Modularity Meets Inheritance

Gilad Bracha
Gary Lindstrom

UUCS-91-017

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

October 13, 1991

Abstract

We "unbundle" several roles of classes in existing languages, by providing a suite of operators independently controlling such effects as combination, modification, encapsulation, name resolution, and sharing, all on the single notion of *module.*

All module operators are forms of inheritance. Thus, inheritance not only is not in conflict with modularity in our system, but is its foundation.

This allows a previously unobtainable spectrum of features to be combined in a cohesive manner, including multiple inheritance, mixins, encapsulation and strong typing.

We demonstrate our approach in a language (called *Jigsaw*, as in the tool, not the puzzle!). Our language is modular in two senses: it manipulates modules, and it is highly modular in its own conception, permitting various module combinators to be included, omitted, or newly constructed in various realizations. We discuss two pragmatic avenues for the exploitation of this approach:

1. Adding modules to languages without modularity constructs.
2. Embedding selected new modularity capabilities within existing object-oriented languages (which we are undertaking as a "proof of concept" in the case of *Modula-3* [5]).[1]

[1]This research was sponsored by (i) the Defense Advanced Research Projects Agency (DOD), monitored by the Department of the Navy, Office of the Chief of Naval Research, under Grant number N00014-91-J-4046, and (ii) the National Science Foundation under Grant No. CCR89-20971. The opinions and conclusions contained in this document are those of the authors and should not be interpreted as representing official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation, or the US Government.

1 Introduction

This paper argues that inheritance, properly formulated, is a powerful modularity mechanism, and can constitute the basis of a module manipulation language.

We arrive at our formulation of inheritance by observing that in languages supporting multiple inheritance (e.g., [10, 23, 27]), classes are burdened with too many roles. The class construct is "large" and monolithic. We opt to simplify classes, and partition their functionality among separate operators.

Classes are reduced to a simple notion of module - a mutually recursive scope. These modules form a uniform space of values upon which operators act. The operators accept modules as arguments, and produce modules as results. The notion of module with its associated operations can thus be viewed as an abstract datatype.

The set of operators we present supports encapsulation, multiple inheritance, mixins and strong typing in a single, cohesive language. These features have not been successfully combined before.

Apart from the obvious relevance to object-oriented programming languages, our framework can be used to introduce modularity to a variety of languages, regardless of whether they support first class objects.

Our approach is itself modular. Language designers can use this approach, and add, remove or replace operators. This makes the benefits of extensibility and modifiability associated with object oriented programming, available at the language design level.

We demonstrate these points via the module manipulation language *Jigsaw*. For concreteness, we assume that *Jigsaw* manipulates modules written in an applicative language with a type system based upon bounded universal quantification [8]. However, the discussion remains virtually unchanged if modules are written in another language. In particular, our operator definitions are not significantly impacted by the use of an imperative language. Similarly, though we assume a subtype relation, we do not rely on its particulars. Hence our approach applies to languages without subtyping as well. These have type equivalence as a trivial subtyping relation.

Jigsaw's semantics are defined by a translation to an untyped λ calculus augmented with basic types, records, record operators, and *let* and *where* constructs.

The interesting part of the translation is that which defines modules and the operations upon them. *Jigsaw* is a typed language that guarantees the type safe use of module operators. However, we have chosen to define the operators in an *untyped* λ calculus. A typed calculus is not used, because we do not know of one that can express all the module operators defined here in their full generality.

Instead, we informally derive sufficient typing constraints on the primitive module operations. These constraints provide guidance as to the formulation of the typechecking rules of *Jigsaw*. Under the crucial assumption that the types of modules are always completely known, typechecking is straightforward. We do not give the formal type rules of *Jigsaw* here,

```
P =  object
   {  x = 0; y = 0;
    dist =  function(aPoint)
                    {
                      sqrt(sqr((x - aPoint.x)) + sqr((y - aPoint.y)))
                    }
   }
```

$$Pgen = \lambda s.\{x = 0, y = 0, dist = \lambda aPoint.\sqrt{((s.x - aPoint.x)^2 + (s.y - aPoint.y)^2)}\}$$

Figure 1: An object and its generator

but mention the relevant typing constraints on each operation.

The remainder of the paper is structured as follows. We begin by reviewing necessary background material in section 2. Section 3 discusses the many roles played by classes in object oriented languages. Section 4 then demonstrates how each of these roles is supported by *Jigsaw*'s operators. Section 5 discusses the application and implementation of the model. Section 6 relates the present paper to previous work by the authors and others. Finally, we present our conclusions.

