4

Functional and Imperative Styles

Functional and imperative programming languages are primarily distinguished by the control over program execution and the data memory management.

- A functional program computes an expression. This computation results in a value. The order in which the operations needed for this computation occur does not matter, nor does the physical representation of the data manipulated, because the result is the same anyway. In this setting, deallocation of memory is managed implicitly by the language itself: it relies on an automatic garbage collector or *GC*; see chapter 9.
- An imperative program is a sequence of instructions modifying a memory state. Each execution step is enforced by rigid control structures that indicate the next instruction to be executed. Imperative programs manipulate pointers or references to values more often than the values themselves. Hence, the memory space needed to store values must be allocated and reclaimed explicitly, which sometimes leads to errors in accessing memory. Nevertheless, nothing prevents use of a GC.

Imperative languages provide greater control over execution and the memory representation of data. Being closer to the actual machine, the code can be more efficient, but loses in execution safety. Functional programming, offering a higher level of abstraction, achieves a better level of execution safety: Typing (dynamic or static) may be stricter in this case, thus avoiding operations on incoherent values. Automatic storage reclamation, in exchange for giving up efficiency, ensures the current existence of the values being manipulated.

Historically, the two programming paradigms have been seen as belonging to different universes: symbolic applications being suitable for the former, and numerical applications being suitable for the latter. But certain things have changed, especially techniques for compiling functional programming languages, and the efficiency of GCs. From another side, execution safety has become an important, sometimes the predominant criterion in the quality of an application. Also familiar is the “selling point” of

the *Java* language, according to which efficiency need not preempt assurance, especially if efficiency remains reasonably good. And this idea is spreading among software producers.

Objective Caml belongs to this class. It combines the two programming paradigms, thus enlarging its domain of application by allowing algorithms to be written in either style. It retains, nevertheless, a good degree of execution safety because of its static typing, its GC, and its exception mechanism. Exceptions are a first explicit execution control structure; they make it possible to break out of a computation or restart it. This trait is at the boundary of the two models, because although it does not replace the result of a computation, it can modify the order of execution. Introducing physically mutable data can alter the behavior of the purely functional part of the language. For instance, the order in which the arguments to a function are evaluated can be determined, if that evaluation causes side effects. For this reason, such languages are called "impure functional languages." One loses in level of abstraction, because the programmer must take account of the memory model, as well as the order of events in running the program. This is not always negative, especially for the efficiency of the code. On the other hand, the imperative aspects change the type system of the language: some functional programs, correctly typed in theory, are no longer in fact correctly typed because of the introduction of references. However, such programs can easily be rewritten.

Plan of the Chapter

This chapter provides a comparison between the functional and imperative models in the Objective Caml language, at the level both of control structure and of the memory representation of values. The mixture of these two styles allows new data structures to be created. The first section studies this comparison by example. The second section discusses the ingredients in the choice between composition of functions and sequencing of instructions, and in the choice between sharing and copying values. The third section brings out the interest of mixing these two styles to create mutable functional data, thus permitting data to be constructed without being completely evaluated. The fourth section describes *streams*, potentially infinite sequences of data, and their integration into the language via pattern-matching.

Comparison between Functional and Imperative

Character strings (of Objective Caml type `string`) and linked lists (of Objective Caml type `'a list`) will serve as examples to illustrate the differences between "functional" and "imperative."

The Functional Side

The function `map` (see page 26) is a classic ingredient in functional languages. In a purely functional style, it is written:

```
# let rec map f l = match l with
        []    →   []
    | h :: q  →   (f h) :: (map f q)  ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

It recursively constructs a list by applying `f` to the elements of the list given as argument, independently specifying its head `(f h)` and its tail `(map f q)`. In particular, the program does not stipulate which of the two will be computed first.

Moreover, the physical representation of lists need not be known to the programmer to write such a function. In particular, problems of allocating and sharing data are managed implicitly by the system and not by the programmer. An example illustrating this follows:

```
# let example = [ "one" ; "two" ; "three" ] ;;
val example : string list = ["one"; "two"; "three"]
# let result = map (function x → x) example  ;;
val result : string list = ["one"; "two"; "three"]
```

The lists `example` and `result` contain equal values:

```
# example = result ;;
- : bool = true
```

These two values have exactly the same structure even though their representation in memory is different, as one learns by using the test for physical equality:

```
# example == result ;;
- : bool = false
# (List.tl example) == (List.tl result) ;;
- : bool = false
```

The Imperative Side

Let us continue the previous example, and modify a string in the list `result`.

```
# (List.hd result).[1] <- 's' ;;
- : unit = ()
# result ;;
- : string list = ["ose"; "two"; "three"]
# example ;;
- : string list = ["ose"; "two"; "three"]
```

Evidently, this operation has modified the list `example`. Hence, it is necessary to know the physical structure of the two lists being manipulated, as soon as we use imperative aspects of the language.

Let us now observe how the order of evaluating the arguments of a function can amount to a trap in an imperative program. We define a mutable list structure with primitive functions for creation, modification, and access:

```
# type 'a ilist = { mutable c : 'a list } ;;
type 'a ilist = { mutable c: 'a list }
# let icreate () = { c = [] }
  let iempty l = (l.c = [])
  let icons x y = y.c <- x :: y.c ; y
  let ihd x = List.hd x.c
  let itl x = x.c <- List.tl x.c ; x ;;
val icreate : unit -> 'a ilist = <fun>
val iempty : 'a ilist -> bool = <fun>
val icons : 'a -> 'a ilist -> 'a ilist = <fun>
val ihd : 'a ilist -> 'a = <fun>
val itl : 'a ilist -> 'a ilist = <fun>
# let rec imap f l =
    if iempty l then icreate()
    else icons (f (ihd l))  (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
```

Despite having reproduced the general form of the `map` of the previous paragraph, with `imap` we get a distinctly different result:

