Appears in the Proceedings of ISOTAS '93.

This version corrects a number of minor problems.

Traces (A Cut at the "Make Isn't Generic" Problem)

Gregor Kiczales
Xerox Palo Alto Research Center*

September 2, 1993

Abstract

Object-oriented techniques are a powerful tool for making a system end-programmer specializable. But, in cases where the system not only accepts objects as input, but also creates objects internally, specialization has been more difficult. This has been referred to as the "make isn't generic problem." We present a new object-oriented language concept, called traces, that we have used successfully to support specialization in cases that were previously cumbersome.

The concept of traces makes a fundamental separation between two kinds of inheritance in object-oriented languages: inheritance of default implementation – an aspect of code sharing; and inheritance of specialization, a sometimes static, sometimes dynamic phenomenon.

The Problem

Object-oriented programming languages have proven to be a powerful tool for making a variety of systems *end programmer specializable* [KL92]. That is, once the system has been written, compiled and distributed, another programmer can modify or extend its behavior. This technique has played a critical role in the reflection and meta-level architectures community, where it has been used to "open up" programming languages [BKK+86, Mae87, Coi87, Fer89, FJ89, MCD91, KdRB91, CI93], operating systems [YTY+89, YTM+91, YMFT91], and even window systems [Rao90, Rao91, Rao93].

A Simple Example

As an example of this style of using object-oriented techniques, consider a simple graphical editor that might be part of the standard toolset in a user interface library. It accepts

*3333 Coyote Hill Rd., Palo Alto, CA 94304; (415)812-4888; Gregor@parc.xerox.com.

a variety of graphical objects as input, displays them on the screen, allows the user to edit them, and then returns the modified objects.

Using existing object-oriented languages — and a healthy dose of careful design and documentation — it would be possible to craft this system so that an end-programmer could extend its behavior, perhaps by adding completely new kinds of graphical objects, or more likely by defining new kinds of objects with behavior that is a specialization of that provided by default kinds.

Such an editor might provide, as part of its standard library, a class called `<line>`[1], whose instances are line segments that can be moved and resized freely. Given this, one might imagine that an end-programmer might want to extend the system, by adding a new kind of line segment, with the specialized behavior that its slope cannot be changed. Assuming a simple documented protocol between the editor and line objects, consisting of messages to get the end points of a line (`end-1` and `end-2`), and a single message to set both endpoints (`set-end-points`), something like the following would serve to extend the system with the new behavior:

> The concept of traces can be added quite naturally to a number of existing object-oriented languages. But in this paper, we present traces using a *very* simple Scheme-based object-oriented language. Its advantage, besides its simplicity, is that, in later parts of the paper, it makes it easier to understand traces and their relation to existing object-oriented concepts. An overview of this simple language can be found in Appendix A, but the key point to notice for the time being is that the body of a class is a procedure that returns a *list* of methods.

```
(define <fixed-slope-line>
  (class (<line>)
    (lambda ()
      (list
        (method set-end-points (new-end-1 new-end-2)
          (if (same-slope? (end-1 self) (end-2 self)
                           new-end-1 new-end-2)
              (call-next-method)
              (warn "Can't change orientation of line.")))))))
```

Having defined this new class of line, the programmer could then create instances of it, and pass those into the editor. They would have all the normal behavior of lines, with the exception that if the editor tried to change their slope — by calling `set-end-points` with arguments that changed the slope — the user would be warned that the slope cannot be changed.

Glass Boxes

This style of system design has been called by a number of names, including "glass-box abstraction" [Rao91] and "open implementations" [Kic92]. Underlying these names is an

[1] In this paper, we are using a naming convention in which class names are bracketed by "<" and ">."

intuition that the extensible system is a "box" into which one passes objects on which it operates. Unlike a black-box or closed implementation, the box is glass, or open, because it follows a well-defined protocol on those objects, and one can thus specialize its behavior by passing in objects which respond to the protocol in specialized ways.

