15
Mutable state and imperative programming

The range of programming abstractions presented so far – first-order and higher-order functions; strong, static typing; polymorphism; algebraic data types; modules and functors – all fall squarely within a view of functional programming that we might term PURE, in which computation is identified solely with the evaluation of expressions. Pure programming has to do with what expressions *are*, not what they *do*. Pure programs have *values* rather than *effects*. Indeed, the slightly pejorative term SIDE EFFECT is used in the functional programming literature for effects that impure programs manifest while they are being evaluated beyond their values themselves.

In a pure functional programming language, there are no side effects. Computation can be thought of as simplifying expressions to their values by repeated substitution of equals for equals. Because this notion of program meaning is so straightforward, functional programs are easier to reason about. Hopefully, the preceding chapters have shown that the functional paradigm is also more powerful than you might have thought.

Strictly speaking, however, pure functional programming is pointless. We write code to have an *effect* on the world. It might be pretty to think that "side effects" aren't the main point. But they're the main point.

Take this simple computation of the twentieth Fibonacci number:

```
# let rec fib n =
#   if n <= 1 then 1
#   else fib (n - 1) + fib (n - 2) ;;
val fib : int -> int = <fun>

# fib 20 ;;
- : int = 10946
```

The computation of `fib 20` proceeds purely functionally – at least until that very last step where the OCaml REPL *prints out the computed value*. Printing is *the* quintessential side effect. It's a thing that

a program does, not a value that a program has. Without that one side effect, the `fib` computation would be useless. We'd gain no information from it.

So we need at least a *little* impurity in any programming system. But there are some algorithms that actually require impurity, in the form of side effects that change state. For instance, we've seen implementation of a dictionary data type in Chapter 12. That implementation allowed for linear time insertion and linear time lookup. More efficient implementations allow for constant time insertion and linear lookup (or vice versa) or for logarithmic insertion and lookup. But by taking advantage of side effects that change state, we can implement mutable dictionaries that achieve constant time insertion and constant time lookup, for instance, with hash tables. (In fact, we do so in Section 15.6.)

In this chapter and the next, we introduce IMPERATIVE PROGRAMMING, a programming paradigm based on side effects and state change. We start with mutable data structures, moving on to imperative control structures in the next chapter.

In the pure part of OCaml, we don't change the state of the computation, as encoded in the computer's memory. In languages that have mutable state, variables name blocks of memory whose contents can change. Assigning a new value to such a variable mutates the memory, changing its state by replacing the original value with the new one. OCaml variables, by contrast, aren't mutable. They name values, and once having named a value, the value named doesn't change.

You might think that OCaml does allow changing the value of a variable. What about, for instance, a global renaming of a variable?

```
# let x = 42 ;;
val x : int = 42
# x ;;            (* x is 42 *)
- : int = 42
# let x = 21 ;;
val x : int = 21
# x ;;            (* ...but now it's 21 *)
- : int = 21
```

Hasn't the value of `x` changed from `42` to `21`?

No, it hasn't. Rather, there are two separate variables that happen to both have the same name, `x`. In the second expression, we are referring to the first `x` variable. In the fourth expression, we are referring to the second `x` variable, which shadows the first one. But the first `x` is still there. We can tell by the following experiment:

```
# let x = 42 ;;   (* establishing first x... *)
val x : int = 42
# x ;;            (* ...whose value is 42 *)
- : int = 42
```

```
# let f () = x ;; (* f returns value in first x *)
val f : unit -> int = <fun>
# let x = 21 ;;   (* establishing second x... *)
val x : int = 21
# x ;;            (* ...with a different value *)
- : int = 21
# f () ;;         (* but f still references first x *)
- : int = 42
```

The definition of the function `f` makes use of the first variable `x`, simply by returning its value when called. Even if we add a new `x` naming a different value, the application `f ()` still returns `42`, the value that the first variable `x` names, thereby showing that the first `x` is still available.

The `let` naming constructs of OCaml thus don't provide for mutable state. If we want to make use of mutable state, for instance for the purpose of building mutable data structures, we'll need new constructs. OCaml provides references for this purpose.

15.1 References

The OCaml language provides an abstract notion of REFERENCE to a block of mutable memory with its REFERENCE TYPES. To maintain the type discipline of the language, we want to keep track of the type of thing stored in the block; although the particular value stored there may change, we don't want its type to vary. Thus, we have separate types for references to integers, references to strings, references to functions from booleans to integers, and so forth. The postfix type constructor `ref` is used to construct reference types: `int ref`, `string ref`, `(bool -> int) ref`, and the like.

To create a value of some reference type, OCaml provides the prefix value constructor `ref`.[1] The supplied expression must be of the type appropriate for the reference type, and the value of that expression is stored as the initial value in the block of memory that the reference references. Here, for instance, we create a reference to a block of memory storing the integer value 42:

```
# let r : int ref = ref 42 ;;
val r : int ref = {contents = 42}
```

As with all variables, `r` is an immutable name, but it is a name for a block of memory that is itself mutable. (The value is printed as `{contents = 42}` for reasons that we allude to in Section 15.2.1.)

The natural operations to perform on a reference value are two: first, DEREFERENCE, that is, retrieve the value stored in the referenced block; and second, UPDATE, modify the value stored in the referenced block (with a value of the same type, of course). Dereferencing is done with the prefix `!` operator, and updating with the infix `:=` operator.

[1] Yes, the same symbol, `ref`, is used at the type level for the type constructor and at the value level for the value constructor. And to make matters more confusing, the type constructor is postfix while the value constructor is prefix. Learning the concrete syntax of a new programming language sure can be frustrating.

```
# !r ;;
- : int = 42
# r := 21 ;;
- : unit = ()
# !r ;;
- : int = 21
```

Here, we've dereferenced the same variable `r` twice (in the two highlighted expressions), getting two different values – first `42`, then `21`. This is quite different from the example with two `x` variables. Here, there is only one variable `r`, and yet a single expression `!r` involving `r` whose value has changed![2]

[2] But like all variables, `r` has not itself changed its value. It still points to the same block of memory.

