Chapter 8

State

Man's yesterday may ne'er be like his morrow;
Nought may endure but Mutability

— *Mutability, st. 4, Percy Bysshe Shelley*

I woke up one morning and looked around the room. Something wasn't right. I realized that someone had broken in the night before and replaced everything in my apartment with an exact replica. I couldn't believe it...I got my roommate and showed him. I said, "Look at this — everything's been replaced with an exact replica!" He said, "Do I know you?"

— *Steven Wright*

8.1 What is State?

8.1.1 Time, State, Identity, and Change

We naturally view the world around us in terms of objects. Each object is characterized by a set of attributes that can vary with time. The **state** of an object is the set of particular attributes it has at a given point in time. For example, the state of a box of chocolates includes its size, shape, color, location, whether its lid is on or off, and the number, types, and positions of the chocolates inside.

Every object has a unique, time-independent attribute that distinguishes it from other objects: its **identity**. The notion of identity is at the very heart of

objectness, for it formalizes the intuition that objects exist over extents of time rather than just at instants of time. Identity allows us to say that an object at one point in time is the "same" as that at another point, regardless of any changes of state that may have taken place in between. It also gives us a way of saying that two objects with otherwise indistinguishable states are "different."

Consider our box of chocolates again. If we open the lid, the state of the box has changed, but we still consider it to be the same box of chocolates. Even after we eat all the goodies inside, we think that the box has become empty, not that we have a different box of chocolates.

On the other hand, suppose we leave an unopened box of chocolates on the kitchen table one day and find an unopened box there the next day. We are likely to assume that it's the same box. However, a housemate might later confess to consuming the entire original box in a fit of the munchies, but then buying a replacement box after feeling pangs of guilt. In light of this confession, we concede that the box on the table is not the same as the one we bought, even though, from our perspective, its state is indistinguishable from that of the box we left there the day before.

How could we monitor similar situations in the future without the help of explicit confessions? Before placing an unopened box of chocolates on the table we could alter the box in some irreversible way. The next day we could check if the box on the table had the same alteration. If the box on the table the next day does not exhibit the alteration, we are sure that the new box is not the same as the original. If it does have the alteration, we aren't 100% sure (our housemate might have diabolically copied our alteration, or a new box by chance might exhibit the same alteration), but there is reasonable evidence that the box is in fact the same one we left the previous day.

This example emphasizes that the notions of time, state, identity, and change are all inextricably intertwined.[1] The purpose of this chapter is to see how these notions are expressed in a computational framework. We shall see that state and its friends provide new ways to decompose problems but can greatly complicate reasoning about programs.

8.1.2 FL Does Not Support State

Computing with time-varying state-based entities is an extremely popular programming paradigm, both in traditional **imperative languages**, such as FORTRAN, COBOL, PASCAL, C, and ADAas well as in **object-oriented languages** like SMALLTALK, C++, C#, and JAVA. We shall call such languages **stateful**.

[1] For a further discussion of this philosophical point in a computational framework, see Chapter 3 of [ASS96].

One reason that stateful languages are so popular is that they resonate with the experience that many programmers have in interacting with objects that change over time in the world. At the opposite side of the spectrum are **stateless** languages like the so-called **purely functional** programming languages such as HASKELL and MIRANDA. **Mostly functional** languages are those, like COMMON LISP, SCHEME, and ML, that add stateful features on top of a stateless function-oriented core.

The FL language we have studied thus far is a stateless language – it provides no support for expressing computational objects with identity and state. In particular, neither variables nor data structures (pairs) may exhibit time-dependent behavior. To underscore this point, we will show the difficulties encountered in modeling a classic example of state – bank accounts – within FL. The goal is to implement the following bank account procedures in FL:

- `(make-account` *amount*`)`: Creates an account with *amount* as the initial balance.
- `(balance` *account*`)`: Returns the balance in *account*.
- `(deposit!` *amount account*`)`: If *amount* is non-negative, increases the balance of *account* by *amount* and returns the symbol `succeeded`. If *amount* is negative, leaves the balance unchanged and returns the symbol `failed`.
- `(withdraw!` *amount account*`)`: If *amount* is less than or equal to the balance of *account*, decreases the balance of *account* by *amount*, and returns the symbol `succeeded`. If *amount* is negative or is greater than the balance of *account*, leaves the balance unchanged and returns the symbol `failed`.

We adopt the convention that names of procedures that change the state of an object (such as `deposit!` and `withdraw!`) end in the '`!`' character (pronounced "bang").

Note that the specifications of `deposit!` and `withdraw!` indicate not only what **value** the procedures return (in both cases, one of the symbols `succeeded` or `failed`) but also what **effect** the procedure has on the state of the account (increasing or decreasing the balance). Even `make-account` has the effect of updating the banking system to include a new account. Such changes in state are referred to as **side effects** or **mutations**. In programming languages supporting state, the specification of a procedure includes both its return value and its side effects.[2]

[2]This is true for languages like SCHEME and C in which procedure calls are *expressions* —

It turns out that it is impossible to write a set of FL procedures that satisfy the above specifications. We will demonstrate this fact by studying a nullary (i.e., zero-argument) procedure `test-deposit!` that performs the following steps in order:

- create an account *acct* with a balance of 100;
- determine the balance *bal* of *acct*;
- deposit 17 dollars into *acct*;
- determine the new balance *bal′* of *acct*;
- return the difference *bal′* − *bal*.

In a stateful language, `(test-deposit!)` should return 17. However, we can show that in FL `test-deposit!` must return 0!

If we try to write `test-deposit!` in FL, we immediately run into a stumbling block. The specified actions are clearly ordered by time, but FL provides no explicit construct for specifying that expressions should be evaluated in any particular order. To get around this problem, we assume the existence of a construct `(begin` E_1 E_2`)` that evaluates E_1 before E_2. Since all FL expressions must return a value, we dictate that the value returned by a `begin` expression is the value of E_2. The formal semantics of `begin` are specified by the operational rewrite rules and the denotational valuation clause in Figure 8.1.

Operational Semantics

$$\texttt{(begin}\ V\ E\texttt{)} \Rightarrow E \qquad [\textit{begin-return}]$$

$$\frac{E_1 \Rightarrow E_1'}{\texttt{(begin}\ E_1\ E_2\texttt{)} \Rightarrow \texttt{(begin}\ E_1'\ E_2\texttt{)}} \qquad [\textit{begin-progress}]$$

Denotational Semantics

$$\mathcal{E}[\![\texttt{(begin}\ E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,(\textit{with-value}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda v\,.\,(\mathcal{E}[\![E_2]\!]\ e)))$$

Figure 8.1: Operational and denotational semantics of `begin`.

Using `begin`, we can write `test-deposit!` in FL as follows:

constructs that appear in value-accepting contexts. But in many languages, procedure calls are *commands* — constructs that do not produce values but are executed for effect only. In such languages, procedure specifications do not describe a return value.

```
(define test-deposit!
  (lambda ()
    (let ((acct (make-account 100)))
      (let ((old (balance acct)))
        (begin (deposit! 17 acct)
               (- (balance acct) old))))))
```

The abstraction can be translated into FLK as follows:

```
(proc ignore
  (call (proc acct
          (call (proc old
                  (begin (call (call deposit! 17) acct)
                         (primop - (call balance acct) old)))
                (call balance acct)))
        (call make-account 100)))
```

We can now use our semantics frameworks to show that the FLK call

```
(call test-deposit! #u)
```

must evaluate to 0 regardless of how `deposit!` is defined. We will assume a CBN version of FLK; since termination is not an issue here, the result will be the same for CBV.

An operational trace of `(call test-deposit! #u)` appears in Figure 8.2. Note the three copies of the expression `(call make-account 100)` generated by substitution. In FL's operational semantics, an expression representing a data structure for all intents and purposes *is* the data structure. Since the second operand of `deposit!` and the operand of the two calls of `balance` are syntactically distinct copies of the `make-account` expression, any operations performed by `deposit!` can't possibly affect the operands of the `balance` calls. If we make the assumption that

$$\texttt{(call balance (call make-account 100))} \stackrel{*}{\Rightarrow} \texttt{100}$$

(this would seem to be required of any reasonable bank account implementation), then the trace shows that `(test-deposit!)` indeed evaluates to 0.

Denotational semantics offers another perspective on this example. Recall that FL's valuation function $\mathcal{E}$ maps expressions and environments to expressible values. If an environment $e_{context}$ is specified, then a given expression must always denote the same meaning relative to $e_{context}$ because $\mathcal{E}$ is a mathematical function. Suppose that e_1 is an environment in which `acct` is bound to a representation of an account with a balance of b dollars (b need not be 100). Then the following must be true:

$$(\mathcal{E}[\![\texttt{(call balance acct)}]\!]\ e_1) = b$$

Now consider the meaning of the following expression E_{core} with respect to e_1:

```
(call (proc ignore
        (call (proc acct
                (call (proc old
                        (begin (call (call deposit! 17) acct)
                               (primop - (call balance acct) old)))
                      (call balance acct)))
              (call make-account 100)))
      #u)
⇒ (call (proc acct
            (call (proc old
                    (begin (call (call deposit! 17) acct)
                           (primop - (call balance acct) old)))
                  (call balance acct)))
          (call make-account 100))
⇒ (call (proc old
            (begin (call (call deposit! 17) (call make-account 100))
                   (primop - (call balance (call make-account 100))
                             old)))
          (call balance (call make-account 100)))
⇒ (begin (call (call deposit! 17) (call make-account 100))
           (primop -
                   (call balance (call make-account 100))
                   (call balance (call make-account 100))))
⇒* (primop -
            (call balance (call make-account 100))
            (call balance (call make-account 100)))
⇒* (primop - 100 100)
⇒ 0
```

Figure 8.2: Operational trace showing that `(call test-deposit! #u)` evaluates to 0.

```
E_core = (call (proc old
                  (begin (call (call deposit! 17) acct)
                         (primop - (call balance acct) old)))
               (call balance acct))
```

This expression contains two occurrences of `(call balance acct)` that are evaluated in environments containing the same binding for `acct`. So both of these occurrences denote the same number b. But then the meaning of `old` is clearly b, so the meaning of

```
(primop - (call balance acct) old)
```

must be 0. Thus, we have shown that E_{core} denotes 0, regardless of which account is denoted by `acct`. So `(test-deposit!)` must also denote 0.

In both the operational and denotational analyses, the fundamental insight is that `(test-deposit!)` returns the difference of two occurrences of the expression `(call balance acct)`, and these must necessarily have the same value. A language in which distinct occurrences of any expression always have the same meaning within a given naming context is said to be **referentially transparent**. Of course, the notion of "naming context" needs to be fully specified. Intuitively, two occurrences of an expression are in the same naming context if they share the same Stoy diagram — i.e., if every occurrence of a free identifier in one refers to the same binding occurrence as the corresponding identifier in the other. Stateless languages, such as our mini-language FL and the real language HASKELL, are referentially transparent, while stateful languages are not.

Referential transparency is a property that we frequently use in mathematical reasoning in the form of "substituting equals for equals." But it is seriously at odds with the notions of state and time. State is predicated on the idea that observable properties of an object can change. But if we make the reasonable assumption that a property of an object can be accessed by applying a single-argument procedure to that object (as in `(balance acct)` above), referential transparency dictates that all occurrences of such an expression within a given environment must denote the same value. Thus, the observable properties of an object cannot change. And if changes to the state of objects cannot be observed, how meaningful is it to talk about one action happening before or after another? We shall have more to say about referential transparency and state in Section 8.2.5.

Finally, suppose we actually try to write the definition of `deposit!` in FL. What kind of difficulties do we run into? Below is a skeleton for such a procedure:

```
(define deposit!
  (lambda (amount account)
    (if (< amount 0)
        'failed
        (begin E_IncreaseBalance
               'succeeded))))
```

The body of `deposit!` returns the right value (one of the symbols `failed` or `succeeded`). But how do we write $E_{IncreaseBalance}$? By the same reasoning used above, no FL expression can possibly alter the state of the account. Obviously, we are missing something. Shortly, we will introduce constructs that allow us to fill in the blanks here, and we will explore how the semantics of FL needs to be changed to accommodate their introduction.

But before we do that, consider the following. Since FL is a universal language, it is capable of expressing *any* computation. So surely examples such as the bank account scenario must be expressible within FL, albeit not necessarily in a way that corresponds to our intuitions about the physical world. Next, we'll examine some ways in which state can be simulated in FL. The purpose of this exploration is to give us insight into the nature of state. Later, we will be able to apply what we learn to the semantics for our modified dialect of FL.

8.1.3 Simulating State In FL

8.1.3.1 Iteration

The simulation of state in FL is exemplified by the handling of iteration. An **iteration** is a computation that characterizes the state of a system in terms of the values of a set of variables known as its **state variables**. The value of each state variable in an iteration at time t is a function of the values of the state variables at time $t - 1$.

As an example of an iteration, consider the problem of reversing the order of cards in a deck of playing cards. A natural solution is to use two piles, called *old* and *new*, where *old* is initially the original deck and *new* is an empty pile. Then, one by one, cards can be moved from the old pile to the new pile until the old pile is empty. At this point, the new pile contains the reversed deck of cards. In this example the state variables are the (ordered) contents of the old and new piles. These two variables completely characterize the state of the system. If a person performing the reversal for some reason had to leave before completing the task, someone else could take over as long as it was apparent which was the old pile and which was the new.

