ABSTRACT
Writing programs in continuation-passing style is a good tool for understanding reified continuations and call/cc. Similarly, writing programs in an object-oriented style is a good tool for understanding object-oriented language features, such as super (or object) method invocation. We present an object-oriented style and a simple example of its use. From this example, we present protocols that make this style more natural. We then enforce the style and protocols by introducing a macro per class. These macros are macro-generating macros that package the syntactic redundancy of the style, approximating a compiler for a conventional object-oriented language. Exercises are scattered throughout to help the energetic reader develop a deeper understanding of the paper’s concepts.

1. INTRODUCTION
We use style to mean an encoding of a language idiom generally in some other language. Furthermore, we also expect that the style make explicit everything that matters. One of the advantages of well-defined programming languages is that built-in idioms are abstracted away. But, this is also one of the disadvantages, especially for those who want to completely understand the language. Also a style has the advantage that since everything is explicit, it is clear exactly how it might be embedded in the encoding language. For example, with Scheme as the encoding language, we claim that any value can be a method, and if that value happens to be a procedure of at least one argument, it can be invoked in object-oriented style.

Our object-oriented style for single inheritance uses a small subset of Scheme [1], with its ability to associate variables to values as our method to define classes and objects. Virtually any system with first-class procedures, lists, and vectors is powerful enough to allow one to write in this style. For updateable fields, we also need updateable vectors.

There are three motivations behind the ideas in this paper. First, we want to remove the confusion by some beginners about what object method invocation is and how it differs from super method invocation. We believe that knowing exactly how, when, and in what lexical scope method tables are built lessens this confusion. Second, we want to clarify the extent to which both kinds of method invocation are just procedure invocations and therefore can lead to long chains of recursive calls (Section 8.4). This seems to have been ignored by most implementors of object-oriented languages, even though two decades ago Cardelli [2], Cook and Palsberg [3], Kamin [6], and Reddy [8] observed how object-oriented programming is related to recursion. Third, we want to demonstrate a style that does not rely on recursion to model classes and objects. This clarifies why the recursion that does arise must be through a dynamically-scoped variable: this (or self).

1.1 What makes CPS a style?
The restrictions of CPS can be described by a grammar: calls to non-primitives must be tail calls, which implies that arguments must be simple. The simple terms are lambda expressions, variables, constants, and combinations of simple terms using primitive calls. The CPS conversion of lambda terms requires adding an extra argument, k, the continuation, to each lambda parameter list, and each lambda body must be CPS’d with respect to k. This, in turn, requires that all calls to non-primitives have an extra argument, an expression whose value is a procedure of one argument. That procedure is a continuation. Instead of returning a value from a procedure, a value is “returned” by invoking a continuation.

But a style is more than a grammar. These grammatical rules do not take into consideration that the continuation-passing definition is advisory. For example, it is possible to decide in one context that member is primitive (i.e., obviously terminates) and in another context that it must, itself, be rewritten in CPS. One must ask oneself, “Will the expression obviously terminate?” If so, then it is okay to treat the expression as if it were a primitive call. But, what if member is passed a circular list and we are relying on it being in CPS for trampolining? Then, it should be treated as a non-primitive call and its definition should be CPS’d. So, CPS is a style with the need for protocols just like the object-oriented style that we discuss here. Hence, the grammar is basically a set of guidelines, where the person who does the manual CPSing of a program gets to use judgment.
1.2 Recursion through this
Before we get started, let’s look at a familiar way of writing and testing an old chestnut: even? and odd? for mutual recursion. We implement them without using letrec or referring to a free variable within a lambda expression.

(define e/o-procs
  (vector
    (lambda (this n)
      (if (zero? n) #t
        ((vector-ref this 1) this (- n 1)))))
    (lambda (this n)
      (if (zero? n) #f
        ((vector-ref this 0) this (- n 1))))))

> e/o-procs
#(#(iseven) #(isodd))

> ((vector-ref e/o-procs 0) e/o-procs 5)
#f

In the definition of e/o-procs, there are neither free variables (only primitives and constants) in the lambda expressions nor any explicit recursion. (When meaningful, we display anonymous procedures with names.) We have two procedures in the vector. The procedure in the zeroth position of the vector pulls out a procedure from the first position of the vector, and the procedure in the first position of the vector pulls out a procedure from the zeroth position. In each such call, it passes the procedure vector as an additional first argument.

A procedure that takes such an additional first argument corresponds to a method, and a vector of such procedures corresponds to an object. What makes object-oriented style more interesting than this simple example is that as each vector is built from another such vector, it is constructed in a reasoned way, but we must wait to see how.

The remainder of the paper is structured as follows. In the next section, we present a variant of the classic point/color point example. In section three, we introduce position environments. In the fourth section, we pick a representation of objects and classes and define nine interface operators. In the fifth section, we present the style informally along with the test program written in our style. In section six, we demonstrate some protocols for giving more intuition about object-oriented style. These include introducing the lexical variable super, and removing direct reliance on positions to access methods and fields. Then in the seventh section, we present an innocent-looking transformation that again aids reading but has significant implications for characterizing the next two transformations. Having set the stage, we demonstrate how macros can allow the user to ignore the interface operators and can impose the protocols of object-oriented style to create a version where the positions are determined at macro-expansion time. In the ninth section, we present new, which not only creates an object but also invokes the appropriate init method. Then, we introduce two macros that remove reliance on quoted symbols. In the final section, we revisit our motivations and make some observations about our approach.

2. THE EXAMPLE
The example assumes some familiarity with object-oriented programming concepts. If this example is not simple to follow, skip this section. Our primary purpose is to characterize the example before we get into the details of our object-oriented style. We use the example of points <p>, and color points <cp>, but we include a method diag, which does a diagonal move, and a method diag&set, which does a diagonal move of n followed by setting the color represented by n. We also have stationary color points <scp>. One of our goals, inspired by the example from Goldberg and Robson [5] (pages 62-65), is to show a simple example that allows us to clarify the distinction between object method calls and super method calls.

Our example is sugared with macros, however, they do not appear until Section 8. A shadow, which is a macro, is what we build before we create its associated class, which is a value as in Smalltalk. A shadow contains the necessary compile-time information to model a class as in C++ or Java. We assume the root shadow <>> and its associated class <o> exist. We use extend-shadow to extend a shadow just as we might extend a class. The list of variables mentioned after the super shadow contains the field variables of the class (shadow). They are followed by a list of method definitions. After each shadow is built, we use create-class to create its associated class. “Shadows” are unusual, however, the method definitions should seem familiar.

(define-syntax <<p>>
  (extend-shadow <<o>> (x y)
    (init
      (method (x y)
        (set! x x)
        (set! y y)
        (init super)))))

  [move
    (method (dx dy)
      (set! x (+ x dx))
      (set! y (+ y dy)))]

  [get-loc
    (method ())
    (list x y))]

  [diag
    (method (a)
      (move this a a))])

  (define <p> (create-class <<p>> <<o>>))

(define-syntax <<cp>>
  (extend-shadow <<p>> (hue)
    (init
      (method (x y hue)
        (set! hue hue)
        (init super x y)))]

  [get-hue
    (method () hue)]

  [diag&set
    (method (a)
      (diag this a)
      (set! hue a)])])

  (define <cp> (create-class <<cp>> <<p>>))
(define-syntax <<scp>>
  (extend-shadow <<cp>> (y)
    ([init
        (method (x y hue))
        (set! y "Stuck: ")
        (init super x y hue))])
  [move
    (method (x y))
    (show y this))
  [diag
    (method a)
    (write hue)
    (diag super a)])
  [show y
    (method () (display y))])
)

(define <scp> (create-class <<scp>> <cp>))

The test program below is simple. We build objects of <p> and <scp> (and <cp>). Next we make method calls using zbr and invoke. We determine that cp is an object of class <p>, but that p is not an object of <scp> (or <cp>). Section 8 and Section 9 explain these features.

