reflection. We show that this programming style simplifies O-O programming via enhanced uniformity, and supports a flexible model of object-orientation that provides an attractive alternative to meta-programming. Finally, we show that these notions of O-O programming are language independent, by implementing a Modular Scheme prototype as a completion of a generic O-O framework for modularity.

Paper Category: Research. **Topic Area:** Language design and implementation.

1 Introduction

Class-based object-oriented programming is usually thought of as creating a graph structured inheritance hierarchy of classes, instantiating some of these classes, and computing with these instances. Instances are typically "first-class" values in the language, i.e. they can be created, stored, accessed, and passed into and out of functions. Classes, on the other hand, are usually not first-class values, and inheritance is often considered an operational and static structuring activity.

Some dynamic languages like CLOS [18] and Smalltalk [15] permit access to classes at run-time, usually as objects of other (meta-)classes. However, even in dynamic O-O languages, there is often a disparity between the manner in which classes and other values are manipulated. Classes are often not on an equal footing with other values; for example, classes are not passed into and out of functions or stored and retrieved as attributes of other classes. When a more equitable status for classes is desired, meta-programming is resorted to. A meta-level architecture assumes the role of capturing and exposing the properties of classes, objects, and their interactions via a collection

*Primary contact author. E-mail: *banavar@cs.utah.edu*, Phone: +1-801-581-8378, fax: +1-801-581-5843.

of collaborating meta-classes. Programmability of these meta-classes is a powerful means by which languages achieve flexibility.

In this paper, we present an alternative model of O-O programming that we assert to be powerful, flexible, and uniform, all without recourse to meta-programming. In this model, classes are regarded as values just like everything else in the language. Classes can be created, stored in variables, passed into and out of functions, nested arbitrarily, and inherited by other classes through expressions over classes. Classes are instantiated and computation performed with these instances by accessing their data attributes and calling their function attributes. Encapsulation, sharing, and static binding are achieved via individual operators over classes. This point of view gives rise to an expressive programming style that models most existing idioms of O-O programming while providing the flexibility to express many others.

We illustrate this programming style with the programming language Scheme [10], extended with an abstraction mechanism called *modules*. Hence, we call our language *Modular Scheme*. We believe that the present day notion of object-orientation is really the most advanced stage of an evolution towards modularity in programming languages. Modularity aims to achieve important requirements of large-scale software development such as encapsulation, separate development and ease of maintenance. Module systems for many languages have traditionally supported these requirements with notions of isolated name spaces, visibility control via export operations and sharing and reuse via import operations. O-O languages support these same requirements, indeed more effectively, via analogous notions of objects, public/private attributes, and reuse via inheritance. In recognition of their role as the fundamental unit of modern-day software construction, we have chosen to refer to a distilled notion of classes as "modules" in this work.

The Scheme module system presented in this paper has the following important features:

1. It supports the requirements of large-scale software development such as encapsulation, separate development, and inter-module conformability.
2. In the spirit of Scheme, it supports modules as first-class entities, and it is dynamic and interactive. Also, the notion of modules and their instances have a clear denotational semantics based upon record-generators.
3. It supports several idioms of object-oriented programming such as single, prefix-based, mixin-based, and multiple inheritance, method definition and call wrapping, abstract classes, class variables, inheritance hierarchy combination, and reflection.
4. It is language independent. In fact, it has been implemented by reusing the design and code of a generic O-O framework for modules.

We introduce our model in Section 2, comparing and contrasting it to conventional module systems. In Section 3, we illustrate how our model supports common notions of single inheritance. Section 4 illustrates support for three variants of multiple inheritance. In Section 5, we briefly cover the semantics and applications of nested modules, a particularly expressive feature of Modular Scheme. In Section 6, we describe an implementation of our language as a completion of a generic O-O framework. We then relate our work with other research, and present conclusions.

2 Modules and Instances

We introduce our model of modules by relating it to module systems for Scheme such as those given in [12, 28]. In these systems, a module is essentially an environment for binding names to values. A module is a name scope that explicitly provides (exports) names and requires (imports) other names. All names in the environment are directly accessible within the environment itself, whereas public names imported from other environments are dynamically bound.

In contrast, modules in our model conceptually *abstract over* environments. Module interconnection is established by actually combining modules, with interconnection validation at combination time. Individual *instances* of modules represent concrete environments such as those in traditional systems. Specifically, modules abstract over the notion of what "self" means until they are instantiated [11]. This enables a remarkable degree of flexibility in their manipulation.

In Modular Scheme, a module consists of a list of attributes with no order significance. Attributes are of two kinds. *Mutable* attributes are similar to Scheme variables, and can store any Scheme value. *Immutable* attributes are symbols bound to Scheme values in a read-only manner, i.e. they can be accessed but not assigned to.

A module is a Scheme value that is created with the mk-module primitive. Modules can be manipulated, but their attributes cannot be accessed or evaluated until they are instantiated via the mk-instance primitive. The syntax of these two primitives is:

(mk-module $\langle$*mutable-attribute-list*$\rangle$ $\langle$*immutable-attribute-list*$\rangle$)
(mk-instance $\langle$*module-expr*$\rangle$)

The attributes of an instance can be accessed via the attr-ref primitive and assigned to via the attr-set! primitive. Procedures within a module can access sibling attributes via the self-ref primitive, and assign to them with the self-set! primitive. (These primitives are explained in more detail in Section 2.3.)

Figure 1 box (a) shows a module bound to a Scheme variable vehicle-fuel. The module has one mutable attribute fuel, and two immutable attributes: empty?, bound to a procedure which checks to see if the fuel tank is empty, and fill, bound to a procedure that fills the fuel tank of the vehicle to capacity. The fill method refers to an attribute capacity that is not defined within the module, but is expected to be the fuel capacity of the vehicle in gallons. In the vocabulary of traditional module systems, the above module exports the three symbols fuel, empty?, and fill and implicitly imports one symbol capacity.

