Compositionally Modular Scheme

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

Abstract. We present a new module system for Scheme that supports a high degree of implementation reuse via module composition. The module system encourages breaking down a program into the smallest possible individually meaningful modules, and recomposing them using a powerful set of adaptation and combination mechanisms. Even hierarchical nesting is achieved via a composition operation. This module system is shown to support a stronger and more flexible notion of compositionality and reuse than traditional class-based inheritance in object-oriented programming. Finally, this module system is itself implemented by reusing a language independent OO framework.

Keywords: module systems, object-oriented programming, inheritance, Scheme.

1 Introduction

Modularity is a fundamental facility for controlling complexity in large systems, via decomposition and abstraction. In particular, software modules allow programmers to develop and maintain pieces of a large system relatively independent of each other. However, decomposition alone does not support *reuse* of software components, which is widely accepted to aid the efficient construction of large systems. For this, it is necessary to provide mechanisms for effective *recomposition*, by which conforming modules can be composed to obtain other modules.

Compositional modularity is a model that supports a simple notion of modules along with a powerful notion of their composition. In addition to meeting requirements of large-scale software development such as encapsulation, separate development, and checking of inter-module conformability, the distinguishing goal of this model is to enable maximal reuse of software components. It encourages breaking down software into the smallest possible individually meaningful units, then recomposing them in various ways to get larger modules. Aspects of modules can be adapted in several ways to make them suitable for composition in new ways.

In essence, compositional modularity distills, unifies, and further advances many existing notions of modularity. In particular, this includes varieties of class-based OO programming, in which inheritance is the primary mechanism for implementation composition to create new classes (modules). Traditionally, OO inheritance is a composite notion, involving module extension, attribute rebinding, encapsulation, static binding, etc. In contrast, individual aspects of inheritance are achieved in our model using separate operations, which can be used in combination to emulate

*Contact: e-mail: *banavar@watson.ibm.com*, Phone: +1-914-784-7755, fax: +1-914-784-7455.

important varieties of composite inheritance. Beyond traditional inheritance, our model also supports a new notion of *compositional nesting*, whereby independently developed modules can even be retroactively nested into conforming modules using a compositional embedding operation. Thus, compositional modularity supports a stronger as well as a more flexible notion of compositionality and reuse than traditional OO inheritance mechanisms.

The flexibility to emulate various notions of inheritance within a single model is itself advantageous over traditional inheritance models. Users can choose the most appropriate inheritance idiom for particular problems. Secondly, users have explicit and fine control over the semantics of module combination such as sharing and conflict resolution, which they traditionally have little control over.

We present the above module system within the context of a programming language named *Compositionally Modular Scheme*, or *CMS* for short. *CMS* is an extension of vanilla Scheme [6]. In the spirit of Scheme, *CMS* 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, described below. Although the model is presented here in the context of Scheme, it is actually independent of the particular programming language within which it is embedded.

In the following section, we place our work in the context of existing work in module systems. Section 3 introduces the basic concepts of the *CMS* module system and shows how *CMS* supports the traditional requirements of modularity. Sections 4 and 5 show *CMS*'s ability to directly emulate the important varieties of composition via OO inheritance. In Section 6, we discuss *CMS*'s support for module nesting. Finally, we sketch the implementation of *CMS* in Section 7 and conclude.

2 Background and Previous Work

Traditionally, a module is understood as an environment for binding names to values. A module is a namespace that explicitly provides (exports) names and requires (imports) other names. All names in the environment are directly accessible within the environment itself, whereas names declared public may be imported by others.