While existing object-oriented languages have been a critical enabling technology for open implementations, there is an important class of problems that they have not handled well. Specifically, the case when the extensible system itself (i.e. the graphics editor), creates new objects internally, that must in some way derive part of their behavior from the objects that came in from the outside.

As an example, consider the case where the graphics editor has an operation that groups a collection of line segments together, to form a polygon. In the heart of such an operation, we would expect to find an instance creation operation that looks something like:

```
(make <polygon> 'lines (list line-1 line-2 ...))
```

This code says to create an instance of the class `<polygon>` passing it an initarg[2] named `line` whose value is a list of the line objects that comprise the polygon.

The key point here is that the class of the resulting polygon is fixed in this code. That is, all polygons created will be instances of the class `<polygon>` — never any subclasses of it — and so will have the standard polygon behavior. But, it would clearly be desirable for programmers of subclasses of `<line>` to be able to affect the behavior of polygons made up out of those lines. For example, the programmer of `<fixed-slope-line>` might want to prevent such a polygon from rotating, since that would change the slope of the line.

The question is how to arrange for the (internally created) polygon objects to inherit the appropriate behavior from the (externally created) line objects. There are two traditional approaches to this problem, neither of which has proven completely satisfactory: delegation and dynamic class selection.

Delegation

Using delegation [Lie86] the strategy would be to arrange for the line objects to be *parts of* the polygon object. The polygon would then delegate part of its behavior (i.e. some aspect of rotation) to the line objects, thereby allowing them to control it.

An initial approach in this case would be for the polygon to accept a `rotate` message, which would work in three steps: (i) asking each of the lines for its endpoints, (ii) calculating the new endpoints for all the lines, and (iii) asking each of the lines to change its endpoints accordingly.

But this doesn't work as well as one might like. It won't quite capture the behavior we are looking for, at least not without some work. Consider the case where only one of the line objects in a polygon is an instance of `<fixed-slope-line>`, and it happens not to be the first one in the list of lines the polygon contains. Then, some of the other lines will have had their positions changed before the fixed slope line is encountered. So,

[2] The term *initarg* is an abbreviation for the more cumbersome *initialization argument.*

when the fixed slope line signals its warning message, some of the lines will already have been changed, and the polygon will be in an inconsistent state.

While the delegation protocol can be enhanced in various ways to deal with this case — for example, the polygon could save the old end-points and restore them if it runs into a problem with any of the lines — these sorts of solutions tend to become cumbersome.

The problem is that *a polygon is really more than a collection of lines*, and any attempt to treat it strictly as such will ultimately founder on that difference.

Dynamic Class Composition

The second approach recognizes the problems inherent in the first, and is based on having the newly created objects (i.e. polygons) be responsible for all their own behavior, but allowing the old objects (i.e. lines) to have a say in how the new object is created: specifically, a say in the class of the new object.

To use this approach, a message would be added to the protocol of line objects, asking them what class any polygons they contribute to should be. So, the code that creates polygons would now look like:

```
(let* ((proposed-classes (mapcar polygon-class  all-the-lines))
       (class (most-specific-class proposed-classes)))
  (make class 'lines (list  line-1  line-2 ...)))
```

Where the function `most-specific-class` is responsible for taking all the proposed classes and finding, or creating, the least specific class that is a subclass of all of them.[3] The polygon still receives the lines as an initarg, but only to read the values of the endpoints and then discards the lines themselves. Given this protocol, the programmer of `<fixed-slope-line>` could update their code as follows:

```
(define <fixed-slope-line>
  (class (<object>)
    (lambda ()
      (list
        (method set-end-points  same code as before )

        (method polygon-class () <no-rotate-polygon>)))))

(define <no-rotate-polygon>
  (class (<polygon>)
    (lambda ()
      (list
        (method rotate (degrees)
          (warn "Can't rotate."))))))
```

[3] It could, and often would be one of the proposed classes itself of course.

This approach works reasonably well, but it can become cumbersome if there are many internal creation operations, because a new message must be added to the protocol for each one. Moreover, it only works in languages like Smalltalk, CLOS and Self, that allow runtime class creation.