(define test
  (lambda (c)
    (let ([p (new <p> 1 2)]
             [cp (new c 18 19 9)])
      (zbr diag # set cp 10)
      (list
        (list
          (invoke <<cp>> get-loc cp)
          (invoke <<cp>> get-hue cp))
        (isa? cp <p>)
        (isa? p c)))))

> (begin (write (test <scp>)) (write (test <cp>)))
9: Stuck: ((18 19) 10) #f #t ((28 29) 10) #t #f

3. POSITION ENVIRONMENTS

In this section, we introduce position environments; a simple operator enumerate-env to make them; an operator append-env to combine them; and an operator trim-env to remove shadowed variables from them. We also introduce installing an environment, an unusual way to determine the value of variables.

We use vectors to represent the data structures of object-oriented style. We eventually want to use variables rather than indices, so we need a representation that associates variables with indices. To do this, we introduce position environments.

A position environment is a function whose domain is variables, represented by symbols, and whose range is positions, represented by nonnegative integers, where no position appears more than once. A position environment is gap-free if it is empty or it associates some variable to position 0 and contains no gaps. We represent a position environment by a list of variable/position pairs, where the positions are in increasing order. We define the value of a variable with such a representation to be the greatest position associated with the variable. A gap-free environment using these pairs is pure if no variable appears more than once. Here is a simple way to make a gap-free environment.

(define enumerate-env
  (lambda (vars)
    (let loop ([vars vars] [i 0])
      (cond
        [(null? vars) ’()]
        [else
          (cons ‘((car vars) i)
                 (loop (cdr vars) (+ i 1)))]))))

> (define p ’(a b c))
> (define q ’(a d c))
> (define penv (enumerate-env p))
> (define qenv (enumerate-env q))
> (define renv ’([a 3] [d 5] [c 6]))
> penv
([a 0] [b 1] [c 2])
> qenv
([a 0] [d 1] [c 2])

We define append-env of a position and a pure environment. It takes the pure environment onto the position environment, but starts the pure environment at one more than the greatest position of the first argument.

(define append-env
  (lambda (e p)
    (cond
      [(null? e) p]
      [else
        (append e
                 (let ([m (+ (cadar (last-pair e)) 1))]
                   (map (lambda (pr)
                          (cons ‘((car pr), (+ (cadr pr) m))
                                p)))]))])))

> (append-env penv qenv)
([a 0] [b 1] [c 2] [a 3] [d 4] [c 5])
> (append-env renv penv)
([a 3] [d 5] [c 6] [a 7] [b 8] [c 9])

We revise append-env below, so we do worry about this definition’s efficiency.

We can trim the shadowed variables from an environment, since we only care about the greatest positions of the variables.

(define trim-env
  (lambda (e)
    (cond
      [(null? e) ’()]
      [else (cons (car e) (trim-env (cdr e)))])))

> (trim-env (append-env penv qenv))
([b 1] [a 3] [d 4] [c 5])
Installing an environment crosses levels of syntax and semantics. We can imagine taking an environment and dropping it into a let expression. Here is how we install env.

> (let ([a 0] [b 1] [c 2])
   (list a b c))
(0 1 2)

This is equivalent to (eval (let ,env (list a b c))), but we prefer omitting uses of eval in explaining environment installation, since eval has flaws that we wish to avoid.

Similarly, we can install a gap-free environment using let*. We use let*, since we are only interested in the greatest position of each variable. Thus, a's value is 3 (not 0), and c's value is 5 (not 2). If we wish to use let, we must trim the environment before we install it.

> (let* ([a 0] [b 1] [c 2] [a 3] [d 4] [c 5])
   (list a b c d))
(3 1 5 4)

When an environment is gap-free, we may represent it with a simple list of variables. We can redefine append-env over gap-free environments and thus represent an environment as a list of variables. The result of append-env is also gap-free. Its definition is just append.

(define append-env append)
> (enumerate-env (append-env p q))
([a 0] [b 1] [c 2] [a 3] [d 4] [c 5])

Exercise 1: Let \(e_1\) be gap-free and \(e_2\) be pure with both represented by a simple list of variables. Rewrite (trim-env (enumerate-env (append-env \(e_1\) \(e_2\)))) so that, except for some anticipated calls to \(\text{nenv}\), we get a one-pass algorithm.

4. CLASSES AND OBJECTS

A class is represented as a list\(^1\) of three items: a field environment, a method environment, and a method vector. A method is any value. If it is a procedure of at least one argument, then it can be invoked in object-oriented style. An object is like a class but differs in its first item, a field vector the size of the field environment of its associated class. The cdr of a class or object is its method table. Thus an object and its associated class have the same method environment and vector. Since the field environment is gap-free and the method environment is pure, we represent each with a simple list of variables. The environments and vectors are in one-to-one correspondence, allowing us to associate a variable with each vector element.

Next we define the nine interface operators \(\text{fx}, \text{mx}, \text{mp}, \text{mp}'!, \text{fp}, \text{f}'p!, \text{mteq?}, \text{n}, \text{and} \text{menv}\).

4.1 Environment extension operators

We introduce interface operators \(\text{fx}, \text{mx}\), which extends the field environment of a class, and \(\text{mx}\), which extends the method environment of a class.

(define (_fx c e) (append-env (car c) e))
(define (_mx c e) (append-env (cadadr c) e))

4.2 Position-based operators

Each position-based interface operator takes a position as its second argument. To access (or update) a method by position, use \(\text{mp}!\). To access (or update) a field by position, use \(\text{fp}!\). (oc is an object or a class.)

(define (_mp oc p) (vector-ref (cadadr oc) p))
(define (_mp! oc p v) (vector-set! (cadadr oc) p v))
(define (_fp oc p) (vector-ref (car oc) p))
(define (_fp! oc p v) (vector-set! (car oc) p v))

4.3 Method table operator

We define \(\text{mteq?}\), which takes an object or class \(\text{oc}\) and an object or a class \(\text{oc2}\). If they share the same method table, it returns true. Since the cdr of a class or object is its method table, it checks to see if the two arguments have the same method table.

(define (_mteq? oc1 oc2) (eq? (cdr oc1) (cdr oc2)))

We have to be sure that each class has a unique method table. We get that property, because each method vector gets a new \(\text{isa?}\) method, thus requiring a new method vector.

4.4 External operators

We define \(\text{n}\), which given a class, creates a new object, and \(\text{mv}\), which given an object and a variable (symbol), accesses a method.

(define (_n c)
  (cons (make-vector (length (car c))) (cdr c)))

(define (_mv m)
  (_mp oc
    (let loop ([m* (cadadr oc)] [pos 0])
      (if (eqv? (car m*) m)
        pos
        (loop (cdr m*) (+ pos 1))))))

We observe that accessing a method by its associated variable (a symbol) requires computing its position. When we use \(\text{mv}\), we say that we are accessing the method externally. These last two are different from the other seven, since these two can be used anywhere, whereas the first six are restricted to class definitions. As such, we refer to the first seven as internal and the remaining two as external interface operators.

\(^1\)We have opted for lists instead of records. We want to build our object-oriented style out of primitive pieces like lists, vectors, and lambda.
Exercise 2: Using the definition of e/o-procs from Section 1.2, implement the procedures even? and odd?. Then implement factorial using this style. Each of the three procedures should take a single integer. 

