02157 Functional Programming

Disjoint Unions and Higher-order list functions

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- Recap
- Disjoint union (or Tagged Values)
  - Groups different kinds of values into a single set.
- Higher-order functions on lists
  - many list functions follow “standard” schemes
  - avoid (almost) identical code fragments by parameterizing functions with functions
  - higher-order list functions based on natural concepts
  - succinct declarations achievable using higher-order functions

Precedence and associativity rules for expressions

Operator	Association	Precedence
`**`	Associates to the right	highest
`* / %`	Associates to the left	
`+ -`	Associates to the left	
`= <> > >= < <=`	No association	
`&&`	Associates to the left	
`||`	Associates to the left	lowest

- a monadic operator has higher precedence than any dyadic
- higher (larger) precedence means earlier evaluation
- function application associates to the left
- abstraction (`fun x -> e`) extends as far to the right as possible

For example:

- `- 2 - 5 * 7 > 3 - 1` means `((-2)-(5*7)) > (3-1)`
- `fact 2 - 4` means `(fact 2) - 4`
- $e_1\ e_2\ e_3\ e_4$ means $((e_1\ e_2)\ e_3)\ e_4$
- `fun x -> x 2` means ????

Precedence and associativity rules for types

- infix type operators: `*` and `->`
- suffix type operator: `list`

Association rules:

- `*` has NO association
- `->` associates to the right

Precedence rules:

- The suffix operator `list` has highest precedence
- `*` has higher precedence than `->`

For example:

- `int*int*int list` **means** `(int*int*(int list))`
- `int->int->int->int` **means** `int->(int->(int->int))`
- `'a*'b->'a*'b list` **means** `('a*'b)->('a*('b list))`

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                      (* 1 *)
  | Square of float                      (* 2 *)
  | Triangle of float*float*float;;      (* 3 *)
```

This declaration provides three rules for generating shapes:

- if r : `float`, then `Circle` r : `Shape` (* 1 *)
- if s : `float`, then `Square` s : `Shape` (* 2 *)
- if (a, b, c) : `float*float*float`,
  then `Triangle`(a, b, c) : `Shape` (* 3 *)

Such types are also called an algebraic data type:

$$
\begin{array}{lcl}
\texttt{Circle} & : & \texttt{float} \to \texttt{Shape} \\
\texttt{Square} & : & \texttt{float} \to \texttt{Shape} \\
\texttt{Triangle} & : & \texttt{float} * \texttt{float} * \texttt{float} \to \texttt{Shape}
\end{array}
$$

Part I: Disjoint Sets – An Example (II)

The tags `Circle`, `Square` and `Triangle` are called

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
```

`Circle`, `Square` and `Triangle` are used
to construct values of type `Shape`,

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 -> float
```

following the structure of shapes.

- a constructor only matches itself

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

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
```

Observe: Constructors need not have arguments

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
```

Observe: type variables are allowed in declarations of algebraic types

Example: Find first position of element in a list

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


let findPos x ys = findPosA 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
```

Exercise

A (teaching) room at DTU is either an auditorium or a databar:

- an auditorium is characterized by a location and a number of seats.
- a databar is characterized by a location, a number of computers and a number of seats.

Declare a type *Room*.

Declare a function:

$$\mathit{seatCapacity} : \mathit{Room} \rightarrow \mathit{int}$$

Declare a function

$$\mathit{computerCapacity} : \mathit{Room} \rightarrow \mathit{int\ option}$$

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
  - contains, 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

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

Remember

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

where

```
map: ('a -> 'b) -> 'a list -> 'b list
```

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 `contains` function using `exists`.

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

Remember

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

where

```
exists: ('a -> bool) -> 'a list -> bool
```

contains is a Library function

Higher-order list functions: `forall`

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 forall : ('a -> bool) -> 'a list -> bool
```

Example

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

Exercises

Declare a function

$$\texttt{disjoint}\ xs\ ys$$

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

Declare a function

$$\texttt{subset}\ xs\ ys$$

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

Remember

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

where

```
forall : ('a -> bool) -> 'a list -> bool
```

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
$c \in \{\text{'A'}, \ldots, \text{'Z'}\} \cup \{\text{'a'}, \ldots, \text{'z'}\}$

Exercise

Declare a function

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

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

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

```
filter: ('a -> bool) -> 'a list -> 'a list
```

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 absolute values:

```
let rec absSum = function
                 | []    -> 0
                 | x::xs -> abs x + absSum xs;;
val absSum : int list -> int

absSum [-2; 2; -1; 1; 0];;
val it : int = 6
```

Folding a function over a list (II)

```
let rec absSum = function
                 | []     -> 0
                 | x::xs -> abs x + absSum xs;;
```

Let $f\ x\ a$ abbreviate `abs` $x + a$ in the evaluation:

$$
\begin{array}{ll}
 & \mathtt{absSum}\ [x_0; x_1; \ldots; x_{n-1}] \\
\rightsquigarrow & \mathtt{abs}\ x_0 + (\mathtt{absSum}\ [x_1; \ldots; x_{n-1}]) \\
= & f\ x_0\ (\mathtt{absSum}\ [x_1; \ldots; x_{n-1}]) \\
\rightsquigarrow & f\ x_0\ (f\ x_1\ (\mathtt{absSum}[x_2; \ldots; x_{n-1}])) \\
\vdots & \\
\rightsquigarrow & f\ x_0\ (f\ x_1\ (\cdots (f\ x_{n-1}\ 0) \cdots))
\end{array}
$$

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:

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

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 absSum xs = foldBack (fun x a -> abs x + a)  xs 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 List.contains 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:

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

where

```
foldBack: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

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

Suppose that $\oplus$ is an infix function.

Then the `fold` function is defined by:

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

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

Examples:

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

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

Part I: Disjoint union (Algebraic data types)
– provides a mean to declare types containing different kinds of values

In Week 6 we extend the notion with recursive definition – provides a mean to declare types for finite trees
Part II: Higher-order list functions

- Many recursive declarations follows the same schema.

Succinct declarations achievable using higher-order functions

Contents

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

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