KSFUPRO1K1U – Functional Programming

Lecture 3: Records, tagged values and lists

Niels Hallenberg

These slides are based on original slides by Michael R. Hansen, DTU. Thanks!!!

The original slides has been used at a course in functional programming at DTU.

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;;
val it : float -> shape = <fun:clo@3>

- 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

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

How would you structure a program for this in C#?

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
```

LISTs

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
```

Lists

A list is a finite sequence of elements having the same type:

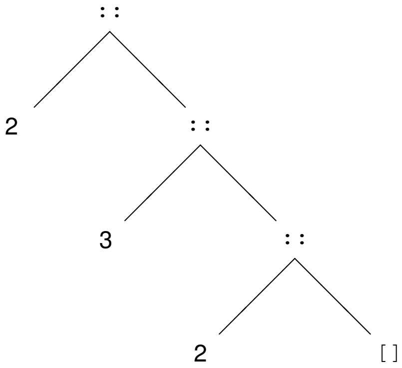
$[v_1; \ldots; v_n]$ ([ ] is called the empty list)

```
[2;3;6];;
val it : int list = [2; 3; 6]

["a"; "ab"; "abc"; ""];;
val it : string list = ["a"; "ab"; "abc"; ""]

[sin; cos];;
val it : (float->float) list = [<fun:...>; <fun:...>]

[(1,true); (3,true)];;
val it : (int * bool) list = [(1, true); (3, true)]

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

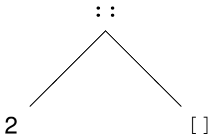
Trees for lists

A non-empty list $[x_1; x_2; \ldots; x_n]$, $n \geq 1$, consists of

- a *head* x_1 and
- a *tail* $[x_2; \ldots; x_n]$



Graph for `[2;3;2]`



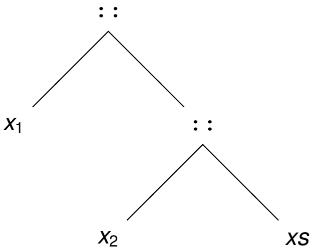
Graph for `[2]`

List constructors: `[]` and `::`

Lists are generated as follows:

- the empty list is a list, designated `[]`
- if x is an element and xs is a list,
  then so is $x :: xs$ (type consistency)

`::` associate to the right, i.e. $x_1{::}x_2{::}xs$ means $x_1{::}(x_2{::}xs)$



Graph for $x_1 :: x_2 :: xs$

Range expressions (1)

A simple range expression $[b\ ..\ e]$, where $e \geq b$, generates the list:

$$[b; b+1; b+2; \ldots; b+n]$$

where n is chosen such that $b+n \leq e < b+n+1$.

Example

```
[ -3 .. 5 ];;
val it : int list = [-3; -2; -1; 0; 1; 2; 3; 4; 5]


[2.4 .. 3.0 ** 1.7];;
val it : float list = [2.4; 3.4; 4.4; 5.4; 6.4]
```

Note that `3.0 ** 1.7 = 6.47300784`.

The range expression generates the empty list when $e < b$:

```
[7 .. 4];;
val it : int list = []
```

Range expressions (2)

The range expression $[b\ ..\ s\ ..\ e]$ generates either an ascending or a descending list:

$$[b\ ..\ s\ ..\ e] = \begin{cases} [b; b+s; b+2s; \ldots; b+ns] & \text{if } s > 0 \text{ and } b+ns \leq e < b+(n+1)s \\ [b; b-s; b-2s; \ldots; b-ns] & \text{if } s < 0 \text{ and } b-ns \geq e > b-(n+1)s \end{cases}$$

depending on the sign of s.

Examples:

```
[6 .. -1 .. 2];;
val it : int list = [6; 5; 4; 3; 2]
```

and the float representation of $0, \pi/2, \pi, \frac{3}{2}\pi, 2\pi$ is generated by:

```
[0.0 .. System.Math.PI/2.0 .. 2.0*System.Math.PI];;
val it : float list =
  [0.0; 1.570796327; 3.141592654; 4.71238898; 6.283185307]
```

Simple recursion on lists

We consider now three simple functions:

- append
- reverse
- isMember

whose declarations follow the structure of lists