```
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x → x) example ;;
Uncaught exception: Failure("hd")
```

What has happened? Just that the evaluation of `(itl l)` has taken place before the evaluation of `(ihd l)`, so that on the last iteration of `imap`, the list referenced by `l` became the empty list before we examined its head. The list `example` is henceforth definitely empty even though we have not obtained any result:

```
# example ;;
- : string ilist = {c=[]}
```

The flaw in the function `imap` arises from a mixing of the genres that has not been controlled carefully enough. The choice of order of evaluation has been left to the system. We can reformulate the function `imap`, making explicit the order of evaluation, by using the syntactic construction **`let .. in ..`**

```
# let rec imap f l =
    if iempty l then icreate()
    else let h = ihd l  in  icons (f h)  (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x → x) example ;;
```

```
- : string ilist = {c=["one"; "two"; "three"]}
```

However, the original list has still been lost:

```
# example ;;
- : string ilist = {c=[]}
```

Another way to make the order of evaluation explicit is to use the sequencing operator and a looping structure.

```
# let imap f l =
    let l_res = icreate ()
    in while not (iempty l) do
         ignore (icons (f (ihd l)) l_res) ;
         ignore (itl l)
       done ;
       { l_res with c = List.rev l_res.c } ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x → x) example ;;
- : string ilist = {c=["one"; "two"; "three"]}
```

The presence of `ignore` emphasizes the fact that it is not the result of the functions that counts here, but their side effects on their argument. In addition, we had to put the elements of the result back in the right order (using the function `List.rev`).

Recursive or Iterative

People often mistakenly associate recursive with functional and iterative with imperative. A purely functional program cannot be iterative because the value of the condition of a loop never varies. By contrast, an imperative program may be recursive: the original version of the function `imap` is an example.

Calling a function conserves the values of its arguments during its computation. If it calls another function, the latter conserves its own arguments in addition. These values are conserved on the *execution stack*. When the call returns, these values are popped from the stack. The memory space available for the stack being bounded, it is possible to encounter the limit when using a recursive function with calls too deeply nested. In this case, Objective Caml raises the exception `Stack_overflow`.

```
# let rec succ n = if n = 0 then 1 else 1 + succ (n-1) ;;
val succ : int -> int = <fun>
# succ 100000 ;;
Stack overflow during evaluation (looping recursion?).
```

In the iterative version `succ_iter`, the stack space needed for a call does not depend on its argument.

```
# let succ_iter n =
    let i = ref 0 in
      for j=0 to n do incr i done ;
      !i ;;
val succ_iter : int -> int = <fun>
# succ_iter 100000 ;;
- : int = 100001
```

The following recursive version has *a priori* the same depth of calls, yet it executes successfully with the same argument.

```
# let succ_tr n =
    let rec succ_aux n accu =
      if n = 0 then accu  else succ_aux (n-1) (accu+1)
    in
      succ_aux 1 n ;;
val succ_tr : int -> int = <fun>
# succ_tr 100000 ;;
- : int = 100001
```

This function has a special form of recursive call, called *tail recursion*, in which the result of this call will be the result of the function without further computation. It is therefore unnecessary to have stored the values of the arguments to the function while computing the recursive call. When Objective Caml can observe that a call is tail recursive, it frees the arguments on the stack before making the recursive call. This optimization allows recursive functions that do not increase the size of the stack.

Many languages detect tail recursive calls, but it is indispensable in a functional language, where naturally many tail recursive calls are used.

Which Style to Choose?

This is no matter of religion or esthetics; *a priori* neither style is prettier or holier than the other. On the contrary, one style may be more adequate than the other depending on the problem to be solved.

The first rule to apply is the rule of simplicity. Whether the algorithm to use implemented is written in a book, or whether its seed is in the mind of the programmer, the algorithm is itself described in a certain style. It is natural to use the same style when implementing it.

The second criterion of choice is the efficiency of the program. One may say that an imperative program (if well written) is more efficient that its functional analogue, but in very many cases the difference is not enough to justify complicating the code to

adopt an imperative style where the functional style would be natural. The function `map` in the previous section is a good example of a problem naturally expressed in the functional style, which gains nothing from being written in the imperative style.

Sequence or Composition of Functions

We have seen that as soon as a program causes side effects, it is necessary to determine precisely the order of evaluation for the elements of the program. This can be done in both styles:

functional: using the fact that Objective Caml is a *strict language*, which means that the argument is evaluated before applying the function. The expression `(f (g x))` is computed by first evaluating `(g x)`, and then passing the result as argument to `f`. With more complex expressions, we can name an intermediate result with the **`let in`** construction, but the idea remains the same: **`let`** *`aux`*`=(g x)` **`in`** `(f aux)`.

imperative: using sequences or other control structures (loops). In this case, the result is not the value returned by a function, but its side effects on memory: `aux:=(g x) ; (f !aux)`.

Let us examine this choice of style on an example. The *quick sort* algorithm, applied to a vector, is described recursively as follows:

1. Choose a pivot: This is the index of an element of the vector;
2. Permute around the pivot: Permute the elements of the vector so elements less than the value at the pivot have indices less than the pivot, and vice versa;
3. sort the subvectors obtained on each side of the pivot, using the same algorithm: The subvector preceding the pivot and the subvector following the pivot.

The choice of algorithm, namely to modify a vector so that its elements are sorted, incites us to use an imperative style at least to manipulate the data.

First, we define a function to permute two elements of a vector:

```
# let permute_element vec n p =
    let aux = vec.(n) in vec.(n) <- vec.(p) ; vec.(p) <- aux  ;;
val permute_element : 'a array -> int -> int -> unit = <fun>
```

The choice of a good pivot determines the efficiency of the algorithm, but we will use the simplest possible choice here: return the index of the first element of the (sub)vector.

