16
Loops and procedural programming

Back in Section 7.3.1, we implemented a function to compute the length of a list, by capturing how the length is *defined*: the length of the empty list is 0; the length of a non-empty list is one more than the length of its tail. This definition can be immediately cashed out as

```
# let rec length (lst : 'a list) : int =
#   match lst with
#   | [] -> 0
#   | _hd :: tl -> 1 + length tl ;;
val length : 'a list -> int = <fun>
```

An alternative approach, in the spirit of imperative programming, is to think not about what the length *is* but about what one *does* when calculating the length: For each element of the list, add one to a counter until the end of the list is reached.

This approach – which we might term PROCEDURAL PROGRAMMING because it emphasizes the steps in the procedure to be carried out – is typical of how introductory programming is taught, with an emphasis on *commands* with *side effects* that are executed repeatedly through *loops*.

In this chapter, we'll provide examples of procedural programming, emphasizing one of the main benefits of the paradigm, SPACE EFFICIENCY. Procedural programming can be more space efficient in a couple of ways. First, it can reduce the need for storing suspended computations in so-called "stack frames", though as we'll see, the functional language technique of tail-recursion optimization can provide this benefit as well. Second, it can reduce the need for copying data structures as they are manipulated.

Although OCaml is at its core a functional programming language, it supports procedural programming as well. There are, for instance,

`while` loops:

$\langle expr \rangle$ ::= `while` $\langle expr_{condition} \rangle$ `do`
$\langle expr_{body} \rangle$
`done`

which specify that the body expression be executed repeatedly so long as the condition expression is `true`.

In addition, the `for` loop, familiar from other procedural languages, is expressed as follows to count up from a start value to an end value:

$\langle expr \rangle$ ::= `for` $\langle var \rangle$ `=` $\langle expr_{start} \rangle$ `to` $\langle expr_{end} \rangle$ `do`
$\langle expr_{body} \rangle$
`done`

or, counting down,

$\langle expr \rangle$ ::= `for` $\langle var \rangle$ `=` $\langle expr_{start} \rangle$ `to` $\langle expr_{end} \rangle$ `do`
$\langle expr_{body} \rangle$
`done`

16.1 Loops require impurity

In a pure language, an expression in a given context always has the same value. Thus, in a `while` loop of the form

`while` $\langle expr_{condition} \rangle$ `do`
$\langle expr_{body} \rangle$
`done`

if the condition expression $\langle expr_{condition} \rangle$ is `true` the first time it's evaluated, it will remain so perpetually and the loop will never terminate. Conversely, if the condition expression is `false` the first time it's evaluated, it will remain so perpetually and the loop body will never be evaluated. Similarly, the body expression $\langle expr_{body} \rangle$ will always evaluate to the same value, so what could possibly be the point of evaluating it more than once?

In summary, procedural programming only makes sense in a language with *side effects*, the kind of impure constructs (like variable assignment) that we introduced in the previous chapter. You can see this need in attempting to implement the `length` function in this procedural paradigm. Here is a sketch of a procedure for calculating the length of a list:

```
let length (lst : 'a list) : int =
  (* initialize the counter *)
  while (* the list is not empty *) do
    (* increment the counter *)
    (* drop an element from the list *)
  done;
  (* return the counter *) ;;
```

We'll need to establish the counter in such a way that its value can change. Similarly, we'll need to update the list each time the loop body is executed. We'll thus need both the counter and the list being manipulated to be references, so that they can change. Putting all this together, we get the following procedure for computing the length of a list:

```
# let length_iter (lst : 'a list) : int =
#   let counter = ref 0 in        (* initialize the counter *)
#   let lst_ref = ref lst in      (* initialize the list *)
#   while !lst_ref <> [] do       (* while list not empty... *)
#     incr counter;               (*   increment the counter *)
#     lst_ref := List.tl !lst_ref (*   drop element from list *)
#   done;
#   !counter ;;                   (* return the counter  value *)
val length_iter : 'a list -> int = <fun>

# length_iter [1; 2; 3; 4; 5] ;;
- : int = 5
```

16.2 Recursion versus iteration

Is this impure, iterative, procedural method better than the pure, recursive, functional approach? It certainly seems more complex, and gaining an understanding that it provides the correct values as specified in the definition of list length is certainly more difficult.