2 Background

2.1 Generators

In object oriented programming, objects include data and code that operates upon that data. Objects are thus inherently self-referential. The standard technique for modeling self reference is fixpoint theory [22]. Using fixpoint theory, an object may be modeled using a record-generating function (called a *generator* following Cook [9]). Figure 1 shows a simple object and its associated generator function. This function takes a record as a parameter, and returns a record as a result. The result record is similar to the object being modeled. The object's methods, such as dist, are represented by function valued fields in the result. The object's data are represented by fields with ordinary values (e.g., x and y). All self reference in the object is replaced by reference to the generator's formal parameter, s. The desired object is the least fixed point of the generator function $Y(Pgen)$.

2.2 Record Operations

We now define our record operations. Similar operations have been used in the study of typed record calculi [7, 11, 31, 26]. However, this paper is not concerned with the typing problems raised by these operators. Here, record operations are only used in the definitions of module operators. These, in turn, are used only when the types of their operands (i.e., modules) are exactly known, so that type safety is easily guaranteed.

Each record operator has a corresponding operator for generators (see section 4). Related operators are distinguished by subscripts (e.g., θ_r is a record operator, and θ_g is a generator operator[2]). We denote records with r, r_1, r_2, names of record attributes with a, b and lists of attribute names with A.

The operators used here are:

- Merge, $\|_r$. $r_1 \|_r r_2$ yields the concatenation of r_1 and r_2. The records must not have any names in common.
- Restrict, $\backslash_r$. $r \backslash_r a$ removes the attribute named a from r. If a is not defined, $r \backslash_r a = r$.
- Project, π_r. $r \pi_r A$ projects the record r on the names A. The names in A must be defined in r.
- Select, $._r$. $r ._r a$ returns the value of the attribute named a in r. The name a must be defined in r.
- Override, $\leftarrow_r$. $r_1 \leftarrow_r r_2$ produces a result that has all the attributes of r_1 and r_2. If r_1 and r_2 have names in common, the result takes its values for the common names from r_2.
- Rename, $[\_ \leftarrow \_]_r$. $r[a \leftarrow b]_r$ renames the attribute named a to b. The name a must be defined in r, and b must not.

[2]For a record operator θ_r, θ_g is what is referred to in [9] as the distributed version of θ.

```
MP = P  override
   {  dist =  function(aPoint)
                    {
                       (x - aPoint.x) + (y - aPoint.y)
                    }
   }
```

$$MPgen = \lambda s.Pgen(s) \leftarrow_r \{dist = \lambda aPoint.(s.x - aPoint.x) + (s.y - aPoint.y)\}$$

Figure 2: A manhattan point inherits from a point

2.3 Inheritance

This subsection discusses the denotational semantics of inheritance [14, 25, 9]. Inheritance provides a way of modifying self-referential structures [9]. When a value is modified via inheritance, all self reference within the result refers to the modified value. Inheritance involves manipulating the self reference within objects. Technically, this is achieved by manipulating generators, *before* taking their fixpoint [25], [9]. Figure 2 illustrates this process. The object MP inherits from P, but specializes the dist method. MP is modeled by a generator that invokes the generator for P. This invocation yields a record that is combined using the override operation with another record which represents the specialized or new methods. In the modifying record, self reference is modeled in the usual way, by reference to the generator's parameter. P's generator is passed this parameter as well, thereby binding self reference in all methods to the modified object.

2.4 Mixins

In Figure 2, the keyword override is followed by a clause modifying the object P (viz. { dist = ... }). It is often desirable to denote such a modification independently, and reuse it. An example is given in Figure 4. Such a denotable modification is called a *mixin*. Mixins represent an important form of reuse, but have been expressible only in dynamically typed languages (e.g., [15, 29]), where inheritance violates encapsulation. Support for mixins in an encapsulated manner has been been the topic of recent research [4, 12].

2.5 Abstract Classes and Frameworks

One of the most useful ideas in object-oriented programming is that of an *abstract class*. An abstract class is an incomplete definition, in which one or more of the methods declared by the class are not given definitions. The expectation is that these missing method definitions will be provided in subsequently defined subclasses. In some languages, abstract classes have no special linguistic support. Programmers define "dummy" routines that typically produce a run-time error. More recent languages [10, 23] explicitly support abstract classes. In these languages, methods that are undefined in the abstract class are identified by special syntax. Here we use the *C++* terminology, and refer to such methods as *pure virtuals.*

Abstract classes are essential to the definition of *frameworks.* A framework is a collection of classes designed to support a particular application in a modifiable and extensible manner. The user of a framework will adopt it as a basis for his or her application, typically modifying some of the framework's abstract classes to tailor them to specific needs. Examples of frameworks are [19, 30, 33].

