Soft Objects:
A Paradigm for
Object Oriented Programming

Ken Haase

ABSTRACT

This paper introduces soft objects, a new paradigm for object oriented programming. This paradigm replaces the traditional notion of object classes with the specification of transforming procedures which transform simpler objects into more complicated objects. These transforming procedures incrementally construct new objects by adding new state or providing handlers for new messages. Unlike other incremental approaches (e.g. the inherited exist handlers of Object Logo [Drescher, 1987]), transforming procedures are strict functions which always return new objects; rather than conflating objects and object abstractions (classes), soft objects distinctly separates objects and their abstractions. The composition of these transforming procedures replaces the inheritance schemes of class oriented approaches; order of composition of transforming procedure makes explicit the inheritance indeterminancies introduced by multiple super classes. Issues regarding semantics, efficiency, and security are discussed in the context of several alternative implementation models and the code of a complete implementation is provided in an appendix.
1 Introduction

Soft objects are a new way of organizing object oriented programs which replaces the popular approach of ‘class abstractions’ [Gabriel, 1987, Moon, 1986, Tex, 1985] with a procedural abstraction based on the specification of transforming procedures. Transforming procedures produce more complicated objects by a series of functional transformations applied to simpler objects.

This paper describes an implementation of soft objects in SCHEME [Rees and Clinger, 1986]. As implemented in SCHEME, a soft object is a procedure whose first argument is a message identifier and whose remaining arguments are parameters for that message. As in other schemes, it is assumed that each message denotes some particular operation whose implementation is handled differently by different objects. In most of this paper, I will assume that message identifiers are symbols; however, in Section 16 (Page 24), I follow [Adams and Rees, 1988] in implementing ‘anonymous messages’ which avoid a variety of problems arising from the association of messages and global identifiers.

An object is ‘sent a message with arguments,’ by calling the object (qua procedure) on the message identifier and the message’s arguments. For instance, suppose the SCHEME identifier WINDOW-1 were bound to a ‘window object’; then the window could be directed to expose itself by sending it an expose message:

(WIN1 expose)

Once it had been exposed, we might ask it to draw attention to itself by beeping; we send a beep message with arguments for frequency and duration:

(WIN1 beep 150 0.5)

specifying that the window produce a 150 hertz beep lasting for half a second.

Suppose we wanted a window which beeped whenever it was exposed; in the soft object paradigm, we can do this by using the handle message to get a new window which combines the two operations:

(define WINDOW-2
  (WINDOW-1 handle expose
    (lambda ()
      (WINDOW-1 expose)
      (WINDOW-1 beep 150 0.5))))

The handle message is functional; it returns a new object to which the identifier WINDOW2 is bound. Whenever this new WINDOW2 is sent an expose message, the corresponding window will first be exposed and then beep. For a variety of reasons, we might choose to abstract the last fragment of code — constructing a window which beeps on exposure from one which does not — into a new procedure BEEP-ON-EXPOSURE:

(define (beep-on-exposure window)
  (window handle expose
    (lambda ()
      (window expose)
      (window special-effect beep)))))

Given this, we can say either

(define WINDOW-2 (beep-on-exposure WINDOW-1))
A procedure which produces a more complicated soft object from a less complicated one is called a *transforming procedure*.

In a traditional object-oriented programming framework, the transforming procedure `beep-on-exposure` might be implemented by a `beep-on-exposure-mixin` class; this *mixin* class (so called because it participates in other class definitions but does not of itself constitute an instantiable class) would specify a new auxiliary method for exposing windows.

Some other languages (in particular, Object LOGO and Object LISP [Drescher, 1987][Cor, 1986]) provide for the incremental local definition of variables and handlers on individual objects. In practical usage, while many of an object’s variables and message handlers are declared locally in this way, shared variables and most message handlers are accessed in a more-or-less conventional manner through an inheritance hierarchy. But unlike previous approaches, this inheritance hierarchy makes no distinction between objects and classes; the ‘classes’ which an object inherits from are themselves full-fledged objects which act more as *prototypes* than as abstract class descriptions. In this framework, the local state of objects is established by an *exist* message sent when an object is created; the handlers for this message (inherited through the prototype hierarchy) set up those state variables which are local to individual objects.

Soft objects take the opposite approach to the sometimes awkward distinction between classes and instances in object oriented languages. Rather than merging the notions of object and class, soft objects use an entirely separate mechanism for defining object abstractions: the specification of transforming procedures which act on objects to produce objects with the new functionality. Functional composition replaces inheritance, and the hierarchy of super classes or prototypes is replaced by the history of an object’s construction.

## 2 Up from the Ur-Object

Soft objects begin with a single object (procedure) called the *ur-object* which accepts only the `handle` message.\(^1\) The arguments to the `handle` message are an alternating list of message identifiers and handler procedures. Sending an object a `handle` message returns a new object whose behaviour is exactly that of the old object except for the message identifiers specified in the arguments to `handle`; for these messages, the new object calls the associated handler procedure instead.\(^2\)

New objects are constructed by ‘teaching’ the ur-object to handle particular messages in particular ways; the objects returned by the `handle` message are new and more complicated objects. The state of a soft object is embedded in the procedures it is given as handlers, rather than being immanent in the object itself. For instance, we can construct an object which stores a retrievable or modifiable property by creating handlers which share a lexical variable:

---

\(^1\)In later sections we will introduce versions of the *ur-object* which accept other messages.

\(^2\)The `handle` message is much like the `extend` messages of ACT-1 and ACT-2 [Lieberman, 1987, Theriault, 1983]. The `extend` message was part of a large framework for message-based programming; this paper emphasizes how the `handle` message integrates into conventional SCHEME and the definition of procedural abstractions over applications of the `handle` message.
(define colored-object
  (let ((color 'WHITE))
    (ur-object 'handle
      'color (lambda () color)
      'set-color! (lambda (new-color) (set! color new-color)))))
  (colored-object 'color)
⇒ WHITE
(colored-object 'set-color! 'BLUE)
⇒ UNSPECIFIED
(colored-object 'color)
⇒ BLUE

Here the value of the ‘color’ property is stored in the environment in which the handlers for the color and set-color! messages are are closed. This is similar to the approach in [Adams and Rees, 1988].

We can abstract the above definition into a HAVE-COLOR procedure:

(define (have-color object)
  (let ((color 'WHITE))
    (object 'handle
      'color (lambda () color)
      'set-color!
        (lambda (new-color) (set! color new-color)))))

or abstract it even further into a two-argument HAVE-PROPERTY procedure:

(define (have-property object property)
  (let ((value #F))
    (object 'handle
      property (lambda () value)
      ;; SYMBOL-APPEND returns a symbol whose name is a
      ;; concatenation of other symbol’s names.
      ;; (symbol-append 'SET- property ')
        (symbol-append 'SET- property ')
        (lambda (new-value) (set! value new-value)))))

This procedure illustrates how a transforming procedure can in fact denote an entire range of abstractions by taking extra arguments. We could have also ‘curried’ the arguments to the procedure to define a procedure which produced transforming procedures.

(define (to-have-property property)
  (lambda (object)
    (let ((value #F))
      (object 'handle
        property (lambda () value)
        (string->symbol
          (string-append "SET-" (symbol->string property) "!"))
        (lambda (new-value) (set! value new-value))))))

allowing a series of definitions like:

(define have-color (to-have-property 'color))
(define have-size (to-have-property 'size))
(define colored-object (have-color (have-size ur-object)))
Endless similar variations are possible; for instance, we might provide defaults for properties (by passing in an initial value to use other than #F) or restricting the values a property may take by some predicate (which would be checked in set-property).

