Essentials of Programming Languages, Third Edition

by Daniel P. Friedman and Mitchell Wand

Chapter 4: State

4.1 Computational Effects

So far, we have only considered the *value* produced by a computation. But a computation may have *effects* as well: it may read, print, or alter the state of memory or a file system. In the real world, we are *always* interested in effects: if a computation doesn't display its answer, it doesn't do us any good!

What's the difference between producing a value and producing an effect? An effect is *global*: it is seen by the entire computation. An effect affects the entire computation (pun intended).

We will be concerned primarily with a single effect: assignment to a location in memory. How does assignment differ from binding? As we have seen, binding is local, but variable assignment is potentially global. It is about the *sharing* of values between otherwise unrelated portions of the computation. Two procedures can share information if they both know about the same location in memory. A single procedure can share information with a future invocation of itself by leaving the information in a known location.

We model memory as a finite map from *locations* to a set of values called the *storable values*. For historical reasons, we call this the *store*. The storable values in a language are typically, but not always, the same as the expressed values of the language. This choice is part of the design of a language.

A data structure that represents a location is called a *reference*. A location is a place in memory where a value can be stored, and a reference is a data structure that refers to that place. The distinction between locations and references may be seen by analogy: a location is like a file and a reference is like a URL. The URL refers to the file, and the file contains some data. Similarly, a reference denotes a location, and the location contains some data.

References are sometimes called *L-values*. This name reflects the association of such data structures with variables appearing on the left-hand side of assignment statements. Analogously, expressed values, such as the values of the right-hand side expressions of assignment statements, are known as *R-values*.

We consider two designs for a language with a store. We call these designs *explicit references* and *implicit references*.

4.2 EXPLICIT-REFS: A Language with Explicit References

In this design, we add references as a new kind of expressed value. So we have

$$\begin{aligned} ExpVal &= Int + Bool + Proc + Ref(ExpVal) \\ DenVal &= ExpVal \end{aligned}$$

Here *Ref*(*ExpVal*) means the set of references to locations that contain expressed values.

We leave the binding structures of the language unchanged, but we add three new operations to create and use references.

- newref, which allocates a new location and returns a reference to it.
- deref, which dereferences a reference: that is, it returns the contents of the location that the reference represents.
- setref, which changes the contents of the location that the reference represents.

We call the resulting language EXPLICIT-REFS. Let's write some programs using these constructs.

Below are two procedures, even and odd. They each take an argument, which they ignore, and return 1 or 0 depending on whether the contents of the location x is even or odd. They communicate not by passing data explicitly, but by changing the contents of the variable they share.

This program determines whether or not 13 is odd, and therefore returns 1. The procedures even and odd do not refer to their arguments; instead they look at the contents of the location to which x is bound.

```
let x = newref(0)
```

```
in letrec even(dummy)
           = if zero?(deref(x))
             then 1
             else begin
                    setref(x, -(deref(x),1));
                    (odd 888)
                  end
          odd(dummy)
           = if zero?(deref(x))
             then 0
             else begin
                    setref(x, -(deref(x),1));
                    (even 888)
                  end
   in begin setref(x,13); (odd 888) end
```

This program uses multideclaration letrec (exercise 3.32) and a begin expression (exercise 4.4). A begin expression evaluates its subexpressions in order and returns the value of the last one.

We pass a dummy argument to even and odd to stay within the framework of our unary language; if we had procedures of any number of arguments (exercise 3.21) we could have made these procedures of no arguments.

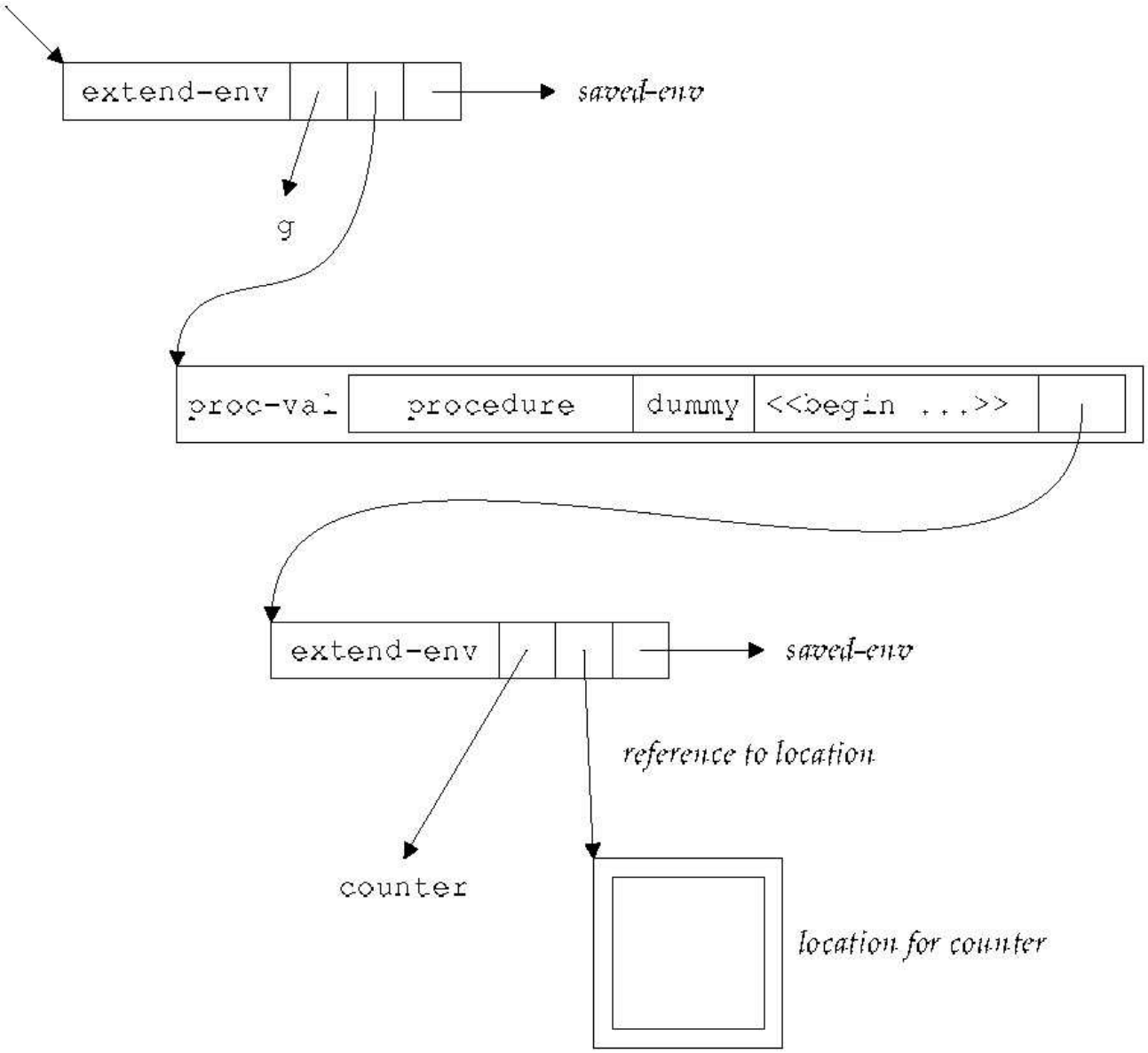
This style of communication is convenient when two procedures might share many quantities; one needs to assign only to the few quantities that change from one call to the next. Similarly, one procedure might call another procedure not directly but through a long chain of procedure calls. They could communicate data directly through a shared variable, without the intermediate procedures needing to know about it. Thus communication through a shared variable can be a kind of information hiding.

