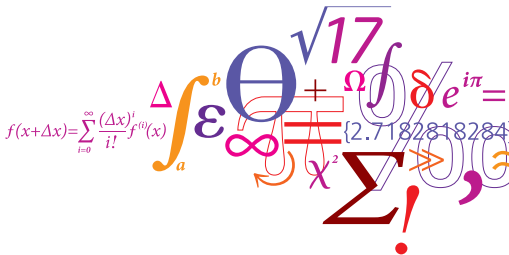


02157 Functional Programming

Collections: Sets and Maps

Michael R. Hansen



DTU Informatics

Department of Informatics and Mathematical Modelling

Sets and Maps as abstract data types

- Useful in the modelling and solution of many problems
- Many similarities with the list library

Recommendation: Use these libraries whenever it is appropriate.

The set concept (1)

A *set* (in mathematics) is a collection of element like

$\{\text{Bob, Bill, Ben}\}, \{1, 3, 5, 7, 9\}, \mathbb{N}$, and \mathbb{R}

- the sequence in which elements are enumerated is of no concern, and
- repetitions among members of a set is of no concern either

It is possible to decide whether a given value is in the set.

$\text{Alice} \notin \{\text{Bob, Bill, Ben}\}$ and $7 \in \{1, 3, 5, 7, 9\}$

The empty set containing no element is written $\{\}$ or \emptyset .

The sets concept (2)

A set A is a *subset* of a set B , written $A \subseteq B$, if all the elements of A are also elements of B , for example

$$\{\text{Ben, Bob}\} \subseteq \{\text{Bob, Bill, Ben}\} \quad \text{and} \quad \{1, 3, 5, 7, 9\} \subseteq \mathbb{N}$$

Two sets A and B are equal, if they are both subsets of each other:

$$A = B \quad \text{if and only if} \quad A \subseteq B \text{ and } B \subseteq A$$

i.e. two sets are equal if they contain exactly the same elements.

The subset of a set A which consists of those elements satisfying a predicate p can be expressed using a *set-comprehension*:

$$\{x \in A \mid p(x)\}$$

For example:

$$\{1, 3, 5, 7, 9\} = \{x \in \mathbb{N} \mid \text{odd}(x) \text{ and } x < 11\}$$

The set concept (3)

Some standard operations on sets:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

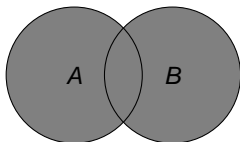
$$A \cap B = \{x \mid x \in A \text{ and } x \in B\}$$

$$A \setminus B = \{x \in A \mid x \notin B\}$$

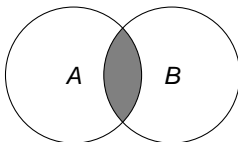
union

intersection

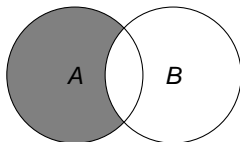
difference



(a) $A \cup B$



(b) $A \cap B$



(c) $A \setminus B$

Figure: Venn diagrams for (a) union, (b) intersection and (c) difference

For example

$$\{\text{Bob, Bill, Ben}\} \cup \{\text{Alice, Bill, Ann}\} = \{\text{Alice, Ann, Bob, Bill, Ben}\}$$

$$\{\text{Bob, Bill, Ben}\} \cap \{\text{Alice, Bill, Ann}\} = \{\text{Bill}\}$$

$$\{\text{Bob, Bill, Ben}\} \setminus \{\text{Alice, Bill, Ann}\} = \{\text{Bob, Ben}\}$$

An abstract Data Type: A type together with a collection of operations, where

- the representation of values is hidden.

An abstract data type for sets must have:

- Operations to generate sets from the elements. Why?
- Operations to extract the elements of a set. Why?
- Standard operations on sets.

The `Set` library of F# supports finite sets. An efficient implementation is based on a balanced binary tree.

Examples:

```
set ["Bob"; "Bill"; "Ben"];;  
val it : Set<string> = set ["Ben"; "Bill"; "Bob"]
```

```
set [3; 1; 9; 5; 7; 9; 1];;  
val it : Set<int> = set [1; 3; 5; 7; 9]
```

Equality of two sets is tested in the usual manner:

```
set["Bob";"Bill";"Ben"] = set["Bill";"Ben";"Bill";"Bob"];;  
val it : bool = true
```

Sets are order on the basis of a lexicographical ordering:

```
compare (set ["Ann";"Jane"]) (set ["Bill";"Ben";"Bob"]);;  
val it : int = -1
```

Selected operations (1)

- `ofList`: `'a list -> Set<'a>`,
where `ofList [a0;...;an-1] = {a0;...;an-1}`
- `toList`: `Set<'a> -> 'a list`,
where `toList {a0,...,an-1} = [a0;...;an-1]`
- `add`: `'a -> Set<'a> -> Set<'a>`,
where `add a A = {a} ∪ A`
- `remove`: `'a -> Set<'a> -> Set<'a>`,
where `remove a A = A \ {a}`
- `contains`: `'a -> Set<'a> -> bool`,
where `contains a A = a ∈ A`
- `minElement`: `Set<'a> -> 'a`
where `minElement {a0, a1, ..., an-2, an-1} = a0 when n > 0`

