

Mixin-based Inheritance

Gilad Bracha*

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

William Cook

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303-0969

Abstract

The diverse inheritance mechanisms provided by Smalltalk, Beta, and CLOS are interpreted as different uses of a single underlying construct. Smalltalk and Beta differ primarily in the direction of class hierarchy growth. These inheritance mechanisms are subsumed in a new inheritance model based on composition of *mixins*, or abstract subclasses. This form of inheritance can also encode a CLOS multiple-inheritance hierarchy, although changes to the encoded hierarchy that would violate encapsulation are difficult. Practical application of mixin-based inheritance is illustrated in a sketch of an extension to Modula-3.

1 Introduction

A variety of inheritance mechanisms have been developed for object-oriented programming languages. These systems range from classical Smalltalk single inheritance [8], through the safer prefixing of Beta [12, 10], to the complex and powerful multiple inheritance combinations of CLOS [6, 9]. These languages have similar object models, and also share the view that inheritance is a mechanism for incremental programming. However, they differ widely in the kind of incremental changes supported.

In Smalltalk, subclasses can add additional methods or replace existing methods in the parent class. As a result, there is no necessary relationship between the behavior of instances of a class and the instances of its subclasses. The subclass methods can invoke any of the original superclass methods via **super**.

In Beta, a *subpattern* (subclass) definition is viewed as an extension of a previously defined *prefix* pattern. As

in Smalltalk, new methods may be defined. However, prefix methods cannot be replaced; instead, the prefix may use the command **inner** to invoke the extended method code supplied by the subpattern. Given that the code in a prefix is executed in any of its extensions, Beta enforces a degree of behavioral consistency between a pattern and its subpatterns.

The underlying mechanism of inheritance is the same for Beta and Smalltalk [3]. The difference between them lies in whether the extensions to an existing definition have precedence over and may refer to previous definitions (Smalltalk), or the inherited definition has precedence over and may refer to the extensions (Beta). This model shows that Beta and Smalltalk have inverted inheritance hierarchies: a Smalltalk subclass refers to its parent using **super** just as a Beta prefix refers to its subpatterns using **inner**.

In the Common Lisp Object System (CLOS) and its predecessor, Flavors [13], multiple parent classes may be merged during inheritance. A class's ancestor graph is linearized so that each ancestor occurs only once [7]. With standard method combination for primary methods, the function `call-next-method` is used to invoke the next method in the inheritance chain.

CLOS supports mixins as a useful technique for building systems out of mixable attributes. A *mixin* is an abstract subclass; i.e. a subclass definition that may be applied to different superclasses to create a related family of modified classes. For example, a mixin might be defined that adds a border to a window class; this mixin could be applied to any kind of window to create a bordered-window class. Semantically, mixins are closely related to Beta prefixes.

Linearization has been criticized for violating encapsulation, because it may change the parent-child relationships among classes in the inheritance hierarchy [16, 17]. But the mixin technique in CLOS depends directly upon linearization and modification of parent-child relationships. Rather than avoid mixins because they violate encapsulation, we argue that linearization is an implementation technique for mixins that obscures their true nature as abstractions.

By modest generalization of the inheritance models

*Supported by grant CCR-8704778 from the National Science Foundation.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1990 ACM 089791-411-2/90/0010-0303...\$1.50

in Smalltalk and Beta, a form of inheritance based on composition of mixins is derived. Mixin-based inheritance supports both the flexibility of Smalltalk and the security of Beta. It also supports the direct encoding of CLOS multiple inheritance hierarchies without duplication of subclass definitions. However, since the hierarchy is encoded as an explicit collection of linearized inheritance chains rather than as a single inheritance graph, some changes to the hierarchy (especially if they might violate Snyder's notion of encapsulation) cannot easily be made.

Section 2 discusses the single-inheritance languages Smalltalk and Beta and shows that they support very different uses of a single underlying construct. Section 3 analyzes multiple inheritance and linearization in CLOS, with special focus on support for mixins. Section 4 presents a generalized inheritance mechanism that supports the style of inheritance in Beta, Smalltalk, and CLOS, with explicit support for mixins. In Section 5 we sketch an extension to Modula-3 that illustrates the use of generalized inheritance. Finally, Section 6 summarizes our conclusions.