Traces

The idea of traces is based on recognizing what is good — and bad — about both of the traditional approaches. It takes from the dynamic class composition approach the realization that an object that is made up out of others cannot successfully delegate its behavior to the others, if *the sum of the parts is greater than the whole*. It takes from the delegation approach the simplicity of creating new objects out of old ones.

The key idea underlying traces is to formalize the notion that, as a computation proceeds, new objects are created, which often arise from (or are created by) other objects. We call the older objects (i.e. lines) *ancestors*, and the newer objects (i.e. polygons) their *progeny*. In addition, we formalize the notion of a *trace* which is a packet of behavior that can be attached to an object (i.e. line), and then automatically propagate to some of the object's progeny (i.e. polygons), where they install the appropriate specialized behavior.

The addition of traces manifests itself in two language constructs: A change to the object creation function, and the addition of a new construct to create traces.

The object creation function must be revised to indicate the ancestors of the object being created. To reflect the sense that it is the ancestors which produce the new object, we have changed the name of the object creation function to `give-rise-to`.[4] So, in the example we have been discussing, the creation of a polygon from a set of lines looks like:

```
(give-rise-to <polygon>
              (list  line-1  line-2 ...)
  'lines (list  line-1  line-2 ...))
```

The first argument, `<polygon>` is the class of the object being created, just as in the earlier calls to `make`. The second argument is a list of the direct ancestors of the object being created. The remaining arguments are initargs and values, just as in the previous calls to `make`.

A new macro, called `trace`, makes trace objects. The trace includes a specification of the path of progeny along which it should propagate, and what methods it should create when it is activated. A trace can be attached to an object using the `traces` initarg. The following code creates a `<line>` object, with a single trace that will propagate to just those progeny of the line that are instances of `<polygon>`. It will specialize those polygons, with a method on `rotate` that prevents them from rotating. (Note that for the time being, we are dropping the specialization that the line itself not be able to rotate — we'll return to that shortly.)

[4] We have considered retaining the name `make` for objects with no ancestors, but will not do so in this paper.

```
(give-rise-to <line>
              '()       ;The line itself has no ancestors.
  'end-1 ...
  'end-2 ...
  'traces (list (trace (<polygon>)
                  (lambda ()
                    (list
                      (method rotate (n-degrees)
                        (warn "Can't rotate.")))))))
```

The `trace` macro has two parts. The first part, called the path, is a list of class names, the second is a procedure that returns a list of methods, just like the body of a class definition.

The path controls both the propagation and activation of traces. This trace has the behavior that it propagates only to instances of `<polygon>` and, when it gets there, since the remainder of the path is empty, it activates itself. Essentially, the path says that the trace should follow that list of classes in `give-rise-to` steps.

When a trace is activated, the method maker procedure is called to produce a list of methods, and these are *prepended* to the methods produced by the class itself. In cases where traces come in from more than one ancestor, their relative precedence is determined by the ordering of the ancestor list.

The following is an example of objects with multiple traces, and traces with longer paths, showing their behavior:

```
(define start
  (give-rise-to <object>
                '()
    'traces (list
              (trace (<a> <b>)
                (lambda () (list (method test () 'start))))
              (trace (<x>)
                (lambda () (list (method test () 'start)))))))

 (define a (give-rise-to <a> (list start)))
 (define b (give-rise-to <b> (list a)))
 (define x (give-rise-to <x> (list start)))

 (test a)   -->   ???
 (test b)   -->   start
 (test x)   -->   start
```

