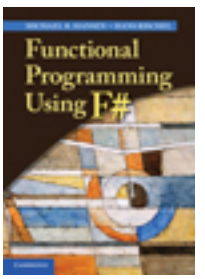


Functional Programming Using F#

Michael R. Hansen, Hans Rischel

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

Online ISBN: 9781139093996

Hardback ISBN: 9781107019027

Paperback ISBN: 9781107684065

Chapter

8 - Imperative features pp. 175-196

Chapter DOI: http://dx.doi.org/10.1017/CBO9781139093996.009

Cambridge University Press

8

Imperative features

Imperative features are language constructs to express commands using and modifying the state of mutable objects. The working of these features are explained using a new component, the *store*, beside the environment. A store consists of a set of locations containing values, together with operators to access and change the store. We explain why the restriction on polymorphically typed expressions is needed because of the imperative features. The F# concept of a mutable record field gives the means of assigning values to object members. The `while` loop is introduced and we study its relationship with iterative functions. The F# and .NET libraries offer imperative collections: arrays, imperative sets and imperative maps.

8.1 Locations

A (mutable) *location* is a part of the computer memory where the F# program may store different values at different points in time. A location is obtained by executing a `let mutable` declaration, for example:

```
let mutable x = 1;;
val mutable x : int = 1
```

The keyword `mutable` requests F# to create a *location*, and the answer tells that `x` is bound to a location of *type* `int`, currently containing the value 1. The situation obtained is illustrated as follows:

$\mathtt{x} \mapsto loc_1 \qquad loc_1 \colon 1$

Locations may be of any type:

```
let mutable y = (3,(5,8));;
val mutable y : int * (int * int) = (3, (5, 8))

let mutable z = [1;2;3];;
val mutable z : int list = [1; 2; 3]

let mutable w = Some (3,(5,8));;
val mutable w : (int * (int * int)) option
                 = Some (3, (5, 8))

let mutable f = sin ;;
val mutable f : (float -> float)
```

After these declarations we have the following situation:

Environment	Store
x $\mapsto$ loc_1	loc_1: 1
y $\mapsto$ loc_2	loc_2: (3,(5,8))
z $\mapsto$ loc_3	loc_3: [1;2;3]
w $\mapsto$ loc_4	loc_4: Some (3,(5,8))
f $\mapsto$ loc_5	loc_5: "the sine function"

A set of locations with contents is called a *store*.

The F# concept of location should be interpreted in an "abstract" sense: An F# location of basic type (like `int`) corresponds to a physical memory location in the computer containing the stored value. An F# location of, for example, a `list` type corresponds on the other hand to a physical memory location containing a link to the list that is stored elsewhere in memory. This difference in management of physical memory is, however, not visible from the F# program.

We shall study the memory management of the system in Chapter 9.

8.2 Operators on locations

There are two operations on a location:

Operator	Symbol	Usage	Legend
Assign	`<-`	exp_1 `<-` exp_2	Assign value to location
ContentsOf		*exp*	Contents of location

An assignment expression exp_1 `<-` exp_2 is evaluated as follows:

1. Evaluate exp_1 to get a location *loc*.
2. Evaluate exp_2 to get a value v.
3. Store the value v in the location *loc*.
4. Deliver the value `()` of type `unit` as the result.

We may, for example, assign the value 7 to `x`:

```
x <- 7;;
val it : unit = ()
```

The evaluation proceeds as follows:

	`x <- 7`	x $\mapsto$ loc_1	loc_1: 1
$\rightsquigarrow$	loc_1 `<-` `7`	x $\mapsto$ loc_1	loc_1: 1
$\rightsquigarrow$	`()`	x $\mapsto$ loc_1	loc_1: 7

This evaluation has a *side effect*: the content of the location loc_1 is changed. It is called a side-effect because it is not visible in the result which is just the uninteresting value `()` of type `unit`. Note that the binding of `x` is *unaffected* by the assignment: `x` remains bound to the same location.

We may do more complex assignments:

```
f <- cos;;
val it : unit = ()

y <- (-2, (3,1));;
val it : unit = ()
```

but the value assigned to a location must have the same type as the location, so the following attempt fails:

```
x <- (2,3);;
  x <- (2,3);;
  ------^^^
...: error FS0001: This expression was expected to
have type
     int
but here has type
    'a * 'b
```

A contentsOf expression *exp* is evaluated as follows:

1. Evaluate exp to get a location *loc*.
2. Deliver the contents v of the location *loc* as the result.

The contentsOf operator has no visible operator symbol and the operator is automatically inserted by the F# compiler according to the following *coercion rule*:

> A contentsOf operator is automatically inserted by the F# compiler whenever a location occurs in a context where a value is required. The right-hand side of a `let`-declaration is such a context.

Assume that we have the binding of `x` and corresponding location loc_1 as above:

$\texttt{x} \mapsto loc_1 \qquad loc_1 : 7$

The evaluation of the expression `x <- x + 1` will then proceed as follows:

	`x <- x + 1`	$\texttt{x} \mapsto loc_1 \quad loc_1 : 7$
$\rightsquigarrow$	loc_1 `<-` loc_1 `+ 1`	$\texttt{x} \mapsto loc_1 \quad loc_1 : 7$
$\rightsquigarrow$	loc_1 `<-` contentsOf(loc_1) `+ 1`	$\texttt{x} \mapsto loc_1 \quad loc_1 : 7$
$\rightsquigarrow$	loc_1 `<- 7 + 1`	$\texttt{x} \mapsto loc_1 \quad loc_1 : 7$
$\rightsquigarrow$	loc_1 `<- 8`	$\texttt{x} \mapsto loc_1 \quad loc_1 : 7$
$\rightsquigarrow$	`()`	$\texttt{x} \mapsto loc_1 \quad loc_1 : 8$

The contentsOf operator is inserted by the coercion rule because the plus operator requires a value as operand.