It is straightforward to express iterations in FL. For example, the above

technique can be applied to list reversal as follows:

```
(define reverse
  (lambda (lst)
    (letrec ((iterate
              (lambda (old-pile new-pile)
                (if (null? old-pile)
                    new-pile
                    (iterate (cdr old-pile)
                             (cons (car old-pile) new-pile)))))))
      (iterate lst '()))))
```

In this case, the state variables are the arguments `old-pile` and `new-pile` to the internal procedure `iterate`. For example, here is a trace of the reversal of a three-element list (where *REVERSE* and *ITERATE* stand for the appropriate expressions):

```
(REVERSE (list 1 2 3))
⇒* (ITERATE (list 1 2 3) (list)       )
⇒* (ITERATE (list 2 3)   (list 1)     )
⇒* (ITERATE (list 3)     (list 2 1)   )
⇒* (ITERATE (list)       (list 3 2 1) )
⇒* (list 3 2 1)
```

The above example suggests a general approach for expressing iterations in FL. State variables simply become the arguments to an iterating procedure, and updating the state variables is expressed by calling the iterating procedure on values computed from the previous values of the state variables.

Note carefully how an iteration manages to circumvent the constraints of referential transparency to represent state and time. The state at any point in time is represented by the values of formal parameter names associated with a particular application of the iterating procedure. In the list reversal example, the state variables correspond to the formal parameters `old-pile` and `new-pile`. The value of a particular variable named `old-pile` or `new-pile` never changes. However, each application of the `iterate` procedure effectively creates new variables that happen to be named by these same identifiers. So for each point in time t, there are distinct variables $\texttt{old-pile}_t$ and $\texttt{new-pile}_t$. State is encoded not as the changing value of a variable, but rather as the values of a sequence of immutable variables.

Events in time are ordered by the only means available for ordering in a stateless language: data dependency. If the value of E_1 is needed to compute E_2, then E_2 is said to have a **data dependency** on E_1. In the list reversal example, since $\texttt{old-pile}_t$ is equal to $\texttt{(cdr old-pile}_{t-1}\texttt{)}$, it has a data dependency on

`old-pile`$_{t-1}$; `new-pile`$_t$ is dependent on both `old-pile`$_{t-1}$ and `new-pile`$_{t-1}$. Data dependencies can be interpreted as a kind of time: if E_2 depends on the result of E_1, it is natural to view the evaluation of E_1 as happening *before* the evaluation of E_2.

8.1.3.2 Single-Threaded Data Flow

Iteration is an instance of a general technique for simulating state in a stateless language. State can always be simulated by adding state variables both as arguments and return values to every procedure in a program whose body either accesses or changes the state variables. The state of the program upon entering a procedure is encoded in the values of the state variable arguments, and the state of the program upon exiting a procedure is encoded in the values of the state variables returned as results. Because state is based on a notion of linearly-ordered time, we must guarantee that the data dependencies among the state variables form a linear chain. State variables satisfying this constraint are said to be passed through the program in a **single-threaded** fashion.

From this perspective, the problem with the bank account procedures is that the state of the system is not appropriately threaded through calls to these procedures. Suppose the state of the banking system is modeled by an entity called a *bank-state*. Then we can simulate state with the bank account procedures by extending each procedure to accept an additional bank-state argument and to return a pair of its usual return value and a (potentially updated) bank-state.

Suppose that every bank account bears a unique *account number*. Then we can represent a bank-state as a list of account-number/current-balance pairs. For example, the bank-state $[\langle 1729, 200\rangle, \langle 6821, 17\rangle]$ indicates that account 1729 has a current balance of 200 dollars and account 6821 has a current balance of 17 dollars. We will allow the same account number to appear more than once in a bank-state; in this case, the leftmost pair with a given account number indicates the current balance of that account. For example, in bank-state $[\langle 6821, 52\rangle, \langle 1729, 200\rangle, \langle 6821, 17\rangle]$, account 6821 has 52 dollars.

Here is an implementation of the `deposit!` procedure in this approach:

```
(define deposit!
  (lambda (amount account bank-state)
    (if (< amount 0)
        (pair 'failed bank-state)
        (let ((old&bank1 (balance account bank-state)))
          (let ((old (left old&bank1))
                (bank1 (right old&bank1)))
            (pair 'succeeded
                  (cons (pair account (+ old amount)) bank1)))))))
```

We assume that accounts are represented by their account numbers and that `balance` has been similarly modified to accept and return a bank-state. When it succeeds, `deposit!` creates a new bank state by prepending a new account-number/current-balance pair to the old one. A bank-state can be threaded through `make-account`[3], `balance`, and `withdraw!` in a similar fashion.

The `test-deposit!` procedure can also be modified to take a bank-state and thread it through each of the bank account operations:

```
(define test-deposit!
  (lambda (bank)
    (let ((acct&bank1 (make-account 100 bank)))
      (let ((acct (left acct&bank1))
            (bank1 (right acct&bank1)))
        (let ((old&bank2 (balance acct bank1)))
          (let ((old (left old&bank2))
                (bank2 (right old&bank2)))
            (let ((sym&bank3 (deposit! 17 acct bank2)))
              (let ((sym (left sym&bank3))
                    (bank3 (right sym&bank3)))
                (let ((new&bank4 (balance acct bank3)))
                  (let ((new (left new&bank4))
                        (bank4 (right new&bank4)))
                    (pair (- new old)
                          bank4)))))))))))
```

Given any initial bank-state, the new version of `test-deposit!` will return a pair of 17 (the desired result) and an updated bank-state.

▷ **Exercise 8.1** Provide definitions of `make-account`, `balance`, and `withdraw!` in which a bank-state is single-threaded through each procedure. ◁

▷ **Exercise 8.2** It is only necessary to single-thread a store through procedures that may update the store. For procedures that only access the store without updating it, it is sufficient to pass the store as an argument; such a procedure need not return a store as its result. An example of such a procedure is `balance`, which reads the balance of a bank account but does not write it.

- Write a version of `balance` that takes an account and a bank-state and returns only the balance of the account.
- Modify the state-simulating definitions of `deposit!` and `test-deposit!` to use

[3] `make-account` must also create a new, previously unused account number. Asking the caller to specify the number is an option, but it is better to include the next available account number as part of the bank state. If we don't care about wasting computational resources, we can compute a fresh account number from the current bank-state representation by adding `1` to the largest account number in the bank.

the new version of `balance`. ◁

8.1.3.3 Monadic Style

The bank-state threading details make the `test-deposit!` code hard to read, but some well-chosen abstractions can significantly increase readability. It helps to have a `with-pair` procedure that decomposes a pair into its component parts and passes these to a receiver procedure that names them:

```
(define with-pair
  (lambda (pair receiver)
    (receiver (left pair) (right pair))))
```

Using `with-pair`, `test-deposit!` can be simplified as follows:

```
(define test-deposit!
  (lambda (bank)
    (with-pair (make-account 100 bank)
      (lambda (acct bank1)
        (with-pair (balance acct bank1)
          (lambda (old bank2)
            (with-pair (deposit! 17 acct bank2)
              (lambda (sym bank3)
                (with-pair (balance acct bank3)
                  (lambda (new bank4)
                    (pair (- new old) bank4))))))))))))
```

Readability can be increased even further by hiding the threading of the bank-state altogether. Suppose that we define an **action** as any procedure that takes a bank-state and returns a pair of a value and a bank-state. In order to **perform** an action, we apply the action to a bank-state, which returns a value/bank-state pair. Such actions can be glued together by the `after` procedure in Figure 8.3, which takes a first action and a procedure that maps the value from performing the first action to a second action and returns a single action that performs the first action followed by the second. The figure also contains a `return` procedure that converts a value into an action and curried versions of `make-account`, `balance`, and `deposit!` that return actions when supplied with their non-bank-state arguments. With these abstractions, the `test-deposit!` procedure can be composed using four occurrences of `after` and one `return` (Figure 8.4).

This final version of `test-deposit!` illustrates a technique for threading state through a program that is known as **monadic style**. This style is based on gluing together state-threading components like the bank account actions in

```
(define after
  (lambda (action receiver)
    (lambda (bank)
      (with-pair (action bank)
        (lambda (val bank1)
          ((receiver val) bank1))))))

(define return
  (lambda (val)
    (lambda (bank) (pair val bank))))

(define *make-account
  (lambda (amount)
    (lambda (bank) (make-account amount bank))))

(define *deposit!
  (lambda (amount acct)
    (lambda (bank) (deposit! amount acct bank))))

(define *balance
  (lambda (acct)
    (lambda (bank) (balance acct bank))))
```

Figure 8.3: Procedures supporting a monadic style of threading bank-states through a program.

```
(define test-deposit!
  (after (*make-account 100)
    (lambda (acct)
      (after (*balance acct)
        (lambda (old)
          (after (*deposit! 17 acct)
            (lambda (sym)
              (after (*balance acct)
                (lambda (new)
                  (return (- new old))))))))))))
```

Figure 8.4: A version of `test-deposit!` written in monadic style.

a way that hides the details of the "plumbing." We have already seen monadic style in the denotational semantics of FL in Section 6.5. There, the *Computation* domain and functions like *with-value* are used to hide the messy details of propagating errors. In Section 8.2.4, we will extend the *Computation* domain to include a threaded store. By changing the meanings of a few functions like *with-value*, it is possible to thread the state through the semantics without changing many of the existing valuation functions. This illustrates the power of the monadic style.

In stateless languages, monadic style is commonly used to express stateful computations. The awkwardness of using a combiner like `after` can be avoided by syntactic sugar. For example, HASKELL supports a "`do` notation" in which the bank account testing function can be written as:

```
testDeposit =
  do a <- makeAccount 100
     b1 <- balance a
     deposit 17 a
     b2 <- balance a
     return (b2-b1)
```

As we shall see, this notation is not far from the way that stateful computations are expressed in stateful languages.

The name "monadic style" is derived from an algebraic structure, the **monad**, that captures the essence of manipulating state-threading components. For more information on monads and how monadic style can be used to express stateful computations in stateless languages like HASKELL, see [Wad95] and [JW93].

8.1.4 Imperative Programming

The bank account example demonstrates how it is possible to simulate state within a stateless language. However, even in monadic style, such simulations can be cumbersome. An alternative strategy is to develop a language paradigm that abstracts over the notion of state in such a way that the details of single-threading are automatically managed by the language. This is the essence of the **imperative programming paradigm**. In the imperative paradigm, all program state is conceptually bundled into a single entity called a **store** that is implicitly single-threaded through the program execution. Elements of the store are addressed by **locations**, unique identifiers that serve as unchanging names for time-dependent values. In the bank account example, bank-states correspond to stores and account numbers correspond to locations.

The advantage of the imperative programming paradigm is that programs can be shorter and more modular when the details of single-threading are im-

plicitly handled by the language. However, implicit single-threading has a down side: making explicit state variables implicit destroys referential transparency and thus makes programs harder to reason about.

The rest of this chapter explores how to model languages that exhibit state. We will see that the notions of store, location, and single-threading crop up in both operational and denotational descriptions of stateful languages.

8.2 Mutable Data: FL!

8.2.1 Mutable Cells

A one-slot cons is called a cell,
A two-slot cons makes pairs as well.
But I would bet a coin of bronze
There isn't any three-slot cons.

— *Guy L. Steele, Jr.*

Data structures whose components can change over time are said to be **mutable**. The simplest kind of mutable data is the **mutable cell**, a data structure characterized by a single time-dependent value called its **content**. A mutable cell corresponds to a one-slot cons cell in SCHEME or a pointer variable in languages like C and PASCAL. We will study mutable data in the context of FL!, a version of FL that supports mutable cells. We will use CBV as the default evaluation strategy for FL! because, as we shall see later, it makes more sense than CBN in languages that support mutation.

We begin by extending CBV FLK with features for supporting mutable cells. The modified kernel, FLK!, has the following syntax:

$$
\begin{array}{lll}
E_{FLK!} ::= \ldots & & \text{[FLK expressions]} \\
\quad \mid \texttt{(cell } E_{content}\texttt{)} & & \text{[Cell]} \\
\quad \mid \texttt{(begin } E_{sequent1}\ E_{sequent2}\texttt{)} & & \text{[Begin]} \\
O \in \text{Primop}_{FLK!} & = & \text{Primop}_{FLK} \cup \{\texttt{cell-ref}, \texttt{cell-set!}, \texttt{cell=?}, \texttt{cell?}\}
\end{array}
$$

Here is an informal description of the extensions:

- `(cell` E`)` returns a new mutable cell whose initial content is the value of E. We shall write cells as *id*:*val*, where *id* is a number that uniquely identifies the cell, and *val* is the content of the cell. In the following example, the expression allocates a cell with *id* number 1729 and content 3:

  `(cell (+ 1 2))` $\xrightarrow[FLK!]{}$ *1729*:*3*

- `(primop cell-ref` E`)` fetches the content of the cell computed by E. If the value of E is not a cell, this expression yields an error.

  `(primop cell-ref (cell (+ 1 2)))` $\xrightarrow[FLK!]{}$ *3*
  `(primop cell-ref (+ 1 2))` $\xrightarrow[FLK!]{}$ *error:not-a-cell*

- `(primop cell-set!` E_1 E_2`)` stores the value of E_2 in the cell computed by E_1. If the value of E_1 is not a cell, this expression yields an error. Since every FLK! expression must return a value, we shall arbitrarily specify that the value returned by a cell assignment expression is the unit value.