Another use of assignment is to create hidden state through the use of private variables. Here is an example.

```
let g = let counter = newref(0)
        in proc (dummy)
            begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Here the procedure g keeps a private variable that stores the number of times g has been called. Hence the first call to g returns 1, the second call to g returns 2, and the entire program has the value -1.

Here is a picture of the environment in which g is bound.



We can think of this as the different invocations of g sharing information with each other. This technique is used by the Scheme procedure gensym to create unique symbols.

Exercise 4.1 [*] What would have happened had the program been instead

```
let g = proc (dummy)
         let counter = newref(0)
         in begin
             setref(counter, -(deref(counter), -1));
             deref(counter)
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

In EXPLICIT-REFS, we can store any expressed value, and references are expressed values. This means we can store a reference in a location. Consider the program

```
let x = newref(newref(0))
in begin
    setref(deref(x), 11);
    deref(deref(x))
   end
```

This program allocates a new location containing 0. It then binds x to a location containing a reference to the first location. Hence the value of deref (x) is a reference to the first location. So when the program evaluates the setref, it is the first location

that is modified, and the entire program returns 11.

4.2.1 Store-Passing Specifications

In our language, any expression may have an effect. To specify these effects, we need to describe what store should be used for each evaluation and how each evaluation can modify the store.

In our specifications, we use σ to range over stores. We write $[l = v]\sigma$ to mean a store just like σ, except that location l is mapped to v. When we refer to a particular value of σ, we sometimes call it the *state* of the store.

We use *store-passing specfications*. In a store-passing specification, the store is passed as an explicit argument to value-of and is returned as an explicit result from value-of. Thus we write

$$\texttt{(value-of } exp_1\ \rho\ \sigma_0\texttt{)} = (val_1, \sigma_1)$$

This asserts that expression exp_1, evaluated in environment ρ and with the store in state σ_0, returns the value val_1 and leaves the store in a possibly different state σ_1.

Thus we can specify an effect-free operation like const-exp by writing

$$\texttt{(value-of (const-exp } n\texttt{)}\ \rho\ \sigma\texttt{)} = (n, \sigma)$$

showing that the store is unchanged by evaluation of this expression.

The specification for diff-exp shows how we specify sequential behavior.

$$\frac{\texttt{(value-of } exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1) \qquad \texttt{(value-of } exp_2\ \rho\ \sigma_1) = (val_2, \sigma_2)}{\texttt{(value-of (diff-exp } exp_1\ exp_2)\ \rho\ \sigma_0) = (\lceil \lfloor val_1 \rfloor - \lfloor val_2 \rfloor \rceil, \sigma_2)}$$

Here we evaluate exp_1 starting with the store in state σ_0. exp_1 returns value val_1, but it might also have some effects that leave the store in state σ_1. We then evaluate exp_2 starting with the store in the state that exp_1 left it, namely σ_1. exp_2 similarly returns a value val_2 and leaves the store in state σ_2. Then the entire expression returns $val_1 - val_2$ without further effect on the store, so it leaves the store in state σ_2.

Let's try a conditional.

$$\frac{\texttt{(value-of } exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{\texttt{(value-of (if-exp } exp_1\ exp_2\ exp_3)\ \rho\ \sigma_0) = \begin{cases} \texttt{(value-of } exp_2\ \rho\ \sigma_1) & \text{if } \texttt{(expval->bool } val_1) = \texttt{\#t} \\ \texttt{(value-of } exp_3\ \rho\ \sigma_1) & \text{if } \texttt{(expval->bool } val_1) = \texttt{\#f} \end{cases}}$$

Starting in state σ_0, an if-exp evaluates its test expression exp_1, returning the value val_1 and leaving the store in state σ_1. The result of the entire expression is then either the result of exp_2 or exp_3, each evaluated in the current environment ρ and in the state σ_1 in which exp_1 left the store.

Exercise 4.2 [*] Write down the specification for a zero?-exp.

Exercise 4.3 [*] Write down the specification for a call-exp.

Exercise 4.4 [**] Write down the specification for a begin expresssion.

Expression ::= begin *Expression* {; *Expression*}* end

A begin expression may contain one or more subexpressions separated by semicolons. These are evaluated in order and the value of the last is returned.

Exercise 4.5 [**] Write down the specification for list (exercise 3.10).

4.2.2 Specifying Operations on Explicit References

In EXPLICIT-REFS, we have three new operations that must be specified: newref, deref, and setref. These are given by the grammar

$$\textit{Expression} ::= \texttt{newref}\ (\textit{Expression})$$
$$\boxed{\texttt{newref-exp (exp1)}}$$

$$\textit{Expression} ::= \texttt{deref}\ (\textit{Expression})$$
$$\boxed{\texttt{deref-exp (exp1)}}$$

$$\textit{Expression} ::= \texttt{setref}\ (\textit{Expression}\ ,\ \textit{Expression})$$
$$\boxed{\texttt{setref-exp (exp1 exp2)}}$$

We can specify the behavior of these operations as follows.

$$\frac{(\texttt{value-of}\ \textit{exp}\ \rho\ \sigma_0) = (\textit{val}, \sigma_1) \qquad l \notin \mathrm{dom}(\sigma_1)}{(\texttt{value-of}\ (\texttt{newref-exp}\ \textit{exp})\ \rho\ \sigma_0) = ((\texttt{ref-val}\ l), [l{=}\textit{val}]\sigma_1)}$$

This rule says that newref-exp evaluates its operand. It extends the resulting store by allocating a new location *l* and puts the value *val* of its argument in that location. Then it returns a reference to a location *l* that is new. This means that it is not already in the domain of σ_1.