3 Implementing soft objects

A simple implementation of soft-objects requires only a handful of procedure definitions, of which only one (ur-object) need be seen by users. We begin with a procedure MAKE-NEW-OBJECT which takes an existing object and an alternating list of message identifiers and handling procedures; it constructs a new object which uses the corresponding handling procedure for messages in this list and otherwise defers by passing the message onto the original object. Here is a possible implementation of MAKE-NEW-OBJECT:

```lisp
(define (make-new-object old-object handlers)
  (define (new-object message . arguments)
    (if (eq? message 'handle) (make-new-object new-object arguments)
       (let ((local-handler (mget message handlers)))
         (if local-handler (apply local-handler arguments)
           (apply old-object message arguments))))
  new-object)
```

The object returned by MAKE-NEW-OBJECT is a procedure which dispatches on messages in one of three ways:

- The handle message constructs yet another object by calling MAKE-NEW-OBJECT to further build on the object.
- Locally defined messages in handlers are handled by calling the corresponding procedure on the list handlers.
- If all else fails, the message is passed on to the old-object from which the object was constructed.

The local handler list handlers could be any sort of lookup table accessed by MGET; we will treat it as an alternating list of handler names and handlers, implementing MGET thus:

```lisp
(define (mget message handlers)
  (if (null? handlers) #F
       (if (eq? (car handlers) message) (cadr handlers)
           (mget message (caddr handlers))))
```

Given a way of making new objects, we construct a primitive object which accepts no messages (the ignomoramus),

```lisp
(define (ignomoramus message . arguments)
  (error "Can't handle message" message arguments))
```

and construct the ur-object on top of it,

```lisp
(define ur-object (make-new-object ignomoramus ()))
```

In later sections, this implementation will be expanded by providing other messages akin to handle. The simplicity of this implementation (four definitions) is compelling and made more so when we recognize that only the object (procedure) ur-object need be visible to users of the object system.
(define (be-a-turtle object)
  (let ((x 0) (y 0) (heading 0) (pen-down? #T))
    (define (move-forward n)
      (let ((new-x (+ x (round (* n (cosine heading))))))
        (new-y (+ y (round (* n (sine heading))))))
      (if pen-down? (draw-line x y new-x new-y))
      (set! x new-x)
      (set! y new-y)))
    (define (turn-right degrees)
      (set! heading (- heading (* degrees (/ 3.1415 180)))))
    (define (turn-left degrees)
      (set! heading (+ heading (* degrees (/ 3.1415 180)))))
    (define (pen-state)
      (if pen-up? 'raise-pen 'lower-pen))
    (define (raise-pen) (set! pen-down? #F))
    (define (lower-pen) (set! pen-down? #T))
    (object 'handle 'forward move-forward
     'right turn-right
     'left turn-left
     'raise-pen raise-pen
     'lower-pen lower-pen
     'pen-state pen-state)))

Figure 1: The procedure **BE-A-TURTLE** provides a turtle graphics implementation to the object it is given. It returns an object accepting the turtle graphics messages like **forward**, **right**, **raise-pen** and **pen-state**. The state of a turtle — x, y, heading, and pen-down? — is encapsulated in the procedures given as handlers to the **handle** message.

4 An Example: Object Oriented Turtles

This section gives an implementation of object oriented turtle graphics using soft objects; some of the organizational ideas here are borrowed from the environment which Gary Drescher developed for QLOGO at Atari’s Cambridge Research Center. In implementing turtle graphics, we assume a variety of procedures whose actions should be obvious from their names: **DRAW-LINE** or **SINE**.

We begin by defining a procedure **BE-A-TURTLE** which defines variable bindings for an X-Y position, a heading in degrees, and a pen-up/pen-down flag; **BE-A-TURTLE** is defined in Figure 1. The state of a turtle — x, y, heading, and pen-down? — is encapsulated in the procedures given as handlers to the **handle** message; we can construct an object with this state by simply calling **be-a-turtle** on the ur-object.

    (define turtle-1 (be-a-turtle ur-object))

and sending **turtle-1** the appropriate messages to do whatever graphics we desire; e.g.

    (turtle-1 'forward 50)

We can also transform **turtle-1** again to handle more sophisticated messages; a simple case would be a transforming procedure like **BE-SQUARE** which teaches the message **SQUARE**.
(define (be-square turtle)
  (define (draw-square size)
    (turtle 'forward size) (turtle 'right 90)
    (turtle 'forward size) (turtle 'right 90)
    (turtle 'forward size) (turtle 'right 90)
    (turtle 'forward size))
  (turtle 'handle 'square draw-square))

We could then draw a square on the screen by the sequence:

(define turtle-2 (be-square turtle-1))
(turtle-2 'square 20)

5 Visible Turtles: A Limited Implementation

The turtles returned by BE-A-TURTLE are 'invisible'; they show no turtle-like appearance on the display screen. We will use a simple trick to implement visible turtles: we will have invisible turtles erase and redisplay a particular 'appearance' before and after each movement. This will only work if the 'appearance' returns the turtle to its starting point and orientation when completed.

The BE-VISIBLE procedure constructs a new object with new handlers for the basic movement messages.

(define (be-visible turtle)
  ;; This forwards a message to turtle, but wraps the
  ;; execution within an erasure and redisplay.
  (define (wrap-message-with-redisplay message)
    (lambda arguments
      (wrapper (lambda () (erase-turtle turtle))
        (lambda () (apply turtle message arguments)))
      (lambda () (draw-turtle turtle)))))
  (turtle 'handle
    'forward (wrap-message-with-redisplay 'forward)
    'right  (wrap-message-with-redisplay 'right)
    'left   (wrap-message-with-redisplay 'left)))

The procedures DRAW-TURTLE and ERASE-TURTLE draw and erase a turtle-esque triangle on the screen by sending graphics messages to the turtle they are passed. The procedure WRAPPER calls each of its arguments in order and returns the result of the second application; the internal procedure WRAP-MESSAGE-WITH-REDISPLAY uses WRAPPER to erase and redraw the turtle before and after (respectively) it sends a particular message to turtle. Given the implementation of BE-VISIBLE, we can construct a 'visible turtle' by composing BE-VISIBLE with BE-A-TURTLE:

(define vturtle-1 (be-visible (be-a-turtle ur-object)))

Whenever vturtle-1 gets a simple movement message (forward, right, or left), it erases itself, executes the movement and draws itself anew. Thus the image of the turtle appears to move around the screen as it obeys its graphical commands.

A problem with this approach is that it only works on 'infant' turtles; since the new versions of forward and other messages are only seen by subsequently defined handlers, handlers already defined will execute their actions without appropriate disappearances and reappearances before and
after execution. For instance, suppose we applied BE-VISIBLE to a turtle which had already been transformed by BE-SQUARE:

```
(define square-turtle (be-square (be-a-turtle ur-object)))
(define buggy-turtle-1 (be-visible square-turtle))
```

When buggy-turtle-1 receives a square message, it is executed by the invisible turtle square-turtle; as a result, no turtle movement occurs as the square is drawn. But even worse, BE-VISIBLE only adds the erase/redisplay sequence to the basic movement messages and not to the square message; as a result, the turtle draws a square and leaves its image behind! And since the implementation of square ends up at a different heading than it started from, the next erasure/redisplay will erase the wrong image; interacting with other movements, this behaviour will leave a trail ‘ghost turtles’ revealing a disconnected history of the turtle’s past positions.

We can solve this problem by specifying transforming procedures which act on all messages received by an object. These procedures, called comprehensive transforming procedures, are discussed in the next section.

6 ‘Comprehensive’ Transforming Procedures

Comprehensive transforming procedures are transforming procedures which return new objects which behave differently on all messages received by the object. In the last section, we implemented a version of ‘visible turtles’ which assigned new handlers for the three movement messages forward, left, and right. However, other movement messages defined for the object might still use the old definitions of these messages and thus not get the effects of appearing and disappearing turtles with each motion. Even worse, older definitions might violate particular contracts (such as erasure/reappearance constraints) assumed by new messages and operations.

