Massachusetts Institute of Technology
Artificial Intelligence Laboratory

A.I. Memo ??? March 1989

***** DRAFT *****
Object-Oriented Programming in Scheme

Norman Adams

Jonathan Rees

Abstract

We describe a small set of additions to Scheme to support object-oriented programming, including a form of multiple inheritance. The extensions proposed are in keeping with the spirit of the Scheme language and consequently differ from Lisp-based object systems such as Flavors and the Common Lisp Object System. Our extensions mesh neatly with the underlying Scheme system. We motivate our design with examples, and then describe implementation techniques that yield efficiency comparable to dynamic object-oriented language implementations considered to be high performance. The complete design has an almost-portable implementation, and the core of this design comprises the object system used in T, a dialect of Scheme. The applicative bias of our approach is unusual in object-oriented programming systems.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contracts N00014-85-K-0124 and N00014-86-K-0180.

This is a revision of a paper that appeared in the *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming.*

1 Introduction and terminology

Scheme[15] is nearly an object-oriented language. This should come as no surprise, since Scheme was originally inspired by Actors, Hewitt's message-passing model of computation[22, 1]. Steele has described the relationship between Scheme and Actors at length[19]. We take advantage of this relationship—and we try not to duplicate functionality that Scheme already provides—to add full support for object-oriented programming. Our extensions are in keeping with the spirit of Scheme: "It was designed to have an exceptionally clear and simple semantics and few different ways to form expressions."[15]

We develop examples of how one might program in Scheme using object-oriented style. Inherited behavior is handled in a straightforward manner. We show that a new disjoint data type for instances must be added to Scheme in order to permit application of generic operations to all Scheme objects, and that making generic operations anonymous supports modularity. We complete our presentation by showing how to obtain good performance for generic operation invocation.

We use the terms *object* and *instance* idiosyncratically: an object is any Scheme value; an instance is a value returned by a constructor in the object system. An *operation*, sometimes called a "generic function," can be thought of as a procedure whose definition is distributed among the various objects that it operates on. The distributed pieces of the definition are called "methods."

2 Object-oriented programming using procedures

2.1 Simple objects

Our first approximation to object-oriented programming in Scheme is straightforward: an instance is represented as a procedure that maps operations to methods. A method is represented as a procedure that takes the arguments to the operation and performs the operation on the instance.

As an example, consider the following definition of a constructor for cells:

```
(define (make-simple-cell value)
  (lambda (selector)
    (cond ((eq? selector 'fetch)
           (lambda () value))
          ((eq? selector 'store!)
           (lambda (new-value)
             (set! value new-value)))
          ((eq? selector 'cell?)
           (lambda () #t))
          (else not-handled))))
```

```
(define a-cell (make-simple-cell 13))
((a-cell 'fetch))        →  13
((a-cell 'store!) 21)    →  unspecified
((a-cell 'cell?))        →  true
((a-cell 'foo))          →  error
```

Each call to `make-simple-cell` returns a new cell. Abstract operations are represented by symbols, here `fetch`, `store`, and `cell?`. An instance is represented by the procedure returned by the constructor. The lexical variables referenced by the methods serve the purpose of instance variables. To perform an operation on an instance, the instance is called, passing the operation as the argument; the resulting method is then applied to the arguments of the operation. There is no explicit notion of class, but the code portion of the closure constructed for the expression

```
(lambda (selector) ...)
```

serves a similar purpose: it is static information shared by all instances returned by the `make-simple-cell` constructor.

An instance returns the object `not-handled` instead of a method to indicate it defines no behavior for the operation. The particular value of `not-handled` is immaterial, as long as it is identifiable as being something other than a method. We will make use of this property later.

```
(define not-handled (list 'not-handled))
```

To improve the readability of operation invocation, we introduce the procedure `operate`:

```
(define (operate selector the-instance . args)
  (apply (the-instance selector) args))
```

This lets us replace `((a-cell 'store!) 34)` with the somewhat more perspicuous

```
(operate 'store! a-cell 34).
```

`Operate` is analogous to `send` in old Flavors[23] and `=>` in CommonObjects[16].

2.2 Inheritance

The following defines a "named" cell that inherits behavior from a simple cell.

```
(define make-named-cell
  (lambda (value the-name)
    (let ((s-cell (make-simple-cell value)))
      (lambda (selector)
        (cond ((eq? selector 'name)
               (lambda () the-name))
              (else (s-cell selector)))))))
```

We say that objects returned by `make-named-cell` have two *components*: the expression `(make-simple-cell ...)` yields the first component, and the expression `(lambda (selector) ...)` yields the second. The programmer controls what state is shared by choosing the arguments passed to the constructors, and by choosing the expressions used to create the components. In this style of inheritance, only behavior is inherited. An instance can name its components, but can assume nothing about how they are implemented.