$$\frac{(\texttt{value-of}\ \textit{exp}\ \rho\ \sigma_0) = (l, \sigma_1)}{(\texttt{value-of}\ (\texttt{deref-exp}\ \textit{exp})\ \rho\ \sigma_0) = (\sigma_1(l), \sigma_1)}$$

This rule says that a deref-exp evaluates its operand, leaving the store in state σ_1. The value of that argument should be a reference to a location *l*. The deref-exp then returns the contents of *l* in σ_1, without any further change to the store.

$$\frac{(\texttt{value-of}\ \textit{exp}_1\ \rho\ \sigma_0) = (l, \sigma_1) \quad (\texttt{value-of}\ \textit{exp}_2\ \rho\ \sigma_1) = (\textit{val}, \sigma_2)}{(\texttt{value-of}\ (\texttt{setref-exp}\ \textit{exp}_1\ \textit{exp}_2)\ \rho\ \sigma_0) = (\lceil 23 \rceil, [l{=}\textit{val}]\sigma_2)}$$

This rule says that a setref-exp evaluates its operands from left to right. The value of the first operand must be a reference to a location *l*. The setref-exp then updates the resulting store by putting the value *val* of the second argument in location *l*. What should a setref-exp return? It could return anything. To emphasize the arbitrary nature of this choice, we have specified that it returns 23. Because we are not interested in the value returned by a setref-exp, we say that this expression is executed *for effect*, rather than for its value.

Exercise 4.6 [*] Modify the rule given above so that a setref-exp returns the value of the right-hand side.

Exercise 4.7 [*] Modify the rule given above so that a setref-exp returns the old contents of the location.

4.2.3 Implementation

The specification language we have used so far makes it easy to describe the desired behavior of effectful computations, but it does not embody a key fact about the store: a reference ultimately refers to a real location in a memory that exists in the real world. Since we have only one real world, our program can only keep track of one state σ of the store.

In our implementations, we take advantage of this fact by modeling the store using Scheme's own store. Thus we model an effect as a Scheme effect.

We represent the state of the store as a Scheme value, but we do not explicitly pass and return it, as the specification suggests. Instead, we keep the state in a single global variable, to which all the procedures of the implementation have access. This is much like even/odd example, where we used a shared location instead of passing an explicit argument. By using a single global variable, we also use as little as possible of our understanding of Scheme effects.

We still have to choose how to model the store as a Scheme value. We choose the simplest possible model: we represent the store as a list of expressed values, and a reference is a number that denotes a position in the list. A new reference is allocated by appending a new value to the list; and updating the store is modeled by copying over as much of the list as necessary. The code is shown in figures 4.1 and 4.2.

```
empty-store : () → Sto
(define empty-store
  (lambda () '()))

usage: A Scheme variable containing the current state
   of the store. Initially set to a dummy value.
(define the-store 'uninitialized)

get-store : () → Sto
(define get-store
  (lambda () the-store))

initialize-store! : () → Unspecified
usage: (initialize-store!) sets the-store to the empty store
(define initialize-store!
  (lambda ()
    (set! the-store (empty-store))))

reference? : SchemeVal → Bool
(define reference?
  (lambda (v)
    (integer? v)))

newref : ExpVal → Ref
(define newref
  (lambda (val)
    (let ((next-ref (length the-store)))
      (set! the-store (append the-store (list val)))
      next-ref)))

deref : Ref → ExpVal
(define deref
  (lambda (ref)
    (list-ref the-store ref)))
```

Figure 4.1: A naive model of the store

```
setref! : Ref × ExpVal → Unspecified
usage: sets the-store to a state like the original, but with
  position ref containing val.
(define setref!
  (lambda (ref val)
    (set! the-store
      (letrec
        ((setref-inner
           usage: returns a list like store1, except that
           position ref1 contains val.
           (lambda (store1 ref1)
             (cond
               ((null? store1)
                (report-invalid-reference ref the-store))
               ((zero? ref1)
```

```
                (cons val (cdr store1)))
               (else
                 (cons
                   (car store1)
                   (setref-inner
                     (cdr store1) (- ref1 1))))))))
        (setref-inner the-store ref)))))
```

Figure 4.2: A naive model of the store, continued

This representation is extremely inefficient. Ordinary memory operations require approximately constant time, but in our representation these operations require time proportional to the size of the store. No real implementation would ever do this, of course, but it suffices for our purposes.

We add a new variant, ref-val, to the data type for expressed values, and we modify value-of-program to initialize the store before each evaluation.

```
value-of-program : Program → ExpVal
(define value-of-program
  (lambda (pgm)
    (initialize-store!)
    (cases program pgm
      (a-program (exp1)
        (value-of exp1 (init-env))))))
```

Now we can write clauses in value-of for newref, deref, and setref. The clauses are shown in figure 4.3.

```
(newref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (ref-val (newref v1))))

(deref-exp (exp1)
  (let ((v1 (value-of exp1 env)))
    (let ((ref1 (expval->ref v1)))
      (deref ref1))))

(setref-exp (exp1 exp2)
  (let ((ref (expval->ref (value-of exp1 env))))
    (let ((val2 (value-of exp2 env)))
      (begin
        (setref! ref val2)
        (num-val 23)))))
```

Figure 4.3: `value-of` clauses for explicit-reference operators

We can instrument our system by adding some procedures that convert environments, procedures, and stores to a more readable form, and we can instrument our system by printing messages at key points in the code. We also use procedures that convert environments, procedures, and stores to a more readable form. The resulting logs give a detailed picture of our system in action. A typical example is shown in figures 4.4 and 4.5. This trace shows, among other things, that the arguments to the subtraction are evaluated from left to right.

```
> (run "
let x = newref(22)
in let f = proc (z) let zz = newref(-(z,deref(x)))
                    in deref(zz)
   in -((f 66), (f 55))")

entering let x
newref: allocating location 0
entering body of let x with env =
((x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering let f
entering body of let f with env =
```

```
((f
  (procedure
    z
    ...
    ((x #(struct:ref-val 0))
     (i #(struct:num-val 1))
     (v #(struct:num-val 5))
     (x #(struct:num-val 10)))))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))

entering body of proc z with env =
((z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)))
```

Figure 4.4: Trace of an evaluation in EXPLICIT-REFS.

```
entering let zz
newref: allocating location 1
entering body of let zz with env =
((zz #(struct:ref-val 1))
 (z #(struct:num-val 66))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

entering body of proc z with env =
((z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22)) (1 #(struct:num-val 44)))

entering let zz
newref: allocating location 2
entering body of let zz with env =
((zz #(struct:ref-val 2))
 (z #(struct:num-val 55))
 (x #(struct:ref-val 0))
 (i #(struct:num-val 1))
 (v #(struct:num-val 5))
 (x #(struct:num-val 10)))
store =
((0 #(struct:num-val 22))
 (1 #(struct:num-val 44))
 (2 #(struct:num-val 33)))

#(struct:num-val 11)
>
```

Figure 4.5: Trace of an evaluation in EXPLICIT-REFS, continued

Exercise 4.8 [*] Show exactly where in our implementation of the store these operations take linear time rather than constant time.