This example puts in sharp relief the difference between the pure language and the impure. In the pure language, an expression in a given lexical context (that is, the set of variable names that are available) always evaluates to the same value. But in this example, two instances of the expression `!r` evaluate to two different values, even though the same `r` is used in both instances of the expression. The assignment has the side effect of changing what value is stored in the block that `r` references, so that reevaluating `!r` to retrieve the stored value finds a different integer.

The expression causing the side effect here was easy to spot. But in general, these side effects could happen as the result of a series of function calls quite obscure from the code that manifests the side effect. This property of side effects can make it difficult to reason about what value an expression has.

In particular, the substitution semantics of Chapter 13 has Leibniz's law as a consequence. Substitution of equals for equals doesn't change the value of an expression. But here, we have a clear counterexample. The first evaluation implies that `!r` and `42` are equal. Yet if we substitute `42` for `!r` in the third expression, we get `42` instead of `21`. Once we add mutable state to the language, we need to extend the semantics from one based purely on substitution. We do so in Chapter 19, where we introduce environment semantics.

15.1.1 Reference operator types

The reference system is specifically designed so as to retain OCaml's strong typing regimen. Each of the operators, for instance, can be seen as a well-typed function. The dereference operator `!`, for instance, takes an argument of type `'a ref` and returns the `'a` referenced. It is thus typed as `(!) : 'a ref -> 'a`. The reference value constructor `ref` works in the opposite direction, taking an `'a` and returning an `'a ref`, so it types as `(ref) : 'a -> 'a ref`.

Finally, the assignment operator `:=` takes two arguments, a refer-

ence to update, of type `'a ref`, and the new `'a` value to store there. But what should the assignment operator return? Assignment is performed entirely for its side effect – the update in the state of memory – rather than for its return value. Given that there is no information in the return value, it makes sense to use a type that conveys no information. This is a natural use for the `unit` type (Section 4.3). Since `unit` has only one value (namely, the value `()`), that value conveys no information. The hallmark of a function that is used only for its side effects (which we might call a PROCEDURE) is the `unit` return type. The typing for assignment is appropriately then `(:=) : 'a ref -> 'a -> unit`.

These typings can be verified in OCaml itself:

```
# (!) ;;
- : 'a ref -> 'a = <fun>
# (ref) ;;
- : 'a -> 'a ref = <fun>
# (:=) ;;
- : 'a ref -> 'a -> unit = <fun>
```

15.1.2 Boxes and arrows

It can be helpful to visualize references using BOX AND ARROW DIAGRAMS. When establishing a reference,

```
# let r = ref 42 ;;
val r : int ref = {contents = 42}
```

we draw a box (standing for a block of memory) named `r` with an arrow pointing to another box (block of memory) containing the integer 42 (Figure 15.1(a)). Adding another reference with

```
# let s = ref 42 ;;
val s : int ref = {contents = 42}
```

generates a new named box and its referent (Figure 15.1(b)), which happens to store the same value. But we can tell that the referents are distinct, since assigning to `r` changes `!r` but not `!s` (Figure 15.1(c)).

```
# r := 21 ;;
- : unit = ()
# !r, !s ;;
- : int * int = (21, 42)
```

To have `s` refer to the value that `r` does, we need to assign to it as well (Figure 15.1(d)).

```
# s := !r ;;
- : unit = ()
```

We can have a reference `s` that points to the *same* block of memory as `r` does (Figure 15.1(e)).

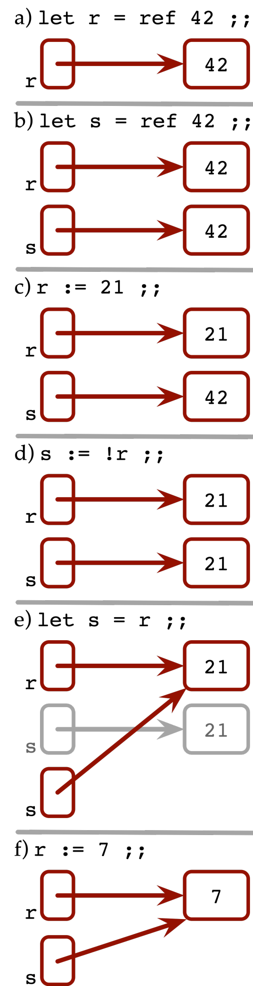

Figure 15.1: Box and arrow diagrams for the state of memory as various references are created and updated.

```
# let s = r ;;
val s : int ref = {contents = 21}
```

Now s and r have the same value (that is, refer to the same block of memory). We say that s is an ALIAS of r. (The old s is shadowed by the new one, as depicted by showing it in gray. Since we no longer have access to it and whatever it references, the gray blocks of memory are garbage. See the discussion in Section 15.1.3.)

Changing the value stored in a block of memory changes the value of all its aliases as well. Here, updating the block referred to by r (Figure 15.1(f)) changes the value for s:

```
# r := 7 ;;
- : unit = ()
# !r, !s ;;
- : int * int = (7, 7)
```

In a language with references and aliases, we are confronted with two different notions of equality. STRUCTURAL EQUALITY holds when two values have the same structure, regardless of where they are stored in memory, such as r and s in Figure 15.1(d). PHYSICAL EQUALITY holds when two values are the identical "physical" block of memory, as r and s in Figure 15.1(e). Values that are physically equal are of course structurally equal as well but the converse needn't hold.

In OCaml, structural equality and inequality are tested with `(=) : 'a -> 'a -> bool` and `(<>) : 'a -> 'a -> bool`, respectively, whereas physical equality and inequality of mutable types are tested with `(==) : 'a -> 'a -> bool` and `(!=) : 'a -> 'a -> bool`.[3]

[3] The behavior of == and != tests on immutable (pure) types is allowed to be implementation-dependent and shouldn't be relied on. These operators should only be used with values of mutable types.

Exercise 154

Construct an example defining values r and s that are structurally but not physically equal. Construct an example defining values r and s that are both structurally and physically equal. Verify these conditions using the equality functions.