Here is an implementation of a comprehensive transforming procedure which does an erasure/reappearance before and after every message it receives:

```
(define (be-visible turtle)
  (define (visible-turtle message . arguments)
    (if (eq? message 'handle) (make-new-object turtle arguments)
      (lambda () (erase-turtle turtle))
      (lambda () (turtles message arguments))
      (lambda () (draw-turtle turtle))))
  (draw-turtle turtle)
  visible-turtle)
```

This comprehensive procedure is much like the implementation of MAKE-NEW-OBJECT in Section 3. In particular, it handles the handle message by itself and otherwise just defers to the turtle it was made from, preceding the deferred message by an erasure and following it with a reappearance. Note that the returned object procedure must handle the handle message, since passing on the handle message would build a new object on the invisible turtle it was passed and not on the newly generated object it produced.

Note that the new implementation of BE-VISIBLE might still process the square message without advancing the turtle’s appearance as the edges of the square are drawn. Composing calls to transforming procedures in a different order will produce turtles with different behaviors. The two turtles

```
(define square-projector (be-visible (be-square (be-a-turtle ur-object))))
(define square-sketcher (be-square (be-visible (be-a-turtle ur-object))))
```
will respond to the square message differently. When square-projector receives a square message, the turtle will vanish, a square will appear, and the turtle will reappear. When square-sketcher receives a square message, the turtle will go forward along one square edge, drawing the edge; it will then turn in place, go forward along another edge, and so on... Either of these might be desirable and are valid variations expressed by the 'inheritance' order implicit in the order of composition of BE-SQUARE and BE-VISIBLE.

An interesting possibility with this approach would have the DRAW-TURTLE and ERASE-TURTLE procedures send a message appear or disappear to the objects upon which they operate. In this way, transforming procedures applied before BE-VISIBLE could determine the appearance of the turtle on the display. This is the approach given in the detailed implementation of Section ??.

Comprehensive transformers can combine objects as well as transforming them; for instance, a transformer might produce a new object which delegates [Lieberman, 1986] messages to one of several other objects. The transforming procedure SCHIZOID, for instance, sends alternating messages to one of two other objects.

(define (schizoid jeekyll hyde)
  (let ((flag? #T))
    (define (schiz message . arguments)
      (cond ((eq? message 'handle) (make-new-object schiz arguments))
            (flag? (set! flag #F) (apply hyde message arguments))
            (ELSE (set! flag #T) (apply jeekyll message arguments))))
    schiz))

Another (likely more useful) possibility is constructing a distributor object which forwards messages to different objects based on the name of the message received. For instance, the following fragment of code might be useful in a window system for defining a sort of window with separate typein and typeout panes:

(define (make-two-pane-window input output)
  (define (new-window message . arguments)
    (cond ((eq? message 'handle) (make-new-object new-window arguments))
          ((input-message? message) (apply input message arguments))
          ((output-message? message) (apply output message arguments))
          (ELSE (apply output message arguments)
                 (apply input message arguments))))
  new-window)

This constructs an object which dispatches messages to one of two windows depending on the type of the message (as determined by INPUT-MESSAGE? and OUTPUT-MESSAGE?).

We can use certain sorts of comprehensive transformers to solve the 'object identity crisis' alluded to in the introduction. The next section introduces a comprehensive transformer SHARE which returns a shareable object whose 'in-place' transformations can be visible to other objects.

7 Shareable Objects

One issue with the use of transforming procedures is the loss of object identity under changes in class definitions. Transforming procedures always produce a new object and so the changes or extensions introduced by a transforming procedure belong only to this new object. For instance, if a window system manager already had a pointer to the window WINDOW-1, it would not get the extra functionality provided by BEEP-ON-EXPOSURE since that functionality belongs only to WINDOW-2 (the new object returned by BEEP-ON-EXPOSURE).
In a traditional class-based paradigm, recompiling the class definition — with the new mixin or new method definition — would propagate the changed functionality to new instances of the class. (Practically, such propagation is usually contingent on the nature of the change and the sophistication of the object compiler).

But allowing such arbitrary changes to existing objects is potentially dangerous. A module (like a window system manager) with references to an object has a particular contract with the object; it might be dangerous to modify the object’s behaviour without understanding the implicit contracts of other modules referring to the object. It is important to be able to control the scope of changes introduced to shared objects and class descriptions in complex systems.

By retaining the functional character of transforming procedures, soft objects eliminate some of the unpredictability of arbitrary class redefinitions. In much the same way that side-effect free programming eliminates many of the conflicts and dangers in procedure specifications, transforming procedures reduce the conflicts arising from multiple access to abstract objects and descriptions shared among multiple subsystems.

And in any case, the loss of object identity and sharing is not irretrievable; we can implement some mechanism for sharing objects between modules which allow one module’s transformations to be visible to other modules. (Section 12 describes ways of restricting the scope of these changes.)

The procedure SHARE takes an object and returns a ‘shared object’ whose transformations are visible to any program pointing to the shared object and not merely to the program applying the transformation. A shared object maintains an internal pointer to an unshared ‘standard’ object; most messages sent to the shared object are simply passed on to this internal object. A shared object is transformed when it receives a change message; it responds to this message by applying a transforming procedure (the argument of the change message) to its internal object and replacing this object by the result of the transformation.

In this way, transforming procedures remain functional while allowing the construction of objects which can be shared between program modules. SHARE might be implemented like this:

```
(define (share object))
(define (shared-object message . arguments)
  (cond ((eq? message 'handle)
          (error "I can only CHANGE inside!"))
        ((eq? message 'change)
         (set! object (apply (car arguments) object (cdr arguments)))
          object)
        (else (apply object message arguments)))
shared-object)
```

allowing the definition of shared objects thus:

```
(define shared-window (share window-1))
```

The object shared-window might be passed to other programs and subsequently transformed by these programs using the change message. Such transformations would affect messages sent to shared-window but would not affect messages sent to window-1, the original window constructed by the window package.

It is worth explaining why the definition of share above needs to reject the handle message. One superficially plausible implementation of SHARE returns objects which accept the handle message; the shared object forwards the handle message to its ‘internal’ object and sets up the returned value as a new internal object:
(define (buggy-share object)
  (define (shared-object message . arguments)
    (cond ((eq? message 'handle) '!!! Buggy handling of handle.
      (set! object (apply object message arguments))
       shared-object)
      ((eq? message 'change)
      (set! object ((car arguments) object))
      object)
     (ELSE (apply object message arguments)))))
  shared-object)
  (define buggy-window (buggy-share window-1))

The problem is that many transforming procedures assume that the behavior of the object the transform remains fixed; in particular, they may forward messages from the object they are creating to the object it is being created from. But in the implementation above, transforming a shared object will yield an object which forwards message back to the shared object. For instance, suppose that the transforming procedure ADD-ONE-TO-SIZE is defined and applied to BUGGY-WINDOW.

(define (add-one-to-size object)
  (object 'handle 'size (lambda () (+ (object 'size) 1))))
(define bigger-buggy-window (add-one-to-size buggy-shared-window))

This transformation makes the internal object to buggy-window have a new handler for size which recursively refers the size message to buggy-window. When an unsuspecting program sends bigger-buggy-window a size message, it recursively sends a size message (in turn) to buggy-window, which sends the message to bigger-window and so forth; this recurs forever.

To get around this problem, a correct implementation of SHARE uses the change message to apply an external transforming procedure to the internal object. As a result, the transforming procedure never sees the mercurial shared object and avoids the infinite recurrence engendered by its potential self-reference.