(a)

```
(define vehicle-fuel (mk-module
                        ((fuel 0))
                        ((empty?  (lambda ()
                                     (= (self-ref fuel) 0)))
                         (fill (lambda ()
                                 (self-set! fuel (self-ref capacity)))))))
```

(b)

```
(define vehicle2-fuel (hide vehicle-fuel '(fuel)))
(describe vehicle2-fuel)
   ⟹
   ((fuel 0)(empty? (lambda () (= (self-ref <priv-attr> 0))) ...
```

(c)

```
(define vehicle-capacity (mk-module ()
                            ((capacity 10)
                             (more-capacity? (lambda (v)
                                               (> (attr-ref v capacity) (self-ref capacity)))))))
```

(d)

```
(define vehicle (merge vehicle2-fuel vehicle-capacity))
(define v1 (mk-instance vehicle))
```

Figure 1: Module definition, combination, and instantiation

2.1 Encapsulation

One of the most important requirements of module systems is encapsulation. This is supported by the primitive **hide**, which returns a new module that encapsulates the given attributes.

(hide ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)

In box (b) of Figure 1, the **hide** expression creates a new module with an encapsulated **fuel** attribute with an internal, inaccessible name. This is shown by the **describe** primitive as **<priv-attr>**. Hiding results in what is known as object-level encapsulation, i.e. the hidden attributes of a particular instance of a module are accessible only by self-reference primitives (e.g. **self-ref**) within that individual instance. They are not accessible externally (e.g. via **attr-ref**), not even by the incoming parameter of a binary method such as the **v** parameter of the **more-capacity?** method of module **vehicle-capacity** shown in box (c) of Figure 1.[1]

2.2 Interconnection

The module **vehicle-capacity** given in Figure 1 box (c) exports two symbols: **capacity**, that represents the fuel capacity of a vehicle in gallons, and **more-capacity?**, bound to a procedure that determines if the incoming vehicle argument has more fuel capacity than the current vehicle.

[1]This style of encapsulation is similar to Smalltalk, and in contrast to the "class-level" encapsulation of C++.

The module **vehicle-fuel** can be combined with **vehicle-capacity** to satisfy its import requirements. This can be accomplished as shown in box (d) via the primitive **merge**, which has the following syntax:

(merge ⟨*module-expr1*⟩ ⟨*module-expr2*⟩)

The primitive **merge** does not permit combining modules with conflicting defined attributes, i.e. attributes that are defined to have the same name.

The new merged module **vehicle** in box (d) exports five symbols and imports none.[2] An instance of this module, such as **v1** in box (d), represents exactly the kind of module interconnectivity that can be specified by the use of import/export operations in traditional module systems.

2.3 Attribute Access

In this section, we describe attributes and their access in more detail. Mutable attributes are bound to fresh locations upon module instantiation, and initialized with the value associated with each attribute. The initialization value of a mutable attribute can be changed via the primitive **override**, described in Section 3. Immutable attributes are bound prior to instantiation, but can be *re-bound* via **override**. We will refer to immutable attributes that are bound to procedures as *methods*, borrowing from O-O programming. Immutable attributes can also be bound to other modules, called *nested modules*, dealt with in Section 5.

The attributes of an instance are accessed with the following primitives:

(attr-ref ⟨*instance-expr*⟩ ⟨*attribute-name*⟩ ⟨*arg-expr**⟩)
(attr-refc ⟨*instance-expr*⟩ ⟨*attribute-name*⟩)
(attr-set! ⟨*instance-expr*⟩ ⟨*attribute-name*⟩ ⟨*expr*⟩)

The values of both mutable and immutable attributes are accessed with the primitive **attr-ref**. If the referenced attribute is a method, it is applied with the given argument(s) and its value returned. Syntactically, accessing the value of a non-method attribute via **attr-ref** is exactly the same as applying a method with no arguments. A method can also be accessed as a first-class closure, without applying it, via the primitive **attr-refc**. For non-method attributes, **attr-refc** is semantically equivalent to **attr-ref**. Mutable attributes are assigned with the primitive **attr-set!**.

A method can access the instance within which it is executing via the expression **(self)**. Thus, a method can access a sibling attribute within the same instance as **(attr-ref (self)** ⟨*attr-name*⟩**)**. However, encapsulated attributes cannot be accessed in this manner. For this, a method uses the analogous primitives **self-ref** and **self-refc** to access the values of attributes, and **self-set!** to assign to mutable attributes, of the instance within which it is executing.

(self-ref ⟨*attribute-name*⟩ ⟨*arg-expr**⟩)

[2]One can check if the import requirements of individual modules are satisfied by using the introspection primitives described in Section 3.2.

(a)	(define new-capacity (mk-module () ((capacity 25)))) (define new-vehicle (override vehicle new-capacity))

(b)	(define vehicle5 (copy-as vehicle '(capacity) '(default-capacity)))
	(describe vehicle5)
	   $\Longrightarrow$
	   ((capacity 10)(default-capacity 10)(fill (lambda () (self-set! fuel (self-ref ...

Figure 2: Rebinding and copying

```
(self-refc ⟨attribute-name⟩)
(self-set! ⟨attribute-name⟩ ⟨expr⟩)
```

Accesses via these primitivies are called *self-references*, whereas accesses via **attr-ref** and **attr-set!** are called *external references*. Figure 1 shows examples of the use of some of these primitives.

2.4 Abstract Modules

An attribute is called *undefined* if it is self-referenced, or referenced from a nested module, but is not specified in the module. A module is *abstract* if any attribute is left undefined. In keeping with dynamic typing in Scheme, an abstract module can be instantiated, since it is possible that some methods can run to completion if they do not refer to undefined attributes. It is a checked run-time error to refer to an undefined attribute.

3 Single Inheritance

We now go beyond traditional module systems and show how our model supports idioms of O-O programming. We start with single inheritance, but we need to first introduce two primitives:

```
(override ⟨module-expr1⟩ ⟨module-expr2⟩)
(copy-as ⟨module-expr⟩ ⟨from-name-list-expr⟩ ⟨to-name-list-expr⟩)
```

The operator **override** produces a new module by combining its arguments. If there are conflicting attributes, it chooses ⟨*module-expr2*⟩'s binding over ⟨*module-expr1*⟩'s in the resulting module. For example, the abstract module **new-capacity** in box (a) of Figure 2 cannot be merged with **vehicle** since the two modules have a conflicting attribute **capacity**. However, **new-capacity** can override **vehicle**, as shown. This way, immutable attributes can be re-bound, and mutable attributes can be associated with new initial values.

The primitive **copy-as** copies the definitions of attributes in ⟨*from-name-list-expr*⟩ to attributes with corresponding names in ⟨*to-name-list-expr*⟩. The *from* argument attributes must be defined.