We have single inheritance if the instance consults one component (not including itself) for behavior. We have multiple inheritance if the instance consults multiple components (not including itself) for behavior. For the above example, that might be done by expanding the `else` clause as follows:

```
        ⋮
    (else (let ((method (s-cell selector)))
            (cond ((eq? method not-handled)
                   (another-component selector))
                  (else method))))
```

This is similar to the style of inheritance in CommonObjects[16] in that instance variables are considered private to the component, and cannot be inherited. However, CommonObjects forces the components to be new, where our formulation has no such requirement; thus distinct instances may share components and, in turn, the components' state. Because classes are not explicit, and it is possible to share state and behavior, one may say this approach provides delegation rather than inheritance[11, 21]. When more than one component defines behavior for an operation, the behavior from the more specific component (the one consulted first) shadows the less specific.

2.3 Operations on self

One frequently wants to write a method that performs further operations on the instance. For example, if we were implementing an open output file as an instance that supported `write-char`, we could implement a `write-line` operation as a loop performing `write-char` operations on the instance itself. In simple cases, a method can refer to the instance by using `letrec` in the instance's implementation:

```
(letrec ((self
          (lambda (selector)
            (cond ((eq? selector 'write-line)
                   (lambda (self string)
                     ...
                     (operate 'write-char self char)))
                  ...))))
  self)
```

When inheritance is involved this will not work. The variable `self` will not be in scope in the code for inherited methods. A component that uses `letrec` as above will be able to name its "component self", but not the composite. A small change to the protocol for invoking methods remedies this: the composite is passed as an argument to every method.

```
(define (operate selector the-instance . args)
  (apply (the-instance selector) the-instance args))
```

2.4 Operations on components

Because components may be given names, a composite's method for a given operation can be defined in terms of the behavior of one of its components. For example, we can define a kind of cell that ignores stores of values that do not satisfy a given predicate:

```
(define make-filtered-cell
  (lambda (value filter)
    (let ((s-cell (make-simple-cell value)))
      (lambda (selector)
        (cond ((eq? selector 'store!)
               (lambda (self new-value)
                 (if (filter new-value)
                     (operate 'store! s-cell new-value))))
              (else (s-cell selector)))))))
```

This practice is analogous to "sending to `super`" in Smalltalk and `call-next-method` in the Common Lisp Object System (CLOS)[5]

There is a problem here. When the composite performs an operation on a component, the "self" argument to the component's method will be the component, not the composite. While this is sometimes useful, we also need some way to perform an operation on a component, passing the composite as the "self" argument. (This is the customary operation of send to super and its analogues.) We can do this using a variant on `operate` that takes as arguments both the composite and the component from which a method is to be obtained:

```
(define (operate-as component selector composite . args)
  (apply (component selector) composite args))

(define (operate selector instance . args)
  (apply operate-as instance selector instance args))
```

3 Integration with the rest of the language

3.1 Operations on non-instances

One often wants to include non-instances in the domain of an abstract operation. For example, the operation `print`, when performed on a non-instance, should invoke a printer appropriate to that object's type. Similarly, the abstract operation `cell?` should apply to any object in the Scheme system, returning false if the object is not a cell. But given our present definition of `operate`, we cannot meaningfully say `(operate` *op* *x*`)` unless *x* is an instance.

We can make `print` work on non-instances if we can associate methods with every non-instance on which we expect to perform operations. This is accomplished with a simple change to `operate-as` (remember that `operate` is defined in terms of `operate-as`):

```
(define (operate-as component selector the-object . args)
  (apply (get-method component selector)
         the-object args))

(define (get-method component selector)
  (if (instance? component)
      (component selector)
      (get-non-instance-method component selector)))
```

We will consider the definition of `instance?` below. As for `get-non-instance-method`, we would like it to behave in a manner consistent with the dispatch routines we have set up for instances, which are specific to the particular kind of object being operated on.

```
(define (get-non-instance-method object selector)
  ((cond ((pair? object)
          get-method-for-pair)
         ((symbol? object)
          get-method-for-symbol)
         ... other types ...)
   object selector))
```

```
(define (get-method-for-pair pair selector)
  (cond ((eq? selector print)
          (lambda (self port) ...)
         ... other methods ...))

(define (get-method-for-symbol symbol selector)
  ...)
```

Unfortunately, any time we want a new operation to work on some kind of non-instance, we have to modify the appropriate non-instance dispatch routine to handle the new operation. This presents problems because these dispatch routines are global resources for which there may be contention. What if two different modules chose to define incompatible behavior for a shared operation? But even worse, in the case of the `cell?` operation, we would have to modify *all* dispatch code, even that for non-instances. That would hardly be modular.

One way to define a default behavior for `cell?` would be to have a "vanilla object," analogous to "class `object`" in many object systems, that acted as a component of every object, instance or non-instance. We could change the dispatch for that component every time we wanted to add a new operation to the system. This approach would also suffer the contention problem mentioned above.