Abstract classes support a powerful form of parameterization, unique to the object oriented paradigm. While standard parametrization allows entities to refer to parameters, abstract classes close the loop by also allowing parameters to refer to the parameterized entity (i.e., via a self construct).

Semantically, an abstract class may be modeled as an *inconsistent generator.* An inconsistent generator has the form $\lambda s\colon \sigma\ .e$, where $e : \sigma'$ and σ is a subtype of σ' [9]. This captures the fact that self reference within the class (σ) assumes more methods than the class provides (σ'). One cannot take the fixpoint of such a generator, since its domain is a proper subtype of its range. This models the fact that abstract classes must not be instantiated.

In many object oriented languages, types are identified with classes and subtyping with inheritance. In such languages, the notion of abstract class is often abused, by being pressed into service as a substitute for a more versatile concept of *interface.* In this case, the abstract class provides no definitions at all, only declarations. This is inescapable in such a language, when multiple implementations of an abstraction are required. In languages where types and subtyping are separated from classes and inheritance, this subterfuge is unnecessary.

The next section outlines functionality required of (possibly abstract) classes in an object oriented language. Following that, we present a set of operators on abstract classes that support the required functionality.

3 Roles of a Class

In a language supporting multiple inheritance, the class construct typically supports a large subset of the following functions:

1. Defining a module.
2. Constructing instances of a module definition.
3. Combining several classes together. This is characteristic of multiple inheritance.
4. Modifying a class. This function is characteristic of all inheritance systems, single or multiple.

5. Resolving name conflicts among class attributes. This can be done in various ways, by *renaming* or by *explicitly specifying* the desired attribute.

6. Defining sharing constraints among classes. When classes are combined, certain attributes or groups of attributes may exist in several of the classes being combined. The question is whether these attributes should be duplicated for each participant class, or shared. Too often the semantic decision has been taken at the language level. In fact, different applications have different needs in this respect, and programmers should be able to make the choice.

7. Restricting modifiability. Usually, all visible attributes of a module are subject to modification. It is sometimes desirable to restrict this flexibility, and state that a certain attribute may not be modified by inheritance. This is useful both from a design point of view, and also for optimization.

8. Determining attribute visibility. Different mechanisms may be available, to determine visibility to users, heirs or "friends".

9. Accessing overridden attributes. It is common that a method in a modified class makes use, during computation, of the method it has overridden, using special notation.

In addition, if the language is strongly typed, we often find that a class fulfills additional roles:

10. Defining a type.

11. Defining a subtyping relation.

Following other modern object-oriented language designs (e.g., [1, 5]), we separate inheritance from subtyping.

The following section presents *Jigsaw*'s operator suite. The roles detailed above are examined in turn, and, for each role, the relevant operator(s) described.

4 The *Jigsaw* Operator Suite

4.1 Module Definition

The primary definitional construct in *Jigsaw* is the *module.* A module is a self-referential scope, binding names to values. A binding of name to a value is a *definition.* Unlike *ML* [20], modules do not bind names to types. Type abbreviations may be used, as syntactic sugar.[3] Typing in *Jigsaw* is purely structural.

[3] In *ML* terms, only **type** declarations, not **datatype** declarations, are supported.

```
module
{  x = 0; y = 0;
   dist =  function(aPoint:{ x:Int, y:Int })
              {
                  sqrt(sqr((x - aPoint.x)) + sqr((y - aPoint.y)))
              }
} : {  define x:Int, y:Int, dist:{ x:Int, y:Int } → Real }
```

Figure 3: A module and its interface

Modules may include not only definitions, but *declarations.* A declaration gives the type of an attribute, but no value for it. Declarations are used to define "abstract classes". Modules may be nested. Every module has an associated *interface*, which gives the types (or interfaces, for nested modules) of all visible attributes of a module. The subtyping relation on interfaces is defined as interface equivalence. Two interfaces are equivalent if they have exactly the same attribute names, and the attributes have equivalent types or interfaces.

Modules have no free variables, and module operators do not require access to the definitions of their operands. This allows for separate compilation, including inheriting from separately compiled modules.