```
let rec f ... xs ... =
    | []     -> v
    | x::xs -> .... f xs ...
```

using just two clauses.

Append

The infix operator `@` (called 'append') joins two lists:

$$[x_1; x_2; \ldots; x_m] \ @ \ [y_1; y_2; \ldots; y_n]$$
$$= [x_1; x_2; \ldots; x_m; y_1; y_2; \ldots; y_n]$$

Properties

$$\begin{aligned} [\,] \ @ \ ys &= ys \\ [x_1; x_2; \ldots; x_m] \ @ \ ys &= x_1 :: ([x_2; \ldots; x_m] \ @ \ ys) \end{aligned}$$

Declaration

```
let rec (@) xs ys =
  match xs with
  | []     -> ys
  | x::xs' -> x::(xs' @ ys);;
val (@) : 'a list -> 'a list -> 'a list
```

Append: evaluation

```
let rec (@) xs ys =
  match xs with
  | []      -> ys
  | x::xs' -> x::(xs' @ ys);;
```

Evaluation

$$
\begin{array}{lll}
 & \texttt{[1;2] @ [3;4]} & \\
\rightsquigarrow & \texttt{1::([2] @ [3;4])} & (\texttt{x} \mapsto 1,\ \texttt{xs}' \mapsto [2],\ \texttt{ys} \mapsto [3;4]) \\
\rightsquigarrow & \texttt{1::(2::([] @ [3;4]))} & (\texttt{x} \mapsto 2,\ \texttt{xs}' \mapsto [\,],\ \texttt{ys} \mapsto [3;4]) \\
\rightsquigarrow & \texttt{1::(2::[3;4])} & (\texttt{ys} \mapsto [3;4]) \\
\rightsquigarrow & \texttt{1::[2;3;4]} & \\
\rightsquigarrow & \texttt{[1;2;3;4]} &
\end{array}
$$

- Execution time is linear in the size of the first list

Append: polymorphic type

The answer from the system is:

```
> val (@) : 'a list -> 'a list -> 'a list
```

- `'a` is a *type variable*
- The type of `@` is *polymorphic* — it has many forms

`'a = int`: Appending integer lists

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

`'a = int list`: Appending lists of integer list

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

`@` is a built-in function

Reverse $\texttt{rev}\ [x_1; x_2; \ldots; x_n] = [x_n; \ldots; x_2; x_1]$

```
let rec naive_rev = function
 | []    -> []
 | x::xs -> naive_rev xs @ [x]
val naive_rev : 'a list -> 'a list
```

An evaluation:

```
     naive_rev[1;2;3]
⇝   naive_rev[2;3] @ [1]
⇝   (naive_rev[3] @ [2]) @ [1]
⇝   ((naive_rev[] @ [3]) @ [2]) @ [1]
⇝   (([] @ [3]) @ [2]) @ [1]
⇝   ([3] @ [2]) @ [1]
⇝   (3::([] @ [2])) @ [1]
⇝   (3::[2]) @ [1]
⇝   [3;2] @ [1]
⇝   3::([2] @ [1])
⇝   ...
⇝   [3;2;1]
```

Takes $O(n^2)$ time — Built-in version (`List.rev`) is efficient $O(n)$
We consider efficiency later.

Membership — equality types

$$
\begin{aligned}
& \texttt{isMember}\ x\ \texttt{[}y_1\texttt{;}y_2\texttt{;}\ \ldots\texttt{;}y_n\texttt{]} \\
= \quad & (x = y_1) \vee (x = y_2) \vee \cdots \vee (x = y_n) \\
= \quad & (x = y_1) \vee (\texttt{member}\ x\ \texttt{[}y_2\texttt{,}\ \ldots\texttt{,}y_n\texttt{]})
\end{aligned}
$$

Declaration

```
let rec isMember x = function
| []     -> false
| y::ys -> x=y || isMember x ys;;
val isMember : 'a -> 'a list -> bool when  'a : equality
```

- 'a is an equality type variable   no function types
- `isMember (1,true) [(2,true); (1,false)]` ⇝ false
- `isMember [1;2;3]  [[1]; []; [1;2;3]]` ⇝ true

Example: sumProd

$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}]$
$= \ (x_0 + x_1 + \ldots + x_{n-1}\ ,\ x_0 * x_1 * \ldots * x_{n-1})$

The declaration is based on the recursion formula:

$\texttt{sumProd}\ [x_0; x_1; \ldots; x_{n-1}] = (x_0 + \texttt{rSum}, x_0 * \texttt{rProd})$
where $(\texttt{rSum}, \texttt{rProd}) = \texttt{sumProd}\ [x_1; \ldots; x_{n-1}]$

This gives the declaration

