Declarative `letrec` and `define`

by Bruce Duba Matthias Felleisen

Rice University

Draft*: January 1991*

1 A Problem with `letrec`

Approximately two years ago, Alan Bawden demonstrated in a message[1] to the bulletin board *comp.lang.scheme* that Scheme's `letrec` is not a purely declarative facility. More precisely, removing `set!` and procedures with side-effects from Scheme does not result in a purely functional language with continuations. In particulaer, reference cells can be implemented via a combination of `letrec` and `call-with-current-continuation` [Bawden: as above]:

```
   (define (make-cell)
     (call-with-current-continuation
       (lambda (return-from-make-cell)
         (letrec ((state
                    (call-with-current-continuation
                      (lambda (return-new-state)
                        (return-from-make-cell
                          (lambda (op)
                            (case op
                              ((set)
                               (lambda (value)
                                 (call-with-current-
continuation
                                   (lambda (return-from-
access)
                                     (return-new-state
```

1. Message-ID: <19890302162742.4.ALAN@PIGPEN.AI.MIT.EDU>

```
                                               (list value return-
from-access))))))
                                  ((get) (car state))))))))
          ((cadr state) 'done)))))

   (define (set-cell cell value) ((cell 'set) value)) ;
added by authors

   (define (ref-cell cell) (cell 'get)) ; added by authors
```

The reason for this phenomenon is the use of `set!` for the definition of `letrec`'s semantics, which can be exposed by using continuation operations. The same is also true for `define` statements since they too are expanded into `set!`'s according to the Report.

2 Declarative Versions of `letrec` and `define`

We believe that `letrec` should not be an imperative construct and that a programmer should not be able to exploit it as one. To fix the problem we propose a change in the semantics of `letrec` and `define` such that they have a purely declarative character. The following subsections describe the changes that are necessary to make `letrec` and `define` fully declarative in the context of full Scheme.

The idea behind the fix is to consider the evaluation of definitional expressions in `define` and `letrec` expressions as separate programs. This solution still permits the use of continuation operations in `letrec` and `define`, but disables their capability to reveal the `set!` in their implementation. By being simple and minimally restrictive, we believe that our proposal is in the spirit of the rest of the Report.

2.1 New Paragraph on `letrec`, Subsection 4.2.2

(`letrec` ⟨bindings⟩ ⟨body⟩) essential syntax

Syntax: ⟨bindings⟩ should have the form

((⟨variable⟩ ⟨init⟩) ...),

and ⟨body⟩ should be a sequence of one or more expressions. It is an error for a ⟨variable⟩ to appear more than once in the list of variables being bound.

Semantics: The ⟨variable⟩s are bound to fresh locations holding undefined values, the ⟨init⟩s are evaluated in the resulting environment (in some unspecified order), each ⟨variable⟩ is assigned to the result of the corresponding ⟨init⟩, the ⟨body⟩ is evaluated in the resulting environment, and the value of the last expression in ⟨body⟩ is returned. Each binding of a ⟨variable⟩ has the entire `letrec` expression as its region, making it possible to define mutually recursive procedures.

```
(letrec ((even?
          (lambda (n)
            (if (zero? n)
                #f
                (odd? (- n 1)))))
         (odd?
          (lambda (n)
            (if (zero? n)
                #f
                (even? (- n 1))))))
  (even? 88))
                          => #t
```

There are two important restrictions on `letrec` expressions:

1. It must be possible to evaluate each ⟨init⟩ without assigning or referring to the value of any ⟨variable⟩. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name.
2. The evaluation of each ⟨init⟩ proceeds as if it were an isolated program by itself, only sharing the lexical environment (including the new bindings) with the rest of the program. This restriction is necessary to enforce that `letrec`-definitions are purely declarative.

In the most common uses of `letrec`, all the ⟨init⟩s are `lambda`-expressions and the restrictions are satisfied automatically.

2.2 New Version of Subsection 5.2.1

At the top level of a program, a definition

(`define` ⟨variable⟩ ⟨expression⟩)

has almost the same effect as the assignment expression

(`set!` ⟨variable⟩ ⟨expression⟩)

if ⟨variable⟩ is bound. If ⟨variable⟩ is not bound, however, then the definition will bind ⟨variable⟩ to a new location before performing the assignment, whereas it would be an error to perform a `set!` on an unbound unbound variable.

```
(define add3
  (lambda (x) (+ x 3)))
(add3 3)                              =>  6
(define first car)
(first '(1 2))                        =>  1
```

In analogy to `letrec` expressions, there are two important restrictions on definitions:

1. It must be possible to evaluate ⟨expression⟩ without assigning or referring to ⟨variable⟩. If this restriction is violated, then it is an error. The restriction is necessary because Scheme passes arguments by value rather than by name.
2. The evaluation of ⟨expression⟩ proceeds as if it were an isolated program by itself, only sharing the lexical environment (including the new bindings) with the rest of the program.