Exercise 4.9 [*] Implement the store in constant time by representing it as a Scheme vector. What is lost by using this representation?

Exercise 4.10 [*] Implement the begin expression as specified in exercise 4.4.

Exercise 4.11 [*] Implement list from exercise 4.5.

Exercise 4.12 [***] Our understanding of the store, as expressed in this interpreter, depends on the meaning of effects in Scheme. In particular, it depends on us knowing *when* these effects take place in a Scheme program. We can avoid this dependency by writing an interpreter that more closely mimics the specification. In this interpreter, value-of would return both a value and a store, just as in the specification. A fragment of this interpreter appears in figure 4.6. We call this a *store-passing interpreter*. Extend this interpreter to cover all of the language EXPLICIT-REFS.

```
(define-datatype answer answer?
  (an-answer
    (val expval?)
    (store store?)))

value-of : Exp × Env × Sto → ExpVal
(define value-of
  (lambda (exp env store)
    (cases expression exp
      (const-exp (num)
        (an-answer (num-val num) store))
      (var-exp (var)
        (an-answer
          (apply-store store (apply-env env var))
          store))
      (if-exp (exp1 exp2 exp3)
        (cases answer (value-of exp1 env store)
          (an-answer (val new-store)
            (if (expval->bool val)
              (value-of exp2 env new-store)
              (value-of exp3 env new-store)))))
      (deref-exp (exp1)
        (cases answer (value-of exp1 env store)
          (an-answer (v1 new-store)
            (let ((ref1 (expval->ref v1)))
              (an-answer (deref ref1) new-store)))))
      ...)))
```

Figure 4.6: Store-passing interpreter for exercise 4.12

Every procedure that might modify the store returns not just its usual value but also a new store. These are packaged in a data type called answer. Complete this definition of value-of.

Exercise 4.13 [***] Extend the interpreter of the preceding exercise to have procedures of multiple arguments.

4.3 IMPLICIT-REFS: A Language with Implicit References

The explicit reference design gives a clear account of allocation, dereferencing, and mutation because all these operations are explicit in the programmer's code.

Most programming languages take common patterns of allocation, dereferencing, and mutation, and package them up as part of the language. Then the programmer need not worry about when to perform these operations, because they are built into the language.

In this design, every variable denotes a reference. Denoted values are references to locations that contain expressed values. References are no longer expressed values. They exist only as the bindings of variables.

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$

Locations are created with each binding operation: at each procedure call, let, or letrec.

When a variable appears in an expression, we first look up the identifier in the environment to find the location to which it is bound, and then we look up in the store to find the value at that location. Hence we have a "two-level" system for var-exp.

The contents of a location can be changed by a set expression. We use the syntax

$$\textit{Expression} ::= \texttt{set}\ \textit{Identifier}\ =\ \textit{Expression}$$

```
assign-exp (var exp1)
```

Here the *Identifier* is not part of an expression, so it does not get dereferenced. In this design, we say that variables are *mutable*, meaning changeable.

This design is called *call-by-value*, or *implicit references*. Most programming languages, including Scheme, use some variation on this design.

Figure 4.7 has our two sample programs in this design. Because references are no longer expressed values, we can't make chains of references, as we did in the last example in section 4.2.

```
let x = 0
in letrec even(dummy)
           = if zero?(x)
             then 1
             else begin
                    set x = -(x,1);
                    (odd 888)
                  end
          odd(dummy)
           = if zero?(x)
             then 0
             else begin
                    set x = -(x,1);
                    (even 888)
                  end
   in begin set x = 13; (odd -888) end

let g = let count = 0
        in proc (dummy)
            begin
             set count = -(count,-1);
             count
            end
in let a = (g 11)
   in let b = (g 11)
      in -(a,b)
```

Figure 4.7: `odd` and `even` in IMPLICIT-REFS

4.3.1 Specification

We can write the rules for dereference and set easily. The environment now always binds variables to locations, so when a variable appears as an expression, we need to dereference it:

```
(value-of (var-exp var) ρ σ) = (σ(ρ(var)),σ)
```

Assignment works as one might expect: we look up the left-hand side in the environment, getting a location, we evaluate the right-hand side in the environment, and we modify the desired location. As with setref, the value returned by a set expression is arbitrary. We choose to have it return the expressed value 27.

$$\frac{(\texttt{value-of}\ exp_1\ \rho\ \sigma_0) = (val_1, \sigma_1)}{(\texttt{value-of}\ (\texttt{assign-exp}\ var\ exp_1)\ \rho\ \sigma_0) = (\lceil 27 \rceil, [\rho(var) = val_1]\sigma_1)}$$

We also need to rewrite the rules for procedure call and let to show the modified store. For procedure call, the rule becomes

```
(apply-procedure (procedure var body ρ) val σ)
= (value-of body [var = l]ρ [l = val]σ)
```

where l is a location not in the domain of σ.

The rule for (let-exp *var* exp_1 *body*) is similar. The right-hand side exp_1 is evaluated, and the value of the let expression is the value of the body, evaluated in an environment where the variable *var* is bound to a new location containing the value of exp_1.

Exercise 4.14 [*] Write the rule for let.

4.3.2 The Implementation

Now we are ready to modify the interpreter. In value-of, we dereference at each var-exp, just like the rules say

```
(var-exp (var) (deref (apply-env env var)))
```

and we write the obvious code for a assign-exp

```
(assign-exp (var exp1)
  (begin
    (setref!
      (apply-env env var)
      (value-of exp1 env))
    (num-val 27)))
```

What about creating references? New locations should be allocated at every new binding. There are exactly four places in the language where new bindings are created: in the initial environment, in a let, in a procedure call, and in a letrec.

In the initial environment, we explicitly allocate new locations.