In the semantics of *Jigsaw*, all modules are modeled as generators. Module combination operators are then modeled as functions that manipulate generators, and return new generators as results. The operator definitions make use of the record operations introduced in section 2.2. All module operators employ the technique demonstrated above to manipulate self reference. Modules with declarations are modeled as inconsistent generators. Module operators can take inconsistent generators as operands and may return them as results. Viewing *Jigsaw* as an abstract datatype, generators are the hidden representation used for modules. Module operators rely on this representation, but users of the operators are isolated from it.

4.2 Instantiation

A module M is instantiated by the expression instantiate M. The result of this expression is an object. The module in Figure 3 can be instantiated into an object equivalent to P in Figure 1.

In an applicative language, all instantiations of a module are identical. Then why distinguish between a module and its instance ? The main reason is typing. It is extremely desirable to use instances polymorphically. On the other hand, module operations require exact knowledge of the type of their operands. Distinguishing modules from instances allows separate type rules to be given for each.

An alternative would be to introduce a new judgement into the type system, indicating that a value is exactly of some type, in addition to the ordinary judgement that a value has some type. This solution is more verbose. Also, our solution is more natural, since modules do denote a different type of value (generators) than objects (which denote records).

Another reason for keeping modules and instances distinct is that the decision to make module instances first class values (as in "Class-based" languages [32]) need not imply that modules themselves are first class values. If modules are identified with instances, the two decisions cannot be separated. We do not want the use of our approach to constrain language designers in this way. Subsection 5.1 discusses a language design where neither modules nor instances are values; subsection 5.2 refers to a language where instances are values, but modules are not; in *Jigsaw*, both modules and their instances are first class values (the fourth option, making modules values while instances are not, is self-contradictory).

The semantics of instantiation are as described in section 2. Instantiating a module is modeled by taking the fixpoint of the module's generator.

4.3 Combining Modules

Two modules may be combined using the merge operation. The result is a new module, in which all names declared in either of the inputs are declared. Name conflicts are not permitted, and result in a static error. Note that the merge operator does not provide any mechanism for resolving such conflicts. Other operators are used for this purpose. This is one example of how definitions are simplified in our approach.

The merge operator, $\|_g$, is defined below. It takes two generators as parameters and produces a new generator as a result. Note that self reference in the two generators is shared in the resulting generator.

$$\|_g = \lambda g_1.\lambda g_2.\lambda s.g_1(s) \|_r g_2(s)$$

$\|_g$ is commutative and associative. The merge operator is discussed further in the context of sharing (subsection 4.6).

4.4 Modification

One module may be modified by another. This is an asymmetric operation, in which one module overrides the other. This is supported by the override operation: M1 override M2. The override operator takes two modules and combines them. If an attribute is defined by both modules, then the type of the attribute in M2 must be a subtype of its type in M1. In that case, the value from M2 will appear in the result. Override is defined as:

$$\leftarrow_g = \lambda g_1.\lambda g_2.\lambda s.g_1(s) \leftarrow_r g_2(s)$$

$\leftarrow_g$ is associative and idempotent, but not commutative. $\leftarrow_g$ may also be derived from the combination of merge and restrict (defined below).

4.5 Name Conflict Resolution

Name conflicts can be resolved in several ways. One can explicitly choose one of the conflicting attributes in preference to all others. This eliminates the conflict, but requires that all modules share a common version of the attribute. This may not always be desired. Furthermore, the types of the conflicting attributes may be incompatible, in which case such sharing is impossible. Sharing is discussed in the following subsection.

An alternative is to eliminate the conflict by renaming. This is always possible, and all attributes remain available. The one drawback is that in a structure-based type system, attribute names are meaningful for subtyping, and renaming may adversely effect polymorphism.

The renaming operator changes the name of a single attribute.

M rename a to b

The effect is equivalent to a textual replacement of all occurrences of the attribute name a in M, by the name b. Attribute a must be declared by M, and b neither declared nor defined.

$$[a \leftarrow b]_g = \lambda g.\lambda s.g(s \leftarrow_r \{a = s._r b\})[a \leftarrow b]_r$$

if g defines a, else

$$[a \leftarrow b]_g = \lambda g.\lambda s.g(s \leftarrow_r \{a = s._r b\})$$

In the underlying record calculus, rename is derived from restrict and merge. Composing the generator versions of restrict and merge in this manner is not possible, due to the presence of self-reference. The type rule for rename must ensure that the attribute is renamed in the type of the result.

4.6 Sharing

When modules are merged in *Jigsaw*, multiple definitions of an attribute give rise to errors. In contrast, multiple declarations of an attribute are shared, and are perfectly legal.