```
# let choose_pivot vec start finish = start ;;
val choose_pivot : 'a -> 'b -> 'c -> 'b = <fun>
```

Let us write the algorithm that we would like to use to permute the elements of the vector around the pivot.

1. Place the pivot at the beginning of the vector to be permuted;
2. Initialize i to the index of the second element of the vector;
3. Initialize j to the index of the last element of the vector;
4. If the element at index j is greater than the pivot, permute it with the element at index i and increment i; otherwise, decrement j;
5. While $i < j$, repeat the previous operation;
6. At this stage, every element with index $< i$ (or equivalently, j) is less than the pivot, and all others are greater; if the element with index i is less than the pivot, permute it with the pivot; otherwise, permute its predecessor with the pivot.

In implementing this algorithm, it is natural to adopt imperative control structures.

```
# let permute_pivot vec start finish ind_pivot =
      permute_element vec start ind_pivot ;
      let i = ref (start+1) and j = ref finish and pivot = vec.(start) in
      while !i < !j do
        if vec.(!j) >= pivot then decr j
        else
          begin
            permute_element vec !i !j ;
            incr i
          end
      done ;
      if vec.(!i) > pivot then decr i ;
      permute_element vec start !i ;
      !i
   ;;
val permute_pivot : 'a array -> int -> int -> int -> int = <fun>
```

In addition to its effects on the vector, this function returns the index of the pivot as its result.

All that remains is to put together the different stages and add the recursion on the sub-vectors.

```
# let rec quick vec start finish =
      if start < finish
      then
        let pivot = choose_pivot vec start finish in
        let place_pivot = permute_pivot vec start finish pivot in
        quick (quick vec start (place_pivot-1)) (place_pivot+1) finish
      else vec ;;
val quick : 'a array -> int -> int -> 'a array = <fun>
```

We have used the two styles here. The chosen pivot serves as argument to the permutation around this pivot, and the index of the pivot after the permutation is an argument to the recursive call. By contrast, the vector obtained after the permutation is not returned by the `permute_pivot` function; instead, this result is produced by side

effect. However, the `quick` function returns a vector, and the sorting of sub-vectors is obtained by composition of recursive calls.

The main function is:

```
# let quicksort vec = quick vec 0 ((Array.length vec)-1) ;;
val quicksort : 'a array -> 'a array = <fun>
```

It is a polymorphic function because the order relation `<` on vector elements is itself polymorphic.

```
# let t1 = [|4;8;1;12;7;3;1;9|] ;;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = [|"the"; "little"; "cat"; "is"; "dead"|] ;;
val t2 : string array = [|"the"; "little"; "cat"; "is"; "dead"|]
# quicksort t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
# t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
```

Shared or Copy Values

When the values that we manipulate are not mutable, it does not matter whether they are shared or not.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ] ;;
val a : int list = [1; 2; 3]
# let b = id a ;;
val b : int list = [1; 2; 3]
```

Whether `b` is a copy of the list `a` or the very same list makes no difference, because these are intangible values anyway. But if we put modifiable values in place of integers, we need to know whether modifying one value causes a change in the other.

The implementation of polymorphism in Objective Caml causes immediate values to be copied, and structured values to be shared. Even though arguments are always passed by value, only the pointer to a structured value is copied. This is the case even in the function `id`:

```
# let a = [| 1 ; 2 ; 3 |] ;;
val a : int array = [|1; 2; 3|]
# let b = id a ;;
val b : int array = [|1; 2; 3|]
# a.(1) <- 4 ;;
- : unit = ()
# a ;;
```

```
- : int array = [|1; 4; 3|]
# b ;;
- : int array = [|1; 4; 3|]
```

We have here a genuine programming choice to decide which is the most efficient way to represent a data structure. On one hand, using mutable values allows manipulations in place, which means without allocation, but requires us to make copies sometimes when immutable data would have allowed sharing. We illustrate this here with two ways to implement lists.

```
# type 'a list_immutable = LInil | LIcons of 'a * 'a list_immutable ;;
# type 'a list_mutable = LMnil | LMcons of 'a * 'a list_mutable ref ;;
```

The immutable lists are strictly equivalent to lists built into Objective Caml, while the mutable lists are closer to the style of C, in which a cell is a value together with a reference to the following cell.

With immutable lists, there is only one way to write concatenation, and it requires duplicating the structure of the first list; by contrast, the second list may be shared with the result.

```
# let rec concat l1 l2 = match l1 with
      LInil → l2
    | LIcons (a,l11) → LIcons(a, (concat l11 l2)) ;;
val concat : 'a list_immutable -> 'a list_immutable -> 'a list_immutable =
  <fun>

# let li1 = LIcons(1, LIcons(2, LInil))
  and li2 =  LIcons(3, LIcons(4, LInil)) ;;
val li1 : int list_immutable = LIcons (1, LIcons (2, LInil))
val li2 : int list_immutable = LIcons (3, LIcons (4, LInil))
# let li3 = concat li1 li2 ;;
val li3 : int list_immutable =
  LIcons (1, LIcons (2, LIcons (3, LIcons (4, LInil))))
# li1==li3 ;;
- : bool = false
# let tlLI l =  match l with
    LInil       → failwith "Liste vide"
  | LIcons(_,x) → x ;;
val tlLI : 'a list_immutable -> 'a list_immutable = <fun>
# tlLI(tlLI(li3)) == li2 ;;
- : bool = true
```

From these examples, we see that the first cells of `li1` and `li3` are distinct, while the second half of `li3` is exactly `li2`.