We have now presented the nine interface operators. Next we present our object-oriented style and the example in this style. We then enforce some protocols on the style. Once we have the protocols in place, we then observe redundancy in the transformed program. Finally, we remove this redundancy with macro-generating macros.

5. OBJECT-ORIENTED STYLE

In this section we present the essence of an object-oriented style along with the test program and four classes written in this style. Upon completion of this section, the style should be clear and what remains has to do with improving the style’s readability through protocols, method lifting strategies, and some macros, which allow the user to express the style more succinctly.

5.1 The style

Our object-oriented style is written in Scheme and uses the environment operator append-env (enumerate-env and trim-env are used in Section 8, only.), and our choice of representation for objects and classes. We liberally borrow terminology from Smalltalk, C++, and Java.

There are several facets of the style that we postpone until our classes and test programs are presented. Here, however, we give just a few fundamental aspects of the style. Some methods are not expressed as procedures built from lambda expressions, only those that are have class, which must be bound to an object or a class, as their first formal parameter. When this is bound to an object, we can access or set its fields. Every object of a given class uses the same position in its field vector for each field defined by the class. Thus, referencing and updating fields of objects of a given class is through constant positions within the field vector of this.

To invoke a method (It may be invoked in other ways.) in object-oriented style, first obtain the method through a constant position of a method vector and then invoke it on some object or class and perhaps some additional arguments. Following the style means that if the method is from an object, then its first argument must be the object. If the method is from a class, then its first argument is either the class (rarely, but see the code of isa? below) or this. There are no restrictions on method bodies.

5.2 The example in our style

Below are four classes written in our object-oriented style. We refer to the class that we are currently defining as the host class. Each method of a host class is determined in one of two ways. Either it is the result of evaluating a Scheme expression (e.g., a lambda expression) or it is a method contributed from a single existing class, called its super class.

Because we know that there is a one-to-one correspondence between the variables in the method environment of a class and the positions in the methods vector of the class, we can think of the methods in the vector as if they had a name. For example, in the root class <o> below, we can associate isa? and init with the first two positions of its method vector.

(define <o> 
(list 
'( () 
'(isa? init) 
(vector 
(lambda (this c) 
(_eq? this c)) 
(lambda (this . args) 
(void)))))

> <o> 
(() 
(isa? init) 
(#(<isa?> #<init>)))

(define <p> 
(list 
(_fx <o> '(x y)) 
(_mx <o> '(move get-loc diag)) 
(vector 
(lambda (this c) 
(or (_eq? this c) 
(((mp <o> 0) <o> c))) 
(lambda (this x y) 
(_fp! this 0 x) 
(_fp! this 1 y)) 
(((mp <o> 1) this))) 
(lambda (this dx dy) 
(_fp! this 0 (+ (_fp this 0) dx)) 
(_fp! this 1 (+ (_fp this 1) dy)))))

> <p> 
((x y) 
(isa? init move get-loc diag) 
(#(<isa?> #<init> #<move> #<get-loc> #<diag>))

In describing class <p> above, we say that the methods isa? and init are overriding since they replace already existing methods in (its super) class <o> associated with the same variable. We say that the methods move, get-loc, and diag are fresh since they do not override existing methods.

The explicit list of variables in a call to _fx are the fresh field variables. They cannot contain duplicates and their order matters. The explicit list of variables in a call to _mx are the fresh method variables. They cannot contain duplicates and their order matters. As the method vector is filled in, each method must fit into the right position. The overriding and contributed methods must be in the same position as in their super class. The fresh methods must be placed after the overriding and contributed methods, and they must follow the order used in the call to _mx. There are no other constraints on how the method vector is built.

Consider this interactive session.

> (define p (_n <p>)) 
> ((_mp p 1) p 12 13)
> p
(#(12 13)
  (isa? init move get-loc diag)
  (#(<isa?> #<init> #<move> #<get-loc> #<diag>))
> ((mv p 'move) p 14 15)
> (define map-nullary-method
    (lambda (m)
      (map (lambda (n) ((mv m n) m)))))
> (map-nullary-method p '(get-loc))
((26 28))
> ((mp p 0) p <cp>)
#t

The object p differs from the class <p> where the field vector and environment are stored. The field environment is (x y), whereas the field vector is #(12 13). We move p, which adds 14 to its x coordinate and 15 to its y coordinate, yielding its new location: (26 28). We finish with a test of isa?. Next, we have the definition of the color point class.

(deff <cp> ((fx <cp> '(hue))
  (mx <cp> '(get-hue diag&set))
  (vector
    (lambda (this c)
      (or (_step? this c)
          ((mp <p> 0) <p> c)))
    (lambda (this x y hue)
      (_fp! this 2 hue)
      ((mp <p> 1) this x y)
      (mp <p> 2)
      (mp <p> 3)
      (mp <p> 4)
    )
    (lambda (this a)
      ((mp this 4) this a)
      (fp this 2 a)))))

> <cp>
((x y hue)
  (isa? init move get-loc diag get-hue diag&set)
  (#(<isa?> #<init> ...)))

We continue the interactive session.

> (define cp (_n <cp>))
> ((mp cp 1) cp 16 17 ?)
> cp
((16 17 ?)
  (isa? init move get-loc diag get-hue diag&set)
  (#(<isa?> #<init> ...)))

> ((mv cp 'diag&set) cp 8)
> (map-nullary-method cp '(get-loc get-hue))
((24 25) 8)
> ((mp cp 0) cp <p>)
#t
> ((mp p 0) p <cp>)
#f

First, the color point cp has been painted 7. Then, its color has been changed to 8 and 8 has been added to both its x and y coordinates. We then follow that with two isa? tests.

In <scp>, we have no methods contributed from any super class, since <scp> is the first one defined. In class <p> no method of its super class, <cp>, has been contributed. In class <cp> the second, third, and fourth positions of <p> contribute a method. Thus we see that the same (eq?) method can be stored in more than one class. Finally, we define the stationary color point class <scp>.

(define <scp>
  (list
    (_fx <scp> '(y))
    (_mx <scp> '(show-y))
    (vector
      (lambda (this c)
        (or (_step? this c)
            ((mp <cp> 0) <cp> c)))
      (lambda (this x y hue)
        (_fp! this 3 "Stuck: ")
        ((mp <cp> 1) this x y hue))
      (lambda (this x y)
        (_fp this 7 this))
      (mp <cp> 3)
      (lambda (this a)
        (write (_fp this 2))
        ((mp <cp> 4) this a))
      (mp <cp> 5)
      (mp <cp> 6)
      (lambda (this)
        (display (_fp this 3))))))

> <scp>
((x y hue y)
  (isa? init ... get-hue diag&set show-y)
  (#(<isa?> #<init> ...)))

Exercise 3: Consider the results of this test.

(define test
  (lambda (c)
    (let ([p (_n <p>)]
          [cp (_n c)])
      ((mp cp 1) cp 18 19 9)
      ((mv cp 'diag&set) cp 10)
      (list
        (map-nullary-method cp '(get-loc get-hue))
        ((mp cp 0) cp <p>)
        ((mp p 0) p <c>)
        )))

> (test <scp>)
9: Stuck: (((18 19) 10) #t #f)

Why is a 9 displayed first and why is location (18 19) displayed instead of (28 29)?

The internal interface operators are used primarily in constructing the field and method environments, accessing (or updating) by position, and comparing two method tables.
When a class is passed to an internal operator, it should be the host's super class. We have no way of enforcing this restriction in this style, and furthermore, there are object-oriented languages where it is also not enforced.