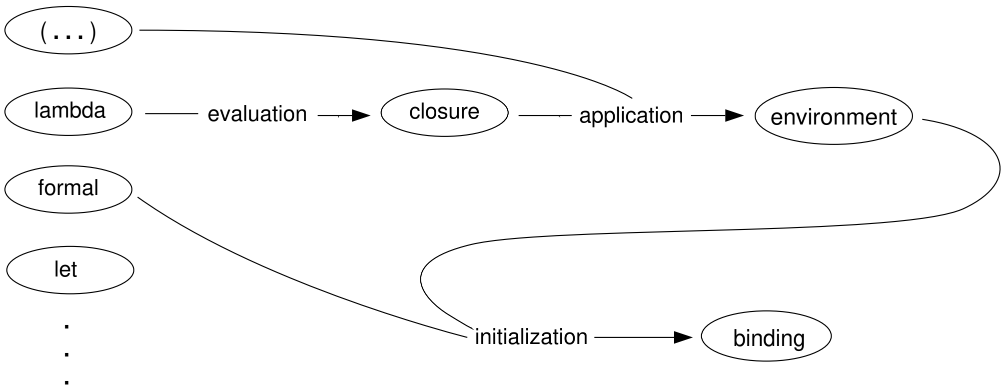

Figure 1: The rise of objects within the evaluator. On the left, are the metaobjects that represent the program text. These are created outside the evaluator and are dropped into it. The lines represent the rise of metaobjects from others within the evaluator.

A More Elaborate Example

Traces make dynamic inheritance of behavior so much easier to handle that we have been able to open implementations of systems that were difficult to handle with previous techniques. As an example of such a system, consider a metaobject protocol (MOP) for a Scheme interpreter.[5] Such a MOP includes three kinds of metaobjects:

- Metaobjects that represent the program graph, including `lambda`s, `formal`s, `applications`, `let`s and `variable-reference`s.
- Metaobjects that represent Scheme runtime objects, including `pair`s, `numbers` and `closure`s.
- Metaobjects that represent internal runtime state of the interpreter, including `environment`s and `binding`s.

The MOP works in the usual way: a documented set of generic functions handles the evaluation of the program, building environment structure as it goes, and creating and passing Scheme data as specified by the program. So, for example, in the execution of the following code fragment:

```
(let ((make-counter
        (lambda (c)
          (lambda (x) (set! c (+ c x)) c))))
  (let ((c1 (make-counter 0)))
    (c1 1)
```

[5]Note that in the MOP we will be presenting, the base language is Scheme, and the metalanguage is the simple object-oriented extension to Scheme we have been discussing. That is, the system is not metacircular.

```
        (c1 3)))
```

First, the evaluation of the first `lambda` expression produces a closure. When that closure is then called by the application on line 4, a new environment is created; it contains a binding for the variable named `c`, for which the initial value is 0. The closure then returned by evaluating `(lambda (x) ...)` is closed over that environment. When that second closure is called (the last two lines) and the `set!` is executed, it is the binding for `c`, created earlier, which is changed.

In terms of the metaphor of a glass box that accepts objects as input and operates on them, the box in this case is the interpreter, and the (meta)objects it accepts are the representation of the program graph.

Consider now the problem of designing the MOP so that an end programmer can specialize the reading and writing of binding values. The natural way to handle reading and writing bindings is with two messages (`get-value` and `set-value`) sent to the binding objects themselves. But, as with the polygons above, binding objects do not appear directly in the program graph that is passed into the MOP-based evaluator, instead they are created during the execution of the program.

As shown in Figure 1, there are three steps of object creation between the program graph metaobjects passed in to the evaluator and the binding metaobjects created internally. First, the lambda expression is evaluated to produce a closure. Later, when that closure is applied to arguments a new environment is created. As part of initializing that environment, bindings are created. But somehow, the user of the MOP must be able to mark one of the lambdas in the program graph, since that is all they have their hands on.

Using traces, the designer of the MOP need only *document the ancestor relationships* as they appear in Figure 1, and then it becomes easy for an end-programmer to define bindings with specialized behavior. The following code creates traces that can be attached to lambda metaobjects, but propagate to that lambda's binding metaobjects and cause messages to be printed when the binding is accessed:

```
(define make-noisy-binding-trace
  (lambda ()
    (trace (<closure> <environment> <binding>)
      (lambda ()
        (list (method get-value ()
                (output self " was read.")
                (call-next-method))
              (method set-value (new)
                (output self " was set.")
                (call-next-method))))))
```

