Chapter 14

Variables

In Section 13 we studied the implementation of mutable data structures. The boxes we studied there could just as well have been vectors or other container types, such as objects with fields.

In traditional languages like C and Java, there are actually two forms of mutation. One is mutation to a container value, such as an object (in Java). A program expression such as

\[ o.f = e \]

evaluates \( o \) to an object, \( e \) to some value, and mutates the field \( f \) of the former object to hold the result of evaluating \( e \) instead. Note that \( o \) can be an arbitrary expression (for instance, it can look up an object in some other data structure) that is evaluated to a value. In contrast, a programmer can also write a method such as

```java
void m (int i) {
    i = 4;
}
```

Here, \( i \) must literally be an identifier; it cannot be an arbitrary expression that evaluates to an identifier. That is, we are not mutating the value contained within a box (or position in a vector, or a field); rather, we are mutating the value bound to an identifier itself. That makes the identifier a variable.

14.1 Implementing Variables

First, let’s extend our language to include variables:

```latex
\texttt{<MCFAE>} ::= \texttt{<num>}
| \{+ \texttt{<MCFAE>} \texttt{<MCFAE>}\}
| \{- \texttt{<MCFAE>} \texttt{<MCFAE>}\}
| \texttt{id}
| \{\texttt{fun} \{\texttt{id}\} \texttt{<MCFAE>}\}
| \{\texttt{FWAE} \texttt{<FWAE>}\}
| \{\texttt{if0} \texttt{<MCFAE>} \texttt{<MCFAE>} \texttt{<MCFAE>}\}
```
Observe that the set expression expects a literal identifier after the keyword.

Implementing variables is a little different from implementing boxes. In the latter case, we first evaluate the position that identifies the box:

\[
\text{setbox} (\text{box-expr} \ \text{value-expr})
\]

\[
\begin{align*}
\text{(type-case Value} & \times \text{Store} \\
& \text{interp box-expr sc store} \\
& [v \times s (\text{box-value box-store})] \\
& :])
\end{align*}
\]

In contrast, in a language with variables, identifiers do not represent boxes. Therefore, the corresponding code:

\[
\text{set} (\text{var value})
\]

\[
\begin{align*}
\text{(type-case Value} & \times \text{Store} \\
& \text{interp var sc store} \\
& [v \times s (\text{var-value var-store})] \\
& :])
\end{align*}
\]

would be counter-productive. Evaluating the identifier would result in a value, which we cannot mutate, as opposed to a location, which we can by updating the store.

This immediately suggests a slightly different evaluation strategy:

\[
\text{set} (\text{var value})
\]

\[
\begin{align*}
\text{(type-case Value} & \times \text{Store} (\text{interp value sc store}) \\
& [v \times s (\text{value-value value-store})] \\
& \text{(local} ([\text{define the-loc} (\text{env-lookup var sc})]) \\
& :)])
\end{align*}
\]

That is, we evaluate the expression that represents the new value to be stored in the interpreter. Instead of evaluating the identifier, however, we only look it up in the environment. This results in a location where the new value should be stored. In particular, notice an unusual pattern: the interpreter dereferences the identifier in the environment, but does not dereference the result (the identifier’s location) in the store. We have not seen this pattern before, and will not see it elsewhere.

Many languages make a distinction between mutable data structures and mutable identifiers. When a mutable identifier appears in the assignment position of an assignment operator (many languages use the same syntactic operator, = or :=, to represent both operations), the language implementation only partially resolves the identifier. This special kind of value—the location of an identifier—is traditionally known as an l-value.

Whence this unusual name? Consider the following two statements in C:

\[
\begin{align*}
& x = 2; \\
& y = x;
\end{align*}
\]
In the second statement, \( x \) must be reduced to a value—i.e., \( \text{store-lookup} \) and \( \text{env-lookup} \) must be composed and applied to its content—whereas in the second statement, \( x \) must only be reduced to a location, not to a value. In languages where locations are not values (more on that below), this odd kind of “value” is known as an “l-value”, since it appears only on the left-hand-side of assignment statements.

Given this insight, we can now easily complete the definition of the assignment statement:

\[
\text{set (var value)}
\]

\[
\text{(type-case} \text{Valuel\times Store (interp value sc store)}
\]

\[
[v\times s (value-value value-store)]
\]

\[
(\text{local} ([\text{define the-loc (env-lookup var sc)}])
\]

\[
(v\times s value-value)
\]

\[
(a\text{Sto the-loc value-value value-store}))
\]

The rest of the interpreter remains unchanged. Note, in particular, that it still employs store-passing style. Figure 14.1 and Figure 14.2 present the core of the interpreter.