2 Single Inheritance Languages

2.1 Smalltalk Inheritance

Inheritance in Smalltalk is a mechanism for incremental derivation of classes. Smalltalk inheritance was adapted from Simula [5, 14], and serves as the prototypical inheritance mechanism. The primary subtlety in the process of inheritance is the interpretation of the special variables **self** and **super**. **Self** represents recursion, or self-reference, within the object instance being defined. The interpretation of **self** has been addressed in previous work [3, 4, 15]; in this paper we focus on the interpretation of **super**. Consider the following pair of Smalltalk classes.

```
class Person
  instance variables: name
  method: display
    name display

class Graduate
  superclass: Person
  instance variables: degree
  method: display
    super display. degree display
```

The class **Person** defines a name field and a method for displaying the name. The subclass **Graduate** extends the display method to include the person's academic degree. For example, a graduate with name "A. Smith" and degree "Ph.D." would respond to the display method

by invoking the **Graduate display** method, which invokes the **Person display** method using **super display** to display the name, and then displays the degree. The net effect would be to print "A. Smith Ph.D.". It would also be possible to prefix the name, as in the case of titles like "Dr.", by printing the title before calling **super**.

The subclass **Graduate** specifies only how **Graduates** differ from **Persons** [19]. This difference may be indicated explicitly as a *delta*, or set of changes. In this case the set of changes is simply the new display method. The original definition is also just a display method. When combined, the new display method replaces the original.

To formalize this process, objects are represented as records whose fields contain methods [1, 15, 18, 3]. The expression $\{a_1 \mapsto v_1, \dots, a_n \mapsto v_n\}$ represents a record with fields a_1, \dots, a_n and associated values v_1, \dots, v_n . The expression $r.a$ represents selection of field a from a record r . Record combination is a binary operator, \oplus , that forms a new record with the fields from its two arguments, where the value is from the *left* argument in case the same field is present in both records. For example, $\{a \mapsto 3, b \mapsto 'x'\} \oplus \{a \mapsto \text{true}, c \mapsto 8\}$ replaces the right value of a to produce $\{a \mapsto 3, b \mapsto 'x', c \mapsto 8\}$.

To interpret **super**, it is necessary for the delta, or modifications, to access the original method inherited from **Person**. This is achieved by supplying the parent class methods as a parameter to the delta. The resulting inheritance mechanism is an asymmetric combination of a parametric delta Δ and a parent specification P :

$$C = \Delta(P) \oplus P.$$

This definition is a form of single inheritance: P refers to the inherited parent while Δ is an explicit set of changes. The two occurrences of P do not indicate that it is instantiated twice, but that its information is used in two contexts: for the interpretation of **super** and to provide methods for the subclass. Suppressing the interpretation of hidden instance variables, the example above has the following form.

$$\begin{aligned} P &= \{\text{display} \mapsto \text{name.display}\} \\ \Delta(s) &= \{\text{display} \mapsto s.\text{display}, \text{degree.display}\} \\ \Delta(P) &= \{\text{display} \mapsto \text{name.display}, \text{degree.display}\} \end{aligned}$$

Although deltas were introduced to make specification of the inheritance mechanism more clear, deltas are not independent elements of a Smalltalk program; they cannot stand on their own and are always part of a subclass definition, which has an explicit parent class.

In Smalltalk a subclass of **Person** may completely replace the display method with, for example, a routine that displays the time of day. In Smalltalk inheritance, the subclass is in control: there is no way to define **Person** so that it forces subclasses to invoke its display method as part of their display operation.

2.2 Beta Inheritance

Inheritance in Beta is designed to provide security from replacement of a method by a completely different method. Inheritance is supported in Beta by *prefixing* of definitions. Beta employs a single definitional construct, the *pattern*, to express types, classes and methods. As this generality can be confusing, we use a simpler syntax that distinguishes among the different roles¹. The example given above is easily recoded in Beta:

```
Person: class
(# name : string;
 display: virtual proc
   (# do name.display; inner #);
#);

Graduate: class Person
(# degree: string;
 display: extended proc
   (# do degree.display; inner #);
#);
```

The definition of Graduate is said to be *prefixed* by Person. Person is the *superpattern* of Graduate, which, correspondingly, is a *subpattern* of Person. Display is declared to be **virtual**, which means that it may be extended in a subpattern. This does not mean that it may be arbitrarily redefined, as in most object-oriented languages.