All the MOP user would then have to do is make sure that the appropriate lambda metaobjects given to the interpreter had such a trace. That is, these lambdas must be created by code something like:

```
(give-rise-to <lambda>
              '()
  'arglist ...
  'body ...
  'traces (list (make-noisy-binding-trace)))
```

In our implementation of this MOP, we have defined a special syntax that makes it convenient to create program graph metaobjects with traces from the text of a program. This syntax, shown below, is the analog of the `:metaclass` option in CLOS and similar options in other MOPs that change the class of the program text metaobjects.

```
(let ((make-counter
        {(make-noisy-binding-trace)}
        (lambda (c)
          (lambda (x) (set! c (+ c x)) c))))
  (let ((c1 (make-counter 0)))
    (c1 1)
    (c1 3)))
```

Without traces, supporting this sort of end-programmer specialization in this MOP would have been truly cumbersome. The protocol would be more complex in that each object creation step would have to define explicit messages that determined the class of the object being created. End programmers would have to define a number of "container classes" whose only role was to participate in one-step of the object creation chain. The true code being injected in the system (i.e. the methods on `get-value` and `set-value`) would end up becoming lost in the noise of the mechanism that gets them where they are going. By making the propagation of traces transparent, this program becomes radically simpler.

Specifying Trace Paths

In the formulation given above, the path a trace should follow is specified as the list of the classes it should follow in `give-rise-to` steps. This is simple, and is often good enough, but it suffers from one of the classic problems in object-oriented programming: it forces the designer of a protocol to represent any distinction among progeny that might be important to a trace as a difference in the class of the object being created, rather than some more fine-grained aspect of the individual object's state. This is essentially an instance of the dilemma of capturing a difference among objects in their class vs. in a slot.

A Protocol for Traces

To provide greater control over trace propagation and activation, we have made traces themselves be objects, with a simple, but powerful protocol that controls their propagation and activation. This protocol is driven by the `give-rise-to` primitive. As part of creating the new object, `give-rise-to` first collects the traces from each of the ancestors, then asks each one whether they: (i) want to propagate to the new object; and if they do, (ii) whether they want to activate at that object; and if they do, (iii) to produce a list of methods.

The protocol includes one message that applies to all objects and three that apply only to traces:

- `(traces` *object*`)`
  This message is sent to all the ancestors specified in a call to `give-rise-to`. It returns a (possibly empty) list of trace objects.

- `(propagate?` *trace class* `.` *initargs*`)`
  This message is sent to all the traces of all the ancestors. It asks whether the trace wants to propagate to the new object. It returns either false, or a (possibly new) trace object which will actually be propagated.

- `(activate?` *trace class* `.` *initargs*`)`
  This message is sent to all the traces returned by `propagate?` It asks whether that trace wants to activate at this object. It returns either false or a trace object, which is the one that will actually be activated.

- `(make-methods` *trace* `.` *initargs*`)`
  This message is sent to all the traces returned by `activate?` It returns a list of methods.

A key point to notice is that because methods on the `propagate?` and `activate?` messages have access to the initargs specified in the call to `give-rise-to`, they can use a more fine-grained resolution than just the simple paths supported by the `trace` macro.

History Traces

As an example of using this protocol, we show the definition of a general-purpose "history trace," that keeps track of all the progeny (direct and indirect) of the object it is installed on. It does this by propagating across all `give-rise-to`s. At each progeny, it installs an `initialize` method that records the new object on the growing list of progeny. Note that each individual history trace keeps track of its own progeny, and if one object happens to be the progeny of more than one trace, there are no bad interactions, it simply appears on both lists, as do all of its progeny.

```
    (define <history-trace>
      (class (<simple-trace>)
        (lambda ()
  (let ((*all-progeny '()))
    (list
     (method all-progeny () *all-progeny)

     (method propagate? (c . ignore) self)  ;  go everywhere
     (method activate?  (c . ignore) self)  ;  activate everywhere

     (method make-methods (class . initargs);  ignore args
```

```
                (list
 (method initialize ignore
                   (call-next-method)
                   (setq *all-progeny (cons self *all-progeny))))))))))
```

Inheritance Reconsidered

A language with traces changes the familiar object-oriented notions of inheritance in two important ways: First, it adds a sense of dynamic inheritance and second, it leads to a programming style in which there is a sharp distinction between inheritance of code in the default implementation and specialization performed by end programmers.