With mutable lists, we have a choice between modifying arguments (function `concat_share`) and creating a new value (function `concat_copy`).

```
# let rec concat_copy l1 l2 =  match l1 with
      LMnil  → l2
    | LMcons (x,l11) → LMcons(x, ref (concat_copy !l11 l2)) ;;
```

```
val concat_copy : 'a list_mutable -> 'a list_mutable -> 'a list_mutable =
  <fun>
```

This first solution, `concat_copy`, gives a result similar to the previous function, `concat`. A second solution shares its arguments with its result fully:

```
# let concat_share l1 l2 =
    match l1 with
      LMnil → l2
    | _     → let rec set_last = function
                  LMnil       → failwith "concat_share : impossible case!!"
                | LMcons(_,l) → if !l=LMnil then l:=l2 else set_last !l
              in
                set_last l1 ;
                l1 ;;
val concat_share : 'a list_mutable -> 'a list_mutable -> 'a list_mutable =
  <fun>
```

Concatenation with sharing does not require any allocation, and therefore does not use the constructor `LMcons`. Instead, it suffices to cause the last cell of the first list to point to the second list. However, this version of concatenation has the potential weakness that it alters arguments passed to it.

```
# let lm1 = LMcons(1, ref (LMcons(2, ref LMnil)))
   and lm2 = LMcons(3, ref (LMcons(4, ref LMnil))) ;;
val lm1 : int list_mutable =
  LMcons (1, {contents=LMcons (2, {contents=LMnil})})
val lm2 : int list_mutable =
  LMcons (3, {contents=LMcons (4, {contents=LMnil})})
# let lm3 = concat_share lm1 lm2 ;;
val lm3 : int list_mutable =
  LMcons (1, {contents=LMcons (2, {contents=LMcons (...)})})
```

We do indeed obtain the expected result for `lm3`. However, the value bound to `lm1` has been modified.

```
# lm1 ;;
- : int list_mutable =
LMcons (1, {contents=LMcons (2, {contents=LMcons (...)})})
```

This may therefore have consequences on the rest of the program.

How to Choose your Style

In a purely functional program, side effects are forbidden, and this excludes mutable data structures, exceptions, and input/output. We prefer, though, a less restrictive definition of the functional style, saying that functions that do not modify their global environment may be used in a functional style. Such a function may manipulate mutable values locally, and may therefore be written in an imperative style, but must not modify global variables, nor its arguments. We permit them to raise exceptions in addition. Viewed from outside, these functions may be considered "black boxes." Their behavior matches a function written in a purely functional style, apart from being able of breaking control flow by raising an exception. In the same spirit, a mutable value which can no longer be modified after initialization may be used in a functional style.

On the other hand, a program written in an imperative style still benefits from the advantages provided by Objective Caml: static type safety, automatic memory management, the exception mechanism, parametric polymorphism, and type inference.

The choice between the imperative and functional styles depends on the application to be developed. We may nevertheless suggest some guidelines based on the character of the application, and the criteria considered important in the development process.

- **choice of data structures**: The choice whether to use mutable data structures follows from the style of programming adopted. Indeed, the functional style is essentially incompatible with modifying mutable values. By contrast, constructing and traversing objects are the same whatever their status. This touches the same issue as "modification in place *vs* copying" on page 99; we return to it again in discussing criteria of efficiency.
- **required data structures**: If a program must modify mutable data structures, then the imperative style is the only one possible. If, on the other hand, you just have to traverse values, then adopting the functional style guarantees the integrity of the data.
  Using recursive data structures requires the use of functions that are themselves recursive. Recursive functions may be defined using either of the two styles, but it is often easier to understand the creation of a value following a recursive definition, which corresponds to a functional approach, than to repeat the recursive processing on this element. The functional style allows us to define generic iterators over the structure of data, which factors out the work of development and makes it faster.
- **criteria of efficiency**: Modification in place is far more efficient than creating a value. When code efficiency is the preponderant criterion, it will usually tip the balance in favor of the imperative style. We note however that the need to avoid sharing values may turn out to be a very hard task, and in the end costlier than copying the values to begin with.
  Being purely functional has a cost. Partial application and using functions passed as arguments from other functions has an execution cost greater than total application of a function whose declaration is visible. Using this eminently functional feature must thus be avoided in those portions of a program where efficiency is crucial.
- **development criteria**: the higher level of abstraction of functional programs permits them to be written more quickly, leading to code that is more compact and contains fewer errors than the equivalent imperative code, which is generally more verbose. The functional style is better suited to the constraints imposed by developing substantial applications. Since each function is not dependent upon its evaluation context, functional can be easily divided into small units that can be examined separately; as a consequence, the code is easier to read.
  Programs written using the functional style are more easily reusable because of its better modularity, and because functions may be passed as arguments to other functions.

These remarks show that it is often a good idea to mix the two programming styles within the same application. The functional programming style is faster to develop and confers a simpler organization to an application. However, portions whose execution time is critical repay being developed in a more efficient imperative style.

Mixing Styles

As we have mentioned, a language offering both functional and imperative characteristics allows the programmer to choose the more appropriate style for each part of the implementation of an algorithm. One can indeed use both aspects in the same function. This is what we will now illustrate.

Closures and Side Effects

The convention, when a function causes a side effect, is to treat it as a procedure and to return the value `()`, of type *unit*. Nevertheless, in some cases, it can be useful to cause the side effect within a function that returns a useful value. We have already used this mixture of the styles in the function `permute_pivot` of quicksort.