Consider the expression

$g_1 \parallel_g g_2 \textit{ where } g_1 = \lambda s.\{a = s.b + 2\}, g_2 = \lambda s.\{b = 5\}$

The generator g_1 represents an abstract class, with an attribute b that is declared but not locally defined. Attribute b is defined in g_2. The application of $\parallel_g$ will not cause difficulties, even though both operands have a b attribute. This reflects the intuition that while definitions must be unique, declarations may be duplicated. Of course, this is only valid as long as the declaration agrees with the definition. The definition must have a type that is a subtype of the declaration. Similarly, two declarations may clash, as long as they have a subtype in common. Existing object oriented languages that recognize the notion of "pure virtual" do not make this distinction, and treat identically all name clashes between classes being combined. In contrast, in *Jigsaw*, declarations can help specify sharing constraints among modules, at the granularity of attributes.

Sharing is facilitated by the restrict operator. The effect of a restrict operation is to eliminate the definition of an attribute, but retain its declaration. Unlike records, it is not generally possible to completely remove an attribute from a module, because the module may contain internal references to the attribute. Restrict creates an abstract class, by making an attribute "pure virtual". Therefore, abstract classes may be created "after the fact". The attribute being restricted must be defined by the argument module.

The restrict operation is defined below, and is associative.

$$\backslash_g a = \lambda g.\lambda s.g(s)\backslash_r a$$

When several modules are combined via merge, sharing of conflicting attributes may be specified by restricting all but one. This supports conflict resolution via explicit specification.

Project is a dual of restrict. Rather than specifying which attribute to remove, we specify which attributes to retain. A module, M, and a list of attributes, A, are the inputs to the project operation. Project requires that all names in A be defined by M. The semantic definition for project is

$$\pi_g\ A = \lambda g.\lambda s.g(s)\pi_r A$$

4.7 Restricting Modifications

The freeze operator accepts an attribute name, a, and a module as parameters, and produces a new module in which all references to a are statically bound.

Some languages support this using the notion of non-virtual attributes (static binding). Static binding can be achieved by simply not referencing an attribute through self. However, this does not allow for changing the status of a virtual attribute to non-virtual (e.g. as in *Beta* [17]). In addition, it complicates the model, since not all attributes are referenced in the same way - there are two kinds, declared differently. In our model, it is preferable to have only virtual attributes declared, and perform the change by means of an operator on generators. The attribute being frozen must be defined.

$$freeze\ a = \lambda g.Y(\lambda f.\lambda s.g(s \leftarrow_r \{a = f(s)._r a\}))$$

This definition deserves some discussion. The result is a generator, the fixpoint of a generator generating function, $q = \lambda f. \ldots$. The generator $Y(q)$ agrees with g, with the exception of its self reference to attribute a. Regardless of the value of s, all references to $s.a$

within the methods of $Y(q)$ are bound to $f(s).{}_r a = Y(q)(s).{}_r a$. When the fixpoint is taken again, all references to $s.{}_r a$ will be equal to $Y(Y(q)).{}_r a = Y(g).{}_r a$.

Freeze has a dual operation, freeze_all_except, that freezes all features of a module M, except those specified in the list A. The attributes listed in A must be defined by M.

$$freeze\_all\_except\ A = \lambda g.Y(\lambda f.\lambda s.g(s \leftarrow_r f(s) \leftarrow_r (s\pi_r A)))$$

Overriding s with $f(s)$, rather than just $\{a = f(s).{}_r a\}$, means that all defined attributes are being frozen. We then override again, with $s\pi_r A$, guaranteeing that the attributes in A will indeed get there values from s, and therefore still be subject to redefinition.

4.8 Attribute Visibility

Visibility control is implemented by means of the operations hide and show. M hide a eliminates a from the interface of M. The attribute a must be defined by M.

$$hide\ a = \lambda g.\lambda s.(freeze\ a)(g)(s)\backslash_r a$$

The *hide* operation involves freezing the attribute, so that all references to it will not be influenced by subsequent changes to self. In addition, the attribute must be removed from the result, and the type rule for hide must remove a from the type of the module.

Conversely, M show A hides everything except the specified attributes. All attributes listed in A must be defined by M.

$$show\ A = \lambda g.\lambda s.(freeze\_all\_except\ A)(g)(s)\pi_r A$$

The duality between show and hide is apparent in the use of π_r instead of $\backslash_r$, and in the use of freeze_all_except instead of freeze.

4.9 Access to Overridden Definitions

Access to overridden definitions is supported through the use of the copy-as operator. M copy a as b creates a copy of the a method, under the name b. The a method can now be overridden, while the old implementation remains available under the name b. M must not declare an attribute b, but must define a.