A chain is a nonempty sequence of classes, linked by super classes, whose first element is a host class and whose last element is &lt;op>. These classes form four chains. The shortest chain is of length one and starts at &lt;op>, whereas the longest chain is of length four and starts at &lt;scp>.

Although &lt;op> and &lt;env> occur in the test only, it is still okay to use them within a method to create an object of any class or to externally access a method. One of the reasons for the confusion of the costs incurred by writing in object-oriented programming languages is that they do not syntactically distinguish between the two different ways of accessing a method: internally and externally. What is better, however, is to know that there is a search, possibly with a method-call cache or a hash table, whenever a method is accessed externally, as in map-mullary-method. For example, a different chain might have the method move associated with a position other than the second position. Clearly we cannot always rely on positions, but we can rely on isa?'s position being 0 and init?'s position being 1.

We shouldn't rely on positions (except for init.'s and isa?'s) in the test program above, because its code is not inside the class chain. Outside the class chain, we use a variable (symbol) to find its associated method.

We discover another bit of object-oriented style with the definitions of &lt;op>, &lt;cp>, and &lt;scp>. Except in class &lt;op>, the isa? method is overridden in each class definition; the first disjunct, (eq? this c), is identical in each isa? method; and the second disjunct in isa? differs only in the reference to the host's super class. The isa? algorithm is a little tricky. The first call to isa? is different from all the others. In the first call to isa?, the value bound to this is an object. In the remaining calls, it is a class. So, the first call compares the method tables of an object and a class, but the other calls compare the method tables of two classes. In a sense, the remaining calls are just doing a kind of member search in a chain that starts at its host's super class.

What remains is to make the programs more eye-pleasing by introducing a small number of protocols and some macro-generating macros to manage redundancy. When we are done, we will have lost none of the power of Scheme and everything will be properly lexically scoped, so that although our tests rely on define (or define-syntax) for globals, they could as easily have relied on let (or let-syntax) for locals.

6. IMPOSING PROTOCOLS

There are three protocols that our object-oriented style uses. By enforcing these protocols, we approximate a conventional object-oriented language. First, we make the method bodies more readable by consistently using super to refer to the super class; second, we make the method bodies less conscious about method positions; and third, we do the same for field positions.

Henceforth, we focus exclusively on the code of the class definitions. Because of the confusion that is likely to arise by seeing lots of versions of all the classes, we confine our remaining transformations to class &lt;scp>. We suggest revising &lt;op>, &lt;cp>, and &lt;scp> to verify your understanding.

Exercise 4: Each of the operators in the interface (such as &lt;fp> and &lt;mp>) takes an object or class as its first argument. Methods have the property that they take an object or class as their first argument. Therefore, we can treat these operators as methods. Then it should also be possible to override these methods. Think about these observations and implement a system where the interface operators are themselves methods. Also, consider representing the field environment as a vector of variables. (This exercise has been inspired by meta-object protocols [7].)

Exercise 5: Use a single vector to represent classes by placing the method vector at index 0, the method environment at index 1, and each component of the field environment starting at index 2. Replace the use of list by build-class using its definition below,

\[
\text{(define build-class)}
\quad \text{(lambda (f-env m-env m-vec)}
\quad \quad \text{(list->vector)}
\quad \quad \quad \text{(cons m-vec (cons m-env f-env)))}
\]

and re-implement the nine interface operators to take advantage of this representation, which shrinks the cost of classes and objects with zero fields.

6.1 Introducing super

Since &lt;scp>'s super class &lt;cp> appears many times in this code, we may abstract it out.

\[
\text{(define &lt;scp>)}
\quad \text{(let ([super &lt;cp>])}
\quad \quad \text{(list)}
\quad \quad \quad \text{(_fx super 'y))}
\quad \quad \quad \text{(_mx super 'show-y))}
\quad \quad \text{(vector)}
\quad \quad \quad \text{(lambda (this c)}
\quad \quad \quad \quad \text{(or (_steq? this c)}
\quad \quad \quad \quad \quad \text{((_mp super 0) super c))}
\quad \quad \quad \quad \text{((_fp this 3 "hue")}
\quad \quad \quad \quad \quad \text{(_mp super 1) this x' y' hue")}
\quad \quad \quad \quad \text{((_mp this ?) this))}
\quad \quad \quad \quad \text{(_mp super 3)}
\quad \quad \quad \quad \text{(lambda (this a)
\quad \quad \quad \quad \quad \text{(_write (_fp this 2))}}
\quad \quad \quad \quad \quad \quad \text{((_mp super 4) this a))}
\quad \quad \quad \quad \quad \text{(_mp super 5)}
\quad \quad \quad \quad \quad \text{(_mp super 6)}
\quad \quad \quad \quad \quad \text{(lambda (this)}
\quad \quad \quad \quad \quad \quad \text{(_display (_fp this 3)))})
\]
This makes the definition of isa? more consistent: if this and the class are _mteq?, then isa? is true, otherwise try the isa? method of its host’s super class passing super along instead of this.

The decision to bind the lexical super to its host’s super class makes it clear that super is always static. One of the confusing aspects of object-oriented programming, especially for neophytes, is whether one searches the chain of the class of this to find its super. This protocol clarifies that there is not even a search for super. super as a variable, however, must be used with caution. For example, binding it to another variable undermines some of the improvements in Section 8.2, and doing so leads inevitably to unexpected errors.

### 6.2 Position variables for methods

Next, we install the enumerated newly-created method environment with a let expression.

```scheme
(define <scp>
  (let* ([isa? 0]
          [init 1]
          [move 2]
          [get-loc 3]
          [diag 4]
          [get-hue 5]
          [diag&set 6]
          [show-y?] 7)
    (let ([super <cp>])
      (list
       (_fx super '(y))
       (_mx super '(show-y))
       (vector
        (lambda (this c)
         (or (_mteq? this c)
             ((_mp super isa?) super c)))
        (lambda (this x­ y­ hue­)
         (_fp! this 3 "Stuck: ")
         ((_mp super init) this x­ y­ hue­)
         (lambda (this x­ y­)
          ((_mp this show-y) this)
          ((_mp super get-loc)
           (lambda (this a)
            (write (_fp this 2))
            ((_mp super diag) this a)
            ((_mp super get-hue)
             ((_mp super diag&set)
              (lambda (this)
               (display (_fp this 3)))))))))

     (define <scp>
       (let* ([x 0] [y 1] [hue 2] [y 3])
         (let ([isa? 0] --- [show-y?] 7)
           (let ([super <cp>])
             (list
              (_fx super '(y))
              (_mx super '(show-y))
              (vector
               (lambda (this c)
                (or (_mteq? this c)
                    ((_mp super isa?) super c)))
               (lambda (this x­ y­ hue­)
                (_fp! this y ": Stuck: ")
                ((_mp super init) this x­ y­ hue­)
                (lambda (this x­ y­)
                 ((_mp this show-y) this)
                 ((_mp super get-loc)
                  (lambda (this a)
                   (write (_fp this 2))
                   ((_mp super diag) this a)
                   ((_mp super get-hue)
                    ((_mp super diag&set)
                     (lambda (this)
                      (display (_fp this 3))))))))))

   where “---” indicates that there is some code that needs to be filled in. It should be obvious, but often just looking at the previous definition suffices.

One let* expression, which installs the field and method environments, would suffice. Even super could be included in the one let* expression. This explains why within a lexical scope no method variable (as a symbol) should be the same as a field variable (as a symbol) and furthermore, the symbol super should not be used as a method or field variable.

**Exercise 6**: Consider this definition of <scp>.