Instead of using such a "vanilla object," we take the approach of associating default behavior with each operation. The default behavior is used when there is no method for the operation in the object being operated on. We change the definition of `operate-as` to implement this:

```
(define (operate-as component selector composite . args)
  (let ((method (get-method component selector)))
    (if (eq? method not-handled)
        (apply (default-behavior selector) composite args)
        (apply method composite args))))
```

The procedures `default-behavior` and `set-default-behavior!` maintain a table mapping selectors to procedures that implement the default behavior.

```
(set-default-behavior! 'cell? (lambda () #f))
(define a-cell (make-simple-cell 5))
(operate 'cell? a-cell)  →  true
(operate 'cell? 1)       →  false
```

Later we will see how to associate default behavior with operations without using side effects.

Returning to the predicate `instance?`: Our new version of `operate-as` indirectly uses `instance?` to determine whether the object is an instance. Defining `instance?` to be the same as `procedure?` almost works, but it fails to account for the distinction between those procedures that obey the protocols of the object system and those that don't: `(operate 'cell? list)` would generate an error instead of returning false.

To implement `instance?`, we need to be able to mark some procedures as being instances. But in Scheme, the only operations on procedures are application and identity comparison (`eq?`). So we must add a new primitive type to Scheme, disjoint from the types already provided, that satisfies the following rules:

`(instance-ref (make-instance` x`))` $\rightarrow$ x
`(instance? (make-instance` x`))` $\rightarrow$ true
`(instance?` y`)` $\rightarrow$ false, if y was not a value
returned by `make-instance`

Instance constructors such as `make-cell` must now mark instances using `make-instance`, and `get-method` must coerce instances to procedures using `instance-ref`.

```
(define (get-method component selector)
  (if (instance? component)
      ((instance-ref component) selector)
      (get-non-instance-method component selector)))

(define (make-simple-cell value)
  (make-instance (lambda (selector)
                   ...)))
```

3.2 Non-operations applied to instances

In purely object-oriented languages, all operations are generic. In our framework this would mean that a procedure call such as `(car x)` would be interpreted the same way as, say, `(operate 'car x)`. Thus the domain of `car` and other primitives would be extended to include all instances that choose to handle the corresponding generic operation.

We consider this kind of extension to be undesirable for a number of reasons, all of which stem from the fact that programs become harder to reason about when all operations are generic. For example, if a call to `car` can invoke an arbitrary method, then a compiler cannot assume that the call will not have side effects. If arithmetic can be extended to work on arbitrary objects, then algebraic properties such as associativity no longer hold. Not only is efficiency impaired, but programs can become harder for humans to understand as well.

In the case of some system procedures, however, extension to instances can be very useful for the sake of modularity. For example, procedures on I/O ports can probably be extended without significant impact on performance, and they may already be generic internally to the system. If such procedures are extended, the contract of the corresponding abstract operation must be made clear to those programmers who define methods for them.

One particular Scheme primitive can be usefully extended to instances without sacrificing efficiency or static properties: procedure call. This is because procedure call has no algebraic or other properties that are in danger of being destroyed; invoking a procedure can potentially do anything at all. So we can arrange for the Scheme implementation to treat a combination

(*proc* *arg* ...)

the same as

`(operate 'call` *proc* *arg* `...)`

whenever the object `proc` turns out to be an instance. The `call` operation should also be invoked whenever an instance is called indirectly via `apply` or any other system procedure that takes a procedure argument.

If in addition we define the `call` operation so that when applied to a procedure, it performs an ordinary procedure call, then any lambda-expression

```
(lambda ( var ...)  body)
```

is indistinguishable from the instance construction (assuming alpha-conversion to avoid name conflicts)

```
(make-instance
 (lambda (selector)
   (cond ((eq? selector 'call)
          (lambda (self  var ...)  body))
         (else not-handled)))).
```

This means that if we take instance construction to be primitive in our language, then `lambda` need not be.

Allowing the creation of instances that handle procedure call is equivalent to allowing the creation of procedures that handle generic operations. Procedures that handle operations can be very useful in a language like Scheme in which procedures have high currency. In T, for example, the operation `setter`, when applied to a procedure that accesses a location, returns by convention a procedure that will store into that location:

```
((operate 'setter car) pair new-value)
```

is equivalent to

```
(set-car! pair new-value).
```

`Car` and other built-in access procedures are set up as if defined by

```
(define car
  (make-instance
   (lambda (selector)
     (cond ((eq? selector 'call)
            (lambda (self pair)
              (primitive-car pair)))
           ((eq? selector 'setter)
            (lambda (self) set-car!))
           (else not-handled)))))
```

T has a generalized `set!` special form, analogous to Common Lisp's `setf`, that provides the more familiar syntax

```
(set! (car pair) obj).
```

To define a new procedure that works with `set!`, it is sufficient to create an instance that handles the `call` and `setter` operations appropriately.