The behavior of the **display** method of a Person is to display the name and then perform the **inner** statement. For a plain Person instance, which has no inner behavior, the **inner** statement is a null operation (i.e. skip or no-op). When a subpattern of Person is defined, the inner statement will execute the corresponding **display** method in the subpattern.

The subpattern Graduate extends the behavior of the Person **display** method by supplying inner behavior. For a Graduate instance G, the initial effect of G.**display** is the same as for a Person: the original method from Person is executed. After the name is displayed, the inner procedure supplied by Graduate is executed to display the graduate's degree. The use of **inner** within Graduate is again interpreted as a no-op. It only has an effect if the **display** method is extended by a subpattern of Graduate. It is impossible to arrange for printing a title, like "Dr.", before the name by inheriting Person; this is because the choice to invoke **inner** after the name has been built into the Person **display** method. In Beta prefixing, the prefix controls the behavior of the result.

The interpretation of the Person pattern is as a parametric definition of attributes, P' . The parameter rep-

¹This syntax is used by the implementors of Beta for tutorial purposes [11].

resents any inner definitions supplied by subpatterns. For an instance of Person, the inner part of P' is bound to the record of null methods: $P'(\emptyset)$.

A subpattern specifies additional attributes which may also refer to any further inner behavior in later subpatterns. If the attributes defined in the subpattern are specified by Δ' , then the result of prefixing by P' is the following composition:

$$C'(\mathbf{inner}) = P'(\Delta'(\mathbf{inner}) \oplus \mathbf{inner}) \oplus \Delta'(\mathbf{inner})$$

This means that the interpretation C' of the subpattern, when supplied an **inner** parameter, is the result of combining the superpattern P' specification with the changes specified by Δ' . By applying P' to $\Delta'(\mathbf{inner}) \oplus \mathbf{inner}$, the inner specification of P' is bound to the fields of the subpattern combined with any further fields supplied by later subpatterns. The prefix methods take precedence over the suffix. In the example above, the equation for C' is greatly simplified by examining the actual uses of **inner**:

$$\begin{aligned} P'(i) &= \{ \text{display} \mapsto \text{name.display}, i.\text{display} \} \\ \Delta'(i) &= \{ \text{display} \mapsto \text{degree.display}, i.\text{display} \} \\ C'(i) &= \{ \text{display} \mapsto \text{name.display}, \\ &\quad \text{degree.display}, \\ &\quad i.\text{display} \} \end{aligned}$$

This formulation does not directly encode the restriction that **inner** within a method m can refer only to the suffix method named m . In this sense **inner** is less general than Smalltalk's **super** construct, but the restriction is justified by the desire for security. An alternative formalization that captures this restriction involves representing each method as a function of its **inner** behavior [3]. Prefixing is then defined as combination of records such that duplicated fields are *composed*. Before calling a method it must be applied to a null command so that **inner** will have no effect. The resulting formalism is equivalent to the one given above, under the condition that the fields of P' and Δ' only access corresponding fields of **inner**.

2.3 Comparing Smalltalk and Beta

The inheritance mechanisms of Smalltalk and Beta are different orientations of a common underlying mechanism. The underlying mechanism is a non-associative binary operator, \triangleright , that performs application of **super/inner** and combination of attributes.

$$\Delta \triangleright P = \Delta(P) \oplus P$$

The relationship between Beta and Smalltalk is demonstrated by comparing the interpretations of inheritance in the two languages. The behavior of a subclass instance can be compared concisely in this framework.