The next example is a symbol generator that creates a new symbol each time that it is called. It simply uses a counter that is incremented at every call.

```
# let c = ref 0;;
val c : int ref = {contents=0}
# let reset_symb = function () → c:=0 ;;
val reset_symb : unit -> unit = <fun>
# let new_symb = function s →  c:=!c+1 ; s^(string_of_int !c) ;;
val new_symb : string -> string = <fun>
# new_symb "VAR" ;;
- : string = "VAR1"
# new_symb "VAR" ;;
- : string = "VAR2"
# reset_symb () ;;
- : unit = ()
# new_symb "WAR" ;;
- : string = "WAR1"
# new_symb "WAR" ;;
- : string = "WAR2"
```

The reference `c` may be hidden from the rest of the program by writing:

```
# let (reset_s , new_s) =
    let c = ref 0
    in let f1 () = c := 0
       and f2 s  = c := !c+1 ; s^(string_of_int !c)
    in (f1,f2) ;;
```

```
val reset_s : unit -> unit = <fun>
val new_s : string -> string = <fun>
```

This declaration creates a pair of functions that share the variable `c`, which is local to this declaration. Using these two functions produces the same behavior as the previous definitions.

```
# new_s "VAR";;
- : string = "VAR1"
# new_s "VAR";;
- : string = "VAR2"
# reset_s();;
- : unit = ()
# new_s "WAR";;
- : string = "WAR1"
# new_s "WAR";;
- : string = "WAR2"
```

This example permits us to illustrate the way that closures are represented. A closure may be considered as a pair containing the code (that is, the **`function`** part) as one component and the local envoronment containing the values of the free variables of the function. Figure 4.1 shows the memory representation of the closures `reset_s` and `new_s`.

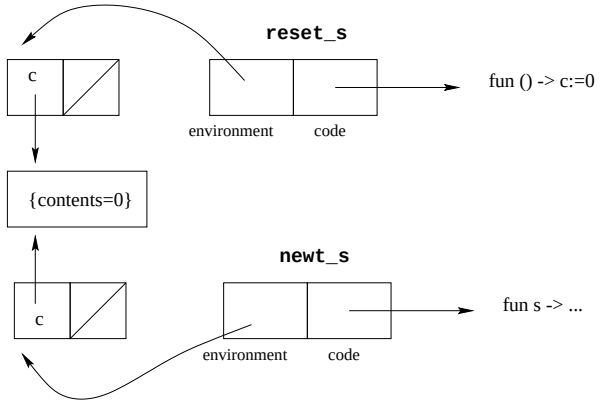

Figure 4.1: Memory representation of closures.

These two closures share the same environment, containing the value of `c`. When either one modifies the reference `c`, it modifies the contents of an area of memory that is shared with the other closure.

Physical Modifications and Exceptions

Exceptions make it possible to escape from situations in which the computation cannot proceed. In this case, an exception handler allows the calculation to continue, knowing that one branch has failed. The problem with side effects comes from the state of the modifiable data when the exception was raised. One cannot be sure of this state if there have been physical modifications in the branch of the calculation that has failed.

Let us define the increment function `(++)` analogous to the operator in C:

```
# let (++) x = x:=!x+1; x;;
val ++ : int ref -> int ref = <fun>
```

The following example shows a little computation where division by zero occurs together with

```
# let x = ref 2;;
val x : int ref = {contents=2}
(* 1 *)
# !((++) x) * (1/0) ;;
Uncaught exception: Division_by_zero
# x;;
- : int ref = {contents=2}
(* 2 *)
# (1/0) * !((++) x) ;;
Uncaught exception: Division_by_zero
# x;;
- : int ref = {contents=3}
```

The variable `x` is not modified during the computation of the expression in (∗1∗), while it is modified in the computation of (∗2∗). Unless one saves the initial values, the form **`try .. with ..`** must not have a **`with ..`** part that depends on modifiable variables implicated in the expression that raised the exception.

Modifiable Functional Data Structures

In functional programming a program (in particular, a function expression) may also serve as a data object that may be manipulated, and one way to see this is to write association lists in the form of function expressions. In fact, one may view association lists of type *`('a * 'b) list`* as partial functions taking a key chosen from the set *`'a`* and returning a value in the set of associated values *`'b`*. Each association list is then a function of type *`'a -> 'b`*.

The empty list is the everywhere undefined function, which one simulates by raising an exception:

```
# let nil_assoc = function x → raise Not_found ;;
val nil_assoc : 'a -> 'b = <fun>
```

We next write the function `add_assoc` which adds an element to a list, meaning that it extends the function for a new entry:

```
# let add_assoc (k,v) l = function x → if x = k then v else l x ;;
val add_assoc : 'a * 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = add_assoc ('1', 1) (add_assoc ('2', 2) nil_assoc) ;;
val l : char -> int = <fun>
# l '2' ;;
- : int = 2
# l 'x' ;;
Uncaught exception: Not_found
```

We may now re-write the function mem_assoc:

```
# let mem_assoc k l = try  (l k) ; true  with  Not_found → false ;;
val mem_assoc : 'a -> ('a -> 'b) -> bool = <fun>
# mem_assoc '2' l ;;
- : bool = true
# mem_assoc 'x' l ;;
- : bool = false
```

By contrast, writing a function to remove an element from a list is not trivial, because one no longer has access to the values captured by the closures. To accomplish the same purpose we mask the former value by raising the exception Not_found.

```
# let rem_assoc k l = function x → if x=k then raise Not_found else l x ;;
val rem_assoc : 'a -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = rem_assoc '2' l ;;
val l : char -> int = <fun>
# l '2' ;;
Uncaught exception: Not_found
```