The T system itself makes heavy use of operations on procedures. The system printer and the trace facility provide two examples: Often, a procedure has a print method that displays values of closed-over variables so that the procedure is easily identifiable during debugging. The `trace` utility creates a "traced" version of a given procedure; the original procedure is recovered by invoking an operation on the traced procedure.

4 Anonymous operations

In our development so far, as in Smalltalk[6] and CommonObjects[16], symbols are used to represent abstract operations. This practice can lead to name clashes. If two modules use the same symbol to represent two different abstract operations, then one module may not work in the presence of the other. For example, the two modules may require different default behaviors for two different abstract operations for which they have coincidentally chosen the same name. Or, if module A defines objects that include as components objects created by module B (i.e. "subclasses" one of B's classes), then A might inadvertently use the name of an operation intended to be internal to B. When a method in B performs this operation on instances created by A, the method in A will be seen instead of the intended method in B.

The problem can be eliminated if operations are represented by unique tokens instead of symbols. New tokens must be obtained by calling a system procedure that generates them. The creator of the token can use lexical scoping to control what other code in the system may perform the corresponding abstract operation. (In contrast to Common Lisp, Scheme uses lexical scoping to control visibility of names between modules.) We call this feature *anonymous operations* because, although the token representing an operation is usually the value of some variable, the name of the variable is immaterial.

Here is the cell example reformulated using anonymous operations. New operations are created with `make-operation`, which takes as an argument the operation's default behavior.

```
(define fetch (make-operation error))
(define store! (make-operation error))
(define cell?
  (make-operation (lambda () #f)))

(define make-simple-cell
  (lambda (value)
    (make-instance
     (lambda (selector)
       (cond ((eq? selector fetch)
              (lambda (self) value))
             ((eq? selector store!)
              (lambda (self new-value)
                (set! value new-value)))
             ((eq? selector cell?)
              (lambda (self) #t))
             (else not-handled))))))

(define a-cell (make-simple-cell 8))
(operate fetch a-cell)  →  8
```

The variables `fetch`, `store!`, and `cell?` are evaluated in the course of dispatching, and the selector argument in calls to `operate` is an operation, not a symbol.

How are we to represent operations? The only property we need is that they be distinct from each other, as determined by `eq?`. We could use symbols generated by `gensym`, or pairs, but a better choice is to represent operations as procedures. This way we can arrange that when an operation is called, it performs itself on its first argument.[1]

[1]The definition of `make-operation` given here requires every evaluation of the inner `lambda`-expression to result in a distinct closure. However, some Scheme compilers may try to "optimize" the storage allocation performed by `make-operation` by reusing the same procedure object every time. This is not a serious difficulty, however, since it is always possible to circumvent such optimizations.

```
(define (make-operation default-behavior)
  (letrec ((this-op
            (lambda (the-object . args)
              (apply operate this-op the-object args))))
    (set-default-behavior! this-op default-behavior)
    this-op))
```

We can now convert from the "message passing" style interface that we have been using to a generic procedure style interface. That is, rather than writing

```
(operate fetch a-cell)
```

we write

```
(fetch a-cell).
```

The advantages of the generic procedure style are described in [7].

If we are allowed to define `call` methods on instances (as described in section 3.2), then we can eliminate the global table maintained by `set-default-behavior!` and access an operation's default method via an operation applied to the operation itself.

```
(define (make-operation a-default-behavior)
  (letrec ((this-op
            (make-instance
             (lambda (selector)
               (cond ((eq? selector call)
                      (lambda (self the-object . args)
                        (apply operate this-op the-object args)))
                     ((eq? selector default-behavior)
                      a-default-behavior))))))
    this-op))

(define default-behavior (make-operation error))
```

5 Efficient implementation

We discuss two efficiency problems: dispatch time and method invocation time. We revise our formulation to cope with the first, explain how a Scheme compiler can help with the second, and then consider the first again, but with respect to composite objects.

5.1 Reducing dispatch time

Dispatch time is the time spent mapping from operation to method. If there are many operations in a component, or many components in a composite, then performing operations will be much slower than procedure call.

Dispatch time in the examples presented is proportional to the number of operations handled. Even if each component performs its dispatch in constant time (say using a hash table) dispatch time would still be proportional to the total number of components from which the composite object inherits. Two problems with performing hashes in components arise.

- If the number of operations per component tends to be small, hash lookup may be slower than linear search. Hashing may be practical in composite objects if we can combine the dispatches of all the components. A way to do this is described below.
- A naive approach would result in a distinct hash table per instance. This is far too costly in space. Having like instances use the same hash table is problematic for two reasons.
  - A shared table cannot map from operations to methods because the methods vary from instance to instance. This is because we have chosen to represent methods as closures that include the environment specific to the instance.
  - In the example above the anonymous operations were bound to global variables; this need not be so. If the operations are locally bound, the keys for the table may change from instance to instance.