16.2.1 Saving stack space

There is one way, however, in which this approach might be superior. Think of the calculation of the length of a list, say `[1; 2; 3]`, using the functional definition. Since the list is non-empty, we need to add one to the result of evaluating `length [2; 3]`, and we'll need to suspend the addition until that evaluation completes. Likewise, to evaluate `length [2; 3]` we'll need to add one to the result of evaluating `length [3]`, again suspending the addition until *that* evaluation completes. Continuing on in this way, at run time we'll eventually have a nested stack of suspended calls. Each element of this stack, carrying information about the suspended computation, is referred to as a STACK FRAME. Only once we reach `length []` can we start unwinding this stack, performing all of the suspended additions specified in

the stack frames, to calculate the final answer. Figure 16.1 depicts this linearly growing stack of suspended calls.

The iterative approach, on the other hand, needs no stack of suspended computations. The single call to `length_iter` invokes the `while` loop to iteratively increment the counter and drop elements from the list. The computation is "flat".

The difference can be seen forcefully when computing the length of a very long list. Here, we've defined `very_long_list` to be a list with one million elements.

```
# let very_long_list = List.init 1_000_000 (fun x -> x) ;;
val very_long_list : int list = [0; 1; 2; 3; 4; 5; 6; 7; ...]
```

The iterative procedure for computing its length works well.

```
# length_iter very_long_list ;;
- : int = 1000000
```

But the functional recursive version overflows the stack dedicated to storing the suspended computations. Apparently, one million stack frames is more than the computer has space for.

```
# length very_long_list ;;
Stack overflow during evaluation (looping recursion?).
```

```
length [1; 2; 3]
⇒ 1 + length [2; 3]
⇒ 1 + (1 + length [3])
⇒ 1 + (1 + (1 + length []))
⇒ 1 + (1 + (1 + 0))
⇒ 1 + (1 + 1)
⇒ 1 + 2
⇒ 3
```

Figure 16.1: The nested stack of suspended calls in evaluating a non-tail-recursive `length` function. We indicate each stack frame with a highlighted box. Notice that the number of stack frames increases as each recursive call is generated.

16.2.2 Tail recursion

The profligate use of space for stack frames is not inherent in all purely functional recursive computations however. Consider the following purely functional method `length_tr` for implementing the length calculation.

```
# let length_tr lst =
#   let rec length_plus lst acc =
#     match lst with
#     | [] -> acc
#     | _hd :: tl -> length_plus tl (1 + acc) in
#   length_plus lst 0 ;;
val length_tr : 'a list -> int = <fun>
```

Here, a local auxiliary function `length_plus` takes two arguments, the list and an integer accumulator of the count of elements counted so far. It returns *the length of its list argument plus the value of its accumulator*. Thus, the call to `length_plus lst 0` calculates the the length of `lst` plus 0, which is just the length desired.

This `length_tr` version of calculating list length still operates recursively; `length_plus` is the locus of the recursion as indicated by the `rec` keyword. The nesting of recursive calls proceeds as shown in Figure 16.2.

```
length_tr [1; 2; 3]
⇒ length_plus [1; 2; 3] 0
⇒ length_plus [2; 3] 1
⇒ length_plus [3] 2
⇒ length_plus [] 3
⇒ 3
```

Figure 16.2: The call structure in evaluating a tail-recursive `length` function. Note the lack of nesting of suspended calls.

As with the previous recursive version, the number of such recursive computations is linear in the length of the list. One might think, then, that the same problem of stack overflow will haunt the `length_tr` implementation as well. Let's try it.

```
# length_tr very_long_list ;;
- : int = 1000000
```

This version doesn't have the same problem. It's easy to see why. For the recursive `length`, the result of each call is a computation using the result of the embedded call to `length`; that computation must therefore be suspended, and a stack frame must be allocated to store information about that pending computation. But the result of each call to the recursive `length_plus` is not just a *computation using* the result of the embedded call to `length_plus`; it *is* the result of that nested call. We don't need to store any information about a suspended computation – no need to allocate a stack frame – because the embedded call result is all that is needed.

Recursive programs written in this way, in which the recursive invocation *is* the result of the invoking call, are deemed TAIL RECURSIVE (hence the `_tr` in the function's name). Tail-recursive functions need not use a stack to keep track of suspended computations. Programming language implementations that take advantage of this possibility by not allocating a stack frame to tail-recursive applications are said to perform TAIL-RECURSION OPTIMIZATION, effectively turning the recursion into a corresponding iteration, and yielding the benefits of the procedural iterative solution. The OCaml interpreter is such a language implementation.