Consider Figure 4, which also demonstrates the use of mixins. The intent here is that the BorderMixin module modifies the Window module by adding a border, to be displayed around the window. This requires a new display routine, which first displays the window's body, and then surrounds it with a border. BorderMixin declares an unimplemented routine display-body, which is invoked within the display routine. Before overriding Window with BorderMixin, Window's display routine is copied as display-body.

```
BorderMixin =  module
   {  borderWidth = 5; borderColor = red;
     display =  function(dontCare: Unit)
                       {
                         displayBorder();
                         displayBody();
                       }
     displayBorder =  function(dontCare: Unit) { ... }
     displayBody : Unit → Unit;
   }
Window =  module
   {  x = 0; y = 0;
     display =  function(dontCare: Unit) { ... }
   }
BorderWindow = Window  copy display  as displayBody ← BorderMixin;
```

Figure 4: Using a Mixin

Note that renaming display to display-body in Window would be inappropriate. When display was modified by BorderMixin, references to display within Window would not be modified. Defining a display-body routine that called display and adding that to Window would yield an infinite recursion once the modification by BorderMixin was performed.

The definition is straightforward

$$copy\ a\ as\ b = \lambda g.\lambda s.let\ super = g(s)\ in\ super \parallel_r \{b = super._r a\}$$

5 Application and Implementation

In view of the difficulty of introducing new languages into widespread use, it is extremely valuable to be able to incorporate new linguistic developments in an evolutionary manner. Adding operators like those defined in this paper to existing languages is therefore an attractive possibility.

We discuss two options for achieving that goal. First, it is possible to add modules to a language that does not have them. Second, our framework can be applied to object oriented languages, to enhance their expressive power.

5.1 Adding Modules to Existing Languages

Many languages do not have adequate modularity constructs. These include widely used programming languages (e.g., *C* [16], *Pascal* [13]), as well as countless special-purpose and "little-languages" [2, Column 9], where the effort of designing specific mechanisms for modularity is difficult to justify, but which could still benefit from such mechanisms.

The simple notions of module and interface defined above are largely language independent. This is because we specify neither the value set used in definitions, nor the form of the types used in declarations. One requirement is that the language being "modularized" support recursion, since modules are mutually recursive scopes. For imperative languages, the operator definitions must be modified, but their essential character remains the same.[4]

Suppose we wish to define and manipulate modules consisting of statements in some programming language, L. The definitions in modules will bind names to denotable values of L. For example, if $L = C$, the denotable values will include C functions and variables. Declarations and module interfaces will bind names to L types (in fact, since modules may be nested, definitions may also bind names to modules, and declarations may bind names to interfaces). Again using C as our example, the typing rules for module operators will rely on C type equivalence as the subtyping relation $\leq$ mentioned above.

The resulting language is not object-oriented, since it does not support first class objects. Nevertheless, it employs inheritance. Inheritance supports module interconnection by combining self reference among modules, and, of course, allows existing code to be extended and modified.

A wide range of languages can be extended as described here. Many of these languages are dynamically typed. This restricts the degree of static interface checking possible. However, any language that is extended with *Jigsaw* style modules gains substantial benefits from encapsulation, separate compilation (for compiled languages), modifiability and the ability to define partially specified modules analogous to abstract classes.

5.2 Extending an Object-Oriented Language

We are currently implementing an upwardly compatible extension of *Modula-3* [5], incorporating most of the operators described in this paper. In this extension, the operators are applied not to the modules of *Modula-3* but to its classes (known as object types).[5]

Naturally, the full flexibility of *Jigsaw* is not supported. Still, the resulting language supports strong typing, multiple inheritance and mixins in a modular manner.

The implementation is efficient enough to fit into a practical programming language like *Modula-3*. *Modula-3* restricts subtyping by making it dependent on the order in which attributes are specified, and on the boundaries between constituent object types. These restrictions, coupled with the fact that our modules never have any free variables, lead to an implementation based upon a straightforward extension of standard dispatch table techniques. Each object type is represented by a dispatch table, and operations such as merge and override involve concatenation of dispatch tables. The tables include both pointers

[4]Imperative versions of these operators have been defined in [3].

[5]An early, less ambitious version of this work appeared in [4].

to code for method execution, and offsets within objects. Offsets are necessary because under multiple inheritance, instances of subclasses do not necessarily share a common prefix with instances of parent classes. The main subtlety lies in manipulating these offsets, since previously published schemes [18, 28, 10] cannot be used in the presence of mixins. The details of the implementation are beyond the scope of this paper, and are discussed in [3].

