To appear in POPL 2007

First-Class Nonstandard Interpretations by Opening Closures

Jeffrey Mark Siskind
School of Electrical and Computer Engineering
Purdue University, USA
qobi@purdue.edu

Barak A. Pearlmutter
Hamilton Institute
NUI Maynooth, Ireland
barak@cs.nuim.ie

Abstract

We motivate and discuss a novel functional programming construct that allows convenient modular run-time nonstandard interpretation via reflection on closure environments. This `map-closure` construct encompasses both the ability to examine the contents of a closure environment and to construct a new closure with a modified environment. From the user's perspective, `map-closure` is a powerful and useful construct that supports such tasks as tracing, security logging, sandboxing, error checking, profiling, code instrumentation and metering, run-time code patching, and resource monitoring. From the implementor's perspective, `map-closure` is analogous to `call/cc`. Just as `call/cc` is a non-referentially-transparent mechanism that reifies the continuations that are only implicit in programs written in direct style, `map-closure` is a non-referentially-transparent mechanism that reifies the closure environments that are only implicit in higher-order programs. Just as CPS conversion is a non-local but purely syntactic transformation that can eliminate references to `call/cc`, closure conversion is a non-local but purely syntactic transformation that can eliminate references to `map-closure`. We show how the combination of `map-closure` and `call/cc` can be used to implement `set!` as a procedure definition and a local macro transformation.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language constructs and features*]: Procedures, functions, and subroutines

General Terms Design, Languages

Keywords Referential transparency, Lambda lifting

1. Motivation

Nonstandard interpretation is a powerful tool, with a wide variety of important applications. Typical techniques for performing nonstandard interpretation are compile-time only, require modification of global resources, or require rewriting of code to abstract over portions subject to nonstandard semantics. This paper proposes a construct to support modular run-time nonstandard interpretation.

For expository purposes, let us consider a very simple example of nonstandard interpretation. Suppose one wished to add complex numbers and complex arithmetic to a programming-language implementation that supports only real arithmetic. One might represent the complex number $a + bi$ as an Argand pair $\langle a, b\rangle$. Extending the programming language to support complex arithmetic can be viewed as a nonstandard interpretation where real numbers r are lifted to complex numbers $\langle r, 0\rangle$, and operations such as $+ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$ are lifted to $+ : \mathbb{C} \times \mathbb{C} \rightarrow \mathbb{C}$.

$$\langle a_1, b_1\rangle + \langle a_2, b_2\rangle = \langle a_1 + a_2, b_1 + b_2\rangle$$

One can accomplish this in SCHEME by redefining the arithmetic primitives, such as `+`, to operate on combinations of native SCHEME reals and Argand pairs $\langle a, b\rangle$ represented as SCHEME pairs `(`a `.` b`)`. For expository simplicity, we ignore the fact that many of SCHEME's numeric primitives can accept a variable number of arguments. We define a new procedure `lift-+` which we use to redefine `+` at the top level.

```
(define (lift-+ +)
 (lambda (x y)
  (let ((x (if (pair? x) x (cons x 0)))
        (y (if (pair? y) y (cons y 0))))
   (cons (+ (car x) (car y))
         (+ (cdr x) (cdr y))))))

(define + (lift-+ +))
```

This raises an important modularity issue. With the above definition, one can take a procedure `f` defined as

```
(define (f x) (+ x x))
```

and correctly evaluate `(f '(2 . 3))` to `(4 . 6)`. One can even take a procedure `g` defined as

```
(define g (let ((y 10)) (lambda (x) (+ x y))))
```

and correctly evaluate `(g '(1 . 2))` to `(11 . 2)`. These examples work correctly irrespective of whether `f` and `g` are defined before or after `+` is redefined. In contrast, consider

```
(define h (let ((p +)) (lambda (x) (p x 5))))
```

The expression `(h '(1 . 2))` will evaluate correctly to `(6 . 2)` only if `h` was defined after `+` has been redefined. This is not the only modularity issue raised by this common technique: for instance, one might wish to confine the nonstandard interpretation to a limited context; one might wish to perform different nonstandard interpretations, either singly or cascaded; and one might wish to avoid manipulation of global resources.

