# 02157 Functional Programming

## Collections: Finite Sets and Maps

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

# Overview

Sets and Maps as abstract data types

- Useful when modelling
- Useful when programming
- Many similarities with the list library

Recommendation: Use these libraries whenever it is appropriate.

Succinct code through applications of higher-order functions

# FSharp's immutable collections

- List: a finite sequence of elements of the same type
  - the sequence in which elements are enumerated is important
  - repetitions among elements of a list matters

- Set: a finite collection of elements of the same type
  - the sequence in which elements are enumerated is of no concern
  - repetitions among members of a set is of no concern

Today

- Map: a finite function from a domain of keys to values
  - the uniqueness of keys is an important property

Today

- Sequence: a possibly infinite sequence of elements of the same type
  - the elements of a sequence are computed by demand

Covered later in the semester

# Types, data types and abstract data types

- A type is generated from basic types `int, float, bool, string, ...` and type variables `'a, 'b, 'c, ...` using type operators `*, ->, list, ...`

- A data type is characterized by
  - a type — `'alist`
  - a set of values — `[ ]`, `[v]`, `[v1; ...vn]`
  - a set of operations — `::`, `@` , `List.rev`, `List.fold`, ...

- A abstract data type is a data type
  - where the representation of values is hidden — LiskovZilles 1974

Examples:

- `List` is a data type but not an abstract one
  — the representation of list values is visible (`[]` and `::`)

- `Set` and `Map` are abstract data types

# The set concept (1)

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

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

- 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.

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

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

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

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

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

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*:

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

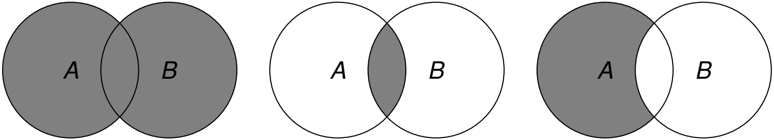
For example:

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

# The set concept (3)

Some standard operations on sets:

$$\begin{array}{lcll} A \cup B & = & \{x \mid x \in A \text{ or } x \in B\} & \text{union} \\ A \cap B & = & \{x \mid x \in A \text{ and } x \in B\} & \text{intersection} \\ A \setminus B & = & \{x \in A \mid x \notin B\} & \text{difference} \end{array}$$




(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

$$\begin{array}{lcl} \{\text{Bob}, \text{Bill}, \text{Ben}\} \cup \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Alice}, \text{Ann}, \text{Bob}, \text{Bill}, \text{Ben}\} \\ \{\text{Bob}, \text{Bill}, \text{Ben}\} \cap \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Bill}\} \\ \{\text{Bob}, \text{Bill}, \text{Ben}\} \setminus \{\text{Alice}, \text{Bill}, \text{Ann}\} & = & \{\text{Bob}, \text{Ben}\} \end{array}$$

# Abstract Data Types

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.

# An Abstract Data Type: `Set<a'>`

An abstract type for sets should at least support the following:

```
empty:       Set<'a>
add:         'a -> Set<'a> -> Set<a'>
union:       Set<'a> -> Set<'a> -> Set<a'>
intersect:   Set<'a> -> Set<'a> -> Set<a'>
difference:  Set<'a> -> Set<'a> -> Set<a'>
contains:    'a -> Set<'a> -> bool
toList:      Set<'a> -> a' list
```

where

- any finite set can be generated by repeatedly adding elements to the empty set;
- union, intersection and difference are fundamental set operations;
- `contains` and `toList` are used to inspect the set

Note:

- the above operations are supported by the library `Set`.
- the representation of sets used by `Set` is hidden from the user.

# Finite sets in F#

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

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 ordered on the basis of a lexicographical ordering:

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

# Immutability of `Set<'a>`

```
let s = Set.ofList [3; 2; 0];;
val s : Set<int> = set [0; 2; 3]

Set.add 1 s;;
val it : Set<int> = set [0; 1; 2; 3]

s;;
val it : Set<int> = set [0; 2; 3]
```

Evaluation of Set.add 1 s does not change the value of s.