### 14.2 Interaction Between Variables and Function Application

Variables and function application appear to be two independent language features, but perhaps they are not. Consider the following program:

\[
\{\text{with } \{v \ 0\} \ \\
\{\text{with } \{f \ \{\text{fun } \{y\} \ \\
\{\text{set } y \ 5\}\}\} \ \\
\{\text{seqn } \{f \ v\} \ \\
\ v\}\}\}
\]

What do we expect it to evaluate to? There are two different, reasonable answers: 0 and 5. The first assumes that the mutation is to the formal variable, \( y \), and does not affect the actual argument, \( v \); the second assumes that this mutation does have the effect of modifying the actual argument.

Our current implementation yields the value 0. This is because the act of invoking the function binds the formal parameter to a new location:

\[
(\text{local} ([\text{define new-loc (next-location arg-store)}])
\]

\[
(\text{interp} (\text{closureV-body fun-value})
\]

\[
(a\text{Sub (closureV-param fun-value)}
\]

\[
\text{new-loc}
\]

\[
(\text{ closureV-sc fun-value}))
\]

\[
(a\text{Sto new-loc}
\]

\[
\text{arg-value}
\]

\[
\text{arg-store}))
\]

The evaluated argument is held in this new location. Therefore, changes to the content of that location in the store do not affect the actual parameter.
Let’s now explore the alternative. This form of evaluation is called call-by-reference, in contrast to the “eager” technique we have studied thusfar, which is known as call-by-value. This new technique gets its name because we will pass a reference to the actual argument, rather than merely its value. Thus, updates to the reference within the called procedure will become visible to the calling context, too.

To explore this design, let’s extend our language further so we have two kinds of procedures: call-by-value (fun) and call-by-reference (refun):

\[
\text{<RMCFAE>} ::= \text{<num>} \\
| {+ \text{<RMCFAE>} \text{<RMCFAE>}} \\
| {- \text{<RMCFAE>} \text{<RMCFAE>}} \\
| \text{id} \\
| \text{fun \{id\} <RMCFAE>} \\
| \text{refun \{id\} <RMCFAE>} \\
| \{\text{FWAE} \text{<FWAE>}} \\
| \{\text{if0 <RMCFAE> <RMCFAE> <RMCFAE>}\} \\
| \{\text{set id <RMCFAE>}\} \\
| \{\text{seqn <RMCFAE> <RMCFAE>}\} \\
| \{\text{<RMCFAE> <RMCFAE>}\}
\]

That is, syntactically a call-by-reference procedure looks the same as a call-by-value procedure other than the distinguishing keyword. It is their interpretation that will distinguish them.

All the code we have developed thusfar remains the same for call-by-value procedure invocation. In particular, with expressions should continue to expand into immediate applications of fun-defined procedures. Let us proceed to defining the interpretation of call-by-reference procedures.

The first step is to evaluate a reference procedure definition. This is straightforward:

\[
\text{refun (bound-id bound-body)} \\
(v \times s \text{(refclosV bound-id bound-body sc store)})
\]

We create a new kind of closure so we can later distinguish what kind of procedure we are about to apply, but its fields are the same:

\[
\text{(define-type RMCFAE-Value} \\
\text{[numV (n number?)]} \\
\text{[closureV (param symbol?)} \\
\text{ (body RMCFAE?)} \\
\text{ (sc SubCache?)])] \\
\text{[refclosV (param symbol?)} \\
\text{ (body RMCFAE?)} \\
\text{ (sc SubCache?)])}
\]

Now let us study the interpretation of application. After evaluating the procedure position, we must check which kind of procedure it is before evaluating the argument. If it’s a call-by-value procedure, we proceed as before:

\[
\text{app (fun-expr arg-expr)}
\]
14.3 Perspective

Should languages have reference procedures? Passing references to procedures has the following dangerous property: the formal parameter becomes an alias of the actual parameter, as all changes to the formal
manifest as changes to the actual also. This is especially insidious because the programmer may not know he is about to apply a reference procedure: Some languages like C offer the ability to mark specific parameters of multi-parameter procedures with keywords such as & and ref, meaning they alone should be passed by reference (these are known as reference parameters). The client of such a procedure may thus find that, mysteriously, the act of invoking this procedure has changed the value of his identifiers. This aliasing effect can lead to errors that are particularly difficult to detect and diagnose.