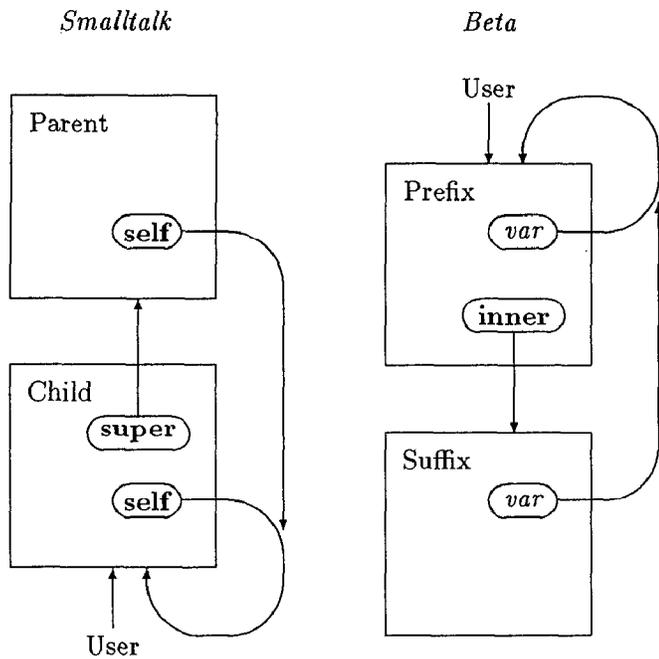


Figure 1: Inverse hierarchies in Smalltalk and Beta.

$$\begin{array}{ll}
 C = \Delta \triangleright P & \text{Smalltalk} \\
 C'(\emptyset) = P' \triangleright \Delta'(\emptyset) & \text{Beta}
 \end{array}$$

In these equations, Δ represents the new explicit information supplied by the subclass/subpattern, while P represents the original attributes contributed by the superclass/superpattern. The combination operator \triangleright favors values from its left argument in case of a duplicate attribute.

It is clear that the mechanism of inheritance is the same; only the direction of growth is different. In Smalltalk the new attributes are favored and may replace inherited ones; in Beta the original attributes are favored. Beta inheritance works in the opposite direction from inheritance in most object oriented languages, due to this role reversal between superpatterns/subpatterns and subclasses/superclasses. Figure 1 show this inversion by illustrating the semantic relationships in Smalltalk and Beta when a superclass is placed above one of its subclasses. The figure includes the interpretation of self-reference, which is implicit in Beta variable (*var*) references [3]. Neither direction of inheritance is able to express the other, and each has its advantages and disadvantages.

3 Multiple Inheritance and Mixins

3.1 CLOS Inheritance

CLOS supports a rich mechanism for multiple inheritance. Although there are several significant aspects of CLOS inheritance, we focus only on standard method combination and primary methods. Here is the example given above, recoded in CLOS.

```

(defclass Person () (name))

(defmethod display ((self Person))
  (display (slot-value self 'name)))

(defclass Graduate (Person) (degree))

(defmethod display ((self Graduate))
  (call-next-method)
  (display (slot-value self 'degree)))

```

The `defclass` construct includes the name of the new class, a list of its superclasses, and a list of its instance variables. The argument list of the `defmethod` form defines the class on which the method is defined. Simple but effective method combination is supported by `call-next-method`, which plays the role of `super` in Smalltalk. But like `inner` in Beta, `call-next-method` provides access only to the next method in the inheritance chain with the same message selector.

A CLOS class may inherit from more than one parent. As a result, a given ancestor may be inherited more than once. For example, the following classes result in `Person` being inherited twice by `Research-Doctor`.

```

(defclass Doctor (Person) ())

(defmethod display ((self Doctor))
  (display "Dr. ")
  (call-next-method))

(defclass Research-Doctor (Doctor Graduate) ())

```

If care is not taken, the `display` method of `Person` will be executed twice, and a `Research-Doctor` will display as "Dr. A. SmithA. Smith Ph.D.". To remedy this situation, CLOS *linearizes* the ancestor graph of a class to produce an inheritance list in which each ancestor occurs only once. The graph of ancestors of `Research-Doctor` is linearized to `Research-Doctor`, `Doctor`, `Graduate`, `Person`. This also solves the problem of method invocation order, because ancestor classes are placed in a linear order.

Each collection of method definitions may invoke methods later in the linearized sequence via `call-next-method`. If the specification of parents P_1, \dots, P_n is

given by $\Delta_1, \dots, \Delta_n$, then the interpretation C of the subclass is defined by iteration of the inheritance operator over the list.

$$C = \Delta_1 \triangleright (\Delta_2 \triangleright (\dots \triangleright (\Delta_n \triangleright \emptyset)))$$

Each specification in the list is applied to the result of the previous specification and combined with it. The more complex method combination mechanisms of CLOS can also be modeled in this framework. For example, if before and after methods were distinguished then the base class, whose methods would be called after all other methods, could arrange for the last before method to call the primary method, and the last primary method to call the after method.