The coercion rule also apply when making a declaration:

```
let t = x;;
val t : int = 8
```

where the right-hand side `x` would otherwise evaluate to a location. The identifier `t` is hence bound to the value 8:

$\texttt{x} \mapsto loc_1 \qquad loc_1 : 8$
$\texttt{t} \mapsto 8$

An assignment to `x` will have no effect on `t` as there is no connection between the value stored in the location loc_1 and the value bound to `t`:

```
x <- 17;;
val it : unit = ()

t ;;
val it : int = 8
```

The coercion rule also apply if we enter the identifier `x` as an expression to be evaluated by F# because this is interpreted as a declaration `let it = x` of the identifier `it`:

```
x;;
val it : int = 17
```

Hence, a location is *not* a value but an expression may evaluate to a location when, for example, used as the left-hand side of an assignment.

Locations cannot occur as components in tuples or tagged values, as elements in lists or as values of functions. The situation for records is different as described in Section 8.5.

A remark on the `mutable` *declaration*

The "mutable" keyword in the declaration `let mutable x = 1` describes a property of the entity to be bound to the identifier `x` and *not* a property of `x`. A syntax like

```
let x = mutable 1   // NOT legal F#
```

might hence have been more appropriate as the declaration creates a *mutable location* to be bound to the identifier `x`. The actual F# syntax has the advantage that the coercion rule automatically applies when the right-hand side evaluates to a location like in the above example `let t = x` where `x` is bound to a *location* but `t` becomes bound to a *value*.

8.3 Default values

A `mutable` declaration requires an *initial value* to be stored in the location. This value is obtained by evaluating the expression on the right-hand side of the declaration. It is sometimes awkward or even impossible for the programmer to make a proper initial value at the time of declaration. One may then use the *default value* of the type in question:

```
Unchecked.defaultof<type>
```

on the right-hand side of the declaration. Such a value is available for any type. It may serve as a placeholder in the location until replaced by a proper value. Default values should be used only for this purpose.

8.4 Sequential composition

The semicolon symbol "`;`" denotes the *sequential composition* operator (while the double semicolon "`;;`" is a terminator symbol). This operator combines two *expressions* exp_1 and exp_2 to form a new *expression*:

exp_1 `;` exp_2

The expression exp_1 `;` exp_2 is evaluated as follows:

1. Evaluate exp_1 and discard the result.
2. Evaluate exp_2 and supply the result as the result of evaluating exp_1 `;` exp_2.

Hence, if exp_2 has type τ then exp_1 `;` exp_2 has type τ as well.

The F# compiler issues a warning if exp_1 is of type different from `unit` as the result of the evaluation might be of some use. This warning is avoided by using the `ignore` function:

`ignore(`exp_1`)` `;` exp_2 or exp_1 `|> ignore ;` exp_2

where `ignore` a `= ()` for any value a.

We may combine two assignments using sequential composition:

```
let mutable x = 5;;
let mutable y = 7;;
x <- y + 1 ; y <- x + 2;;
(x,y);;
val it : int * int = (8 ,10)
```

Note that the second assignment uses the new value stored in the location denoted by `x`.

The operator "`;`" may be omitted if the expressions are written on separate lines, that is,

exp_1
exp_2

means `(`exp_1`)` `;` exp_2.

8.5 Mutable record fields

A mutable record field is obtained by prefixing the label in the *record type declaration* by the keyword `mutable`, for example:

```
type intRec  = { mutable count : int };;
```

Executing a declaration of a value of this type

```
let r1 = { count = 0 };;
val r1 : intRec = {count = 0;}
```

creates the following entities:

1. A value (record) of type `intRec`,
2. a location containing the value 0 (due to keyword `mutable` in the record type),
3. a local binding inside the record of the record label `count` to the location, and
4. a binding of the identifier `r1` to the record.

So the following is added to environment and store:

Environment	Store
r1 $\mapsto$ { count $\mapsto$ loc_2 }	loc_2: 0

One may assign a value to the `count` field in `r1`:

```
r1.count <- 5;;
val it : unit = ()
```

This assignment changes the contents of the associated location:

Environment	Store
r1 $\mapsto$ { count $\mapsto$ loc_2 }	loc_2: 5

One may declare a function incrementing the counter value of an `intRec` record and delivering the new counter value as the result:

```
let incr (x: intRec) =
    x.count <- x.count + 1
    x.count;;
val incr : intRec -> int

incr r1;;
val it : int = 6

incr r1;;
val it : int = 7
```

We may even declare a function returning a closure with an internal counter:

```
let makeCounter() =
    let counter = { count = 0 }
    fun () -> incr counter;;
val makeCounter : unit -> (unit -> int)
```

```
let clock = makeCounter();;
val clock : (unit -> int)

clock();;
val it : int = 1

clock();;
val it : int = 2
```