One awkwardness with this implementation of shared objects is that it is unnecessary to know whether an object is shared or not before transforming it. One way around this would be to define a change message for every object and always use this message for applying transforming procedures. Such an extension could be implemented by modifying the object (procedure) returned by MAKE-NEW-OBJECT to handle the change message:

(define (make-new-object old-object handlers)
  (define (new-object message . arguments)
    (if (eq? message 'handle) (make-new-object new-object arguments)
       (if (eq? message 'change) ((car arguments) new-object)
       (let ((local-handler (mget message handlers)))
         (if local-handler (apply local-handler arguments)
         (apply old-object message arguments)))))
  new-object)

This new version of make-new-object returns an object which handles a change message by applying the message's argument (a transforming procedure) to itself and returning the result.

The security problems of being able to modify shared objects remains a serious concern. Section 12 explores the possibility of constructing restricted objects which can be more safely shared.
8 Extracting Handlers: The how Message

This section introduces a way of accessing the handlers constructed or assigned to an object along its history of transformations. The how message has a single argument — a message identifier — and returns the procedure which an object uses for handling that message.

We introduce the how message to soft objects by extending MAKE-NEW-OBJECT to return an object which handles the how message.

\[
\text{(define (make-new-object old-object handlers)}
\begin{align*}
& \text{(define (object message . arguments)}
& \text{ (if (eq? message 'handle) (make-new-object object arguments)}
& \text{ (if (eq? message 'how)}
& \text{ (or (mget (car arguments) handlers)}
& \text{ (old-object 'how (car arguments)))}
& \text{ (let ((local-handler (mget message handlers)))}
& \text{ (if local-handler (apply local-handler arguments)}
& \text{ (apply old-object message arguments))))})
\end{align*}
\]

When an object receives the how message it looks up the message's argument on its list of handlers; if it is there, it returns the corresponding handler; otherwise, it passes the how message down the line to the object it was generated from. To ensure that this message relay terminates, we extend the ignoramus object to handle how in a minimal fashion: \(^3\)

\[
\text{(define (ignoramus message . arguments)}
\begin{align*}
& \text{ (if (eq? message 'how) #f}
& \text{ (error "Can't handle message" message arguments))}
& \text{ ;; redefine UR-OBJECT if necessary....}
& \text{ (define ur-object (make-new-object ignoramus()))}
\end{align*}
\]

The most obvious use of the how message is in 'compiling out' message lookup in certain situations. Whenever the object and message for a message send are known, the send (and its subsequent lookup) can be replaced by storing and calling procedures returned by how. In particular, transforming procedures provide an opportunity for this because the object they are transforming is generally known when the handlers they add to it are defined. If any of the new handlers refer to the original object's way of handling the message, this reference can be computed before the handler itself is constructed. We could apply this transformation to BE-SQUARE:

\[
\text{(define (fast-be-square turtle)}
\begin{align*}
& \text{ (let ((forward (turtle 'how 'forward))}
& \text{ (right (turtle 'how 'right)))}
& \text{ (define (draw-square size)}
& \text{ (forward size) (right 90)}
& \text{ (forward size) (right 90)}
& \text{ (forward size) (right 90)}
& \text{ (forward size)}
& \text{ (turtle 'handle 'square draw-square))}
\end{align*}
\]

\(^3\) A more complete implementation, as we shall see later, will have an ignoramus object which at least knows the how of how.
The resulting turtle will draw squares more quickly because it need not look up the handlers for forward and right. It is easy to imagine syntactic extensions which make such handler references easy; for instance, a with-handlers special form might fetch the handlers for particular messages and bind them locally within its scope. Section 14 discusses several optimization techniques based on how; but the semantics of how still present some interesting problems.

One such difficulty is the interaction of how messages with comprehensive transforming procedures like BE-VISIBLE, SCHIZOID or SHARE. The next section considers these issues and presents a solution which provides a precise semantics for how and describes how comprehensive transforming procedures can be written to preserve that semantics.

9 The Problems with HOW

When objects are simply accumulations of handlers constructed by the repeated composition of simple transforming procedures, the handling of the how message is a straightforward lookup of message names. However, in the presence of comprehensive transforming procedures, the meaning of how becomes complicated by the additional semantics placed on every message by such procedures.

The comprehensive transforming procedures presented as examples in Section 6 are particularly tricky in this regard. How should an object constructed by SHARE — which might be internally transformed at any point in the future — respond to the how message: by returning the current handler for a particular message or by returning a delayed pointer to a handler yet to be? Or consider objects constructed by SCHIZOID, which alternates between the ultimate targets of the messages it receives. Which of a SCHIZOID object’s ‘personalities’ should be used to provide the handler for a particular message?

One way to finesse the issue is to construct a handler which sends the message being asked about; thus, a comprehensive transforming procedure like SCHIZOID or SHARE would include the following case in its dispatcher:

```scheme
(define how-handler
  (lambda (how)
    (apply (car arguments) args))))
```

I will refer to this treatment of how as ‘delayed-message semantics’. For SHARE, this fragment would implement the interpretation that how return a procedure which corresponds to the shared object’s eventual methods for dealing with the message; for SCHIZOID, the returned ‘handler’ would refer to the two component ‘personalities’ in just as erratic a manner as the original object.

Delayed method semantics assures a uniform treatment of how in the soft object paradigm. Alternative semantics for how can be specially provided for by other messages introduced by comprehensive transforming procedures. For instance, SHARE could support a how-now message which returns the current handler for a message and SCHIZOID could define how-jekyll and how-hyde messages which refer how messages to the appropriate objects.

But implementing delayed message semantics in the straightforward fashion introduces some undesirable effects; in particular, the use of these handlers in optimization becomes a burden rather than a bonus, because they insert an additional procedure call before the message lookup rather than removing the message lookup altogether. We would like comprehensive transformers to return a result from how which retains the correct semantics while continuing to provide some enhanced efficiency. But since a comprehensive transforming procedure can impose any transformation on its input messages, there is no way to consistently provide an optimization of that transformation.

One solution is to push the burden of specializing how onto the comprehensive transforming procedures. Since every comprehensive transforming procedure must already handle the how mes-
sage in some fashion, we can simply have particular comprehensive transformers provide optimized results for how messages.

To illustrate this, let us return to the comprehensive transformer BE-VISIBLE of Section 5. Since all messages handled by visible turtles are forwarded to a single object, it is possible to return optimized values from how. We can extend the comprehensive transformer BE-VISIBLE to handle how.

(define (be-visible turtle)
  (define (wrap-handler-with-redisplay handler)
    (lambda args
      (wrapper (lambda () (erase-turtle turtle))
        (lambda () (apply handler args))
        (lambda () (draw-turtle turtle))))))

(define (visible-turtle message . arguments)
  (if (eq? message 'handle) (make-new-object turtle arguments)
    (if (eq? message 'how)
      (let ((handler (turtle 'how (car arguments))))
        (if handler
          (wrap-handler-with-redisplay handler)
          #F))
      (wrapper (lambda () (erase-turtle turtle))
        (lambda () (apply turtle message arguments))
        (lambda () (draw-turtle turtle))))))

(draw-turtle turtle)

visible-turtle)

This version of BE-VISIBLE returns procedures from how which wrap the execution of turtle’s handler in an erasure and redisplay. This is much like the way the non-comprehensive implementation of BE-VISIBLE wrapped each primitive movement message (forward, right, etc) with erasures and redisplay.

We can abstract this piece of code into a procedure which generates comprehensive transformers that wrap an arbitrary preamble and postamble around each message send, yet return optimized results to the how message. This implements an abstraction of the cliche implemented by the above definition of BE-VISIBLE:

(define (before-and-after preamble postamble)
  (lambda (old-object)
    (define (object message . arguments)
      (cond ((eq? message 'handle) (make-new-object object arguments))
        ;; The how message constructs a new procedure which
        ;; calls preamble and postamble around the inherited method.
        ((eq? message 'how)
          (let ((method (old-object 'how (car arguments))))
            (lambda args
              (wrapper preamble
                (lambda () (apply method args))
                postamble))))
          (ELSE (wrapper preamble
            (lambda () (apply old-object message arguments))
            postamble))))

object))

13
BEFORE-AND-AFTER returns a comprehensive transforming procedure which wraps an object’s message passes with a preamble and postamble while returning results from the how message which avoid searching for inner handlers for wrapped messages.