```scheme
(define <scp>
  (let* ([x 0] [y 1] [hue 2] [y 3])
    (let ([isa? 0] --- [show-y?] 7)
      (let ([super <cp>])
        (list
         (_fx super '(y))
         (_mx super '(show-y))
         (vector
          '([isa? , (lambda (this c) ----)]
             [init , (lambda (this x­ y­ hue­) ----)]
             [move , (lambda (this x­ y­) ----)]
             [diag , (lambda (this a) ----)]
             [show-y , (lambda (this) ----)])))))))

Implement _mxct_, which takes a class and a list of overriding and fresh method pairs and returns the appropriate method vector at class-construction time.

We have reached the end of the discussion of object-oriented style. The style, especially with _mxct_, is good enough for most purposes, but we can improve its readability significantly.

### 6.3 Position variables for fields

We treat the field environment in a similar fashion. The field environment supports the protected variables. Since y in <scp> is the y at position 3 (not the y at position 1), we install the enumerated newly-created field environment with let*. This also says that any occurrences of x and hue are lexically visible here. The only way that one should be permitted to access (_fp this 1), however, is by invoking a method defined in class <scp>, which is rational behavior, or literally using the 1, which is irrational behavior. Here is the revised <scp>.

```scheme

7. LIFTING METHODS

We present three ways to lift methods. First, we lift the methods in a rather naive way; second, we use a scoping trick to obviate the need for worrying about contributed methods; and third, we bind each super method lexically.

7.1 Naive lifting

We can once again improve readability of this code. Since the methods to be placed in the method vector are not explicitly recursive, we can lift them out with a let expression. Furthermore, we can use the variable associated with the method as the binding variable within the let. The order of the bindings in the let expression does not matter, but the order of the methods in the vector does.

(define <scp>
  (let* ([x 0] [y 1] [hue 2] [y 3])
    (let ([isa? 0] ---- [show-y 7])
      (let ([super <cp>]
            (let ([isa? (lambda (this c) ----)]
                  [init (lambda (this x y hue) ----)]
                  [move (lambda (this x y) ----)]
                  [get-loc (let super get-loc)]
                  [diag (lambda (this a) ----)]
                  [get-hue (let super get-hue)]
                  [diag&set (let super diag&set)]
                  [show-y (lambda (this) ----)])
                  [list (_fx super '(y))
                      (_mx super '(!(show-y)))
                      (vector isa? --- show-y))])))

7.2 Triply-nested let

Next, we show a scoping trick to create a method vector. This trick allows us to use scoping alone to figure out the contributed and overriding methods.

(define <scp>
  (let* ([x 0] [y 1] [hue 2] [y 3])
    (let ([isa? 0] ---- [show-y 7])
      (let ([super <cp>]
            (let (g0 (lambda (this c)
                        (or (_mteq? this c)
                            (h0 super c)))
                  [g1 (lambda (this x y hue)
                         (_fp! this y "Stuck: ")
                         (h1 this x y hue))]
                  [g2 (lambda (this x y)
                         (_mp this show-y this))]
                  [g3 (lambda (this a)
                        (write (_fp this hue))
                        (h3 this a))]
                  [g4 (lambda (this)
                        (display (_fp this y)))
                        (let ([isa? h0] ---- [diag&set h6])
                        (let ([isa? g0]
                                [init g1]
                                [move g2]
                                [diag g3]
                                [show-y g4])
                                [list (_fx super '(y))
                                    (_mx super '(!(show-y)))
                                    (vector isa? --- show-y))))))))))

We know that we can scramble the order of the bindings in the let expression without affecting its semantics. This version uses as many temporaries g0-g4 as there are overriding and fresh methods. The code still closes the method expressions in the position environments. Then it binds all the super methods, however, any that are being overridden get rebound in the last let of the triply-nested let. If a compiler could determine that _mp is effect free, no code should be generated for the binding of the super init, isa?, move, and diag methods, since they are being lexically shadowed. Thus, the same variable, for example init, changes from being bound to a position, to being bound to a super method, then to being bound to an overriding method.

7.3 Quadruply-nested let

We can include an additional outer let layer using the following strategy. Since we know the super class, we bind each super method to a super temporary, h0-h6. Then we replace each super method access (_mp super ----) within the body of this new let by its super temporary.

(define <scp>
  (let ([super <cp>]
        (let* ([x 0] [y 1] [hue 2] [y 3])
          (let ([isa? 0] ---- [show-y 7])
            (let ([h0 (_mp super isa?)]
                  [h1 (lambda (this x y hue)
                    (_fp! this y ": Stuck: ")
                    (h1 this x y hue))]
                  [h2 (lambda (this x y)
                    (_mp this show-y this))]
                  [h3 (lambda (this a)
                    (write (_fp this hue))
                    (h3 this a))]
                  [h4 (lambda (this)
                    (display (_fp this y))))
                  (let ([isa? h0] ---- [diag&set h6])
                    (let ([isa? g0]
                           [init g1]
                           [move g2]
                           [diag g3]
                           [show-y g4])
                         [list (_fx super '(y))
                              (_mx super '(!(show-y)))
                              (vector isa? --- show-y))))))))

We have chosen this unorthodox approach because we want to point out the extent to which scope can be used to solve problems. We could have used the naively lifted version of <scp> instead of build-shadow below, but then we would have needed an additional position environment operator. Also, with the binding of the super methods, it becomes even more obvious that super method calls are static.
Exercise 7: Our characterization of field variables yields protected variables. We can make some of them private instead of protected. Assume that unique is a variable that will never be used as a field or method variable. When we build, but not install, the host's field environment, we replace each private variable by unique. Thus, the only way that the field can be accessed is through a method in its host. Using unique, determine the field environment of classes <cp> and <scp> so that hue of <cp> is private. Since get-hue gets contributed to any class that has <cp> in its super class chain, we can revise diag of <scp> to use it instead of hue. 

Exercise 8: Our characterization of method variables yields protected variables. We can make some of them private instead of protected. We associate with each method variable a boolean indicating whether (#t) or not (#f) a variable is private. Then accessing using .mv is permissible if the method variable is #t. But, _mp is where the problem gets interesting. If we limit the use of super to methods that are being overridden with the same variable, then even if we make the variable private in the super class, we should still be able to access it. But, we should not be able to access a private method variable in the super class with an object. When we install the method environment in the host, we use different unique variables for each method that has been designated private in the super class. If we installed with let, we could use one unique variable as in the preceding exercise. Make move private in class <cp> by revising the class chain. 

8. SIMPLIFYING THE SYNTAX

We now recognize four improvements that we can make to the readability of the code. First, we observe that the isa? method in each class other than <cp> looks the same. Second, we do not need to worry about which methods are being contributed. If they come from the super class and are not being overridden, then they are being contributed. Third, we make field reference appear like Scheme variable lookup and field update appear like set! Fourth, we limit internal method access to calls that appear syntactically like procedure calls. The second version of extender combined with build-shadow, presented below, is our eventual goal. To reach this goal, we need the notion of a shadow, which we now describe.

8.1 Defining a shadow per class

A shadow is a macro that manages the static information of a class. That includes the field and method environments. The behavior of a class is the method vector, which we can mostly ignore. We present a simple way with one complex macro to capture the protocols of object-oriented style.

Each shadow can be used in three ways. First, it might be used as a super shadow in the definition of another shadow, just as a class might be used as a super class. This leads to the higher-order macro extend-shadow that takes the super shadow as a parameter. Second, it might be used for accessing a method externally at macro-expansion time. This leads to the higher-order macro invoke, which we describe in Section 9. Third, it might be used to build a class with a super class. The super class is needed, of course, for accessing the contributed methods. This leads to the higher-order macro create-class that takes the host shadow as a parameter. So the shadow macro contains a dispatch to select one of these three tasks. Before we analyze them, let's look at how they would be used to define <scp>, assuming that shadow <<<>> and its associated class <cp> exist.

