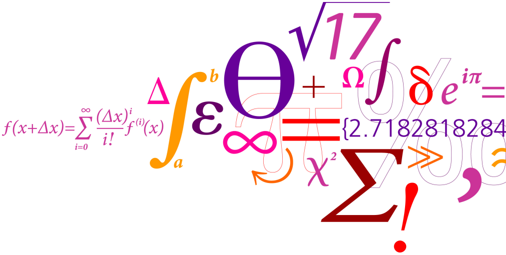
02157 Functional Programming

Lecture 3: Programming as a model-based activity

Michael R. Hansen



DTU Compute
Department of Applied Mathematics and Computer Science

Overview

- RECAP
  - Higher-order functions (lists)
  - Type inference
- Syntax, semantics and pragmatics (briefly)
  - Value polymorphism restriction — relates to syntax and semantics
  - Simplification of programs — relates to pragmatics
- Programming as a modelling activity — relates to pragmatics
  - Type declarations (type abbreviations)
  - Cash register
  - Map coloring

Exploiting functional decomposition

Example: takeWhile

A predicate is a truth-valued function.

- $\texttt{takeWhile} : (\texttt{'a} \to \texttt{bool}) \to \texttt{'a list} \to \texttt{'a list}$, where

$$\texttt{takeWhile}\ p\ [x_1; x_2; \ldots; x_n] = [x_1; x_2; \ldots; x_k]$$

where $p\ x_i = \mathrm{true}, 1 \leq i \leq k$ and ($k = n$ or $k < n$ and $p\ x_{k+1} = \mathrm{false}$).

```
let rec takeWhile p =
   function
   | x::xs when p x -> x::takeWhile p xs
   | _              -> []

let xs = [3; 5; 1; 6];;

takeWhile (fun x -> x>1) xs;;
val it : int list = [3; 5]

takeWhile  (fun x -> x%2=1) xs;;
val it : int list = [3; 5; 1]
```

A higher-order list function that takes a predicate as argument

Programming languages: Syntax, Semantics and Pragmatics

- The syntax is concerned with the (notationally correct) grammatical structure of programs.
- The semantics is concerned with the meaning of syntactically correct programs.
- The pragmatics is concerned with practical (adequate) application of language constructs in order to achieve certain objectives.

Syntactical constructs in F#

- Constants: 0, 1.1, true, ...
- Patterns:
  $x \quad \_ \quad (p_1, \ldots, p_n) \quad p_1 :: p_2 \quad p_1 | p_2 \quad p \ \texttt{when}\ e \quad p \ \texttt{as}\ x \quad p : t \ldots$
- Expressions:
  $x \quad (e_1, \ldots, e_n) \quad e_1 :: e_2 \quad e_1 e_2 \quad e_1 \oplus e_2 \quad \texttt{let}\ p_1 = e_1\ \texttt{in}\ e_2 \quad e : t$

  $\texttt{match}\ e\ \texttt{with}\ \textit{clauses} \quad \texttt{fun}\ p_1 \cdots p_n \texttt{->} e \quad \texttt{function}\ \textit{clauses} \quad \ldots$
- Declarations $\texttt{let}\ f\ p_1 \ldots p_n = e \quad \texttt{let rec}\ f\ p_1 \ldots p_n = e$, $n \geq 0$
- Types
  $\texttt{int} \quad \texttt{bool} \quad \texttt{string} \quad {'a} \quad t_1 * t_2 * \cdots * t_n \quad t\ \texttt{list} \quad t_1 \texttt{->} t_2 \ldots$

where the construct *clauses* has the form:

$\texttt{|}\ \texttt{p}_1\ \texttt{->}\ \texttt{e}_1\ \texttt{|}\ \ldots\ \texttt{|}\ \texttt{p}_n\ \texttt{->}\ \texttt{e}_n$

In addition to that

- precedence and associativity rules, parenthesis around p and e and type correctness

Semantics of F# programs

The semantics can be described by rules: $\dfrac{\text{List of hypothesis}}{\text{Conclusion}}$

Conclusion and hypotheses are *judgements*: $env \vdash e => v$

- "within environment env, evaluation of expression e terminates and gives value v".

We show a few rules from semantics of micro-ML: Sestoft2012

$$\frac{env(x) = v}{env \vdash x => v} \qquad \frac{env \vdash e_1 => v_1 \quad env \vdash e_2 => v_2 \quad v_1 + v_2 = v}{env \vdash e_1 + e_2 \;=>\; v}$$

Let-expression:

$$\frac{env \vdash e_r => v_r \quad env[x \mapsto v_r] \vdash e_b => v_b}{env \vdash \texttt{let}\ x = e_r\ \texttt{in}\ e_b \;=>\; v_b}$$

If-then-else:

$$\frac{env \vdash e => \mathit{false} \quad env \vdash e_2 => v_2}{env \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \;=>\; v_2} \text{ and } \frac{env \vdash e => \mathit{true} \quad env \vdash e_1 => v_1}{env \vdash \texttt{if}\ e\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \;=>\; v_1}$$

Semantics: Functions and function application

What is the value of the following expression?

```
let a = 1
in let f = fun x -> x+a
   in let a = 2
      in f 3;;
```

The semantics of a function, e.g. `fun` $x -> e$ in environment env_{dec}:

(x, e, env_{dec}) is called a closure

.

