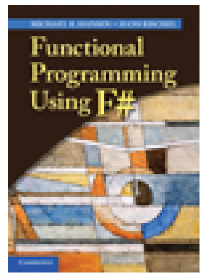


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

Book DOI: http://dx.doi.org/10.1017/CBO9781139093996

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

9 - Efficiency pp. 197-218

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.010

Cambridge University Press

9

Efficiency

The efficiency of a program is measured in terms of its memory requirements and its running time. In this chapter we shall introduce the concepts *stack* and *heap* because a basic understanding of these concepts is necessary in order to understand the memory management of the system, including the *garbage collection*.

Furthermore, we shall study techniques that in many cases can be used to improve the efficiency of a given function, where the idea is to search for a more general function, whose declaration has a certain form called *iterative* or *tail recursive*. Two techniques for deriving tail-recursive functions will be presented: One is based on using *accumulating parameters* and the other is based on the concept of a *continuation*, that represents the rest of the computation. The continuation-based technique is generally applicable. The technique using accumulating parameters applies in certain cases only, but when applicable it usually gives the best results. We give examples showing the usefulness of these programming techniques.

We relate the notion of iterative function to `while` loops and provide examples showing that tail-recursive programs are in fact running faster than the corresponding programs using while loops.

The techniques for deriving tail-recursive functions are useful programming techniques that often can be used to obtain performance gains. The techniques do not replace a conscious choice of good algorithms and data structures. For a systematic study of efficient algorithms, we refer to textbooks on "Algorithms and Data Structures."

9.1 Resource measures

The performance of an algorithm given by a function declaration in F# is expressed by figures for the resources used in the *evaluation* of a function value:

- *Use of computer memory:* The *maximum size* of computer memory needed to represent *expressions* and *bindings* during the evaluation.
- *Computation time:* The *number* of individual *computation steps*.

The important issue is to estimate how these figures depend on the "size‘" of the argument for "large" arguments, for example, number of digits of integer argument, length of list argument, depth (i.e. number of levels) of tree argument, etc. These performance figures are essentially language independent, so implementations of the same algorithm in another programming language will show a similar behaviour.

Efficiency in performance is not the only important issue in programming. Correctness and readability are often more important because the program should be understandable to the readers (including the programmer herself). The choice of function declaration should therefore be based on a trade-off between performance and readability (that is, simplicity), using the simplest declaration for any particular function in a program – unless, there is a risk that it becomes a performance bottleneck for the overall program.

9.2 Memory management

The memory used by an F# program is spilt into a *stack* and a *heap*, where primitive values, such as numbers and truth values are allocated on the stack, while composite values such as lists and trees, closures and (most) objects are allocated on the heap. A basic understanding of the stack and the heap is necessary to understand the memory resources required by a program.

Consider the following declaration at the outermost level:

```
let xs = [5;6;7];;
val xs : int list = [5; 6; 7]

let ys = 3::4::xs;;
val ys : int list = [3; 4; 5; 6; 7]

let zs = xs @ ys;;
val zs : int list = [5; 6; 7; 3; 4; 5; 6; 7]

let n = 27;;
val n : int = 27
```

The stack and the heap corresponding to these declarations are shown in Figure 9.1.

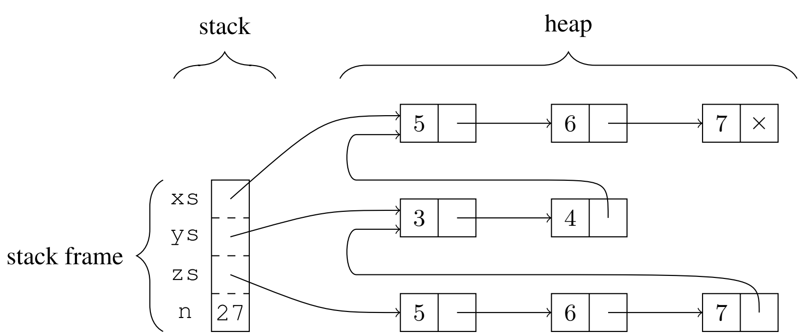


Figure 9.1 Memory: Stack and Heap for top-level declarations

The stack contains an entry for each binding. The entry for the integer `n` contains the integer value 27, while the entries for the lists `xs`, `ys` and `zs` contain links (i.e. memory pointers) pointing at the implementations of these lists. A list $[x_0; \ldots; x_{n-1}]$ is implemented by a linked data structure, where each list element x_i is implemented by a *cons cell* containing the value x_i and a link to the cons cell implementing the next element in the list:

- The entry for `xs` in the stack contains a link to the cons cell for its first element 5 in the heap.
- The entry for `ys` in the stack contains a link to the cons cell for its first element 3. This cons cell contains a link to the cons cell for the next element 4 and that cons cell contains in turn a link to the first cons cell of `xs`.
- The entry for `zs` in the stack contains a link to the first cons cell of a copy of the linked list for `xs` (the first argument of `@` in `xs @ ys`). The last cons cell of that copied linked list contains a link to the start of the linked list for `ys`.

Since a list is a functional (immutable) data structure, we have that:

1. The linked lists for `ys` is not copied when building a linked list for `y::ys`.
2. Fresh cons cells are made for the elements of `xs` when building a linked list for `xs @ ys`, as the last cons cell in the new linked list for `xs` must refer to the first cons cell of the linked list for `ys`. The running time of `@` is, therefore, linear in the length of its first argument. This running time is in agreement with the declaration of append in Section 4.4 and with the linked-list based implementation used by the built-in append function.

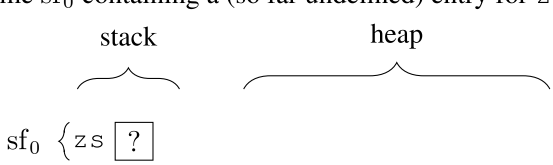
These two properties will be exploited later in this section.