15.1.3 References and pointers

You may have seen this kind of thing before. In programming languages like C, references to blocks of memory are manipulated through POINTERS to memory, which are explicitly created (with `malloc`) and freed (with `free`), dereferenced (with `*`), and updated (with `=`). Some correspondences between OCaml and C syntax for these operations are given in Table 15.1.

Notable differences between the OCaml and C approaches are:

- In OCaml, unlike in C, references can't be created without initializing them. Referencing uninitialized blocks of memory is a recipe for difficult to diagnose bugs. OCaml's type regime eliminates this entire class of bugs, since a reference type like `int ref` specifies

Operation	*OCaml*	*c*
Create, initialize	`ref 42`	
Create, name		`int *r = malloc(sizeof int);`
Create, initialize, name	`let r = ref 42`	`int *r = malloc(sizeof int);` `*r = 42;`
Dereference	`!r`	`*r`
Update	`r := 21`	`*r = 21`
Free		`free(r)`

Table 15.1: Approximate equivalencies between OCaml references and c pointers.

that the block must at all times store an `int` and the operators maintain this invariant.

- In c, nothing conspires to make sure that the size of the block allocated is appropriate for the value being stored, leading to the possibility of BUFFER OVERFLOWS – assignments that overflow one block of memory to overwrite others. Buffer overflows allow for widely exploited security holes in code. In OCaml, the strong typing again eliminates this class of bug. Similarly, BUFFER OVER-READS occur when a program reading from a block of memory continues to read past the end of the block into adjacent memory, potentially compromising the security of information in the adjacent block. An infamous example is the HEARTBLEED bug in OpenSSL, so notorious that it even acquired its own logo (Figure 15.2).

Figure 15.2: The logo for HEARTBLEED, a buffer over-read bug in the widely used OpenSSL library (written in c) for securing web interactions. The bug was revealed in 2014 after two years undiscovered in the field.

- In c, programs must free memory explicitly in order to reclaim the previously allocated memory for future use. When blocks are freed while still being used, the memory can be overwritten, leading to MEMORY CORRUPTION and once again to insidious bugs. Conversely, not freeing blocks even when they are no longer needed, called a MEMORY LEAK, leads to programs running out of memory needlessly.

  OCaml has no function for explicitly freeing memory. Instead, blocks of memory that are no longer needed, as determined by the OCaml run-time system itself, are referred to as GARBAGE. The run-time system reclaims garbage automatically, in a process called GARBAGE COLLECTION. Since computers can typically analyze the status of memory blocks better than people, the use of garbage collection eliminates memory corruption and memory leaks.

  However, the garbage collection approach takes the *timing* of memory reclamation out of the hands of the programmer. The run-time system may decide to perform computation-intensive garbage collection at inopportune times. For applications where careful control of such timing issues is necessary, the garbage collection approach

may be undesirable; use of a language, like c, that allows explicit allocation and deallocation of memory may be necessary.[4]

[4] A new class of functional programming languages is exploring the design space of languages with high-level abstraction mechanisms as in OCaml, including strongly typed safe references, while providing finer control of memory deallocation, in order to obtain the best of both the explicit approach and the garbage collection approach. The prime example is Mozilla's Rust language.

Problem 155

For each of the following expressions, give its type and value, if any.

1. ```
   let a = ref 3 in
   let b = ref 5 in
   let a = ref b in
   !(!a) ;;
   ```
2. ```
   let rec a, b = ref b, ref a in
   !a ;;
   ```
3. ```
   let a = ref 1 in
   let b = ref a in
   let a = ref 2 in
   !(!b) ;;
   ```
4. ```
   let a = 2 in
   let f = (fun b -> a * b) in
   let a = 3 in
   f (f a) ;;
   ```

15.2 *Other primitive mutable data types*

In addition to references, OCaml provides two other primitive data types that allow for mutability: mutable record fields and arrays. We mention them briefly for completeness; full details are available in the OCaml documentation.

15.2.1 *Mutable record fields*

Records (Section 7.4) are compound data structures with named fields, each of which stores a value of a particular type. As introduced, each field of a record, and hence records themselves, are immutable. However, when a record type is defined with the `type` construct, and the individual fields are specified and typed, its individual fields can also be marked as allowing mutability by adding the keyword `mutable`.

For instance, we can define a `person` record type with immutable name fields but a mutable address field.

```
# type person = {lastname : string;
#                firstname : string;
#                mutable address : string} ;;
type person = {
  lastname : string;
  firstname : string;
  mutable address : string;
}
```

Once constructed, the address of a person can be updated.

```
# let sms = {lastname = "Shieber";
#            firstname = "Stuart";
#            address = "123 Main"} ;;
val sms : person =
```

```
  {lastname = "Shieber"; firstname = "Stuart"; address = "123
  Main"}
# sms.address <- "124 Main" ;; (* I moved next door *)
- : unit = ()
```

To update a mutable record, the operator `<-` is used, rather than `:=` as for references.

In fact, reference types and their operators can be thought of as being implemented using mutable records by the following type and operator definitions:

```
type 'a ref_ = {mutable contents : 'a} ;;

let ref_ (v : 'a) : 'a ref_ = {contents = v} ;;
let (:=) (r : 'a ref_) (v : 'a) : unit = r.contents <- v ;;
let (!) (r : 'a ref_) : 'a = r.contents ;;
```

This should explain the otherwise cryptic references to `contents` when the REPL prints values of reference type.

15.2.2 Arrays

Arrays are a kind of cross between lists and tuples with added mutability. Like lists, they can have an arbitrary number of elements all of the same type. Unlike lists (but like tuples), they cannot be extended in size; there is no `cons` equivalent for arrays. Finally, each element of an array can be individually indexed and updated. An example may indicate the use of arrays:

```
# let a = Array.init 5 (fun n -> n * n) ;;
val a : int array = [|0; 1; 4; 9; 16|]
# a ;;
- : int array = [|0; 1; 4; 9; 16|]
# a.(3) <- 0 ;;
- : unit = ()
# a ;;
- : int array = [|0; 1; 4; 0; 16|]
```

Here, we've created an array of five elements, each the square of its index. We update the third element to be 0, and examine the result, which now has a 0 in the appropriate location.