6 Related Work

6.1 Generator Operations

Many of the operators presented here were first proposed by Cook in [9]. There, a general mechanism for deriving generator operations from record operations was described. However, the operators defined by Cook were used to illustrate the principle of manipulating self-reference by means of generators. In modeling language constructs, more elaborate operators were used. In particular, it was necessary to introduce parametric abstractions called wrappers. These were later elevated to explicit language constructs called mixins in [4], and, independently, in [12].

The novelty here is in providing a comprehensive suite of operations, and making them explicit linguistic constructs. In addition, the uniform use of generators to model all definitional structures is new. The operator suite also includes new operations (namely hide, show, freeze, freeze-except and copy-as).

6.2 Mixins

This work grew out of an earlier study of mixin-based inheritance [4]. Some of the limitations of mixin based inheritance have been addressed here. These include the absence of fine-grain sharing, of renaming facilities and of a symmetric merge operation.

Until now, mixins have been modeled as parametric abstractions called wrappers. Cook used an operator combining a generator and a wrapper in his compositional semantics of inheritance [9]. This operator was also used by Hense [12]. In [4], the override operation was defined as a binary operation on wrappers, enabling composition of mixins. The main purpose of wrappers was to allow access to overridden definitions. The required functionality can be achieved using explicit operator for this purpose. This allows the use of generators instead of wrappers, simplifying definitions. This reflects our strategy of simplifying the structure and pushing more functionality into the operator set.

6.3 Mitchell

Mitchell, in [24], presented an extension to the *ML* module system that is in some ways similar to our work. Mitchell also chose to incorporate inheritance into a module language, an extension of the *ML* module system [20]. Some similar operations are supported, embedded in a more conventional syntax. Underlying both systems are denotational models involving the manipulation of self reference, and typing based on bounded quantification. There are many differences, however.

Central to this paper is the notion that inheritance itself can be used as a modularity mechanism. Inheritance is an essential part of the module language, "gluing" modules together by merging self-reference. Such a formulation of inheritance must preserve encapsulation. This contrasts with Mitchell's view of inheritance as "a mechanism for using one declaration in writing another" [sic]. Even though inheritance is part of the module system, it is not essential to it. Instead, the *ML* notions of structures and functors are used to define and interconnect modules. Some of the inheritance constructs defined in [24] violate encapsulation (*viz.* `copy except`, `copy only`). These constructs inherently require knowledge of the internal structure of the "parent" module.

A consequence of the semantics of `copy except`, `copy only` is that separate compilation is compromised. A parent module must always be compiled before its use, and any change to it requires recompilation of its heir modules [21]. We support inheriting from separately compiled modules without restriction.

Our approach has the benefits of simplicity and modularity. It does not rely upon dependent sums or products, or on multiple universes of types. It is explicitly formulated as an abstract data type for manipulating modules, where all functionality is supported by operators. Making the structure explicit makes it easier to apply the framework to a broad spectrum of languages. Language designers may easily add or modify operations as necessary. An expression based language also allows users to compose operations more freely.

Our framework supports abstract classes and mixins.[6] Mixins cannot be expressed in the framework of [24], and there is no explicit support for abstract classes (though the traditional device of giving dummy definitions for pure virtual methods is always available, with its concomitant disadvantages).

On the other hand, Mitchell's approach supports modules implementing abstract data types. This allows for combining traditional algebraic (or higher order) data types with object-oriented formulations. Our model supports only the pure object oriented approach. We would like to extend our framework with an analogous set of operators for abstract data types. However, we face technical difficulties related to the typing of existential data types.

A related issue is our use of structural subtyping, in contrast to "name-based" subtyping in [24]. Both forms are useful; currently, we focus on structural subtyping, which is more appropriate between different modules or programs [6].

Finally, unlike [24], we give semantic definitions of all operations.

[6] Abstract classes are mentioned in [24], but only as substitutes for interfaces.

7 Conclusion

We have presented a collection of operations on modules, that supports a uniquely wide range of object oriented programming techniques, including frameworks, multiple inheritance and mixins, as well as separate compilation, all in a type safe and encapsulated manner. The operations are based on a uniform representation of modules as generators. This representation, together with the operations, define modules as an abstract data type.

Module operations are based on the novel notion that inheritance is productively viewed as a mechanism for *modular* program composition. As such, inheritance is independent of the notion of first-class instances, with which it is usually associated.

The operations can be incorporated into a variety of languages, are semantically well defined, and efficiently implementable.