Basic operations on Stack and Heap

The consecutive piece of stack memory corresponding to bindings at the same level is called a *stack frame*. During the evaluation of an expression a new stack frame is added whenever new bindings arise, for example, due to local declarations and expressions or because a function is called. This is illustrated using the following declarations:

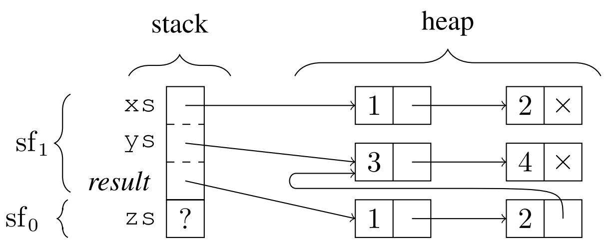
```
let zs = let xs = [1;2]
         let ys = [3;4]
         xs@ys;;
```

The evaluation of the outermost declaration will start with an empty heap and a stack frame sf_0 containing a (so far undefined) entry for `zs`:



Pushing a stack frame

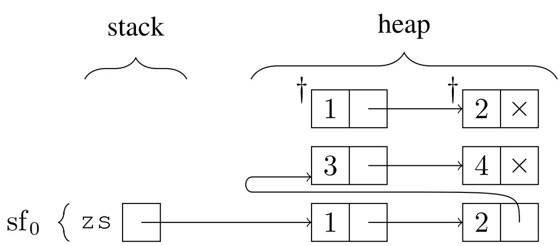
The start of the evaluation of the local declarations will *push* an new stack frame on top of sf_0. This stack frame has entries for the locally declared variables `xs` and `ys` and some extra entries including one for the *result* of the local expression `xs @ ys`:



Notice that a copy of the list `xs` is made in the heap during the evaluation of `xs @ ys`.

Popping a stack frame

When the result of the local expression `xs @ ys` has been computed, the stack frame sf_1 is *popped*, that is, removed from the stack, and the reference to the first cons cell of `xs @ ys` is copied to the stack entry for `zs`:



The resulting heap after the evaluation of the declaration for `zs` contains two cons cells marked with '†'. These cells are obsolete because they cannot be reached from any binding, and they are therefore later removed from the heap by the *garbage collector* that manages the heap behind the scene.

The management of the stack follows the evaluation of declarations and function calls in a simple manner, and the used part of the stack is always a consecutive sequence of the relevant stack frames. We illustrate this by a simple example. Consider the following declarations:

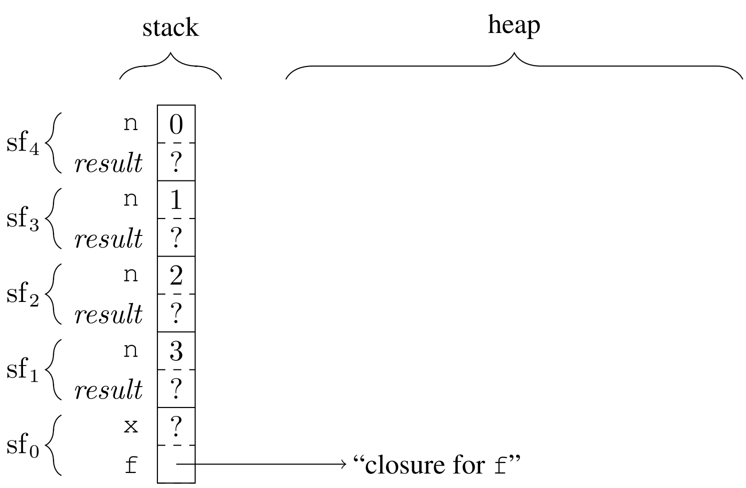
```
let rec f n =
    match n with
    | 0 -> 0
    | n -> f(n-1) + n;;

let x = f 3;;
```

The first part of the evaluation of `f 3` makes repeated bindings of `n` corresponding to the recursive function calls:

$$
\begin{array}{ll}
 & \texttt{f 3} \\
\rightsquigarrow & (\texttt{f n},\ [\texttt{n} \mapsto 3]) \\
\rightsquigarrow & (\texttt{f(n-1)} + \texttt{n},\ [\texttt{n} \mapsto 3]) \\
\rightsquigarrow & \texttt{f 2} + (\texttt{n},\ [\texttt{n} \mapsto 3]) \\
\rightsquigarrow & (\texttt{f n},\ [\texttt{n} \mapsto 2]) + (\texttt{n},\ [\texttt{n} \mapsto 3]) \\
\ldots & \\
\rightsquigarrow & (\texttt{f n},\ [\texttt{n} \mapsto 0]) + (\texttt{n},\ [\texttt{n} \mapsto 1]) + (\texttt{n},\ [\texttt{n} \mapsto 2]) + (\texttt{n},\ [\texttt{n} \mapsto 3])
\end{array}
$$

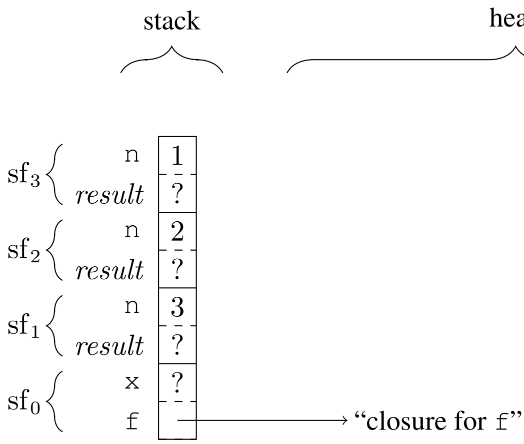
These bindings are implemented by four stack frames $\mathrm{sf}_1, \ldots, \mathrm{sf}_4$ pushed on top of the initial stack frame sf_0 corresponding to `f` and `x`. Each of the stack frames $\mathrm{sf}_1, \ldots, \mathrm{sf}_4$ corresponds to an uncompleted evaluation of a function call:



The next evaluation step marks the completion of the "innermost" functions call `f 0`

$$
\begin{array}{ll}
 & (\texttt{f n},\ [\texttt{n} \mapsto 0]) + (\texttt{n},\ [\texttt{n} \mapsto 1]) + (\texttt{n}\ ,[\texttt{n} \mapsto 2]) + (\texttt{n},\ [\texttt{n} \mapsto 3]) \\
\rightsquigarrow & 0 + (\texttt{n},\ [\texttt{n} \mapsto 1]) + (\texttt{n},\ [\texttt{n} \mapsto 2]) + (\texttt{n},\ [\texttt{n} \mapsto 3])
\end{array}
$$

and the binding $\texttt{n} \mapsto 0$ is hence no longer needed. The implementation releases the memory used to implement this binding by popping the frame sf_4 off the stack:



When the evaluation terminates the stack frames sf_3, sf_2 and sf_1 are all popped and the initial stack frame sf_0 contains a binding of `x` to 6.

The stack management using the push and pop operations is very simple because the stack is maintained as a contiguous sequence of the relevant stack frames. The stack memory will hence never be fragmented.

Garbage and garbage collection

We shall now study garbage collection closer using the declarations:

```
let xs = [1;2];;

let rec g = function
    | 0 -> xs
    | n -> let ys = n::g(n-1)
           List.rev ys;;
val g : int -> int list

g 2;;
val it : int list = [1; 1; 2; 2]
```

Application of this function will produce garbage due to the local declaration of a list and due to the use of `List.rev`. The stack and the heap upon the termination of `g 2` is shown in Figure 9.2. The stack contains just one stack frame corresponding to the top-level declarations.

The heap contains five cons cells marked with '†', that are obsolete because they cannot be reached from any binding, and they are removed from the heap by the garbage collector. It is left for Exercise 9.1 to produced this stack and heap for an evaluation of `g 2`. The amount of garbage produced using `g` grows with the size of the argument, and it is easy to measure how much garbage the system has to collect.

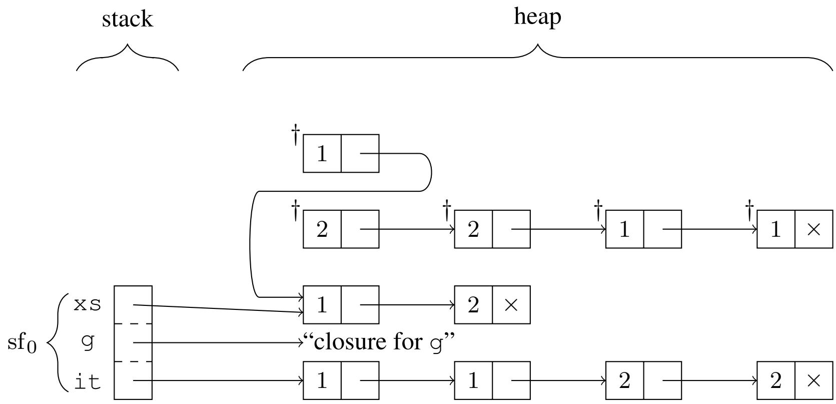


Figure 9.2 Memory: Stack and Heap upon termination of evaluation of `g 2`

Measuring running time and garbage collection

The directive `#time`, that works as a toggle, can be used in the interactive F# environment to extract information about running time and garbage collection of an operation:

```
#time;;

--> Timing now on

g 10000;;
Real: 00:00:01.315, CPU: 00:00:01.326,
GC gen0: 356, gen1: 24, gen2: 0
val it : int list = [9999; 9997; 9995; 9993; 9991; 9989; 9987;...]
```

The measurement includes two times: The *Real time* is the clock time elapsed during the execution of the operation, in this case 1.315 second. The *CPU time* is the total time spent by the operation on all CPUs (or cores) on your computer. If you are not exploiting the parallelism of multiple cores, then these two times should approximately be the same.

The garbage collector manages the heap as partitioned into three groups or *generations*: `gen0`, `gen1` and `gen2`, according to their age. The objects in `gen0` are the youngest while the objects in `gen2` are the oldest. The typical situation is that objects die young, that is, garbage typically occurs among young objects, and the garbage collector is designed for that situation. During the above evaluation of `g 10000`, the garbage collector reclaimed (collected) 356 objects among the youngest ones from group `gen0` and 24 objects from `gen1`.

The limits of the stack and the heap

The stack and heap sizes are resources that we must be aware of. The following examples illustrate maximal stack and heap sizes and shows that the maximal heap size is order of magnitudes larger than the maximal stack size.