The remainder of this paper discusses a novel mechanism, `map-closure`, which allows such nonstandard interpretation in code written in a functional style, and which avoids these modularity issues. As discussed in section 5, `map-closure` is a powerful and useful construct that supports such tasks as tracing, security logging, sandboxing, error checking, profiling, code instrumentation and metering, run-time patching, and resource monitoring.

POPL'07 January 17–19, 2007, Nice, France.

2. The MAP-CLOSURE Construct

The inherent difficulty in the above example is that while `f` and `g` access the addition procedure using the global variable `+`, which has been overloaded, `h` is closed over its own local variable `p`. In most higher-order programming languages, closures are opaque: there is no way to examine or modify closure environments. To address the above modularity issue we introduce a reflection operator, `map-closure`, which can examine the contents of a closure environment and construct a new closure with a modified environment.

We adopt a simple model for closures in which they are represented as pairs containing an environment and a code pointer. The environment in turn is represented as a mapping of free variable names to values, $\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}$. The behavior of `map-closure` is simple: it takes two procedure arguments f and g, and returns a procedure h whose code is identical to that of g but whose environment is $\{x_1 \mapsto f(x_1, v_1), \ldots, x_n \mapsto f(x_n, v_n)\}$. We also allow `map-closure` to apply to primitive procedures, taking their environments to be empty.

Note that `map-closure` is not referentially transparent. Moreover, the above formulation of `map-closure` is appropriate only for dynamically-typed languages, since it uses a single f to transform all of the closed-over values in g and these may be of different types. Incorporating `map-closure` into a statically typed language would be an interesting topic for future research, as would be the formulation of a referentially-transparent construct for first-class nonstandard interpretation.

Given `map-closure`, we can define a procedure that substitutes x for y in z.

```
(define (substitute x y z)
 (cond
  ((equal? y z) x)
  ((pair? z)
   (cons (substitute x y (car z))
         (substitute x y (cdr z))))
  ((procedure? z)
   (map-closure (lambda (n z) (substitute x y z))
                z))
  (else z)))
```

With this, we can overload the arithmetic operators inside closures.

```
(define (with-complex thunk)
 ((substitute (lift-+ +) + thunk)))
```

If our implementation represents constants as free variables, we can alternatively lift not only the arithmetic operators but also numeric values.

```
(define (lift-+ +)
 (lambda (x y)
  (cons (+ (car x) (car y))
        (+ (cdr x) (cdr y)))))

(define (lift-r->c x)
 (cond
  ((real? x) (cons x 0))
  ((pair? x)
   (cons (lift-r->c (car x))
         (lift-r->c (cdr x))))
  ((procedure? x)
   (map-closure (lambda (n x) (lift-r->c x)) x))
  (else x)))

(define (with-complex thunk)
 ((substitute (lift-+ +) + (lift-r->c thunk))))
```

Note that this removes the dispatches at every call to `+` in the previous version, a sort of dynamic typing, and instead lifts all real values into complex numbers once, at reflection time: a sort of runtime static typing. However, just as `call/cc` [1] complicates certain compiler optimizations, the presence of `map-closure` forces compilers wishing to perform certain types of optimization, such as constant folding, to do additional static analysis.

It is important to emphasize the power that `map-closure` affords as illustrated by `with-complex` in the above example. Suppose a user obtains some library code that uses addition. As written, the library code only performs real arithmetic since it uses the built in addition primitive. The user can extend that library code to apply to complex arithmetic *without access to or modification of the source code*. The user does not need to wrap either the definition of addition nor the procedures in the library with calls to `with-complex`. All the user need do is wrap call sites to the library with calls to `with-complex`. In fact, the user need not do even this: wrapping the top-level main program with a single call to `with-complex` suffices.