Clearly, one may also create references and work by side effect on such values. However, one must take some care.

```
# let add_assoc_again (k,v) l =  l := (function x → if x=k then v else !l x) ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
```

The resulting value for l is a function that points at itself and therefore loops. This annoying side effect is due to the fact that the dereferencing !l is within the scope of the closure **function** *x* →. The value of !l is not evaluated during compilation, but at run-time. At that time, l points to the value that has already been modified by add_assoc. We must therefore correct our definition using the closure created by our original definition of add_assoc:

```
# let add_assoc_again (k, v) l =  l := add_assoc (k, v) !l ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
# let l = ref nil_assoc ;;
val l : ('_a -> '_b) ref = {contents=<fun>}
# add_assoc_again ('1',1) l ;;
- : unit = ()
```

```
# add_assoc_again ('2',2) l ;;
- : unit = ()
# !l '1' ;;
- : int = 1
# !l 'x' ;;
Uncaught exception: Not_found
```

Lazy Modifiable Data Structures

Combining imperative characteristics with a functional language produces good tools for implementing computer languages. In this subsection, we will illustrate this idea by implementing data structures with deferred evaluation. A data structure of this kind is not completely evaluated. Its evaluation progresses according to the use made of it.

Deferred evaluation, which is often used in purely functional languages, is simulated using function values, possibly modifiable. There are at least two purposes for manipulating incompletely evaluated data structures: first, so as to calculate only what is effectively needed in the computation; and second, to be able to work with potentially infinite data structures.

We define the type `vm`, whose members contain either an already calculated value (constructor `Imm`) or else a value to be calculated (constructor `Deferred`):

```
# type 'a v =
      Imm of 'a
    | Deferred of (unit → 'a);;
# type 'a vm = {mutable c : 'a v };;
```

A computation is deferred by encapsulating it in a closure. The evaluation function for deferred values must return the value if it has already been calculated, and otherwise, if the value is not already calculated, it must evaluate it and then store the result.

```
# let eval e = match e.c with
    Imm a → a
  | Deferred f → let u = f () in e.c <- Imm u ; u ;;
val eval : 'a vm -> 'a = <fun>
```

The operations of deferring evaluation and activating it are also called *freezing* and *thawing* a value.

We could also write the conditional control structure in the form of a function:

```
# let if_deferred c e1 e2 =
    if eval c then eval e1 else eval e2;;
val if_deferred : bool vm -> 'a vm -> 'a vm -> 'a = <fun>
```

Here is how to use it in a recursive function such as factorial:

```
# let rec facr n =
    if_deferred
          {c=Deferred(fun () → n = 0)}
          {c=Deferred(fun () →  1)}
          {c=Deferred(fun () → n*(facr(n-1)))};;
val facr : int -> int = <fun>
# facr 5;;
- : int = 120
```

The classic form of **if** can not be written in the form of a function. In fact, if we define a function `if_function` this way:

```
# let if_function c e1 e2 = if c then e1 else e2;;
val if_function : bool -> 'a -> 'a -> 'a = <fun>
```

then the three arguments of `if_function` are evaluated at the time they are passed to the function. So the function `fact` loops, because the recursive call *fact*(*n*-1) is always evaluated, even when `n` has the value `0`.

```
# let rec fact n = if_function (n=0) 1 (n*fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
Stack overflow during evaluation (looping recursion?).
```

Module Lazy

The implementation difficulty for frozen values is due to the conflict between the eager evaluation strategy of Objective Caml and the need to leave expressions unevaluated. Our attempt to redefine the conditional illustrated this. More generally, it is impossible to write a function that freezes a value in producing an object of type *vm*:

```
# let freeze e = { c = Deferred (fun () → e) };;
val freeze : 'a -> 'a vm = <fun>
```

When this function is applied to arguments, the Objective Caml evaluation strategy evaluates the expression `e` passed as argument before constructing the closure **fun** () → *e*. The next example shows this:

```
# freeze (print_string "trace"; print_newline(); 4*5);;
trace
- : int vm = {c=Deferred <fun>}
```

This is why the following syntactic form was introduced.

Syntax : **lazy** *expr*

Warning This form is a language extension that may evolve in future versions.

When the keyword **lazy** is applied to an expression, it constructs a value of a type declared in the module Lazy:

```
# let x = lazy (print_string "Hello"; 3*4) ;;
val x : int Lazy.status ref = {contents=Lazy.Delayed <fun>}
```

The expression (*print_string* "Hello") has not been evaluated, because no message has been printed. The function force of module Lazy allows one to force evaluation:

```
# Lazy.force x ;;
Hello- : int = 12
```

Now the value x has altered:

```
# x ;;
- : int Lazy.t = {contents=Lazy.Value 12}
```

It has become the value of the expression that had been frozen, namely 12.

For another call to the function force, it's enough to return the value already calculated:

```
# Lazy.force x ;;
- : int = 12
```

The string "Hello" is no longer prefixed.

"Infinite" Data Structures

The second reason to defer evaluation is to be able to construct potentially infinite data structures such as the set of natural numbers. Because it might take a long time to construct them all, the idea here is to compute only the first one and to know how to pass to the next element.