```
(define-syntax <<<>> (extend-shadow <<<>> ---))
(define <p> (create-class <<<>> <p>))
```

```
(define-syntax <<<>> (extend-shadow <<<>> ---))
(define <cp> (create-class <<<>> <cp>))
```

```
(define-syntax <<<>> (extend-shadow <<<>> (y))
  (init
    (lambda (this x` y` hue`)
      (fp! this y "Stuck: ")
    (mp super init) this x` y` hue`))
  [move
    (lambda (this x` y`)
      (mp this show-y) this)]
  [diag
    (lambda (this a)
      (write (_fp this hue))
      (mp super diag this a))]
  [show-y
    (lambda (this)
      (display (:fp this y)))]

(define <scp> (create-class <<<>> <cp>))
```

Exercise 9: Complete <<<>> and <<<>>.

The macro extend-shadow below expands into an invocation of the macro super-shadow with three operands. The first is a (super) shadow. The second operand is a list of fresh (host) field variables that we are making available. This means, for example, that the shadow <<<>> has one more item in its field environment than those of its super shadow, <<<>>.

Last, there are "let" pairs that define the overriding and fresh methods. Here is where we take advantage of the fixed structure of the isa? method by automatically including it in every shadow, except <<<>>. The expansion of super-shadow generated by the expansion of extend-shadow takes three operands. The first of these items "_" is the syntactic context that dictates which macro expansion a variable comes from. By passing it along, we can use it to control variable hygiene. The remaining operands are the fresh field variables and the overriding and fresh method "let" pairs of the host class, with the addition of the isa? method "let" pair.

```
(define-syntax extend-shadow
  (lambda (x)
    (syntax-case x ()
      [(_ sup-shadow (f-var ...)) (m-var e) ...]
      (with-implicit (_ super isa?)
        (\(sup-shadow _ (f-var ...) \("isa?\) (lambda (this c)
          (or (_meta? this c)
               (\(mp super 0) super c)))
            (m-var e) ...)))])))
```

```
8.2 Simplifying fields and method calls
Now we redefine extender in Figure 2 so that field variables look like regular lexical variables and internal method calls look like regular procedure calls. Thus, occurrences of (λp this f) can be replaced with Scheme’s variable f, and occurrences of (λp! this f v) can be replaced with (set! f v). Similarly, occurrences of ((λp o m) o arg ...) can be replaced with (m o arg ...). Also occurrences of (h* this arg ...) where h* is a lexically scoped super method variable for the super method m, can be replaced with (m super arg ...) (See Section 2 for examples.)
(define-syntax with-implicit
  (syntax-rules ()
    [(_ (ctx id ...) body0 body1 ...) (with-syntax ([id (datum->syntax-object #'ctx 'id)] ...) body0 body1 ...)])

(define-syntax extender
  (lambda (x)
    (syntax-case x ()
      [(_ ctx [i n k] ...) (all-f ...) ([f j] ...) ([m i] ...) ([m-var g e] ...)
       (with-syntax ([h (generate-temporaries #'(s ...))])
        (with-implicit (ctx super)
          #'(let ([f j] ...)
            (let ([m i] ...)
              (lambda (super)
                (let ([h (append-env super k)] ...)
                  (let ([g e] ...)
                    (let ([m h] ...)
                      (let ([m-var g] ...)
                        (list 'all-f ...)'(m ...)(vector m ...))))))))))

      )]

(define-syntax assv-macro
  (lambda (x)
    (syntax-case x ()
      [(_ i ([k0 h0] [kl h1] ...))
        (if (eql? (syntax-object->datum '#'k0) (syntax-object->datum '#'i)) '#'h0
          #'(assv-macro i ([kl h1] ...)))]

      )]

(define-syntax build-shadow
  (lambda (x)
    (syntax-case x ()
      [(_ ctx sup-f sup-m (f-var ...) ([m-var g e] ...))
        (let ([sup-f (syntax-object->datum '#sup-f)]
            [sup-m (syntax-object->datum '#sup-m)]
            [f-vars (syntax-object->datum '#f-var ...)]
            [m-vars (syntax-object->datum '#m-var ...)])
          (let ([f (append-env sup-f f-vars)]
              [m (append-env sup-m (fresh-m-vars m-vars sup-m))])
            (with-syntax (['[s k] ...](datum->syntax-object '#ctx (enumerate-env sup-m))]
              [[[m i] ...](datum->syntax-object '#ctx (enumerate-env m)])
              [[[f j] ...](datum->syntax-object '#ctx (trim-env (enumerate-env f)))]
              [[[all-f ...](datum->syntax-object '#ctx f)]
                (lambda (xx)
                  (syntax-case xx ()
                    [(_)
                      #'(extend ctx ([s k] ...) (all-f ...) ([f j] ...) ([m i] ...) ([m-var g e] ...))]
                    [(_ an-m-var oc) #'(mp oc (assv-macro an-m-var (in m i) ...))]
                    [(_ ctx (f-var' (... ...)) ([m-var' e'] ...) (with-syntax (['[g' (f-var' ...) (generate-temporaries '#'m-var' (...) ...)]
                      #'(build-shadow ctx (all-f ...) (m ...)
                        (f-var' (...) ...)
                        ([m-var' g' e'] (...) ...)))]))])])

      )]

(define fresh-m-vars
  (lambda (m-vars sup-m-vars)
    (cond
      [(null? m-vars) '()]
      [(assv (car m-vars) sup-m-vars) (fresh-m-vars (cdr m-vars) sup-m-vars)]
      [else (cons (car m-vars) (fresh-m-vars (cdr m-vars) sup-m-vars))])))

Figure 1: The build-shadow macro using extender
Figure 2: The extender macro for fields and methods
This extender includes the definition of the local macro, method, which makes anonymous methods within an extend-shadow just as lambda makes anonymous procedures. The difference is that with method it assumes implicitly that the first argument is bound to the variable this. Thus, the number of formal parameters in a method expression is one fewer than those of the equivalent lambda expression.

The local macro method looks more complicated than it is. Basically, it is the same as lambda but includes this as its first formal parameter. Because of variable hygiene, however, we have to make the macro system believe that this had been one of the operands of method. The "_" syntactic context indicates exactly which macro call to associate with the new variable. Thus, this is now ready to be treated as a piece of syntax. We place it as the first formal parameter of the resulting lambda expression. This use of with-implicit forces all free occurrences of this in body0 body1 ... to be treated as bound in the lambda expression.

Also, we improve the syntax of field access and update. Consider the line that starts out (let-syntax ([f ...]). Where j is f’s position, then each f is to be replaced by (fp this j), and each (set! f v) is to be replaced by (fp! this j v). In the first extender we constructed the subset of the field environment with trim-env. We need this particular subset of the field environment, since we care only about protected positions.

Next, we focus on (let-syntax ([m ...]). Each m call generates one of two calls. If it is a super call, we generate a procedure call using one of the procedures bound to, for example, the h0–h6 temporaries. Otherwise we generate the object method call. In the second variant, we use a let expression just in case the argument is other than a variable.

By including the super and the f ... m ... in the exact match list, we know that we only concern ourselves with the specific field and method variables we care about. All the other cases treat f ... m ... as lexical variables. The variable super is now treated as both a regular lexical variable and as a target of comparison (See Section 6.1.). Furthermore, we cannot forget to capture (f ... m ...) using the "_" passed in through trim-body.