Equality of records with mutable fields is defined as for records without such fields. Consider the declarations:

```
let x = { count = 0 };;
val x : intRec = {count = 0;}
let y = { count = 0 };;
val x : intRec = {count = 0;}
let z = y;;
val z : intRec = {count = 0;}
```

The values bound to `x`, `y` and `z` are considered equal:

```
x = y;;
val it : bool = true
y = z;;
val it : bool = true
```

An assignment to the `count` field in the record bound to `y` has interesting consequences:

```
y.count <- 1;;
val it : unit = ()
x = y;;
val it : bool = false
y = z;;
val it : bool = true
z;;
val it : intRec = {count = 1;}
```

The assignment to the `count` field of `y` has hence not only changed `y` but also `z`. Environment and store give the explanation: the declarations create the following environment and store (prior to the assignment of `y.count`) where `x=y` and `y=z`:

Environment	Store
x $\mapsto$ { count $\mapsto$ loc_3 }	loc_3: 0
y $\mapsto$ { count $\mapsto$ loc_4 }	loc_4: 0
z $\mapsto$ { count $\mapsto$ loc_4 }	

The assignment changes the store but leaves the environment unchanged:

Environment	Store
x $\mapsto$ { count $\mapsto$ loc_3 }	loc_3: 0
y $\mapsto$ { count $\mapsto$ loc_4 }	loc_4: 1
z $\mapsto$ { count $\mapsto$ loc_4 }	

The crucial point is that the records bound to y and z *share* the location loc_4. One says that z is an *alias* of y. Sharing and aliasing can have unexpected and unpleasant effects in imperative programming. These phenomena do not exist in pure functional programming where a value is immutable – not to be changed.

It should be remembered that a record is a *value* in F# – assignment to a record is not possible. If required, one may declare a location containing a record:

```
let mutable t = x;;
val mutable t : intRec = {count = 0;}
```

This gives the following environment and store:

Environment	Store
x $\mapsto$ { count $\mapsto$ loc_3 }	loc_3: 0
y $\mapsto$ { count $\mapsto$ loc_4 }	loc_4: 1
z $\mapsto$ { count $\mapsto$ loc_4 }	
t $\mapsto$ loc_5	loc_5: { count $\mapsto$ loc_3 }

and one may, for example, assign the value of y to t:

```
t <- y;;
val it : unit = ()
t;;
val it : intRec = {count = 1;}
```

This assignment changes the contents of loc_5 to the value {count $\mapsto$ loc_4}.

The above examples illustrate some of the (pleasant and unpleasant) features of records with mutable fields. The real importance is, however, their key role in handling objects from F#. An assignable member of an object appears in F# as a mutable record field that can be assigned using the <− operator, for example:

```
open System.Globalization;;
open System.Threading;;
Thread.CurrentThread.CurrentCulture <- CultureInfo "en-US";;
```

modifying the value of the CurrentCulture member of the CurrentThread object.

8.6 References

The F# compiler does *not* accept use of locally declared mutables in locally declared functions.[1] The above clock example could hence not be made without using records.

The ref type provides a shorthand for a record type containing a single mutable field, and the ref function provides a shorthand for a value of this type. They appear[2] as defined as follows:

```
type 'a ref  = { mutable contents: 'a }
let ref v    = { contents = v }
```

[1] The restriction is related to the memory management where problems might arise if a function was allowed to return a closure using a locally defined mutable.

[2] The symbol ! cannot be used as a *user-defined* prefix operator.

```
let (~!) r   = r.contents
let (:=) r v = r.contents <- v
```

The declaration

```
let x = ref [1; 2];;
val x : int list ref = {contents = [1;2];}
```

will hence give the following extension of environment and store:

Environment	Store
x ↦ { contents ↦ *loc* }	*loc*: [1;2]

with a location *loc* of type `int list` containing the value `[1;2]`, and a binding of `x` to the record {contents ↦ *loc*}.

The operators `!` and `:=` work as follows:

```
!x;;
val it : int list = [1; 2]]
x := [3;4];;
val it : unit = ()
!x;;
val it : int list = [3; 4]
```

The `makeCounter` example in the previous section can be made using references:

```
let incr r = (r := !r + 1 ; !r);;
let makeCounter() =
    let counter = ref 0
    fun () -> incr counter;;
```

8.7 While loops

If b denotes an expression of type `bool` and e denotes an expression of any type, then

`while` b `do` e

will be an expression of type `unit`. This expression is evaluated as follows:

1. Evaluate the expression b.
2. If the result is true, then evaluate the expression e and repeat the evaluation of the expression `while` b `do` e. If the result is false, then terminate the evaluation of `while` b `do` e and return the result `()` of type `unit`.

These rules are expressed in the following *evaluation steps* for a `while` loop:

`while` b `do` e	$\rightsquigarrow$	e `;` `while` b `do` e	if b evaluates to true
`while` b `do` e	$\rightsquigarrow$	()	if b evaluates to false

where we have used the sequential composition operator "`;`" (cf. Section 8.4).

A `while` loop is only useful when evaluated in a context where some identifiers are bound to mutables. The expression e should contain assignment to some of these mutables and b should comprises tests of some of their values.