  `(primop cell-set! (cell (+ 1 2)) 4)` $\xrightarrow[FLK!]{}$ *unit*

- `(primop cell=?` E_1 E_2`)` returns *true* if E_1 and E_2 evaluate to the same cell and *false*if they evaluate to different cells. If at least one of E_1 or E_2 is not a cell, the expression yields an error.

```
(let ((c1 (cell 1))
      (c2 (cell 1)))
  (let ((c3 c1))
    (list (primop cell=? c1 c1)
          (primop cell=? c1 c2)
          (primop cell=? c1 c3))))
```
$\xrightarrow[FLK!]{}$ [*true*, *false*, *true*]

- `(primop cell?` E`)` returns *true* if E evaluates to a cell and *false*if it evaluates to some other value.

  `(pair (primop cell? 0) (primop cell? (cell 0)))` $\xrightarrow[FLK!]{}$ $\langle$*false*, *true*$\rangle$

- `(begin` E_1 E_2`)` first evaluates E_1, then evaluates E_2, and then returns the value of E_2. The value of E_1 is discarded.

```
(call (proc c
        (begin (primop cell-set! c 4)
               (primop cell-ref c)))
      (cell (+ 1 2)))
```
$\xrightarrow[FLK!]{}$ *4*

FL! is built on top of the kernel provided by FLK!. It has the same syntax as FL except that it includes the `cell` construct and a version of `begin` that can have an arbitrary number of sequents.

$E_{FL!}$ ::=	...	[FL expressions]
	| `(cell` E`)`	[Cell]
	| `(begin` $E_{sequent}$`*)`	[Begin]

The extended `begin` construct is defined by the following desugarings:

$\mathcal{D}_{\text{exp}}[\![\texttt{(begin)}]\!] = \texttt{\#u}$

$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E\texttt{)}]\!] = \mathcal{D}_{\text{exp}}[\![E]\!]$

$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E_1\ E_2\ E_{rest}\texttt{*)}]\!] =$
$\quad\texttt{(begin } \mathcal{D}_{\text{exp}}[\![E_1]\!]\ \mathcal{D}_{\text{exp}}[\![\texttt{(begin } E_2\ E_{rest}\texttt{*)}]\!]\texttt{)}$

This is the first time we've seen a sugar construct that has the same phrase tag as a kernel construct. This situation is common in practice. Of course, the desugaring for such a construct must guarantee that the general sugar form rewrites to the more restricted kernel form.

Like other primitive operator names, `cell-ref` and `cell-set!` are standard identifiers in FL!, where, respectively, they stand for procedures that access the content of a cell and change the content of a cell. Because these names are verbose, we will also introduce shorter synonyms in the standard environment: the name `^` is a synonym for `cell-ref`, and `:=` is a synonym for `cell-set!`.

8.2.2 Examples of Imperative Programming

The imperative programming paradigm is characterized by the use of side effects to perform computations. Because it is equipped with mutable cells, FL! supports the imperative paradigm. In this section, we present a few FL! programs that illustrate the imperative programming style.

8.2.2.1 Factorial

Here is an imperative version of an iterative factorial procedure written in FL!:

```
(define factorial
  (lambda (n)
    (let ((num (cell n))
          (ans (cell 1)))
      (letrec ((loop
                (lambda ()
                  (if (= (cell-ref num) 0)
                      (cell-ref ans)
                      (begin
                        (cell-set! ans (* (cell-ref num)
                                          (cell-ref ans)))
                        (cell-set! num (- (cell-ref num) 1))
                        (loop))))))
        (loop)))))
```

`num` and `ans` are cells that serve as the state variables of the iteration. The nullary `loop` procedure corresponds to a `while` loop in traditional imperative languages. On each round through the loop, the contents of the state variables are updated appropriately. The loop terminates when the content of `num` becomes zero.

It is instructive to compare the imperative version to a purely functional version:

```
(define factorial
  (lambda (n)
    (letrec ((loop (lambda (num ans)
                     (if (= num 0)
                         ans
                         (loop (- num 1) (* num ans))))))
      (loop n 1))))
```

In the functional version, every call to `loop` creates a new pair of variables named `num` and `ans`. In contrast, the imperative version shares one `num` and one `ans` variable across all the calls to `loop`. The correctness of the imperative version depends crucially on the order of the assignment expressions `(cell-set! ans ...)` and `(cell-set! num ...)`. If these expressions are swapped, then the imperative `factorial` no longer computes the right answer. This bug is due purely to the time-based nature of the imperative paradigm; the functional version does not exhibit the potential for this bug since all expressions have time-independent values. This illustrates one of the dangers of imperative programming: since many dependencies are implicit rather than explicit, subtle bugs are more likely, and they are harder to locate.

8.2.2.2 Bank Accounts

Using mutable cells, it is straightforward to implement the bank account scenario introduced in Section 8.1.2 and examined further in Section 8.1.3.

```
(define make-account
  (lambda (amount)
    (if (< amount 0)
        'failed
        (cell amount))))

(define balance
  (lambda (account) (cell-ref account)))
```

```
(define deposit!
  (lambda (amount account)
    (if (< amount 0)
        'failed
        (begin
          (cell-set! account (+ amount (cell-ref account)))
          'succeeded))))

(define withdraw!
  (lambda (amount account)
    (let ((bal (cell-ref account)))
      (if (or (< amount 0) (> amount bal))
          'failed
          (begin
            (cell-set! account (- bal amount))
            'succeeded)))))
```

Each account is represented by a distinct cell, and the bank account operations examine and change the content of this cell. Figure 8.5 shows the transcript of an interpreter session testing bank account objects.

```
(define a (make-account 100))
(define b (make-account 100))

(balance a) ⟶FL! 100
(balance b) ⟶FL! 100

(deposit! 17 b) ⟶FL! 'succeeded

(balance a) ⟶FL! 100
(balance b) ⟶FL! 117

(deposit! 23 a) ⟶FL! 'succeeded
(deposit! -23 b) ⟶FL! 'failed
(withdraw! 120 a) ⟶FL! 'succeeded
(withdraw! 120 b) ⟶FL! 'failed


(balance a) ⟶FL! 3
(balance b) ⟶FL! 117
```

Figure 8.5: Sample interactions with bank account objects.

While it is natural to represent accounts directly as cells, it is also somewhat insecure to do so. Every account should maintain the invariant that the balance

never slips below zero. But if an account is just a cell, then it is possible to violate this invariant by using `cell-set!` to directly store a negative number into an account. In general, it is wise to package up mutable data in a way that guarantees that important invariants cannot be violated (either accidentally or maliciously) by some other part of a software system.

First-class procedures provide an elegant means of encapsulating state so that it can only be manipulated in constrained ways. Figure 8.6 presents an alternate implementation in which bank accounts are represented as procedures that dispatch a message. The advantage to this approach is that the procedure provides a security wall for accessing and updating the account balance. In particular, the alternate implementation guarantees that the balance can never fall below zero.

8.2.2.3 Pattern Matching Revisited

The pattern matcher presented in Section 6.2.4.3 passes a dictionary through the computation in a single-threaded fashion. This means that the time-based sequence of dictionary values can alternately be represented as the changing content of a mutable cell. Figure 8.7 presents an imperative version of the `match-sexp` procedure that is based on this idea. (Procedures not defined in the figure are assumed to be the same as before.)

The `match-with-dict` procedure from Section 6.2.4.3 has been replaced by the internal `match!` procedure. Rather than taking a dictionary as its third argument, `match!` implicitly takes the current value of `dict-cell` as its argument. The `failed-flag` cell is used to simplify the handling of unsuccessful pattern matches.

8.2.3 An Operational Semantics for FLK!

In order to model the state exhibited by FLK!, we will use the notions of a **location** and a **store** introduced above. A location is a unique identifier for a mutable entity, and a store is a structure that associates each location with its value at a particular point in time. There are many ways to represent locations and stores. In our operational treatment, we will represent locations as numeric literals and stores as a sequence of location/value pairs:

$$\begin{array}{lllll} L & \in & \text{Location} & = & \mathit{Nat} \\ S & \in & \text{Store} & = & \text{Assignment}^* \\ Z & \in & \text{Assignment} & = & \text{Location} \times \text{ValueExp} \end{array}$$

We will assume the existence of a partial function *get* that finds the first value associated with a location in a store:

```
(define make-account
  (lambda (amount)
    (if (< amount 0)
        'failed
        (let ((account (cell amount)))
          (lambda (message)
            (cond ((sym=? message 'balance)
                   (cell-ref account))
                  ((sym=? message 'deposit!)
                   (lambda (amount)
                     (if (< amount 0)
                         'failed
                         (begin
                           (cell-set! account
                                      (+ amount (cell-ref account)))
                           'succeeded))))
                  ((sym=? message 'withdraw!)
                   (lambda (amount)
                     (let ((bal (cell-ref account)))
                       (if (or (< amount 0) (> amount bal))
                           'failed
                           (begin
                             (cell-set! account (- bal amount))
                             'succeeded)))))))))))

(define balance (lambda (account) (account 'balance)))

(define deposit!
  (lambda (amount account) ((account 'deposit!) amount)))

(define withdraw!
  (lambda (amount account) ((account 'withdraw!) amount)))
```

Figure 8.6: A message passing implementation of bank accounts.

```
;;; Imperative version of the MATCH-SEXP program. Procedures not defined
;;; here are the same as before.
(define match-sexp
  (lambda (pat sexp)
    (let ((dict-cell (cell (dict-empty)))
          (failed-flag (cell #f)))
      (letrec ((match!
                ;; MATCH! sets FAILED-FLAG true upon failure, and
                ;; updates the content of the DICT-CELL otherwise.
                ;; It always returns unit.
                (lambda (pat sexp)
                  (cond
                   ((failed-flag?) #u)
                   ((null? pat)
                     (if (null? sexp)
                         #u
                         (fail!)))
                   ((null? sexp) (fail!))
                   ((pattern-constant? pat)
                    (if (sexp=? pat sexp) #u (fail!)))
                   ((pattern-variable? pat)
                    (dict-bind! (pattern-variable-name pat) sexp))
                   (else
                    (begin (match! (car pat) (car sexp))
                           (match! (cdr pat) (cdr sexp)))))))

               (failed-flag? (lambda () (cell-ref failed-flag)))

               (fail! (lambda () (cell-set! failed-flag #t)))

               (dict-bind!
                (lambda (sym sexp)
                  (let ((new-dict (dict-bind
                                   pat sexp (cell-ref dict-cell))))
                    (if (failed? new-dict)
                        (fail!)
                        (cell-set! dict-cell new-dict))))))
        (begin
          (match! pat sexp)
          (if (cell-ref failed-flag)
              'failed
              (cell-ref dict-cell)))))))
```

Figure 8.7: Version of `match-sexp` written in an imperative style.

get : Location $\rightarrow$ Store $\rightharpoonup$ ValueExp
$(get\ L\ \langle L, V\rangle\ .\ S) = V$
$(get\ L_1\ \langle L_2, V\rangle\ .\ S) = (get\ L_1\ S)$, where $L_1 \neq L_2$

The FLK! SOS uses the syntactic domain $E_{mixed} \in$MixedExp, which has a grammar isomorphic to FLK! except for the addition of a `(*cell*` L`)` construct that is used to represent cell values:

E_{mixed} ::= ... [FLK! expressions]
| `(*cell*` L`)` [Cell Value]

The `*cell*` construct may not appear in a user program. ValueExp is the same as that for CBV FLK except that it also contains cell values:

$$V \in \text{ValueExp} = \text{Lit} \cup \{\texttt{(proc}\ I\ E\texttt{)}\} \cup \{\texttt{(pair}\ V_1\ V_2\texttt{)}\} \cup \{\texttt{(*cell*}\ L\texttt{)}\}$$

An operational semantics for FLK! is specified by

$$\langle CF_{FLK!}, \Rightarrow, FC_{FLK!}, IF_{FLK!}, OF_{FLK!}\rangle,$$

where the rewrite rules defining $\Rightarrow$ are specified later and

$$\begin{aligned} CF_{FLK!} &= \text{MixedExp} \times \text{Store} \\ FC_{FLK!} &= \text{ValueExp} \times \text{Store} \\ IF_{FLK!} &= \lambda E\,.\,\langle E, [\,]_{Assignment}\rangle \\ OF_{FLK!} &= \lambda\langle V, S\rangle\,.\,(output\ V) \end{aligned}$$

$$\begin{aligned} (output\ L) &= L \\ (output\ \texttt{(proc}\ I\ E\texttt{)}) &= \texttt{procedure} \\ (output\ \texttt{(pair}\ V_1\ V_2\texttt{)}) &= \texttt{(pair}\ (output\ V_1)\ (output\ V_2)\texttt{)} \\ (output\ \texttt{(*cell*}\ L\texttt{)}) &= \texttt{cell}. \end{aligned}$$

The first component (code component) of an FLK! configuration is a mixed expression that serves the same role as an entire FLK configuration. An FLK! configuration has an additional state component: a store that models the current mapping of locations to value expressions. A computation begins with the initial expression and an empty store, $[\,]_{\text{Assignment}}$, and runs until the code component becomes a value (or the configuration becomes stuck). At this point, an approximation to the final value is returned as the result of the original FLK! expression.

