02157 Functional Programming

Tagged values and Higher-order list functions

Michael R. Hansen

DTU Informatics
Department of Informatics and Mathematical Modelling

Part I: Disjoint Sets – An Example

A *shape* is either a circle, a square, or a triangle

- the union of three disjoint sets

```
type shape =
    Circle of float
  | Square of float
  | Triangle of float*float*float;;
```

The *tags* `Circle`, `Square` and `Triangle` are *constructors*:

```
- Circle 2.0;;
> val it : shape = Circle 2.0

- Triangle(1.0, 2.0, 3.0);;
> val it : shape = Triangle(1.0, 2.0, 3.0)

- Square 4.0;;
> val it : shape = Square 4.0
```

Constructors in Patterns

A shape-area function is declared

```
let area = function
  | Circle r          -> System.Math.PI * r * r
  | Square a          -> a * a
  | Triangle(a,b,c) ->
      let s = (a + b + c)/2.0
      sqrt(s*(s-a)*(s-b)*(s-c)) ;;
> val area : shape -> real
```

following the structure of shapes.

- a constructor only matches itself

$$
\begin{array}{ll}
 & \texttt{area (Circle 1.2)} \\
\leadsto & (\texttt{System.Math.PI * r * r}, [\texttt{r} \mapsto \texttt{1.2}]) \\
\leadsto & \ldots
\end{array}
$$

Enumeration types – the months

Months are naturally defined using tagged values::

```
type month = January | February | March | April
             | May | June | July | August | September
             | October | November | December ;;
```

The days-in-a-month function is declared by

```
let daysOfMonth = function
  | February                             -> 28
  | April | June | September | November -> 30
  | _                                    -> 31 ;;
val daysOfMonth : month -> int
```

The option type

```
type 'a option = None | Some of 'a
```

Distinguishes the cases ”nothing” and ”something”.

predefined

The constructor Some and None are polymorphic:

```
Some false;;
val it : bool option = Some false

Some (1, "a");;
val it : (int * string) option = Some (1, "a")

None;;
val it : 'a option = None
```

Example

Find first position of element in a list:

```
let rec findPosI p x = function
   | y::_ when x=y -> Some p
   | _::ys          -> findPosI (p+1) x ys
   | []             -> None;;
val findPosI : int -> 'a -> 'a list -> int option when ...


let findPos x ys = findPosI 0 x ys;;
val findPos : 'a -> 'a list -> int option when ...
```

Examples

```
findPos 4 [2 .. 6];;
val it : int option = Some 2

findPos 7 [2 .. 6];;
val it : int option = None

 Option.get(findPos 4 [2 .. 6]);;
val it : int = 2
```

Part 2:Motivation

Higher-order functions are

- everywhere

  $\Sigma_{i=a}^{b} f(i),\ \frac{df}{dx},\ \{x \in A \mid P(x)\}, \ldots$

- powerful

  Parameterized modules succinct code ...

HIGHER-ORDER FUNCTIONS ARE USEFUL

now down to earth

- Many recursive declarations follows the same schema.

For example:

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

Succinct declarations achievable using higher-order functions

Contents

- Higher-order list functions (in the library)
  - map
  - exists, forall, filter, tryFind
  - foldBack, fold

Avoid (almost) identical code fragments by
parameterizing functions with functions

A simple declaration of a list function

A typical declaration following the structure of lists:

```
let rec posList = function
    | []    -> []
    | x::xs -> (x > 0)::posList xs;;
val posList : int list -> bool list

posList [4;  -5; 6];;
val it : bool list = [true; false; true]
```

Applies the function `fun x -> x > 0` to each element in a list

Another declaration with the same structure

```
let rec addElems = function
   | []        ->  []
   | (x,y)::zs -> (x+y)::addElems zs;;
val addElems : (int * int) list -> int list

addElems [(1,2) ;(3,4)];;
val it : int list = [3; 7]
```

Applies the addition function + to each pair of integers in a list

The function: `map`

Applies a function to each element in a list

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n] \;=\; [f(v_1); f(v_2); \ldots; f(v_n)]$$

Declaration

Library function

```
let rec map f = function
   | []    -> []
   | x::xs -> f x :: map f xs;;
val map : ('a -> 'b) -> 'a list -> 'b list
```

Succinct declarations can be achieved using map, e.g.

```
let posList = map (fun x -> x > 0);;
val posList : int list -> bool list

let addElems = map (fun (x,y) -> x+y);;
val addElems : (int * int) list -> int list
```

Exercise

Declare a function

$$g\ [x_1, \ldots, x_n] = [x_1^2 + 1, \ldots, x_n^2 + 1]$$

Remember

$$\texttt{map}\ f\ [v_1; v_2; \ldots; v_n] \ = \ [f(v_1); f(v_2); \ldots; f(v_n)]$$

Higher-order list functions: `exists`

Predicate: For some x in xs : $p(x)$.

$$\texttt{exists}\, p\; xs = \begin{cases} \text{true} & \text{if } p(x) = \texttt{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec exists p = function
  | []    -> false
  | x::xs -> p x || exists p xs;;
val exists : ('a -> bool) -> 'a list -> bool
```

Example

```
exists (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = true
```

Exercise

Declare `isMember` function using `exists`.