The `while` loop:

`while` b `do` e

has the same evaluation steps as the expression:

`wh()`

where the function `wh` is declared by:

```
let rec wh() = if b then (e ; wh()) else ();;
```

The evaluation of `wh()` will evaluate the expression e repeatedly until b becomes false and that is exactly what is done by the evaluation of the `while` loop (we assume that the identifier `wh` does not occur in b or e). Thus, any `while` loop can be expressed by a recursive function declaration.

It should be noted that the F# compiler generates essentially the same binary code for the `while`-loop and the function `wh` (the recursive call `wh()` is compiled to a branch instruction). There is hence *no* performance advantage in using the loop instead of the recursive declaration. See also Section 9.5, especially the examples on Page 211.

8.8 Imperative functions on lists and other collections

The F# library contains functions `iter` and `iteri` on lists:

```
List.iter:   ('a -> unit) -> 'a list -> unit
List.iteri:  (int -> 'a -> unit) -> 'a list -> unit
```

These functions are used to *iterate* the effect of an imperative function over the elements of a list. Let

`lst = [`v_0`; `v_1`; ...; `v_{n-1}`]`

be a list with elements of type `'a` and let

`f: 'a -> unit`

be an imperative function. The evaluation of the expression

```
List.iter f lst
```

will then successively apply the function `f` to the elements $v_0, v_1, \ldots, v_{n-1}$ of the list. The result (of type `unit`) of the evaluation is of no interest and the interesting part of the evaluation is the side-effect. The following is a (not very interesting) application of `List.iter`:

```
let mutable sum = 0;;
let f x = sum <- sum + x;;
List.iter f [1; 2; -3; 5];;
val it : unit = ()
sum;;
val it : int = 5
```

The function `iteri` includes the index k of the element v_k in the computations. Let `f` be a function of type

```
f: int -> 'a -> unit
```

The evaluation of the expression

```
List.iteri f lst
```

will then successively evaluate the function calls:

$$\texttt{f 0}\ v_0,\ \ \texttt{f 1}\ v_1,\ \ldots,\ \ \texttt{f (}n-1\texttt{)}\ v_{n-1}$$

The interesting part of the evaluation is again the side-effect. The following is another (not very interesting) application of `List.iteri`:

```
let mutable t = 0;;
let f k x = t <- t + k * x;;
List.iteri f [1; 2; -3; 5];;
t;;
val it : int = 11
```

The evaluation of `List.iteri f [1; 2; -3; 5]` accumulates the value:

```
0 + 0 * 1 + 1 * 2 + 2 * (-3) + 3 * 5 = 11
```

in the variable `t`.

The functions `iter` and `iteri` on other collections like `Seq`, `Set` and `Map` work in a similar way.

We refer to Exercise 9.14 for an analysis of the run time of the function `List.iter` and `Set.iter`.

8.9 Imperative tree traversal

The tree traversal functions introduced in Section 6.3 and Section 6.6 have imperative counterparts where an imperative function is called whenever an element in a tree is visited. Such functions are useful in many applications and give for instance a convenient way of producing output while traversing a tree. In solving a programming problem you will often make problem-specific imperative traversal functions corresponding to a problem-specific tree type. The declarations in this section and in Section 8.13 could then serve as models for such declarations.

The imperative traversal functions on a binary tree are obtained directly from the definition of the traversal in Section 6.3 using the same `binTree<'a>` type:

```
let rec preIter f = function
    | Leaf           -> ()
    | Node(tl,x,tr) -> f x ; preIter f tl ; preIter f tr;;
val preIter : ('a -> unit) -> BinTree<'a> -> unit
```

```
let rec inIter f = function
    | Leaf           -> ()
    | Node(tl,x,tr) -> inIter f tl ; f x ; inIter f tr;;
val inIter : ('a -> unit) -> BinTree<'a> -> unit
```

and similar for `postIter`. Applying, for example, `preIter` to the tree `t4` in Section 6.3 gives:

```
preIter (fun x -> printf " %d" x) t4;;
 5 0 -3 2 7
```

We may in a similar way define a function for imperative depth-first traversal of list trees as described in Section 6.6:

```
let rec depthFirstIter f (Node(x,ts)) =
    f x ; List.iter (depthFirstIter f) ts;;
val depthFirstIter : ('a -> unit) -> ListTree<'a> -> unit
```

Applying this function to the tree `t1` in Section 6.6 we get:

```
depthFirstIter (fun x -> printf "  %d" x) t1;;
  1  2  5  3  4  6  7
```

The `breadthFirstIter` function is declared in Section 8.13.

8.10 Arrays

The addresses in the *physical* memory of the computer are integers. Consider a sequence of n equally sized contiguous memory locations $loc_0, loc_1, \ldots, loc_{n-1}$ as shown in Figure 8.1.

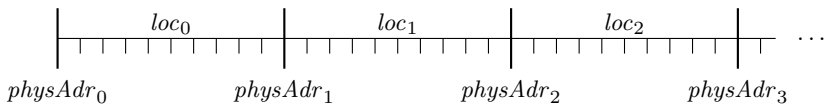


Figure 8.1 Memory layout of an array

The physical address $physAdr_k$ of the k'th location loc_k can in this situation be computed by the formula:

$$physAdr_k = physAdr_0 + k \cdot s$$

where s denotes the size of one location. The machine code computation of $physAdr_k$ requires hence only two arithmetic operations.