The key contribution here is that `map-closure` is a general-purpose programming construct that allows nonstandard interpretation of library code without programmer access to and modification of said library code. Moreover, if primitives are not (prematurely) inlined, `map-closure` allows such nonstandard interpretation without compiler access to or modification of source code.

For purposes of exposition we have motivated `map-closure` using the well-known concept of complex arithmetic. Our interest in this mechanism arose in a slightly different context: an attempt to integrate a first-class derivative-taking operator into SCHEME using the method of forward-mode automatic differentiation [2], which performs arithmetic on dual numbers $x + x'\varepsilon$ in a fashion roughly analogous to that used for complex numbers [3].

There is an interesting analogy between `map-closure` and `call/cc`. Just as `call/cc` reifies the continuations that are only implicit in a program written in direct style, `map-closure` reifies the closure environments that are only implicit in a higher-order program. It is well known that CPS conversion [4, 5] eliminates `call/cc`. Similarly, closure conversion [6] eliminates `map-closure`. We illustrate this using a simple dialect of SCHEME defined in the next section. This implementation is available at `http://www.bcl.hamilton.ie/~qobi/map-closure/`.

With both of the above implementations of `with-complex` the scope of the lifting is limited to the dynamic extent of the invocation of the thunk argument. However, it is tempting to try to implement complex arithmetic in a form where the lifting is permanent. This can be accomplished by invoking `(call/cc with-complex)`, assuming that the implementation does not impose any arbitrary restrictions on the arguments to `map-closure`. Note that this applies to both the past and the future. It modifies all bindings, both past and future, via recursive descent into the continuation. The mechanics and consequences of allowing `map-closure` to be applied to continuations are discussed below. The next section describes a simple functional language which facilitates exploration of the constructs under consideration. We then review the process of closure conversion, show how closure conversion eliminates `map-closure`, and discuss how `map-closure` in combination with `call/cc` allows `set!` to be implemented as a defined procedure and a local macro transformation.

3. A Functional Subset of SCHEME

We formulate these ideas using a simple functional language that resembles SCHEME [7], differing in the following respects:

- The only data types supported are Booleans, reals, pairs, and procedures.

- Only a subset of the builtin SCHEME procedures and syntax are supported.
- Rest arguments are not supported.
- The constructs `cons` and `list` are macros:

  (cons e_1 e_2) $\rightsquigarrow$ ((cons-procedure e_1) e_2)

  (list) $\rightsquigarrow$ '()

  (list e_1 $e_2 \ldots$) $\rightsquigarrow$ (cons e_1 (list $e_2 \ldots$))
- Procedure parameters p can be variables, `'()` to indicate an argument that is ignored, or (cons p_1 p_2) to indicate the appropriate destructuring.
- All procedures take exactly one argument and return exactly one result. This is accomplished in part by the basis, in part by the following transformations:

  (e_1) $\rightsquigarrow$ (e_1 '())

  (e_1 e_2 e_3 $e_4 \ldots$) $\rightsquigarrow$ (e_1 (cons* e_2 e_3 $e_4 \ldots$))

  (lambda () e) $\rightsquigarrow$ (lambda ((cons*)) e)

  (lambda (p_1 p_2 $p_3 \ldots$) e)
  $\rightsquigarrow$ (lambda ((cons* p_1 p_2 $p_3 \ldots$)) e)

  together with a `cons*` macro

  (cons*) $\rightsquigarrow$ '()

  (cons* e_1) $\rightsquigarrow$ e_1

  (cons* e_1 e_2 $e_3 \ldots$) $\rightsquigarrow$ (cons e_1 (cons* e_2 $e_3 \ldots$))

  and by allowing `list` and `cons*` as parameters.