# Selected further operations (1)

- `ofList: 'a list -> Set<'a>`,
  where `ofList` $[a_0; \ldots; a_{n-1}] = \{a_0; \ldots; a_{n-1}\}$
- `remove: 'a -> Set<'a> -> Set<'a>`,
  where `remove` $a\,A = A \setminus \{a\}$
- `minElement: Set<'a> -> 'a`
  where `minElement` $\{a_0, a_1, \ldots, a_{n-2}, a_{n-1}\} = a_0$ when $n > 0$
  (assuming that the enumeration respect the ordering)

Notice that minElement on a non-empty set is well-defined due to the ordering:

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

# Selected further operations (2)

- `filter: ('a -> bool) -> Set<'a> -> Set<'a>`, where `filter` $p\ A = \{x \in A \mid p(x)\}$
- `exists: ('a -> bool) -> Set<'a> -> bool`,
  where `exists` $p\ A = \exists x \in A. p(x)$
- `forall: ('a -> bool) -> Set<'a> -> bool`,
  where `forall` $p\ A = \forall x \in A. p(x)$
- `fold: ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a`,
  where

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

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

`Set.fold (-) 0 (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?

The function:

```
areNb: Country -> Country -> Map -> bool
```

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 =  ?
```

Remember:

```
contains:   'a -> Set<'a> -> bool
exists: ('a -> bool) -> Set<'a> -> bool
```

# Example: Map Coloring (2)

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>;;
```

The function

```
canBeExtBy: Map -> Color -> Country -> bool
```

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 =  ?
```

Remember

```
forall: ('a -> bool) -> Set<'a> -> bool
```

# Example: Map Coloring (3)

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 better readable.

# Example: Map Coloring (4)

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>;;
```

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 =  ?
```

Remember

```
fold:     ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a
foldBack: ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b
```

# Example: Map Coloring (5)

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>;;
```

The function

```
colCntrs: Map -> Set<Country> -> Coloring
```

is based on repeated extension of colorings by countries using the `extColoring` function:

```
let colCntrs m cs =  ?
```

Remember

```
fold:     ('a -> 'b -> 'a) -> 'a -> Set<'b> -> 'a
foldBack: ('a -> 'b -> 'b) -> Set<'a> -> 'b -> 'b

extColoring: Map -> Coloring -> Country -> Coloring
```

# Example: Map Coloring (6)

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 $A'$: $m : A' \to B$.
The set $A'$ is called the *domain* of $m$: $\mathrm{dom}\, m = A'$.

A map $m$ can be described in a tabular form:

$$
\begin{array}{c|c}
a_0 & b_0 \\
\hline
a_1 & b_1 \\
\hline
 & \\
\multicolumn{2}{c}{\vdots} \\
 & \\
\hline
a_{n-1} & b_{n-1}
\end{array}
$$

- 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:

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

# Selected map operations in F#

- `ofList: ('a*'b) list -> Map<'a,'b>`
  `ofList` $[(a_0, b_0); \ldots; (a_{n-1}, b_{n-1})] = m$

- `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 $a \in \mathrm{dom}\ m$;
  otherwise an exception is raised

- `tryFind: 'a -> Map<'a,'b> -> 'b option`
  `tryFind` $a\ m =$ `Some` $(m(a))$, if $a \in \mathrm{dom}\ m$; `None` otherwise

- `foldBack:('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c`
  `foldBack` $f\ m\ c = f\ a_0\ b_0\ (f\ a_1\ b_1\ (f \ldots (f\ a_{n-1}\ b_{n-1}\ c) \cdots))$

# Immutability of `Map<'Key,'Value>`

```
let m = Map.ofList [(0,"a") ; (2 "c"); (3,"d")];;
val m : Map<int,string> =
        map [(0, "a"); (2, "c"); (3, "d")]

Map.add 1 "b" m;;
val it : Map<int,string> =
        map [(0, "a"); (1, "b"); (2, "c"); (3, "d")]

Map.tryFind 1 m;;
val it : string option = None
```

Evaluation of Map.add 1 ”b” m does not change the value of m.

# 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 f reg1 [];;
val it : (string * int) list =
    [("a1", 25); ("a2", 4); ("a3", 5)]
```

What is f?

Remember

```
foldBack:('a->'b->'c->'c) -> Map<'a,'b> -> 'c -> 'c
```

The higher-order Map 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)
```

# Summary

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