This addressing scheme is used to implement arrays. An *array* of length n consists of n locations $loc_0, loc_1, \ldots, loc_{n-1}$ of the same type. The numbers $0, 1, \ldots, n-1$ are called the *indices* of the elements. The type of the array is written

$$\tau \texttt{ [ ]}$$

where τ is the type of the elements.

Arrays have the advantage over lists that any array location can be accessed and modified in a constant (short) time, that is, in a small number of computations which is independent of the size of the array. On the other hand, an array is a mutable object — the old value is lost when a location is modified. Furthermore, an array cannot be extended by more elements in a simple way as the adjacent physical memory (after the last element in the array) might be occupied for other use. A selection of operations on arrays is shown in Table 8.1.

An array can be entered using the "`[|...|]`" notation, for example:

```
let a = [|4;5;6;7|];;
val a : int [] = [|4; 5; 6; 7|]

let b = [|'a' .. 'f'|];;
val b : char [] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'|]
```

Individual array locations are assigned using indexing:

```
b.[2] <- 'z';;
val it : unit = ()
b;;
val it : char [] = [|'a'; 'b'; 'z'; 'd'; 'e'; 'f'|]
```

Name	Type	Function
`.[ ]`	`'a [] * int -> 'a` "location"	*arr*`.[`k`]` is loc_k. May raise IndexOutOfRangeException
`Array.length`	`'a [] -> int`	Length n of array
`Array.ofList`	`'a list -> 'a []`	`Array.ofList [`x_0`;...]` is new array of values $x_0, \ldots$
`Array.toList`	`'a [] -> 'a list`	`Array.toList` *arr* is list `[`val_0`; ...;` val_{n-1}`]`
`Array.ofSeq`	`seq<'a> -> 'a []`	Array init'ed to seq elem's
`Array.toSeq`	`'a [] -> seq<'a>`	Seq of array val's
`Array.map`	`'a -> 'b -> 'a [] -> 'b []`	`Array.map` f *arr* is new array of $f\ val_0, f\ val_1, \ldots$
`Array.mapi`	`int -> 'a -> 'b -> 'a [] -> 'b []`	`Array.mapi` f *arr* is new array of $f\ 0\ val_0, f\ 1\ val_1, \ldots$
`Array.iter`	`'a -> unit -> 'a [] -> unit`	`Array.iter` f *arr* is the effect of $f\ val_0$`;` $f\ val_1$`; ...`
`Array.iteri`	`int -> 'a -> unit -> 'a [] -> unit`	`Array.iteri` f *arr* is the effect of $f\ 0\ val_0$`;` $f\ 1\ val_1$`; ...`
`Array.fold`	`'b -> 'a -> 'b -> 'b -> 'a [] -> 'b`	`Array.fold` f b *arr* is $f\ (\ldots(f\ b\ val_0)\ldots)\ val_{n-1}$
`Array.foldBack`	`'a -> 'b -> 'b -> 'a [] -> 'b -> 'b`	`Array.foldBack` f *arr* b is $f\ val_0\ (\ldots(f\ val_{n-1}\ b)\ldots)$

Metasymbol *arr* denotes an array of locations $loc_0, \ldots, loc_{n-1}$ containing the values $val_0, \ldots, val_{n-1}$

Table 8.1 *Operations on arrays*

Example: Histogram

Arrays are very convenient when counting frequencies and making a histogram. The following small program reeds a text file given by its directory path and count the frequency of each character `'A'` to `'Z'` (not distinguishing small and capital letters) and prints the resulting histogram. The reader may consult Section 10.3 about the used text I/O functions and Section 10.7 about `printf` formats.

```
open System;;
open System.IO;;

let Acode = int 'A'

let histogram path =
    let charCount   = [| for n in 'A'..'Z' -> 0 |]
    let file = File.OpenText path
    while (not file.EndOfStream) do
        let ch = char( file.Read() )
        if (Char.IsLetter ch) then
            let n = int (Char.ToUpper ch)  - Acode
            charCount.[n] <- charCount.[n] + 1
        else ()
    file.Close()
    let printOne n c = printf "%c:  %4d\n" (char(n + Acode)) c
    Array.iteri printOne charCount;;
```

Calling the function `histogram` on the path of the source file `histogram.fsx` will, for example, give the output:

```
A:    20
B:     0
C:    22
...
X:     1
Y:     3
Z:     1
```

where we have only shown some of the lines.

8.11 Imperative set and map

The .NET library `System.Collections.Generic` contains classes implementing imperative sets and maps:

Set	Map	Data representation
`SortedSet<'a>`	`SortedDictionary<'a,'b>`	Search tree
`HashSet<'a>`	`Dictionary<'a,'b>`	Hash-key table

The classes `SortedSet<'a>` and `SortedDictionary<'a,'b>` are imperative versions of `set<'a>` and `map<'a,'b>` where the member functions modify the collection

in a "destructive" update without retaining the old value. They should only be used in algorithms using and maintaining a single "current" collection without ever referring to "old" values.