This phenomenon cannot occur with call-by-value: changes to the variable in the called procedure do not affect the caller. There is, therefore, nearly universal agreement in modern languages that arguments should be passed by value. If the called procedure intends to mutate a value, it must consume a box (or other container data structure); the caller must, in turn, signal acceptance of this behavior by passing a box as the actual argument. The caller then has the freedom to inspect the content of the (possibly mutated) box and determine whether to accept this mutation in the remainder of the program, or to reject it by ignoring the altered content of the box.

Why did languages introduce reference parameters? For one thing, they are “cheaper”: they do not require additional allocation. (We can see this difference clearly when we contrast the two kinds of procedure application.) However, the problems they introduce arguably far outweigh this small savings in memory (which can anyway be reclaimed by modern memory management systems).

Reference parameters do, however, also confer a small expressiveness benefit. Without reference parameters, we cannot define a procedure that swaps the content of two variables. In the following code,

```plaintext
{with {swap {fun {x}
  {fun {y}
    {with {z x}
      {seqn {set x y}
        {set y z}))))}

{with {a 3}
  {with {b 2}
    {seqn {{swap a} b}
      b))}}
```

the result of the computation is still 2, because the mutations to x and y inside the procedure do not affect a and b. In contrast,

```plaintext
{with {swap {refun {x}
  {refun {y}
    {with {z x}
      {seqn {set x y}
        {set y z}))))}

{with {a 3}
  {with {b 2}
    {seqn {{swap a} b}
      b))}}
```

results in the value 3: since x and y are just aliases to a and b, mutations to the former are reflected as
mutations to the latter. (Note that both procedures must be refuns and not funs, else the swap is at best partial.)

This example also, however, illustrates why aliasing can cause problems. The implementor of the procedure may have used mutation accidentally, without meaning to affect the caller. The procedure boundary abstraction has, however, been compromised by the aliasing, and accidental side-effects can leak into the calling contexts, exposing unnecessary implementation details of the procedure.

In the early days of programming language design, before programs were particularly sophisticated, the ability to write simple abstractions such as swap was considered valuable (since it is used, for instance, in the implementation of some sorting algorithms). Today, however, we recognize that such abstractions are rather meager in the face of the needs of modern systems. We pay greater attention, instead, to the need for creating useful abstraction boundaries between units of modularity such as procedures: the fewer hidden interactions they have, and the less they interfere with one another, the more easily we can reason about their behavior in isolation.

Exercise 14.3.1 While call-by-value preserves the value of variables in the calling context, it does not protect all values. In particular, in many call-by-value languages, a composite data structure (such as a vector) passed as an argument may be mutated by the callee, with the effects visible to the caller.

1. Does this behavior contradict the claim that the language is passing “values” as arguments? Use our investigation of mutable data structures in Section 13 to make your argument rigorous.
   Hint: Implement an interpreter for a language with both boxes and call-by-reference application, then argue about similarities and differences.

2. Languages like ML tackle this problem by forcing programmers to annotate all mutable data structures using references, the ML counterpart to boxes. Any data structure not so mutated is considered immutable. What trade-offs does ML’s design introduce?

Exercise 14.3.2 There appears to be a neutral ground between call-by-value and call-by-reference. Consider the following proposed syntax:

```plaintext
{with {swap {fun {x}
    {fun {y}
        {with {z x}
            {seqn {set x y}
                {set y z}}}{}
    {with {a 3}
        {with {b 2}
            {seqn {{swap {ref a}} {ref b}}
                b}}}}}}
```

The ref notation is an indicator to the interpreter to pass the variable’s location rather than its value; that is, by using {ref a} and {ref b}, the invoker of the procedure indicates his willingness to have his variables be aliased and thus, potentially, be mutated.

1. Modify the interpreter to support the use of ref for procedure arguments.
2. *Does this proposal result in a procedural abstraction of the process of swapping the values of two variables?* If it does, this would reconcile the design tension between the two invocation techniques: it avoids the difficulty of call-by-value (the inability to write a swap procedure) as well as that of call-by-reference (aliasing of parameters without the caller’s knowledge). Discuss.