These abbreviations have a real benefit. Not only do we get to use the familiar Scheme syntax, but we also get to remove the two lets that had been used to install the method and trimmed field environments of the first extender. Why? We can do this because we are inserting the actual positions in the expanded code.

Exercise 10: Now that we no longer need the run-time field environment, we observe that the only place it is used is in the definition of _n, which we redefine.

(define (_n c) (cons (make-vector (car c)) (cdr c)))

This definition clarifies that we need only store the field environment’s length, which can be determined at macro-expansion time. Change extender appropriately.

8.3 Expanding create-class

Below is the expansion of (create-class <<scp>> <cp>), where we manually put back the lets and or, and, where we use "-" as above. It differs from the definition that appears in Section 7.3 in four ways. First, the field and method environments have been built at macro-expansion time, thus obviating the need for fx and _mx; second, the installed let* and installed let have vanished; third, there is a let binding that refers to this, which vanishes at compile time; and fourth, (_m super isa?) has not been replaced by h0, since it does not take this as its implied argument.

(define <<scp>>
 (let ([super <cp>])
 (let ([h0 (_m super 0)])
  ...
 (h6 (_m super 6)))
 (let ([g0 (lambda (this c)
 (or (_m eq? this c)
 ((_m super 0) super c)))]
 [g1 (lambda (this x’ y’ hue’)
 (fp! this 3 "Struck: ")
 (h1 this x’ y’ hue’)])
 [g2 (lambda (this x’ y’)
 (let ([o- this]
 ((_m o? o’)])]
 [g3 (lambda (this a)
 (write (fp this 2))
 (h4 this a)])
 [g4 (lambda (this)
 (display (fp this 3)))]
 (let ([isa? h0]
  ---
 [diagiset h6])
 (let ([isa? g0]
 [init g1]
 [move g2]
 [diag g3]
 [show g4])
 (list
 '“(x y hue y)
 ’(isa? --- show g)
 (vector isa? --- show g))))))))

Exercise 11: We can extend Exercise 2 to include adding fresh and overriding methods. We can get recursion by using zero fields and forcing all but the first method access to be internal. Implement factorial, even?, and odd? as follows. Build <factorial> with fact as its only fresh method. Next, implement the two mutually-recursive methods iseven and isodd by defining a three element chain: <o>, <even?>, and <odd?>. The class <even?> contains the method iseven and the class <odd?> contains the method isodd. First solve it for a chain of <o> and <even?>/ <odd?>. ◇

Exercise 12: Design a syntax for a call like we use in the super call of isa, so that it expands to h0 instead of (_m super 0). ◇

8.4 Lexical scope versus protected scope

In this section we analyze how the new extender interacts with lexical scope. This semantics differs from that
of Section 8.1. There, we interposed the protected scope by installing artificial position environments, which should not necessarily shadow lexical variables. (This use of "shadow" should not be confused with the one described in Section 8.1.)

We have to make decisions about what variables lexically shadow what other variables. In `<escp>` below, what values of hue would we expect to be displayed if we passed `(create-class `<escp>` `<scp>`) to test? We would hope that any introduced lexical scope in the method such as the `(let ([hue "inside "] )) or the formal parameter of init would shadow the field variable, and indeed it does. Even the one hue bound outside the method expression of show-y displays the value of the lexical variable hue instead of the value of the protected field variable.

```
(define-syntax `<escp>
 (extend-shadow `<escp>` ()
  ([init
    (method (x - y - hue)
      (display hue)
      (init super x - y - hue)])
    show-y
    (let ([hue "outside "]
      [diag* (lambda (x y)
               (display "moving ")]))
      (method ()
        (display hue)
        (diag* 5 5)
        (let ([hue "inside "]
          [diag (lambda (n self)
                  (diag self n))]
          (display hue)
          (diag 5 this)))])))))
```

The expression `(diag 5 this)` is lexically scoped within `(let ([diag ...]))`, but the variable `diag` in the expression `(diag self n)` does indeed refer to the method. If we were to run our test program using `<escp>`, it would loop indefinitely, because of the mutually recursive tail calls from `diag` (contributed from `<scp>`) to `move` (contributed from `<scp>`), then to the host method `show-y`, and then back to `diag`. This loop would happen even if we were to replace self with another randomly chosen variable `super`, since the expansion of the `diag procedure` would then have been

```
(lambda (n super 1)
 (let ([o - super 1])
   ((mp o - 3) o - n)))
```

But, if we were to α-substitute `diag` by `diag` then `(test)` would terminate, after displaying `moving`.

Any free variables outside a method expression are lexically scoped. This decision is quite arbitrary, but any reasonable characterization of which variables shadow which other variables can be implemented using `syntax-case` (see Dybvig [4]). We believe that our structuring of the scopes is reasonable, since otherwise the programmer would have to know all the protected variables in the host’s chain. This is what we need if-shadowed for (Figure 2). It chooses the third operand if a variable inside a method expression is shadowed by a lexical variable bound in the scope outside of the method expression but within an `extend-shadow` expression. Our characterization is a two-edged sword, however. A program may work and then fail to work if a lexical variable bound outside a method expression but within an `extend-shadow` expression is inadvertently α-substituted by a protected variable. A general solution that circumvents this problem is to require that when in doubt, field and method variables should be private (see Exercises 7 and 8).

The example below shows that we have lost no lexical scope.

```
(define `<escp>`-maker
  (lambda (x)
    (let-syntax
      ([(<escp>)
        (extend-shadow `<escp>` ()
          (let ([x y]
              (method (q r . args)
                (+ x y q r (car args)))))])
        (lambda (a)
          (create-class `<escp>` s)))))
    > (let ([escp (n ((<escp>-maker 1 `<escp>')))]
             (mp escp 1) escp 10 20 7)
             (nv escp 'e) escp 1 1 1))
         114
```

First the digit 1 appears, since `x` is a free variable. Even though `x` is a protected variable within the method expression, the occurrence of `x` in `(write x)` is free within the `extend-shadow` expression. That would be true even if we introduced a fresh field or method variable, `x`. The protected variable `x` has the value 10 and it shadows the `x` outside the `extend-shadow` expression. The 14 is larger than 20, since `y` is scoped between a method expression and an `extend-shadow` expression. But, if we were to α-substitute `x` by `z`, the value would be 8. If we then replace `[y 1]` by `[t 5]`, the result would be 24, since now `y` would be protected instead of lexical and would have the value 20.

9. MORE EXTERNAL OPERATORS

We have three more external operators to define. The first is a more general variant of `n`. Each of the other two allow us to avoid computing a method variable (symbol) when we externally access a method.

We want to invoke the method `init`, which takes any number of arguments on each invocation of `n`. We support this protocol with the macro `new`, below.

```
(define-syntax new
  (syntax-rules ()
    [(_ c arg ...)
      (let ([o (n c)])
        ((mp o 1) o arg ...))])
```

15
There are two aspects of `new` that deserve mention. First, `new` takes at least one argument. That one argument should be a class. Second, we know that we are going to invoke the method `init`, whose position is 1.