The process of linearization has been criticized for violating encapsulation [17]. One argument is that the relationship between a class and its declared parents may be modified during linearization. This is demonstrated by the example above, where in the linearization the class `Graduate` is placed between `Doctor` and `Person`, in contradiction of the explicit declaration of `Doctor` that it inherits directly from `Person`. Only by being aware of the entire class hierarchy can the programmer foresee this.

Using linearization, a CLOS multiple inheritance hierarchy is reduced to a collection of inheritance chains, each of which can be interpreted using single inheritance. However, a slight change to the original CLOS hierarchy may result in a very different collection of inheritance chains. This is especially true if the changes violate Snyder's notion of encapsulation, as when a base class is factored into two classes, because one of the new factors may interact with other classes during linearization. A less severe problem is that a given class may occur in many chains, so if the collection was implemented in a single-inheritance language, subclasses would have to be duplicated. In order to eliminate this duplication, the single-inheritance model must be generalized to allow explicit naming and reuse of the deltas defined by subclasses.

3.2 Mixin Programming

In this section we discuss a common programming technique used in CLOS, called *mixins*. A mixin is an abstract subclass that may be used to specialize the behavior of a variety of parent classes. It often does this by defining new methods that perform some actions and then call the corresponding parent methods. Mixins are very similar to the deltas introduced informally in Section 2.1. For example, the notion of a graduate degree as part of a name can be written as an independent mixin.

```
(defclass Graduate-mixin () (degree))
```

```
(defmethod display ((self Graduate-mixin))
  (call-next-method)
  (display (slot-value self 'degree)))
```

This example illustrates a characteristic of mixins: they invoke `call-next-method` even though they do not appear to have any parents. This would obviously lead to an error if an instance of a mixin were created. Linearization places the mixin into an inheritance chain before other classes that support the method. This occurs in the new definition of `Graduate`: because `Graduate-mixin` is listed before `Person`, the `Person` display method will be invoked by `Graduate-mixin` display.

```
(defclass Graduate (Graduate-mixin Person) ())
```

In CLOS, mixins are simply a coding convention and have no formal status. Although locally unbound uses of `call-next-method` are a clear indication that a class is a mixin, the concept has no formal definition, and any class could be used as a mixin if it contributes partial behavior.

Using `Graduate-mixin` it is now possible to define different kinds of classes that have "graduated" behavior. In this example, the guard dog might have an obedience school degree.

```
(defclass Guard-Dog (Graduate-mixin Dog) ())
```

Neither Smalltalk nor Beta fully support mixins. In Smalltalk, the effect of a mixin can be achieved by explicitly creating subclasses and copying the mixin code into the subclass, preventing code-sharing and abstraction. In Beta, an individual class closely resembles a mixin. However, it cannot be attached to independently-defined classes. Instead, the client class must be built with the mixin as a prefix. If a family of mixed versions of a given class is needed, then the entire class must be copied for each prefixed mixin. Thus, in Smalltalk the mixin must be copied, while in Beta the base class must be copied. This is consistent with our analysis of the direction of growth in Beta and Smalltalk.

Mixin programming takes advantage of multiple inheritance in a subtle and unintuitive way: mixins depend upon linearization to place them in an appropriate location in the inheritance chain and to insert other classes between the mixin and its parents. When mixins are viewed as abstract subclasses, or class definitions parameterized by their parents, it is clear that linearization plays the role of application, by binding the mixin's formal parent parameter to a specific class. This process of abstraction and application can be made more explicit by generalizing the inheritance mechanism common to Smalltalk and Beta.

4 Inheritance as Composition of Mixins

Mixins are the basis for a compositional inheritance mechanism that generalizes Smalltalk and Beta, while supporting the encoding of an encapsulated version of a CLOS multiple inheritance hierarchy. The basic idea of the generalization is to take mixins as the primary definitional construct. Inheritance is then formulated as composition of mixins. New attributes may be composed in either the Smalltalk or Beta style (either overwriting or extending). Since mixins and composition are explicit, there is no need for implicit linearization: a programmer would explicitly select the order of all mixin components. If a component is composed more than once it will appear as multiple copies in the result; duplication is avoided by explicitly applying two components to a shared parent.