(a)

```
(define-class vehicle #f
      ((fuel 0))
      ((capacity 10)
       (fill (lambda () (self-set! fuel (self-ref capacity))))
       (display (lambda () (format #t "fuel = ~ a (capacity ~ a) "
                      (self-ref fuel) (self-ref capacity))))))
```

(b)

```
(define-class land-vehicle vehicle
  ()
  ((wheels 4)
   (display  (lambda ()
              (self-ref super-display)
              (format #t "wheels = ~ a "  (self-ref wheels))))))
```

(c)

```
(define land-vehicle
     (hide (override (copy-as vehicle '(display) '(super-display))
                    (mk-module ()
                         ((wheels 4)
                          (display (lambda ()
                                       (self-ref super-display)
                                       (format #t "wheels = ~ a "  (self-ref wheels)))))))
         '(super-display)))
```

Figure 3: Single inheritance

An example is shown in box (b) of Figure 2.

3.1 Super

With the operators discussed thus far, we can describe support for single inheritance. Many single inheritance systems such as Smalltalk-80 and Modula-3 (object types, modulo static typing) [7] share the notion of a class consisting of methods and encapsulated instance variables. In these systems, it is possible to specify a class declaration similar to that shown in box (a) of Figure 3. The **define-class** construct will be explained below. In this example, the attribute **fuel** is intended to be encapsulated as an instance variable, and the Scheme constant **#f** (false) indicates that the class has no superclasses. Such a class declaration is equivalent to writing a **mk-module** expression, and hiding the **fuel** attribute of the resultant module.

Subsequently, a subclass **land-vehicle** of **vehicle** can typically be specified in such systems in a manner similar to box (b). In this definition, a new attribute **wheels** is added, and the **display** binding is overridden with a method that accesses the shadowed method as **(self-ref super-display)**. Such a subclass definition is exactly equivalent to specifying the module expression shown in box (c). In this expression, a module with attributes **wheels** and **display** is created, and is used to

(a)	(let ((conflicts (conflicts-between mod1 (attrs-of mod2)))) (copy-as mod1 conflicts (prepend "super-" conflicts)))
(b)	(hide vehicle (mutable-attrs-of vehicle))
(c)	(defined? vehicle (self-refs-in vehicle more-capacity?))

Figure 4: Introspection

override the superclass vehicle in which the display attribute is copied as super-display. We then hide away the copied super-display attribute to get a module with exactly one display method in the public interface, as desired.

One can write a macro in Modular Scheme to translate define-class expressions such as those for vehicle and land-vehicle into module expressions. In fact, a library of several such useful macros accompanies Modular Scheme. One macro for single inheritance accepts the following syntax:

(define-class ⟨*name*⟩ ⟨*super*⟩ ⟨*inst-var-list*⟩ ⟨*method-list*⟩)

The general form of the module expression shown in box (c) of Figure 3 turns out to be a useful idiom in Modular Scheme. It can be used for expressing other effects such as prefix-based inheritance, wrapping, and mixin combination, described later. We shall refer to this form as the *copy-override-hide* idiom.

3.2 Introspection

The macro define-class given above automatically finds conflicting attributes between two modules by using an *introspective* primitive called conflicts-between. It then uses the copy-override-hide idiom as shown in Figure 3 to achieve single inheritance.

There are several primitives available for determining various kinds of information about modules and instances. Some of them are:

(conflicts-between ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)
(module-of ⟨*instance-expr*⟩)
(attrs-of ⟨*module-expr*⟩)
(mutable-attrs-of ⟨*module-expr*⟩)
(defined? ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)
(self-refs-in ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)

As mentioned, the primitive conflicts-between returns a list of attribute names that are defined in the given module and also exist in the given list. For example, all the attributes of a module that conflict with another module can be copied by using the expression in box (a) of Figure 4.

The module from which an instance was created can be obtained with the primitive module-of. Thus, (module-of (self)) is similar to self class in Smalltalk, like current in Eiffel [24], and myclass in [6]. The names of the publicly accessible attributes of a module are accessible via the attrs-of primitives. For example, the mutable attributes of a module can be encapsulated with the expression in box (b) of Figure 4. The primitive defined? is used to determine if an attribute is defined in a module. It returns #f if any one of the given attributes names is undefined in the given module. If all of them are defined, it returns the incoming list of attribute names.

It is sometimes useful to know the names of public attributes that are self-referenced within a method. The primitive self-refs-in returns a flat list comprising the set of all the self-referenced public attributes within the bindings of the given attribute names. Argument attribute names that are non-existent or bound to non-method values are ignored. For example, to determine if a method will execute without run-time errors relating to locally undefined public attributes (private attributes are always defined), one can evaluate the expression in box (c).

It is worth mentioning that the introspective operations given above do not violate encapsulation since we do not permit any access to the private attributes of modules.

3.3 Adaptation

Apart from hide and copy-as, there are three other primitives which can be used to create new modules by adapting some aspect of the attributes of existing modules.

(freeze ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)
(rename ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩ ⟨*to-name-list-expr*⟩)
(restrict ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩)

The primitive freeze statically binds self-references to the given attributes, provided they are defined in the module. Freezing the attribute capacity in the module vehicle causes self-references to capacity to be statically bound, but the attribute capacity itself is available in the public interface for further manipulation, e.g. rebinding by combination.[3] As shown in box (a) of Figure 5, frozen self-references to capacity are transformed to refer to a private version of the attribute. Operationally, the binding of the private version is shared with the public version, as long as the public version is not re-bound to a new value via overriding. This implies that frozen references to mutable attributes are always shared, since mutable attributes can never be re-bound; they can just be initialized to new values.

The primitive rename changes the names of the definitions of, *and* self-references to, attributes in ⟨*from-name-list-expr*⟩ to the corresponding ones in ⟨*to-name-list-expr*⟩. An example is shown in box (b). Undefined attributes, i.e. attributes that are not defined but are self-referenced, can also be renamed.

[3]This effect is similar to converting accesses to a virtual C++ method into accesses to a non-virtual method. The difference is that C++ allows non-virtual methods to be in the public interface of a class — the general philosophy here is that all public attributes are rebindable, or virtual, like in Smalltalk.

(a)	(define vehicle3 (freeze vehicle '(capacity))) (describe vehicle3) $\Longrightarrow$ ((capacity 10)(fill (lambda () (self-set! fuel (self-ref <priv-attr>)) ...
(b)	(define vehicle4 (rename vehicle '(capacity) '(fuel-capacity))) (describe vehicle4) $\Longrightarrow$ ((fuel-capacity 10)(fill (lambda () (self-set! fuel (self-ref fuel-capacity)))...
(c)	(define vehicle6 (restrict vehicle '(capacity))) (describe vehicle6) $\Longrightarrow$ ((fill (lambda () (self-set! fuel (self-ref capacity))...