To fix these problems, we separate operation dispatch from method invocation: `make-instance` now takes two arguments, a dispatcher and a method invoker. We then have:

(instance? (make-instance d i)) $\rightarrow$ true
(instance-dispatcher (make-instance d i)) $\rightarrow$ d
(instance-invoker (make-instance d i)) $\rightarrow$ i
(instance? y) $\rightarrow$ false, if y was not a value
returned by make-instance

A dispatcher maps an operation to a *method index.* The invoker takes a method index and all the arguments for an operation and performs the corresponding method. The following example illustrates this:

```
(define make-simple-cell
  (let ((dispatcher
         (lambda (op)
           (cond ((eq? op fetch) 0)
                 ((eq? op store!) 1)
                 ...
                 (else not-handled)))))
    (lambda (value)
      (make-instance
        dispatcher
        (lambda (index self . args)
          (apply (case index
                   ((0) (lambda (self) value))
                   ((1) (lambda (self new-value)
                          (set! value new-value)))
                   ...)
                 self args))))))
```

Notice that instances returned by make-simple-cell use the same dispatcher. We can arrange for the keys to the dispatch table, here implemented by cond, not to change by having the dispatcher make copies of the relevant global variables.

We are free to implement the mapping performed by the dispatcher, and the mapping performed by the invoker, as we see fit. The choices of cond and case, respectively, are expository.

The code for performing operations must be changed so that it calls both the dispatcher and the invoker:

```
(define (operate-as component selector composite . args)
  (let ((token (get-token component selector)))
    (if (eq? token not-handled)
        (apply (default-behavior selector) composite args)
        (apply invoke-method composite token args))))

(define (get-token component selector)
  (if (instance? component)
      ((instance-dispatcher component)
       selector)
      (get-non-instance-token component selector)))

(define (invoke-method component token composite . args)
  (apply (if (instance? component)
             (instance-invoker component)
             (non-instance-invoker component))
         token composite args))
```

5.2 Fast method invocation

To reduce the amount of time spent in the storage allocator and garbage collector, we would like to avoid allocating a new closure every time the invoker applies a method. Because the method is only called from the invoker, the compiler can customize the calling sequence from the invoker to each method. The environment for each method is a subset of the environment for the invoker, so each method can use the invoker's environment. Thus the code for the invoker reduces to no more than a jump through a branch table to the appropriate method.[2] The arguments are simply "passed through." If we assume the general calling sequence puts a pointer to the callee's environment in a register, then the code for the methods can access their local state as indirect loads off that register.

[2] In practice, having the compiler recognize that the invoker has exactly the right form might be more trouble than having the object system introduce a new primitive the compiler can recognize. We leave this to the implementors.

```
          ; assume first argument is in A1
          load table(pc)[a1],a1
          jump (a1)
table:    address of code for method 1
          address of code for method 2
          ...
method1:  ...
method2:  ...
          ...
```

Happily, this style of transformation has been part of the Scheme culture since near its inception[18, 19, 20]. More recently, the T compiler[8, 9] performs a similar set of transformations in compiling its variant of `make-instance`. For a Scheme compiler already performing environment and closure analysis, only small changes are needed to implement invokers efficiently.

5.3 Efficiency of composite objects; dispatch time revisited

The separation of an instance into an invoker and dispatcher makes it possible for a composite object to construct a single dispatcher that combines the component dispatchers. Invokers corresponding to the component dispatcher must also be combined into a single invoker. A constructor for a composite instance with two components in addition to its own behavior might look roughly as follows:

```
(define constructor
  (let ((dispatcher
         (combine-dispatchers dispatcher1
                              dispatcher2
                              simple-dispatcher)))
    (lambda (a b c)
      (make-instance
        dispatcher
        (combine-invokers invoker1
                          invoker2
                          simple-invoker)))))
```

The method index returned by the dispatcher in the last example is an integer. We call such a dispatcher a *simple dispatcher*. Its corresponding invoker is a *simple invoker*.

A dispatcher created by `combine-dispatchers` returns a method index that is a pair *invoker number, integer index*. An invoker returned by `combine-invokers` performs a two stage dispatch based on such a method index. The invoker number selects a simple invoker, and the integer index is passed to that invoker to perform the method. The system can be implemented such that there are only these two kinds of invokers and dispatchers by having `combine-dispatchers` (`combine-invokers`) always create structures containing only simple dispatchers (invokers).

The implementation of `combine-dispatchers` and `combine-invokers` requires that dispatchers and invokers be able to divulge some information about themselves. This is easy if dispatchers and invokers are themselves instances!

5.4 Caching

Many object system implementations perform some variety of caching to minimize the amount of time in operation dispatch. The introduction of dispatchers makes a variety of caching strategies easy in our framework too. For the sake of caching, the dispatcher plays the same role as the class plays in other object systems[3]. For caching to work correctly, the dispatchers must be pure functions. Unfortunately, the dispatcher given for `make-simple-cell` depended on the values of global variables which could change at runtime. Re-coding the dispatcher to make private copies of the relevant global variables fixes the problem.

6 An example

The example program given here illustrates multiple inheritance, sending to self, and sending to a component.