Given this abstraction of transforming procedures, we could have defined the BE-VISIBLE comprehensive transformer as follows:

\[
\begin{align*}
&\text{(define (be-visible turtle)} \\
&\quad \text{((before-and-after)} \\
&\quad \quad \text{(lambda () (erase-turtle turtle)))} \\
&\quad \quad \text{(lambda () (draw-turtle turtle))}} \\
&\quad \text{turtle))}
\end{align*}
\]

The turtle objects returned by BE-VISIBLE would handle the handle messages as before but, in response to the how message, returns a newly consed procedure which satisfies the delayed message semantics without an implicit message send and handler lookup hiding within it.

In conclusion, we adopt a consistent delayed message semantics for the how message; this adoption demands that the following equivalence hold:

\[
\text{(object 'how message) } \equiv \text{ (lambda args (apply object message args))}
\]

Individual comprehensive transformers may support special messages which return internal handlers with other semantics, but these are particular to the sorts of transformations they perform. Finally, a comprehensive transformer may choose to optimize the procedures returned by how; generating procedures which can be defined which generate classes of such comprehensive transformers.

10 Redefining HANDLE

Why might we wish to redefine the way objects handle the handle message? Suppose we wanted all subsequently defined handlers — for a particular object — to actually execute some preamble and postamble code. This is like BEFORE-AND-AFTER in the future tense; all future handlers, as opposed to all existing handlers, have a modified behavior. Thus we might define a transforming procedure:

\[
\begin{align*}
&(\text{define (buggy-wrap-future-handlers object preamble postamble)}} \\
&(\text{define (wrap-handler handler)} \\
&\quad \text{(lambda args)} \\
&\quad \quad \text{(wrapper preamble}} \\
&\quad \quad \quad \text{(lambda () (apply handler args))} \\
&\quad \quad \text{postamble)))} \\
&(\text{define (wrap-all-handlers handler-list)} \\
&\quad \text{(if (null? handler-list)} \\
&\quad \quad \text{()}} \\
&\quad \quad \text{(cons (car handler-list)} \\
&\quad \quad \quad \text{(cons (wrap-handler (cadr handler-list)} \\
&\quad \quad \quad \quad \text{(wrap-all-handlers (cddr handler-list)))))} \\
&\quad \text{(wrap-all-handlers (cddr handler-list))))} \\
&(\text{object 'handle 'handle}} \\
&\quad \text{(lambda handler-list}} \\
&\quad \quad \text{(apply object 'handle (wrap-all-handlers handler-list))}))
\end{align*}
\]

This procedure is supposed to return an object which wraps all handlers passed to it with some preamble and postamble. The problem with this definition is the fact that the binding of object in the definition is the object originally transformed by buggy-wrap-future-handlers and not
whatever object — somewhere down the line — will receive a handle message. The underlying difficulty is that a new handling of handle is introduced with each new object by make-new-object; simply defining a new version of handle at some particular level will not propagate to outer levels of definition.

To allow the redefinition of the handle message, we must pass a pointer to the outermost object receiving the handle message. This self pointer — so named because it refers to the object which originally received the message — must be sent along as messages are referred towards the ur-object.

Given the need for a passed in self-pointer, there are two implementation options: give an extra 'self-pointer' argument to the handle message in particular or modify the message protocol so that every top-level message passes along a self-pointer. Passing a long a self pointer might be useful for other purposes, but introduces significant overhead for each message pass; for simplicity's sake, we'll examine the addition of the 'self' argument to only

Adding an extra argument only to handle is by far the simplest approach; we remove the handle case from objects constructed by make-new-object and add a single handle handler to ur-object.

```
(define (make-new-object old-object handlers)
  (define (object message . arguments)
    (if (eq? message 'how)
      (or (mget (car arguments) handlers)
          (old-object 'how (car arguments)))
      (let ((local-handler (mget message handlers)))
        (if local-handler
            (apply local-handler arguments)
            (apply old-object message arguments))))

  (define (handle-handle self . handlers)
    (make-new-object self handlers))

  (define ur-object
    (make-new-object ignoramus (list 'handle handle-handle)))

In this implementation, the first argument to a handle message is a self pointer; it is used as the base for the new object constructed by handle.

In this framework, we could implement wrap-future-handlers:

```
(define (wrap-future-handlers object preamble postamble)
  (define (wrap-handler handler)
    (lambda args
      (wrapper preamble
        (lambda () (apply handler args)
          postamble)))

  (define (wrap-all-handlers handler-list)
    (if (null? handler-list) ()
      (cons (car handler-list)
        (cons (wrap-handler (cadr handler-list))
          (wrap-all-handlers (cddr handler-list))))))

  (object 'handle object 'handle
    (lambda (self . handler-list)
      (apply object 'handle self
        (wrap-all-handlers handler-list))))
```
In addition to redefining `handle`, this general approach can be used to implement specializations of `handle` which specifying particular sorts of redefinitions. For instance, we might implement `HANDLE-FOLLOW` thus:

```
(define (handle-follow object)
  (object 'handle object 'follow
    (lambda (self message follow-by . follow-args)
      (self 'handle self message
        (lambda args
          (let ((result (apply self message args)))
            (apply self follow-by follow-args)
            result))))))
```

allowing us (to reprise the introductory example of Section 1) to specify:

```
(define window-2 (window-1 'follow window-1 'expose
  'beep 150 .5))
```

to produce a window that beeps when it has been exposed (assuming that the window provided as `WINDOW-1` had been at some point transformed by `HANDLE-FOLLOW`).

A simple change to `follow` would use `how` to fetch direct handlers for both the original handler for the message and a curried call to the `follow-by`:

```
(define (handle-follow object)
  (object 'handle object 'follow
    (lambda (self message follow-by . follow-args)
      (let ((main (self 'how message))
            (follow (self 'how follow-by)))
        (self 'handle self message
          (lambda args
            (let ((result (apply main args)))
              (apply follow follow-args)
              result))))))
```

How about redefining the handler for `how`? This is a little bit more complicated because, unlike `handle` the behavior of `how` is changed whenever a new message is defined. When we add a handler for a message `foo` we are also adding a new handler for `how` which responds differently on the message `foo`.

Allowing the redefinition of a message like `handle` or `how` demands that we be able to implement the message with a single procedure (the new definition) whose arguments reflect the relevant properties `particular` to the object receiving the message. In the case of `handle`, the only relevant property was the identity of the object receiving the message; but in the case of `how`, the relevant properties are the individual sets of handlers defined by each transforming procedure in the object’s history.

One could implement a ‘redefinable `how`’ by accumulating and passing in the handlers established by each transforming procedure, but such an implementation would be quite awkward and inefficient. Furthermore, as currently implemented, the semantics of `how` consistently reflect the semantics of messages in general. Allowing the arbitrary redefinition of `how` threatens to violate this consistency to little advantage.
11 How how

In the previous section, we examined the issues in using handle to redefine the way the handle
message itself is handled. In this section, we describe the extension of how to handle the how
message.

We can extend the handling of how introduced above:

\[
\begin{align*}
&(\text{if (eq? message 'how)} \\
&(\text{or (mget (car arguments) handlers)} \\
&(\text{old-object 'how (car arguments))}))
\end{align*}
\]

to include a special case for the how of how:

\[
\begin{align*}
&(\text{if (eq? message 'how)} \\
&(\text{if (eq? (car arguments) 'how)} \\
&(\text{lambda (msg) (object 'how msg))} \\
&(\text{or (mget (car arguments) handlers)} \\
&(\text{old-object 'how (car arguments))}))))
\end{align*}
\]

which simply returns a procedure that describes the delayed message semantics. With this extension
the new definition of make-new-object looks like this:

\[
\begin{align*}
&(\text{define (make-new-object old-object handlers)} \\
&(\text{define (object message . arguments)} \\
&(\text{if (eq? message 'how)} \\
&(\text{if (eq? (car arguments) 'how)} \\
&(\text{lambda (msg) (object 'how msg))} \\
&(\text{or (mget (car arguments) handlers)} \\
&(\text{old-object 'how (car arguments))}))))
\end{align*}
\]

\[
\begin{align*}
&(\text{let ((local-handler (mget message handlers)}) \\
&(\text{if local-handler (apply local-handler arguments)} \\
&(\text{apply old-object message arguments))}))
\end{align*}
\]

object)

The procedure returned by the how of how is precisely the same as the objects implemented in
Chapter 3 of [Abelson and Sussman, 1985]; it is a procedure which looks up handlers for a particular
message.

If we assume that the ignoramus procedure is not a 'proper' object:

\[
\begin{align*}
&(\text{define (ignoramus message . arguments)} \\
&(\text{(if (eq? message 'how) #f} \\
&(\text{(error "Can't handle message" message arguments))))}
\end{align*}
\]

we can be assured that all objects (including the ur-object as built by make-new-object, but not
the ignoramus procedure) will handle the how of how appropriately.

This implementation of how introduces the following equivalence of interest:

\[
\text{(obj 'how 'how) } \equiv \text{(proj 'how 'how) } \equiv \text{((proj 'how 'how) 'how) } . . .
\]

Or, in other words, the how message is the eigenvalue of the handler for the how message!
12 Restricted Objects

This section examines ways of restricting access to soft objects between modules; a given module may construct an object to be manipulated internally but pass a restricted object to outside modules. A simple comprehensive transforming procedure for producing such objects might be implemented like this:

```
(define (restrict object messages)
  (define (make-handlers messages)
    (if (null? messages) messages
      (cons (car messages)
        (cons (object 'how (car messages))
          (make-handlers (cdr messages)))))
    (apply ur-object 'handle ur-object (make-handlers messages)))

RESTRICT does not inherit messages in the same fashion as other transforming procedures; strictly speaking, RESTRICT is not a transforming procedure at all but is rather a sort of 'reforming procedure' which returns a new object that delegates [Lieberman, 1986] particular functions to the unrestricted object. An advantage of constructing a wholly new object is that the restricting procedure need not restrict both each message in general and the how message in particular. Were access to the original object's how provided, a program could 'cheat' and execute a restricted message by:

```
((object 'how restricted-message) arguments...)
```

An interesting variation on RESTRICT uses an alternative ur-object which provides a more useful failure message:

```
(define (restrict object messages)
  (define (useful-ignoramus message . arguments)
    (if (object 'how message)
      (error "Message is restricted, sorry." message arguments)
      (error "Don't know how to handle message" message arguments)))
  (define (make-handlers messages)
    (if (null? messages) messages
      (cons (car messages)
        (cons (object 'how (car messages))
          (make-handlers (cdr messages))))
    (let ((guard-object
          (make-new-object useful-ignoramus (list handle handle-handle)))
          (apply guard-object 'handle guard-object (make-handlers messages))))

This tells the user whether a message pass failed because of a restriction; we must use a newly constructed ur-object equivalent because there is no other way to change the way a failed message lookup is handled.

If each message invocation pass a self pointer as handle does — an option we considered in Section 10, we could have the ignoramus procedure send the current self a lookup-failure message which could be redefined for different sorts of objects.

13 Restricting Redefinition

One other interesting sort of restriction — particularly important when objects are being shared between programs and processes — is restricting the transformations applicable to an object. The
ability to redefine the handling of `handle` allows us to limit or specialize the function of `handle`. For instance, we might restrict the `redefinition` of messages by `handle` to a particular set of messages:

\[
\begin{align*}
&\text{(define (restrict-redefinition for-object to-messages)} \\
&(\text{define (new-handle self . handlers)}) \\
&(\text{define (check-handler-list handlers)} \\
&(\text{cond ((null? handlers) #t)}) \\
&\quad ;; \text{Handler is ok to redefine} \\
&\quad ((\text{memq (car handlers) to-messages)} \\
&\quad (\text{check-handler-list (cdr (cdr handlers))})) \\
&\quad ;; \text{Handler is already defined; signal an error.} \\
&\quad ((\text{for-object 'how (car handlers)} \\
&\quad (\text{error "Can't redefine message" (car handlers) self}) \\
&\quad ;; \text{Handler must not be defined currently, continue.} \\
&\quad (\text{ELSE (check-handler-list (cdr (cdr handlers))}))))) \\
&(\text{check-handler-list handlers)} \\
&(\text{apply for-object 'handle self handlers)}) \\
&(\text{for-object 'handle for-object 'handle new-handle)})
\end{align*}
\]

This provides a new definition of `handle` which checks the messages being defined to ensure that any message redefinitions confine themselves to a limited range of messages.

We could combine the restrictions imposed by `RESTRICT-REDEFINITION` with the specialized functionality provided by `follow` to produce objects whose redefinitions are limited in both subject and style. The two distinct combinations

\[
\begin{align*}
&(\text{define (restrict-1 object to-messages)} \\
&(\text{restrict-redefinition (handle-follow object) to-messages})) \\
&(\text{define (restrict-2 object to-messages)} \\
&(\text{handle-follow (restrict-redefinition object to-messages))})
\end{align*}
\]

each provide different restrictions. `RESTRICT-1` permits any message to be ‘followed’ by another, but only allows particular messages to be wholly redefined. `RESTRICT-2` restricts both the ‘follow’ extension and complete redefinition to a particular set of messages.

It is probably a wise approach to only `SHARE` objects which are restricted in some fashion akin to this. But even this will not ensure security, since a transforming procedure could add messages to an object by defining a new procedure and avoiding the restricted `handle` message entirely.

14 Optimizations: Saving Time

The `how` message allows message lookup to be factored out of message sending and receipt whenever the object and message are both known. This can be used for both static and dynamic optimizations, either in a compiler or ‘syntaxter’ before execution time or in objects which, at execution time, anticipate and prepare for future executions of similar messages.

The static optimizations possible with `how` were briefly demonstrated in Section 8. For instance, the procedure `BE-SQUARE` could provide an ‘optimized’ implementation of `square` which factored out the lookup for basic movement messages:

\[4\text{This version of `BE-SQUARE` also uses the new form of `handle` (passing along a `self` pointer) introduced in Section 10.}\]
(define (be-square turtle)
  (let ((forward (turtle 'how 'forward))
         (right (turtle 'how 'right)))
    (define (draw-square size)
      (define (draw-side) (forward size) (right 90))
      (draw-side) (draw-side) (draw-side) (draw-side))
    (turtle 'handle turtle 'square draw-square)))

This approach might be abstracted into a procedure WITH-HANDLERS-FOR which takes an object, a list of messages, and a procedure to call on the handlers for those messages:

(define (with-handlers-for object messages procedure)
  (apply procedure (map (lambda (message) (object 'how message))
                         messages)))

;;; Allowing us to define BE-SQUARE thus:
(define (be-square turtle)
  (with-handlers-for turtle '(forward right)
    (lambda (forward right)
      (define (draw-square size)
        (forward size) (right 90)
        (forward size) (right 90)
        (forward size) (right 90)
        (forward size))
      (turtle 'handle turtle 'square draw-square))))