The mixin composition operator, \star , is the Beta inheritance operator, but is used in a slightly more general form. Mixin composition takes two mixins as parameters, and returns a new mixin as a result.

$$M_1 \star M_2 = \text{fun}(i) M_1(M_2(i) \oplus i) \oplus M_2(i)$$

In case of conflict, \star gives priority to the first parameter. In M_1 , **super/inner** is bound during the inheritance operation to M_2 . In M_2 , **super/inner** is bound to the formal parameter i of the result. Assuming the basic attribute combination operator \oplus is associative, \star is associative. In addition, if \oplus were commutative, then \star would be commutative.

Ordinary classes are viewed as degenerate mixins that do not make use of their **inner/super** parameter. Mixins thereby generalize Smalltalk classes, Beta patterns and CLOS style mixins. Abstract classes are viewed as mixins that refer to fields not defined in **self**. A mixin is *complete* if it does not refer to its parent parameter, and defines all fields that it refers to in itself. Otherwise, it is *partial*. Only complete mixins may be instantiated meaningfully.

5 Application to an Existing Language

5.1 Choice of Language

We have chosen Modula-3 [2] as a basis for an extension incorporating mixin-based inheritance. Modula-3 is well suited for such an extension, because it supports single inheritance and is strongly typed. Single inheritance naturally generalizes to mixin-based inheritance. Strong typing provides a framework in which mixins can be used safely and efficiently.

5.2 Modula-3 Inheritance

Modula-3 supports inheritance via object types. Object types are roughly analogous to classes in most object-oriented languages. An example of object types in Modula-3 is

```
type Person =
  object name: string2
  methods display() := displayPerson
end;

type Graduate = Person
  object degree: string
  methods display := displayGraduate
end;

procedure displayPerson(self: Person) =
  begin
    self.name.display();
  end displayPerson;

procedure displayGraduate(self: Graduate) =
  begin
    Person.display(self);
    self.degree.display()
  end displayGraduate;
```

In the example, **Person** defines an instance variable **name** and a method **display**. The method is defined by providing a name, followed by a *signature*, or formal parameter list. In this case, the signature is empty. The method is then assigned a value, which is a separately defined procedure, **displayPerson**. If **o** is an object of type **Person**, **o.display()** is interpreted as **displayPerson(o)**.

The definition of **Graduate** has two parts: A preexisting definition, **Person**, and a modification given by the **object ... methods ... end** clause. **Graduate** is a *subtype* of **Person**, which is its *supertype*. **Graduate** inherits from **Person**, but includes a *method override* for **display**. The method override names the method being overridden, and then assigns a new value to it, namely **displayGraduate**. A signature is not given, since it will always be identical to the signature of the corresponding method in the supertype. The overridden methods of **Person** may be referred to by **Graduate** through the syntax **Person.methodname**. This is similar to **super** in Smalltalk, but more general.

An **object ... methods ... end** clause corresponds to the notion of delta discussed above. As in Smalltalk, deltas may not be defined independently of a parent. The following section presents an extension, whereby such deltas become independent constructs.

²Modula-3 uses TEXT for character strings. However, we will assume that `string` has been defined.

5.3 Extending Modula-3

We extend Modula-3 by generalizing object types to mixins. A mixin may be an explicit modification, of the form **object ... methods ... end**. Alternately, a mixin may be the result of combining two previously defined mixins.

```
Mixin = object ... methods ... end |
        Mixin1 * Mixin2
```

The concrete syntax used in the examples below, differs from the notation used until now in three respects. First, the order of the operands of the mixin operator is reversed, so that priority is given to the right hand operand. Second, the mixin operation is not written explicitly, but is implicit between each pair of mixins in a mixin definition. Finally, an optional **super** clause is added to modifications. The first two changes reflect existing Modula-3 syntax, where a modification is written to the right of a base definition, with no composition operator in between. Adopting these changes helps make the extension upwardly compatible. The third change is for typechecking purposes, as explained below. The resulting syntax is

```
Mixin' = object ... methods ... end |
         object ... methods ... super ... end |
         Mixin'2 Mixin'1.
```

The following is equivalent to the CLOS mixin example given above.

```
type GraduateMixin =
  object degree: string
  methods display := displayGraduateMixin
  super display() := No_Op
end;

mixin_procedure
displayGraduateMixin(self: GraduateMixin) =
  begin
    super.display()
    self.degree.display();
  end displayGraduateMixin;

procedure No_Op(self: root) = begin end No_Op;

type Graduate = Person GraduateMixin;
```

Since GraduateMixin is defined independently of any parent, the signature of `display` cannot be inferred, and must be given in a special **super** clause. Similarly, `display`'s overridden value is not known, but may be assigned a default. In this case, the default value is `No_Op`,

a procedure that will work on any type, since it is defined on `root`, the root of the type hierarchy. `DisplayGraduateMixin` refers to the overridden `display` method through the pseudo-variable **super**, using the syntax `super.methodname`. Procedures that reference **super** are distinguished, using the keyword **mixin_procedure**.