3. *Suppose programmers are allowed to apply \( \texttt{ref} \) to variables elsewhere in the program. What type should the interpreter use to represent the resulting value? How does this compare to an l-value? Does this introduce the need for additional operators in the language? How does this relate to the \& operator in C?*
14.3. PERSPECTIVE

;; interp : MCFAE SubCache Store → Value×Store

(define (interp expr sc store)
  (type-case MCFAE expr
    [num (n) (v×s (numV n) store)]
    [add (l r)
     (type-case Value×Store (interp l sc store)
      (v×s (l-value l-store)
        (type-case Value×Store (interp r sc l-store)
         (v×s (r-value r-store)
           (v×s (num+ l-value r-value)
             r-store)))))
     [sub (l r)
      (type-case Value×Store (interp l sc store)
       (v×s (l-value l-store)
         (type-case Value×Store (interp r sc l-store)
          (v×s (r-value r-store)
            (v×s (num− l-value r-value)
              r-store)))))
     [id (v) (v×s (store-lookup (env-lookup v sc) store))]
     [fun (bound-id bound-body)
      (v×s (closureV bound-id bound-body sc))]
     [app (fun-expr arg-expr)
      (type-case Value×Store (interp fun-expr sc store)
       (v×s (fun-value fun-store)
         (type-case Value×Store (interp arg-expr sc fun-store)
          (v×s (arg-value arg-store)
            (local ([define new-loc (next-location arg-store)])
             (interp (closureV-body fun-value)
              (aSub (closureV-param fun-value)
               new-loc
               (closureV-sc fun-value))
              (aSto new-loc
               arg-value
               arg-store)))])))]
    ...)

Figure 14.1: Implementing Variables, Part 1
\[\text{if0 } (\text{test pass fail})\]
\[
\text{(type-case Value} \times \text{Store } (\text{interp test sc store}) \\
\quad [v \times s (\text{test-value test-store}) \\
\quad \quad (\text{if } (\text{num-zero? test-value}) \\
\quad \quad \quad (\text{interp pass sc test-store}) \\
\quad \quad \quad (\text{interp fail sc test-store})))])
\]

\[\text{set } (\text{var value})\]
\[
\text{(type-case Value} \times \text{Store } (\text{interp value sc store}) \\
\quad [v \times s (\text{value-value value-store}) \\
\quad \quad (\text{local } [(\text{define the-loc (env-lookup var sc)}) \\
\quad \quad \quad (v \times s \text{ value-value} \\
\quad \quad \quad \quad (\text{aSto the-loc value-value value-store})))])]
\]

\[\text{seqn } (e1 \ e2)\]
\[
\text{(type-case Value} \times \text{Store } (\text{interp e1 sc store}) \\
\quad [v \times s (\text{e1-value e1-store}) \\
\quad \quad (\text{interp e2 sc e1-store})))])
\]

Figure 14.2: Implementing Variables, Part 2
14.3. PERSPECTIVE

(define-type RMCFAE-Value
  [numV (n number?)]
  [closureV (param symbol?)
    (body RMCFAE?)
    (sc SubCache?)]
  [refclosV (param symbol?)
    (body RMCFAE?)
    (sc SubCache?)])

;; interp : RMCFAE SubCache Store → Value×Store

(define (interp expr sc store)
  (type-case RMCFAE expr
    [num (n) (v×s (numV n) store)]
    [add (l r)
      (type-case Value×Store (interp l sc store)
        [v×s (l-value l-store)
          (type-case Value×Store (interp r sc l-store)
            [v×s (r-value r-store)
              (v×s (num+ l-value r-value)
                r-store))])])
    [sub (l r)
      (type-case Value×Store (interp l sc store)
        [v×s (l-value l-store)
          (type-case Value×Store (interp r sc l-store)
            [v×s (r-value r-store)
              (v×s (num- l-value r-value)
                r-store))])])
    [id (v) (v×s (store-lookup (env-lookup v sc) store) store)]
    [if0 (test pass fail)
      (type-case Value×Store (interp test sc store)
        [v×s (test-value test-store)
          (if (num-zero? test-value)
            (interp pass sc test-store)
            (interp fail sc test-store))])])
  ...

Figure 14.3: Implementing Call-by-Reference, Part 1
Figure 14.4: Implementing Call-by-Reference, Part 2