02157 Functional Programming

Lecture 2: Functions, Types and Lists

Michael R. Hansen

DTU Compute
Department of Applied Mathematics and Computer Science

Outline

- Functions as ”first-class citizens”
- Types, polymorphism and type inference
  - type constraints (equality and comparison)
  - type declarations
- Lists
- Selected language constructs

Goal: By the end of the day you are acquainted with a major part of the F# language.

Functions as ”first-class citizens”

- functions can be passed as arguments to functions
- functions can be returned as values of functions

like any other kind of value.

There is nothing special about functions in functional languages

A function that takes a function as argument or produces a function as result is also called a higher-order function.

Higher-order functions are useful

- succinct code
- highly parametrized programs
- Program libraries typically contain many such functions

Anonymous functions

Function expressions with general patterns, e.g.

```
function
 | 2          -> 28   // February
 |4|6|9|11    -> 30   // April, June, September, November
 | _          -> 31   // All other months
;;
```

Simple function expressions, e.g.

```
fun r -> System.Math.PI * r * r ;;
val it : float -> float = <fun:clo@10-1>

it 2.0 ;;
val it : float = 12.56637061
```

Anonymous functions

Simple functions expressions with *currying*

$$\texttt{fun}\; x\; y\; \cdots\; z \;\rightarrow e$$

with the same meaning as

$$\texttt{fun}\; x \rightarrow (\texttt{fun}\; y \rightarrow (\cdots\; (\texttt{fun}\; z \rightarrow e) \cdots))$$

For example: The function below takes an integer as argument and returns a function of type `int -> int` as value:

```
fun x y -> x + x*y;;
val it : int ->  int -> int = <fun:clo@2-1>

let f = it 2;;
val f : ( int -> int)

f 3;;
val it : int = 8
```

Functions are first class citizens:
the argument and the value of a function may be functions

Function declarations

A simple function declaration:

$$\texttt{let}\ f\ x\ =\ e \qquad \text{means} \qquad \texttt{let}\ f\ =\ \texttt{fun}\ x\ \rightarrow e$$

A declaration of a curried function

$$\texttt{let}\ f\ x\ y\ \cdots z =\ e$$

has the same meaning as:

$$\texttt{let}\ f\ =\ \texttt{fun}\ x \rightarrow (\texttt{fun}\ y \rightarrow (\cdots\ (\texttt{fun}\ z \rightarrow e)\cdots))$$

For example:

```
let addMult x y = x + x*y;;
val addMult : int -> int -> int

let f = addMult 2;;
val f : (int -> int)

f 3;;
val it : int = 8
```

An example

Suppose that we have a cube with side length s, containing a liquid with density ρ. The weight of the liquid is then given by $\rho \cdot s^3$:

```
let weight ro s = ro * s ** 3.0;;
val weight : float -> float -> float
```

We can make *partial evaluations* to define functions for computing the weight of a cube of either water or methanol:

```
let waterWeight = weight 1000.0;;
val waterWeight : (float -> float)

waterWeight 2.0;;
val it : float = 8000.0

let methanolWeight = weight 786.5 ;;
val methanolWeight : (float -> float)

methanolWeight 2.0;;
val it : float = 6292.0
```

The formula $\rho \cdot s^3$ is represented just once in the program

Infix functions

The prefix version $(\oplus)$ of an infix operator $\oplus$ is a curried function.

For example:

```
(+);;
val it : (int -> int -> int) = <fun:it@1>
```

Arguments can be supplied one by one:

```
let plusThree = (+) 3;;
val plusThree : (int -> int)

plusThree 5;;
val it : int = 8
```

Function composition: $(f \circ g)(x) = f(g(x))$

For example, if $f(y) = y + 3$ and $g(x) = x^2$, then $(f \circ\ g)(z) = z^2 + 3$.

The infix operator << in F# denotes function composition:

```
let f y = y+3;;

let g x = x*x;;

let h = f << g;;        // h = (f o g)
val h : int -> int

h 4;;                   // h(4) = (f o g)(4)
val it : int = 19
```

Type of (<<) ?