Thus, this putative advantage of loop-based procedures over recursive functions – the ability to perform computations space-efficiently – can often be replicated in functional style through careful tail-recursive implementation where needed.

You'll see discussion of this issue, for instance, in the description of functions in the `List` library, which calls out those functions that are not tail-recursive.[1] For instance, the library function `fold_left` is implemented in a tail-recursive manner, so it can fold over very long lists without running out of stack space. By contrast, the `fold_right` implementation is not tail-recursive, so may not be appropriate when processing extremely long lists.

[1] From the `List` library documentation: "Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. . . . The above considerations can usually be ignored if your lists are not longer than about 10000 elements."

16.3 Saving data structure space

Another advantage of procedural programming is the ability to avoid building of new data structures. Think of the `map` function over lists, which can be implemented as follows:

```
# let rec map (fn : 'a -> 'b) (lst : 'a list) : 'b list =
#   match lst with
#   | [] -> []
#   | hd :: tl -> fn hd :: map fn tl ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

We can use `map` to increment the values in a list:

```
# let original = [1; 2; 3] ;;
val original : int list = [1; 2; 3]
# map succ original ;;
- : int list = [2; 3; 4]
```

The result is a list with different values. Most notably, the result is a `new` list. The `original` is unchanged.

```
# original ;;
- : int list = [1; 2; 3]
```

The new list is created by virtue of the repeated construction of conses with the `::` operator highlighted in the `map` definition above. Every time `map` is called to operate over a list, more conses will be needed. There's no free lunch here. Under the hood, every cons takes up space; storage must be allocated for each one. If we start with a list of length n, we'll end up allocating n more conses to compute the map.

16.3.1 Problem section: Metering allocations

We can determine how many allocations are going on by metering them. Imagine there were a module `Metered` satisfying the following signature:

```
# module type METERED =
#   sig
#     (* reset () -- Resets the count of allocations *)
#     val reset : unit -> unit
#     (* count () -- Returns the number of allocations
#        since the last reset *)
#     val count : unit -> int
#     (* cons hd tl -- Returns the list cons of `hd` and
#        `tl`, increasing the allocation count accordingly *)
#     val cons : 'a -> 'a list -> 'a list
#     (* pair first second -- Returns the pair of `first`
#        and `second`, increasing the allocation count
#        accordingly *)
#     val pair : 'a -> 'b -> 'a * 'b
#   end ;;
module type METERED =
  sig
    val reset : unit -> unit
    val count : unit -> int
    val cons : 'a -> 'a list -> 'a list
    val pair : 'a -> 'b -> 'a * 'b
  end
```

The functions `cons` and `pair` could be used to replace their built-in counterparts for consing (`::`) and pairing (`,`) to track the number of allocations required.

Problem 168

Implement the module `Metered`.

Problem 169

Reimplement the `zip` function of Section 10.3.2 using metered conses and pairs.

Having metered the `zip` function, we can observe the count of allocations.

```
# Metered.reset () ;;
- : unit = ()
# zip [1; 2; 3; 4; 5] [5; 4; 3; 2; 1] ;;
- : (int * int) list = [(1, 5); (2, 4); (3, ...); ...]
# Metered.count () ;;
- : int = 10
```

16.3.2 *Reusing space through mutable data structures*

Now consider, by contrast to the functional `map` over lists above, a function (call it `map_array`) to map a function over a mutable data structure, an array. Instead of returning a new data structure, we'll mutate the values in the original data structure. For that reason, `map_array` doesn't itself need to return an array. [2]

```
# let map_array (fn : 'a -> 'a) (arr : 'a array) : unit =
#   for i = 0 to Array.length arr - 1 do
#     arr.(i) <- fn arr.(i)
#   done ;;
val map_array : ('a -> 'a) -> 'a array -> unit = <fun>
```

[2] The function being applied must be of type `'a -> 'a` since the output of the function is being stored in the same location as the input, and must thus be of the same type. For that reason, `map_array` can't be as polymorphic as `map`.

We can perform a similar computation, mapping the successor function over the elements of an array.

```
# let original = [|1; 2; 3|] ;;
val original : int array = [|1; 2; 3|]
# map_array succ original ;;
- : unit = ()
```

We see the effect of the map this time not in the return value but in the modified `original` array.