Figure 5: Adaptation

The primitive restrict simply removes the definitions of the given (defined) attribute names from the module, i.e. makes them undefined. An example is shown in box (c) of Figure 5.

These module manipulation primitives are applicative, in the sense that they return new modules without destructively modifying their arguments. However, destructive versions of the operators are also available, so that composite module operations can be expressed without compromising efficiency by making unnecessary copies. The name of the destructive version of a primitive is written as the corresponding name of the non-destructive primitive suffixed with a "!", e.g. hide! is the destructive version of hide. Destructive module primitives are to be used with caution, and used only for optimizing stable programs.

3.4 Prefixing

The programming language Beta [21] supports a form of single inheritance called *prefixing* which is quite different from the single inheritance presented in Section 3. In prefixing, a superclass method that expects to be re-bound by a subclass definition uses a construct called inner somewhere in its body. In instances of the superclass, calls to inner amount to null statements, or no-ops. Subclasses can redefine the method, and in turn call inner. In subclass instances, the superclass method is executed first, and the subclass' redefinition is executed upon encountering the inner statement.

The above effect can be achieved in Modular Scheme with the macro define-prefix, illustrated with the vehicle example in Figure 6. Box (a) focuses on the display method of the vehicle class. The expression (self-ref inner-display) corresponds to the inner construct. This class definition expands to the module expression shown in box (b), where a dummy inner-display attribute is merged in. In fact, the define-prefix macro adds such a dummy attribute (prepended with inner-) for every

(a)

```
(define-prefix vehicle #f
    (...)
    (...(display (lambda () ... (self-ref inner-display) ...))))
```

(b)

```
(define vehicle (mk-module (...)
                    (...
                     (display (lambda () ... (self-ref inner-display) ...))
                     (inner-display #t))))
```

(c)

```
(define-prefix land-vehicle vehicle
   (...)
   (...(display (lambda () ... (self-ref inner-display) ...))))
```

(d)

```
(define land-vehicle
   (hide (override (copy-as (merge-inner sub) '(display) '(sub-display))
                   (rename vehicle '(inner-display) '(sub-display)))
         '(sub-display))))
```

Figure 6: Prefixing

immutable attribute in the definition. A subclass land-vehicle of vehicle is defined in box (c), which expands to the expression in box (d). In this expression, the subclass is first merged in with a dummy inner-display with the function (merge-inner sub). The subclass' display method is then copied as sub-display and overridden with the superclass in which inner-display is renamed to sub-display. Lastly, sub-display is hidden away so that there is only one display method.

This example also uses the copy-override-hide idiom introduced in Section 3.1. The difference here is that the superclass overrides the subclass as opposed to the reverse in Section 3.1. Indeed, this is the difference between prefix-based and super-based forms of single inheritance.

3.5 Wrapping

Wrapping, similar to the CLOS notion of :around methods, is useful in many contexts. In fact, wrapping method definitions can be used to simulate :before and :after methods of CLOS as well, since new code can be interposed before or after the call to the old code. It is easy to wrap method definitions using the copy-override-hide idiom shown earlier. Modular Scheme provides a macro called wrap-method to achieve this effect, but we shall omit its description here to conserve space.

A more interesting and less often explored effect is to wrap self-referenced *calls to* particular methods. Say we have a module veh-sim shown in box (a) of Figure 7, which is intended to be combined with the vehicle module. Its method sim-fill calls the undefined method fill upon some condition fill-condition. Say we want to count the number of calls to fill that sim-fill makes. We do not want to wrap the method fill in vehicle, since we want to count only calls from sim-fill. Also,

(a)

```
(define veh-sim (mk-module (...)
                    ((sim-fill (lambda (v)
                                  (if (fill-condition v)
                                      (self-ref fill)))))))
```

(b)

```
(define counted-veh-sim
    (let ((count-sim (merge veh-sim (mk-module ((count 0)) ()))))
      (wrap-call count-sim fill
               (lambda ()
                  (self-set! count (+ (self-ref count) 1))
                  (self-ref fill)))))
```

(c)

```
(hide (merge (rename count-sim '(fill) '(wrap-fill))
             (mk-module ()
                  ((wrap-fill (lambda ()
                                  (self-set! count (+ (self-ref count) 1))
                                  (self-ref fill))))))
     '(wrap-fill))
```

Figure 7: Wrapping calls to methods

we cannot wrap the **sim-fill** method to do this, since every call to it does not necessarily result in a call to **fill**, due to the **fill-condition** test.

Thus, we need to wrap *calls to* **fill** from the **veh-sim** module using the **wrap-call** macro shown in box (b). We add a mutable attribute **count** to **veh-sim**, and wrap its calls to **fill** to increment the counter. The module expression that the **wrap-call** expands into is given in box (c). In this expression, we first **rename** the undefined attribute **fill** to **wrap-fill**, thus changing the self-references correspondingly. We then merge in a **wrap-fill** method that increments **count** and calls the old **fill** method in the resulting module.

The general form of the expression in box (c) is another useful idiom in the language, and will be referred to as the *rename-merge-hide* idiom. The distinction between the copy-override-hide idiom and the rename-merge-hide idiom is worth exploring. Figure 8 pictorially shows the use of these idioms for method definitions in the first row and method calls in the second. In the figure, shaded boxes represent hidden methods.