But the explicit optimization of WITH-HANDLERS-FOR is not strictly necessary, since a compiler could perform the transformation which a programmer applies when she recognizes that both object and message are known at handler construction time. Such an optimization would transform a lambda expression like:

(lambda (size)
  (define (draw-side) (turtle 'forward size) (turtle 'right 90))
  (draw-side) (draw-side) (draw-side) (draw-side))

into an expression which uses the definition of turtle already in the environment:

(let ((forward (turtle 'how 'forward)) (right (turtle 'how 'right))
        (lambda (size)
          (define (draw-side) (forward size) (right 90))
          (draw-side) (draw-side) (draw-side) (draw-side)))

All that is necessary is to recognize that turtle is an object which can accept the how message; this could be managed by either explicit declaration, compiler analysis (to some limited extent), or dispatching at execution time.

How does this sort of optimization interact with the share comprehensive transformer? What happens if we request a handler from a shared object (using how) and the transform the shared object. We can finesse the issue by just implementing the handling of how by a direct (albeit inefficient) specification of the desired message semantics:
(define (share object)
  (define (shared-object message . arguments)
    (cond ((eq? message 'handle)
           (error "I can only CHANGE inside!"))
           ((eq? message 'how)
            (let ((msg (car arguments)))
              (lambda args (apply shared-object msg args))))
           ((eq? message 'change)
            (set! object (apply (car arguments) object (cdr arguments)))
            object)
           (ELSE (apply object message arguments))))
  shared-object)

but we can do better by returning a handler from how which stores a cached handler and assures
the validity of the handler by checking if it's internal pointer to object has changed:

(define (share object)
  (define (shared-object message . arguments)
    (cond ((eq? message 'handle)
           (error "I can only CHANGE inside!"))
           ((eq? message 'how)
            (let ((handler (object how (car arguments)))
                  (for-obj object))
              (lambda args
                        (cond ((eq? object for-obj) (apply handler args))
                              (ELSE (set! handler (object how (car arguments)))
                                    (set! for-obj object)
                                    (apply handler args)))))))
           ((eq? message 'change)
            (set! object (apply (car arguments) object (cdr arguments)))
            object)
           (ELSE (apply object message arguments))))
  shared-object)

The handler returned by how keeps track of the most recently valid implementation for the message
and the precise object for which it was valid. If the object changes, the handler is re-fetched.
Note that such an implementation would not be possible if transforming procedures were not strict
functions; if a transforming procedure changed objects internally, there would be no way to detect
such changes.

The methods above all provide static optimizations of the execution of individual handlers, but
do not provide for faster execution of messages which must be looked up at execution time. As
implemented with linear handler lists and MGET (Section 3), sending a message takes \(O(n)\) time in
the number of messages defined for the object. This linear data structure could be improved upon
and one such dynamic improvment could use a faster implementation-specific 'lookup' technology:
(define (make-faster object)
  (let ((table (make-fast-lookup-table)))
    (define (fast-object message . arguments)
      (let ((fast-handler (lookup message table)))
        (if fast-handler (apply fast-handler arguments)
          (let ((slow-handler (fast-object 'how message)))
            (lookup-store! table message slow-handler)
            (apply slow-handler arguments)))))))

MAKE-FASTER is a transforming procedure which, rather than returning an object with different semantics, returns a new object whose performance on messages would improve over time, as more and more messages got moved into the fast lookup table internal to the object.

An object transformed by MAKE-FASTER would accept the handle message to produce new objects, but handlers defined for such objects would still be looked up using MGET’s O(n) search. Such objects would still be faster in processing messages which referred to other messages, but the cost of the initial message lookup would remain. Such a transformed object could eventually be transformed by MAKE-FASTER again, but repeated applications of MAKE-FASTER might produce a significant space overhead. (The next section discusses the issues of space overhead engendered by soft objects.)

An implementation independent approach to speeding up soft objects would be to provide a transforming procedure which installs a fixed size message cache in front of an object; Deutsch [Deutsch and Schiffman, 1984] reports that even caching a single message produced enormous improvements in the performance of a Smalltalk system. We could implement a transforming procedure cache-messages like this:

(define (cache-messages object)
  (let ((cache (make-message-cache 10)))
    (define (caching-object message . arguments)
      (let ((cache-entry (cache-get message cache)))
        (if cache-entry
          (apply cache-entry arguments)
          (let ((handler (object 'how message)))
            (if (false? handler)
              (apply object message arguments)
              (sequence (cache-put message handler cache)
                        (apply handler arguments)))))))
    caching-object))

In this definition, the procedures MAKE-MESSAGE-CACHE, CACHE-GET and CACHE-PUT all access some storage data structure which implements a particular caching strategy. As an example, we might implement a ‘least commonly used’ aging strategy which uses a list of cache entries where each entry consists of a message name, its handler, and a usage count.

(define (make-message-cache size)
  (cons (list #F #F 0) (make-message-cache (- size 1))))
(define (cached-message cache-entry) (list-ref cache-entry 2))
(define (cached-handler cache-entry) (list-ref cache-entry 2))
(define (cache-usage cache-entry) (list-ref cache-entry 2))
(define (use-cache-entry cache-entry)
  (set-car! (list-tail cache-entry 2) (1+ (cache-usage cache-entry))))
Using this structure, the CACHE-GET procedure simply looks up a message in the cache and advances its usage count:

```
(define (cache-get message cache)
  (cond ((null? cache) #f)
        ((eq? (cached-message (first cache)) message)
          ;; Advance the usage count.
          (use-cache-entry (first cache))
          (cached-handler (first cache)))
        (else (cache-get message (rest cache))))
```

The CACHE-PUT procedure finds the message with the lowest usage count and replaces it with the message and handler it is storing:

```
(define (cache-put message handler cache)
  (define (get-entry-to-replace entry cache-entries)
    (if (null? cache-entries) entry
      (if (>= (cache-usage (first cache-entries)) (cache-usage entry))
        (get-entry-to-replace entry (rest cache-entries))
        (get-entry-to-replace (first cache-entries) (rest cache-entries))))
  (let ((entry-to-replace (get-entry-to-replace (first cache) (rest cache)))
        (set-car! entry-to-replace message)
        (set-car! (cdr entry-to-replace) handler)
        (set-car! (cddr entry-to-replace) 1)))
```

15 Optimizations: Saving Space

In the sections above, one unaddressed problem was the space overhead of soft objects. Unlike class oriented schemes, soft objects use space linear in the number of messages they handle and the state required by those messages. In a class oriented scheme, where messages are associated with classes, the space taken for an object is only linear in the state required by the messages the object accepts. For objects which accept \( m \) messages using \( n \) ‘units’ of internal state, soft objects require space \( O(m + n) \) while class based schemes only require space \( O(n) \).

The worst case for soft objects — relative to class-based object systems — is a large number of small-state objects each of which accepts a large number of messages. A class based system would need only to represent the state for each object, while soft objects would need to represent both the state and each object’s individual handlers for the messages referencing that state.

Ameliorating this problem will require more extensive machinery than any introduced thus far. Rather than detail all of the machinery, an approach involving a sophisticated compiler can be sketched.

Consider a particular object created by a series of transforming procedures; if we perform beta-substitution on these transforming procedures and their internal calls to handle (assuming for the moment a fixed semantics for handle), we get a massive procedure which calls mget on a series of constant handler lists made up of references to a set of handler procedures closed in a series of overlapping environments. Through appropriate renaming of variables, we can produce a program which closes all of the handlers in the same environment. Using some clever understanding of MGET, we can reduce this call with a constant handler list to a complicated conditional expression. This expression is equivalent to our created object when closed in the appropriate environment. Given these optimizations, a given object need only take a small constant more space than its size as an environment, just as in a class-based object system. Since the handler names in the list are constant,
the conditional expression can actually use a wide variety of more efficient search procedures for finding the correct handler, yielding an implementation which is more efficient in both space and time.

Admittedly, this is a lot of work for a compiler to go through. However, if it is driven by either user declarations or frequency analyses, it could choose to expand a selected subset of commonly used compositions of transforming procedures.

In the case where an object’s handlers for handle may be redefined, a transforming procedure optimized in this way would have to check the handle semantics for the object it is transforming. If the object’s handling of handle were non-standard, it could either use a space-expensive implementation, or continue with the optimization by substituting the special definition of handle into the transforming procedures it is optimizing and seeing if the techniques of merging environments together and optimizing MGET will succeed with the new substitution.

Even without optimizations like these, in many cases the space costs of soft objects are not severe. Excluding of the worst case of many small-state objects handling many messages, soft objects are reasonably efficient in terms of both space and time. Many applications of object oriented systems, such as window systems or small personal databases, maintain a significant state to messages ratio for a moderate number of objects (in the hundreds).

16 Using Anonymous Messages

In the preceding descriptions, we have assumed that messages are identified by symbols; this simplified the presentation and made it easy to distinguish what expressions were messages (the quoted ones). However, the use of symbols to identify messages suffers from a ‘collision problem’; two independent modules might specify distinct messages with the same name. If the modules ever interact, this is a sure recipe for chaos. Following Rees and Adams [Adams and Rees, 1988], this section describes how soft objects can be extended to use ‘anonymous messages’ defined locally to particular modules.

The approach of Rees and Adams is to use constructed procedures to denote messages; these procedures then serve double-duty as generic functions which send themselves as messages. To be precise, a message like expose would be implemented by a procedure like this:

\[(\text{define } \text{(expose window)} \text{(window expose))}\]

The procedure expose (the value of the identifier expose) can be used to identify the expose message. Different message-procedures can be constructed locally to a program module and thus avoid the name conflicts involved in using symbols as message identifiers. And since the message identifier is a generic function we can say either

\[(\text{window expose)} \text{ or } \text{(expose window)}\]

We can define a generator make-message which constructs procedures like expose:

\[(\text{define make-message)}\]
\[\text{(define message object . arguments)}\]
\[\text{(apply object message arguments)}\]
\[\text{message)}\]

Assuming that one’s compiler is not clever enough to reduce all messages to the same procedure, the returned procedure serves as a perfectly adequate tag for messages, allowing us to define messages thus:
(define handle (make-message))
(define how (make-message))
(define transform (make-message))
(define pen-state (make-message))

Alternatively, we can define a syntactic extension define-message which expands into a definition which preserves the message name:

(define-message msg)
   is equivalent to
   (define (msg object . args) (apply object msg args))

In some SCHEME implementations, this has the advantage that the message object foo will print out in a fashion which identifies the message name it corresponds to.

If we are handling the definition of handle-like messages by passing a self pointer along, we can implement the corresponding generic function so that it uses the object operated on as a self pointer:

(define (handle object . handlers)
   (apply object handle object handlers))

This always use the object originally receiving the message as a self pointer. However, we must not use this generic function directly when redefining the handle or related messages, since we must preserve the original self pointer instead. We can do this by simply sending the handle message rather than calling the handle procedure.

A useful syntactic extension for defining handle-like messages is the syntax define-meta-message which expands into a definition like handle above:

(define-meta-message msg)
   is equivalent to
   (define (msg object . arguments)
      (apply object msg object arguments))

One advantage of using symbols to identify messages (as opposed to using anonymous identifiers) is the textual composability of symbols. For instance, in Section 2, we defined a transforming procedure have-property which constructed the messages property and SET-property!:

(define (have-property object property)
   (let ((value #f))
      (object 'handle
          property (lambda () value)
          (symbol-append 'set-property ')
          (lambda (new-value) (set! value new-value)))))

If messages are objects, rather than symbols, such construction of message names will be problematic. But by having have-property be passed two messages (a getter and a setter), rather than a symbolic property name, is just as effective and ultimately safer, since it avoids message name conflicts between distinct modules of a program. Such an implementation would look like this:

(define (have-property object getter setter)
   (let ((value #f))
      ;; We presume here that handle is now bound to an anonymous message.
      (object handle
          getter (lambda () value)
          setter (lambda (v) (set! value v)))))

25
17 An Example Implementation

This implementation uses message objects as described above; in particular, it assumes that the forms define-message and define-meta-message exist. In MIT Scheme (and several other dialects), these could be defined thus:

\[
\begin{align*}
&\text{(define-macro (define-message name)} \\
&\quad\text{'(define ,(name object . args)} \\
&\quad\text{\hspace{1cm} (apply object ,name args)))} \\
&\text{(define-macro (define-meta-message name)} \\
&\quad\text{'(define ,(name object . args)} \\
&\quad\text{\hspace{1cm} (apply object ,name object args)))}
\end{align*}
\]

And so we begin,

; Defining the two central messages.
(define-meta-message handle)
(define-message how)
; Making new objects.
(define (make-new-object old-object handlers)
  
  (define (object message . arguments)
    
    (if (eq? message how)
      
      (if (eq? (car arguments) how)
        
        (lambda (msg) (object how msg))
      
      (or (mget (car arguments) handlers)
        
        (old-object how msg)))
    
    (let (((local-handler (mget message handlers)))
      
      (if local-handler
        
        (apply local-handler arguments)
      
      (apply old-object message arguments))))

    object)

; Looking up handlers (an \(O(n)\) proposition.
(define (mget message handlers)
  
  (if (null? handlers) #F

    (if (eq? (car handlers) message) (cadr handlers)
      
      (mget message (cddr handlers))))

; The not-quite-an-object that knows nothing.
(define (ignoramus message . arguments)
  
  (if (eq? message how)
    
    (if (eq? (car arguments) how)
      
      (lambda (msg) (ignoramus how msg))

      #F)

    (error "Can't handle message" message arguments))

; This handles the handle message by calling MAKE-NEW-OBJECT.
(define (handle-handle self . handlers)
  
  (make-new-object self handlers))

; The base object upon which other objects are built.
(define ur-object
  
  (make-new-object ignoramus (list handle handle-handle)))
17.1 Sharing objects

; A version of SHARE.
(define-message change)
(define (share object)
  (define (shared-object message . arguments)
    (cond ((eq? message handle)
          (error "I can only CHANGE inside!"))
          ((eq? message change)
           (set! object (apply (car arguments) object (cdr arguments)))
           object)
          (else (apply object message arguments))))
  shared-object)

17.2 Restricting messages and extensions

; A version of RESTRICT.
(define (restrict object messages)
  (define (useful-ignoramus message . arguments)
    (if (object 'how message)
      (error "Message is restricted, sorry." message arguments)
      (error "Don't know how to handle message" message arguments))
  (define (make-handlers messages)
    (if (null? messages) messages
      (cons (car messages)
            (cons (object 'how (car messages))
                  (make-handlers (cdr messages))))))
  (let ((guard-object
         (make-new-object useful-ignoramus (list handle handle-handle)))
         (apply guard-object 'handle guard-object (make-handlers messages))))
; A version of RESTRICT-REdefinition.
(define (restrict-redefinition for-object to-messages)
  (define (new-handle self . handlers)
    (define (check-handler-list handlers)
      (cond ((null? handlers) #t)
        ;; Handler is ok to redefine
        ((memq (car handlers) to-messages)
         (check-handler-list (cdr (cdr handlers))))
        ;; Handler is already defined; signal an error.
        ((for-object how (car handlers))
         (error "Can't redefine message" (car handlers) self))
        ;; Handler must not be defined currently, continue.
        (else (check-handler-list (cdr (cdr handlers))))))
    (check-handler-list handlers)
    (apply for-object handle self handlers)
    (for-object handle for-object handle new-handle))

18 Bibliography

References


19 Acknowledgements

Alan Bawden and Gary Drescher read drafts of this paper and provided insightful comments; their help is much appreciated. I first began to consider issues about object-oriented programming in 1983, when working on QLOGO (an object oriented version of LOGO) with Gary Drescher at Atari's Cambridge Research Laboratory. The idea of 'transforming procedures' originally emerged as 'training procedures' in a project for a computer animation seminar organized by David Zeltzer of the MIT Media Lab.