15.3 References and mutation

To provide an example of the use of mutating references, we consider the task of counting the occurrences of an event. We start by establishing a location to store the current count as an `int ref` named `gctr` (for "global counter").

```
# let gctr = ref 0 ;;
val gctr : int ref = {contents = 0}
```

Now we define a function that "bumps" the counter (adding 1) and then returns the current value of the counter.

```
# let bump () =
#   gctr := !gctr + 1;
#   !gctr ;;
val bump : unit -> int = <fun>
```

We've used a new operator here, the binary sequencing operator (`;`), which is a bit like the pair operator (`,`) in that it evaluates its left and right arguments, except that the sequencing operator returns the value only of the second.[5] But then what could possibly be the point of evaluating the first argument? Since the argument isn't used for its value, it must be of interest for its side effects. That is the case in this example; the expression `gctr := !gctr + 1` has the side effect of updating the counter to a new value, its old value (retrieved with `!gctr`) plus one.[6] Since the sequencing operator ignores the value returned by its first argument, it requires that argument to be of type `unit`, the type for expressions with no useful value.[7]

We can test it out.

```
# bump () ;;
- : int = 1
# bump () ;;
- : int = 2
# bump () ;;
- : int = 3
```

Again, you see the hallmark of impure code – the same expression in the same context evaluates to different values. The change between invocations happens because of the side effects of the earlier calls to `bump`. We can see evidence of the side effects also in the value of the counter, which is globally visible.

```
# !gctr ;;
- : int = 3
```

In the case of the `bump` function, it is the intention to provide these side effects. They are what generates the counting functionality. However, it is not necessarily the intention to make the current counter visible to users of the `bump` function. Doing so enables unintended side effects, like manipulating the value stored in the counter outside of the manipulation by the `bump` function itself, enabling misuses such as the following:

```
# gctr := -17 ;;
- : unit = ()
# bump () ;;
- : int = -16
```

[5] You can think of P `;` Q as being syntactic sugar for `let () =` P `in` Q.

[6] This part of the `bump` function that does the actual incrementing of an `int ref` is a common enough activity that OCaml provides a function `incr : int ref -> unit` in the `Stdlib` library for just this purpose. It works as if implemented by

```
let incr (r : int ref) : unit =
  r := !r + 1 ;;
```

We could therefore have substituted `incr gctr` as the second line of the `bump` function.

[7] Sometimes, you may want to sequence an expression that returns a value other than `()`. The `ignore` function of type `'a -> unit` in `Stdlib` comes in handy in such cases.

To eliminate this abuse we'd like to avoid a global variable for the counter. We've seen this kind of information hiding before – in the use of local variables within functions, and in the use of signatures to hide auxiliary values and functions from users of modules, all instances of the edict of compartmentalization. But in the context of assignment, making `gctr` a local variable (we'll call it `ctr`) requires some thought. A naive approach doesn't work:

```
# let bump () =
#   let ctr = ref 0 in
#   ctr := !ctr + 1;
#   !ctr ;;
val bump : unit -> int = <fun>
```

Exercise 156
What goes wrong with this definition? Try using it a few times and see what happens.

The problem: This code establishes the counter variable `ctr` upon *application* of `bump`, and establishes a new such variable at each such application. Instead, we want to define `ctr` just once, upon the *definition* of `bump`, and not its applications.

In this case, the compact notation for function definition, which conflates the defining of the function and its naming, is doing us a disservice. Fortunately, we aren't obligated to use that syntactic sugar. We can use the desugared version:

```
let bump =
  fun () ->
    ctr := !ctr + 1;
    !ctr ;;
```

Now the naming (first line) and the function definition (second line and following) are separate. We want the definition of `ctr` to outscope the function definition but fall within the local scope of its naming:

```
# let bump =
#   let ctr = ref 0 in
#   fun () ->
#     ctr := !ctr + 1;
#     !ctr ;;
val bump : unit -> int = <fun>
```

The function is defined within the scope of – and therefore can access and modify – a local variable `ctr` whose scope is *only* that function. This definition operates as before to deliver incremented integers:

```
# bump () ;;
- : int = 1
# bump () ;;
- : int = 2
# bump () ;;
- : int = 3
```

but access to the counter variable is available only within the function, as it should be, and not outside of it:

```
# !ctr ;;
Line 1, characters 1-4:
1 | !ctr ;;
     ^^^
Error: Unbound value ctr
Hint: Did you mean gctr?
```

This example – the counter with local, otherwise inaccessible, persistent, mutable state – is one of the most central to understand. We'll see a dramatic application of this simple pattern in Chapter 18, where it underlies the idea of instance variables in object-oriented programming.

Problem 157

Suppose you typed the following OCaml expressions into the OCaml REPL sequentially.

```
1  let p = ref 11 ;;
2  let r = ref p ;;
3  let s = ref !r ;;
4  let t =
5    !s := 14;
6    !p + !(!r) + !(!s) ;;
7  let t =
8    s := ref 17;
9    !p + !(!r) + !(!s) ;;
```

Try to answer the questions below about the status of the various variables being defined before typing them into the REPL yourself.

1. After line 1, what is the type of `p`?
2. After line 2, what is the type of `r`?
3. After line 3, which of the following statements are true?
   (a) `p` and `s` have the same type
   (b) `r` and `s` have the same type
   (c) `p` and `s` have the same value (in the sense that `p = s` would be `true`)
   (d) `r` and `s` have the same value (in the sense that `r = s` would be `true`)
4. After line 6, what is the value of `t`?
5. After line 9, what is the value of `t`?

15.4 Mutable lists

To demonstrate the power of imperative programming, we use OCaml's imperative aspects to provide implementations of two mutable data structures: mutable lists and mutable queues.

As noted in Section 11.1, the OCaml list type operates as if defined by

```
type 'a list =
  | Nil
  | Cons of 'a * 'a list ;;
```

A mutable list allows values constructed in this way to be updated; we thus take values of type `'a mlist` to be references to such compound structures.

```
# type 'a mlist = 'a mlist_internal ref
#  and 'a mlist_internal =
#    | Nil
#    | Cons of 'a * 'a mlist ;;
type 'a mlist = 'a mlist_internal ref
and 'a mlist_internal = Nil | Cons of 'a * 'a mlist
```

(In this mutually recursive pair of type definitions, the intention is to make use of values of type `'a mlist`. The auxiliary type `'a mlist_internal` is just an expedient, required because OCaml needs a name for the type of values that references refer to.)