In contrast, a compositional module represents an *abstracted* environment. To understand this notion, consider a parameterized module. A parameterized module abstracts over some subset of names referenced within its environment. It can be instantiated into a concrete module by supplying particular bindings for these abstracted names. Although simple, this technique allows the module to be reused in many applicable situations. A compositional module takes this notion of abstracting over names to its logical conclusion — it abstracts over *all* the names that can ever be referenced from within its environment. This idea can be formalized as a *closed record generator*

$$\lambda e.\ \lambda s.\ \{a_1 = v_1, \ldots, a_n = v_n\}$$

which is a record abstracted over its own *self* (denoted as s) as well as its surrounding environment, e. Within the record, all names are accessed via the s or the e parameter. Such a structure is instantiated into concrete modules by supplying it an environment and taking its fixpoint. The crucial advantage of such an abstracted namespace, however, is that the s and e parameters can be manipulated in many desirable ways before actually instantiating it into concrete namespaces. For example, two such structures can be combined by appropriately composing their s parameters, or one such structure can be embedded into another by supplying the e parameter of the former with the s parameter of the latter. The reader is referred to [1, 2] for a development of the formalism. This ability to manipulate the namespace enables a high degree of compositionality and reuse.

The design of *CMS* is based upon the above semantic notion of modules that goes back to record calculi [11, 5]. Classes were modeled as record generators by Cook [7], who also first introduced some of the operators used here. Based on this, Bracha and Lindstrom in [2] developed a suite of operators to support sharing, encapsulation, and static binding. In this paper, we further augment the above model with the notion of compositional nesting, enabled by the e parameter of the previous paragraph. More importantly, we develop a consolidated notion of compositional modularity, realize it as a new and realistic module system for Scheme, show how to emulate other familiar composition mechanisms using this system, and illustrate typical programming styles and idioms in the language.

Several module systems have been proposed for Scheme [8, 20, 18]. *CMS* is different from these systems in its explicit goal of supporting reuse via module composition. In *CMS*, interconnection of modules is not done via import/export declarations, but rather by explicitly combining the modules involved, possibly after adaptation. (One simple notion of adaptation by renaming was supported in [20].) Some previous systems (e.g. [8, 18]) support explicit interfaces. Although the *CMS* language presented here does not support this, interfaces can be built up dynamically by specifying a module's public interface attributes and providing error stubs for methods. Subsequently, implementation modules for this interface can be composed with it overriding the stub, and private attributes in the resulting module encapsulated via a retroactive hiding operation.

In the context of Scheme, it is natural to support modules as first-class values. The uniformity and expressive power obtained by using first-class modules was recognized in the early language Pebble [3]. More recently, many other languages such as FX [19] and Rascal [12] also support first-class modules.

Some Scheme implementations support first-class environments, which can be dynamically created and extended, and expressions evaluated within them. The environment can also be captured at any point by using a special primitive, such as the-environment. However, such simple notions of environments are not very powerful — the only useful operation defined on them is eval.

A more powerful notion of first-class environments with reflective operations has been proposed in the language Rascal [12]. While *CMS* and Rascal are similar in that they support operations on first-class abstractions, the approaches used are entirely different. Rascal uses the approach of reflection, adaptation, and reification of first-class environments. *CMS* uses the approach of module combination and adaptation with the specific goal of reuse. Rascal does not support *CMS*'s wide array of adaptation mechanisms such as name conflict resolution, static binding, retroactive encapsulation, and compositional nesting.

From one perspective, the operations provided by *CMS* can be viewed as "meta-level" primitives to achieve various goals of module composition. In some respects, e.g. inheritance (and method dispatch to some extent), *CMS* provides the programmer flexibility akin to that provided by metaobject protocols (MOPs) [14], without actually exposing the meta-architecture implementation to direct user programming. However, a full-blown MOP gives the user much more control over various other aspects of a language implementation as well, such as object layout.

Further comparisons with specific OO languages will be made as we proceed.

3 Modules and Instances

In *CMS*, a module consists of a set of attributes (symbol-binding pairs) with no order significance. 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. Attributes are of two kinds depending upon their mutability after instantiation: *mutable* attributes are those that are bound to locations (similar to Scheme variables), and can store any Scheme value; and *immutable* attributes are those that are bound to Scheme values in a read-only manner, i.e. they can be accessed but not assigned to. The syntax of the above primitives is:

(mk-module ⟨*mutable-attribute-list*⟩ ⟨*immutable-attribute-list*⟩)
(mk-instance ⟨*module-expr*⟩)

Expressions that create modules, such as the mk-module expression above, are notated as ⟨*module-expr*⟩. Similarly, instance expressions are notated as ⟨*instance-expr*⟩.