The operations define a module manipulation language that is applicative and expression-oriented, rather than statement oriented. For such a language, realistic programming features can be effectively modeled using simple semantic constructs, such as generators. The language is itself modular, allowing for easier extension, modification and experimentation. Finally, we believe such a language is also easier to learn, use and reason about.

References

[1] America, P. A parallel object-oriented language with inheritance and subtyping. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990), pp. 161–168.

[2] Bentley, J. L. *More Programming Pearls*. Addison-Wesley, Reading, Massachusetts, 1988.

[3] Bracha, G. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1991. In preparation.

[4] Bracha, G., and Cook, W. Mixin-based inheritance. In *Proc. of the Joint ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications and the European Conference on Object-Oriented Programming* (Oct. 1990).

[5] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kalsow, B., and Nelson, G. Modula-3 report (revised). Tech. Rep. 52, Digital Equipment Corporation Systems Research Center, Dec. 1989.

[6] Cardelli, L., Donahue, J., Jordan, M., Kalsow, B., and Nelson, G. The Modula-3 type system. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Jan. 1989), Association for Computing Machinery, pp. 202–212.

[7] Cardelli, L., and Mitchell, J. C. Operations on records. Tech. Rep. 48, Digital Equipment Corporation Systems Research Center, Aug. 1989.

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

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

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

[11] Harper, R., and Pierce, B. A record calculus based on symmetric concatenation. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Jan. 1991), pp. 131–142.

[12] Hense, A. V. Denotational semantics of an object oriented programming language with explicit wrappers. Tech. Rep. A 11/90, Fachbereich Informatik, Universitaet des Saarlandes, Nov. 1990.

[13] Jensen, K., and Wirth, N. *Pascal User Manual and Report*, second ed. Springer-Verlag, 1978.

[14] Kamin, S. Inheritance in Smalltalk-80: A denotational definition. In *Proc. of the ACM Symp. on Principles of Programming Languages.* Association for Computing Machinery, 1988, pp. 80–87.

[15] Keene, S. E. *Object-Oriented Programming in Common Lisp.* Addison-Wesley, 1989.

[16] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.

[17] Kristensen, B. B., Madsen, O. L., Moller-Pedersen, B., and Nygaard, K. The Beta Programming Language. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 7–48.

[18] Krogdahl, S. An efficient implementation of Simula classes with multiple prefixing. Tech. Rep. 83, Institute for Informatics, University of Oslo, 1984.

[19] Linton, M. A., Calder, P. R., and M.Vlissides, J. InterViews: A C++ graphical interface toolkit. Tech. Rep. CSL-TR-88-358, Stanford University, July 1988.

[20] MacQueen, D. Modules for standard ML. In *Proc. of the ACM Conf. on Lisp and Functional Programming* (Aug. 1984), pp. 198–207.

[21] Madhav, N., September 1991. Personal communication.

[22] Manna, Z. *The Mathematical Theory of Computation.* McGraw-Hill, 1974.

[23] Meyer, B. *Object Oriented Software Construction.* Prentice-Hall International, Hertfordshire, England, 1988.

[24] Mitchell, J., Meldal, S., and Madhav, N. An extension of standard ML modules with subtyping and inheritance. In *Proc. of the ACM Symp. on Principles of Programming Languages* (Jan. 1991), pp. 270–278.

[25] Reddy, U. S. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM Conf. on Lisp and Functional Programming* (1988), pp. 289–297.

[26] Remy, D. Typechecking records and variants in a natural extension to ML. In *Proc. of the ACM Symp. on Principles of Programming Languages* (1989), pp. 77–88.

[27] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An introduction to Trellis/Owl. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 9–16.

[28] Stroustrup, B. Multiple Inheritance for C++. In *European Unix User Group Spring Conference* (May 1987).

[29] Ungar, D., and Smith, R. SELF: The power of simplicity. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (Oct. 1987).

[30] Vlissides, J., and Linton, M. Unidraw: A framework for building domain-specific graphical editors. Tech. Rep. CSL-TR-89-380, Stanford University, July 1989.

[31] Wand, M. Type inference for record concatenation and multiple inheritance. In *Proc. IEEE Symposium on Logic in Computer Science* (1989), pp. 92–97.

[32] Wegner, P. The object-oriented classification paradigm. In *Research Directions in Object-Oriented Programming.* MIT Press, 1987, pp. 479–560.

[33] Weinand, A., Gamma, E., and Marty, R. ET++ - an object-oriented application framework in C++. In *Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1988), pp. 46–57.