The above, together with the standard SCHEME macro expansions, a macro for `if`

(if e_1 e_2 e_3) $\rightsquigarrow$
 ((if-procedure e_1 (lambda () e_2) (lambda () e_3)))

and a transformation of `letrec` into the Y-combinator suffice to transform any program into the following core language:

$$e ::= (\texttt{quote}\ v) \mid x \mid (e_1\ e_2) \mid (\texttt{lambda}\ (x)\ e)$$

4. Closure Conversion

The essence of closure conversion is to reify environments that contain the values of free variables in procedures by replacing procedures with pairs of environments and a transformed procedure. These transformed procedures have no free variables, and instead access the values of free variables from the reified environment passed as an argument. This can be implemented as a purely syntactic source-to-source transformation, as shown in figure 1.

We omit a number of bookkeeping details tangential to the issues we wish to explore. However, one bookkeeping issue relevant to our purpose does arise. We would like our new reflective mechanism to be invariant to choice of variable names. We therefore introduce a new data type, *name*, to key environments. The interface for names consists of the procedures `name?` and `name=?`, and the syntax (name x) which returns a unique name associated with the (potentially alpha-renamed) variable x.

Given this transformation, `map-closure` can be transformed to

```
(lambda (c (cons (cons f fc) (cons g gc)))
 (cons g
       (map (lambda (gn gv)
             (cons gn (f fc gn gv)))
            gc)))
```

The techniques described in this section and shown in figure 1 suffice to implement the examples of section 2. While the simple implementation in figure 1 represents reified closure environments as alists and transformed procedures as pairs, `map-closure` does not expose this structure. An alternate implementation could thus use an alternate representation with suitable replacements for `lookup`, `map`, and the locations in the transformation where closures are constructed. Such an implementation might represent names as offsets into environment tuples.

5. The Utility of MAP-CLOSURE

Both alone and in combination with `call/cc`, `map-closure` is a powerful and general-purpose construct that can solve important software-engineering tasks. It is a portable mechanism for performing run-time dynamic nonstandard interpretation, a technique of increasing importance that arises in many guises ranging from security and debugging to web applications (mechanisms like AJAX that overload I/O operations to use HTTP/HTML). Consider the following examples as an indication of its myriad potential uses.

Programmers often desire the ability to examine an execution trace. Figure 2 contains a `trace` procedure that traces *all* procedure entrances and exits during the invocation of `thunk`. Such a facility can easily be adapted to perform security logging.

Virtual machines are often able to execute code in a sandbox so as to constrain the allowed actions and arguments. Figure 2 contains a `sandbox` procedure that invokes `thunk` in a context where *all* procedure invocations must satisfy the `allowed?` predicate or else the `raise-exception` procedure is called. Such a facility is useful both for security and error checking.

Many programming-language implementations contain a facility to profile code. Figure 2 contains a `profile` procedure that constructs a table of the invocation counts of *all* procedures invoked during the invocation of `thunk`. Such a facility can easily be adapted to instrument and meter code in other ways.

One of the hallmarks of classical LISP implementations is the ability to patch code in a running system by changing the function bindings of symbols. The designers of COMMON LISP were aware that this mechanism could not be used to patch code referenced in closure slots. They addressed this with a kludge: treating a `funcall` to a symbol as a `funcall` to its function binding. Figure 2 contains a more principled approach to this problem. The procedure `patch` replaces *all* live instances of `old` with `new`.

Finally, many programming-language implementations contain a facility to determine the amount of live storage. Figure 2 contains a `room` procedure that returns a list of the number of live pairs and the number of live closure slots.