Dynamic Inheritance

A language with traces, has the traditional static inheritance among classes, but it also has *dynamic* inheritance of behavior from ancestors to progeny. In fact, the progeny paths along which dynamic inheritance can take place are much more important to the end programmer than the static inheritance, since we don't allow the addition of methods to a class once it has been defined. The documentation of a system written using traces always includes those progeny paths, perhaps in a form something like that shown in Figure 1.

Just as it is possible to build automatic tools for browsing class graphs — that is static inheritance — moderately sophisticated code analysis techniques should make it possible to build automatic tools for browsing the dynamic inheritance of a program.

It is important to note that this dynamic inheritance is different from that found in languages such as Self. In those languages — where there is no notion of class — it is true that prototypes are constructed dynamically, as are "instances" of them. But, any given object is cloned from a single other object, from which it takes all of its behavior. This is different than the behavior of `give-rise-to` where the class specifies the bulk of the new object's behavior, and traces on the ancestors provide specialization. Also note that, in traces, ancestors don't necessarily have the behavior they give to their progeny. They are instead simply carriers of that behavior.

A New Programming Style

In our use of traces, we have developed a programming style in which the end programmer specializing a system never defines subclasses of the supplied classes. Instead, they always use traces, even when the object they want to specialize is one they are creating directly.

Returning to the example of the fixed slope line, the end programmer using this style would not define a new class called `<fixed-slope-line>`. Instead, they would simply create an instance of `<line>` with a trace that activates itself immediately and supplies the appropriate behavior. The complete definition of fixed slope lines, using traces, is then:

```
(define make-fixed-slope-line
```

```
(lambda ()
  (give-rise-to <line>
                '()
    'traces (list
              (trace ()
                (lambda ()
                  (list
                    (method set-end-points ...))))
              (trace (<polygon>)
                (lambda ()
                  (list
                    (method rotate ...))))))))
```

This style is attractive in that it provides a strong syntactic distinction between code that defines a default implementation, which is mostly written using classes,[6] and code that specializes existing code, which is entirely written using traces.

Performance

We have not yet developed a high-performance implementation of a language with traces in it. Traces have been so useful in our work on metaobject protocols that we have been happy to pay the costs associated with our very simple unoptimized implementation. We believe that we can nonetheless make some simple observations about the performance possibilities for traces.

In some sense, traces can be seen as "lightweight classes" with `give-rise-to` performing dynamic class composition. Seeing it this way makes it clear that something like traces could be layered on top of languages like Smalltalk, CLOS or Self. In CLOS and Smalltalk, `give-rise-to` would do dynamic composition of classes. In Self, the whole thing could probably be built using Self's ability to directly manipulate vectors of methods that form objects. Method dispatch would just be the host language's method dispatch. This approach would likely result in good performance for method invocation, at the expense of potentially terrible performance for some calls to `give-rise-to`.

Arranging for traces to have fast method dispatch *and* fast object creation could be much more difficult. Potentially sophisticated flow analysis will be required, to allow the compiler to infer what traces might actually arrive at each `give-rise-to`. We are deferring any such effort until we have much more experience working with traces, so that we can have a better sense of how to invest our energies.

Summary

We have presented a new object-oriented language concept, traces, together with a concrete embedding of that language in a simple object-oriented extension to Scheme. Traces

[6]Even code defining a default implementation might use traces, to capture default dynamic inheritance of behavior.

require two additions to existing object-oriented languages: an explicit notion of ancestor-progeny paths, and trace objects, which are carriers of behavior that can follow those paths. This provides us with a powerful way to open up certain kinds of systems that were difficult to handle with previous object-oriented languages. Traces also provide us with two critical distinctions in programs: first between static and dynamic inheritance; and second, between inheritance of behavior in the default implementation and specialization by the end programmer.