```
let rec sumProd = function
    | []    -> (0,1)
    | x::rest ->
            let (rSum,rProd) = sumProd rest
            (x+rSum,x*rProd);;
val sumProd : int list -> int * int

sumProd [2;5];;
val it : int * int = (7, 10)
```

Example: `split`

Declare an F# function `split` such that:

$$\texttt{split}\ [x_0;x_1;x_2;x_3;\ldots;x_{n-1}] = ([x_0;x_2;\ldots],[x_1;x_3;\ldots])$$

The declaration is

```
let rec split = function
    | []         -> ([],[])
    | [x]        -> ([x],[])
    | x::y::xs -> let (xs1,xs2) = split xs
                  (x::xs1,y::xs2);;
```

Notice

- a convenient division into three cases, and
- the recursion formula

$$\texttt{split}\ [x_0;x_1;x_2;\ldots;x_{n-1}] = (x_0 :: \texttt{xs1}, x_1 :: \texttt{xs2})$$

where $(\texttt{xs1},\texttt{xs2}) = \texttt{split}\ [x_2;\ldots;x_{n-1}]$

An exercise

From list of pairs to pair of lists:

$$\texttt{unzip}\ [(x_1, y_1); (x_2, y_2); \ldots; (x_n, y_n)] \\ = ([x_1; x_2; \ldots; x_n], [y_1; y_2; \ldots; y_n])$$

Many functions on lists are predefined, e.g. `@`, `List.length`, `List.rev`, `List.zip` and many more.

Match on results of recursive call

We consider declarations on the form:

let rec f ... xs ... =
    ....
    let $pat(\overline{y})$ = f xs
    $e(\overline{y})$

Recall unzip and split from above.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

Goal and approach

Goal: the main concepts of the problem formulation are traceable in the program.

Approach: to name the important concepts of the problem and associate types with the names.

- This model should facilitate discussions about whether it fits the problem formulation.

Aim: A succinct, elegant program reflecting the model.

The problem

An electronic cash register contains a data register associating the name of the article and its price to each valid article code. A purchase comprises a sequence of items, where each item describes the purchase of one or several pieces of a specific article.

The task is to construct a program which makes a bill of a purchase. For each item the bill must contain the name of the article, the number of pieces, and the total price, and the bill must also contain the grand total of the entire purchase.

A Functional Model

- Name key concepts and give them a type

A signature for the cash register:

```
type articleCode = string
type articleName = string
type price       = int
type register    = (articleCode * (articleName*price)) list
type noPieces    = int
type item        = noPieces * articleCode
type purchase    = item list
type info        = noPieces * articleName * price
type infoseq     = info list
type bill        = infoseq * price

exception FindArticle
makeBill: register -> purchase -> bill
```

Example

The following declaration names a register:

```
let reg = [("a1",("cheese",25));
           ("a2",("herring",4));
           ("a3",("soft drink",5)) ];;
```

The following declaration names a purchase:

```
let pur = [(3,"a2"); (1,"a1")];;
```

A bill is computed as follows:

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

Functional decomposition (1)

Type: findArticle: articleCode → register → articleName * price

```
exception FindArticle;;

let  rec findArticle ac = function
   | (ac',adesc)::reg ->  if ac=ac' then adesc
                          else findArticle ac reg
   | _                -> raise FindArticle;;
```

The specified type is an instance of the inferred type:

```
val findArticle : 'a -> ('a * 'b) list -> 'b
                                   when 'a : equality
```

An article description is found as follows:

```
findArticle "a2" reg;;
val it : string * int = ("herring", 4)
```

Functional decomposition (2)

Type: makeBill: register → purchase → bill

```
let rec makeBill reg = function
    | []            -> ([],0)
    | (np,ac)::pur ->
        let (aname,aprice) = findArticle ac reg
        let tprice         = np*aprice
        let (billtl,sumtl) = makeBill reg pur
        ((np,aname,tprice)::billtl, tprice+sumtl);;
```

The specified type is an instance of the inferred type:

```
val makeBill :
  ('a * ('b * int)) list -> (int * 'a) list
                -> (int * 'b * int) list * int
    when 'a : equality

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

Patterns with guards: Three versions of `findArticle`

The if-then-else expression in

```
let  rec findArticle ac = function
   | (ac',adesc)::reg ->  if ac=ac' then adesc
                          else findArticle ac reg
   | _                -> raise FindArticle;;
```

may be avoided using clauses with guards:

```
let  rec findArticle ac = function
   | (ac',adesc)::reg when ac=ac' -> adesc
   | (ac',adesc)::reg             -> findArticle ac reg
   | _                            -> raise FindArticle;;
```

This may be simplified using wildcards:

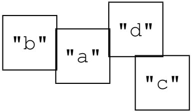
```
let  rec findArticle ac = function
   | (ac',adesc)::_ when ac=ac' -> adesc
   | _::reg                     -> findArticle ac reg
   | _                          -> raise FindArticle;;
```

Summary

- A succinct model is achieved using type declarations.
- Easy to check whether it fits the problem.
- Conscious choice of variables (on the basis of the model) increases readability of the program.
- Standard recursions over lists solve the problem.