Attributes and Their Access. Immutable attributes correspond to the fixed "behavior" of the abstraction represented by a module, whereas mutable attributes correspond to its "state." Thus, mutable attributes are bound to fresh locations upon module instantiation, and initialized with the value associated with each attribute. Immutable attributes that are bound to procedures are referred to as *methods*, borrowing from OO programming. Immutable attributes can also be bound to other modules, called *nested modules*.

The values of mutable and immutable attributes are accessed with the primitive (attr-ref ⟨*instance-expr*⟩ ⟨*attribute-name*⟩ ⟨*arg-expr**⟩) . If the referenced attribute is a method, it is applied with the given argument(s) and its value returned. Mutable attributes are assigned with the primitive (attr-set! ⟨*instance-expr*⟩ ⟨*attribute-name*⟩ ⟨*expr*⟩).

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 (described below) cannot be accessed in this manner. For this, a method uses the analogous primitive (self-ref ⟨*attribute-name*⟩ ⟨*arg-expr**⟩) to access the values of attributes, and (self-set! ⟨*attribute-name*⟩ ⟨*expr*⟩) to assign to mutable attributes, of the instance within which it is executing. Accesses via these primitives are called *self-references*, whereas accesses via attr-ref and attr-set! are called *external references*.

Figure 1 (a) shows a module bound to a Scheme variable fueled-vehicle. 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,

(a)

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

(b)

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

(c)

```
(define capacity-module
   (mk-module ()
        ((capacity 10)
         (greater-capacity? (lambda (in) (> (self-ref capacity) (attr-ref in capacity)))))))
(define vehicle (merge encap-fueled-vehicle capacity-module))
```

(d)

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

(e)

```
(define v1 (mk-instance vehicle))
```

Figure 1: Basic module operations. (a) Definition via mk-module (b) Encapsulation via hide (c) Combination via merge (d) Rebinding via override, (e) Instantiation via mk-instance.

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.

Encapsulation. The primitive hide returns a new module that encapsulates the given attributes.

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

In Figure 1 (b), the hide expression creates a new module with an encapsulated fuel attribute that has an internal, inaccessible name. This is shown by the describe primitive as <priv-attr>.

It is important to note that such retroactive encapsulation *shrinks* the interface of a module. As a result, functions expecting an instance of a particular module may not necessarily operate correctly on an instance of the module subjected to a hide operation. This represents the widely accepted notion of separating inheritance from subtyping [4].

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

The module fueled-vehicle can be combined with capacity-module to satisfy its import requirements. This can be accomplished via the primitive (merge ⟨*module-expr1*⟩ ⟨*module-expr2*⟩). The new merged module vehicle in 1 (c) exports four symbols and imports none.

The primitive merge does not permit combining modules with conflicting defined attributes, i.e. attributes that are defined to have the same name. If there are name conflicts, one can use the operator (override ⟨*module-expr1*⟩ ⟨*module-expr2*⟩). In the presence of conflicting attributes, override creates a new module by choosing the right operand's binding over the left operand's in the resulting module. For example, the module new-capacity in Figure 1 (d) 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.

Abstract modules and interfaces. An attribute is called *undefined* if it is self-referenced (see above), or referenced from a nested module, but is not specified in the module. If it is specified, it is called *defined*. 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.

The role of abstract classes in OO programming is to specify the interface of a set of similar classes, without specifying the implementation. As mentioned earlier, this can be done in *CMS* by binding abstract methods with dummy error methods, and subsequently overriding these methods.

Adaptation. Thus far, we have mostly shown how *CMS* supports the notions of traditional module systems. In this section, we go beyond traditional module systems, and describe operators to adapt particular aspects of the attributes of existing modules, in order to make them suitable for composition in new ways. Besides hide, there are four other primitives which can be used to create new modules by adapting some aspect of the attributes of existing modules.