The core rewrite rules for the FLK! semantics appear in Figure 8.8. The `cell` construct and the primitive operators `cell-ref` and `cell-set!` are the only constructs that directly manipulate the store. The [*cell-alloc*] axiom allocates a new location, L_{fresh}, which does not appear in the store, and extends the

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle \texttt{(cell } E\texttt{)}, S\rangle \Rightarrow \langle \texttt{(cell } E'\texttt{)}, S'\rangle} \quad [\textit{cell-progress}]$$

$$\langle \texttt{(cell } V\texttt{)}, S\rangle \Rightarrow \langle \texttt{(*cell* } L_{fresh}\texttt{)}, (\langle L_{fresh}, V\rangle \texttt{ . } S)\rangle, \text{ where } L_{fresh} \text{ is a location that does not appear in } S. \quad [\textit{cell-alloc}]$$

$$\langle \texttt{(primop cell-ref (*cell* } L\texttt{))}, S\rangle \Rightarrow \langle V, S\rangle, \text{ where } (\textit{get } L\ S) = V \quad [\textit{cell-ref}]$$

$$\langle \texttt{(primop cell-set! (*cell* } L\texttt{) } V\texttt{)}, S\rangle \Rightarrow \langle \texttt{\#u}, \langle L, V\rangle \texttt{ . } S\rangle \quad [\textit{cell-set!}]$$

$$\langle \texttt{(primop cell=? (*cell* } L\texttt{) (*cell* } L\texttt{))}, S\rangle \Rightarrow \langle \texttt{\#t}, S\rangle \quad [\textit{cell=?-true}]$$

$$\langle \texttt{(primop cell=? (*cell* } L_1\texttt{) (*cell* } L_2\texttt{))}, S\rangle \Rightarrow \langle \texttt{\#f}, S\rangle, \text{ where } L_1 \neq L_2 \quad [\textit{cell=?-false}]$$

$$\langle \texttt{(primop cell? (*cell* } L\texttt{))}, S\rangle \Rightarrow \langle \texttt{\#t}, S\rangle \quad [\textit{cell?-true}]$$

$$\langle \texttt{(primop cell? } V\texttt{)}, S\rangle \Rightarrow \langle \texttt{\#f}, S\rangle, \text{ where } V \neq \texttt{(*cell* } L\texttt{)} \quad [\textit{cell?-false}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle \texttt{(begin } E_1\ E_2\texttt{)}, S\rangle \Rightarrow \langle \texttt{(begin } E_1\,'\ E_2\texttt{)}, S'\rangle} \quad [\textit{begin-first}]$$

$$\langle \texttt{(begin } V\ E\texttt{)}, S\rangle \Rightarrow \langle E, S\rangle \quad [\textit{begin-rest}]$$

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle \texttt{(rec } I\ E\texttt{)}, S\rangle \Rightarrow \langle \texttt{(rec } I\ E'\texttt{)}, S'\rangle} \quad [\textit{rec-body}]$$

$$\langle \texttt{(rec } I\ V\texttt{)}, S\rangle \Rightarrow \langle [\texttt{(rec } I\ V\texttt{)}/I]\, V, S\rangle \quad [\textit{cbv-rec}]$$

Figure 8.8: Core rewrite rules for FLK!.

store with a new binding between L_{fresh} and the given value. The result of this operation is a `*cell*` value that maintains an index into the store. The [*cell-ref*] rule uses *get* to extract the binding at the location specified by the `*cell*` value. Even though *get* is only a partial function, a `cell-ref` expression can never get stuck because every location appearing in a `*cell*` value must appear in the store. The [*cell-set!*] rule returns a unit value but also prepends a new location/value pair to the store to reflect the assignment.

We have chosen to represent stores as explicit sequences of bindings, but other representations are certainly possible (e.g., representing stores as functions that map locations to values). In our approach, the number of bindings in a store is equal to the number of allocations and assignments performed by the program. An implementation based on such a strategy would be disastrously inefficient: the size of the store would grow throughout the computation, and cell references would take time linear in the size of the growing store. But our goal here is to give a simple semantics for stores, not to implement them efficiently. Any reasonable implementation of FLK! would represent stores in a way that takes advantage of the state-based nature of addressable memory in physical computers.

Of the remaining rules in Figure 8.8, the `begin` rules are straightforward, but the `rec` rules deserve some explanation. Handling `rec` is a bit tricky in the presence of side effects. The basic problem is illustrated by the following FL! example:

```
(let ((counter (cell 0)))
  (begin ((rec fact
            (begin (cell-set! counter
                     (+ (cell-ref counter) 1))
                   (lambda (n)
                     (if (= n 0)
                         1
                         (* n (fact (- n 1)))))))
          5)
         (cell-ref counter)))
```

Here the value computed by the `rec` expression is a factorial procedure. But we're not so much interested in the value of the `rec` as we are in the value of the `counter` cell at the end of the expression. This value tells us how many times `counter` is incremented during the evaluation of the `rec` expression. Presumably, the content of `counter` should be 1. However, if the CBN rule

$$\langle (\texttt{rec}\ I\ E), S\rangle \Rightarrow \langle [(\texttt{rec}\ I\ E)/I]E, S\rangle \qquad [\textit{cbn-rec}]$$

were used, then the value of the above expression would be 6 because the `begin`

expression that is the body of the `rec` would be copied in each unwinding and would be evaluated six times.

To avoid this behavior, the [*cbv-rec*] rule only unwinds the `rec` when the body is a value. The [*rec-body*] rule takes care of rewriting the body of the `rec` into a member of ValueExp. This means that any side effects encountered during the evaluation of the `rec` body are performed only once. In a CBV semantics, the rewriting of the `rec` body will only terminate in a non-stuck state when all uses of the formal parameter introduced by `rec` that appear in the body are "shielded" from immediate evaluation by a `proc`.

The rest of the rules for FLK! appear in Figure 8.9. These rules never actually examine or update the store. Rather, they just specify the "plumbing" that passes the store through the computation in a single-threaded fashion. This guarantees that any changes made by `cell` or `cell-set!` are visible to later uses of `cell-ref`. Except for the additional shuffling of the store component, these rules are the same as those for CBV FLK. The [*FLK-prim*] rule says that the behavior of primitive applications from FLK (as specified by $\Rightarrow_{FLK}$) is inherited by FLK! (as specified by $\Rightarrow$), where the store is unaffected by all such applications.

As a simple example of the FLK! SOS, consider the operational evaluation of the expression `(call` E_{proc} `(cell 3))` where E_{proc} is:

```
(proc c
  (begin (primop cell-set! c
           (primop + 1
            (primop cell-ref c)))
         (primop cell-ref c)))
```

Figure 8.10 shows the transition sequence associated with this expression. Note how the cell value `(*cell* 0)` serves as an unchanging index into the time-dependent store.

▷ **Exercise 8.3** The `begin` construct need not be primitive in FLK!. A desugaring of `begin` into other FLK! constructs must take advantage of the fact that the only notion of time in FL has to do with data dependency. That is, the only thing that forces an expression to be evaluated is that its value is used in the evaluation of another expression. This suggests the following desugaring for `begin` into other FLK! expressions.

$$\mathcal{D}_{\text{exp}}[\![\texttt{(begin } E_1 \ E_2\texttt{)}]\!] = \texttt{(call (proc (}I_{ignore}\texttt{) } \mathcal{D}_{\text{exp}}[\![E_2]\!]\texttt{) } \mathcal{D}_{\text{exp}}[\![E_1]\!]\texttt{)}$$

where $I_{ignore} \notin FreeIds[\![E_2]\!]$.

a. The desugaring for `begin` given above uses constructs only from FLK, which does not support state. Is it possible to determine whether `begin` actually works as advertised (i.e., evaluates E_1 before E_2) in a language that does not support state? Explain your answer, using examples where appropriate.

$$\langle (\texttt{call (proc } I\ E)\ V), S\rangle \Rightarrow \langle [V/I]E, S\rangle \qquad [\textit{cbv-call}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle (\texttt{call } E_1\ E_2), S\rangle \Rightarrow \langle (\texttt{call } E_1\,'\ E_2), S'\rangle} \qquad [\textit{rator-progress}]$$

$$\frac{\langle E_2, S\rangle \Rightarrow \langle E_2\,', S'\rangle}{\begin{array}{l}\langle (\texttt{call (proc } I\ E_1)\ E_2), S\rangle \\ \Rightarrow \langle (\texttt{call (proc } I\ E_1)\ E_2\,'), S'\rangle\end{array}} \qquad [\textit{rand-progress}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle (\texttt{if } E_1\ E_2\ E_3), S\rangle \Rightarrow \langle (\texttt{if } E_1\,'\ E_2\ E_3), S'\rangle} \qquad [\textit{test-progress}]$$

$$\langle (\texttt{if \#t } E_1\ E_2), S\rangle \Rightarrow \langle E_1, S\rangle \qquad [\textit{if-true}]$$

$$\langle (\texttt{if \#f } E_1\ E_2), S\rangle \Rightarrow \langle E_2, S\rangle \qquad [\textit{if-false}]$$

$$\frac{\langle E_{left}, S\rangle \Rightarrow \langle E_{left}\,', S'\rangle}{\langle (\texttt{pair } E_{left}\ E_{right}), S\rangle \Rightarrow \langle (\texttt{pair } E_{left}\,'\ E_{right}), S'\rangle} \qquad [\textit{left-progress}]$$

$$\frac{\langle E_{right}, S\rangle \Rightarrow \langle E_{right}\,', S'\rangle}{\langle (\texttt{pair } V_{left}\ E_{right}), S\rangle \Rightarrow \langle (\texttt{pair } V_{left}\ E_{right}\,'), S'\rangle} \qquad [\textit{right-progress}]$$

$$\frac{\langle E, S\rangle \Rightarrow \langle E', S'\rangle}{\langle (\texttt{primop } O\ E), S\rangle \Rightarrow \langle (\texttt{primop } O\ E'), S'\rangle} \qquad [\textit{unary-arg}]$$

$$\frac{\langle E_1, S\rangle \Rightarrow \langle E_1\,', S'\rangle}{\langle (\texttt{primop } O\ E_1\ E_2), S\rangle \Rightarrow \langle (\texttt{primop } O\ E_1\,'\ E_2), S\rangle} \qquad [\textit{binary-arg1}]$$

$$\frac{\langle E_2, S\rangle \Rightarrow \langle E_2\,', S'\rangle}{\langle (\texttt{primop } O\ V_1\ E_2), S\rangle \Rightarrow \langle (\texttt{primop } O\ V_1\ E_2\,'), S'\rangle} \qquad [\textit{binary-arg2}]$$

$$\frac{(\texttt{primop } O_{FLK}\ V^*) \Rightarrow_{FLK} V_{result}}{\langle (\texttt{primop } O_{FLK}\ V^*), S\rangle \ \Rightarrow\ \langle V_{result}, S\rangle,} \qquad [\textit{FLK-prim}]$$

where $O_{FLK} \in \text{Primop}_{FLK!} - \{\texttt{cell-ref}, \texttt{cell-set!}, \text{ and } \texttt{cell?}\}$

Figure 8.9: FLK! rewrite rules for single-threading the store.

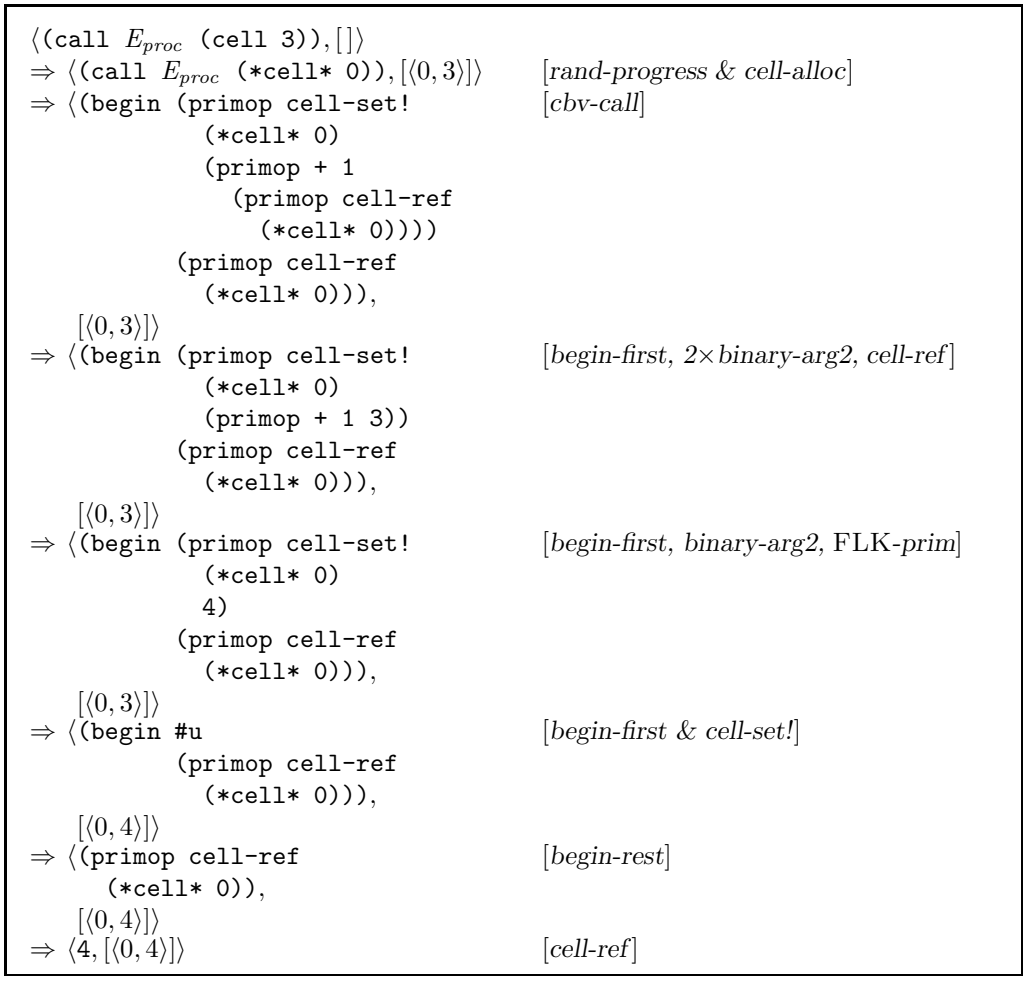

Figure 8.10: Operational evaluation of a sample FLK! expression.

b. Explain why the above desugaring would not work for a CBN version of FL!.

c. Write a desugaring for `begin` in CBV FLK! that does not require any condition involving the free variables of E_1 or E_2. (Hint: use thunks!)

d. Is it possible to write a desugaring for `begin` that works in both CBV and CBN FLK! ? If so, give the desugaring; if not, explain why not. ◁

▷ **Exercise 8.4** The introduction of side effects can complicate reasoning about programs. For example, program transformations that are safe in FL aren't necessarily safe in FL!.