For let, we change the corresponding line in value-of to allocate a new location containing the value, and to bind the variable to a reference to that location.

```
(let-exp (var exp1 body)
  (let ((val1 (value-of exp1 env)))
    (value-of body
      (extend-env var (newref val1) env))))
```

For a procedure call, we similarly change apply-procedure to call newref.

```
apply-procedure : Proc × ExpVal → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var (newref val) saved-env))))))
```

Last, to handle letrec, we replace the extend-env-rec clause in apply-env to return a reference to a location containing the appropriate closure. Since we are using multideclaration letrec (exercise 3.32), extend-env-rec takes a list of procedure names, a list of bound variables, a list of procedure bodies, and a saved environment. The procedure location takes a variable and a list of variables and returns either the position of the variable in the list, or #f if it is not present.

```
(extend-env-rec (p-names b-vars p-bodies saved-env)
  (let ((n (location search-var p-names)))
    (if n
      (newref
        (proc-val
          (procedure
            (list-ref b-vars n)
            (list-ref p-bodies n)
            env)))
      (apply-env saved-env search-var))))
```

Figure 4.8 shows a simple evaluation in IMPLICIT-REFS, using the same instrumentation as before.

```
> (run "
let f = proc (x) proc (y)
         begin
          set x = -(x,-1);
          -(x,y)
         end
in ((f 44) 33)")
```

```
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

newref: allocating location 4
entering body of proc x with env =
((x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44)))

newref: allocating location 5
entering body of proc y with env =
((y 5) (x 4) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 #(struct:num-val 44))
 (5 #(struct:num-val 33)))

#(struct:num-val 12)
>
```

Figure 4.8: Sample evaluation in IMPLICIT-REFS

Exercise 4.15 [*] In figure 4.8, why are variables in the environment bound to plain integers rather than expressed values, as in figure 4.5?

Exercise 4.16 [*] Now that variables are mutable, we can build recursive procedures by assignment. For example

```
letrec times4(x) = if zero?(x)
                   then 0
                   else -((times4 -(x,1)), -4)
in (times4 3)
```

can be replaced by

```
let times4 = 0
in begin
    set times4 = proc (x)
                  if zero?(x)
                  then 0
                  else -((times4 -(x,1)), -4);
    (times4 3)
   end
```

Trace this by hand and verify that this translation works.

Exercise 4.17 [**] Write the rules for and implement multiargument procedures and let expressions.

Exercise 4.18 [**] Write the rule for and implement multiprocedure letrec expressions.

Exercise 4.19 [**] Modify the implementation of multiprocedure letrec so that each closure is built only once, and only one location is allocated for it. This is like exercise 3.35.

Exercise 4.20 [**] In the language of this section, all variables are mutable, as they are in Scheme. Another alternative is to allow both mutable and immutable variable bindings:

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) + ExpVal \end{aligned}$$

Variable assignment should work only when the variable to be assigned to has a mutable binding. Dereferencing occurs implicitly when the denoted value is a reference.

Modify the language of this section so that let introduces immutable variables, as before, but mutable variables are introduced by a letmutable expression, with syntax given by

```
Expression ::= letmutable Identifier = Expression in Expression
```

Exercise 4.21 [**] We suggested earlier the use of assignment to make a program more modular by allowing one procedure to communicate information to a distant procedure without requiring intermediate procedures to be aware of it. Very often such an assignment should only be temporary, lasting for the execution of a procedure call. Add to the language a facility for *dynamic assignment* (also called *fluid binding*) to accomplish this. Use the production

$$Expression ::= \texttt{setdynamic}\ Identifier\ \texttt{=}\ Expression\ \texttt{during}\ Expression$$
$$\boxed{\texttt{setdynamic-exp}\ (var\ exp_1\ body)}$$

The effect of the setdynamic expression is to assign temporarily the value of exp_1 to *var*, evaluate *body*, reassign *var* to its original value, and return the value of *body*. The variable *var* must already be bound. For example, in

```
let x = 11
in let p = proc (y) -(y,x)
   in -(setdynamic x = 17 during (p 22),
        (p 13))
```

the value of x, which is free in procedure p, is 17 in the call (p 22), but is reset to 11 in the call (p 13), so the value of the expression is 5 – 2 = 3.

Exercise 4.22 [**] So far our languages have been expression-oriented: the primary syntactic category of interest has been expressions and we have primarily been interested in their values. Extend the language to model the simple statement-oriented language whose specification is sketched below. Be sure to *Follow the Grammar* by writing separate procedures to handle programs, statements, and expressions.

Values As in IMPLICIT-REFS.

Syntax Use the following syntax:

```
Program   ::= Statement
Statement ::= Identifier = Expression
              ::= print Expression
              ::= {{Statement}*(;) }
              ::= if Expression Statement Statement
              ::= while Expression Statement
              ::= var {Identifier}*(,) ; Statement
```

The nonterminal *Expression* refers to the language of expressions of IMPLICIT-REFS, perhaps with some extensions.

Semantics A program is a statement. A statement does not return a value, but acts by modifying the store and by printing.

Assignment statements work in the usual way. A print statement evaluates its actual parameter and prints the result. The if statement works in the usual way. A block statement, defined in the last production for *Statement*, binds each of the declared variables to an uninitialized reference and then executes the body of the block. The scope of these bindings is the body.

Write the specification for statements using assertions like

```
(result-of stmt ρ σ0) = σ1
```

Examples Here are some examples.

```
(run "var x,y; {x = 3; y = 4; print +(x,y)}") % Example 1
7
(run "var x,y,z; {x = 3;                      % Example 2
                  y=4;
                  z=0;
                  while not(zero?(x))
                    {z = +(z,y); x = -(x,1)};
                  print z}")
12
(run " var x; {x = 3;                         % Example 3
               print x;
               var x; {x = 4; print x};
               print x}")
3
4
3
(run "var f,x; {f = proc(x,y) *(x,y);         % Example 4
                x=3;
                print (f 4 x)}")
12
```

Example 3 illustrates the scoping of the block statement.

Example 4 illustrates the interaction between statements and expressions. A procedure value is created and stored in the variable f. In the last line, this procedure is applied to the actual parameters 4 and x; since x is bound to a reference, it is dereferenced to obtain 3.

Exercise 4.23 [*] Add to the language of exercise 4.22 read statements of the form read *var*. This statement reads a nonnegative integer from the input and stores it in the given variable.

Exercise 4.24 [*] A do-while statement is like a while statement, except that the test is performed *after* the execution of the body. Add do-while statements to the language of exercise 4.22.

Exercise 4.25 [*] Extend the block statement of the language of exercise 4.22 to allow variables to be initialized. In your solution, does the scope of a variable include the initializer for variables declared later in the same block statement?

Exercise 4.26 [***] Extend the solution to the preceding exercise so that procedures declared in a single block are mutually recursive. Consider restricting the language so that the variable declarations in a block are followed by the procedure declarations.

Exercise 4.27 [**] Extend the language of the preceding exercise to include *subroutines*. In our usage a subroutine is like a procedure, except that it does not return a value and its body is a statement, rather than an expression. Also, add subroutine calls as a new kind of statement and extend the syntax of blocks so that they may be used to declare both procedures and subroutines. How does this affect the denoted and expressed values? What happens if a procedure is referenced in a subroutine call, or vice versa?