The primitive (restrict ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩) simply removes the definitions of the given (defined) attributes from the module, i.e. makes them undefined.

The primitive (rename ⟨*module-expr*⟩ ⟨*from-name-list-expr*⟩ ⟨*to-name-list-expr*⟩) 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*⟩. Undefined attributes, i.e. attributes that are not defined but are self-referenced, can also be renamed.

```
(describe (rename vehicle '(capacity) '(fuel-capacity)))
    ⟹
    ((fuel-capacity 10)(fill (lambda () (self-set! fuel (self-ref fuel-capacity)))...
```

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

```
(describe (copy-as vehicle '(capacity) '(default-capacity)))
    ⟹
    ((capacity 10)(default-capacity 10)(fill (lambda () (self-set! fuel (self-ref capacity...
```

The primitive (freeze ⟨*module-expr*⟩ ⟨*attr-name-list-expr*⟩) statically binds self-references to the given attributes, provided they are defined in the module.

```
(describe (freeze vehicle '(capacity)))
    ⟹
    ((capacity 10)(fill (lambda () (self-set! fuel (self-ref <priv-attr>)) ...
```

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. (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.) As shown above, frozen self-references to capacity are transformed to refer to a private version of the attribute.

The above 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.

4 Single Inheritance

Super-based single inheritance. The operators discussed above can be used in combination to get composite effects of single inheritance of classes, such as in Smalltalk-80. A class consists of methods and encapsulated instance variables, which can be "extended" via inheritance. In *CMS*, a similar notion of inheritable classes can be supported using a macro such as define-class below:

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

The macro specifies the name of the class, its superclasses, a list of encapsulated instance variables and their initializers, and a publicly visible list of methods. Figure 2 (a) shows a class vehicle with no superclasses (indicated by the Scheme constant #f) with one encapsulated instance variable fuel and three public attributes. This macro definition simply expands into a mk-module expression followed by a hide operation on the fuel attribute.

Subsequently, a subclass land-vehicle of vehicle can be specified as in 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). To get the proper effect of rebinding of the display method, this macro expands into the module expression in box (c), explained below.

In this expansion, a module corresponding to the subclass, with attributes wheels and display, is created. This module cannot simply override the superclass module, since in that case, the superclass display method will be wiped out. Neither can the superclass' display method be renamed to super-display before overriding, since in this case, self-references to display in the superclass will also get renamed. The crucial aspect of single inheritance is to have the self-references in the superclass access the *rebound* definitions of methods. Thus, the superclass' display method must be *copied* as super-display before the override operation. The copied super-display attribute is finally hidden away to get a module with exactly one display method in the public interface, as desired.

CMS supports several primitives for determining various kinds of "meta-level" information about modules and instances. For example, the macro define-class above can find conflicting at-

(a)

```
(define-class vehicle #f
    ((fuel 0))
    ((capacity 10)
     (fill (lambda () (self-set! fuel (self-ref capacity)) (self-ref display)))
     (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)))
```

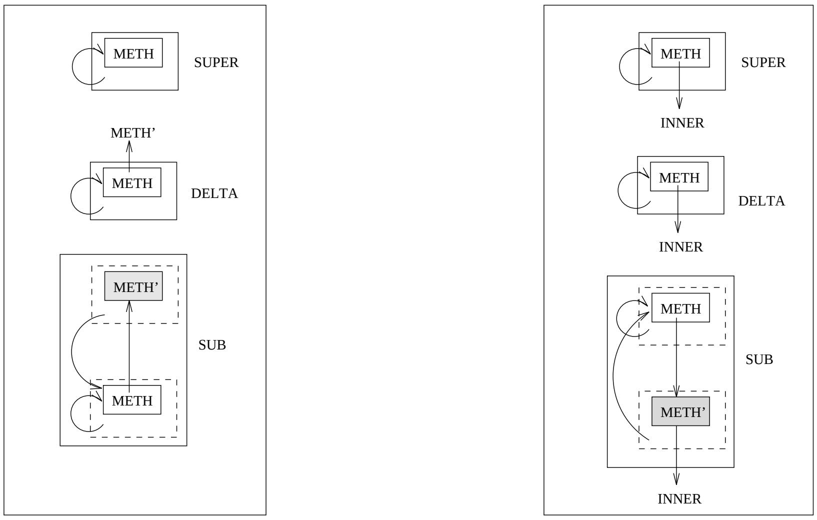
Figure 2: Super-based single inheritance. (a) Superclass, (b) Subclass, (c) Expansion of macro in (b).