We might want operators that allow us to avoid computing a method variable (symbol) when we are externally accessing it. We can do this in two ways. One way is by accessing the method using a quoted variable with `mbv` (invoke method by variable) below. This, however, still requires a run-time search to access the method.

```scheme
(define-syntax mbv
  (syntax-rules ()
    [(_ m oc arg ...)
      (let ((oc oc))
        (mbv oc "m" oc arg ...))]))
```

Alternatively, we can invoke a method whose position has been determined at macro-expansion time with `invoke` below. This is where we use the clause in build-shadow whose pattern is `(_, an-m-var oc)`. But, we need to refer to some shadow that contains the appropriate method variable `m`.

```scheme
(define-syntax invoke
  (syntax-rules ()
    [(_ shadow m oc arg ...)
      (let ((oc oc))
        (invoke shadow m oc arg ...))]))
```

We cannot always use `invoke`. Only rarely is it necessary to use `mbv`, and it is even less likely that one would need the full unbridled power of `mbv`.

Consider `test-<scp>` below.

```scheme
(define test-<scp>
  (lambda ()
    (let ([p (new <p> 1 2)]
          [scp (new <scp> 18 19 9)])
      (invoke scp diag&set scp 10)
      (list
        (list
          (invoke scp diag&set scp)
          (invoke scp get-hue scp)
          (isa? scp scp)
          (isa? p scp))))))
```

We define a useful predicate `isa?`, and we restrict external method access to be through the use of `invoke` or `mbv`. Hence, `map-nullary-method` can no longer be used. Why?

Exercise 13: Hand-build `<<o>>` and `<o>` without using build-<<o>>, build-shadow, or create-class. We know that when we create class `<o>`, we do it by passing in `#f`. Therefore, we can do some of the expansion ourselves. The `#f` gets bound to `super`, but we know that `<<o>>` has no super methods, so only the second and fourth parts of the quadruply-nested `let` are used. Then it becomes obvious that we can go from this doubly-nested to a single `let`.

Exercise 14: Implement the following variant on the scope rules. Instead of allowing protected variables to shadow some lexical variables, just have the host’s protected variables as possibly shadowing some lexical variables. Next, implement scope rules to make lexical variables shadow protected variables.

Exercise 15: Why is it okay to use `<cp>` in `(invoke scp)` instead of `<scp>` even though `scp` is of class `<scp>`? What would happen if we replaced it with `<p>`?

Exercise 16: Implement the macro-expansion-time analog of `isa?`, which expands to `#f` if one shadow is in the chain of another shadow. Otherwise, it expands to `#f`.

10. CONCLUSIONS

Our goal has been to demonstrate that by thinking about the actual structure of the method and field environments and their associated vectors, we can better appreciate the sublety of the semantics of object-oriented programming.

We have done this by first developing an object-oriented style, then protocols, and finally build-shadow with some supporting macros. Because enough of the system has been built with macros, all the shadows can be built at macro-expansion time.

Let us revisit our goals. Most conventional languages use implementation technology that makes writing recursive programs a risky enterprise. Now, object-oriented programming has inadvertently made recursion the natural way to write programs. But still the implementation technology does not yet allow the recursion to be exploited, since most (object-oriented) languages do not support tail-call optimization.

Also, a goal has been to clarify just how different super method invocation is from object method invocation. Knowing that `super` is static should make the distinction obvious. Furthermore, the quadruply-nested `let` of Section 7.3 treats super calls as regular procedure calls, where the super procedure is bound to a lexical variable.

The build-shadow macro is merely a compiler for the language supported in the definition of `<scp>` and `<scp>` of Section 2. The compiler, among other activities, replaces all the position variables with their corresponding positions. And, as we demonstrate in Section 8.3, this compiler produces code virtually the same as our examples of object-oriented style in Section 5.2.

We have been able to preserve the property that when the scope of a variable is in question within an extend-shadow expression, lexical scope shadows the protected scope of field and method variables. We show in Section 8.4 that extend-shadow and any method within extend-shadow can be wrapped in lexical scope, so no scope whatsoever has

---

5The designers of C# claim, however, to support tail-call optimization, at least in spirit.
been lost. Also, the body of a method, itself, can evaluate to a procedure. A fresh or overriding method passed to \texttt{extend-shad} don’t be introduced with a \texttt{method} expression (or \texttt{lambda} expression). It can be any legal expression. Furthermore, if the expression’s value happens to be a procedure of at least one argument, even if it is not wrapped within the \texttt{method} keyword, it can be invoked in object-oriented style. This is a feature, however, that should be used with extreme caution since then it could be a variable bound to a first-class continuation or it could be a procedure that reveals the internal representation of objects and classes.

We have assumed that the user always writes syntactically correct programs. Robustness of macros, however, must be taken quite seriously and tools should be developed to guarantee that the patterns in a macro are not only matched, but that other protocols, such as \textit{The fresh variables introduced in each host have no duplicates}, are also supported.

This development says something about macros, in general. Macros have come a long way from the early days of Lisp. Now, with hygiene, \texttt{with-syntax}, \texttt{syntax-case}, etc., we see that macros are powerful enough to write sophisticated compilers. More importantly, the compiler can be written so that the expressions are not explicitly traversed by the macro writer's code, improving its portability.

Our analogy with continuation-passing style is not fully developed. We know that both methods and continuations get their extra power with an extra argument. That extra power comes from a careful harnessing of different dynamic aspects of computation. Our future plans include formally characterizing our object-oriented style. There is clearly a style here, but we have so far just been able to describe it informally. A concern with formalizing our style is that the subtle issues of wiring the lexical scope and making the encoding robust enough to be an embedding might be overlooked. Also, we hope to have transformers to object-oriented style just as there are transformers to continuation-passing style.

From the beginning, our plan has been the transmutation of symbols into variables. This effort has taken longer than one would have expected, given the relative simplicity of its solution. Nevertheless, this is just our first result of this kind where symbols are not accepted as the natural currency of computation. A symbol, of course, is a run-time value. Now that our macro systems have become so sophisticated, we have an opportunity to think much harder about processing programs well before they run and turning other classes of symbols into variables.

11. ACKNOWLEDGEMENTS

I want to thank Mitch Wand for his thorough reading and numerous suggestions. I appreciate the careful reading of drafts by Erik Hilsdale, Oleg Kislavov, Ron Garcia, Jeremy Siek, Steve Ganz, Jonathan Sobel, James Pendry, Kevin Millikin, Ken Anderson, Abdulaziz Ghuloum, and Thomas Reicharder. I am grateful for conversations that I have had with Jon Rossie about his views on object-oriented programming. A recent observation by Michael Greenberg resulted in many improvements. Mark Meiss pointed out how \texttt{let*} was the correct binding operator for field positions. I want to thank the students in my Fall 2003 graduate-level programming languages course for offering valuable criticism and working the exercises. Kent Dybvig made some very insightful observations that led to a clearer presentation and a simpler system. I am, as always, most grateful for his efforts on Chez Scheme. Amir Sabry's amusement at my way of explaining object-oriented programming as a natural extension of \texttt{letrec} spurred this effort to its natural conclusion.

I am surrounded by "magnificent macrologists." Oscar Waddell showed me how to permit wrapping of scope around a method expression. I want to acknowledge Erik Hilsdale for his part in implementing JOB (Java Objects for Scheme) and Anurag Mendhekar and Chris Haynes for their part in implementing Scheme++ (C++ Objects for Scheme), both logical predecessors of this effort. Erik also helped craft \texttt{build-shadow} and the two \texttt{extender} macros, which were based on ideas we developed in JOB. Oleg Kislavov proposed the current \texttt{isa}? removing the last vestige of an explicit use of recursion, and he contributed some useful improvements to the second \texttt{extender}. Abdulaziz Ghuloum put the finishing touches on the second \texttt{extender} by allowing lexical variables bound within an \texttt{extend-shadow} expression to lexically shadow protected variables and by having lexical variables hold the values of super methods.

12. REFERENCES