The `HashSet<'a>` and `Dictionary<'a,'b>` are implemented using hash-key technique: The basic data structure is an array (say of length `N`) where an element a (entry (a, b)) is stored in the array location with index:

$$index(a) = \text{hash}(a) \ \texttt{\%} \ \texttt{N}$$

where `hash` is the hash function of the equality type `'a`. This is a rather efficient scheme, but it runs into problems when multiple elements (entries) have the same index and hence should be stored in the same array location. This *collision* problem is solved by storing the colliding elements (entries) in a linked structure that can be accessed via the index value. The `HashSet<'a>` and `Dictionary<'a,'b>` collections have the following characteristics:

- The operations are very efficient.
- They are strictly imperative with destructive updating.
- They do not offer a traversal of the elements (entries) in sorted order.

A selections of operations on the imperative set and map classes are shown in Tables 8.2 and 8.3.

Indexing in a `Directory` by a key can be used to update a value by assignment:

map`.[`*key*`]` `<-` *newValue*

and this construction may also be used to add a new entry to the directory.

Name	Type	Function
`SortedSet<`*type*`>`	`unit ->` *Set*`<`*type*`>`	Create an empty SortedSet
`HashSet<`*type*`>`	`unit ->` *Set*`<`*type*`>`	Creates an empty HashSet
hashSet`.Count`	`int`	No. of elements in *hashSet*
set`.Add`	*type* `-> unit`	Add element to *set*
set`.Contains`	*type* `-> bool`	Value contained in *set*?
set`.Remove`	*type* `-> bool`	Remove value from *set*, `false`: value not inset
set`.UnionWith`	*Set*`<`*type*`>` `-> unit`	Add elements in other set
set`.IntersectWith`	*Set*`<`*type*`>` `-> unit`	Remove elements of other set
set`.IsProperSubsetOf`	*Set*`<`*type*`>` `-> bool`	Is proper subset of set ?
set`.IsSubsetOf`	*Set*`<`*type*`>` `-> bool`	Is subset of set ?
set`.Overlaps`	*Set*`<`*type*`>` `-> bool`	Overlaps set ?

Metasymbols: *Set*`<`*type*`>` denotes `SortedSet<`*type*`>` or `HashSet<`*type*`>`
set and *hashSet* denote values of types *Set*`<`*type*`>` and `HashSet<`*type*`>`

Table 8.2 *A selection of operations on* `SortedSet` *and* `HashSet`

Name	Type	Function
`SortedDictionary<`*kTyp*`,`*vTyp*`>`	`unit -> `*Map*`<`*kTyp*`,`*vTyp*`>`	Creates empty Sorted Dictionary
`Dictionary<`*kTyp*`,`*vTyp*`>`	`unit -> `*Map*`<`*kTyp*`,`*vTyp*`>`	Creates empty Dictionary
dir`.Count`	`int`	No of elements in *map*
map`.Add`	*kTyp* `*` *vTyp* `-> unit`	Add entry to *map*
map`.ContainsKey`	*kTyp* `-> bool`	*map* contains key ?
map`.ContainsValue`	*vTyp* `-> bool`	*map* contains value ?
map`.Remove`	*kTyp* `-> bool`	Remove an entry from *map* `false`: entry not found
map`.TryGetValue`	*kTyp* `-> bool *` *vTyp*	Search entry by key
dir`.[ ]`	*kTyp* `->` *vTyp* "`mutable`"	Value location for key

Metasymbols: *Map*`<`*kTyp*`,`*vTyp*`>` denotes `SortedDictionary<`*kTyp*`,`*vTyp*`>` or `Dictionary<`*kTyp*`,`*vTyp*`>`, *map* denotes a value of type *Map*`<`*kTyp*`,`*vTyp*`>` while *dir* denotes a value of type `Dictionary<`*kTyp*`,`*vTyp*`>`

Table 8.3 *A selection of operations on* `SortedDictionary` *and* `Dictionary`

8.12 Functions on collections. Enumerator functions

Operations on collections should preferably be done using standard library functions, but this is not always feasible. This section presents means for defining functions on collections in a way that resembles the definition of functions on lists using list patterns as described in Chapter 4.

Enumerator functions

The `System.Collections.Generic` library contains imperative features for element-wise traversal of any of the collections – including the F# collections `list`, `set`, `map`, etc. The `enumerator` function of the book (to be declared on Page 193) makes these features available in a functional setting. Applying `enumerator` to a collection yields a function:

`enumerator(`*collection*`): unit -> `*elementType* `option`

where *elementType* is determined as follows:

collection	*elementType*
NonMapOrDictionaryCollection`<'a>`	`'a`
MapOrDictionaryCollection`<'a,'b>`	`KeyValuePair<'a,'b>`

An element *entry* of type `KeyValuePair<'a,'b>` corresponds to an entry in the map or dictionary, and it has components:

entry`.Key`	of type `'a`
entry`.Value`	of type `'b`

Applying `enumerator` to a set creates an imperative *enumerator function* where successive calls yield the elements in the set:

```
let f = enumerator (Set.ofList [3 ; 1; 5]);;
val f : (unit -> int option)
```

```
f();;
val it : int option = Some 1

f();;
val it : int option = Some 3

f();;
val it : int option = Some 5

f();;
val it : int option = None
```

Applying the function `enumerator` to a dictionary creates a similar enumerator function but the entries are obtained as values of the corresponding `KeyValuePair` type:

```
let d = SortedDictionary<string,int>();;
d.Add("cd",3) ; d.Add("ab",5);;
let g = enumerator d;;
val g : (unit -> KeyValuePair<string,int> option)
g();;
val it : KeyValuePair<string,int> option =
    Some [ab, 5] {Key = "ab"; Value = 5;}
g();;
val it : KeyValuePair<string,int> option =
    Some [cd, 3] {Key = "cd"; Value = 3;}
g();;
val it : KeyValuePair<string,int> option = None
```

The elements in a `set<'a>` or `SortedSet<'a>` collection are traversed in the order given by the ordering in the type `'a` while the elements in a `HashSet<'a>` collection are traversed in some order depending on the hashing function and the order in which elements were added. The entries in a `map<'a,'b>` or `SortedDictionary<'a,'b>` collection are traversed in the order given by the ordering in the type `'a` while the elements in a `Dictionary<'a,'b>` collection are traversed in some order depending on hashing function and the creation of the collection.