tributes between modules by querying for the names of their public attributes. Similarly, the self-references within a module and the module of an instance can also be queried for.

Prefixing. The programming language Beta [15] supports a form of single inheritance called *prefixing* which is quite different from the single inheritance presented earlier. 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 module operators of *CMS* can be used in combination to produce the effect of prefix-based single inheritance as well. This is shown pictorially in Figure 3, where super-based and prefix-based forms of inheritance can be contrasted side by side. Both forms essentially use the same sequence of module operations: copy, override, and hide. The difference is that the superclass overrides the subclass in the case of prefix-based inheritance, as opposed to the reverse for super-based inheritance. Indeed, this is the difference between prefix-based and super-based forms of single inheritance.

The general form of the module expressions shown in the figure turns out to be a frequent idiom in *CMS*. We shall refer to this form as the *copy-override-hide* idiom. The other common idiom in *CMS* is the *rename-override-hide* idiom. Since the rename operation can be applied to both defined



(A) (B)

Figure 3: Pictorial representation of single inheritance. (a) Super-based: (hide (override (copy-as SUPER METH METH') DELTA) METH'), (b) Prefix-based: (hide (override (copy-as DELTA METH METH') (rename SUPER INNER METH')) METH').

and undefined (but self-referenced) attributes, the rename-override-hide idiom can be applied to obtain a diverse number of useful effects, some of which are described in the following section.

5 Multiple Inheritance

We have seen in the previous section how to express the creation of a subclass from a single superclass and a specification of the incremental changes. 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 compositional modularity is that the programmer has numerous options for, and fine-grained control over, decisions taken while combining multiple modules.

Mixins and linearized multiple inheritance. A free-standing module that represents incremental changes to existing modules is sometimes known as a "mixin," since it can be combined with any conforming module. A module representing the characteristics of a land vehicle, such as that given in the inner mk-module expression in Figure 2 (c), is an example of a mixin.

One might conceivably want to combine multiple mixins with a base abstraction. For example, one can envision combining two mixins named land-vehicle-chars and sea-vehicle-chars with vehicle to produce an amphibian module. To do this in *CMS*, all one must do is to cascade copy-override-hide expressions in the desired order, thus performing an explicit linearization of all the modules involved, and combining them in the manner of single inheritance.

Linearization of multiple superclasses is the standard technique for multiple inheritance used in

(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 4: Multiple Inheritance with no common ancestors. (a) A color module, (b) Combining vehicle and color into car-class.

languages such as Flavors and Loops, where the graph of ancestor classes of a class are linearized into a single inheritance hierarchy. However, each of these languages specifies a different default rule for the linearization of ancestor classes. For example, both these 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" [9]. It is perhaps more desirable to let the programmer select the precedence order of superclasses as dictated by individual applications, as in the case of *CMS*. (In the case of CLOS, a programmer with considerable expertise can use the meta-object protocol of the language and adapt the default rule.)

Multiple inheritance with no common ancestors. Consider the case of multiple superclasses that are not linearized, and have no common ancestor. Say we have a module color defined as in Figure 4 (a). We can combine color with the module land-vehicle shown earlier into car-class, as shown in box (b). 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 of course be extended to more than two superclasses. Also, if there are self-references to conflicting attributes in superclasses, it may be more appropriate to copy and restrict them before merging.