The shortest such mutable list is simply a reference to the `Nil` value (in this case, coerced to an integer mutable list).

```
# let r : int mlist = ref Nil ;;
val r : int mlist = {contents = Nil}
```

We can build longer mutable lists by consing on a couple of integers. We'll do that bit by bit to allow naming of the intermediate lists.

```
# let s : int mlist = ref (Cons (1, r)) ;;
val s : int mlist = {contents = Cons (1, {contents = Nil})}
# let t : int mlist = ref (Cons (2, s));;
val t : int mlist =
  {contents = Cons (2, {contents = Cons (1, {contents = Nil})})}
```

We can compute the length of such a list using the usual recursive definition.

```
# let rec mlength (lst : 'a mlist) : int =
#   match !lst with
#   | Nil -> 0
#   | Cons (_hd, tl) -> 1 + mlength tl ;;
val mlength : 'a mlist -> int = <fun>
```

Comparing this with the definition of `length : 'a list -> int` from Section 7.3.1, the only difference here is the dereferencing of the mutable list before it can be matched. Sure enough, this definition works on the example mutable lists above.

```
# mlength r ;;
- : int = 0
# mlength s ;;
- : int = 1
# mlength t ;;
- : int = 2
```

Box and arrow diagrams (Figure 15.3) help in figuring out what's going on here.

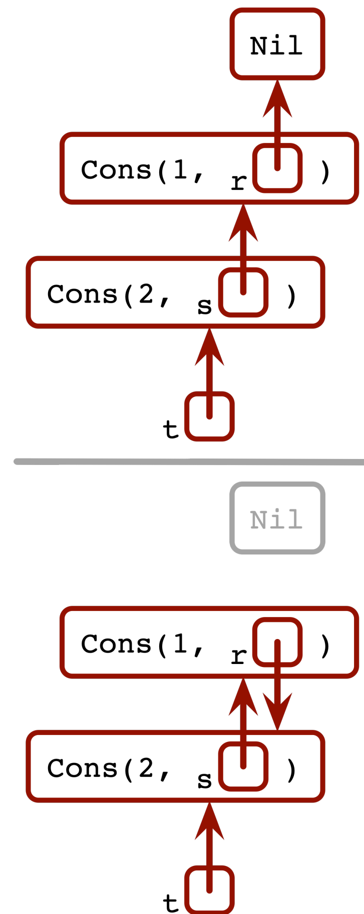

Figure 15.3: Pictorial representation of (top) the state of memory after building some mutable list structures, and (bottom) updating with `r := !t`. The `nil` has become garbage and the mutable lists `r`, `s`, and `t` now have cycles in them.

Exercise 158

Write functions `mhead` and `mtail` that extract the head and the (dereferenced) tail from a mutable list. For example,

```
# mhead t ;;
- : int = 2
# mtail t ;;
- : int mlist = {contents = Cons (1, {contents = Nil})}
```

Because the lists are mutable, we can modify the tail of `s` (that is, `r`) to point to `t`.

```
# r := !t ;;
- : unit = ()
```

Since the tail of `s` points to `t` and the tail of `t` to `s`, we've constructed a CYCLIC data structure. Doing so uncovers a bug in the `mlength` function,

```
# mlength t ;;
Stack overflow during evaluation (looping recursion?).
```

demonstrating once again how adding impure features to a language introduces new and quite subtle complexities.

Problem 159

For each of the following expressions, give its type and value, if any.

1. `let a = ref (Cons (2, ref (Cons (3, ref Nil)))) ;;`
2.
```
let Cons (_hd, tl) = !a in
let b = ref (Cons (1, a)) in
tl := !b ;
mhead (mtail (mtail b)) ;;
```

Problem 160

Provide an implementation of the `mlength` function that handles cyclic lists, so that

```
# mlength t ;;
- : int = 3
```

You'll notice that the requirement to handle cyclic lists dramatically increases the complexity of implementing `length`. (Hint: Keep a list of sublists you've already visited and check to see if you've already visited each sublist. What is a reasonable value to return in that case?)

Problem 161

Define a function `mfirst : int -> 'a mlist -> 'a list` that returns a list (immutable) of the first `n` elements of a mutable list:

Problem 162

Write code to define a mutable integer list `alternating` such that for all integers `n`, the expression `mfirst n alternating` returns a list of alternating 1s and 2s, for example,

```
# mfirst 5 alternating ;;
- : int list = [1; 2; 1; 2; 1]
# mfirst 8 alternating ;;
- : int list = [1; 2; 1; 2; 1; 2; 1; 2]
```

15.5 Imperative queues

By way of review, the pure functional queue data structure in Section 12.4 implemented the following signature:

```
# module type QUEUE = sig
#   type 'a queue
#   val empty_queue : 'a queue
#   val enqueue : 'a -> 'a queue -> 'a queue
```

```
#   val dequeue : 'a queue -> 'a * 'a queue
# end ;;
module type QUEUE =
  sig
    type 'a queue
    val empty_queue : 'a queue
    val enqueue : 'a -> 'a queue -> 'a queue
    val dequeue : 'a queue -> 'a * 'a queue
  end
```

Each call to enqueue and dequeue returns a new queue, differing from its argument queue in having an element added or removed.

In an imperative implementation of queues, the enqueuing and dequeuing operations can and do mutate the data structure, so that the operations don't need to return an updated queue. The types for the operations thus change accordingly. We'll use the following `IMP_QUEUE` signature for imperative queues:

```
# module type IMP_QUEUE = sig
#   type 'a queue
#   val empty_queue : unit -> 'a queue
#   val enqueue : 'a -> 'a queue -> unit
#   val dequeue : 'a queue -> 'a option
# end ;;
module type IMP_QUEUE =
  sig
    type 'a queue
    val empty_queue : unit -> 'a queue
    val enqueue : 'a -> 'a queue -> unit
    val dequeue : 'a queue -> 'a option
  end
```

Here again, you see the sign of a side-effecting operation: the `enqueue` operation returns a `unit`. Dually, to convert a procedure that modifies its argument and returns a `unit` into a pure function, the standard technique is to have the function return instead a modified copy of its argument, leaving the original untouched. Indeed, when we generalize the substitution semantics of Chapter 13 to handle state and state change in Chapter 19, we will use just this technique of passing a representation of the computation state as an argument and returning a representation of the updated state as the return value.