In the code above, `GraduateMixin` plays a role similar to a subclass in Smalltalk. Reversing `GraduateMixin`'s position in the definition of `Graduate` reverses its role to that of a Beta subpattern. This is illustrated below, where `PersonMixin` functions as superpattern.

```
type PersonMixin =
  object name: string
  methods display := displayPersonMixin
  super display() := No_Op
end;
```

```
type Graduate = GraduateMixin PersonMixin;
```

```
mixin_procedure
displayPersonMixin(self: PersonMixin) =
  begin
    self.name.display();
    super.display()
  end displayPersonMixin;
```

`PersonMixin` is in control, when combined with `GraduateMixin`. `Graduate.display()` invokes `displayPersonMixin`, where `super.display()` calls `displayGraduateMixin`. In `displayGraduateMixin` `super.display` will use the default value, `No_Op`, corresponding to an empty **inner** clause in Beta.

The examples above have an important advantage over their Smalltalk and Beta counterparts; all parts of the definition can be reused, without being textually copied.

As a final example, we recode our earlier CLOS multiple inheritance example:

```
type Doctor =
  object
  methods display := displayDoctor
  super display() := No_Op
end;
```

```
type ResearchDoctor =
  PersonMixin GraduateMixin Doctor;
```

```
mixin_procedure displayDoctor(self: Doctor) =
  begin
    display("Dr. ");
    super.display()
  end displayDoctor;
```

Note how the linear sequence of definitions is given explicitly, without reliance on linearization.

5.3.1 Typing

This section presents the typing rules for mixins in the Modula-3 extension. The typing of mixins has not been addressed in prior work, since mixins have not been introduced into a strongly typed language before.

Type identity is defined as in Modula-3. Two types are identical iff their expanded definitions are identical. The subtyping relation on mixins, $T \ll S$ (read T is a subtype of S , or S is a supertype of T) is defined as follows:

1. **object ...end** \ll **root**. All mixins are subtypes of **root**.
2. If $T_1 = T_2$ T_3 then $T_1 \ll T_2$ and $T_1 \ll T_3$, where the $=$ sign denotes type identity.
3. \ll is reflexive and transitive.

For example `ResearchDoctor` \ll `Doctor`, as well as `ResearchDoctor` \ll `GraduateMixin`, `ResearchDoctor` \ll `PersonMixin`. What is less obvious is that if

```
PGMixin = PersonMixin GraduateMixin;
```

then `ResearchDoctor` \ll `PGMixin`. This follows from the fact that `ResearchDoctor` = `PGMixin Doctor` by the definition of type identity. Recall that the mixin combination operator, \star , is associative. This is reflected in the subtyping rules.

Additional rules for mixin composition include :

- A method should be mentioned in the **super** clause of a type, if it has been overridden. In practice, a method override may be omitted from the **super** clause, if its signature can be inferred from context. An example might be the definition of `Graduate` in section 5.2. This exception is made for compatibility with existing Modula-3 code.
- The pseudo-variable **super** may only be used in procedures declared as *mixin procedures*. The procedure's first parameter must be of a type that includes a method override for every method referenced through **super**.
- A mixin procedure can be invoked only as a method. This guarantees that there is no way for the overridden methods of a mixin instance to be accessed from outside the instance.

All rules given in this section can be statically enforced. This is a necessary condition for safety, and for an efficient implementation. These rules are specific to Modula-3, and an extension of another language

would certainly differ in many details. However, the basic strategy of generalizing object types (or classes, in other languages) to mixins is fundamental to any such extension.

6 Conclusion

The inheritance mechanisms in the languages Beta, Smalltalk, and CLOS are representative of three different design choices for inheritance. Although the mechanisms are, on the surface, very dissimilar, we identify a common underlying structure. This underlying mechanism combines two sets of attributes such that duplicate attribute definitions are given a value from one set, where the value that is used may refer to the value being eliminated.