For method definitions, both the idioms are used when a method METH of M1 is being redefined by M2, and the old definition of the method is referred to in the redefinition as METH'. The difference is that copy-override-hide is used when M1's references to METH are to refer to the new METH in the combined module. Rename-merge-hide is used when M1's references are to refer to the old definition renamed as METH', while M2's references are to refer to the redefinition. As an example scenario, rename-merge-hide is not appropriate to achieve the right effect of single

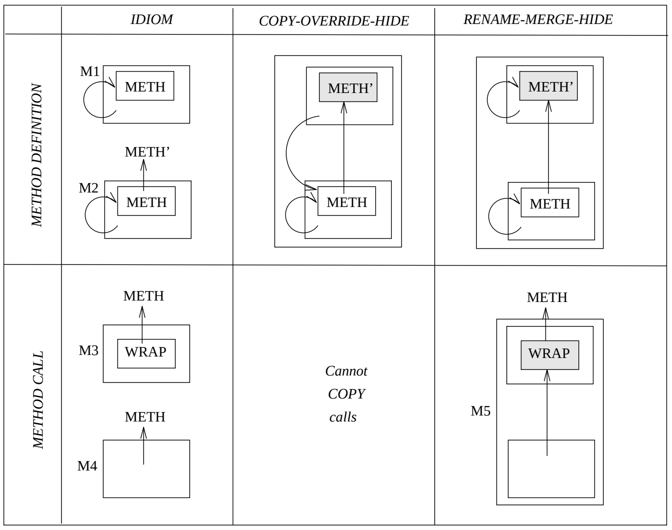

Figure 8: Idioms in Modular Scheme

inheritance. For example, in box (c) of Figure 3, renaming **vehicle**'s **display** method instead of copying it would not work, since in that case self-references to **display** in **vehicle** would also be renamed — we want self-references in the superclass to refer to the new, rebound **display** method.

For method calls, only the rename-merge-hide idiom applies, since undefined attributes cannot be copied. In Figure 8, module M4 has a call to METH which is wrapped to produce M5 as shown. An example was given in the first half of this section.

4 Multiple Inheritance

We have seen in the previous section how to express the creation of a subclass from a single superclass. With multiple inheritance, there is the additional problem of how to compose the superclasses by resolving conflicts and sharing attributes between them. Typically, a language supporting multiple inheritance makes available to the programmer a small number of choices for attribute sharing and conflict resolution. The advantage of O-O programming with operator-based inheritance is that the programmer has numerous options for, and fine-grained control over, decisions taken while combining multiple modules. Also, inheritance history does not intrude into operator semantics, making the approach more compositional.

4.1 Mixins and Linearized Multiple Inheritance

Consider the case of linearized multiple inheritance as in Flavors and Loops, where the graph of ancestor classes of a class are linearized into a single inheritance hierarchy. Each of these languages specifies a different default rule for the linearization of ancestor classes. For example, both these

(a)

```
(define land-veh-chars (mk-module ()
                         ((wheels 4)
                          (display (lambda () (self-ref super-display)
                                     (format #t "wheels = ~ a " (self-ref wheels)))))))
(define sea-veh-chars (mk-module ()
                        ((surface #t)
                         (display (lambda () (self-ref super-display)
                                    (format #t "surface = ~ a " (self-ref surface)))))))
```

(b)

```
(define-subclass land-vehicle (land-veh-chars vehicle))
(define-subclass sea-vehicle (sea-veh-chars vehicle))
(define-subclass amphibian (land-veh-chars sea-veh-chars vehicle))
```

(c)

```
(define amphibian
    (hide (override (copy-as vehicle '(display) '(super-display))
           (hide (override (copy-as sea-veh-chars '(display) '(super-display))
                           land-veh-chars)
                 '(super-display)))
          '(super-display)))
```

Figure 9: Linearized Multiple Inheritance

languages do a depth-first, left-to-right traversal of ancestor classes up to join classes, i.e. classes that are encountered more than once, which get traversed on their first visit in Flavors and last visit in Loops.

It has been argued that currently used linearizations do not ensure that "the inheritance mechanism behaves "naturally" relative to the incremental design of the inheritance hierarchy" [13]. Perhaps it is better to let the programmer select the precedence order of superclasses as dictated by individual applications. In the case of CLOS, a programmer with considerable expertise can use the meta-object protocol of the language and adapt the default rule. In contrast, programming with operator-based inheritance gives the programmer direct control over combination, as shown in Figure 9.

Say we want to create modules for land vehicles and sea vehicles as subclasses of vehicle. We can define modules with the characteristics of land vehicles (number of wheels) and sea vehicles (surface vessel or submarine) as shown in box (a). In these modules, one can think of the expression (self-ref super-display) as being the equivalent of call-next-method in CLOS. Abstract modules such as these are sometimes called "mixins" — reusable abstractions that require other abstractions in order to be usefully applied. Such abstractions have been characterized as functions from classes to classes [4]. However, the approach of operator-based inheritance given here uniformly treats all aspects of inheritance as operations over modules, as was first developed in [3].

With the definitions in box (a), we can create land-vehicle and sea-vehicle "subclasses" of vehicle

(a)	(define color
	    (mk-module
	        ((color 'white))
	        ((set-color (lambda (new-color) (self-set! color new-color)))
	         (display (lambda () (format #t "color = ~ a" (self-ref color)))))))

(b)	(define car-class
	   (hide (merge (merge (rename color '(display) '(color-display))
	                    (rename land-vehicle '(display) '(vehicle-display)))
	            (mk-module ()
	               ((display (lambda () (self-ref vehicle-display)
	                              (self-ref color-display))))))
	       '(color-display vehicle-display))

Figure 10: Multiple Inheritance with no common ancestors

as shown in box (b). This is achieved using the copy-override-hide idiom (Figure 8), but with the macro **define-subclass** which accepts a slightly different syntax. Similarly, we can "chain" the creation of subclasses so that the call to **super-display** in each class calls the **display** method of the next lower precedence superclass. Thus, we can create an **amphibian** class that inherits both the characteristics of land and sea vehicles. The **define-subclass** macro for **amphibian** expands to the module expression shown in box (c) (note the cascaded use of the copy-override-hide idiom), and extends to an arbitrary number of superclasses, as desired.

4.2 Multiple Inheritance with No Common Ancestors

Let us now consider the case of multiple superclasses that are not linearized, and have no common ancestor. Say we have a module **color** defined as in box (a) of Figure 10. We can combine **color** with the module **land-vehicle** shown earlier into **car-class**, as shown in box (b). This expression uses the rename-merge-hide idiom introduced in Section 3.5. The method **display** that conflicts in the "superclasses" **vehicle** and **color** is renamed in each and the superclasses are merged together. A new module that defines a display method that calls the renamed display methods is then merged in to create the desired **car-class**. This example can be extended to more than two superclasses, and can be automated via a macro that uses the introspective primitive **conflicts-between** to rename attributes.