4.4 MUTABLE-PAIRS: A Language with Mutable Pairs

In exercise 3.9 we added lists to our language, but these were immutable: there was nothing like Scheme's set-car! or set-cdr! for them.

Now, let's add mutable pairs to IMPLICIT-REFS. Pairs will be expressed values, and will have the following operations:

```
newpair   : Expval × Expval → MutPair
left          : MutPair →Expval
right         : MutPair →Expval
setleft       : MutPair ×Expval → Unspecified
setright      : MutPair ×Expval → Unspecified
```

A pair consists of two locations, each of which is independently assignable. This gives us the domain equations:

$$
\begin{aligned}
ExpVal &= Int + Bool + Proc + MutPair \\
DenVal &= Ref(ExpVal) \\
MutPair &= Ref(ExpVal) \times Ref(ExpVal)
\end{aligned}
$$

We call this language MUTABLE-PAIRS.

4.4.1 Implementation

We can implement this literally using the reference data type from our preceding examples. The code is shown in figure 4.9.

```
(define-datatype mutpair mutpair?
  (a-pair
    (left-loc reference?)
    (right-loc reference?)))

make-pair : ExpVal × ExpVal → MutPair
(define make-pair
  (lambda (val1 val2)
    (a-pair
      (newref val1)
      (newref val2))))

left : MutPair → ExpVal
(define left
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref left-loc)))))

right : MutPair → ExpVal
(define right
  (lambda (p)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (deref right-loc)))))

setleft : MutPair × ExpVal → Unspecified
(define setleft
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! left-loc val)))))

setright : MutPair × ExpVal → Unspecified
(define setright
  (lambda (p val)
    (cases mutpair p
      (a-pair (left-loc right-loc)
        (setref! right-loc val)))))
```

Figure 4.9: Naive implementation of mutable pairs

Once we've done this, it is straightforward to add these to the language. We add a mutpair-val variant to our data type of expressed values, and five new lines to value-of. These are shown in figure 4.10. We arbitrarily choose to make setleft return 82 and setright return 83. The trace of an example, using the same instrumentation as before, is shown in figures 4.11 and 4.12.

```
(newpair-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (mutpair-val (make-pair val1 val2))))

(left-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
```

```
      (left p1))))

(right-exp (exp1)
  (let ((val1 (value-of exp1 env)))
    (let ((p1 (expval->mutpair val1)))
      (right p1))))

(setleft-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setleft p val2)
        (num-val 82)))))

(setright-exp (exp1 exp2)
  (let ((val1 (value-of exp1 env))
        (val2 (value-of exp2 env)))
    (let ((p (expval->mutpair val1)))
      (begin
        (setright p val2)
        (num-val 83)))))
```

Figure 4.10: Integrating mutable pairs into the interpreter

```
> (run "let glo = pair(11,22)
in let f = proc (loc)
            let d1 = setright(loc, left(loc))
            in let d2 = setleft(glo, 99)
            in -(left(loc),right(loc))
in (f glo)")
;; allocating cells for init-env
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let glo
;; allocating cells for the pair
newref: allocating location 3
newref: allocating location 4
;; allocating cell for glo
newref: allocating location 5
entering body of let glo with env =
((glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4))))

entering let f
;; allocating cell for f
newref: allocating location 6
entering body of let f with env =
((f 6) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 #(struct:num-val 11))
 (4 #(struct:num-val 22))
 (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
 (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2)))))
```

Figure 4.11: Trace of evaluation in MUTABLE-PAIRS

```
;; allocating cell for loc
newref: allocating location 7
entering body of proc loc with env =
((loc 7) (glo 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
```

```
  (2 #(struct:num-val 10))
  (3 #(struct:num-val 11))
  (4 #(struct:num-val 22))
  (5 #(struct:mutpair-val #(struct:a-pair 3 4)))
  (6 (procedure loc ... ((glo 5) (i 0) (v 1) (x 2))))
  (7 #(struct:mutpair-val #(struct:a-pair 3 4))))

#(struct:num-val 88)
>
```

Figure 4.12: Trace of evaluation in MUTABLE-PAIRS, cont'd

4.4.2 Another Representation of Mutable Pairs

The representation of a mutable pair as two references does not take advantage of all we know about *MutPair*. The two locations in a pair are independently assignable, but they are not independently allocated. We know that they will be allocated together: if the left part of a pair is one location, then the right part is in the next location. So we can instead represent the pair by a reference to its left. The code for this is shown in figure 4.13. Nothing else need change.

```
mutpair? : SchemeVal → Bool
(define mutpair?
  (lambda (v)
    (reference? v)))

make-pair : ExpVal × ExpVal → MutPair
(define make-pair
  (lambda (val1 val2)
    (let ((ref1 (newref val1)))
      (let ((ref2 (newref val2)))
        ref1))))

left : MutPair → ExpVal
(define left
  (lambda (p)
    (deref p)))

right : MutPair → ExpVal
(define right
  (lambda (p)
    (deref (+ 1 p))))

setleft : MutPair × ExpVal → Unspecified
(define setleft
  (lambda (p val)
    (setref! p val)))

setright : MutPair × ExpVal → Unspecified
(define setright
  (lambda (p val)
    (setref! (+ 1 p) val)))
```

Figure 4.13: Alternate representation of mutable pairs

Similarly, one could represent any aggregate object in the heap by a pointer to its first location. However, a pointer does not by itself identify an area of memory unless it is supplemented by information about the length of the area (see exercise 4.30). The lack of length information is a source of classic security errors, such as out-of-bounds array writes.

Exercise 4.28 [**] Write down the specification rules for the five mutable-pair operations.

Exercise 4.29 [**] Add arrays to this language. Introduce new operators newarray, arrayref, and arrayset that create, dereference, and update arrays. This leads to

$$ArrVal = (Ref(ExpVal))^*$$
$$ExpVal = Int + Bool + Proc + ArrVal$$
$$DenVal = Ref(ExpVal)$$

Since the locations in an array are consecutive, use a representation like the second representation above. What should be the result of the following program?

```
let a = newarray(2,-99)
    p = proc (x)
         let v = arrayref(x,1)
         in arrayset(x,1,-(v,-1))
in begin arrayset(a,1,0); (p a); (p a); arrayref(a,1) end
```

Here newarray(2,-99) is intended to build an array of size 2, with each location in the array containing -99. begin expressions are defined in exercise 4.4. Make the array indices zero-based, so an array of size 2 has indices 0 and 1.

Exercise 4.30 [**] Add to the language of exercise 4.29 a procedure arraylength, which returns the size of an array. Your procedure should work in constant time. Make sure that arrayref and arrayset check to make sure that their indices are within the length of the array.