Another subtlety introduced by the addition of mutability is the type of the `empty_queue` value. In the functional signature, we had `empty_queue : 'a queue`; the `empty_queue` value was an empty queue. In the mutable signature, we have `empty_queue : unit -> 'a queue`; the `empty_queue` value is a function that returns a (new, physically distinct) empty queue. Without this change, the `empty_queue` value would be "poisoned" as soon as something was inserted in it, so that further references to `empty_queue` would see the modified

(non-empty) value. Instead, the `empty_queue` function can generate a new empty queue each time it is called.

15.5.1 Method 1: List references

Perhaps the simplest method to implement an imperative queue is as a (mutable) reference to an (immutable) list of the queue's elements.

```
# module SimpleImpQueue : IMP_QUEUE =
#   struct
#     type 'a queue = 'a list ref
#     let empty_queue () = ref []
#     let enqueue elt q =
#       q := (!q @ [elt])
#     let dequeue q =
#       match !q with
#       | first :: rest -> (q := rest; Some first)
#       | [] -> None
#   end ;;
module SimpleImpQueue : IMP_QUEUE
```

This is basically the same as the list implementation from Section 12.4, but with the imperative signature. Nonetheless, internally the operations are still functional, and enqueuing an element requires time linear in the number of elements in the queue. (Recall from Section 14.5 that the functional append function (here invoked as `Stdlib.(@)`) is linear.)

We'll examine two methods for generating constant time implementations of an imperative queue.

15.5.2 Method 2: Two stacks

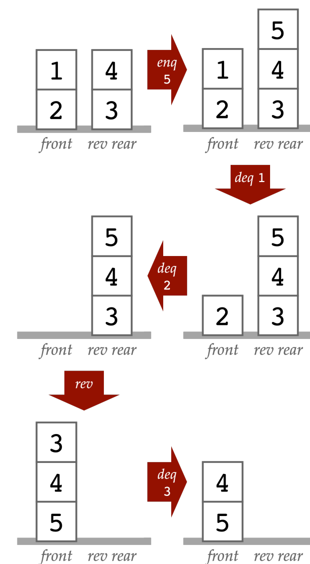

Figure 15.4: Pictorial representation of implementing a queue with two stacks.

An old trick is to use two stacks to implement a queue. The two stacks hold the front of the queue (the first elements in, and hence the first out) and the reversal of the rear of the queue. For example, a queue containing the elements 1 through 4 in order might be represented by the two stacks (implemented as `int lists`) `[1; 2]` and `[4; 3]`, or pictorially as in Figure 15.4 (upper left).

Enqueuing works by adding an element (5 in upper right) to the *rev rear* stack. Dequeuing works by popping the top element in the *front* stack, if there is one (middle right and left and lower right). If there are no elements to dequeue in the *front* stack (middle left), the *rev rear* stack is reversed onto the *front* stack first (lower left).

The stacks can be implemented with type `'a list ref` and the two stacks packaged together in a record.

```
module TwoStackImpQueue : IMP_QUEUE =
  struct
    type 'a queue = {front   : 'a list ref;
```

```
                     revrear : 'a list ref}
    ...
```

The empty queue has two empty lists.

```
module TwoStackImpQueue : IMP_QUEUE =
  struct
    type 'a queue = {front   : 'a list ref;
                     revrear : 'a list ref}
    let empty_queue () =
      {front = ref []; revrear = ref []}
    ...
```

Enqueuing simply places the element on the top of the rear stack.

```
module TwoStackImpQueue : IMP_QUEUE =
  struct
    type 'a queue = {front   : 'a list ref;
                     revrear : 'a list ref}
    let empty_queue () =
      {front = ref []; revrear = ref []}
    let enqueue elt q =
      q.revrear := elt :: !(q.revrear)
    ...
```

Dequeuing is the more complicated operation.

```
# module TwoStackImpQueue : IMP_QUEUE =
#   struct
#     type 'a queue = {front   : 'a list ref;
#                      revrear : 'a list ref}
#     let empty_queue () =
#       {front = ref []; revrear = ref []}
#     let enqueue elt q =
#       q.revrear := elt :: !(q.revrear)
#     let rec dequeue q =
#       match !(q.front) with
#       | h :: t -> (q.front := t; Some h)
#       | [] -> if !(q.revrear) = [] then None
#               else ((* reverse revrear onto front *)
#                     q.front := List.rev (!(q.revrear));
#                     (* clear revrear *)
#                     q.revrear := [];
#                     (* try the dequeue again *)
#                     dequeue q)
#   end ;;
module TwoStackImpQueue : IMP_QUEUE
```

As in method 1, the enqueue operation takes constant time. But dequeuing usually takes constant time too, unless we have to perform the reversal of the rear stack. Since the stack reversal takes time linear in the number of enqueues, the time to enqueue and dequeue elements is, on average, constant time per element.

Exercise 163

An alternative is to use mutable record fields, so that the queue type would be

```
type 'a queue = {mutable front   : 'a list;
                 mutable revrear : 'a list}
```

Reimplement the `TwoStackImpQueue` module using this type for the queue implementation.

15.5.3 Method 3: Mutable lists

To allow for manipulation of both the head of the queue (where enqueuing happens) and the tail (where dequeuing happens), a final implementation uses mutable lists. The queue type

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    ...
```

provides a reference to the front of the queue as well as a reference to the last cons in the queue if there is one. When the queue is empty, both of these lists will be `ref Nil`.

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    let empty_queue () = {front = ref Nil;
                          rear = ref Nil}
    ...
```

Enqueuing a new element differs depending on whether the queue is empty. If it already contains at least one element, the rear will have a head and a `ref Nil` tail (because the rear always points to the last cons.