```
# original ;;
- : int array = [|2; 3; 4|]
```

By using imperative techniques, we gain access to the incremented values, and without incurring the cost of allocating further storage. There is a cost, however. We no longer have access to the original `unincremented` values. They've been destroyed, replaced by the new values. Under what conditions the tradeoff – reduced storage versus loss of access to prior results – is a judgement call. But as an issue of efficiency, we'd want to heed Knuth's warning against premature optimization.

16.4 In-place sorting

As another example of the use of procedural programming to reduce storage requirements, we consider one of the most elegant sorting algorithms, QUICKSORT. Quicksort works by selecting a *pivot* value – the first element of the list, say – and partitioning the list into those elements less than the pivot and those that are greater. The two sub-lists are recursively sorted, and then concatenated to form the final sorted list. A recursive implementation of quicksort, following the `SORT` signature of Section 14.2, is as follows:

```
# module QuickSort : SORT =
#   struct
#     (* partition lt pivot xs -- Returns two lists
#        constituting all elements in `xs` less than (according
#        to `lt`) than the `pivot` value and greater than the
#        pivot `value`, respectively *)
#     let rec partition lt pivot xs =
#       match xs with
#       | [] -> [], []
#       | hd :: tl ->
#          let first, second = partition lt pivot tl in
#          if lt hd pivot then hd :: first, second
#          else first, hd :: second
#
#     (* sort lt xs -- Quicksorts `xs` according to the comparison
#        function `lt` *)
#     let rec sort (lt : 'a -> 'a -> bool)
#                  (xs : 'a list)
#                : 'a list =
#       match xs with
#       | [] -> []
#       | pivot :: rest ->
#          let first, second = partition lt pivot rest in
#          sort lt first @ [pivot] @ sort lt second
#   end ;;
module QuickSort : SORT
```

Problem 170

Implement a metered version of quicksort, and experiment with how many allocations are needed to sort lists of different lengths.

Just as we built a version of `map` that mutated an array to map over its elements, we can build a version of quicksort that mutates an array to sort its elements. This approach, IN-PLACE sorting, is much more space-efficient. As we'll see, though, there is a cost in transparency of the implementation.

The type for an in-place sort differs from its pure alternative, which allocates extra space. A signature for an in-place sorting module makes clear the differences.

```
# module type SORT_IN_PLACE =
#   sig
```

```
#     (* sort lt xs -- Sorts the array `xs` in place in increasing
#        order by the "less than" function `lt`. *)
#     val sort : ('a -> 'a -> bool) -> 'a array -> unit
#   end ;;
module type SORT_IN_PLACE =
  sig val sort : ('a -> 'a -> bool) -> 'a array -> unit end
```

First, we're sorting a mutable data structure, an array, rather than a list. Second, the `sort` function returns a `unit` as it works by side effect rather than by returning a sorted version of the unchanged argument list. The sorting function, then, begins with a header line

```
let sort (lt : 'a -> 'a -> bool) (arr : 'a array) : unit =
```

The primitive operation of in-place sorting is the swapping of two elements in the array, specified by their indices. We'll make use of a function `swap` to perform this operation.

```
let swap (i : int) (j : int) : unit =
  let temp = arr.(i) in
  arr.(i) <- arr.(j);
  arr.(j) <- temp
```

We'll need to partition a *region* of the array, by which we mean a contiguous subportion of the array between two indices. For that purpose, we'll have a function `partition` that takes two indices (`left` and `right`) demarcating the region to partition (the elements between the indices inclusive). The `partition` function returns the index of the split point in the region, the position that marks the border between the left partition and the right partition where the pivot element resides. We note that for our purposes, there should and will always be at least two elements in the region; otherwise, no recursive sorting is necessary, hence no need to partition.

To partition the region, we select the leftmost element as the pivot. We keep a "current" index that moves from left to right as we process each element in the region. At the same time, we maintain a moving "border" index, again moving from left to right. At any point, all of the elements to the left of the border will be guaranteed to be less than the pivot value. Those between the border and the current index are greater than or equal to the pivot. Those to the right of the current index are yet to be processed. Eventually, when we've processed all elements, we swap the pivot element itself into the correct position at the border. Here's the implementation of this quite subtle process:

```
let partition (left : int) (right : int) : int =

  (* region has at least two elements *)
  assert (left < right);
```

```
  (* select the pivot element to be the first element in
     the region *)
  let pivot_val = arr.(left) in
  (* all elements to the left of `border` are guaranteed
     to be strictly less than pivot value *)
  let border = ref (left + 1) in
  (* current element being partitioned, starting just
     after pivot *)
  let current = ref (left + 1) in

  (* process each element, moving those less than the
     pivot to before the border *)
  while !current <= right do
    if lt arr.(!current) pivot_val then
      begin
        (* current should be left of pivot *)
        swap !current !border; (* swap into place at border *)
        incr border (* move border to the right to make room *)
      end;
    incr current
  done

  (* the split point is just to left of the border *)
  let split = !border - 1 in
  (* move pivot into place at the split point *)
  swap left split;
  (* return the split index *)
  split
```

With the availability of the `partition` function, we can implement a function `sort_region` to sort a region, again picked out by two indices.