Facilities such as the above are normally implemented as system internals. Figure 2 shows that many such facilities can be implemented as user code with `map-closure`.[1]

6. MAP-CLOSURE + CALL/CC = SET!

It is interesting to consider the application of `map-closure` to a continuation made explicit by `call/cc`. The source-to-source transformation of closure conversion described in section 4 does not allow this, because it does not closure-convert continuations. However, we could convert the program to continuation-passing style (CPS) first and then apply closure conversion, thus exposing all continuations to closure conversion as ordinary procedures. Figure 3 describes this process. The transformations shown are standard, with one exception: the `map-closure` procedure itself needs to be handled specially, as (prior to closure conversion) it cannot be expressed as a user-defined procedure, and must be treated as a primitive. However, it is unique among primitives in that it invokes a procedural argument. Since this procedural argument will

[1] The examples in figure 2 use a number of constructs not (yet) provided by our prototype implementation, namely (implicit) `begin`, `write`, `newline`, `eq?`, `memq`, and named `let`. It is straightforward to add such constructs.

```
(C c (quote v))        ~>  (quote v)
(C c (name x))         ~>  (name x)
(C c x)                ~>  (lookup (name x) c)
(C c (lambda (x) e))   ~>  (cons (lambda (c1 x) (let ((c2 (cons (cons (name x) x) c1))) (C c2 e)))
                                 (list (cons (name x1) (C c x1)) ...))
                           where x1 ... are free in (lambda (x) e)
(C c (e1 e2))          ~>  (let ((x (C c e1))) ((car x) (cdr x) (C c e2)))
e0                     ~>  (let ((x1 (cons (lambda (c x) (x1 x)) '())) ...
                                 (cons-procedure
                                  (cons (lambda (c1 x1) (cons (lambda (c2 x2) (cons x1 x2)) '())) '()))
                                 (map-closure
                                  (cons (lambda (c (cons (cons f fc) (cons g gc)))
                                         (cons g (map (lambda (gn gv) (cons gn (f fc gn gv))) gc)))
                                        '()))
                                 (pair? (cons (lambda (c x) (and (pair? x) (not (procedure? (car x)))))
                                              '()))
                                 (procedure? (cons (lambda (c x) (and (pair? x) (procedure? (car x))))
                                                   '())))
                            (let ((x (list (cons (name x1) x1) ...
                                           (cons (name cons-procedure) cons-procedure)
                                           (cons (name map-closure) map-closure)
                                           (cons (name pair?) pair?)
                                           (cons (name procedure?) procedure?))))
                             (C x e0)))
                           where x1 ... are free in e0 except cons-procedure, map-closure, pair?, and procedure?. This
                           assumes that x1 ... are bound to procedures that do not internally invoke procedural arguments.
```

Figure 1. Closure-conversion algorithm that applies to the top-level expression e_0.

be in CPS after conversion, the CPS version of `map-closure` must invoke this argument with an appropriate continuation.

The combination of `map-closure` and `call/cc` is very powerful: it can be used to implement `set!` as a procedure definition in a language that does not have any built-in mutation operations. The intuition behind this is that `set!` changes the value of a variable for the remainder of the computation; `call/cc` exposes the remainder of the computation as a reified continuation; `map-closure` can make a new continuation just like the old one except that one particular variable has a new value; and thus invoking this new continuation instead of the old continuation has precisely the same result as `set!`. The simple definition shown in figure 4 accomplishes this intuition. There is, however, one minor complication: the recursion in `set-in` is necessary because the target variable might be present in closures nested in the environments of other closures. As a result, unlike most SCHEME implementations, where `set!` takes constant time, the implementation in figure 4 must traverse the continuation to potentially perform substitution in multiple environments that close over the mutated variable.

While the ability to implement `set!` as a procedure definition combined with a local macro transformation is surprising and intriguing, it might be reasonable to consider this to be something of a curiosity. The combination of `map-closure` and `call/cc` is extremely powerful, and thus potentially difficult to implement efficiently. However `map-closure` in the absence of `call/cc` is still a useful construct for implementing nonstandard interpretation, and seems amenable to more efficient implementation. Thus, implementations supporting `map-closure` might not in general be expected to allow its application to continuations. Of the examples in figure 2, only `patch` and `room` rely on this ability.

7. Discussion

Functor-based module systems [8], overloading mechanisms such as aspect-oriented programming [9], and `map-closure` are related, in that all three support nonstandard interpretation. The difference is in the scope of that nonstandard interpretation. In a functor-based module system, the scope is lexical. With overloading, the scope is global. With `map-closure`, the scope is dynamic.