All Scheme implementations must support top level definitions.

Some implementations of Scheme use an initial environment in which all possible variables are bound to locations, most of which contain undefined values. Top level definitions in such an implementation are truly equivalent to assignments.

2.3 New Paragraph on `call-with-current-continuation`, Subsection 6.9

(`call-with-current-continuation` *proc*) essential procedure

Proc must be a procedure of one argument. The procedure `call-with-current-continuation` packages up the current continuation (see the rationale below) as an "escape procedure" and passes it as an argument to *proc*. The escape procedure is a Scheme procedure of one argument that, if it is later passed a value, will ignore whatever continuation is in effect at that later time and will give the value instead to the continuation that was in effect when the escape procedure was created.

The escape procedure that is passed to *proc* has unlimited extent just like any other procedure in Scheme. It may be stored in variables or data structures and may be called as many times as desired.

The following examples show only the most common uses of `call-with-current-continuation`. If all real programs were as simple as these examples, there would be no need for a procedure with the power of
`call-with-current-continuation`.

```
(call-with-current-continuation
  (lambda (exit)
    (for-each (lambda (x)
                (if (negative? x)
                    (exit x)))
              '(54 0 37 -3 245 19))
    #t))                          =>  -3

(define list-length
  (lambda (obj)
    (call-with-current-continuation
      (lambda (return)
        (letrec ((r
                   (lambda (obj)
                     (cond ((null? obj) 0)
                           ((pair? obj)
                            (+ (r (cdr obj)) 1))
                           (else (return #f))))))
          (r obj))))))

(list-length '(1 2 3 4))              =>  4

(list-length '(a b . c))              =>  #f
```

The construction and the use of escape procedures is subject to the restrictions for the evaluation of ⟨init⟩ expressions in `letrec` (see Subsection 4.2.2 above) and the evaluation of ⟨expression⟩ in definitions (see Subsection 5.2.1 above). In both cases, the evaluation of the respective expressions acts as if a *new* program were evaluated at the top-level in the current lexical environment. This is equivalent to the statement that the expression is evaluated in the initial continuation. It follows that the evaluation is forced to return a value, to diverge, or to result in *wrong*; and that escape procedures constructed during the evaluation of the expressions only represent the rest of the computation of the definitional expressions. For example,

```
   (letrec ((stop (call-with-current-continuation (lambda
(x) x)))) stop)
```

returns a procedure that terminates a program evaluation and returns its argument as the result of the program. The definiton

```
   (define stop (call-with-current-continuation (lambda (x)
x))
```

defines the same procedure.

Rationale for `call-with-current-continuation:`
A common use of `call-with-current-continuation` is for structured, non-local exits from loops or procedure bodies, but in fact `call-with-current-continuation` is extremely useful for implementing a wide variety of advanced control structures.

Whenever a Scheme expression is evaluated there is a continuation wanting the result of the expression. The continuation represents an entire (default) future for the computation. If the expression is evaluated at top level, for example, then the continuation will take the result and return it to the programming environment for further processing. Most of the time the continuation includes actions specified by user code, as in a continuation that will take the result, multiply it by the value stored in a local variable, add seven, and give the answer to the top level continuation. Normally these ubiquitous continuations are hidden behind the scenes and programmers don't think much about them. On rare occasions, however, a programmer may need to deal with continuations explicitly. `Call-with-current-continuation` allows Scheme programmers to do that by creating a procedure that acts just like the current continuation.

Most programming languages incorporate one or more special-purpose escape constructs with names like `exit`, `return`, or even `goto`. In 1965, however, Peter Landin [Landin65] invented a general purpose escape operator called the J-operator. John Reynolds [Reynolds72] described a simpler but equally powerful construct in 1972. The `catch` special form described by Sussman and Steele in the 1975 report on Scheme is exactly the same as Reynolds's construct, though its name came from a less general construct in MacLisp. Several Scheme implementors noticed that the full power of the `catch` construct could be provided by a procedure instead of by a special syntactic construct, and the name `call-with-current-continuation` was coined in 1982. This name is descriptive, but opinions differ on the merits of such a long name, and some people use the name `call/cc` instead.

2.4 Additions to the denotational semantics (Section 7)

Change Last paragraph of the Introduction to Section 7.2

If P is a program in which all variables are defined before being referenced or assigned, then the meaning of P is

$$E[\![\texttt{((lambda (}I^*\texttt{) }P'\texttt{)}\ \langle\text{undefined}\rangle\ \texttt{...)}]\!]$$

where I^* is the sequence of variables defined in P, P' is the sequence of expressions obtained by replacing every definition

$$\texttt{(define}\ \text{I Exp)}$$

in P by an assignment,

$$\texttt{(set!}\ \text{I}\ \texttt{(letrec}\ \text{((V Exp)) V))}$$

where V is an identifier that does not occur in Exp; $\langle$undefined$\rangle$ is an expression that evaluates to *undefined*, and E is the semantic function that assigns meaning to expressions.