- List three transforms that are safe in FL but not in FL!. Provide counter-examples to demonstrate why they are not safe in FL!.
- List three transforms that are safe in both FL and FL!.
- Consider transforms that do not mention any of the new features of FLK!. Are there any such transforms that are safe in FL! but not in FL? If so, exhibit such a transform. If not, explain. ◁

8.2.4 A Denotational Semantics for FLK!

Now we'll study the semantics of FLK! from the denotational perspective. As in the operational approach, notions of location and store will be used to model state. The notion of computation will be modified so that stores flow through a computation in a single-threaded fashion. The power of the computation abstraction will be illustrated by the fact that only those constructs that explicitly refer to the store need new valuation clauses; other constructs are described by their (unmodified) FLK valuation clauses.

8.2.4.1 Stores

The denotational treatment of stores and locations is summarized in Figure 8.11.

Here locations are represented as natural numbers and stores are represented as functions that map locations to elements of the *Assignment* domain. Stores do not map locations directly to values because it is necessary to encode the fact that not all locations have values assigned to them. The distinguished element *unassigned* in the lifted sum domain *Assignment* is used to indicate that a location is unassigned. *unassigned* serves the same purpose for stores that *unbound* serves for environments.

The domain *Storable* of storable entities varies from language to language. In FLK!, which is a CBV language, *Storable* = *Value*, but a CBN version of

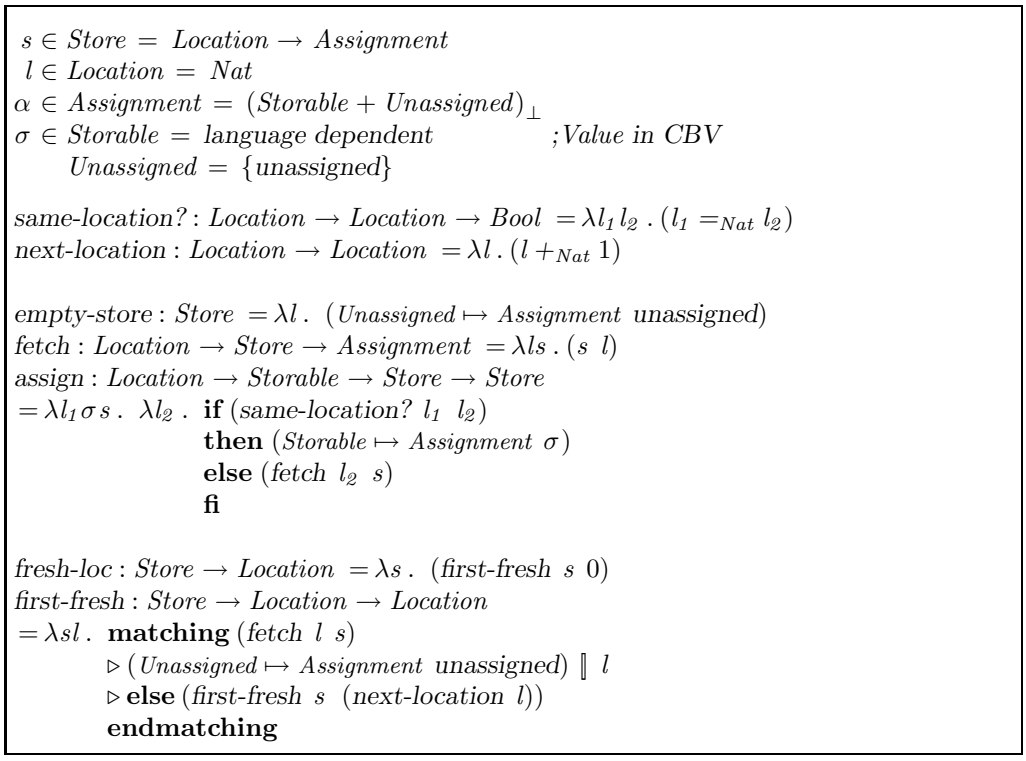

Figure 8.11: Denotational treatment of stores.

FLK! would have *Storable* = *Computation*. In both CBV and CBN FLK!, it happens that *Storable* = *Denotable*, but this need not be the case in general. For example, in PASCAL, procedures can be named and (with certain restrictions) be passed as arguments, but they may not be assigned to variables or stored as the components of data structures.

There are several auxiliary functions for manipulating stores. *fetch* and *assign* are functions on stores that are reminiscent of *lookup* and *extend* on environments. The purpose of *fresh-loc* is to return an unassigned location from the given store. Since locations are natural numbers, one way of doing this is by scanning the store starting with location 0 and incrementing the location until an unassigned location is found. We assume an unbounded store, so that *fresh-loc* never fails to return a fresh location. To model a bounded store (which would be more realistic), *fresh-loc* could potentially return an indication that the attempt to find a fresh location failed.

8.2.4.2 Computations

Previously, a computation was just an expressible value. But in the presence of state, a computation needs to embody the single-threaded nature of stores. The following domain definition captures this idea:

$$c \in \mathit{Computation} = \mathit{Store} \rightarrow (\mathit{Expressible} \times \mathit{Store})$$

Here, a computation accepts an initial store and returns two entities:

- The expressible value computed by the computation.
- A final store that reflects all the allocations and assignments performed by the computation.

When composing two computations, single-threadedness can be achieved by supplying the final store of the first computation as the initial store of the second.

It is not difficult to show that the new *Computation* domain is pointed. This means that it is possible to find fixed points over computations, as required in the semantics of `rec`.

Recall that numerous auxiliary functions must be defined as part of the computation abstraction. Figure 8.12 shows the definitions of these functions for the store-based version of *Computation*. *val-to-comp* injects a value into a computation by injecting it into an expressible value and passing a store around it unchanged. Similarly, *error-comp* passes a store around an error expressible value.

The main means of gluing computations together is *with-value*. It takes a computation c_1 and a function f that maps a value to a computation c_2 and

$c \in \mathit{Computation} = \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store})$

$\mathit{expr\text{-}to\text{-}comp} : \mathit{Expressible} \to \mathit{Computation} = \lambda x\,.\ \lambda s\,.\ \langle x, s\rangle$

$\mathit{val\text{-}to\text{-}comp} : \mathit{Value} \to \mathit{Computation} = \lambda v\,.\ (\mathit{expr\text{-}to\text{-}comp}\ (\mathit{Value} \mapsto \mathit{Expressible}\ v))$

$\mathit{err\text{-}to\text{-}comp} : \mathit{Error} \to \mathit{Computation} = \lambda I\,.\ (\mathit{expr\text{-}to\text{-}comp}\ (\mathit{Error} \mapsto \mathit{Expressible}\ I))$

$\mathit{error\text{-}comp} : \mathit{Computation} = (\mathit{err\text{-}to\text{-}comp}\ \texttt{error})$

$\mathit{with\text{-}value} : \mathit{Computation} \to (\mathit{Value} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda c f\ .\ \lambda s_1\ .$ **matching** $(c\ s_1)$
$\quad \triangleright \langle(\mathit{Value} \mapsto \mathit{Expressible}\ v), s_2\rangle \Vert (f\ v\ s_2)$
$\quad \triangleright \langle(\mathit{Error} \mapsto \mathit{Expressible}\ \mathit{error}), s_2\rangle \Vert (\mathit{error\text{-}comp}\ s_2)$
$\quad$ **endmatching**

with-values, *with-boolean*, *with-procedure*, etc. can be written in terms of *with-value*.

$\mathit{check\text{-}location} : \mathit{Value} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation}$
$= \lambda v f\ .$ **matching** v
$\quad \triangleright (\mathit{Location} \mapsto \mathit{Value}\ l) \Vert (f\ l)$
$\quad \triangleright$ **else** *error-comp*
$\quad$ **endmatching**

check-boolean, *check-procedure*, etc. are similar.

Figure 8.12: Store-based implementation of the computation abstraction.

returns the computation that results from composing c_1 and c_2. Like the action-combining `after` procedure in Section 8.1.3.3, the main purpose of *with-value* is to support the monadic style of threading state by handling the "plumbing" between computations: the value argument to f is the (non-error) expressible value produced by c_1 and the initial store of c_2 is the final store of c_1. In the case where c_1 produces an error rather than a value, f is ignored and the resulting computation is equivalent to c_1. It is instructive to unwind the type of f:

$$\mathit{Value} \to \mathit{Computation} \;=\; \mathit{Value} \to \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store})$$

This makes it clear that f can be viewed as a function that maps two (curried) arguments (a value and store) to two (paired) results (an expressible value and store).

Other *with-* functions, like *with-values*, *with-procedure*, *with-boolean* can be written in the same style as *with-value*. There is a parallel collection of *check-* functions that differ from the *with-* functions only in that their initial argument is a value rather than a computation.

In the presence of state, there are a few more auxiliary functions involving computations that are especially handy. These are defined in Figure 8.13. *allocating* allocates a location for a storable value and passes it (and the up-

$$\begin{array}{l}
\mathit{allocating} : \mathit{Storable} \to (\mathit{Location} \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda\sigma f \;.\; \lambda s \;.\; (f \;\; (\mathit{fresh\text{-}loc}\;\; s) \;\; (\mathit{assign}\;\; (\mathit{fresh\text{-}loc}\;\; s) \;\; \sigma \;\; s)) \\
\\
\mathit{fetching} : \mathit{Location} \to (\mathit{Storable} \to \mathit{Computation}) \to \mathit{Computation} \\
= \lambda l f \;.\; \lambda s \;.\; \mathbf{matching}\;(\mathit{fetch}\;\; l \;\; s) \\
\qquad\qquad \triangleright\,(\mathit{Storable} \mapsto \mathit{Assignment}\;\sigma) \;[\!]\; (f \;\; \sigma \;\; s) \\
\qquad\qquad \triangleright\,\mathbf{else}\;(\mathit{error\text{-}comp}\;\; s) \\
\qquad\qquad \mathbf{endmatching} \\
\\
\mathit{update} : \mathit{Location} \to \mathit{Storable} \to \mathit{Computation} \\
= \lambda l \sigma \;.\; \lambda s \;.\; \langle (\mathit{Value} \mapsto \mathit{Expressible}\;\; (\mathit{Unit} \mapsto \mathit{Value}\;\; \mathit{unit})), (\mathit{assign}\;\; l \;\; \sigma \;\; s) \rangle \\
\\
\mathit{sequence} : \mathit{Computation} \to \mathit{Computation} \to \mathit{Computation} \\
= \lambda c_1\, c_2 \;.\; (\mathit{with\text{-}value}\;\; c_1 \;\; (\lambda v \;.\; c_2))
\end{array}$$

Figure 8.13: Auxiliary functions for store-based computations.

dated store) to a computation-producing function. *fetching* finds the storable value at a location and passes it (and the unchanged store) to a computation-producing function. *update* takes a location and storable value and returns a unit-producing computation whose final store includes an assignment between the location and value. *sequence* glues two computations together by supplying

the final store of the first as the initial store of the second; the expressible value produced by the first is ignored.

Reasoning about computations directly in terms of the auxiliary functions can be very tedious. Figure 8.14 presents a number of high-level equalities that greatly facilitate reasoning about computations. We leave the proofs of these

1. $(\textit{with-value}\ (\textit{val-to-comp}\ v)\ f) = (f\ v)$
2. $(\textit{with-value}\ c\ (\lambda v\,.\,(\textit{val-to-comp}\ v))) = \mathrm{c}$
3. $(\textit{with-procedure}\ (\textit{val-to-comp}\ (\textit{Procedure} \mapsto \textit{Value}\ p))\ f) = (f\ p)$
   Similarly for *with-boolean*, *with-integer*, etc.
4. $(\textit{with-value}\ (\textit{with-value}\ c\ f)\ g) = (\textit{with-value}\ c\ (\lambda v\,.\,(\textit{with-value}\ (f\ v)\ g)))$
5. $(\textit{with-value}\ (\textit{check-location}\ v\ f)\ g)$
   $= (\textit{check-location}\ v\ (\lambda l\,.\,(\textit{with-value}\ (f\ l)\ g)))$
   similarly for *check-boolean*, *check-integer*
6. $(\textit{with-value}\ (\textit{allocating}\ \sigma\ f)\ g) = (\textit{allocating}\ \sigma\ (\lambda l\,.\,(\textit{with-value}\ (f\ l)\ g)))$
7. $(\textit{with-value}\ (\textit{fetching}\ l\ f)\ g) = (\textit{fetching}\ l\ (\lambda\sigma\,.\,(\textit{with-value}\ (f\ \sigma)\ g)))$
8. $(\textit{with-value}\ (\textit{update}\ l\ \sigma)\ f)$
   $= (\textit{sequence}\ (\textit{update}\ l\ \sigma)\ (f\ (\textit{Unit} \mapsto \textit{Value}\ \textit{unit})))$
9. $(\textit{with-value}\ (\textit{sequence}\ c1\ c2)\ f) = (\textit{sequence}\ c1\ (\textit{with-value}\ c2\ f))$

Figure 8.14: Useful equalities on computations. It is assumed that newly introduced variables do not conflict with free identifiers elsewhere in the expression.

equalities as exercises for the reader. We require the first four equalities in Figure 8.14 to be true of any notion of computation that we introduce, and equalities 5–9 to be true of any notion of computation that supports state.