Example: Map Coloring.

A map should be colored so that neighbouring countries get different colors



The types for country and map are “straightforward”:

- `type country = string`

  Symbols: c, c1, c2, c’; Examples: ”a”, ”b”, . . .

- `type map=(country*country) list`

  Symbols: m; Example: val exMap = [(”a”,”b”); (”c”,”d”); (”d”,”a”)]

How many ways could above map be colored?

Abstract models for color and coloring

- `type color = country list`

  Symbols: col; Example: ["c"; "a"]

- `type coloring = color list`

  Symbols: cols; Example: [["c"; "a"]; ["b"; "d"]]

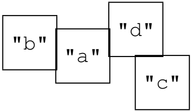
Be conscious about symbols and examples

```
colMap:  map -> coloring
```

Meta symbol: Type		Definition	Sample value
c:	country	string	"a"
m:	map	(country*country) list	[("a","b");("c","d");("d","a")]
col:	color	country list	["a";"c"]
cols:	coloring	color list	[["a";"c"];["b";"d"]]

Figure: *A Data model for map coloring problem*

Algorithmic idea



Insert repeatedly countries in a coloring.

	country	old coloring	new coloring
1.	"a"	[]	[["a"]]
2.	"b"	[["a"]]	[["a"] ; ["b"]]
3.	"c"	[["a"] ; ["b"]]	[["a";"c"] ; ["b"]]
4.	"d"	[["a";"c"] ; ["b"]]	[["a";"c"] ; ["b";"d"]]

Figure: Algorithmic idea

Functional decomposition (I)

To make things easy

Are two countries neighbours?

areNb: map $\rightarrow$ country $\rightarrow$ country $\rightarrow$ bool

```
let areNb m c1 c2 = isMember (c1,c2) m || isMember (c2,c1) m;;
```

Can a color be extended?

canBeExtBy: map $\rightarrow$ color $\rightarrow$ country $\rightarrow$ bool

```
let rec canBeExtBy m col c =
  match  col with
  | []        -> true
  | c'::col' -> not (areNb m c' c) && canBeExtBy m col' c;;

canBeExtBy exMap ["c"] "a";;
val it : bool = true

canBeExtBy exMap ["a"; "c"] "b";;
val it : bool = false
```

Functional composition (I)

Combining functions make things easy

Extend a coloring by a country:

extColoring: map $\rightarrow$ coloring $\rightarrow$ country $\rightarrow$ coloring

```
Examples:
  extColoring exMap [] "a"          =  [["a"]]
  extColoring exMap [["b"]] "a"     =  [["b"] ; ["a"]]
  extColoring exMap [["c"]] "a"     =  [["a"; "c"]]
```

```
let rec extColoring m cols c =
    match cols with
    | []          -> [[c]]
    | col::cols'  -> if canBeExtBy m col c
                     then (c::col)::cols'
                     else col::extColoring m cols' c;;
```

Function types, consistent use of symbols, and examples make program easy to comprehend

Functional decomposition (II)

To color a neighbour relation:

- Get a list of countries from the neighbour relation.
- Color these countries

Get a list of countries without duplicates:

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

let rec countries = function
    | []            -> []
    | (c1,c2)::m -> addElem c1 (addElem c2 (countries m));;
```

Color a country list:

```
let rec colCntrs m = function
    | []     -> []
    | c::cs -> extColoring m (colCntrs m cs) c;;
```

Functional composition (III)

The problem can now be solved by
combining well-understood pieces

Create a coloring from a neighbour relation:

colMap: map $\rightarrow$ coloring

```
let colMap m = colCntrs m (countries m);;

colMap exMap;;
val it : string list list = [["c"; "a"]; ["b"; "d"]]
```

On modelling and problem solving

- Types are useful in the specification of concepts and operations.
- Conscious and consistent use of symbols enhances readability.
- Examples may help understanding the problem and its solution.
- Functional paradigm is powerful.

    Problem solving by combination of well-understood pieces

These points are not programming language specific