Application of (non-recursive) functions:

$$\frac{env \vdash e_1 => (x, e, env_{dec}) \quad env \vdash e_2 => v_2 \quad env_{dec}[x \mapsto v_2] \vdash e => v}{env \vdash e_1 e_2 => v}$$

- the closure for f is $(x, x + a, [a \mapsto 1])$
- f 3 is evaluated in environment $[a \mapsto 2, f \mapsto (x, x + a, [a \mapsto 1])]$
- x+a is evaluated in the environment $[a \mapsto 1, x \mapsto 3]$
- the value of the expression is 4

What if f 3 is changed to f a?

Semantics: observations

- The environment in which a function is declared is a part of its closure F# has static binding
  – the fresh binding for a does not influence the meaning of f

- Just a slight extension is required in order to cope with recursive functions.

- The informal treatment of step-by-step evaluations in this course ($e_1 \rightsquigarrow e_2$) is based on a formal semantics for functional programming languages

- The pure functional programming part of F# consists of a small number of constructs with simple well-defined semantics.

- Syntax and semantics of programming languages is a part of 02141 Computer Science Modelling

Examples on errors with polymorphic functions (1)

```
let empty = [] @ [];;
stdin(3,5): error FS0030: Value restriction.
The value 'empty' has been inferred to have generic type
val empty : '_a list

Either define 'empty' as a simple data term,
make it a function with explicit arguments or,
if you do not intend for it to be generic,
   add a type annotation.

let empty = [];;  // WORKS!
val empty : 'a list
```

Examples on errors with polymorphic functions (2)

```
List.rev [];;
stdin(5,1): error FS0030: Value restriction.
The value 'it' has been inferred to have generic type
val it : '_a list

Either define 'it' as a simple data term,
make it a function with explicit arguments or,
if you do not intend for it to be generic,
   add a type annotation.
```

```
List.rev ([]: int list);;  // WORKS!
val it : int list = []
```

- a top-level expression is considered a top-level declaration of `it`:

```
let it = List.rev ([]: int list);;
val it : int list = []
```

Examples on errors with polymorphic functions (3)

```
let takeAll = takeWhile (fun _ -> true);;
stdin(15,5): error FS0030: Value restriction
The value 'takeAll' has been inferred to have generic type
    val takeAll : ('_a list -> '_a list)

Either make the arguments to 'takeAll' explicit or
if you do not intend for it to be generic,
   add a type annotation.
```

```
let takeAll xs = takeWhile (fun _ -> true) xs;;  // WORKS!
val takeAll : xs:'a list -> 'a list
```

- What is the problem solved by this restriction?
- What is the solution to this problem?

An ”imperative” issue with Polymorphism

- A type issue to be addressed in the imperative fragment of F#.

The following hypothetical code does NOT compile:

```
let mutable a = []                // let a = ref []

let f x = a <- x::a
 val f: 'a -> unit

f 1;;

f "ab";;

a;;
val it :  ???? list = ["ab";1] // a BIG type problem
```

A solution: Value Restriction on Polymorphism

A value expression is an expression that is not reduced further by an evaluation, for example:

```
[]        Some []      fun xs -> takeWhile (fun _ -> true) xs
```

The following are not value expressions:

```
[]@[]        List.rev []        List.map (fun _ -> true)
```

as they can be further evaluated.

Every top-level declarations `let` *id* $=$ *e* is subject to the following restriction on *e*:

- All monomorphic expressions are OK,
- all value expressions are OK, even polymorphic ones, and
- at top-level, polymorphic non-value expressions are forbidden

An expression `ref` *e* in a declaration of the form `let` *id* $=$ `ref` *e* (or equivalently `let mutable` *id* $=$ *e*) is NOT considered a value expression (even when *e* is a constant)

Examples revisited

A polymorphic value expression:

```
let empty = [];;   // WORKS!
val empty : 'a list
```

A monomorphic expression, that is not a value expression:

```
List.rev ([]: int list);;   // WORKS!
val it : int list = []
```

A declaration of a function with an explicit parameter

```
let f xs = takeWhile (fun _ -> true) xs;;   // WORKS!
val f : xs:'a list -> 'a list
```

is an abbreviation for

```
let f = fun xs -> takeWhile (fun _ -> true) xs;;
```

and a closure is a value expression

On “simplifying” programs (1)

Programs with the following flavour are too often observed:

```
let f(x) = match (x) with
           |(a,z) -> if (not(a) = true) then true
                     else if (fst(z) = true) then snd(z)
                          else false;;
```

- What is the type of *f*?
- What is *f* computing?