Types and type checking

Purposes:

- Modelling, readability: types are used to indicate the intention behind a program
- Safety, efficiency: ”Well-typed programs do not go wrong”
  Robin Milner
  - Catch errors at compile time
  - Verification of type properties is not needed at runtime

A type checker is an algorithm used at an early phase in the compiler to check whether a program contains type errors.

Fundamental type-checking problem

All non-trivial semantic properties of programs are undecidable

Rice's theorem

Examples: p terminates on all its input

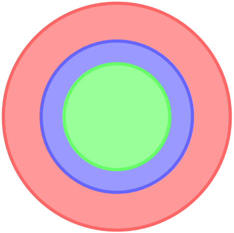
Cannot be checked for programs p belonging to a Turing-powerful language

Consequence: A type-checking algorithm provides an approximation:

ill-typed, bad programs

ill-typed, good programs

well-typed, good programs



Type inference

The type system of F# allows for polymorphic types, that is, types with many forms. Polymorphic types are expressed using type variables 'a, 'b, 'c, ...

The *most general type* or *principal type* is inferred by the system.

Examples:

```
let id x = x
val id : 'a -> 'a

let pair x y = (x,y)
val pair : 'a -> 'b -> 'a * 'b
```

The inferred types are most general in the sense that all other types for id and pair are instances of the inferred types.

applications of the functions?

By the type of a function, we (usually) mean the most general type

Remark: identity function id is a built-in function

A simple list recursion

List.append is a function from the `List` library:

- `List.append` $[x_0; \ldots; x_{n-1}]\,[y_0; \ldots; x_{m-1}] =$
  $[x_0; \ldots; x_{n-1}; y_0; \ldots; x_{m-1}]$

There is a convenient infix notation for `List.append` *xs ys* in F#:

xs `@` *ys*

The declaration of `(@)` *xs ys* follows the structure of *xs*:

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

[["a"]; ["ab";"abc"; ""]; []] @ [["x"]; ["xy"; "xyz"]];;
val it : string list list =
    [["a"]; ["ab"; "abc"; ""]; []; ["x"]; ["xy"; "xyz"]]
```

Polymorphic type inference – informally

Given a declaration, for example,

```
let rec (@) xs ys =
   match xs with
   | []        -> ys                    (* C 1 *)
   | x::xtail -> x::(xtail @ ys);;      (* C 2 *)
```

- Guess types for the arguments of `(@)`: *xs* :' *u* and *ys* :' *v*
- Add type constraints based on the body of the declaration:
  1. `[]`: 'a list and 'u = 'a list, where 'a is a fresh type variable — C 1
  2. x: 'a, xtail: 'a list, (xtail `@` ys): 'a list, x::(xtail `@` ys): 'a list exploiting the type og `::` — C 2
  3. ys : 'a list, 'v = 'a list ys must have the same type as x::(xtail `@` ys) — C 1,2

Every sub-expression is now consistently typed.

The most general type or principle type of (`@`) is:

```
'a list -> 'a list -> 'a list
```

- First inference algorithm for ML DamasMilner82
- A nice introduction and F# implementation: Sestoft12

Basic Types: equality and comparison

Equality and comparison are defined for the basic types of F#, including integers, floats, booleans, characters and strings.

Examples:

```
true < false;;
val it : bool = false

'a' < 'A';;
val it : bool = false

"a" < "ab";;
val it : bool = true
```

Composite Types: equality and comparison

Equality and comparison carry over to composite types
as long as function types are not involved:

Equality is defined structurally on values with the same type:

```
(1, true, 7.4) = (2-1, not false, 8.0 - 0.6);;
val it : bool = true

[[1;2]; [3;4;5]] = [[1..2]; [3..5]];;
val it : bool = true
```

Comparison is typically defined using lexicographical ordering:

```
[1; 2; 3] < [1; 4];;
val it : bool = true

(2, [1; 2; 3]) > (2, [1;4]);;
val it : bool = false
```

Polymorphic types: equality and comparison constraints (I)

Polymorphic types may be accompanied with equality and comparison constraints like:

- when 'a : comparison
- when 'b : equality

For example, there is a built-in function:

$$\texttt{compare}\; x\; y \;=\; \begin{cases} > 0 & \text{if } x > y \\ 0 & \text{if } x = y \\ < 0 & \text{if } x < y \end{cases}$$

with the type:

```
'a -> 'a -> int  when 'a : comparison
```

For example:

```
compare (2, [1; 2; 3]) (2, [1;4]);;
val it : int = -1
```

Polymorphic types: equality and comparison constraints (II)

The built-in function List.contains can be declared as follows:

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

contains [3;4] [[1..2]; [3..5]];;
val it : bool = false
```

Notice:

- The equality constraint in the type
- Lazy (short-circuit) evaluation of $e_1||e_2$ causes termination as soon as an element y equal to x is found
- Yet an evaluation following the structure of lists

Let-expressions

A let-expression e_l has the (verbose) form

```
let x = e1 in e2
```

or the following short form exploiting indentation:

```
let x = e1
e2
```

The expression provides a local definition for x in e2.

A let-expression e_l is evaluated in an environment env as follows:

If

1. $v1$ is the value obtained by evaluating $e1$ in env,
2. env' is the environment obtained by adding the binding $x \mapsto v$ to env and
3. $v2$ is the value obtained by evaluating $e2$ in env'

then

$$(\texttt{let}\ x = e1\ \texttt{in}\ e2, env) \rightsquigarrow (v2, env)$$

Let-expression – an example

Examples

```
let g x = let a = 6
          let b = x + a
          x + b;;
val g : int -> int

g 1;;
val it : int = 8
```

Note: a and b are not visible outside of g

Tuples

An ordered collection of n values $(v_1, v_2, \ldots, v_n)$ is called an n-tuple

Examples

`(3, false);` `val it = (3, false) : int * bool`	2-tuples (pairs)
`(1, 2, ("ab",true));` `val it = (1, 2, ("ab", true)) :` ?	3-tuples (triples)

Equality defined componentwise, ordering lexicographically

```
(1, 2.0, true) = (2-1, 2.0*1.0, 1<2);;
val it = true : bool
```

```
compare (1, 2.0, true) (2-1, 3.0, false);;
val it : int = -1
```

provided = is defined on components

Tuple patterns

Extract components of tuples

```
let ((x,_),(_,y,_)) = ((1,true),("a","b",false));;
val x :  int = 1
val y :  string = "b"
```

Pattern matching yields bindings

Restriction

```
let (x,x) = (1,1);;
...
 ...  ERROR ...  'x' is bound twice in this pattern
```

Restriction can be circumvented using `when` clauses, for example:

```
let f = function
        | (x,y) when x=y -> x
        | (x,y)          -> x+y
```

Pattern matching on results of recursive calls

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

$$\texttt{sumProd}\ [\,] = (0,1)$$

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)
```

A blackboard exercise

A function from the `List` library:

- $\texttt{List.unzip([(}x_0,y_0\texttt{);(}x_1,y_1\texttt{);...;(}x_{n-1},y_{n-1}\texttt{)]}$
  $= \texttt{([}x_0\texttt{;}x_1\texttt{;...;}x_{n-1}\texttt{],[}y_0\texttt{;}y_1\texttt{;...;}y_{n-1}\texttt{])}$

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`let square x = x * x`	`int -> int`	Default

A squaring function on floats: `square: float -> float`

Declaration	
`let square(x:float) = x * x`	Type the argument
`let square x:float = x * x`	Type the result
`let square x = x * x: float`	Type expression for the result
`let square x = x:float * x`	Type a variable

You can mix these possibilities

Summary

- Functions as ”first-class citizens”
- Types, polymorphism and type inference
  - type constraints (equality and comparison)
  - type declarations
- Lists
- Selected language constructs