The rename-merge-hide idiom works fine for this example, since there are no self-references to the renamed attribute in the superclasses. However, the right effect of inheritance can only be obtained with **copy-as**, so that self-references in the superclasses are not changed, followed by a **merge**, so that accidental conflicts between superclasses do not get quietly re-bound. The problem with copying conflicting attributes and merging is that the conflicts will still persist. This can be

```
(override (override (rename land-vehicle '(fuel) '(land-fuel))
                    (rename sea-vehicle '(fuel) '(sea-fuel)))
          (mk-module ()
               (display (lambda ()
                          (format ... (self-ref land-fuel) (self-ref sea-fuel) ... ))))))
```

Figure 11: Multiple Inheritance with common ancestors

remedied by restrict'ing (Section 3.3) after copying, and then merging and hiding. There is some similarity between such a copy-restrict-merge-hide operation and the copy-override-hide idiom.

4.3 Multiple Inheritance with Common Ancestors

In the case of superclasses with a common ancestor, such as in the "diamond" problem of multiple inheritance, the situation gets more complex. In this case, the attributes of the common ancestor are clearly conflicting in the superclasses. Furthermore, there is the choice of inheriting either a single copy or multiple copies of mutable attributes from the common ancestor.[4]

To illustrate, consider the two previously given modules land-vehicle and sea-vehicle which have each inherited from the vehicle module. Say we want to create an amphibian module that inherits from these two modules, but needs two copies of the fuel attribute to model two different kinds of fuels for amphibians. This can be achieved with the expression in Figure 11. In this example, the fuel attribute is renamed for each type of module. The two modules are then overridden since the conflicting attributes capacity and fill are known to be identical, and the method display will be overridden in the final module. A new display method that displays all the attributes in an appropriate way is included in the final composition to get the desired module.

The distinction between programming with first-class modules and the operational style more often found in O-O languages is illustrated by this example. Firstly, problems of conflicts and sharing clearly manifest themselves, and compell the programmer to resolve them as the particular situation demands using introspection and inheritance operators. For example, conflicts between superclasses can be inspected with conflicts-between, and superclasses can be overridden in some appropriate order to resolve attribute conflicts. If multiple copies of mutable attributes from the common ancestor are desired, they can be renamed within each superclass, as shown in the example above. However, the burden of resolving conflicts in each individual case can be removed by writing macros that perform a user-chosen method of composition. Secondly, the inheritance history of superclasses is not important; only the attributes of superclasses must be known in order to derive a subclass.

[4]This, of course, is the rationale for virtual and non-virtual base classes in C++.

5 Nested Modules

Since modules are first-class values, attributes of modules can themselves be modules. Modules that are bound to immutable attributes of other modules are referred to as *nested modules*. We will give brief examples in this section to give some insight into the semantics and applications of nested modules — a full treatment is beyond the scope of this paper.

5.1 Semantics

The methods of modules can explicitly refer to bindings in their surrounding scopes using the following primitives, which refer to the given name in a lexically surrounding scope that has a binding for the name.

(env-ref ⟨*attribute-name*⟩ ⟨*arg-expr**⟩)
(env-refc ⟨*attribute-name*⟩)
(env-set! ⟨*attribute-name*⟩ ⟨*expr*⟩)

These three primitives serve functions analogous to the three self-reference primitives. It is a checked run-time error to refer to a name that does not have a binding in some surrounding scope.

Modules follow static scoping rules just like the rest of Scheme. The environment of a module is determined by the lexical placement of the mk-module expression that creates it. In Figure 12 box (a), type1 and type2 are nested modules whose fill methods refer to the capacity attribute of the outer module. Individual vehicles are represented by instances of the nested modules.

The mk-module expressions for these nested modules are evaluated at the time the outer module vehicle-category is instantiated. Thus, nested modules have an instance of their surrounding module as their environment[5], and are bound to their environment at the time of instantiation of the outer module. Thus, v1 in box (a) is a vehicle that shares the capacity attribute of vehicle-category with other instances of type1 and type2. Thus, the attribute capacity serves the purpose of a "class variable" — a variable that is shared among the instances of a class.

With static scoping, a module and its nested modules interact in non-obvious ways. For example, env-ref's in nested modules are equivalent to self-ref's in the outer module. Thus, changes to a module's attributes, e.g. via rename, freeze (static binding), and hide, result in modifications to the environment of nested modules, and hence modify the environment references in nested modules. However, once the outer module is instantiated, environment references in nested modules are permanently bound, regardless of whether the nested module is moved to and combined in another environment with other nested modules created in yet other environments. This is analogous to the creation and manipulation of first-class closures, i.e. procedures with environment references, in Scheme.

[5] Non-nested modules have as their environment the Scheme environment in effect when they are created.

(a)

```
(define vehicle-category
    (mk-module ()
        ((capacity 10)
         (type1  (mk-module (...) ((fill (lambda ... (env-ref capacity) ... )))))
         (type2  (mk-module (...) ((fill (lambda ... (env-ref capacity) ... )))))
         (car    (lambda () (override (self-ref type1)
                                (mk-module ((color 'white)) ()))))))))
(define mycategory (mk-instance vehicle-category))
(define v1 (mk-instance (attr-ref mycategory type1)))
```

(b)

```
(define cap (mk-module ()
                ((type1 (mk-module () ((capacity 10)))))))
(define veh-cap
  (merge (merge (restrict (copy-as vehicle-category '(type1) '(veh-type1)))
                (restrict (copy-as cap '(type1) '(cap-type1))))
         (mk-module ()
             (vehicle (lambda ()
                 (override (self-ref veh-type1) (self-ref cap-type1)))))))
```

(c)

```
(define manager (mk-module ()
                  ((new  (lambda () (mk-instance (self-ref class))))
                   (ref  (lambda (inst attr args)
                            (eval (append `(attr-ref ,inst ,attr) args)))))))
```

Figure 12: Nested Modules

5.2 Applications

Shared repository. A module provides a local namespace for nested modules. It can serve as a shared data repository for nested modules, in addition to serving as a "factory" that produces initialized instances of nested modules. An interesting consequence of this is that names that are not bound within the "top-level" environment can be considered persistent names — this is left as future work.

Hierarchy combination. An inheritance "hierarchy" in O-O programming is usually thought of as a graph of inheriting classes. In Modular Scheme, an inheritance hierarchy is represented simply by a collection of module expressions, some of which are mk-module expressions and others which combine and adapt these modules. Such a hierarchy of modules can be nested within another module. That is, the base class of the hierarchy can be a nested module, and other modules that inherit from it can be computed via module expressions within methods of the outer module (since modules are first-class). For example, a hierarchy consisting of a type1 module and its "subclass" car are shown in box (a) of Figure 12.