Notice that `minElement` is well-defined due to the ordering:

```
Set.minElement (Set.ofList ["Bob"; "Bill"; "Ben"]);;
val it : string = "Ben"
```


Selected operations (2)

- union: $\text{Set}\langle 'a \rangle \rightarrow \text{Set}\langle 'a \rangle \rightarrow \text{Set}\langle 'a \rangle$,
where union $A B = A \cup B$
- intersect: $\text{Set}\langle 'a \rangle \rightarrow \text{Set}\langle 'a \rangle \rightarrow \text{Set}\langle 'a \rangle$,
where intersect $A B = A \cap B$
- difference: $\text{Set}\langle 'a \rangle \rightarrow \text{Set}\langle 'a \rangle \rightarrow \text{Set}\langle 'a \rangle$,
where difference $A B = A \setminus B$
- exists: $('a \rightarrow \text{bool}) \rightarrow \text{Set}\langle 'a \rangle \rightarrow \text{bool}$,
where exists $p A = \exists x \in A. p(x)$
- forall: $('a \rightarrow \text{bool}) \rightarrow \text{Set}\langle 'a \rangle \rightarrow \text{bool}$,
where forall $p A = \forall x \in A. p(x)$
- fold: $('a \rightarrow 'b \rightarrow 'a) \rightarrow 'a \rightarrow \text{Set}\langle 'b \rangle \rightarrow 'a$,
where

$$\begin{aligned} & \text{fold } f \ a \ \{b_0, b_1, \dots, b_{n-2}, b_{n-1}\} \\ & = f(f(f(\dots f(f(a, b_0), b_1), \dots), b_{n-2}), b_{n-1}) \end{aligned}$$

These work similar to their List siblings, e.g.

$$\text{Set.fold } (-) \ 0 \ (\text{set } [1; 2; 3]) = ((0 - 1) - 2) - 3 = -6$$

where the ordering is exploited.

Example: Map Coloring (1)

Maps and colors are modelled in a more natural way using sets:

```
type country = string;;
type map     = Set<country*country>;;
type color   = Set<country>;;
type coloring = Set<color>;;
```

WHY?

Two countries c_1, c_2 are neighbors in a map m ,
if either $(c_1, c_2) \in m$ or $(c_2, c_1) \in m$:

```
let areNb c1 c2 m =
  Set.contains (c1,c2) m || Set.contains (c2,c1) m;;
```

Color col and be extended by a country c given map m ,
if for every country c' in col : c and c' are not neighbours in m

```
let canBeExtBy m col c =
  Set.forall (fun c' -> not (areNb c' c m)) col;;
```

Example: Map Coloring (2)

The function

```
extColoring: map -> coloring -> country -> coloring
```

is declared as a recursive function over the coloring:

WHY not use a fold function?

```
let rec extColoring m cols c =
  if Set.isEmpty cols
  then Set.singleton (Set.singleton c)
  else let col = Set.minElement cols
        let cols' = Set.remove col cols
        if canBeExtBy m col c
        then Set.add (Set.add c col) cols'
        else Set.add col (extColoring m cols' c);;
```

Notice similarity to a list recursion:

- base case [] corresponds to the empty set
- for a recursive case $x::xs$, the head x corresponds to the minimal element col and the tail xs corresponds to the "rests" set $cols'$

The list-based version is more efficient (why?) and more readable.

Example: Map Coloring (3)

A set of countries is obtained from a map by the function:

```
countries: map -> Set<country>
```

that is based on repeated insertion of the countries into a set:

```
let countries m =  
  Set.fold  
    (fun set (c1,c2) -> Set.add c1 (Set.add c2 set))  
    Set.empty  
  m;;
```

The function

```
colCntrs: map -> Set<country> -> coloring
```

is based on repeated insertion of countries in colorings using the `extColoring` function:

```
let colCntrs m cs = Set.fold (extColoring m) Set.empty cs;;
```

Example: Map Coloring (4)

The function that creates a coloring from a map is declared using functional composition:

```
let colMap m = colCntrs m (countries m);;  
  
let exMap = Set.ofList [("a","b"); ("c","d"); ("d","a")];;  
  
colMap exMap;;  
val it : Set<Set<string>>  
    = set [set ["a"; "c"]; set ["b"; "d"]]
```

The map concept

A *map* from a set A to a set B is a *finite* subset A' of A together with a *function* m defined on A' : $m : A' \rightarrow B$.

The set A' is called the *domain* of m : $\text{dom } m = A'$.

A map m can be described in a tabular form:

a_0	b_0
a_1	b_1
	⋮
a_{n-1}	b_{n-1}

- An element a_i in the set A' is called a *key*
- A pair (a_i, b_i) is called an *entry*, and
- b_i is called the *value* for the key a_i .