Acknowledgments

The concept of traces arose out of discussions with members of a group working on a MOP-based Scheme compiler. Over the past several years, this group has included: Hal Abelson, J. Michael Ashley, Jim des Rivières, Mike Dixon, John Lamping, Anurag Mendhekar, Luis Rodriguez, Erik Ruf and Amin Vahdat. Brian Smith contributed numerous insights regarding object individuation criteria. Discussions with Craig Chambers, Dan Friedman, Chris Haynes, Satoshi Matsuoka, Mario Tokoro and Akinori Yonezawa, helped form the idea into a workable language.

A A Simple Object-Oriented Language

To simplify the presentation, this paper presents traces by starting with a new and very simple object-oriented language. This appendix presents an overview of that starting point.

The language is a variant of Scheme, with objects added. To do this, we have added: *objects*, *selectors*, *methods* and *classes*. An object is simply a list of methods, a method is a pair of a selector and a procedure. A class is a linguistic convenience for producing a list of methods that comprise an object. (Users can, and sometimes do, make the list of methods directly without a class.)

A regular Scheme application, whose head is a selector, is called a *message send*. The first argument, which must be an object, is searched for its first method that matches the selector, and that method is then run.

As an example of using the language, consider the following definitions of the classes `<position>` and `<line>`. These are the definitions that are assumed to be part of the default library provided by the graphics editor discussed in the body of the paper.

The class `<position>` defines immutable cartesian positions, that support two messages, `pos-x` and `pos-y`, to access the x and y coordinates of a position. The class `<line>` defines a line segment, of which the two endpoints are represented by position objects. Line objects support operations to access each of the endpoints, and to set the two endpoints together.

```
(define pos-x (make-selector))
(define pos-y (make-selector))

(define <position>
  (class (<object>)
    (lambda ()
```

```
    (let (*x *y)
      (list
        (method initialize initargs
          (set! *x (getl initargs 'x))
          (set! *y (getl initargs 'y)))

        (method pos-x () *x)
        (method pos-y () *y))))))


(define end-1 (make-selector))
(define end-2 (make-selector))

(define set-end-points (make-selector))

(define <line>
  (class (<object>)
    (lambda ()
      (let (*end-1 *end-2)
        (list
          (method initialize initargs
            (set! *end-1 (getl 'initargs 'end-1))
            (set! *end-2 (getl 'initargs 'end-2)))

          (method end-1 () *end-1)
          (method end-2 () *end-2)

          (method set-end-points (new-end-1 new-end-2)
            (set! *end-1 new-end-1)
            (set! *end-2 new-end-2)))))))
```

The `method` macro is like Scheme's `lambda`. It returns a method for the selector that is its first argument (evaluated when the method is constructed). The rest of the args of a `method` form are just like the args to a `lambda` form: the parameters to the method and the code body.

Within the body of a method, `call-next-method` can be used, as in CLOS, to call the next method for that generic-function in the object's list of methods. Also note that the lexical variable `self` is bound within the body of a method to the object that was the first argument to the generic function.

The object creation primitive, `make`, takes a class and initargs as arguments, just as it does in CLOS. After the object is created, it is automatically initialized. Methods on `initialize` receive all the initiargs passed to `make` and can process them as they like. The convention is for methods on `initialize` to always invoke `call-next-method`. (Bottoming out at the most general class, `<object>`.)

```
(define p1 (make <position> 'x 3 'y 5))          a positon
(define p2 (make <position> 'x 9 'y 5))          a positon
```

```
(pos-x p1)                                          returns 3

(define l1
  (make <line> 'end-1 p1
               'end-2 p2))                          a line

(set-end-points l1                                  rotate and resize
                (make <position> 'x 0 'y 0)
                (make <position> 'x 10 'y 10))
```

Each time an instance of the class is created, the class's method maker procedure is called to produce a new list of methods. The resulting list is stored in the new instance, as its list of methods. If the class has superclasses, the methods from all the classes in the class precedence list are appended. In the starting point language, `make` is implemented as:

```
(define make
  (lambda (class . initargs)
    (let* ((cpl (class-precedence-list class))
           (methods (apply append (mapcar make-methods cpl)))
           (object (make-object methods)))
      (apply initialize object initargs)
      object)))
```

A final point is to note that this very simple semantics means that no special support must be provided to distinguish slots, methods, internal slots, internal methods, class slots etc. Ordinary lexical scoping can be used to handle all of these, and with more power than is commonly found in existing object-oriented languages.