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. (This, of course, is the rationale for virtual and non-virtual

```
(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 5: Multiple Inheritance with common ancestors. Creating an amphibian module from land-vehicle and sea-vehicle, which have each inherited from vehicle.

base classes in C++.)

To illustrate, consider two modules land-vehicle and sea-vehicle which have each inherited from the previously given 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 5. 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.

An important distinction between traditional inheritance and compositional modularity is illustrated by this example. In systems of traditional inheritance where there are default rules for resolving the diamond problem, a subclass might break if inheritance relationships, an implementation detail, are changed. This amounts to violation of encapsulation. In the case of compositional modularity, problems of conflicts and sharing clearly manifest themselves, and compell the programmer to explicitly resolve them as the particular situation demands using introspection and inheritance operators. For example, conflicts between superclasses can be inspected and superclasses can be overridden in some appropriate order to share the attributes. Or, if multiple copies of mutable attributes from the common ancestor are desired, they can be renamed (or hidden) within each superclass, as shown above.

6 Module Nesting

Since modules are first-class, a module can be bound to an attribute of another, giving rise to a *nested module*. Hierarchical nesting of modules has numerous applications. It helps control problems associated with flat global name-spaces, such as name pollution and accidental name conflicts. A module can serve as a shared data repository for nested modules, and could perhaps serve as a "factory" that produces initialized instances of nested modules. Furthermore, nesting can nicely solve some real-world modeling problems such as the *prototype abstraction relation problem* ([16], page 123). Madsen [16] has also shown that nested classes can be used to emulate the functionality of meta-classes.

Since modules are first-class in *CMS*, a module can contain a nested module as well as methods

(a)	(define vehicle-category (mk-module () ((capacity 10) (type1 (mk-module (...) ((fill (lambda... (env-ref capacity)...))))) (type2 (mk-module (...) ((fill (lambda... (env-ref capacity)...)))))))) (define mycategory (mk-instance vehicle-category)) (define v1 (mk-instance (attr-ref mycategory type1)))
(b)	(define veh-type (mk-module (...) ((fill (lambda ... (env-ref capacity) ...))))) (define new-vehicle-category (nest type3 veh-type vehicle-category))

Figure 6: Nested Modules. (a) Lexical nesting, (b) Compositional nesting via the nest operator.

that compute composition operations on the nested module. The outer module can be thought of as containing an inheritance hierarchy of modules. Such modules can themselves be manipulated in several ways to realize a useful application known as inheritance hierarchy combination [17].

Lexical Nesting. In *CMS*, modules follow static scoping rules just like the rest of Scheme. The methods of modules can refer to bindings in their surrounding environments using primitives such as (env-ref ⟨*attribute-name*⟩ ⟨*arg-expr**⟩), analogous to the self-reference primitives given earlier. These primitives refer to the given name in a lexically surrounding scope that has a binding for that name. The environment of a module is determined by the lexical placement of the mk-module expression that creates it. An example is shown in Figure 6 (a).

Nested modules have an instance of their surrounding module as their environment, and are bound to their environment at the time of instantiation of the outer module. Hence, lexical scoping is maintained regardless of whether nested modules are moved to and combined in other environments with other nested modules created in yet other environments. This is analogous to the creation and manipulation of first-class closures in Scheme.

Compositional Nesting. A fundamental requirement of modularity is that individual modules, whether nested or not, must be specifiable independent of any particular context. As a result, independently developed modules must be composable not only at the same level, but also in a hierarchical sense. This means that one must be able to *retroactively* nest a separately developed module within any other conforming module.

The benefits of compositional nested modules derives from the ability to retroactively nest modules. Nested modules can be independently developed, thus supporting team development even in the presence of hierarchical structure. Furthermore, a compositional nested module can be embedded into, and thus reused in, any module that generates a conforming environment.

Modules can be retroactively nested via (nest ⟨*attr-name*⟩ ⟨*nested-mod-expr*⟩ ⟨*outer-mod-expr*⟩). This primitive returns a new module containing an attribute ⟨*attr-name*⟩ bound to the nested module ⟨*nested-mod-expr*⟩ within the given outer module. An example is shown in Figure 6 (b).