To improve readability and hide the details of the object system implementation, some syntactic sugar is introduced for expressions that create instances and dispatchers. An expression of the form

`(dispatcher (`$disp_1$ $disp_2$ `...)`
    op_1 op_2 `...)`

yields a dispatcher for instances that handle the operations op_i directly and have components with dispatchers $disp_1$ $disp_2$ .... An expression of the form

`(instance` $disp$
    `(`$component_1$ $component_2$ `...)`
  `((`op_1 $formal_{11}$ $formal_{12}$ `...)` $body_1$`)`
  `((`op_2 $formal_{21}$ $formal_{22}$ `...)` $body_2$`)`
  `...)`

yields an instance with components $component_1$, $component_2$, ..., and methods

`(lambda (`$formal_{1i}$ `...)` $body_i$`)`

for the corresponding operations. The operations, and their order, in the instance and its components must match those of the given dispatcher.

The example defines a kind of cell that maintains a history of all the values that have been stored into it. These cells are implemented as objects that have two components: a "filtered cell" (section 2.4), and a sequence. The cell's history may be accessed using sequence operations.

```
; ------------------ Sequences

(define seq-ref (make-operation #f))
(define seq-set! (make-operation #f))
(define seq-length (make-operation #f))

(define sequence-dispatcher
  (dispatcher () seq-ref seq-set! seq-length print))

(define (make-simple-sequence size)
  (let ((v (make-vector size)))
    (instance sequence-dispatcher
        ()
      ((seq-ref self n)
       (vector-ref v n))
      ((seq-set! self n val)
       (vector-set! v n val))
      ((seq-length self)
       (vector-length v))
      ((print self port)
       (format port "#<Sequence ~s>" (seq-length self))))))
```

A sequence is an object that behaves just like a vector, but is manipulated using generic operations.

```
; ------------------ Filtered cells

(define fetch (make-operation #f))
(define store! (make-operation #f))

(define cell-dispatcher
  (dispatcher () fetch store! print))

(define (make-filtered-cell value filter)
  (instance cell-dispatcher
      ()
    ((fetch self) value)
    ((store! self new-value)
     (if (filter new-value)
         (set! value new-value)
         (discard self new-value)))
    ((print self port)
     (format port "#<Cell ~s>" value))))
```

```
; ------------------ Cells with history

(define position (make-operation #f))
(define discarded-value (make-operation #f))

(define discard
  (make-operation
   (lambda (cell value)
     (format t "Discarding ~s~%" value))))

(define cell-with-history-dispatcher
  (dispatcher (cell-dispatcher
               sequence-dispatcher)
    position
    store!
    discard
    discarded-value
    print))
```

```
; ------------------ Cells with history, continued

(define (make-cell-with-history initial-value filter size)
  (let ((cell (make-filtered-cell initial-value
                                  filter))
        (seq (make-simple-sequence size))
        (most-recent-discard #f)
        (pos 0))
    (let ((self
           (instance cell-with-history-dispatcher
               (cell seq)
             ((position self) pos)
             ((store! self new-value)
              (operate-as cell store! self new-value)
              (seq-set! self pos new-value)
              (set! pos (+ pos 1)))
             ((discard self value)
              (set! most-recent-discard value))
             ((discarded-value self)
              most-recent-discard)
             ((print self port)
              (format port
                      "#<Cell-with-history ~s>"
                      (fetch self))))))
      (store! self initial-value)
      self)))
```

The two methods for the `store!` operation, together with the somewhat frivolous `discard` operation, illustrate sending to self and sending to a component. The `store!` method for cells with history overrides the `store!` method from the cell component. It invokes the `store!` operation on the component using `operate-as`. The component's method in turn calls a `discard` operation on the composite if the stored value fails to pass the filter. If the first `store!` method had simply invoked the `store!` operation on the component, the `discard` would have been applied to the component, not the composite, and the wrong method would have been invoked.

	T object system	Amos no cache	Amos with cache	Tektronix Smalltalk	MacScheme SCOOPS	Symbolics Flavors
1 by 1	37,000	20,300	20,000	76,000	1,700	54,000
10 by 10	2,300	3,300	20,000	76,000	1,000	54,000
20 by 5	1,700	3,300	20,000	76,000	1,000	54,000
proc. call	150,000	—	—	—	19,000	245,000

Table 1: Comparative performance of object systems (operations per second)

7 Role of the programming environment

We have not treated many problems that are usually addressed in object oriented programming languages. For example, when a constructor is redefined, instances that were created before the redefinition are "orphaned" — they may not be consistent with the new definition. We consider this a problem that should be solved by the programming environment. To do this, the programming environment may need to impose constraints on the implementation of the object system. There is a similar problem associated with redefining operations.

In our view, there is nothing special about instances in this regard; the same problem comes up with procedures. What should the programming environment do about orphaned closures when the definition of the procedure that created them is modified? This is an orthogonal problem area that should be solved independently of object-oriented programming.