8.2.4.3 Valuation Clauses

The denotational specification of FLK! is summarized in Figure 8.15. The *Value* domain has been extended with locations, which represent cell values. Since FLK! is a CBV language, both *Denotable* and *Storable* equal *Value*. As always, $\mathcal{E}$ has the signature Exp $\to$ *Environment* $\to$ *Computation*. There are two semantic functions for primitives. $\mathcal{P}_{FLK!}$ is the version for FLK!, while $\mathcal{P}_{FLK}$ is the version inherited from FLK.

With the help of the auxiliary functions, the valuation clauses are surprisingly compact. In fact, only one clause (`rec`) explicitly mentions the store! `begin` sequences two computations. `cell` allocates a location for its content and returns

$c \in \mathit{Computation} = \mathit{Store} \to (\mathit{Expressible} \times \mathit{Store})$
$v \in \mathit{Value} = \mathit{Unit} + \mathit{Bool} + \mathit{Int} + \mathit{Sym} + \mathit{Pair} + \mathit{Procedure} + \mathit{Location}$
$\delta \in \mathit{Denotable} = \mathit{Value}$
$\sigma \in \mathit{Storable} = \mathit{Value}$
$p \in \mathit{Procedure} = \mathit{Denotable} \to \mathit{Computation}$

$\mathcal{E} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation}$
$\mathcal{P}_{FLK} : \mathrm{Primop} \to \mathit{Value}^* \to \mathit{Expressible}$
$\mathcal{P}_{FLK!} : \mathrm{Primop} \to \mathit{Value}^* \to \mathit{Computation}$

$\mathcal{E}[\![\texttt{(begin } E_1 \ E_2\texttt{)}]\!] = \lambda e\,.\ (\mathit{sequence}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\mathcal{E}[\![E_2]\!]\ e))$

$\mathcal{E}[\![\texttt{(cell } E\texttt{)}]\!]$
$= \lambda e\,.\ (\mathit{with\text{-}value}\ (\mathcal{E}[\![E]\!]\ e)$
$\qquad (\lambda v\,.\ (\mathit{allocating}\ v\ (\lambda l\,.\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l))))))$

$\mathcal{P}_{FLK!}[\![\texttt{cell-ref}]\!] = \lambda[v]\,.\ (\mathit{check\text{-}location}\ v\ (\lambda l\,.\ (\mathit{fetching}\ l\ (\lambda v\,.\ (\mathit{val\text{-}to\text{-}comp}\ v)))))$

$\mathcal{P}_{FLK!}[\![\texttt{cell-set!}]\!] = \lambda[v_1, v_2]\,.\ (\mathit{check\text{-}location}\ v_1\ (\lambda l\,.\ (\mathit{update}\ l\ v_2)))$

$\mathcal{P}_{FLK!}[\![\texttt{cell=?}]\!] =$
$\quad \lambda[v_1, v_2]\,.\ (\mathit{check}\text{-location}\ v_1$
$\qquad (\lambda l_1\ .\ (\mathit{check}\text{-location}\ v_2$
$\qquad\quad (\lambda l_2\ .\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Bool} \mapsto \mathit{Value}\ (l_1 = l_2)))))))$

$\mathcal{P}_{FLK!}[\![\texttt{cell?}]\!] = \lambda[v]\,.$ **matching** v
$\qquad \triangleright\, (\mathit{Location} \mapsto \mathit{Value}\ l)\ [\![\ (\mathit{val\text{-}to\text{-}comp}\ (\mathit{Bool} \mapsto \mathit{Value}\ \mathit{true}))$
$\qquad \triangleright$ **else** $(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Bool} \mapsto \mathit{Value}\ \mathit{false}))$
$\qquad$ **endmatching**

$\mathcal{P}_{FLK!}[\![O]\!]$; $O \in$Primop $- \{\texttt{cell-ref}, \texttt{cell-set!}, \texttt{cell?}\}$
$= \lambda v^*\,.\ (\mathit{expr\text{-}to\text{-}comp}\ (\mathcal{P}_{FLK}[\![O]\!]\ v^*))$

$\mathcal{E}[\![\texttt{(rec } I\ E\texttt{)}]\!] = \lambda e\,.\ \mathbf{fix}_{\mathit{Computation}}\ (\lambda c\,.\ \lambda s\,.\ \mathcal{E}[\![E]\!]\ [I :: (\mathit{extract\text{-}value}\ c\ s)]e\ s)$

$\mathit{extract\text{-}value} : \mathit{Computation} \to \mathit{Store} \to \mathit{Binding}$
$= \lambda cs\,.$ **matching** $(c\ s)$
$\qquad \triangleright < (\mathit{Value} \mapsto \mathit{Expressible}\ v),\ s' > [\![\ (\mathit{Denotable} \mapsto \mathit{Binding}\ v)$
$\qquad \triangleright < (\mathit{Error} \mapsto \mathit{Expressible}\ \mathit{error}),\ s' > [\![\ \bot_{\mathit{Binding}}$
$\qquad$ **endmatching**

Figure 8.15: Essential valuation clauses for FLK!. Clauses not shown here inherit their definition from FLK.

the location as its resulting value. `cell-ref` fetches the value of a location and returns it, while `cell-set!` updates a location to contain a new value. `cell?` simply checks the tag on a value. Other primitives are handled by passing them off to $\mathcal{P}_{FLK}$ and converting the result into a computation. This works because none of the primitives inherited from FLK has any effect on the store.

The only really tricky clause is the one for `rec`. The valuation clause presented here is a variant of the CBV version presented in Section 7.1.3. The only difference is that it is necessary to supply *extract-value* with the current store in order to coerce the computation into a binding.

And that's it! By the magic of the monadic style, all the other valuation clauses are inherited unchanged from the denotational definition of CBV FLK. For example, the clause for `call` is still:

$$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \\ \lambda e\,.\ (\textit{with-procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\ (\textit{with-value}\ (\mathcal{E}[\![E_2]\!]\ e)\ p)))$$

The valuation clauses are very concise, but their level of abstraction can make them difficult to understand. To get a better feel for the valuation clauses, it can be helpful to strip away the abstractions by "in-lining" the auxiliary functions. For example, here is a version of the `call` clause without any auxiliary functions:

$$\begin{array}{l}\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \\ \quad \lambda e s_0\,.\ \textbf{matching}\ (\mathcal{E}[\![E_1]\!]\ e\ s_0) \\ \qquad \triangleright \langle (\textit{Value} \mapsto \textit{Expressible}\ (\textit{Procedure} \mapsto \textit{Value}\ p)), s_1\rangle\ [\![\\ \qquad\ \textbf{matching}\ (\mathcal{E}[\![E_2]\!]\ e\ s_1) \\ \qquad\ \triangleright \langle (\textit{Value} \mapsto \textit{Expressible}\ v), s_2\rangle\ [\![\ (p\ v\ s_2) \\ \qquad\ \triangleright \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_2\rangle\ [\![\ \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_2\rangle \\ \qquad\ \textbf{endmatching} \\ \qquad \triangleright \langle (\textit{Value} \mapsto \textit{Expressible}\ v), s_1\rangle\ [\![\ \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_1\rangle \\ \qquad \triangleright \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_1\rangle\ [\![\ \langle (\textit{Error} \mapsto \textit{Expressible}\ \textit{error}), s_1\rangle \\ \qquad \textbf{endmatching}\end{array}$$

The single-threaded nature of the store that is implicit in the original clause is explicit in the expanded clause. Evaluating E_1 in e with s_0 yields an expressible value (call it x_1) and a store s_1. If the x_1 is a procedure value p, E_2 is evaluated in e with s_1 to yield a second expressible value (call it x_2) and another store, s_2. If x_2 is a value v, then p, whose signature is

$$\textit{Value} \to \textit{Store} \to (\textit{Expressible} \times \textit{Store})$$

is applied to the value and the store. In error situations (x_1 is not a procedure or x_2 is not a value), expressible error values are propagated along with the updated store.

You may find it helpful to perform this sort of expansion on other valuation clauses. After you have done several, you may start to appreciate the purpose

of the auxiliary functions! As usual, it is also instructive to make sure that all of the valuation clauses type check.

8.2.5 Referential Transparency, Interference, and Purity

We noted earlier (page 319) that stateless languages like FL are **referentially transparent**. Referential transparency is an important property when reasoning about programs, especially when analyzing and transforming programs.

For example, consider the following program transformations:

T1: `(+` E_a E_a`)` $\longrightarrow$ `(*` `2` E_a`)`

T2: `(+` E_b E_c`)` $\longrightarrow$ `(+` E_c E_b`)`

Under what conditions are such transformations **safe**, i.e., guaranteed to preserve the meaning of a program?[4]

In a referentially transparent language like FL, these two transformations are always safe. In **T1**, E_a always has the same value no matter how many times it is evaluated. In **T2**, reordering E_b and E_c cannot change their values because they are still in the same naming context as before.

However, in a stateful language like FL!, neither of these transformations is always safe. For example, in **T1**, suppose that E_a increments a counter in addition to returning a result. Then `(+` E_a E_a`)` will increment the counter twice, but `(*` `2` E_a`)` will only increment it once. In **T2**, suppose that E_b increments a counter whose value is returned by E_c. Then swapping E_b and E_c changes the value returned by E_c. The problem in these cases is that expressions can depend on the implicit store threaded through their evaluation, so it is generally not safe to replace them by a value or change their relative positions. In particular, an expression can depend on the store by:

- allocating a location in the store (which includes initialization in our semantics),
- reading the value stored at a location, or
- writing a value into a location.

Nevertheless, there are still many situations in which the transformations are safe, even in a stateful language. Let us say that an expression E_1 **interferes** with E_2 when E_1 allocates or writes a store location that is read and/or written by E_2. Then **T1** is safe as long as E_a does not interfere with itself or the rest

[4]For the purposes of this discussion, we choose to treat all errors and divergence as observationally equivalent. That is, we do not care if a transformation changes the error signaled by a program or changes an error-signaling program to a diverging one (or vice versa).

of the program and **T2** is safe as long as E_b or E_c do not interfere with each other. Classical compiler optimizations like code motion, common subexpression elimination, and dead code removal require reasoning about the interference between expressions.

A particularly simple form of non-interference involves expressions that do not depend on the store at all. An expression is **pure** when it does not allocate, read, or write any store locations. A pure expression does not interfere with any other expression, and so it can be treated as if it were in a referentially transparent language. For instance, it is safe to replace a pure expression by an expression having the same value or to move a pure expression to a different position in the same naming context.

Neither interference nor purity is a computable property. However, there are conservative approximations to these properties that are computable. For example, a common syntactic technique for approximating purity is to observe the following in a language with cells:

- variable references and abstractions (`lambda` expressions) do not depend on the store and so are syntactically pure;
- conditionals, `let` expressions, `pair` expressions, and primitive applications (except those involving cell primitives) are syntactically pure if all their subexpressions are syntactically pure;
- all other expressions, including primitive applications of cell primitives and procedure applications, are assumed to be impure.

Expressions that are pure by these rules are called **syntactic values**. We shall use this notion later in our discussion of polymorphic types, type reconstruction, and abstract types (Chapters **??** and 15). Chapter 16 will present a more flexible mechanism for statically determining the side effects (and therefore interference properties) an expression may have.

▷ **Exercise 8.5** Show that the store-based definition of *Computation* is pointed. ◁

▷ **Exercise 8.6**

a. Prove that the first four equalities in Figure 8.14 hold when $Computation = Expressible$.

b. Prove that all nine equalities in Figure 8.14 hold when $Computation = Store \to (Expressible \times Store)$. ◁

▷ **Exercise 8.7**

a. What is the value of `(rec a a)` under the call-by-value denotational semantics for FLK in the previous chapter?

b. What is the value of `(rec a a)` under the operational semantics for FLK!?

c. What is the value of `(rec a a)` under the denotational semantics for FLK!?

d. Explain any discrepancy in your answers to the first three parts of this question.

◁

▷ **Exercise 8.8** The FL! language definition includes a simple immutable data structure called the *pair*. In this problem, we introduce a mutable pair. Mutable pairs are a simple kind of mutable structure similar to the mutable records found in many imperative languages. (See Section 10.1.4 for a discussion of mutable data structures.)

Suppose the FLK! language is extended in the following way:

E ::= ...
| `(mpair` E_l E_r`)` | `(mfst` E_{mp}`)` | `(msnd` E_{mp}`)`
| `(set-mfst!` E_{mp} E_l`)` | `(set-msnd!` E_{mp} E_r`)`

The new constructs have the following informal semantics:

- `(mpair` E_l E_r`)` creates a new mutable pair value with two fields called *mfst* and *msnd*. The values of E_l and E_r are stored in the *mfst* and *msnd* fields, respectively.
- If E_{mp} evaluates to a mutable pair, then `(mfst` E_{mp}`)` returns the content of the *mfst* field of the pair. Otherwise, `mfst` produces an error. Similarly for `msnd`.
- If E_{mp} evaluates to a mutable pair, then `(set-mfst!` E_{mp} E_l`)` mutates the mutable pair so that the *mfst* field contains the value of E_l. If E_{mp} evaluates to anything else, or if evaluating E_l gives an error, then `set-mfst!` generates an error. Similarly for `set-msnd!`.