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    let empty_queue () = {front = ref Nil;
                          rear = ref Nil}
    let enqueue elt q =
      match !(q.rear) with
      | Cons (hd, tl) -> (assert (!tl = Nil);
                          tl := Cons (elt, ref Nil);
                          q.rear := !tl)
      | Nil -> ...
```

If the queue is empty, we establish a single element mutable list with front and rear pointers to its single element.

```
module MutableListQueue : IMP_QUEUE =
  struct
    type 'a queue = {front : 'a mlist;
                     rear : 'a mlist}
    let empty_queue () = {front = ref Nil;
                          rear = ref Nil}
    let enqueue elt q =
      match !(q.rear) with
      | Cons (hd, tl) -> (assert (!tl = Nil);
                          tl := Cons (elt, ref Nil);
                          q.rear := !tl)
      | Nil -> (assert (!(q.front) = Nil);
                q.front := Cons (elt, ref Nil);
                q.rear := !(q.front))
    ...
```

Finally, dequeuing involves moving the front pointer to the next element in the list, and updating the rear to `Nil` if the last element was dequeued and the queue is now empty.

```
# module MutableListQueue : IMP_QUEUE =
#   struct
#     type 'a queue = {front : 'a mlist;
#                      rear : 'a mlist}
#
#     let empty_queue () = {front = ref Nil;
#                           rear = ref Nil}
#     let enqueue elt q =
#       match !(q.rear) with
#       | Cons (_hd, tl) -> (assert (!tl = Nil);
#                            tl := Cons (elt, ref Nil);
#                            q.rear := !tl)
#       | Nil -> (assert (!(q.front) = Nil);
#                 q.front := Cons (elt, ref Nil);
#                 q.rear := !(q.front))
#     let dequeue q =
#       match !(q.front) with
#       | Cons (hd, tl) ->
#           (q.front := !tl;
#            (match !tl with
#             | Nil -> q.rear := Nil
#             | Cons (_, _) -> ());
#            Some hd)
#       | Nil -> None
#   end ;;
module MutableListQueue : IMP_QUEUE
```

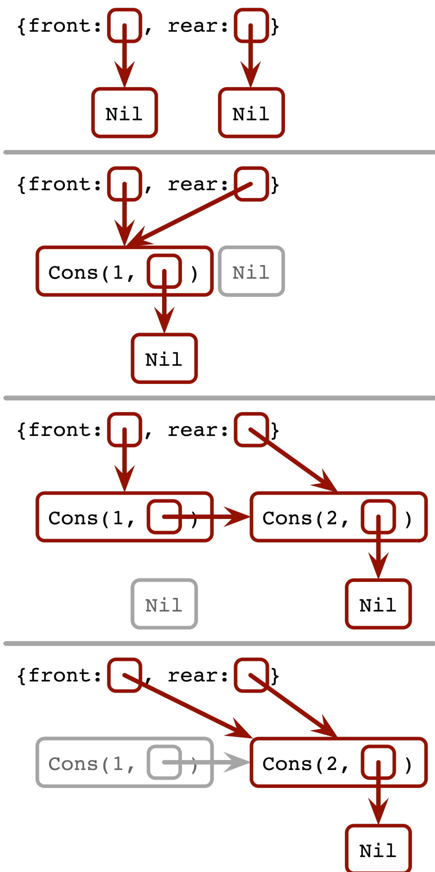

Figure 15.5: Pictorial representation of implementing a queue with a mutable list.

Figure 15.5 depicts the queue data structure as it performs the following operations:

```
# let open MutableListQueue in
# let q = empty_queue () in
# enqueue 1 q;
# enqueue 2 q;
```

```
# dequeue q ;;
- : int option = Some 1
```

15.6 Hash tables

A hash table is a data structure implementing a mutable dictionary. We've seen functional key-value dictionaries already in Section 12.6, which implement a signature like the following:

```
module type DICT =
  sig
    type key
    type value
    type dict

    (* An empty dictionary *)
    val empty : dict
    (* Returns as an option the value associated with the
       provided key. If the key is not in the dictionary,
       returns None. *)
    val lookup : dict -> key -> value option
    (* Returns true if and only if the key is in the
       dictionary. *)
    val member : dict -> key -> bool
    (* Inserts a key-value pair into the dictionary. If the
       key is already present, updates the key to have the
       new value. *)
    val insert : dict -> key -> value -> dict
    (* Removes the key from the dictionary. If the key is
       not present, returns the original dictionary. *)
    val remove : dict -> key -> dict
  end ;;
```

In a mutable dictionary, the data structure state is actually modified by side effect when inserting or removing key-value pairs. Consequently, those functions need not (and should not) return an updated dictionary. (As with mutable lists, because dictionaries can be modified by side effect, care must also be taken with specifying an empty dictionary. Instead of a single empty dictionary value, we provide a function from `unit` that returns a new empty dictionary.) An appropriate signature for a mutable dictionary, then, is

```
# module type MDICT =
#   sig
#     type key
#     type value
#     type dict
#
#     (* Returns an empty dictionary. *)
#     val empty : unit -> dict
#     (* Returns as an option the value associated with the
#        provided key. If the key is not in the dictionary,
```