In our system, methods for a component must be defined monolithically. Again, the task of managing distributed definitions can be the responsibility of the programming environment.

8 Experimental results

The design described here is essentially the same as the T object system, about which little has been published. (T itself is described in [14] and [13].) T's object system, including anonymous operations and single inheritance, has been in production use since 1983, and its object system has proved to be quite valuable. Instances are implemented just as compactly as closures: an instance with n "instance variables" generally consumes only $n+1$ words of storage. Recently a form of multiple inheritance (`join`) has been added to T, but T does not yet have any equivalent of dispatchers. Method lookup is by linear search, and in the case of multiple inheritance, operation dispatch performs one procedure call for each component.

We have measured one aspect of the performance of various dynamic language object systems: number of operations (sends) per second that can be performed on an instance. The data appear in Table 1. Our goals were (1) to compare an implementation based on our design to other dynamic language object systems; and (2) to assess the effect of adding dispatchers to T's object system. Of the implementations measured, Tektronix Smalltalk[2] and Symbolics Flavors are considered to be high performance.

We performed three benchmarks. "1 by 1" measures the time to perform an operation on an instance of one component with one operation. "10 by 10" measures the time to perform an operation on an instance of 10 components, each component handling 10 operations; similarly "20 by 5" measures instances of 20 components, each component handling 5 operations. We measured number of procedure calls per second in the various implementations as a reference to which to compare the other measurements.

We measured several implementations, including a prototype of our design, with dispatchers, in an almost portable subset of T. We refer to this prototype as Amos, for "a minimal object system." Amos uses linear search for method lookup, but it does combine components. We also measured a version of Amos which maintained a small method lookup cache in each operation.

For the T object system, the compiler performs the analysis described above, and the code for operations (essentially `operate` and its sub-functions) is hand-coded assembly language.

SCOOPS is an object system for Scheme based on Flavors. MacScheme SCOOPS is an implementation of SCOOPS quite specific to MacScheme.

Tektronix Smalltalk, and Amos with cache, use method lookup caches. Symbolics Flavors uses a hash table for method lookup.

The T object system, Amos, and Tektronix Smalltalk were measured on comparable 16.7 Mhz Motorola 68020 Unix systems. MacScheme SCOOPS was measured on an Apple Macintosh II (a 15.7 Mhz 68020), Symbolics Flavors was measured on a Symbolics 3650. In overall performance, for the sort of code tested, the 3650 is considered as much as twice as fast as the 16.7 Mhz 68020 systems tested.

The results show that dispatchers are effective, and that our design can be implemented with efficiency comparable to other dynamic language object systems. Because Amos and the T object system use linear search for method lookup, performance falls off with increasing numbers of operations that the instance handles. T object system performance also decreases with increasing number of components because it does not combine the components in any way. Amos does not have this problem.

For instances with small numbers of methods and components, the T object system is comparable to the other systems measured. The addition of dispatchers would make it competitive for instances with larger numbers of operations and components.

Because the benchmarks are so simple, any sort of caching is extremely effective. In practice, we expect the linear search in dispatch will slow the system substantially for instance handling many operations. Replacing the linear search with hashing is practical for Amos but not for T, because Amos combines dispatchers.

9 Comparison with other work

Existing Lisp-based object-oriented languages include Flavors[12], the Common Lisp Object System (CLOS)[5], and CommonObjects.[16] All of these systems are substantially more complicated than our proposal, particularly in their treatment of method combination and multiple inheritance. They extend the underlying language with a new kind of variable, instance variables. Instance variables are not lexically scoped, since their scope (i.e. the program text for methods of the corresponding class) is not textually related to their point of definition (the class definition). Flavors and CLOS support follow our approach of implementing anonymous generic operations as procedures. CommonObjects shares some of our goals, particularly strong support for encapsulation.

Smalltalk[6] makes no distinction between language and environment. It includes many features that defeat encapsulation, as such features tend to make the job of the programming environment easier, a Smalltalk priority.

Our applicative bias is apparent in our design in that there are no initialization methods, and there are no side-effects to invokers (our analogue of classes). Initialization is done applicatively in the contructor procedures rather than by methods that side-affect an instance's state. Side-affecting a class is a common way to achieve method definition in other systems.

Our `call` operation corresponds to the `value` message in Smalltalk, and procedures play the role of Smalltalk blocks by handling the `call` operation. However, Smalltalk's construct for creating blocks is unrelated to the construct for creating other kinds of objects. This appears to be an unnecessary source of language complexity.

Oaklisp[10] also aims for minimality. As in our system, Oaklisp can define `lambda` in terms of object system primitives. In Oaklisp, however, procedures are degenerate generic operations, whereas in our system procedures are degenerate instances. While we retain Scheme's lexical scoping, Oaklisp has two kinds of bindings and scope rules: formal parameters are lexical and are bound in the usual way, while instance variables are defined by the class definition, which cannot, in general, be statically related to the points of use of the instance variables. Also, while we retain Scheme's almost-applicative style, Oaklisp relies on side-effects for defining methods and initializing instance variables.