Some rules of thump:

- Avoid obvious superfluous use of parenthesis
- Simplify Boolean expressions
- A match with just one clause can be avoided
- Use of `fst` and `snd` is avoided using patterns
- `if-then-else` expressions having truth values are avoided using Boolean expressions

On “simplifying” programs (2)

```
let f(x) = match (x) with
           |(a,z) -> if (not(a) = true) then true
                     else if (fst(z) = true) then snd(z)
                          else false;;
```

Avoid obvious superfluous use of parenthesis:

```
let f x = match x with
          |(a,z) -> if not a = true then true
                    else if fst z = true then snd z
                         else false;;
```

Simplify Boolean expressions:

```
let f x = match x with
          |(a,z) -> if not a then true
                    else if fst z then snd z
                         else false;;
```

- `e = true` has the same truth value as `e`

On “simplifying” programs (3)

```
let f x = match x with
          |(a,z) -> if not a then true
                    else if fst z then snd z
                         else false;;
```

Avoid a match with just one clause:

```
let f(a,z) = if not a then true
             else if fst z then snd z else false;;
```

Avoid `fst` and `snd`:

```
let f(a,(b,c)) = if not a then true
                 else if b then c else false;;
```

On “simplifying” programs (4)

- `if-then-else` expressions having truth values are avoided using Boolean expressions

```
let f(a,(b,c)) = if not a then true
                 else if b then c else false;;
```

`if` *p* `then` *q* `else false` is the same as *p* `&&` *q*:

```
let f(a,(b,c)) = if not a then true else b && c;;
```

`if` *p* `then true else` *q* is the same as *p* `||` *q*:

```
let f(a,(b,c)) = not a || b && c;;
```

On "simplifying" programs (5)

```
let f(a,(b,c)) = not a || b && c;;
```

- The type of *f* is `bool*(bool*bool) -> bool`
- $f(a,(b,c))$ is the value of the proposition "*a* implies (*b* and *c*)"

Introducing implication:

```
let (.=>) p q = not p || q;;

let f(a,(b,c)) = a .=> (b && c);;
```

Type declarations (abbreviations)

A declarations of a monomorphic type has the form:

$$\texttt{type}\ T = t$$

where t is a type expression not containing type variables.

A declaration of a polymorphic type has the form:

$$\texttt{type}\ T <'v_0,'v_1,\ldots,'v_n> = t$$

where t is a type expression containing type variables $'v_0, 'v_1, \ldots, 'v_n$.

Type constraints may be added to the type-parameter list.

Type declarations (abbreviations): Examples

```
type Name = string
type Id = int
type Assoc<'K, 'V  when 'K : equality> = ('K * 'V) list

type Participants = Assoc<Id,Name>
let ex:Participants = [(164255, "Bill") ; (173333,"Eve")]
val ex : Participants = [(164255, "Bill"); (173333, "Eve")]

ex =  ([(0,"")]: (int*string) list) ;;
val it : bool = false

let rec insert k v (ass:Assoc<'K,'V>) = (k,v)::ass;;
val insert : 'K -> 'V -> Assoc<'K,'V>
                 -> ('K * 'V) list when 'K : equality

insert 1 "a" [(0,"")];;
val it : (int * string) list = [(1, "a"); (0, "")]
```

- The declared types work as abbreviations

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

makeBill: Register -> Purchase -> Bill
```

Is the model adequate?

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

```
let rec findArticle ac = function
    | (ac',adesc)::_ when ac=ac' -> adesc
    | _::reg                     -> findArticle ac reg
    | _                          ->
          failwith(ac + " is an unknown article code");;
val findArticle : string -> (string * 'a) list -> 'a
```

Note that the specified type is an instance of the inferred type.

An article description is found as follows:

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

findArticle "a5" reg;;
System.Exception: a5 is an unknown article code
   at FSI_0016.findArticle[a] ...
```

Note: failwith is a built-in function that raises an exception

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 :
    (string * ('a * int)) list -> (int * string) list
                -> (int * 'a * int) list * int
```

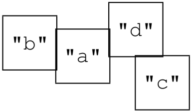
```
makeBill reg pur;;
val it : (int * string * int) list * int =
  ([(3, "herring", 12); (1, "cheese", 25)], 37)
```

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.

Color a map 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”]]

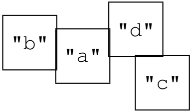
Be conscious about symbols and examples

```
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