The enumerator may, for example, be used to define the `tryFind` function on sets:

```
let tryFind p (s: Set<'a>) =
     let f = enumerator s
     let rec tFnd () =
         match f() with
         | None   -> None
         | Some x ->
             if (p x) then Some x else tFnd()
     tFnd();;
val tryFind : ('a -> bool) -> Set<'a> -> 'a option
                                when 'a : comparison
```

```
let s = Set.ofList [1;3;4;5];;
val s : Set<int> = set [1; 3; 4; 5]

tryFind (fun n -> n%2 = 0) s;;
val it : int option = Some 4
```

We refer to Exercise 9.13 for an analysis of the run time of this function and the version declared on Page 109 .

Declaring the `enumerator` *function*

The declarations of the `enumerator` function is based on the concept of an *enumerator object* for a collection. An enumerator object is a mutable data structure that is able to describe a traversal of the collection by pointing to each element one after the other. The implementation of the enumerator object depends on the data representation of the associated collection but all enumerator objects are of the same polymorphic type `IEnumerator<'c>`. This polymorphism has been obtained by letting any enumerator implement the *interface*:

```
type IEnumerator<'c> =
    abstract Current : 'c
    abstract MoveNext : unit -> bool;;
```

for some type `'c`. An enumerator object *enum* points to the element *enum*`.Current` and it is forwarded to the next element by evaluating *enum*`.MoveNext()`. An initial call of `MoveNext` is required to get a fresh enumerator to point to the first element, and the value of `MoveNext` becomes `false` when the enumerator gets beyond the last element in the collection.

Each collection has its own `GetEnumerator` member to create an enumerator object. This object gets the following type:

Collection	Enumerator object
NonMapOrDictionaryCollection`<'a>`	`IEnumerator<'a>`
MapOrDictionaryCollection`<'a,'b>`	`IEnumerator<KeyValuePair<'a,'b>>`

The implementation is made such that the `GetEnumerator` member for any specific collection can be considered an instance of a polymorphic `GetEnumerator` member working on any collection. This polymorphism has been obtained by letting each collection implement the interface:

```
type IEnumerable<'c> =
    abstract GetEnumerator : unit -> IEnumerator<'c>
```

Any collection is hence construed as an object of type `IEnumerable<'c>` where `'c` is the type parameter of the enumerator object as described above.

The `enumerator` function refers to the collection using the `IEnumerable` type and may hence be applied to any collection. It creates a reference `e` to an enumerator object and this reference is then used inside a local function `f` that is returned as the result. A reference is required because of the restriction on the use of mutable in closures:

```
open System.Collections.Generic;;

let enumerator (m: IEnumerable<'c>) =
    let e = ref (m.GetEnumerator())
    let f () =
        match (!e).MoveNext() with
        | false -> None
        | _     -> Some ((!e).Current)
    f;;
val enumerator : IEnumerable<'c> -> (unit -> 'c option)
```

Enumerators versus list patterns

The elegant and efficient use of list patterns as described in Chapter 4 depends on the fact that the tail of a list is represented directly by a sub-component of the data structure representing the list. Matching a list with the pattern `x::xs` is hence a very fast operation, and the same applies to matching with other list patterns.

The situation is different for set and map collections. Consider, for example, a set `s` represented by a balanced search tree. Finding the first (least) element `x` of `s` is a very fast operation, but the remaining part of the search tree consists of two separate trees and is *not* represented by a sub-tree of the search tree. The time for computing `Set.remove x s` is in fact proportional to the depth of the search tree and hence proportional to the logarithm of the number of elements in the set. The computation time for making a complete recursion over a set with successive computations `Set.remove x s` of trees representing sub-sets is hence proportional to $n \log n$ where n is the number of elements in the set.

The enumerator function of a set (or map) represented by a search tree is based on the idea of in-order traversal of the tree. The enumerator uses an imperative data structure to represent a stage in this traversal, where a specific element (entry) is reached. Each call of the enumerator function steps this data structure forward to the next element (entry) by a small modification with constant computation time. The computation time for making a complete recursion over a search-tree based set (or map) using the enumerator function is hence proportional to the number of elements in the set (entries in the map), and the time performance equals the time performance of pattern-matching for lists.

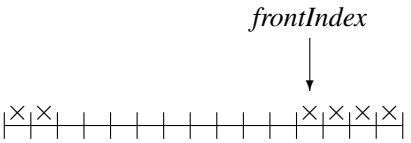
The enumerator functions for the hash-key based `HashSet` and `Dictionary` collections work in a different way because we have a different data representation using an array as described in Section 8.11. The traversal scans forward through the array (and through each linked structure of colliding elements). The time used by a complete recursion over a hashed collection using the enumerator function is hence proportional to the number of elements (entries) in the set (map), and this corresponds to the time used by a complete recursion over a list using pattern-matching.