As formulated here, anonymous operations are essentially capabilities[4]. An operation is an unforgeable token that gives a particular kind of access to some otherwise opaque objects.

10 Conclusions

We have shown how to add object-oriented programming facilities to Scheme with only small additions to the language. In fact, the object system can be explained in portable Scheme with the addition of a single new, but simple, primitive type. The system supports applicative programming style, which to our knowledge is unique among Lisp-based object-oriented programming systems. For efficiency, we depend on known and implemented compilation techniques for Scheme.

The programmer's interface to the object system needs work. With the syntax described in Section 6, the programmer must make sure the dispatcher agrees with the invoker. This annoying problem arises from the great flexibility our system has in creating objects. We can cope with this redundancy in any of several ways: (1) Eliminate it by introducing some kind of `constructor` syntax. This might require the addition of automatically generated initialization methods, and thus side effects, which we prefer to avoid. (2) Check it dynamically at instance creation time—the necessary information is available, but this is expensive, with one comparison per operation handled by the instance. (3) Check it statically, using something like a Milner-style type inference system.

We believe that it is possible to build a programming environment as good as Smalltalk's without sacrificing support for encapsulation.

Acknowledgements

Kent Pitman worked with us on the original formulation of the T object system and was its first user. Richard Kelsey, David Kranz, and Jim Philbin helped make T's object system fast. Will Clinger suggested that the T object system could be made more efficient by introducing an explicit analogue of classes, and encouraged us to work out the details. Will Clinger, Ken Dickey, Bill Havens, Uwe Pleban, Steve Rehfuss, and Steve Vegdahl provided helpful comments on drafts of the paper.

References

[1] Gul A. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, 1986.

[2] Patrick J. Caudill and Allen Wirfs-Brock "A Third Generation Smalltalk-80 Implementation." *SIGPLAN Notices* 21(11), 1986.

[3] Thomas J. Conroy and Eduardo Pelegri-Llopart. "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations." in *Smalltalk-80: Bits of History, Words of Advice.* G. Krasner, ed. Addison-Wesley, 1983.

[4] J. Dennis and E. Van Horn. "Programming semantics for multiprogrammed computations." *Communications of the ACM* 9(3), 1963.

[5] Richard P. Gabriel, et al. "Common lisp object system specification." ANSI X3J13 Document 87-002, 1987.

[6] Adele Goldberg and David Robson. *Smalltalk-80, the language and its implementation.* Addison-Wesley, 1983.

[7] James Kempf, et al. "Experience with CommonLoops." *SIGPLAN Notices* 22(12), 1987.

[8] David Kranz, et al. "Orbit: an optimizing compiler for Scheme." *SIGPLAN Notices* 21(7), 1986

[9] David Andrew Kranz. *Orbit: An Optimizing Compiler for Scheme.* Ph.D. Thesis, Yale University, May 1988.

[10] Kevin J. Lang and Barak A. Pearlmutter. "Oaklisp: an object-oriented Scheme with first class types." *SIGPLAN Notices* 21(11), 1986.

[11] Henry Lieberman. "Using prototypical objects to implement shared behavior in object-oriented systems." *SIGPLAN Notices* 21(11), 1986.

[12] David A. Moon. "Object-oriented programming with flavors." *SIGPLAN Notices* 21(11), 1986.

[13] Jonathan A. Rees and Norman I. Adams. "T, a dialect of Lisp, or lambda: the ultimate software tool." In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, 1982.

[14] Jonathan A. Rees and Norman I. Adams. "The T manual, fourth edition." Yale University Computer Science Department, 1984.

[15] Jonathan A. Rees, et al. "Revised^3 report on the algorithmic language Scheme." *SIGPLAN Notices* 21(12), 1986.

[16] Alan Snyder. "CommonObjects: an overview." *SIGPLAN Notices* 21(10), 1986,

[17] Alan Snyder. "Encapsulation and inheritance in object-oriented programming languages." *SIGPLAN Notices* 21(11), 1986.

[18] Guy L. Steele, Jr. "Lambda: the ultimate imperative." MIT AI Memo 353, 1976.

[19] Guy L. Steele, Jr. “Lambda: the ultimate declarative.” MIT AI Memo 379, 1976.

[20] Guy L. Steele, Jr. “Rabbit: a compiler for Scheme.” Technical Report 454, MIT AI Lab, 1978.

[21] Lynn Andrea Stein. “Delegation is inheritance.” *SIGPLAN Notices* 22(12), 1987.

[22] Gerald Jay Sussman and Guy L. Steele, Jr. “Scheme: an interpreter for extended lambda calculus.” MIT AI Memo 349, 1975.

[23] Daniel Weinreb and David Moon. *Lisp Machine Manual.* MIT AI Lab, 1981.