```
let isMember x ys = exists ????? ;;
val isMember : 'a -> 'a list -> bool when 'a : equality
```

Remember

$$\texttt{exists}\ p\ xs = \begin{cases} \text{true} & \text{if } p(x) = \texttt{true} \text{ for some } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `all`

Predicate: For every x in xs : $p(x)$.

$$\texttt{forall}\, p\; xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Declaration

Library function

```
let rec forall p = function
  | []    -> true
  | x::xs -> p x && forall p xs;;
val all : ('a -> bool) -> 'a list -> bool
```

Example

```
forall (fun x -> x>=2) [1; 3; 1; 4];;
val it : bool = false
```

Exercises

Declare a function

`disjoint` *xs ys*

which is true when there are no common elements in the lists *xs* and *ys*, and false otherwise.

Declare a function

`subset` *xs ys*

which is true when every element in the lists *xs* is in *ys*, and false otherwise.

Remember

$$\texttt{forall}\; p\; xs = \begin{cases} \text{true} & \text{if } p(x) = \text{true, for all elements } x \text{ in } xs \\ \text{false} & \text{otherwise} \end{cases}$$

Higher-order list functions: `filter`

Set comprehension: $\{x \in xs : p(x)\}$

`filter` p xs is the list of those elements x of xs where $p(x) =$ `true`.

Declaration

Library function

```
let rec filter p = function
  | []      -> []
  | x::xs -> if p x then x :: filter p xs
                 else filter p xs;;
val filter : ('a -> bool) -> 'a list -> 'a list
```

Example

```
filter System.Char.IsLetter ['1'; 'p'; 'F'; '-'];;
val it : char list = ['p'; 'F']
```

where `System.Char.IsLetter` c is true iff
$c \in \{'A', \ldots, 'Z'\} \cup \{'a', \ldots, 'z'\}$

Exercise

Declare a function

$$\texttt{inter}\ xs\ ys$$

which contains the common elements of the lists xs and ys — i.e. their intersection.

Remember:
$\texttt{filter}\ p\ xs$ is the list of those elements x of xs where $p(x) = \texttt{true}$.

Higher-order list functions: `tryFind`

$$\texttt{tryFind}\ p\ xs = \begin{cases} \texttt{Some}\ x & \text{for an element } x \text{ of } xs \text{ with } p(x) = \mathrm{true} \\ \texttt{None} & \text{if no such element exists} \end{cases}$$

```
let rec tryFind p = function
  | x::xs when p x -> Some x
  | _::xs          -> tryFind p xs
  | _              -> None ;;
val tryFind : ('a -> bool) -> 'a list -> 'a option
```

```
tryFind (fun x -> x>3) [1;5;-2;8];;
val it : int option = Some 5
```

Folding a function over a list (I)

Example: sum of norms of geometric vectors:

```
let norm(x1:float,y1:float) = sqrt(x1*x1+y1*y1);;
val norm : float * float -> float


let rec sumOfNorms = function
    | []    -> 0.0
    | v::vs -> norm v + sumOfNorms vs;;
val sumOfNorms : (float * float) list -> float


let vs = [(1.0,2.0); (2.0,1.0); (2.0, 5.5)];;
val vs : (float * float) list
       = [(1.0, 2.0); (2.0, 1.0); (2.0, 5.5)]

sumOfNorms vs;;
val it : float = 10.32448591
```

Folding a function over a list (II)