The dynamic scope of `map-closure` affords interesting control over modularity. One can apply a nonstandard interpretation to only part of a program. Or, different nonstandard interpretations to different parts of a program. Or, to different invocations of the same part of a program. One can compose multiple nonstandard interpretations, controlling the composition order when they do not commute. For example, composing complex arithmetic with logging arithmetic in different orders would allow one to control whether one logged the calls to complex arithmetic or the calls to the operations used to implement complex arithmetic. With `map-closure`, nonstandard interpretations become first-class entities.

If all aggregate data structures are Church-encoded as closures, CPS conversion followed by closure conversion subsumes store conversion: it explicitly threads a store, represented as an environment, through the program. However, compilers that perform both CPS conversion and closure conversion generally do so in the opposite order. Just as `call/cc` affords one the power of explicit continuations while allowing one to write in direct style, `map-closure` affords one the power of explicit closure environments while allowing one to write in higher-order style. The combination of `call/cc` and `map-closure` affords the power of explicit store threading while allowing one to write in a direct higher-order style.

In the implementation of `set!` in figure 4, the original continuation is not mutated but discarded. Instead of discarding this original continuation, it can be preserved and invoked later in order to implement such control structures as `fluid-let` [5] and `amb` [10] with associated side effects that are undone upon backtracking [11]. Side effects that can be undone can be used to implement PROLOG-style logic variables and unification [12]. All this can be implemented as defined procedures and local macro transformations in a language that has no explicit mutation operations, but that supports `call/cc` and `map-closure`, allowing `map-closure` to apply to continuations.

Like other powerful constructs, `map-closure` may seem difficult to implement efficiently. However, the same was said of constructs like recursion, dynamic typing, garbage collection, and

```
(define (trace thunk)
 ((let wrap ((x thunk))
   (cond ((pair? x) (cons (wrap (car x)) (wrap (cdr x))))
         ((procedure? x)
          (lambda (arguments)
           (write (list +1 procedure arguments))
           (newline)
           (let ((result ((map-closure (lambda (n x) (wrap x)) x) arguments)))
            (write (list -1 procedure result))
            (newline)
            result)))
         (else x)))))

(define (sandbox allowed? raise-exception thunk)
 ((let wrap ((x thunk))
   (cond ((pair? x) (cons (wrap (car x)) (wrap (cdr x))))
         ((procedure? x)
          (lambda (arguments)
           (if (allowed? x arguments) ((map-closure (lambda (n x) (wrap x)) x) arguments) (raise-exception))))
         (else x)))))

(define (profile thunk)
 (let* ((table '())
        (result ((let wrap ((x thunk))
                  (cond ((pair? x) (cons (wrap (car x)) (wrap (cdr x))))
                        ((procedure? x)
                         (lambda (arguments)
                          (set! table (let increment ((table table))
                                       (cond ((null? table) (list (cons x 1)))
                                             ((eq? (car (car table)) x)
                                              (cons (cons (car (car table)) (+ (cdr (car table)) 1)) (cdr table)))
                                             (else (cons (car table) (increment (cdr table)))))))
                          ((map-closure (lambda (n x) (wrap x)) x) arguments)))
                        (else x))))))
  (write table)
  (newline)
  result))

(define (patch old new)
 (call/cc (lambda (c)
           ((let wrap ((x c))
             (cond ((eq? x old) new)
                   ((pair? x) (cons (wrap (car x)) (wrap (cdr x))))
                   ((procedure? x) (map-closure (lambda (n x) (wrap x)) x))
                   (else x)))
            #f))))

(define (room)
 (let ((pairs 0) (slots 0) (objects '()))
  (call/cc (lambda (c)
            (let walk ((x c))
             (cond ((memq x objects) #f)
                   (else (set! objects (cons x objects))
                         (cond ((pair? x) (set! pairs (+ pairs 1)) (walk (car x)) (walk (cdr x)))
                               ((procedure? x) (map-closure (lambda (n x) (set! slots (+ slots 1)) (walk x)) x))))))))
  (list pairs slots)))
```