We denote the sets of entries of a map as follows:

$$\text{entriesOf}(m) = \{(a_0, b_0), \dots, (a_{n-1}, b_{n-1})\}$$

Selected map operations in F#

- `ofList: ('a*'b) list -> Map<'a,'b>`
`ofList [(a0, b0); ...; (an-1, bn-1)] = m`
- `add: 'a -> 'b -> Map<'a,'b> -> Map<'a,'b>`
`add a b m = m'`, where `m'` is obtained `m` by overriding `m` with the entry `(a, b)`
- `find: 'a -> Map<'a,'b> -> 'b`
`find a m = m(a)`, if `a ∈ dom m`;
 otherwise an exception is raised
- `tryFind: 'a -> Map<'a,'b> -> 'b option`
`tryFind a m = Some (m(a))`, if `a ∈ dom m`; `None` otherwise
- `foldBack: ('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c`
`foldBack f m c = f a0 b0 (f a1 b1 (f ... (f an-1 bn-1 c) ...))`

A few examples

```
let reg1 = Map.ofList [("a1",("cheese",25));  
                      ("a2",("herring",4));  
                      ("a3",("soft drink",5))];;  
val reg1 : Map<string,(string * int)> =  
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));  
       ("a3", ("soft drink", 5))]
```

An entry can be added to a map using `add` and the value for a key in a map is retrieved using either `find` or `tryFind`:

```
let reg2 = Map.add "a4" ("bread", 6) reg1;;  
val reg2 : Map<string,(string * int)> =  
  map [("a1", ("cheese", 25)); ("a2", ("herring", 4));  
       ("a3", ("soft drink", 5)); ("a4", ("bread", 6))]
```

```
Map.find "a2" reg1;;  
val it : string * int = ("herring", 4)
```

```
Map.tryFind "a2" reg1;;  
val it : (string * int) option = Some ("herring", 4)
```


An example using Map.foldBack

We can extract the list of article codes and prices for a given register using the fold functions for maps:

```
let reg1 = Map.ofList [("a1",("cheese",25));  
                      ("a2",("herring",4));  
                      ("a3",("soft drink",5))];;  
  
Map.foldBack (fun ac (_,p) cps -> (ac,p)::cps) reg1 [];  
val it : (string * int) list =  
    [("a1", 25); ("a2", 4); ("a3", 5)]
```

This and other higher-order functions are similar to their List and Set siblings.

Example: Cash register (1)

```
type articleCode = string;;
type articleName = string;;
type noPieces    = int;;
type price       = int;;

type info        = noPieces * articleName * price;;
type infoseq     = info list;;
type bill        = infoseq * price;;
```

The natural model of a register is using a map:

```
type register    = Map<articleCode, articleName*price>;;
```

since an article code is a *unique identification* of an article.

First version:

```
type item        = noPieces * articleCode;;
type purchase    = item list;;
```

Example: Cash register (1) - a recursive program

```

exception FindArticle;;

(* makebill: register -> purchase -> bill *)
let rec makeBill reg = function
  | []          -> ([],0)
  | (np,ac)::pur ->
      match Map.tryFind ac reg with
      | None          -> raise FindArticle
      | Some(aname,aprice) ->
          let tprice          = np*aprice
          let (infos,sumbill) = makeBill reg pur
          ((np,aname,tprice)::infos, tprice+sumbill));;

let pur = [(3,"a2"); (1,"a1")];;
makeBill reg1 pur;;
val it : (int * string * int) list * int =
  ([[(3, "herring", 12); (1, "cheese", 25)]], 37)

```

- the lookup in the register is managed by a `Map.tryFind`

Example: Cash register (2) - using List.foldBack

```
let makeBill' reg pur =
  let f (np,ac) (infos,billprice)
      = let (aname, aprice) = Map.find ac reg
          let tprice      = np*aprice
              ((np,aname,tprice)::infos, tprice+billprice)
      List.foldBack f pur ([],0);;

makeBill' reg1 pur;;
val it : (int * string * int) list * int =
  ((3, "herring", 12); (1, "cheese", 25)], 37)
```

- the recursion is handled by List.foldBack
- the exception is handled by Map.find

Example: Cash register (2) - using maps for purchases

The purchase: 3 herrings, one piece of cheese, and 2 herrings, is the same as a purchase of one piece of cheese and 5 herrings.

A purchase associated number of pieces with article codes:

```
type purchase = Map<articleCode,noPieces>;;
```

A bill is produced by folding a function over a map-purchase:

```
let makeBill'' reg pur =
  let f ac np (infos,billprice)
    = let (aname, aprice) = Map.find ac reg
        let tprice      = np*aprice
            ((np,aname,tprice)::infos, tprice+billprice)
      Map.foldBack f pur ([],0);;

let purMap = Map.ofList [("a2",3); ("a1",1)];;
val purMap : Map<string,int> = map [("a1", 1); ("a2", 3)]

makeBill'' reg1 purMap;;
val it = ([(1, "cheese", 25); (3, "herring", 12)], 37)
```

- The concepts of sets and maps.
- Fundamental operations on sets and maps.
- Applications of sets and maps.