Consider first the following function that can generate a list:

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
val bigList : int -> int list

bigList 120000;;
val it : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;...]

bigList 130000;;
Process is terminated due to StackOverflowException.
```

A call `bigList` n will generate n consecutive stack frames each with a binding of `n` and the examples show that 120000 such stack frames are manageable while 130000 are not.

Another declaration of a function that can generate the same lists as the above one is given below. This function can generate lists that are about 100 times longer than those generated above, and when memory problems arise it is because the heap is exhausted:

```
let rec bigListA n xs = if n=0 then xs
                        else bigListA (n-1) (1::xs);;
val bigListA : int -> int list -> int list

let xsVeryBig = bigListA 12000000 [];;
val xsVeryBig : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1;...]

let xsTooBig = bigListA 13000000 [];;
System.OutOfMemoryException:
   Exception of type 'System.OutOfMemoryException' was thrown.
   at FSI_0002.bigListA(Int32 n, FSharpList`1 xs)
   at <StartupCode$FSI_0014>.$FSI_0014.main@()
Stopped due to error
```

In the next sections we study techniques that can be used to minimize the memory usage.

9.3 Two problems

In this section we reconsider the declarations of the factorial function `fact` (Page 6) and the reverse function for lists `naiveRev` (Page 80). We shall see that evaluation of a function value for `fact` uses more computer memory than necessary, and that the evaluation of a function value for `naiveRev` requires more evaluation steps than necessary. More efficient implementations for these functions are given in the next section.

The factorial function

The factorial function has previously been declared by:

```
let rec fact = function
    | 0 -> 1
    | n -> n * fact(n-1);;
val fact : int -> int
```

We have seen that the evaluation of the expression `fact` (N) proceeds through a number of evaluation steps building an expression with a size proportional to the argument N upon which the expression is evaluated:

$$
\begin{array}{ll}
 & \texttt{fact}(N) \\
\rightsquigarrow & (\texttt{n * fact(n-1)}\,,\, [\texttt{n} \mapsto N]) \\
\rightsquigarrow & N * \texttt{fact}(N-1) \\
\rightsquigarrow & N * (\texttt{n * fact(n-1)}\,,\, [\texttt{n} \mapsto N-1]) \\
\rightsquigarrow & N * ((N-1) * \texttt{fact}(N-2)) \\
\vdots & \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * (2 * 1))) \cdots))) \\
\rightsquigarrow & N * ((N-1) * ((N-2) * (\cdots (4 * (3 * 2)) \cdots))) \\
\vdots & \\
\rightsquigarrow & N!
\end{array}
$$

The maximal size of the memory needed during this evaluation is proportional to N, because the F# system must remember (in the heap) all N factors of the expression: $N \star ((N-1) \star ((N-2) \star (\cdots (4 \star (3 \star (2 \star 1))) \cdots)))$ during the evaluation. Furthermore, during the evaluation the stack will grow until it has $N+1$ stack frame corresponding to the nested calls of `fact`.

The reverse function

The naive declaration for the reverse function is as follows:

```
let rec naiveRev = function
    | []    -> []
    | x::xs -> naiveRev xs @ [x];;
val naiveRev : 'a list -> 'a list
```

A part of the evaluation of the expression `naiveRev` $[x_1, x_2, \ldots, x_n]$ is:

$$
\begin{array}{ll}
 & \texttt{naiveRev}\,[x_1, x_2, \ldots, x_n] \\
\rightsquigarrow & \texttt{naiveRev}\,[x_2, \ldots, x_n]\,\texttt{@}\,[x_1] \\
\rightsquigarrow & (\texttt{naiveRev}\,[x_3, \ldots, x_n]\,\texttt{@}\,[x_2])\,\texttt{@}\,[x_1] \\
\vdots & \\
\rightsquigarrow & ((\cdots(([\,]\,\texttt{@}\,[x_n])\,\texttt{@}\,[x_{n-1}])\,\texttt{@} \cdots \texttt{@}\,[x_2])\,\texttt{@}\,[x_1])
\end{array}
$$

There are $n + 1$ evaluation steps above and heap space of size proportional to n is required by the F# system to represent the last expression. These figures are to be expected for reversing a list of size n.

However, the further evaluation

$$((\cdots(([\,]@[x_n])@[x_{n-1}])@\cdots@[x_2])@[x_1]) \rightsquigarrow [x_n, x_{n-1}, \ldots, x_2, x_1]$$

requires a number of evaluation steps that is proportional to n^2.

To see this, observe first that $m+1$ evaluation steps are needed to evaluate the expression $[y_1, \ldots, y_m]\,@\,zs$ as $y_1 :: (y_2 :: \ldots :: (y_m :: zs)\ldots)$.

Thus,

$$\begin{array}{lr}
[\,]@[x_n] \rightsquigarrow [x_n] & \text{requires 1 step} \\
[x_n]@[x_{n-1}] \rightsquigarrow [x_n, x_{n-1}] & \text{requires 2 steps} \\
\vdots & \\
[x_n, x_{n-1}, \ldots, x_2]@[x_1] \rightsquigarrow [x_n, x_{n-1}, \ldots, x_2, x_1] & \text{requires } n \text{ steps}
\end{array}$$

Hence, the evaluation of $((\cdots(([\,]@[x_n])@[x_{n-1}])@\cdots@[x_2])@[x_1])$ requires

$$1 + 2 + \cdots n = \frac{n(n+1)}{2}$$

steps, which is proportional to n^2.

9.4 Solutions using accumulating parameters

In this section we will show how to obtain much improved implementations of the above functions by considering more general functions, where the argument has been extended by an extra component ("m" and "*ys*"):

$$\begin{array}{lcl}
\texttt{factA}\,(n, m) & = & n! \cdot m, \text{ for } n \geq 0 \\
\texttt{revA}\,([x_1, \ldots, x_n], ys) & = & [x_n, \ldots, x_1]\,@\,ys
\end{array}$$

Note, that $n! = \texttt{factA}\,(n, 1)$ and $\texttt{rev}\,[x_1, \ldots, x_n] = \texttt{revA}\,([x_1, \ldots, x_n], [\,])$. So good implementations for the above functions will provide good implementations for the factorial and the reverse functions also.

The factorial function

The function `factA` is declared by:

```
let rec factA = function
    | (0,m) -> m
    | (n,m) -> factA(n-1,n*m);;
val factA : int * int -> int
```

Consider the following evaluation:

$$
\begin{array}{ll}
 & \texttt{factA(5,1)} \\
\rightsquigarrow & (\texttt{factA(n,m)},\ [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow & (\texttt{factA(n-1,n*m)},\ [\texttt{n} \mapsto 5, \texttt{m} \mapsto 1]) \\
\rightsquigarrow & \texttt{factA(4,5)} \\
\rightsquigarrow & (\texttt{factA(n,m)},\ [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow & (\texttt{factA(n-1,n*m)},\ [\texttt{n} \mapsto 4, \texttt{m} \mapsto 5]) \\
\rightsquigarrow & \texttt{factA(3,20)} \\
\rightsquigarrow & \ldots \\
\rightsquigarrow & \texttt{factA(0,120)} \\
\rightsquigarrow & (\texttt{m},\ [\texttt{m} \mapsto 120]) \\
\rightsquigarrow & 120
\end{array}
$$

This evaluation of `factA(5,1)` has the properties we are looking for:

1. It does not build large expressions.
2. The number of steps needed to evaluate $\texttt{factA}(n,m)$ is proportional to n.

The argument pattern `m` in the above declaration is called an *accumulating parameter*, since the result is gradually built in this parameter during the evaluation.

The main part of the above evaluation of `factA(5,1)` is the gradual evaluation of arguments in the recursive calls of the function:

$$(5,1),\ (4,5),\ (3,20),\ (2,60),\ (1,120),\ (0,120)$$

Each of these values is obtained from the previous one by applying the function:

```
fun (n,m) -> (n-1, n*m)
```

so the evaluation of the arguments can be viewed as repeated (or iterated) applications of this function.

The use of `factA` gives a clear improvement to the use of `fact`. Consider the following example measuring the time of 1000000 computations of 16! using these two function:

```
let xs16 = List.init 1000000 (fun i -> 16);;
val xs16 : int list = [16; 16; 16; 16; 16; 16; 16; 16; ...]

#time;;

for i in xs16 do let _ = fact i in ();;
Real: 00:00:00.051, CPU: 00:00:00.046,
GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()

for i in xs16 do let _ = factA(i,1) in ();;
Real: 00:00:00.024, CPU: 00:00:00.031,
GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()
```

The performance gain of using `factA` is actually much better than the factor 2 indicated by the above examples becomes the run time of the `for` construct alone is about 12 ms:

```
for i in xs16 do let _ = () in ();;
Real: 00:00:00.012, CPU: 00:00:00.015,
GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()
```

The reverse function

The function `revA` is declared by:

```
let rec revA = function
    | ([], ys)     -> ys
    | (x::xs, ys) -> revA(xs, x::ys);;
```

Consider the following evaluation (where the bindings are omitted):

```
   revA([1,2,3],[])
⇝ revA([2,3],1::[])
⇝ revA([2,3],[1])
⇝ revA([3],2::[1])
⇝ revA([3],[2,1])
⇝ revA([],3::[2,1])
⇝ revA([],[3,2,1])
⇝ [3,2,1]
```

This evaluation of `revA([1,2,3],[])` again has the properties we are looking for:

1. It does not build large expressions.
2. The number of steps needed to evaluate `revA`(xs, ys) is proportional to the length of *xs*.

It makes a big difference for lists with large length n whether the number of evaluation steps is proportional to n or to n^2.

The argument pattern `ys` in the above declaration is the accumulating parameter in this example since the result list is gradually built in this parameter during the evaluation.

Note, that each argument in the recursive calls of `revA` is obtained from the argument in the previous call by applying the function:

```
fun (x::xs, ys) -> (xs, x::ys)
```

The use of `revA` gives a dramatically improvement to the use of `naiveRev`. Consider the following example measuring the time used for reversing the list of elements from 1 to 20000:

```
let xs20000 = [1 .. 20000];;

naiveRev xs20000;;
Real: 00:00:07.624, CPU: 00:00:07.597,
GC gen0: 825, gen1: 253, gen2: 0
val it : int list = [20000; 19999; 19998; 19997; 19996;...]
```

```
revA(xs20000,[]);;
Real: 00:00:00.001, CPU: 00:00:00.000,
GC gen0: 0, gen1: 0, gen2: 0
val it : int list = [20000; 19999; 19998; 19997; 19996; ...]
```

The naive version takes 7.624 seconds while the iterative version takes just 1 ms. One way to consider the transition from the naive version to the iterative version is that the use of append (`@`) has been reduced to a use of cons (`::`) and this has a dramatic effect of the garbage collection. No object is reclaimed by the garbage collector when `revA` is used, whereas 825+253 obsolete objects were reclaimed using the naive version and this extra memory management takes time.

Returning to the list-generating functions on Page 204, the function `bigListA` is a more general function than `bigList`, where the argument `xs` is the accumulating parameter.

9.5 Iterative function declarations

The above declarations for `factA`, `revA` and `bigListA` have a certain form that we will study in this section.

A declaration of a function $g : \tau$ `->` τ' is said to be an *iteration of a function* $f : \tau$ `->` τ if it is an instance of the *schema*:

let rec g z = if p z then g(f z) else h z;;

for suitable predicate $p : \tau$ `->` `bool` and function $h : \tau$ `->` τ'.

A function declaration following the above schema is called an *iterative* declaration. It is *tail-recursive* in the sense that every recursive call of the function is a *tail call*, that is, the last operation that is evaluated in the body of the declaration. For convenience we only study tail-recursive declarations of the above form in this subsection.

The function `factA`

The function `factA` is an iterative function because it can be declared as:

```
let rec factA(n,m) = if n<>0 then factA(n-1,n*m) else m;;
```

which is an instance of the above schema with:

```
let f(n,m) = (n-1, n*m)

let p(n,m) = n<>0

let h(n,m) = m;;
```

The function `revA`

The function `revA` is also an iterative function:

```
let rec revA(xs,ys) =
    if (not (List.isEmpty xs))
    then revA(List.tail xs, (List.head xs)::ys)
    else ys;;
```

which is an instance of the above schema with:

```
let f(xs,ys) = (List.tail xs, (List.head xs)::ys)

let p(xs,ys) = not (List.isEmpty xs)

let h(xs,ys) = ys
```

When a declaration of a function in an obvious way can be transformed into the above form, we will call it an iterative function without further argument.

The `fold` *function on lists*

The `fold` function on lists as declared in Section 5.1:

```
let rec fold f e = function
    | x::xs -> fold f (f e x) xs
    | []    -> e  ;;
```

is an iterative function. The declaration can be written as:

```
let rec fold f e xs =
    if not (List.isEmpty xs)
    then fold f (f e (List.head xs)) (List.tail xs)
    else e;;
```

which is an instance of the above schema. The above function `revA` is actually an application of this iterative function:

```
let revA(xs,ys) = fold (fun e x -> x::e) ys xs;;
```

Evaluation of iterative functions

The evaluation for an arbitrary iterative function:

let rec g z = if p z then g(f z) else h z;;

proceeds in the same manner as the evaluations of `factA` and `revA`:

We define the n'th *iteration* $f^n x$, for $n \geq 0$, of a function $f : \tau$ -> τ as follows:

$$\begin{aligned} f^0 x &= x \\ f^{k+1} x &= f(f^k x), \text{ for } k \geq 0 \end{aligned}$$

Thus,

$$f^0 x = x, \quad f^1 x = f x, \quad \ldots, \quad f^n x = \underbrace{f(f(\cdots f}_{n} x \cdots))$$

Suppose that

$$\begin{array}{lll} p(f^i x) & \rightsquigarrow & \texttt{true} \quad \text{for all } i : 0 \leq i < n, \text{ and} \\ p(f^n x) & \rightsquigarrow & \texttt{false} \end{array}$$

Then, the evaluation of the expression $g\ x$ proceeds as follows:

$$\begin{array}{ll} & g\ x \\ \rightsquigarrow & (\texttt{if}\ p\ \texttt{z then}\ g(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}, [\texttt{z} \mapsto x]) \\ \rightsquigarrow & (g(f\ \texttt{z}), [\texttt{z} \mapsto x]) \\ \rightsquigarrow & g(f^1 x) \\ \rightsquigarrow & (\texttt{if}\ p\ \texttt{z then}\ g(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}, [\texttt{z} \mapsto f^1 x]) \\ \rightsquigarrow & (g(f\ \texttt{z}), [\texttt{z} \mapsto f^1 x]) \\ \rightsquigarrow & g(f^2 x) \\ \rightsquigarrow & \ldots \\ \rightsquigarrow & (\texttt{if}\ p\ \texttt{z then}\ g(f\ \texttt{z})\ \texttt{else}\ h\ \texttt{z}, [\texttt{z} \mapsto f^n x]) \\ \rightsquigarrow & (h\ \texttt{z}, [\texttt{z} \mapsto f^n x]) \\ \rightsquigarrow & h(f^n x) \end{array}$$

This evaluation has three desirable properties:

1. It does not build large expressions, as the argument $f\ \texttt{z}$ of $g(f\ \texttt{z})$ is evaluated at each step due to the eager evaluation strategy of F#,
2. there are n recursive calls of g, and
3. there is only one environment used at each stage of this evaluation.

The first property implies that heap allocation of long expressions with pending operations can be avoided, the second property implies a linear unfolding of the recursive function g, and the last property implies that just one stack frame is needed during an evaluation of $g\ x$ (ignoring stack frames needed due to calls of other functions).

Since `bigListA` is a tail-recursive function, the stack will not grow during the evaluation of `bigListA` $n\,xs$ and the heap is hence the limiting memory resource when using this function as we learned in connection with the examples on Page 204.

Iterations as loops

We observed in Section 8.7 that every while loop can be expressed as an iteration. It is also the case that every iterative function g:

```
let rec g z = if p z then g(f z) else h z;;
```

can be expressed as a while loop:

```
let rec g z =
    let zi = ref z
    while p !zi do zi := f !zi
    h(!zi);;
```

Using this translation scheme for the iterative version `factA` of the factorial function, we arrive at the declaration:

```
let factW n =
    let ni = ref n
    let r  = ref 1
    while !ni>0 do
         r := !r * !ni ; ni := !ni-1
    !r;;
```

where it is taken into account that the argument `z` in the translation scheme in this case is a pair `(n,r)`.

There is no efficiency gain in transforming an iteration to a while-loop. Consider for example 1000000 computations of 16! using `factA(16,1)` and `factW 16`:

```
#time;;

for i in 1 .. 1000000 do let _ = factA(16,1) in ();;
Real: 00:00:00.024, CPU: 00:00:00.031,
GC gen0: 0, gen1: 0, gen2: 0
val it : unit = ()