```
let rec sumOfNorms = function
    | []    -> 0.0
    | v::vs -> norm v + sumOfNorms vs;;
```

Let $f\ v\ s$ abbreviate $\texttt{norm}\ v + s$ in the evaluation:

$$\begin{array}{ll} & \texttt{sumOfNorms}\,[v_0; v_1; \ldots; v_{n-1}] \\ \rightsquigarrow & \texttt{norm}\, v_0 + (\texttt{sumOfNorms}\,[v_1; \ldots; v_{n-1}]) \\ = & f\ v_0\ (\texttt{sumOfNorms}\,[v_1; \ldots; v_{n-1}]) \\ \rightsquigarrow & f\ v_0\ (f\ v_1\ (\texttt{sumOfNorms}[v_2; \ldots; v_{n-1}])) \\ \vdots & \\ \rightsquigarrow & f\ v_0\ (f\ v_1\ (\cdots (f\ v_{n-1}\ 0.0) \cdots)) \end{array}$$

This repeated application of f is also called a folding of f.

Many functions follow such recursion and evaluation schemes

Higher-order list functions: foldBack (1)

Suppose that $\otimes$ is an infix function. Then

$$\texttt{foldBack}\ (\otimes)\ [a_0;\ a_1;\ \ldots\ ;\ a_{n-2};\ a_{n-1}]\ e_b = a_0 \otimes (a_1 \otimes (\ldots (a_{n-2} \otimes (a_{n-1} \otimes e_b)) \ldots))$$

$$\texttt{List.foldBack (+) [1; 2; 3] 0} = 1 + (2 + (3 + 0)) = 6$$
$$\texttt{List.foldBack (-) [1; 2; 3] 0} = 1 - (2 - (3 - 0)) = 2$$

Using the cons operator gives the append function @ on lists:

$$\texttt{foldBack (fun x rst -> x::rst)}\ [x_0;\ x_1;\ \ldots;\ x_{n-1}]\ \texttt{ys}$$
$$= x_0\texttt{::}(x_1\texttt{::}\ \ldots\ \texttt{::}(x_{n-1}\texttt{::ys})\ \ldots\ ))$$
$$= [x_0;\ x_1;\ \ldots;\ x_{n-1}]\ \texttt{@ ys}$$

so we get:

```
let (@) xs ys = List.foldBack (fun x rst -> x::rst) xs ys;;
val ( @ ) : 'a list -> 'a list -> 'a list

[1;2] @ [3;4] ;;
val it : int list = [1; 2; 3; 4]
```

Declaration of `foldBack`

```
let rec foldBack f xlst e =
  match xlst with
  | x::xs -> f x (foldBack f xs e)
  | []    -> e  ;;
val foldBack : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

```
let sumOfNorms vs = foldBack (fun v s -> norm v + s)  vs 0.0;;

let length xs = foldBack (fun _ n -> n+1) xs 0;;

let map f xs = foldBack (fun x rs -> f x :: rs) xs [];;
```

Exercise: union of sets

Let an insertion function be declared by

```
let insert x ys = if isMember x ys then ys else x::ys;;
```

Declare a union function on sets, where a set is represented by a list without duplicated elements.

Remember:

$$\texttt{foldBack}\,(\oplus)\,[x_1;x_2;\ldots;x_n]\;b \;\rightsquigarrow\; x_1 \oplus (x_2 \oplus \cdots \oplus (x_n \oplus b)\cdots)$$

Higher-order list functions: `fold` (1)

Suppose that $\oplus$ is an infix function.

Then the `fold` function has the definitions:

$$\texttt{fold}\ (\oplus)\ e_a\ [b_0;\ b_1;\ \ldots\ ;\ b_{n-2};\ b_{n-1}\ ]$$
$$= \ ((\ldots((e_a \oplus b_0) \oplus b_1)\ldots) \oplus b_{n-2}) \oplus b_{n-1}$$

i.e. it applies $\oplus$ from left to right.

Examples:

$$\texttt{List.fold (-) 0 [1; 2; 3]} = ((0-1)-2)-3 = -6$$
$$\texttt{List.foldBack (-) [1; 2; 3] 0} = 1-(2-(3-0)) = 2$$

Higher-order list functions: fold (2)

```
let rec fold f e = function
  | x::xs -> fold f (f e x) xs
  | []    -> e  ;;
val fold : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

Using cons in connection with fold gives the reverse function:

```
let rev xs  = fold (fun rs x -> x::rs) [] xs;;
```

This function has a linear execution time:

```
   rev [1;2;3]
⇝  fold  (fun ...)  []           [1;2;3]
⇝  fold  (fun ...)  (1::[])      [2;3]
⇝  fold  (fun ...)  [1]          [2;3]
⇝  fold  (fun ...)  (2::[1])     [3]
⇝  fold  (fun ...)  [2;1]        [3]
⇝  fold  (fun ...)  (3::[2;1])   []
⇝  fold  (fun ...)  [3;2;1]      []
⇝  [3;2;1]
```

Summary

- Many recursive declarations follows the same schema.

For example:

```
fun f []      =  ...
  | f(x::xs) =  ... f(xs) ...
```

Succinct declarations achievable using higher-order functions

Contents

- Higher-order list functions (in the library)
  - map
  - exists, forall, filter, tryFind
  - foldBack, fold

Avoid (almost) identical code fragments by
parameterizing functions with functions