Figure 2. Typical LISP and SCHEME system functionality implemented as user code with `map-closure`.

```
(C c (quote v))        ⇝  (c (quote v))
(C c (name x))         ⇝  (c (name x))
(C c x)                ⇝  (c x)
(C c (lambda (x) e))   ⇝  (c (lambda (c1 x) (C c1 e)))
(C c (e1 e2))          ⇝  (C (lambda (x1) (C (lambda (x2) (x1 c x2)) e1)) e2)
e0                     ⇝  (let ((x1 (lambda (c x) (c (x1 x)))) ...
                                 (call/cc (lambda (c x1) (x1 c (lambda (c2 x2) (c x2)))))
                                 (cons-procedure (lambda (c1 x1) (c1 (lambda (c2 x2) (c2 (cons x1 x2))))))
                                 (map-closure (lambda (c (cons f g))
                                               (c (map-closure (lambda (x) (f (lambda (x) x) x)) g)))))
                           (C (lambda (x) x) e0))
```

where $x_1 \ldots$ are free in e_0 except `call/cc`, `cons-procedure`, and `map-closure`. This assumes that $x_1 \ldots$ are bound to procedures that do not internally invoke procedural arguments.

Figure 3. CPS-conversion algorithm that applies to the top-level expression e_0.

```
(define (set-in n v c)
 (cond ((procedure? c) (map-closure (lambda (n1 v1) (if (name=? n n1) v (set-in n v v1))) c))
       ((pair? c) (cons (set-in n v (car c)) (set-in n v (cdr c))))
       (else c)))

(define (set n v) (call/cc (lambda (c) ((set-in n v c) #f))))

(define-syntax set! (syntax-rules () ((set! x e) (set (name x) e))))
```

Figure 4. An implementation of `set!` using `map-closure` and `call/cc`.

`call/cc` when first introduced. Of particular concern is that it may appear that `map-closure` precludes compiler optimizations such as inlining, especially of primitive procedures. Well known techniques (e.g., declarations, module systems, and flow analysis) allow SCHEME compilers to perform inlining despite the fact that the language allows redefinition of (primitive) procedures. These techniques can be extended and applied to allow inlining in the presence of `map-closure`. Even without such techniques, `map-closure` does not preclude inlining: a compiler can generate whatever code it wishes, so long as the run-time system can reconstruct the closure-slot information that `map-closure` passes to its first argument, and any information needed to construct the result closure. Each invocation of `map-closure` might even perform run-time compilation, including optimizations such as inlining.

The history of programming-language research is replete with examples of powerful constructs that were initially eschewed for performance reasons but later became widely adopted as their power was appreciated and performance issues were addressed. We hope that this will also be the case for `map-closure`.

Note that, by design, `map-closure` does not expose the internal representation of closures and environments to the user. This design also preserves hygiene: the lexical hierarchy of variable scoping. Since `map-closure` does not allow one to add, remove, or rename variables, it is not possible to create unbound variable references or change the lexical scoping of variables through shadowing or unshadowing at run time.

An alternate, more traditional way to provide the functionality of `map-closure` would be to provide an interface to access the environment and code components of closures and construct new closures out of such environment and code components, along with an interface to access environment components and construct new environments. However, such an alternate interface would expose the internal representation of closures and environments to the user, perhaps via interfaces and data types that differ in detail between implementations, and might well break hygiene. On the other hand, `map-closure` exposes only one new data type: names as passed as the first argument to the first argument of `map-closure`. The values passed as the second argument to the first argument of `map-closure` and the values returned by the first argument of `map-closure` are ordinary SCHEME values.