```
#        returns None. *)
#     val lookup : dict -> key -> value option
#     (* Returns true if and only if the key is in the
#        dictionary. *)
#     val member : dict -> key -> bool
#     (* Inserts a key-value pair into the dictionary. If the
#        key is already present, updates the key to have the
#        new value. *)
#     val insert : dict -> key -> value -> unit
#     (* Removes the key from the dictionary. If the key is
#        not present, leaves the original dictionary unchanged. *)
#     val remove : dict -> key -> unit
#   end ;;
module type MDICT =
  sig
    type key
    type value
    type dict
    val empty : unit -> dict
    val lookup : dict -> key -> value option
    val member : dict -> key -> bool
    val insert : dict -> key -> value -> unit
    val remove : dict -> key -> unit
  end
```

In a HASH TABLE implementation of this signature, the key-value pairs are stored in a mutable array of a given size at an index specified by a HASH FUNCTION, a function from keys to integers within the range provided. The idea is that the hash function should assign well distributed locations to keys, so that inserting or looking up a particular key-value pair involves just computing the hash function to generate the location where it can be found. Thus, insertion and lookup are constant-time operations.

An important problem to resolve is what to do in case of a HASH COLLISION, when two different keys hash to the same value. We assume that only a single key-value pair can be stored at a given location in the hash table – called CLOSED HASHING – so in case of a collision when inserting a key-value pair, we keep searching in the table at the sequentially following array indices until an empty slot in the table is found. Similarly, when looking up a key, if the key-value pair stored at the hash location does not match the key being looked up, we sequentially search for a pair that does match. This process of trying sequential locations is known as LINEAR PROBING. Frankly, linear probing is not a particularly good method for handling hash collisions (see Exercises 165 and 166), but it will do for our purposes here.

To define a new kind of hash table, we need to provide types for the keys and values, a size for the array, and an appropriate hash function. We package all of this up in a module that can serve as the argument to

a functor.

```
# module type MDICT_ARG =
#   sig
#     (* Types to be used for the dictionary keys and values *)
#     type key
#     type value
#     (* size -- Number of elements that can be stored in the
#        dictionary *)
#     val size : int
#     (* hash_fn key -- Returns the hash value for a key. *)
#     val hash_fn : key -> int
#   end ;;
module type MDICT_ARG =
  sig type key type value val size : int val hash_fn : key -> int
  end
```

Here is the beginning of an implementation of such a functor:

```
module MakeHashtableDict (D : MDICT_ARG)
       : (MDICT with type key = D.key
                 and type value = D.value) =
  struct
    type key = D.key
    type value = D.value

    (* A hash record is a key value pair *)
    type hashrecord = { key : key;
                        value : value }
    (* An element of the hash table array is a hash record
       (or empty) *)
    type hashelement =
      | Empty
      | Element of hashrecord
    (* The hash table itself is a (mutable) array of hash
       elements *)
    type dict = hashelement array

    let empty () = Array.make D.size Empty
    ...
  end ;;
```

With a full implementation of the `MakeHashtableDict` functor (Exercise 164), we can build an `IntStringHashtbl` hash table module for hash tables that map integers to strings as follows:[8]

[8] The hash function we use here is an especially poor choice; we use it to make it easy to experiment with hash collisions.

```
# module IntStringHashtbl : (MDICT with type key = int
#                                  and type value = string) =
#   MakeHashtableDict (struct
#                       type key = int
#                       type value = string
#                       let size = 100
#                       let hash_fn k = (k / 3) mod size
#                      end) ;;
module IntStringHashtbl :
```

```
sig
  type key = int
  type value = string
  type dict
  val empty : unit -> dict
  val lookup : dict -> key -> value option
  val member : dict -> key -> bool
  val insert : dict -> key -> value -> unit
  val remove : dict -> key -> unit
end
```

Let's experiment:

```
# open IntStringHashtbl ;;
# let d = empty () ;;
val d : IntStringHashtbl.dict = <abstr>
# insert d 10 "ten" ;;
- : unit = ()
# insert d 9 "nine" ;;
- : unit = ()
# insert d 34 "34" ;;
- : unit = ()
# insert d 1000 "a thousand" ;;
- : unit = ()
# lookup d 10 ;;
- : IntStringHashtbl.value option = Some "ten"
# lookup d 9 ;;
- : IntStringHashtbl.value option = Some "nine"
# lookup d 34 ;;
- : IntStringHashtbl.value option = Some "34"
# lookup d 8 ;;
- : IntStringHashtbl.value option = None
# remove d 9 ;;
- : unit = ()
# lookup d 10 ;;
- : IntStringHashtbl.value option = Some "ten"
# lookup d 9 ;;
- : IntStringHashtbl.value option = None
# lookup d 34 ;;
- : IntStringHashtbl.value option = Some "34"
# lookup d 8 ;;
- : IntStringHashtbl.value option = None
```

Exercise 164

Complete the implementation by providing implementations of the remaining functions `lookup`, `member`, `insert`, and `remove`.

Exercise 165

Improve the collision handling in the implementation by allowing the linear probing to "wrap around" so that if it reaches the end of the array it keeps looking at the beginning of the array.

Exercise 166

A problem with linear probing is that as collisions happen, contiguous blocks of the array get filled up, so that further collisions tend to yield long searches to get past

these blocks for an empty location. Better is to use a method of rehashing that leaves some gaps. A simple method to do so is QUADRATIC PROBING: each probe increases quadratically, adding 1, then 2, then 4, then 8, and so forth. Modify the implementation so that it uses quadratic probing instead of linear probing.

15.7 Conclusion

With the introduction of references, we move from thinking about what expressions *mean* to what they *do*. The ability to mutate state means that data structures can now undergo change. By modifying existing data structures, we may be able to avoid building new copies, thereby saving some space. More importantly, performing small updates may be much faster than constructing large copies, leading to improvements in both space and time complexity.

But making good on these benefits requires much more subtle reasoning about what programs are up to. The elegant substitution model – which says that expressions are invariant under substitution of one subexpression by another with the same value – doesn't hold when side effects can change those values out from under us. Aliasing means that changes in one part of the code can have ramifications far afield. Modifying data structures means that the hierarchical structures can be modified to form cycles, with the potential to fall into infinite loops. (We explore the changes needed to the substitution semantics of Chapter 13 to allow for mutable state in Chapter 19.)

Nonetheless, the underlying structure of modern computer hardware is based on stateful memory to store program and data, so that at some point imperative programming is a necessity. Imperative programming can be a powerful way of thinking about implementing functionality.

❧

We've now introduced essentially all of the basic language constructs that we need. In the following chapters, we deploy them in new combinations that interact to provide additional useful programming abstractions – providing looping constructs to enable the repetition of side effects (Chapter 16); the ability to perform a computation "lazily", delaying it until its result is needed (Chapter 17); and the encapsulation of computations within data objects that they act on (Chapter 18).

15.8 Supplementary material

- Lab 12: Imperative programming and references