References

[BKK+86] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik, and F. Zdybel. Commonloops: Merging Lisp and object-oriented programming. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[CI93] Shigeru Chiba and Yuuji Ichisugi. Open c++. In *European Conference on Object Oriented Programming*, 1993. to appear.

[Coi87] Pierre Cointe. The ObjVlisp kernel: A reflexive lisp architecture to define a uniform object-oriented system. In Pattie Maes and Daniele Nardi, editors, *Meta-Level Architectures and Reflection*, pages 155–176. North-Holland, 1987.

[Fer89] Jacques Ferber. Computational reflection in class based object oriented language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), New Orleans, LA.*, SIGPLAN Notices 24(10), pages 317–326. ACM, October 1989.

[FJ89] Brian Foote and Ralph E. Johnson. Reflective facilities in smalltalk-80. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), New Orleans, LA.*, SIGPLAN Notices, 24(10), pages 327–335, October 1989.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Kic92] Gregor Kiczales. Towards a new model of abstraction in software engineering. In *Proceedings of the IMSA'92 Workshop on Reflection and Meta-level Architectures*, 1992. Also to appear in forthcoming PARC Technical Report.

[KL92] Gregor Kiczales and John Lamping. Issues in the design and documentation of class libraries. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 435–451, 1992.

[Lie86] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPSLA '86 Conference Proceedings, Sigplan Notices* **21***(11)*. ACM, Nov 1986.

[Mae87] Pattie Maes. Concepts and experiments in computational reflection. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 147–155, 1987.

[MCD91] J. Malenfant, P. Cointe, and C. Dony. Reflection in prototype-based object-oriented programming languages. In *Proceedings of the OOPSLA Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, 1991.

[Rao90] Ramana Rao. Implementational reflection in Silica. In *Informal Proceedings of ECOOP/OOPSLA '90 Workshop on Reflection and Metalevel Architectures in Object-Oriented Programming*, October 1990.

[Rao91] Ramana Rao. Implementational reflection in Silica. In Pierre America, editor, *Proceedings of European Conference on Object-Oriented Programming (ECOOP)*, volume 512 of *Lecture Notes in Computer Science*, pages 251–267. Springer-Verlag, 1991.

[Rao93] Ramana Rao. The silica window systems: The metalevel approach applied more broadly. In A. Paepcke, editor, *Object-Oriented Programming: The CLOS Perspective*, pages 133–153. The MIT Press, Cambridge, MA, 1993.

[YMFT91] Yasuhiko Yokote, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. Evaluation of muse reflective object management. In *Proceedings of the 8th Conference of Japan Society for Software Science and Technology*, September 1991. (in Japanese, also available as a technical report SCSL-TM-91-019, Sony Computer Science Laboratory Inc.).

[YTM+91] Yasuhiko Yokote, Fumio Teraoka, Atsushi Mitsuzawa, Nobuhisa Fujinami, and Mario Tokoro. The Muse object architecture: A new operating system structuring concept. *Operating Systems Review*, 25(2), April 1991. (also available as a technical report SCSL-TR-91-002, Sony Computer Science Laboratory Inc.).

[YTY+89] Yasuhiko Yokote, Fumio Teraoka, Masaki Yamada, Hiroshi Tezuka, and Mario Tokoro. The design and implementation of the Muse object-oriented distributed operating system. In *Proceedings of the First Conference on Technology of Object-Oriented Languages and Systems*, October 1989. (also available as a technical report SCSL-TR-89-010, Sony Computer Science Laboratory Inc.).