Also note that names are opaque. They are created by new syntax to allow implementations to treat them as variables in every sense. They can only be compared via identity, so an implementation is free to represent names in the same way as variable addresses: stack offsets, absolute global addresses, etc. In fact, just as implementations can have different representations of variable addresses for variables of different types and lifetimes, implementations can have similarly different representations of names. Moreover names can be avoided entirely by using a weaker variant of `map-closure` that only exposes closure-slot values. Such a weaker variant suffices for many applications, including all examples here except for the implementation of `set!`.

Closure conversion is not the only implementation strategy for `map-closure`. For instance, a native implementation could operate directly on higher-order code. Such an implementation would only need a mechanism for accessing slots of existing closures and creating closures with specified values for their slots. These mechanisms already exist in any implementation of a higher-order language, and must simply be repackaged as part of the implementation of a `map-closure` primitive. Furthermore, native implementations of `map-closure` are possible in systems that use alternate closure representations, such as linked or display closures, unlike the flat-closure representation used here. While the *implementation* of `map-closure` for different representations of closures and environments would be different, programs that *use* `map-closure` would be portable across all such implementations. This is not the case with the aforementioned alternate interface.

Nonstandard interpretation is ubiquitous in programming language theory, manifesting itself in many contexts. It could be reasonably suggested that the lack of a simple way to easily perform a nonstandard interpretation may have held back the application of this powerful idea, and resulted in a great deal of implementation effort building systems that each perform some specific nonstandard interpretation. For this reason `map-closure`, or some other construct that provides first-class dynamic nonstandard interpretation, may prove a surprisingly handy tool. In fact, the authors have already found it quite useful in the implementation of automatic differentiation in a functional programming language.

Acknowledgments

This work was supported, in part, by NSF grant CCF-0438806, Science Foundation Ireland grant 00/PI.1/C067, and a grant from the Higher Education Authority of Ireland. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] W. Clinger, D. P. Friedman, and M. Wand, "A scheme for a higher-level semantic algebra," in *Algebraic Methods in Semantics*, J. Reynolds and M. Nivat, Eds. Cambridge University Press, 1985, pp. 237–250.

[2] R. E. Wengert, "A simple automatic derivative evaluation program," *Comm. of the ACM*, vol. 7, no. 8, pp. 463–4, 1964.

[3] W. K. Clifford, "Preliminary sketch of bi-quaternions," *Proceedings of the London Mathematical Society*, vol. 4, pp. 381–395, 1873.

[4] J. C. Reynolds, "Definitional interpreters for higher-order programming languages," in *Proceedings of the 25th ACM National Conference*, 1972, reprinted in *Higher Order and Symbolic Computing*, 11(4):363–397, 1998.

[5] G. L. Steele, Jr. and G. J. Sussman, "Lambda, the ultimate imperative," MIT Artificial Intelligence Laboratory, A. I. Memo 353, Mar. 1976.

[6] T. Johnsson, "Lambda lifting: Transforming programs to recursive equations," in *Functional Programming Languages and Computer Architecture*. Nancy, France: Springer-Verlag, Sept. 1985.

[7] W. Clinger and J. Rees, *Revised^4 Report on the Algorithmic Language* SCHEME, Nov. 1991.

[8] D. MacQueen, "Modules for Standard ML," in *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, Austin, TX, 1984, pp. 198–207.

[9] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming*, 1997, pp. 220–242.

[10] J. McCarthy, "A basis for a mathematical theory of computation," in *Computer Programing and Formal Systems*, P. Braffort and D. Hirschberg, Eds. Amsterdam: Elsevier North-Holland, 1963.

[11] J. M. Siskind and D. A. McAllester, "SCREAMER: A portable efficient implementation of nondeterministic COMMON LISP," University of Pennsylvania Institute for Research in Cognitive Science, tech. report IRCS-93-03, 1993.

[12] ——, "Nondeterministic LISP as a substrate for constraint logic programming," in *Proceedings of the Eleventh National Conference on Artificial Intelligence*, July 1993, pp. 133–8.