Beta and Smalltalk both support single inheritance, in which a single existing definition may be extended with new attributes. In Smalltalk the new attributes may replace existing attributes, which can be accessed directly via **super**. In contrast, Beta prohibits the extensions from replacing existing attributes; a new definition for an existing attribute has an effect only by being invoked when the original attribute executes the **inner** command. These two mechanisms have inverse relationships between inherited definition and extensions: the Smalltalk *subclass/superclass* relationship is analogous to the *superpattern/subpattern* relationship in Beta, where **super** is analogous to **inner**.

CLOS supports multiple inheritance, in which several existing definitions may be combined together. To avoid duplication of components, CLOS linearizes the set of primitive components in the inherited definitions. This linear list of components is then combined together by the same mechanism underlying Smalltalk and Beta: attribute values appearing earlier in the list replace (and may refer to) those appearing later. One disadvantage of linearization is that the relationships between primitive components may be changed. However, we show that linearization is the basis for the useful technique of mixin programming.

We propose that the underlying inheritance mechanism, which appears in two different restricted forms in Beta and Smalltalk, and is hidden behind linearization in CLOS, be used as the foundation for a general inheritance construct. In this formulation, mixins become the basic definitional construct, and inheritance is interpreted as mixin composition. Since the composition of mixins is explicit, the problem of linearization violating encapsulation does not arise.

It does not appear difficult to extend Beta and Smalltalk to support mixins and generalized inheritance. This work could be applied to CLOS, which already supports mixins, to make them more explicit and

less susceptible to encapsulation problems. A sketch of an extension to Modula-3 illustrates a possible design for mixins and generalized inheritance.

References

- [1] CARDELLI, L. A semantics of multiple inheritance. In *Semantics of Data Types* (1984), vol. 173 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 51-68.
- [2] 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.
- [3] COOK, W. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.
- [4] COOK, W., AND PALSBERG, J. A denotational semantics of inheritance and its correctness. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1989), pp. 433-444.
- [5] DAHL, O.-J., MYHRHAUG, B., AND NYGAARD, K. The SIMULA 67 common base language. Tech. rep., Norwegian Computing Center, Oslo, Norway, 1970. Publication S-22.
- [6] DEMICHEL, L., AND GABRIEL, R. The Common Lisp Object System: An overview. In *European Conference on Object-Oriented Programming* (June 1987), pp. 151-170.
- [7] DUCOURNAU, R., AND HABIB, M. On some algorithms for multiple inheritance in object-oriented programming. In *European Conference on Object-Oriented Programming* (1987), pp. 243-252.
- [8] GOLDBERG, A., AND ROBSON, D. *Smalltalk-80: the Language and Its Implementation*. Addison-Wesley, 1983.
- [9] KEENE, S. E. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, 1989.
- [10] 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.
- [11] KRISTENSEN, B. B., MADSEN, O. L., MOLLER-PEDERSEN, B., AND NYGAARD, K. The Beta programming language - a Scandinavian approach to object-oriented programming, Oct. 1989. OOPSLA Tutorial Notes.
- [12] KRISTENSEN, B. B., MADSEN, O. L., MOLLER-PEDERSEN, B., AND NYGAARD, K. Abstraction mechanisms in the Beta programming language. *Information and Control* (1983).
- [13] MOON, D. A. Object-oriented programming with Flavors. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 1-8.
- [14] PAPAZOGLU, M. P., GEORGIADIS, P. I., AND MARITSAS, D. G. An outline of the programming language Simula. *Computer Language* 9, 2 (1984), 107-131.
- [15] 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.
- [16] SNYDER, A. Encapsulation and inheritance in object-oriented programming languages. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications* (1986), pp. 38-45.
- [17] SNYDER, A. Inheritance and the development of encapsulated software systems. In *Research Directions in Object-Oriented Programming*. MIT Press, 1987, pp. 165-188.
- [18] WAND, M. Type inference for record concatenation and multiple inheritance. In *Proc. IEEE Symposium on Logic in Computer Science* (1989), pp. 92-97.
- [19] WEGNER, P., AND ZDONIK, S. B. Inheritance as a mechanism for incremental modification. In *European Conference on Object-Oriented Programming* (1988), pp. 55-77.