8.13 Imperative queue

The `System.Collections.Generic` library contains an imperative queue implementation in the form of a `Queue<'a>` class with members:

```
q.Enqueue: 'a -> unit
q.Dequeue: unit -> 'a
q.Count: int
```

The queue is implemented using an array where the front queue element is stored in an array location with index *frontIndex* while the rest of the queue is stored in the succeeding locations with a wraparound to the beginning of the array if the queue extends beyond the end of the array.



The `Dequeue` operation returns the array element with index *frontIndex* and advances *frontIndex* to the next array position (with a possible wraparound) while the `Enqueue` operation stores the enqueued value in the first free array location. An `Enqueue` operation with a filled array causes an *array replacement* where a new, larger array is reserved and all queue elements are moved to the new array upon which the old array is abandoned.

A queue can be used to make the following elegant and interesting implementation of the breadth-first traversal of list trees in Section 6.6.

```
type ListTree<'a> = Node of 'a * (ListTree<'a> list);;

let breadthFirstIter f ltr =
    let remains = Queue<ListTree<'a>>()
    remains.Enqueue ltr
    while (remains.Count <> 0) do
        let (Node (x,tl)) = remains.Dequeue()
        List.iter (remains.Enqueue) tl
        f x;;
val breadthFirstIter : ('a -> unit) -> listTree<'a> -> unit
```

The idea is to let the *queue* `remains` contain those list trees where the nodes remain to be visited, initially the tree `ltr`. A list tree `Node (x,tl)` is dequeued, the elements of the list `tl` of sub-trees are enqueued one-by-one using `List.iter` – and the root node `x` is visited. This procedure is repeated until the queue becomes empty.

8.14 Restrictions on polymorphic expressions

The purpose of the restriction on polymorphic expressions in Section 4.5 is to ensure that the use of mutables is type safe. The problem is illustrated by the following hypothetical example:

```
let mutable a = [];;  // NOT allowed !!
val mutable a = []

let f x = a <- (x :: a);;
val f : 'a -> unit

f(1);;
it : unit = ()
f("ab");;
it : unit = ()
a;;
it : ? list = ["ab"; 1]  *** Oops! type error !
```

The point is that F# would be forced to infer a type of `f` prior to any use of the function. This would result in the type `'a -> unit` because apparently values of any type can be cons'ed onto the *empty* list. Hence each of the applications `f(1)` and `f("ab")` would type check because `int` as well as `string` are instances of the polymorphic type `'a`. The type check would hence fail to discover the illegal expression `"ab"::[1]` emerging during the evaluation of `f("ab")`.

The declaration

```
let mutable a = [];;
```

is construed as binding `a` to the value of the polymorphic expression "`mutable []`" and this expression is *not* considered a value expression. The declaration is hence rejected by the restriction on the use of polymorphic expressions.

Summary

The chapter provides a semantical framework, the *store*, for understanding the imperative features of F# that operate on and modify the state of mutable objects. A store consists of a set of locations containing values, together with operators to access and change the store. The main imperative constructs of F# is introduced together with extracts of .NET libraries for imperative collections, including arrays, sets and maps. We explain why the restriction on polymorphically typed expressions is needed because of the imperative features.

Exercises

8.1 Make a drawing of the environment and store obtained by the following declarations and assignments:

```
let mutable x = 1;;
let mutable y = (x,2);;
let z = y;;
x <- 7;;
```

8.2 The sequence of declarations:

```
let mutable a = []
let f x = a <- (x :: a)
f(1);;
```

are accepted by F#. Explain why.

8.3 Make a drawing of the environment and store obtained by the following declarations and assignments:

```
type t1 = { mutable a : int };;
type t2 = { mutable b : int ; c : t1 };;
let x = { a = 1 };;
let y = { b = x.a ; c = x };;
x.a <- 3;;
```

8.4 Declare `null` to denote the default value of the record type:

```
type t = { mutable link : t ; data : int };;
```

Declare some other values of type `t` and use assignment to build chains and circles of values of type `t`. Declare a function to insert an element in the front of a chain of values of type `t`.

8.5 Give a declaration of the `gcd` function using a `while` loop instead of recursion (cf. Section 1.8).

8.6 Declare a function for computing Fibonacci numbers F_n (see Exercise 1.5) using a `while` loop. Hint: introduce variables to contain the two previously computed Fibonacci numbers.

8.7 Use a `HashSet` traversal `for` loop to declare a function

```
HashSetFold: ('b -> 'a -> 'b) -> 'b -> HashSet<'a> -> 'b
```

such that

$f\ b\ set\ = f\ (\ldots(f\ (f\ b\ a_0)\ a_1)\ldots)\ a_{n-1}$

where $a_0, \ldots, a_{n-1}$ are the elements of the `HashSet` *set*.

8.8 Declare a `DictionaryFold` function. The type should correspond to the type of `Map.fold`.

8.9 Make declarations of `breadthFirst` and `breadthFirstFold` for list trees using an imperative queue.

Hint: unfold the while-loop in the declaration of `breadthFirstIter` to a local recursive function and use argument and value of this function to build the result.