```
let rec sort_region (left : int) (right : int) : unit =
  if left >= right then ()
  else
    let split = partition left right in
    (* recursively sort left and right regions *)
    sort_region left (split - 1);
    sort_region (split + 1) right
```

Finally, to sort the entire array, we can sort the region between the leftmost and rightmost indices

```
sort_region 0 ((Array.length arr) - 1)
```

Putting this all together leads to the implementation shown in Figure 16.3. (We've placed the `swap` and `partition` functions within the `sort` function so that they are within the scope of (and can thus access) the `lt` and `arr` arguments of `sort`.)

You'll note that the in-place quicksort is considerably longer than the pure version. In part that is because of the much more detailed work that must be done in partitioning a region, maintaining complex

```
module QuickSort : SORT_IN_PLACE =
  struct
    let sort (lt : 'a -> 'a -> bool) (arr : 'a array) : unit =

      (* swap i j -- Update the `arr` array by swapping the
         elements at indices `i` and `j` *)
      let swap (i : int) (j : int) : unit =
        let temp = arr.(i) in
        arr.(i) <- arr.(j);
        arr.(j) <- temp in

      (* partition left right -- Partition the region of the
         `arr` array between indices `left` and `right`
         *inclusive*, returning the split point, that is, the
         index of the pivot element. Assumes the region
         contains at least two elements. At the end,
         everything to left of the split is less than the
         pivot; everything to the right is greater. *)
      let partition (left : int) (right : int) : int =

        (* region has at least two elements *)
        assert (left < right);

        (* select the pivot element to be the first element in
           the region *)
        let pivot_val = arr.(left) in
        (* all elements to the left of `border` are guaranteed
           to be strictly less than pivot value *)
        let border = ref (left + 1) in
        (* current element being partitioned, starting just
           after pivot *)
        let current = ref (left + 1) in
```

Figure 16.3: Implementation of an in-place quicksort.

```
      (* process each element, moving those less than the
         pivot to before the border *)
      while !current <= right do
        if lt arr.(!current) pivot_val then
          begin
            (* current should be left of pivot *)
            swap !current !border; (* swap into place *)
            incr border (* move border right to make room *)
          end;
        incr current
      done;

      (* the split point is just to left of the border *)
      let split = !border - 1 in
      (* move pivot into place at the split point *)
      swap left split;
      (* return the split index *)
      split in

    (* sort_region left right -- quicksort the subarray of
       the `arr` array between indices `left` and `right`
       *inclusive* *)
    let rec sort_region (left : int) (right : int) : unit =
      if left >= right then ()
      else
        let split = partition left right in
        (* recursively sort left and right regions *)
        sort_region left (split - 1);
        sort_region (split + 1) right
    in

    (* sort the whole `arr` array *)
    sort_region 0 ((Array.length arr) - 1)
end
```

Figure 16.3: (continued) Implementation of an in-place quicksort.

invariants concerning the left, right, current, and border indices and the elements in the various subregions. In part the length is a result of considerably more documentation in the implementation, but that is not a coincidence. The implementation requires this additional documentation to be remotely as understandable as the pure version. (Even still, an understanding of the in-place version is arguably more complex. It's hard to imagine understanding how the partition function works without manually "playing computer" on some examples to verify the procedure.)

The payoff is that the in-place version needs to allocate only a tiny amount of space beyond the storage in the various stack frames for the function applications – just the storage for the `current` and `border` elements. Is the cost in code complexity and opaqueness worth it? That depends on the application. If sorting huge amounts of data is necessary, the reduction in space may be needed.

Problem 171

A completely in-place version of mergesort that uses only a fixed amount of extra space turns out to be quite tricky to implement. However, a version that uses only a single extra array is possible, and still more space-efficient than the pure version described in Section 14.2. Implement a version of mergesort that uses a single extra array as "scratch space" for use while merging. To sort a region, we sort the left and right subregions recursively, then merge the two into the scratch array, and finally copy the merged region back into the main array.

16.5 Supplementary material

- Lab 12: Procedural programming and loops