For example, here are some expressions involving mutable pairs:

```
(let ((foo (mpair 1 2)))
  (begin
    (set-mfst! foo 6)
    (+ (mfst foo) (msnd foo)))) ⟶eval 8

(let ((bar (mpair 8 (mpair 4 3))))
  (begin
    (set-mfst! bar (msnd bar))
    (set-msnd! (msnd bar) (mfst (mfst bar)))
    (+ (mfst (msnd bar)) (msnd (msnd bar))))) ⟶eval 8
```

a. Extend the denotational semantics of FLK! to handle `mpair`, `mfst`, `msnd`, `set-mfst!`, and `set-msnd!`.

   i. Describe any additions or modifications you make to the semantic domains of FLK!.

ii. Give valuation clauses for the five constructs. (You should not have to modify any of the existing valuation clauses.)

iii. Define any auxiliary functions necessary for your valuation clauses.

b. Consider the following potential desugaring for `mpair`, `mfst`, and `set-mfst!` (`msnd` and `set-msnd!` would be handled similarly):

$$\mathcal{D}[\![\texttt{(mpair } E_l \ E_r\texttt{)}]\!] = \texttt{(pair (cell } \mathcal{D}[\![E_l]\!]\texttt{) (cell } \mathcal{D}[\![E_r]\!]\texttt{))}$$

$$\mathcal{D}[\![\texttt{(mfst } E_{mp}\texttt{)}]\!] = \texttt{(cell-ref (left } \mathcal{D}[\![E_{mp}]\!]\texttt{))}$$

$$\mathcal{D}[\![\texttt{(set-mfst! } E_{mp} \ E_l\texttt{)}]\!] = \texttt{(cell-set! (left } \mathcal{D}[\![E_{mp}]\!]\texttt{) } \mathcal{D}[\![E_l]\!]\texttt{)}$$

Is this desugaring consistent with the semantics of mutable pairs? If it is, explain why; if not, show an expression whose meaning differs under this desugaring. ◁

▷ **Exercise 8.9** A common problem when working with state is data consistency. For example, consider a database application that manages the accounts of a bank. Transferring an amount of money between two accounts implies subtracting the amount from the first account and adding it to the second one. If we transfer money only between accounts of the same bank, the total amount of money present in all the accounts should remain the same. However, if something bad occurs between the subtraction and the addition (e.g., a system crash), a certain amount might simply vanish! To prevent this, in database programming, all modifications to the database are required to occur within a *transaction*.

Intuitively, a transaction is a series of modifications to a database that become permanent only when the transaction is successfully terminated (the technical term is *committed*). If the user decides to *abort* (i.e., cancel) the transaction, or the system crashes before the transaction is committed, all the modifications are "undone."

Abe Stract, president and CEO of Intrusive Databases, Inc., decides to add transactions to FL!. In Abe's language, the store will act as the database: queries of the database are `cell-ref`s, and modifications are performed by `cell-set!`. It is an error to perform a `cell-set!` when there is no active transaction.

Abe extends the grammar of FLK! by the following clauses:

$E ::= \ldots$	[As before]
| `(begin-transaction!)`	[Begin Transaction]
| `(commit!)`	[Commit Transaction]
| `(abort!)`	[Abort Transaction]

The informal semantics of transactions are:

- `(begin-transaction!)` begins a transaction. The transaction continues until either a `commit!` or an `abort!` is encountered — it is an error if the program ends and a transaction has not been ended or aborted.
- `(commit!)` successfully terminates the current transaction. It is an error if no transaction is in progress.

- `(abort!)` ends the current transaction and undoes all of its modifications. It is an error if no transaction is in progress.

Like `cell-set!`, the three transaction operations all return *unit*.

Transactions may be nested, in which case `abort!` and `commit!` only end the current (innermost) transaction. An `abort!` of a transaction undoes the modifications of the transaction including modifications made by nested transactions.

Here is how Abe might write a transfer between two bank accounts (represented as cells) using transactions:

```
(define transfer
  (lambda (from to amount)
    (begin (begin-transaction!)
           (cell-set! from (- (cell-ref from) amount))
           (cell-set! to   (+ (cell-ref to)   amount))
           (if (< (cell-ref from) 0)
               (begin (abort!)
                      'failed)
               (begin (commit!)
                      'succeeded)))))
```

Here are more examples of the behavior of transactions; we assume the expressions are evaluated in order.

```
(define cell-1 (cell 0))
(define cell-2 (cell 10))

(define inc!
  (lambda (a-cell) (cell-set! a-cell (+ (cell-ref a-cell) 1))))

(define current-state
   (lambda ()
      (list (cell-ref cell-1) (cell-ref cell-2))))
```

`(current-state)` $\xrightarrow[FL!]{}$ $[0, 10]$

```
(begin (begin-transaction!)
       (inc! cell-1)
       (commit!)
       (current-state))
```
$\xrightarrow[FL!]{}$ $[1, 10]$

```
(begin (begin-transaction!)
       (inc! cell-2)
       (abort!)
       (current-state))
```
$\xrightarrow[FL!]{}$ $[1, 10]$

```
(begin (begin-transaction!)
       (inc! cell-1)
       (begin (begin-transaction!)
              (inc! cell-2)
              (abort!))
       (commit!)
       (current-state)) ⟶_{FL!} [2, 10]

(begin (begin-transaction!)
       (inc! cell-1)
       (begin (begin-transaction!)
              (inc! cell-2)
              (commit!))        ;; End inner transaction,
       (abort!)                 ;; but abort! outer transaction.
       (current-state)) ⟶_{FL!} [2, 10]

(begin (begin-transaction!)     ;; commit! returns #u
       (inc! cell-2)
       (commit!)) ⟶_{FL!} unit

(current-state) ⟶_{FL!} [2, 12]
```

Abe also points out some programs that generate errors (each interacts with the database in a completely independent session):

```
(begin-transaction!) ⟶_{FL!} error : transaction − not − terminated

(commit!) ⟶_{FL!} error : no − current − transaction

(let ((a-cell (cell 0)))
   (begin (cell-set! a-cell 5)
          (cell-ref a-cell))) ⟶_{FL!} error : not − in − a − transaction

(let ((a-cell (cell 0)))
  (begin (begin-transaction!)
         (cell-set! a-cell 5)
         (commit!)
         (cell-set! a-cell 7)   ;; commit! ends transaction,
                                ;; so invalid modification
         (cell-ref a-cell))) ⟶_{FL!} error : not − in − a − transaction
```

a. Extend the operational semantics of FLK! (Section 8.2.3) to handle transactions:

   i. Define the configurations, the set of final configurations, the input function, and the output function.

   ii. Provide transition rules for `begin-transaction!`, `commit!`, `abort!`, and `cell-set!`:

b. Modify the denotational semantics of FLK! to handle transactions.

   i. Give the necessary additions or modifications to the semantic domains of FLK!.

ii. Some auxiliary functions used by the FLK! denotational semantics might need to be modified (e.g., as a result of the changes in the semantic domains). Give their new definitions.

iii. Write the valuation clauses for the three new constructs. ◁

▷ **Exercise 8.10** Clark Smarter of the Photocopy Research Center has developed a new backtracking construct for FLK! called `try`:

E ::= ...
| `(try` E_1 E_2`)` [Backtracking]

The informal semantics of `(try` E_1 E_2`)` is as follows: First E_1 is evaluated, and if E_1 returns *true*, then `try` ignores E_2 and returns *true*. If E_1 evaluates to *false*, then the side effects of E_1 are discarded, and the value of `try` is the value of E_2. If the value of E_1 is neither *true* nor *false*, then the `try` expression yields an error. `try` is thus an elementary backtracking construct. It allows the exploration of one alternative, and, if that does not work, restores the initial state and tries a second alternative.

Here's an example of a program that uses `try`:

```
(let ((balance (cell 200)))
  (let ((withdraw (lambda (n)
                    (begin (cell-set! balance
                                      (- (cell-ref balance) n))
                           (> (cell-ref balance) 0)))))
    ; First try to withdraw 250; if that fails, withdraw 10 from
    ; the original balance.
    (begin (try (withdraw 250)
                (withdraw 10))
           (cell-ref balance))))
```

FL› 190

Clark knows the pitfalls of informal semantics. When writing up the documentation for `try`, he decides to give an operational and denotational semantics for his new construct.

a. First Clark tries to find an operational semantics for `try`:

   i. In attempting to give an operational semantics for `try`, Clark realizes that he must extend the configuration space CF, so he adds a new intermediate expression to E. Describe the new intermediate form and its purpose. (Hint: you may want to think about the next part before answering this one.)

   ii. Provide all of the rewrite rules which are necessary to handle the `try` construct.

b. Next Clark wants to find a denotational semantics for `try`. Help Clark by writing the valuation clause that handles the `try` expression.

c. Clark shows his operational and denotational semantics definitions of `try` to language implementor Hardy Ware. Hardy says, "These semantic definitions are all well and good, but implementing `try` efficiently is going to be tough."

i. Explain what Hardy means by describing what difficulties would be encountered in implementing `try` efficiently on physical computers where state-based memory devices implement the binding of locations to values.

ii. Sketch a strategy for implementing `try` that does not require making a copy of the entire store. ◁

8.3 Mutable Variables: FLAVAR!

In FLK!, the only entity that can change over time is the contents of a mutable cell. So-called "variables" are actually constants whose value cannot change during the execution of a program. While mutable cells are sufficient for implementing any state-based program, they are not always convenient to use. Here we explore a variant of FLK! called FLAVAR! in which every variable becomes a mutable entity. We will also revisit the issue of parameter-passing in the context of state by examining four parameter-passing mechanisms for FLAVAR!.

8.3.1 Mutable Variables

In FL!, it can be difficult to modify a program to make a previously constant quantity mutable. For example, suppose an FLK program binds the variable `addresses` to a list of names and addresses. Since both variables and pairs are immutable in FL!, the meaning of `addresses` cannot change during the execution of the program. Suppose that we later decide to modify the program so that it dynamically updates the address list. Then it is necessary to rebind `addresses` to a mutable cell whose contents is a list. Furthermore, we must find all references to `addresses` in the existing program and replace them by `(cell-ref addresses)`.[5]

Most programming languages offer a more convenient way of making such changes: **mutable variables**. A variable is mutable if the value it is bound to can change over time. The variables of FL and FL! are somewhat misnamed, because their values can't vary over time; rather, they are names for constants. In contrast, variables in languages like SCHEME, C, PASCAL, and FORTRAN can have their values changed by assignment during the execution of the program. In these languages, modifying the address program would not require finding and updating all references to `addresses`, because all variables are assignable by default. On the other hand, programs in these languages can be tougher to reason about because it can be hard to determine which variables change

[5] We shall see in Section 17.7 that compilers often perform a program transformation like this called **assignment conversion.**

over time and which do not. This situation can be improved if the languages provide a mechanism for declaring that certain named entities are immutable (e.g., constant declarations).

We have two motivations for studying mutable variables:

- Many real languages support mutable variables.
- Mutable variables shift the way we think about naming. In languages with mutable variables, names do not denote values, but instead denote locations in the store at which values are stored.

8.3.2 FLAVAR!

We will study mutable variables in FLAVAR!, a dialect of FL! that supports assignments to variables. The syntax of FLAVAR! (and its kernel, FLAVARK!) is the same as that for FL! (and its kernel, FLK!) except for the addition of a SCHEME-like `set!` construct:

$E_{FLAVARK!}$ ::= ... [FLK! Expressions]
| `(set!` *I E*`)` [Assignment]

Informally, `(set!` *I E*`)` assigns the value of the expression *E* to the variable named by *I*. For example,

```
(let ((a 3))
  (begin (set! a 4)
         a))
```
$\xrightarrow[FLAVAR!]{}$ *4*

Note the differences between the cell assignment operator, `cell-set!`, and the variable assignment construct, `set!`. The former changes the value of a first-class data value (a cell), while the latter changes the value of a variable (which is not a first-class value). In `(cell-set!` E_1 E_2`)`, E_1 can be any expression that evaluates to a cell, while in `(set!` *I E*`)`, *I* is constrained to be an identifier visible in the current scope. Mutable cells and mutable variables are orthogonal language features. FLAVAR! contains both.

The semantics of FLAVARK! is based on the denotational semantics of FLK! presented in the previous section. We will only note the ways in which the semantics for FLAVARK! differs from that for FLK!. Some of the differences are highlighted in Figure 8.16. The key feature of FLAVARK! is that variables, like mutable cells, are represented as locations in the store. This means that locations are the only entity in the language that can be named; i.e., *Denotable* = *Location*. The association between a name *I* and a value *v* that is represented by a single environment binding in FL! is represented by *two* bindings in FLAVAR!:

$\delta \in \mathit{Denotable} = \mathit{Location}$
$\sigma \in \mathit{Storable} = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$\mathit{val\text{-}to\text{-}storable} : \mathit{Value} \to \mathit{Storable} = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$\mathcal{E}[\![I]\!] = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \mathit{depends\ on\ parameter\ passing\ mechanism}$

$$\mathcal{E}[\![\texttt{(set! } I \ E\texttt{)}]\!] = \lambda e\,.\ (\mathit{with\text{-}value}\ \ (\mathcal{E}[\![E]\!]\ e)\ (\lambda v\,.\ (\mathit{with\text{-}denotable}\ \ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\ (\mathit{update}\ l\ \ (\mathit{val\text{-}to\text{-}storable}\ v))))))$$

Figure 8.16: Semantics of mutable variables. The definitions of *Storable*, *val-to-storable*, and the valuation clauses for I and `call` depend on the parameter passing mechanism.

an environment binding between a name I and a location l, and an assignment between l and v. The indirection through l allows the value associated with the name to be changed. The details of how the locations are allocated, how they are looked up, and what values may legally be stored in them are determined by the parameter passing mechanism of the language. We shall discuss several mechanisms shortly.