Entire hierarchies such as the above can be "combined" with other hierarchies by manipulating the outer modules. Consider a hierarchy cap with a nested module type1 with a single attribute capacity as given in box (b) of Figure 12. Suppose we wish to extend the hierarchy vehicle-category with the hierarchy cap, so that an attribute capacity is added to the type1 module (i.e. the *superclass*), which will be automatically inherited by car (i.e. the *subclass*). This can be achieved with the expression shown in box (b) of Figure 12. Several applications of this style of hierarchy combination are given in [25].

Manager modules. *Reflection* is a means by which programs can access and manipulate themselves. Modular Scheme supports a form of reflective programming on modules with the introspective primitives given in Section 3.2 in conjunction with what are called *manager modules*. A manager module consists of a nested module along with methods that manipulate some extensible functionality to be supported on the nested module. In a sense, a manager module can be used to simulate a meta-class in more conventional designs — this has already been shown by using block structure in the programming language Beta ([23], page 124).

For example, a generic manager module can be specified as in box (c) of Figure 12. This module specifies a method new that returns an instance of an undefined attribute called class, and a method ref that accesses the attribute attr of inst. Basically, the new and ref methods act as surrogates for mk-instance and attr-ref for modules bound to the attribute class.[6] The idea is that any module can be bound to the attribute class, and the new and ref methods can be specialized appropriately for that module via manipulation of the manager module.

6 Implementation

Most of the notions of module manipulation, nesting, and introspection described above are independent of the language into which they are embedded. In fact, an object-oriented *framework* [17] called ETYMA incorporating these generic notions has been designed and implemented in C++. The language described here has been realized as one *completion* of ETYMA. Among the other completions of ETYMA are a linker and a compiler for an interface definition language, described in [2]. This section describes the O-O design and implementation of the modular extension to a basic Scheme interpreter.

The overall architecture of the implementation is shown in Figure 13. The interpreter for Modular Scheme is implemented as an extension of the Scheme interpreter provided in the STk [14] package. The basic Scheme interpreter is implemented in C. The code for the extension is designed as a set of C++ classes that inherit from classes in ETYMA. The Scheme library shown

[6] The new and ref methods could actually be named mk-module and attr-ref.

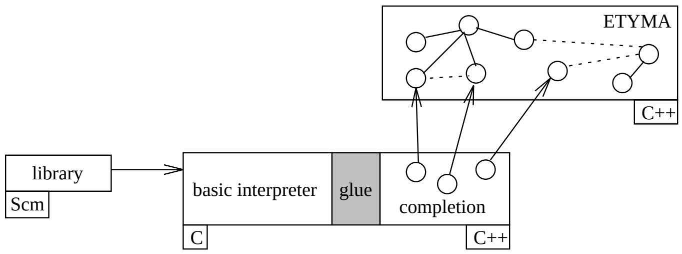

Figure 13: Architecture of extended Scheme interpreter

on the left includes the macros for O-O programming presented in this paper, as well as other functions.

A complete description of the design of the C++ framework ETYMA is beyond the scope of this paper. Briefly, ETYMA models generic linguistic notions that are found in modular languages such as Modular Scheme, thus constituting a meta-architecture for modular systems. For example, ETYMA specifies abstract classes Module and Instance that model the corresponding notions, with the operators for manipulation as their methods. ETYMA also specifies classes Interface, PrimValue, Method, Location, and Label that model the corresponding notions. Standard concrete implementations of C++ classes for modules, instances, and interfaces are available as StdModule, StdInstance, and StdInterface. All these classes collaborate with one another to model the linguistic concepts of object-oriented languages.

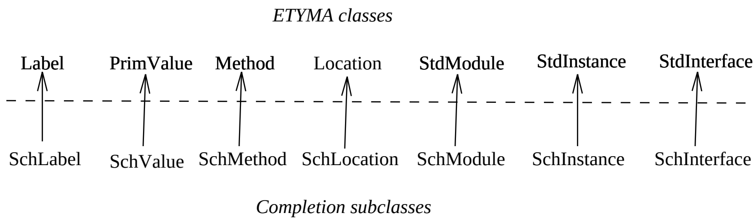

Figure 14: Design of classes in the framework completion

Reuse parameter		New	Reused	% reuse
ETYMA	Classes	7	25	78
	Methods	67	275	80.4
	Lines of Code	1550	5000	76.3
ETYMA + STk	Lines of Code	1800	20000	91.7

Figure 15: Reuse of framework design and code

In order to construct an interpreter for Modular Scheme, we modeled Scheme concepts as subclasses of generic concepts in ETYMA. The only subclasses created to implement Modular Scheme are shown in Figure 14. The reusability of the framework design, in conjunction with the extensibility of the basic Scheme interpreter, made the degree of reuse so high in this case that most of Modular Scheme was designed and implemented in about a week. Figure 15 shows several measures of reuse for this completion. The percentages for class and method reuse give an indication of *design* reuse, whereas those for lines of code give a measure of *code* reuse.

7 Related Work

The design of Modular Scheme is based upon a semantic notion of modules that goes back to record calculi [16, 8]. Classes were modeled as record generators by Cook [11], who also first introduced some of the operators used here. Based on this, Bracha and Lindstrom in [5] developed a comprehensive suite of operators to support sharing, encapsulation, and static binding. Here, we have augmented the above model with the detailed semantics of nested modules, designed and implemented a generic framework based on these notions, completed it to realize a realistic language design, and illustrated typical programming styles and idioms in the language.

Modular programming has traditionally dealt with issues of structuring, encapsulation, and independent development of software. Known by various names such as packages, structures, etc., modules have long played the role of static design artifacts [22, 1]. However, it has not yet been widely recognized that O-O programming is but a sophisticated form of modular programming. The power of first-class modules as given here is even less recognized. In the context of Scheme, several module systems have been developed [12, 28]. These systems essentially provide the functionality described in Section 2, although [12] supports explicit interfaces. Lisp packages [27] and Scheme first-class environments are much restricted forms of modularity compared to the system presented here.

Beta provides a uniform model of programming via *patterns*. However, it does not support first-class modules or operator-based inheritance. Beta's style of prefixing is described and simulated in Section 3.4. Beta supports arbitrary nesting of modules with which meta-classes can be simulated, as with manager modules (Section 5.2). There is also some similarity between manager modules and the concept of "object managers" given in the language Paragon [26], a language for programming with abstract data types.