The nest expression in the example produces a module that contains the attribute type3 bound to a nested module just as if it was directly lexically nested.

In an interactive language such as *CMS*, modules that contain env-ref's can be specified in the "top-level" environment. However, since modules abstract over their environments, env-ref's in such modules are not automatically bound to names occurring in the top-level environment. Instead, when such a module is instantiated via mk-instance, its environment is bound to the Scheme environment at the point of instantiation. Alternatively, one can use the primitive (bind-env ⟨*module-expr*⟩ [⟨*environment*⟩]) to explicitly bind a module to the optional argument ⟨*environment*⟩, which defaults to the Scheme environment at that point.

7 Implementation and Future Work

The underlying concepts of compositional modularity are independent of the language within which they are embedded. In fact, we have designed and implemented a generic reusable set of C++ classes that embody the language independent aspects of the model. This set of classes, also known as an OO *framework* [13], can be subclassed and instantiated to implement processors for particular languages. We have implemented an interpreter for *CMS* by extending an existing Scheme interpreter implementation (available as part of the STk package [10]) with classes derived from the reusable OO framework mentioned above. Due to the reusability of the framework, we obtained very high levels of reuse (between 70 and 90%) for both the framework design (number of classes and methods reused) as well as for the framework code (number of lines of code). The implementation of *CMS* is interesting in its own right; please see [1] for details.

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 separate compilation and linking techniques paralleling the semantic composition operators.

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 have shown how the language *CMS* meets these requirements in a uniform and flexible manner, via first-class modules and a powerful set of operations to combine them by sharing and resolving name conflicts, adapt them by encapsulating and statically binding attributes, and embed one into another.

We have shown that the *CMS* module system achieves its distinguishing goal of supporting implementation reuse via module composition. We show this by demonstrating that it meets and exceeds similar facilities supported by traditional class-based OO inheritance systems. It is flexible enough to emulate a broad array of existing inheritance idioms including super-based, prefix-based, mixin-based, and several varieties of multiple inheritance. Furthermore, our new notion of compositional nesting permits users to seperately develop modules and retroactively embed them into conforming modules via a composition operation.

In effect, the module system presented here unifies and advances many existing notions of modularity. The module system has been implemented as a language independent set of classes, from which an interpreter for *CMS* has been derived. Finally, the *CMS* module system is completely consistent with Scheme's 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." [6]

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

References

[1] Guruduth Banavar. *An Application Framework for Compositional Modularity.* PhD thesis, University of Utah, Salt Lake City, Utah, 1995.

[2] 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.

[3] Rod Burstall and Butler Lampson. A kernel language for abstract data types and modules. In Giles Kahn, David MacQueen, and Gordon Plotkin, editors, *Proceedings, International Symposium on the Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 1–50. Springer, 1984.

[4] 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.

[5] Luca Cardelli and John C. Mitchell. Operations on records. Technical Report 48, DEC SRC, August 1989.

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

[7] 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.

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

[9] 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.

[10] Erick Gallesio. STk reference manual. Available with STk release, version 2.1, 1993/94.

[11] Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *POPL*, pages 131–142, January 1991.

[12] Suresh Jagannathan. Metalevel building blocks for modular systems. *ACM Transactions on Programming Languages and Systems*, 16(3):456–492, May 1994.

[13] Ralph E. Johnson and Vincent F. Russo. Reusing object-oriented designs. Technical Report UIUCDCS-R-1991-1696, Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois, 1991.

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

[15] 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.

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

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

[18] Jonathan Rees. Another module system for Scheme. Included in the Scheme 48 distribution, 1993.

[19] Mark A. Sheldon. Static dependent types for first class modules. In *ACM Conference on Lisp and Functional Programming*, June 1990.

[20] 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: October 26, 1995