We define a generic data structure *'a enum* which will allow us to enumerate the elements of a set.

```
# type 'a enum = { mutable i : 'a; f :'a → 'a } ;;
type 'a enum = { mutable i: 'a; f: 'a -> 'a }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x ;;
val next : 'a enum -> 'a = <fun>
```

Now we can get the set of natural numbers by instantiating the fields of this structure:

```
# let nat = { i=0; f=fun x → x + 1 };;
val nat : int enum = {i=0; f=<fun>}
# next nat;;
- : int = 0
# next nat;;
- : int = 1
# next nat;;
- : int = 2
```

Another example gives the elements of the Fibonnacci sequence, which has the defini-

tion:

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_{n+2} = u_n + u_{n+1} \end{cases}$$

The function to compute the successor must take account of the current value, (u_{n-1}), but also of the preceding one (u_{n-2}). For this, we use the state `c` in the following closure:

```
# let fib = let fx = let c = ref 0 in fun v → let r = !c + v in c:=v ; r
            in { i=1 ; f=fx } ;;
val fib : int enum = {i=1; f=<fun>}
# for i=0 to 10 do  print_int (next fib); print_string "  " done ;;
1  1  2  3  5  8  13  21  34  55  89  - : unit = ()
```

Streams of Data

Streams are (potentially infinite) sequences containing elements of the same kind. The evaluation of a part of a stream is done on demand, whenever it is needed by the current computation. A stream is therefore a *lazy* data structure.

The *stream* type is an abstract data type; one does not need to know how it is implemented. We manipulate objects of this type using constructor functions and destructor (or selector) functions. For the convenience of the user, Objective Caml has simple syntactic constructs to construct streams and to access their elements.

Warning Streams are an extension of the language, not part of the stable core of Objective Caml.

Construction

The syntactic sugar to construct streams is inspired by that for lists and arrays. The empty stream is written:

```
# [< >] ;;
- : 'a Stream.t = <abstr>
```

One may construct a stream by enumerating its elements, preceding each one with an with a single quote (character `'`):

```
# [< '0; '2; '4 >] ;;
- : int Stream.t = <abstr>
```

Expressions not preceded by an apostrophe are considered to be sub-streams:

```
# [< '0; [< '1; '2; '3 >]; '4 >] ;;
- : int Stream.t = <abstr>
# let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
```

```
- : int Stream.t = <abstr>
# let concat_stream a b = [< a ; b >] ;;
val concat_stream : 'a Stream.t -> 'a Stream.t -> 'a Stream.t = <fun>
# concat_stream [< '"if"; '"c";'"then";'"1" >] [< '"else";'"2" >] ;;
- : string Stream.t = <abstr>
```

The `Stream` module also provides other construction functions. For instance, the functions `of_channel` and `of_string` return a stream containing a sequence of characters, received from an input stream or a string.

```
# Stream.of_channel ;;
- : in_channel -> char Stream.t = <fun>
# Stream.of_string ;;
- : string -> char Stream.t = <fun>
```

The deferred computation of streams makes it possible to manipulate infinite data structures in a way similar to the type *'a enum* defined on page 109. We define the stream of natural numbers by its first element and a function calculating the stream of elements to follow:

```
# let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
val nat_stream : int -> int Stream.t = <fun>
# let nat = nat_stream 0 ;;
val nat : int Stream.t = <abstr>
```

Destruction and Matching of Streams

The primitive `next` permits us to evaluate, retrieve, and remove the first element of a stream, all at once:

```
# for i=0 to 10 do
    print_int (Stream.next nat) ;
    print_string "  "
    done ;;
0  1  2  3  4  5  6  7  8  9  10  - : unit = ()
# Stream.next nat ;;
- : int = 11
```

When the stream is exhausted, an exception is raised.

```
# Stream.next [< >] ;;
Uncaught exception: Stream.Failure
```

To manipulate streams, Objective Caml offers a special-purpose matching construct called *destructive matching*. The value matched is calculated and removed from the stream. There is no notion of exhaustive match for streams, and, since the data type is lazy and potentially infinite, one may match less than the whole stream. The syntax for matching is:

Syntax : **match** *expr* **with parser [< '**p_1 ... **>] ->** $expr_1$ **|** ...

The function `next` could be written:

```
# let next s = match s with parser [< 'x >] → x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat;;
- : int = 12
```

Note that the enumeration of natural numbers picks up where we left it previously.

As with function abstraction, there is a syntactic form matching a function parameter of type *Stream.t*.

Syntax : **parser** *p -¿ ...*

The function `next` can thus be rewritten:

```
# let next = parser [<'x>] → x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat ;;
- : int = 13
```

It is possible to match the empty stream, but take care: the stream pattern `[<>]` matches every stream. In fact, a stream `s` is always equal to the stream `[< [<>]; s >]`. For this reason, one must reverse the usual order of matching:

```
# let rec it_stream f s =
    match s with parser
      [< 'x ; ss >] → f x ; it_stream f ss
    | [<>] → () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# let print_int1 n = print_int n ; print_string" " ;;
val print_int1 : int -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Since matching is destructive, one can equivalently write:

```
# let rec it_stream f s =
    match s with parser
      [< 'x >] → f x ; it_stream f s
    | [<>] → () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Although streams are lazy, they want to be helpful, and never refuse to furnish a first element; when it has been supplied once it is lost. This has consequences for matching. The following function is an attempt (destined to fail) to display pairs from a stream of integers, except possibly for the last element.

```
# let print_int2 n1
  n2 =
```

```
    print_string "(" ; print_int n1 ; print_string "," ;
    print_int n2 ; print_string ")" ;;
val print_int2 : int -> int -> unit = <fun>
# let rec print_stream s =
    match s with parser
      [< 'x; 'y >] → print_int2 x y; print_stream s
    | [< 'z >] → print_int1 z; print_stream s
    | [<>] → print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)Uncaught exception: Stream.Error("")
```

The first two two members of the stream were displayed properly, but during the evaluation of the recursive call (*print_stream* `[<3>]`), the first pattern found a value for `x`, which was thereby consumed. There remained nothing more for `y`. This was what caused the error. In fact, the second pattern is useless, because if the stream is not empty, then first pattern always begins evaluation.