A popular language family for O-O programming with Lisp is the CLOS family of languages [18, 20]. CLOS supports a quite different model of O-O programming than the one described here, with multiple-dispatch, generic functions, and weak encapsulation. Modular Scheme, on the other hand, supports only single dispatch. CLOS also supports a protocol to interact with its meta-architecture. Dexterity of multiple inheritance as given in Section 4.1 was a primary practical motivation for the CLOS MOP.

Systems such as the CLOS meta-object protocol (MOP) [19] and Open C++ [9] expose the implementation objects of the language processor to the programmer via a controlled protocol. Many aspects of the language's implementation, such as object data layout, are controllable via such a meta-protocol. The approach to O-O programming described here is to provide the flexibility of meta-programming without exposing the meta-architecture to direct user programming. Our approach does not give the user the full power of altering a language's behavior as a MOP can. However, we favor a small set of well-designed primitives that can as effectively provide a uniform and flexible model of O-O programming.

Other specific related languages and semantics are cited throughout the paper.

8 Conclusions

Module systems and O-O programming have long strived to achieve the requirements of large-scale programming such as encapsulation, component-wise development, and reuse. In this paper, we showed that these requirements can be met in a uniform and flexible manner by programming with first-class modules and operator-based inheritance. Modules are manipulated with a suite of operators that individually achieve effects such as encapsulation, combination, sharing, and introspection. This model of modularity has been smoothly integrated into the programming language Scheme while keeping with its original design philosophy that "... a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today." [10]

The above language is expressive and flexible enough to model most previously existing techniques of O-O programming, without recourse to meta-programming. We have shown by examples that the language can emulate an unprecedentedly broad array of idioms such as single, prefix-based, mixin-based, and multiple inheritance, abstract classes, wrapping of method definitions and calls, class variables, inheritance hierarchy combination, and a form of reflection. Thus, the language provides mechanisms to support all of the above, but does not enforce any one inheritance policy. In effect, this language represents a unification of the design space of dynamic, single-dispatch, O-O programming languages.

Finally, we show that the underlying concepts of this programming model are language independent. We have designed and implemented a generic reusable O-O framework in C++ that incorporates the basic abstractions of modularity, and completed it to realize an interpreter for our language. Language independence is conclusively proved by showing that the design and code reuse levels for our completion are very high.

Some important areas of future work remain. Static typing is desirable and possible within our model, although it would introduce several restrictions to the programming style presented here. Compilation is a much more challenging issue, especially to devise composable techniques paralleling the semantic one, and to express in a language independent manner in our generic framework.

Acknowledgements.

We gratefully acknowledge support and several useful comments on this work from Jay Lepreau, Bjorn Freeman-Benson, Bryan Ford, Doug Orr, Robert Mecklenburg, and Nevenka Dimitrova.

References

[1] Reference manual for the Ada programming language. ANSI/MIL-STD-1815 A, 1983.

[2] Guruduth Banavar, Gary Lindstrom, and Douglas Orr. ETYMA: A framework for modular systems. CS Dept. TR UUCS-94-035, University of Utah, December 1994. Short version persented at the workshop on O-O Compilation at OOPSLA '94, Portland, OR.

[3] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, March 1992. Technical report UUCS-92-007; 143 pp.

[4] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proc. OOPSLA Conference*, Ottawa, October 1990. ACM.

[5] Gilad Bracha and Gary Lindstrom. Modularity meets inheritance. In *Proc. International Conference on Computer Languages*, pages 282–290, San Francisco, CA, April 20–23, 1992. IEEE Computer Society. Also available as Technical Report UUCS-91-017.

[6] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In Norman Meyrowitz, editor, *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 457–467, 1989.

[7] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Technical Report 31, Digital Equipment Corporation Systems Research Center, August 1988.

[8] Luca Cardelli and John C. Mitchell. Operations on records. Technical Report 48, Digital Equipment Corporation Systems Research Center, August 1989.

[9] Shigeru Chiba and Takashi Masuda. Designing an extensible distributed language with a meta-level architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming*. Springer Verlag, 1993. LNCS 707.

[10] William Clinger and Jonathan Rees. Revised^4 report on the algorithmic language scheme. *ACM Lisp Pointers*, 4(3), 1991.

[11] William Cook and Jen Palsberg. A denotational semantics of inheritance and its correctness. In *Proc. ACM Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 433–444, 1989.

[12] Pavel Curtis and James Rauen. A module system for scheme. In *Conference Record of the ACM Lisp and Functional Programming*. ACM, 1990.

[13] R. Ducournau, M. Habib, M. Huchard, and M. L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings of OOPSLA*, pages pages 164 – 175, October 1994.

[14] Erick Gallesio. STk reference manual. Version 2.1, 1993/94.

[15] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[16] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. of the ACM Symp. on Principles of Programming Languages*, pages 131–142, January 1991.

[17] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS 91-1696, University of Illinois at Urbana-Champagne, May 1991.

[18] Sonya E. Keene. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, Reading, MA, 1989.

[19] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, MA, 1991.

[20] Gregor Kiczales and Luis Rodriguez. Efficient method dispatch in PCL. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 99–105. ACM, 1990.

[21] Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Moller-Pedersen, and Kristen Nygaard. The BETA programming language. In *Research Directions in Object-Oriented Programming*, pages pages 7 – 48. MIT Press, 1987.

[22] David MacQueen. Modules for Standard ML. LFCS report, Dept. of Computer Science, Univ. of Edinburgh, Scotland, 1986. Part III of *Standard ML*, by Robert Harper, David MacQueen and Robin Milner.

[23] Ole Lehrmann Madsen. Block structure and object-oriented languages. In *Research Directions in Object-Oriented Programming*, pages pages 113 – 128. MIT Press, 1987.

[24] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1987.

[25] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *OOPSLA Proceedings*, pages 25–40, October 1992.

[26] Mark Steven Sherman. *Paragon: A Language Using Type Hierarchies for the Specification, Implementation and Selection of Abstract Data Types.* Springer-Verlag, New York, NY, 1985.

[27] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Bedford, MA, 1984.

[28] Sho-Huan Simon Tung. Interactive modular programming in scheme. In *Proceedings of the ACM Lisp and Functional Programming Conference*, pages pages 86 – 95. ACM, 1992.

Last modified: February 28, 1995