The other interesting aspect of the FLAVAR! semantics is the valuation clause for `set!`. In `(set!` I E`)`, E is evaluated and stored in the location named by I. The auxiliary function *val-to-storable*, which depends on the definition of *Storable*, is needed to inject the value into the *Storable* domain. Note that in the expression `(set! a a)`, the left and right occurrences of `a` are treated differently. A location is found for the left occurrence, but the value stored at that location is found for the right occurrence. For this reason, the location is called the **L-value** (left value) of the variable, and the value stored at that location is called the **R-value** (right value) of the variable. Determining the R-value associated with an L-value is called **dereferencing** the variable. The notions of L-value and R-value can be extended to expressions. Variables can be viewed as cells in which dereferencing corresponds to automatically performing a `cell-ref` upon every variable reference, and `(set!` I E`)` performs a `cell-set!` of the L-value of I to the R-value of E.

8.3.3 Parameter Passing Mechanisms for FLAVAR!

Parameter passing mechanisms for languages with mutable variables are determined by the domain *Storable*, the function *val-to-storable*, and the valuation clauses for `call` and *I*. Figures 8.17 and 8.18 summarize four parameter passing mechanisms for FLAVAR!. These are explained in the following sections.

$\sigma \in \mathit{Storable} = \mathit{Value}$

$\mathit{val\text{-}to\text{-}storable} = \lambda v\,.\,v$

$$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \lambda e\,.\,(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\,(\mathit{with\text{-}value}\ (\mathcal{E}[\![E_2]\!]\ e)\ (\lambda v\,.\,(\mathit{allocating}\ v\ p)))))$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{fetching}\ l\ \mathit{val\text{-}to\text{-}comp})))$$

Call-by-Value

$\sigma \in \mathit{Storable} = \mathit{Computation}$

$\mathit{val\text{-}to\text{-}storable} = \mathit{val\text{-}to\text{-}comp}$

$$\mathcal{E}[\![\texttt{(call } E_1 \ E_2\texttt{)}]\!] = \lambda e\,.\,(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e)\ (\lambda p\,.\,(\mathit{allocating}\ (\mathcal{E}[\![E_2]\!]\ e)\ p)))$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{fetching}\ l\ (\lambda c\,.\,c))))$$

Call-by-Name

Figure 8.17: Parameter passing mechanisms in FLAVAR!, part I.

8.3.3.1 Call-by-value

The CBV mechanism for FLAVAR! is similar to CBV for FL and FLK! except that a procedure call allocates a new location for the argument value and passes this location (rather than the value) to the procedure. Since the meaning of an identifier is a location and not a value, every variable reference requires both a lookup in the environment (to find the location) and a fetch from the store (to dereference the location). In CBV, only elements of the domain *Value* are storable. For example:

$$\begin{array}{rcll} \sigma & \in & \mathit{Storable} & = & \mathit{Memo} \\ \mathit{mm} & \in & \mathit{Memo} & = & \mathit{Computation} + \mathit{Value} \end{array}$$

$\mathit{val\text{-}to\text{-}storable} = \lambda v\,.\,(\mathit{Value} \mapsto \mathit{Memo}\ v)$

$$\begin{aligned} \mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,&(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e) \\ &(\lambda p\,.\,(\mathit{allocating}\ (\mathit{Computation} \mapsto \mathit{Memo}\ (\mathcal{E}[\![E_2]\!]\ e))\ p))) \end{aligned}$$

$$\begin{aligned} \mathcal{E}[\![I]\!] = \lambda e\,.\,&(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I) \\ &(\lambda l\,.\,(\mathit{fetching}\ l \\ &\quad(\lambda \mathit{mm}\,.\ \textbf{matching}\ \mathit{mm} \\ &\quad\quad\triangleright\,(\mathit{Computation} \mapsto \mathit{Memo}\ c) \\ &\quad\quad\ [\!]\ (\mathit{with\text{-}value}\ c \\ &\quad\quad\quad(\lambda v\,.\,(\mathit{sequence}\ (\mathit{update}\ l\ (\mathit{Value} \mapsto \mathit{Memo}\ v)) \\ &\quad\quad\quad\quad(\mathit{val\text{-}to\text{-}comp}\ v)))) \\ &\quad\quad\triangleright\,(\mathit{Value} \mapsto \mathit{Memo}\ v)\ [\!]\ (\mathit{val\text{-}to\text{-}comp}\ v) \\ &\quad\quad\textbf{endmatching}\ )))) \end{aligned}$$

Call-by-Need (Lazy Evaluation)

$$\sigma \in \mathit{Storable} = \mathit{Value}$$

$\mathcal{E} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation}$
$\mathcal{LV} : \mathrm{Exp} \to \mathit{Environment} \to \mathit{Computation}$

$\mathit{val\text{-}to\text{-}storable} = \lambda v\,.\,v$

$$\begin{aligned} \mathcal{E}[\![\texttt{(call } E_1\ E_2\texttt{)}]\!] = \lambda e\,.\,&(\mathit{with\text{-}procedure}\ (\mathcal{E}[\![E_1]\!]\ e) \\ &(\lambda p\,.\,(\mathit{with\text{-}location}\ (\mathcal{LV}[\![E_2]\!]\ e)\ p))) \end{aligned}$$

$$\mathcal{E}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{fetching}\ l\ \mathit{val\text{-}to\text{-}comp})))$$

$$\mathcal{LV}[\![I]\!] = \lambda e\,.\,(\mathit{with\text{-}denotable}\ (\mathit{lookup}\ e\ I)\ (\lambda l\,.\,(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l))))$$

$$\begin{aligned} &\mathcal{LV}[\![E_{other}]\!]\ ;\ \text{where } E_{other} \text{ is not } I \\ &= \lambda e\,.\,(\mathit{with\text{-}value}\ (\mathcal{E}[\![E_{other}]\!]\ e) \\ &\qquad(\lambda v\,.\,(\mathit{allocating}\ v\ (\lambda l\,.\,(\mathit{val\text{-}to\text{-}comp}\ (\mathit{Location} \mapsto \mathit{Value}\ l))))) \end{aligned}$$

Call-by-Reference

Figure 8.18: Parameter passing mechanisms in FLAVAR!, part II.

```
(let ((a 0)
      (f (lambda (x) (+ x x))))
  (f (begin (set! a (+ a 1)) a)))
```
$\xrightarrow[CBV\ \ FLAVAR!]{}$ *2*

```
((lambda (x) 3) (/ 1 0))
```
$\xrightarrow[CBV\ \ FLAVAR!]{}$ *error*

8.3.3.2 Call-by-name

CBN in FLAVAR! is similar to CBN in FL, except that here it is *Storable* (not *Denotable*) that equals *Computation*. The `call` clause indicates that the computation of the argument expression (not its value) is stored at a newly allocated location. In FLAVAR!, computations are functions that accept a store, so the current store is supplied to a computation every time the variable that names it is referenced. If the computation performs a side effect, this side effect will be performed every time the variable is looked up. Consider the following example:

```
(let ((a 0)
      (f (lambda (x) (+ x x))))
  (f (begin (set! a (+ a 1)) a)))
```
$\xrightarrow[CBN\ \ FLAVAR!]{}$ *3*

In the example, calling `f` binds `x` to a location that holds the computation ($\mathcal{E}[\![$`(begin (set! a (+ a 1)) a)`$]\!]\ e_1$), where e_1 is an environment with bindings for `a` and `f`. Each variable reference to `x` within the procedure body `(+ x x)` performs this computation with the current store. So the left reference to `x` increments `a` and returns 1, while the right reference to `x` increments `a` again and returns 2. This behavior illustrates the perils of mixing state with CBN parameter passing.

As in FL, certain computations in FLAVAR! correspond to errors or nontermination. Because such computations are nameable in CBN (by an indirection through the store), procedures can be non-strict:

```
((lambda (x) 3) (/ 1 0))
```
$\xrightarrow[CBN\ \ FLAVAR!]{}$ *3*

8.3.3.3 Call-by-need (Lazy Evaluation)

The presence of state in FLAVAR! suggests a parameter passing mechanism based on the memoization trick introduced in the FL interpreter. That is, a formal parameter name can be bound to a location that originally stores the computation of the argument expression. The first time the parameter is referenced, the computation is performed, but the resulting value is cached at the location and is used on every subsequent reference. Thus, the argument expression is evaluated at most once and is never evaluated at all if the parameter

is never referenced. This mechanism is called **call-by-need** or **lazy evaluation**. Because the acronym CBN is already taken, we will abbreviate call-by-need as CBL (call-by-lazy).

Call-by-need can exhibit the desirable behavior of both CBV and CBN:

```
(let ((a 0)
      (f (lambda (x) (+ x x))))
  (f (begin (set! a (+ a 1)) a)))
```
$\xrightarrow[CBL\ \ FLAVAR!]{}$ *2*

```
((lambda (x) 3) (/ 1 0))
```
$\xrightarrow[CBL\ \ FLAVAR!]{}$ *3*

However, because side effects in argument expressions are performed at the time of lookup rather than at the time of call, CBL can exhibit different behavior from CBV. For example, consider the following expression:

```
(let ((a 0))
  (let ((f (lambda (x)
             (begin (set! a 17)
                    (+ x x)))))
    (f (begin (set! a (+ a 1)) a))))
```

Under CBV, the call to `f` first increments `a` and then binds `x` to a location holding `1`. The assignment of `17` to `a` does not affect `x`, so the result is *2*. However, under CBL, the call to `f` binds `x` to a location that holds the computation of `(begin (set! a (+ a 1)) a)`. This computation is not performed until the first reference of `x`, which occurs *after* `a` has been set to `17`. So CBL returns *36* for this expression.

8.3.3.4 Call-by-reference

So far, all the parameter passing mechanisms we have discussed allocate a new location for every argument. But in the case where the argument expression is a variable reference, there is already a location associated with the variable. This suggests a mechanism that uses the existing location rather than allocating a new one. Such a mechanism is termed **call-by-reference (CBR)**. FORTRAN and PASCAL and are examples of languages that support CBR.

In CBR, there is the question of what to do with an argument that is not a manifest identifier. For example, in the application `(test (+ 1 2))`, the value of `(+ 1 2)` has no associated location. Languages handle this situation in different ways. In PASCAL, it is an error to supply anything but an identifier as a CBR argument. In FORTRAN, however, a new location will be allocated for any argument that is not a manifest identifier. The semantics in Figure 8.18 takes this latter approach. In fact, this is the *only* mechanism for creating

new variables in CBR FLAVAR!. This is a somewhat unrealistic aspect of our language; real CBR languages have special declarations for introducing new variables.

The denotational semantics for CBR models the special handling of variable arguments by providing two valuation functions for expressions: $\mathcal{E}$ and $\mathcal{LV}$. $\mathcal{LV}$ finds the L-value of an expression, while $\mathcal{E}$ finds the R-value of an expression. For an expression that is an identifier, $\mathcal{LV}$ returns the location of that identifier. For any other expression, $\mathcal{LV}$ allocates a new location for the R-value of the expression and returns this location. The key feature of the CBR semantics is that $\mathcal{LV}$ (rather than $\mathcal{E}$) is used to evaluate the operand of a procedure application.

In FLAVAR!, procedure calls are expressions that return results, but in many imperative languages, procedure calls are commands that do not return results. In such languages, CBR is useful as a means of extracting a result from a procedure call. One (or more) arguments to a procedure can be a variable that the procedure uses to communicate the result(s) back to the caller. Here is an example of this idiom in CBR FLAVAR!:

```
(let ((a 0)
      (double (lambda (in out)
                (set! out (+ in in)))))
  (begin
   ;; A is 0 here.
   (double 17 a)
   ;; Now A is 34.
   (+ a 1))) ⟶ 35
```

The `double` procedure takes a numeric argument (`in`) and variable (`out`) for returning the result of doubling `in`. In the example, the variable `a` is used to communicate the result of the doubling operation back to the point of call.

One characteristic of CBR (or any paradigm that allows mutable entities to be passed as arguments) is that two different names may refer to the same location. This situation is known as **aliasing**. Consider the following example:

```
(let ((x 1))
  (let ((test (lambda (a)
                (begin
                  (set! x 20)
                  (+ a x)))))
    (test x))) ⟶ 40
```

Within the call `(test x)`, both `x` and `a` are aliases for the same location, so the assignment to `x` changes `a`. Aliasing is often considered undesirable because it complicates reasoning about programs.

CBR is similar to passing a mutable cell as an argument to a procedure. The difference is that variables are more restricted than cells. A mutable cell is a first-class value: it may be named, passed as an argument to a procedure, returned as a result from a procedure, and stored in any data structure, including another cell. On the other hand, while a variable may be named by an identifier and passed as an argument to a procedure, it cannot be returned as a result from a procedure, and it cannot be stored in a data structure (including another variable). Unlike cells, therefore, variables are not first-class values. Although this restricts the expressive power of variables, it permits variables to be implemented more efficiently than cells. A variable may be allocated on a stack, while cells generally must be allocated from a garbage-collected heap. We will have much more to say about tradeoffs between expressiveness and efficiency when we study pragmatic issues later on.

▷ **Exercise 8.11**

a. Give a translation of call-by-value FLAVARK! into call-by-value FLK!. You do not need to translate `rec`.

b. Give a translation of call-by-reference FLAVARK! into call-by-value FLK!. You do not need to translate `rec`. ◁