4.5 Parameter-Passing Variations

When a procedure body is executed, its formal parameter is bound to a denoted value. Where does that value come from? It must be passed from the actual parameter in the procedure call. We have already seen two ways in which a parameter can be passed:

- Natural parameter passing, in which the denoted value is the same as the expressed value of the actual parameter (page 75).
- Call-by-value, in which the denoted value is a reference to a location containing the expressed value of the actual parameter (section 4.3). This is the most commonly used form of parameter-passing.

In this section, we explore some alternative parameter-passing mechanisms.

4.5.1 CALL-BY-REFERENCE

Consider the following expression:

```
let p = proc (x) set x = 4
in let a = 3
   in begin (p a); a end
```

Under call-by-value, the denoted value associated with x is a reference that initially contains the same value as the reference associated with a, but these references are distinct. Thus the assignment to x has no effect on the contents of a's reference, so the value of the entire expression is 3.

With call-by-value, when a procedure assigns a new value to one of its parameters, this cannot possibly be seen by its caller. Of course, if the parameter passed to the caller contains a mutable pair, as in section 4.4, then the effect of setleft or setright will be visible to a caller. But the effect of a set is not.

Though this isolation between the caller and callee is generally desirable, there are times when it is valuable to allow a procedure to be passed locations with the expectation that they will be assigned by the procedure. This may be accomplished by passing the procedure a reference to the location of the caller's variable, rather than the contents of the variable. This parameter-passing mechanism is called *call-by-reference*. If an operand is simply a variable reference, a reference to the variable's location is passed. The formal parameter of the procedure is then bound to this location. If the operand is some other kind of expression, then the formal parameter is bound to a new location containing the value of the operand, just as in call-by-value. Using call-by-reference in the above example, the assignment of 4 to x has the effect of assigning 4 to a, so the entire expression would return 4, not 3.

When a call-by-reference procedure is called and the actual parameter is a variable, what is passed is the *location* of that variable, rather than the contents of that location, as in call-by-value. For example, consider

```
let f = proc (x) set x = 44
in let g = proc (y) (f y)
   in let z = 55
      in begin (g z); z end
```

When the procedure g is called, y is bound to the location of z, not the contents of that location. Similarly, when f is called, x becomes bound to that same location. So x, y, and z will all be bound to the same location, and the effect of the set x = 44 is to set that location to 44. Hence the value of the entire expression is 44. A trace of the execution of this expression is shown in figures 4.14 and 4.15; in this example, x, y, and z all wind up bound to location 5.

```
> (run "
let f = proc (x) set x = 44
in let g = proc (y) (f y)
in let z = 55
in begin
    (g z);
    z
   end")
newref: allocating location 0
newref: allocating location 1
newref: allocating location 2
entering let f
newref: allocating location 3
entering body of let f with env =
((f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))

entering let g
newref: allocating location 4
entering body of let g with env =
((g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2)))))

entering let z
newref: allocating location 5
entering body of let z with env =
((z 5) (g 4) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))
```

Figure 4.14: Sample evaluation in CALL-BY-REFERENCE

```
entering body of proc y with env =
((y 5) (f 3) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2))))
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

entering body of proc x with env =
((x 5) (i 0) (v 1) (x 2))
store =
((0 #(struct:num-val 1))
 (1 #(struct:num-val 5))
 (2 #(struct:num-val 10))
 (3 (procedure x ... ((i 0) (v 1) (x 2)))))
```

```
 (4 (procedure y ... ((f 3) (i 0) (v 1) (x 2))))
 (5 #(struct:num-val 55)))

#(struct:num-val 44)
>
```

Figure 4.15: Sample evaluation in CALL-BY-REFERENCE, cont'd

A typical use of call-by-reference is to return multiple values. A procedure can return one value in the normal way and assign others to parameters that are passed by reference. For another sort of example, consider the problem of swapping the values in two variables:

```
let swap = proc (x) proc (y)
            let temp = x
            in begin
                set x = y;
                set y = temp
               end
in let a = 33
   in let b = 44
      in begin
          ((swap a) b);
          -(a,b)
         end
```

Under call-by-reference, this swaps the values of a and b, so it returns 11. If this program were run with our existing call-by-value interpreter, however, it would return -11, because the assignments inside the swap procedure then have no effect on variables a and b.

Under call-by-reference, variables still denote references to expressed values, just as they did under call-by-value:

$$\begin{aligned} ExpVal &= Int + Bool + Proc \\ DenVal &= Ref(ExpVal) \end{aligned}$$

The only thing that changes is the allocation of new locations. Under call-by-value, a new location is created for every evaluation of an operand; under call-by-reference, a new location is created for every evaluation of an operand *other than a variable*.

This is easy to implement. The function apply-procedure must change, because it is no longer true that a new location is allocated for every procedure call. That responsibility must be moved upstream, to the call-exp line in value-of, which will have the information to make that decision.

```
apply-procedure : Proc × Ref → ExpVal
(define apply-procedure
  (lambda (proc1 val)
    (cases proc proc1
      (procedure (var body saved-env)
        (value-of body
          (extend-env var val saved-env))))))
```

We then modify the call-exp line in value-of, and introduce a new function value-of-operand that makes the necessary decision.

```
(call-exp (rator rand)
  (let ((proc (expval->proc (value-of rator env)))
        (arg (value-of-operand rand env)))
    (apply-procedure proc arg)))
```

The procedure value-of-operand checks to see if the operand is a variable. If it is, then the reference that the variable denotes is returned and then passed to the procedure by apply-procedure. Otherwise, the operand is evaluated, and a reference to a new location containing that value is returned.

```
value-of-operand : Exp × Env → Ref
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
```

```
      (else
        (newref (value-of exp env))))))
```

We could modify let to work in a similar fashion, but we have chosen not to do so, so that some call-by-value functionality will remain in the language.

More than one call-by-reference parameter may refer to the same location, as in the following program.

```
let b = 3
in let p = proc (x) proc(y)
            begin
             set x = 4;
             y
            end
   in ((p b) b)
```

This yields 4 since both x and y refer to the same location, which is the binding of b. This phenomenon is known as *variable aliasing*. Here x and y are aliases (names) for the same location. Generally, we do not expect an assignment to one variable to change the value of another, so aliasing makes it very difficult to understand programs.

Exercise 4.31 [*] Write out the specification rules for CALL-BY-REFERENCE.

Exercise 4.32 [*] Extend the language CALL-BY-REFERENCE to have procedures of multiple arguments.

Exercise 4.33 [**] Extend the language CALL-BY-REFERENCE to support call-by-value procedures as well.

Exercise 4.34 [*] Add a call-by-reference version of let, called letref, to the language. Write the specification and implement it.