for i in 1 .. 1000000 do let _ = factW 16 in ();;
Real: 00:00:00.048, CPU: 00:00:00.046,
GC gen0: 9, gen1: 0, gen2: 0
val it : unit = ()
```

which shows that the tail-recursive function actually is faster than the imperative while-loop based version.

9.6 Tail recursion obtained using continuations

A tail-recursive version of a function can in some cases be obtained by introducing an accumulating parameter as we have seen in the above examples, but this technique is insufficient in the general case. However, there is a general technique that can transform an arbitrary declaration of a recursive function $f : \tau_1$`->`τ_2 into a tail-recursive one. The technique adds an extra argument c that is a function. At present we assume that each branch in the recursive declaration of f contains at most one recursive call of f. The tail recursive version f`C` of f is then of type τ_1`->(`τ_2`->`τ_2`)->`τ_2 with parameters v and c of types $v : \tau_1$ and $c : \tau_2$`->`τ_2.

The evaluation of a function value $f(v)$ comprises recursive calls of f with arguments $v_0, v_1, \ldots, v_n$ where $v_0 = v$ and where v_n corresponds to a base case in the declaration of f. The corresponding evaluation of f`C`:

$$f\texttt{C}\ v_0\ c_0 \leadsto f\texttt{C}\ v_1\ c_1 \leadsto \ldots \leadsto f\texttt{C}\ v_n\ c_n \leadsto c_n(f\ v_n) \leadsto \ldots$$

contains functions $c_0, c_1, \ldots, c_n$ with the crucial property:

$$c_k(f\ v_k) = f(v) \quad \text{for} \quad k = 0, 1, \ldots, n$$

This property expresses that the function c_k contains the *rest* of the computation once you have computed $f(v_k)$. It is therefore called a *continuation*. The evaluation of fC starts with $c_0 = $ id where id is the pre-defined identity function satisfying id $a = a$ for any a. The effects of the recursive calls of f are gradually accumulated in the continuations c_k during the evaluation of fC v id, and the evaluation ends by applying the continuation c_n to the value $f(v_n)$ in a base case.

The notion of a continuation has a much wider scope than achieving tail-recursive functions (the focus in this chapter) and we refer to [12] for an in-depth study of this concept.

Consider, for example, the simple declaration of bigList from Section 9.2:

```
let rec bigList n = if n=0 then [] else 1::bigList(n-1);;
val bigList : int -> int list
```

that was used to illustrate the stack limit problems due to the fact that it is not a tail-recursive function. The continuation-based version bigListC n c has a extra argument

```
c: int list -> int list
```

that is a continuation. The declaration of bigListC is:

```
let rec bigListC n c =
    if n=0 then c []
    else bigListC (n-1) (fun res -> c(1::res));;
val bigListC : int -> (int list -> 'a) -> 'a
```

The base case of bigListC is obtained from the the base case of bigList by feeding that result into the continuation c. For the recursive case, let res denote the value of the recursive call of bigList(n-1). The rest of the computation of bigList n is then 1::res. Hence, the continuation of bigListC(n-1) is

```
fun res -> c(1::res)
```

because c is the continuation of bigListC n.

The function is called using the pre-defined identity function id as continuation:

```
bigListC 3 id;;
val it : int list = [1; 1; 1]
```

The important observations are:

- bigListC is a tail-recursive function, and
- the calls of c are tail calls in the base case of bigListC as well as in the continuation: fun res -> c(1::res).

The stack will hence neither grow due to the evaluation of recursive calls of bigListC nor due to calls of the continuations that have been built in the heap.

Consider the examples:

```
bigListA 12000000 [];;
Real: 00:00:01.142, CPU: 00:00:01.138,
GC gen0: 34, gen1: 22, gen2: 0
val it : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;...]
```

```
bigListC 12000000 id;;
Real: 00:00:05.742, CPU: 00:00:05.538,
GC gen0: 60, gen1: 48, gen2: 3
val it : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;...]

bigListC 16000000 id;;
Real: 00:00:08.586, CPU: 00:00:08.314,
GC gen0: 80, gen1: 60, gen2: 3
val it : int list = [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1;...]

bigListC 17000000 id;;
System.OutOfMemoryException: Exception of type
'System.OutOfMemoryException' was thrown. Stopped due to error
```

These examples show:

1. The version using an accumulating parameter is much faster (about five times) than that using continuations.
2. The version using continuations can handle about 30% longer lists.

The run-time disadvantage of a continuation-based declaration is even more clear with an iterative function like `factA` where no data structure is required to be built in the heap, when compared to `bigListC`. See Exercise 9.6.

More general recursions

We shall now study the use of continuations in a more general setting with several recursive calls of the function. This situation occurs, for example, in connection with binary trees.

Consider a function `count`, that counts the number of nodes in binary tree on the basis of the type `BinTree<'a>` defined in Section 6.4:

```
type BinTree<'a> = | Leaf
                   | Node of BinTree<'a> * 'a * BinTree<'a>;;

let rec count = function
    | Leaf          -> 0
    | Node(tl,n,tr) -> count tl + count tr + 1;;
```

A counting function: `countA: int -> BinTree<'a> -> int` using an accumulating parameter will not be tail-recursive due to the expression containing recursive calls on the left as well as the right sub-trees of a node (try, for example, Exercise 9.8). A tail-recursive version can, however, be developed for a continuation-based version:

```
countC: BinTree<'a> -> (int -> 'b) -> 'b
```

The base case `countC Leaf c` returns `c 0`. The continuation of `countC tl` in the case: `countC (Node(n,tl,tr)) c` is the function that takes the result `vl` for the left subtree and calls `countC tr`. The continuation of `countC tr` must take the result `vr` for the right subtree and feed `vl+vr+1` into the continuation `c`:

```
let rec countC t c =
  match t with
  | Leaf          -> c 0
  | Node(tl,n,tr) ->
      countC tl (fun vl -> countC tr (fun vr -> c(vl+vr+1)));;
val countC : BinTree<'a> -> (int -> 'b) -> 'b

countC (Node(Node(Leaf,1,Leaf),2,Node(Leaf,3,Leaf))) id;;
val it : int = 3
```

Note that both calls of `countC` are tail calls, and so are the calls of the continuation `c`, and the stack will therefore not grow due to the evaluation of `countC` and the associated continuations.

The comparison of `count` and `countC` shows similar figures as the comparison of `bigList` and `bigListC`: The continuation-based version can handle much larger trees since the stack space will not be exhausted (try Exercise 9.11); but it is about 4 times slower than `count` when counting a balanced tree with 20000000 nodes:

```
let rec genTree xs =
    match xs with
    | [| |]   -> Leaf
    | [| x |] -> Node(Leaf,x,Leaf)
    | _    -> let m = xs.Length / 2
              let xsl = xs.[0..m-1]
              let xm  = xs.[m]
              let xsr = xs.[m+1 ..]
              Node(genTree xsl, xm, genTree xsr);;
val genTree : 'a [] -> BinTree<'a>

let t n = genTree [| 1..n |];;

let t20000000 = t 20000000;;

count t20000000;;
Real: 00:00:00.453, CPU: 00:00:00.889,
                    GC gen0: 0, gen1: 0, gen2: 0
val it : int = 20000000

countC t20000000  id;;
Real: 00:00:01.733, CPU: 00:00:01.716,
GC gen0: 305, gen1: 1, gen2: 0
val it : int = 20000000
```

It is possible to replace one of the continuations in the recursive case of the declaration of `countC` by a simple accumulator and arrive at a tail-recursive function with the type

```
countAC : BinTree<'a> -> int -> (int -> 'b) -> 'b
```

such that `count` t $=$ `countAC` t `0 id`. The declaration and analysis of this function is left for Exercise 9.9.

Summary

We have introduced the concepts stack and heap that are needed in order to get a basic understanding of the memory management in the system.

Furthermore, we have introduced the concept of tail-recursive functions and two techniques for deriving a tail-recursive version of a given function, where one is based on accumulating parameters and the other on the notion of a continuation. The stack will not grow during the evaluation of tail-recursive functions (ignoring the calls of other recursive functions), and using these techniques will in many typical cases give good performance gains.

A transformation from tail-recursive functions to loops was shown, together with experiments showing that the tail-recursive functions run faster than the corresponding imperative while-loop based versions.

Exercises

9.1 Consider the function `g` declared on Page 202 and the stack and heap after the evaluation of `g 2` shown in Figure 9.2. Reproduce this resulting stack and heap by a systematic application of push and pop operations on the stack, and heap allocations that follow the step by step evaluation of `g 2`.

9.2 Show that the `gcd` function on Page 16 is iterative.

9.3 Declare an iterative solution to exercise 1.6.

9.4 Give iterative declarations of the list function `List.length`.

9.5 Express the function `List.fold` in terms of an iterative function `itfold` iterating a function of type `'a list * 'b -> 'a list * 'b`.

9.6 Declare a continuation-based version of the factorial function and compare the run time with the results in Section 9.4.

9.7 Develop the following three versions of functions computing Fibonacci numbers F_n (see Exercise 1.5):

1. A version `fibA: int -> int -> int -> int` with two accumulating parameters n_1 and n_2, where `fibA` $n\,n_1\,n_2 = F_n$, when $n_1 = F_{n-1}$ and $n_2 = F_{n-2}$. Hint: consider suitable definitions of F_{-1} and F_{-2}.
2. A continuation-based version `fibC: int -> (int -> int) -> int` that is based on the definition of F_n given in Exercise 1.5.

Compare these two functions using the directive `#time`, and compare this with the while-loop based solution of Exercise 8.6.

9.8 Develop a version of the counting function for binary trees

```
countA: int -> BinTree<'a> -> int
```

that makes use of an accumulating parameter. Observe that this function is not tail recursive.

9.9 Declare a tail-recursive functions with the type

```
countAC : BinTree<'a> -> int -> (int -> 'b) -> 'b
```

such that `count` t = `countAC` t `0 id`. The intuition with `countAC` $t\ a\ c$ is that a is the number of nodes being counted so far and c is the continuation.

9.10 Consider the following list-generating function:

```
let rec bigListK n k =
    if n=0 then k []
    else bigListK (n-1) (fun res -> 1::k(res));;
```

The call `bigListK 130000 id` causes a stack overflow. Analyze this problem.

9.11 Declare tail-recursive functions `leftTree` and `rightTree`. By use of `leftTree` it should be possible to generate a big unbalanced tree to the left containing $n+1$ values in the nodes so that n is the value in the root, $n-1$ is the value in the root of the left subtree, and so on. All subtree to the right are leaves. Similarly, using `rightTree` it should be possible to generate a big unbalanced tree to the right.

1. Use these functions to show the stack limit when using `count` and `countA` from Exercise 9.8.
2. Use these functions to test the performance of `countC` and `countAC` from Exercise 9.9.

9.12 Develop a continuation-based version of the function `preOrder` from Section 6.4, and compare the performance of the two functions.

9.13 Compare the run times of the two versions of the function `tryFind` that are declared on Page 109 and on Page 191.

9.14 Comparison of the efficiency of iteration functions for list and sets.
In this exercise you should declare functions
`iter`*CollM*: `('a -> unit) -> ` *Coll*`<'a> -> unit`
so that `iter`*CollM f col* performs $f\,v_0;\ f\,v_1;\ \ldots;\ f\,v_n$ when *col* has $v_0, v_1, \ldots, v_n$ as the elements, and M is the method of traversal that can be based on a tail-recursive function or using an enumerator.

1. Declare a tail-recursive function to iterate a function over the elements of a list.
2. Declare a enumerator-based version. See Page 192.
3. Declare a tail-recursive version that iterate over the elements of a set on the basis of the recursion scheme that repeatedly removes the minimal elements from the set. (See e.g. the declaration of `tryFind` on Page 109.)
4. Compare the run times of the above iteration functions and the library functions `List.iter` and `Set.iter`. Use, for example, sets and lists containing the integers from 0 to 10000000 and the function `ignore`.