To obtain the desired result, we must sequentialize the matching:

```
# let rec print_stream s =
    match s with parser
      [< 'x >]
       → (match s with parser
             [< 'y >] → print_int2 x y; print_stream s
           | [<>] → print_int1 x; print_stream s)
    | [<>] → print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)3
- : unit = ()
```

If matching fails on the first element of a pattern however, then we again have the familiar behavior of matching:

```
# let rec print_stream s =
    match s with parser
      [< '1; 'y >] → print_int2 1 y; print_stream s
    | [< 'z >] → print_int1 z; print_stream s
    | [<>] → print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>] ;;
(1,2)3
- : unit = ()
```

The Limits of Matching

Because it is destructive, matching streams differs from matching on sum types. We will now illustrate how radically different it can be.

We can quite naturally write a function to compute the sum of the elements of a stream:

```
# let rec sum s =
    match s with parser
      [< 'n; ss >] → n+(sum ss)
    | [<>] → 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

However, we can just as easily consume the stream from the inside, naming the partial result:

```
# let rec sum s =
    match s with parser
      [< 'n; r = sum >] → n+r
    | [<>] → 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

We will examine some other important uses of streams in chapter 11, which is devoted to lexical and syntactic analysis. In particular, we will see how consuming a stream from the inside may be profitably used.

Exercises

Binary Trees

We represent binary trees in the form of vectors. If a tree a has height h, then the length of the vector will be $2^{(h+1)}-1$. If a node has position i, then the left subtree of this node lies in the interval of indices $[i+1\ ,\ i+1+2^h]$, and its right subtree lies in the interval $[i+1+2^h+1\ ,\ 2^{(h+1)}-1]$. This representation is useful when the tree is almost completely filled. The type `'a` of labels for nodes in the tree is assumed to contain a special value indicating that the node does not exist. Thus, we represent labeled trees by the by vectors of type `'a array`.

1. Write a function , taking as input a binary tree of type `'a bin_tree` (defined on page 50) and an array (which one assumes to be large enough). The function stores the labels contained in the tree in the array, located according to the discipline described above.
2. Write a function to create a leaf (tree of height 0).

3. Write a function to construct a new tree from a label and two other trees.
4. Write a conversion function from the type *'a bin_tree* to an array.
5. Define an infix traversal function for these trees.
6. Use it to display the tree.
7. What can you say about prefix traversal of these trees?

Spelling Corrector

The exercise uses the lexical tree , from the exercise of chapter 2, page 63, to build a spelling corrector.

1. Construct a dictionary from a file in ASCII in which each line contains one word. For this, one will write a function which takes a file name as argument and returns the corresponding dictionary.
2. Write a function `words` that takes a character string and constructs the list of words in this string. The word separators are space, tab, apostrophe, and quotation marks.
3. Write a function `verify` that takes a dictionary and a list of words, and returns the list of words that do not occur in the dictionary.
4. Write a function `occurrences` that takes a list of words and returns a list of pairs associating each word with the number of its occurrences.
5. Write a function `spellcheck` that takes a dictionary and the name of a file containing the text to analyze. It should return the list of incorrect words, together with their number of occurrences.

Set of Prime Numbers

We would like now to construct the infinite set of prime numbers (without calculating it completely) using lazy data structures.

1. Define the predicate `divisible` which takes an integer and an initial list of prime numbers, and determines whether the number is divisible by one of the integers on the list.
2. Given an initial list of prime numbers, write the function `next` that returns the smallest number not on the list.
3. Define the value *setprime* representing the set of prime numbers, in the style of the type *'a enum* on page 109. It will be useful for this set to retain the integers already found to be prime.

Summary

This chapter has compared the functional and imperative programming styles. They differ mainly in the control of execution (implicit in functional and explicit in impera-

tive programming), and in the representation in memory of data (sharing or explicitly copied in the imperative case, irrelevant in the functional case). The implementation of algorithms must take account of these differences. The choice between the two styles leads in fact to mixing them. This mixture allows us to clarify the representation of closures, to optimize crucial parts of applications, and to create mutable functional data. Physical modification of values in the environment of a closure permits us to better understand what a functional value is. The mixture of the two styles gives powerful implementation tools. We used them to construct potentially infinite values.

To Learn More

The principal consequences of adding imperative traits to a functional language are:

- To determine the evaluation strategy (strict evaluation);
- to add implementation constraints, especially for the GC (see Chapter 9);
- For statically typed languages, to make their type system more complex;
- To offer different styles of programming in the same language, permitting us to program in the style appropriate to the algorithm at hand, or possibly in a mixed style.

This last point is important in Objective Caml where we need the same parametric polymorphism for functions written in either style. For this, certain purely functional programs are no longer typable after the addition. Wright's article ([Wri95]) explains the difficulties of polymorphism in languages with imperative aspects. Objective Caml adopts the solution that he advocates. The classification of different kinds of polymorphism in the presence of physical modification is described well in the thesis of Emmanuel Engel ([Eng98]).

These consequences make the job of programming a bit harder, and learning the language a bit more difficult. But because the language is richer for this reason and above all offers the choice of style, the game is worth the candle. For example, strict evaluation is the rule, but it is possible to implement basic mechanisms for lazy evaluation, thanks to the mixture of the two styles. Most purely functional languages use a lazy evaluation style. Among languages close to ML, we would mention Miranda, LazyML, and Haskell. The first two are used at universities for teaching and research. By contrast, there are significant applications written in Haskell. The absence of controllable side effects necessitates an additional abstraction for input/output called *monads*. One can read works on Haskell (such as [Tho99]) to learn more about this subject. Streams are a good example of the mixture of functional and imperative styles. Their use in lexical and syntactic analysis is described in Chapter 11.