Exercise 4.35 [**] We can get some of the benefits of call-by-reference without leaving the call-by-value framework. Extend the language IMPLICIT-REFS by adding a new expression

Expression ::= `ref` *Identifier*

`ref-exp (var)`

This differs from the language EXPLICIT-REFS, since references are only of variables. This allows us to write familiar programs such as swap within our call-by-value language. What should be the value of this expression?

```
let a = 3
in let b = 4
   in let swap = proc (x) proc (y)
                  let temp = deref(x)
                  in begin
                      setref(x,deref(y));
                      setref(y,temp)
                     end
      in begin ((swap ref a) ref b); -(a,b) end
```

Here we have used a version of let with multiple declarations (exercise 3.16). What are the expressed and denoted values of this language?

Exercise 4.36 [*] Most languages support arrays, in which case array references are generally treated like variable references under call-by-reference. If an operand is an array reference, then the location referred to, rather than its contents, is passed to the called procedure. This allows, for example, a swap procedure to be used in commonly occurring situations in which the values in two array elements are to be exchanged. Add array operators like those of exercise 4.29 to the call-by-reference language of this section, and extend value-of-operand to handle this case, so that, for example, a procedure application like

```
((swap (arrayref a i)) (arrayref a j))
```

will work as expected. What should happen in the case of

```
((swap (arrayref a (arrayref a i))) (arrayref a j)) ?
```

Exercise 4.37 [**] *Call-by-value-result* is a variation on call-by-reference. In call-by-value-result, the actual parameter must be a variable. When a parameter is passed, the formal parameter is bound to a new reference initialized to the value of the actual parameter, just as in call-by-value. The procedure body is then executed normally. When the procedure body returns, however,

the value in the new reference is copied back into the reference denoted by the actual parameter. This may be more efficient than call-by-reference because it can improve memory locality. Implement call-by-value-result and write a program that produces different answers using call-by-value-result and call-by-reference.

4.5.2 Lazy Evaluation: CALL-BY-NAME and CALL-BY-NEED

All the parameter-passing mechanisms we have discussed so far are *eager*: they always find a value for each operand. We now turn to a very different form of parameter passing, called *lazy evaluation*. Under lazy evaluation, an operand in a procedure call is not evaluated until it is needed by the procedure body. If the body never refers to the parameter, then there is no need to evaluate it.

This can potentially avoid non-termination. For example, consider

```
letrec infinite-loop (x) = infinite-loop(-(x,-1))
in let f = proc (z) 11
   in (f (infinite-loop 0))
```

Here infinite-loop is a procedure that, when called, never terminates. f is a procedure that, when called, never refers to its argument and always returns 11. Under any of the mechanisms considered so far, this program will fail to terminate. Under lazy evaluation, however, this program will return 11, because the operand (infinite-loop 1) is never evaluated.

We now modify our language to use lazy evaluation. Under lazy evaluation, we do not evaluate an operand expression until it is needed. Therefore we associate the bound variable of a procedure with an unevaluated operand. When the procedure body needs the value of its bound variable, the associated operand is evaluated. We sometimes say that the operand is *frozen* when it is passed unevaluated to the procedure, and that it is *thawed* when the procedure evaluates it.

Of course we will also have to include the environment in which that procedure is to be evaluated. To do this, we introduce a new data type of *thunks*. A thunk consists of an expression and an environment.

```
(define-datatype thunk thunk?
  (a-thunk
    (exp1 expression?)
    (env environment?)))
```

When a procedure needs to use the value of its bound variable, it will evaluate the associated thunk.

Our situation is somewhat more complicated, because we need to accommodate both lazy evaluation, effects, and eager evaluation (for let). We therefore let our denoted values be references to locations containing either expressed values or thunks.

$$\begin{aligned} DenVal &= Ref(ExpVal + Thunk) \\ ExpVal &= Int + Bool + Proc \end{aligned}$$

Our policy for allocating new locations will be similar to the one we used for call-by-reference: If the operand is a variable, then we pass its denotation, which is a reference. Otherwise, we pass a reference to a new location containing a thunk for the unevaluated argument.

```
value-of-operand : Exp × Env → Ref
(define value-of-operand
  (lambda (exp env)
    (cases expression exp
      (var-exp (var) (apply-env env var))
      (else
        (newref (a-thunk exp env))))))
```

When we evaluate a var-exp, we first find the location to which the variable is bound. If the location contains an expressed value, then that value is returned as the value of the var-exp. If it instead contains a thunk, then the thunk is evaluated, and that value is returned. This design is called *call by name*.

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
```

```
        (value-of-thunk w)))))
```

The procedure value-of-thunk is defined as

```
value-of-thunk : Thunk → ExpVal
(define value-of-thunk
  (lambda (th)
    (cases thunk th
      (a-thunk (exp1 saved-env)
        (value-of exp1 saved-env)))))
```

Alternatively, once we find the value of the thunk, we can install that expressed value in the same location, so that the thunk will not be evaluated again. This arrangement is called *call by need*.

```
(var-exp (var)
  (let ((ref1 (apply-env env var)))
    (let ((w (deref ref1)))
      (if (expval? w)
        w
        (let ((val1 (value-of-thunk w)))
          (begin
            (setref! ref1 val1)
            val1))))))
```

This is an instance of a general strategy called *memoization*.

An attraction of lazy evaluation in all its forms is that in the absence of effects, it supports reasoning about programs in a particularly simple way. The effect of a procedure call can be modeled by replacing the call with the body of the procedure, with every reference to a formal parameter in the body replaced by the corresponding operand. This evaluation strategy is the basis for the lambda calculus, where it is called *β-reduction*.

Unfortunately, call-by-name and call-by-need make it difficult to determine the order of evaluation, which in turn is essential to understanding a program with effects. If there are no effects, though, this is not a problem. Thus lazy evaluation is popular in functional programming languages (those with no effects), and rarely found elsewhere.

Exercise 4.38 [*] The example below shows a variation of exercise 3.25 that works under call-by-need. Does the original program in exercise 3.25 work under call-by-need? What happens if the program below is run under call-by-value? Why?

```
let makerec = proc (f)
               let d = proc (x) (f (x x))
               in (f (d d))
in let maketimes4 = proc (f)
                     proc (x)
                      if zero?(x)
                      then 0
                      else -((f -(x,1)), -4)
   in let times4 = (makerec maketimes4)
      in (times4 3)
```

Exercise 4.39 [*] In the absence of effects, call-by-name and call-by-need always give the same answer. Construct an example in which call-by-name and call-by-need give different answers.

Exercise 4.40 [*] Modify value-of-operand so that it avoids making thunks for constants and procedures.

Exercise 4.41 [**] Write out the specification rules for call-by-name and call-by-need.

Exercise 4.42 [**] Add a lazy let to the call-by-need interpreter.