The Domain of Expression Continuations

Add the following remark to the domain equation for `K`:
Expression continuations are *strict* in $\perp$ and values from `X` (errors).

Add at the end of Subsection 7.2.3

Answers, Expressed Values, and the Initial Continuation

We assume the existence of two functions relating expressed values, `E`, with answers, `A`. First, we assume that there is an initial continuation:

$$\in \ : \ \mathtt{E} \longrightarrow \mathtt{S} \longrightarrow \mathtt{A}$$

Second, we assume that the initial continuation has a left-inverse:

$$\mathrm{out}_A \ : \ \mathtt{A} \longrightarrow \mathtt{E} \times \mathtt{S}$$

such that

$$\mathrm{out}_A(\in \epsilon\, \sigma) = \langle \epsilon, \sigma \rangle.$$

Add to the semantic function *E* after last clause

$E[\![(\mathrm{letrec}\ ((\mathrm{I}_1\mathrm{E}_1) \cdots (\mathrm{I}_n\mathrm{E}_n))\mathrm{E})]\!] =$
$\lambda\, \rho\, \kappa.$
$tievals(\lambda\, \alpha^*\, \sigma.\underline{\mathrm{let}}\langle l_1, \ldots, l_n\rangle = \alpha^*\ \mathrm{min}$
⟨newdimen | ⟨p⟩⟩⟨setbox⟩4 =
$tievals(\lambda\, \alpha^*\, \sigma.$⟨p⟩=⟨wd⟩4⟨kern⟩⟨p⟩$\underline{\mathrm{let}}\,\rho = extends\,\rho\langle \mathrm{I}_1, \ldots,$
$\mathrm{I}_n\rangle\alpha\ \mathrm{min}$
⟨newdimen | ⟨p⟩⟩⟨setbox⟩4 =
$tievals(\lambda\alpha^*\sigma.$⟨p⟩=⟨wd⟩4⟨kern⟩⟨p⟩$\underline{\mathrm{let}}\langle\epsilon_1, \sigma_1\rangle = \mathrm{out}_A(E[\![\mathrm{E}_{i_1}]\!]\rho \in$
$\sigma)\ \mathrm{min}$
⟨newdimen|⟨p⟩⟩⟨setbox⟩4= $tievals(\lambda\alpha^*\sigma.$⟨p⟩=⟨wd⟩4⟨kern⟩⟨p⟩$\vdots$
⟨newdimen | ⟨p⟩⟩⟨setbox⟩4 =
$tievals(\lambda\ \alpha^*\ \sigma.$⟨p⟩=⟨wd⟩4⟨kern⟩⟨p⟩$\underline{\mathrm{let}}\langle\epsilon_{j+1},\ \sigma_{j+1}\rangle =$
$\mathrm{out}_A(E[\![\mathrm{E}_{i_{j+1}}]\!]\rho \in \sigma_j)\ \mathrm{min}$
⟨newdimen|⟨p⟩⟩⟨setbox⟩4= $tievals(\lambda\alpha^*\sigma.$⟨p⟩=⟨wd⟩4⟨kern⟩⟨p⟩$\vdots$
⟨newdimen | ⟨p⟩⟩⟨setbox⟩4 =
$tievals(\lambda\alpha^*\sigma.$⟨p⟩=⟨wd⟩4⟨kern⟩⟨p⟩ $E[\![\mathrm{E}]\!]\rho\kappa(updatel_{i_1}\epsilon_1 \ldots (updatel_{i_n}\epsilon_n\sigma_n)))$
⟨newdimen | ⟨p⟩⟩⟨setbox⟩5 =
$tievals$⟨p⟩=⟨wd⟩5⟨kern⟩⟨p⟩$\langle unspecified, \ldots,$
$unspecified\rangle$% n-times
where $i_1, \ldots, i_n$ is an arbitrary permutation of $1, \ldots, n$

3 Side-effects of our Changes

The new definitions eliminate the connection between the report and the programming environment of actual implementations, i.e., the report no longer mentions the top-level read-eval-print loop. We believe that this is desirable in face of the emerging stand-alone compilers that are independent of user interactions.

4 Implementation Effort

The implementation effort should be minimal. We have implemented the proposal in Chez Scheme. The fixes rely on non-RRRS features and are not portable but we guess that a similar implementation should be possible for other versions of Scheme.