Functional Programming

Advanced Functional Programming

Boris Düdder

2019/07/16

Department of Computer Science,
University of Copenhagen
Sigurdsgade 41
DK-2100 Copenhagen, Denmark
Email: boris.d@di.ku.dk

UNIVERSITY OF COPENHAGEN

F# is a strongly typed language

Type	int	float	char	string	float	float
Three	3	3.0	'3'	"3"	3e0	3.0e0

Type	syntax	Examples	Value
int, int32	<int or hex>	3, 0x3	3
	<int or hex>l	31, 0x31	

Metatype	Type name	Description
Boolean	bool	
Integer	int	
	byte	
	sbyte	
	int8	
	uint8	
	int16	
	uint16	
	int32	
	uint32	
	int64	
	uint64	
Real	float	
	double	
	single	
	float32	
	decimal	
Character	char	
	string	
None	unit	
Object	obj	
Exception	exn	

<table>
<tr><th>Operator</th><th>Associativity</th><th>Description</th></tr>
<tr><td>+<expr>, -<expr>,
~~~<expr></td><td>Left</td><td>Unary identity, negation, and bitwise negation operator</td></tr>
<tr><td>f <expr></td><td>Left</td><td>Function application</td></tr>
<tr><td><expr> ** <expr></td><td>Right</td><td>Exponent</td></tr>
<tr><td><expr> * <expr>,
<expr> / <expr>,
<expr> % <expr></td><td>Left</td><td>Multiplication, division and remainder</td></tr>
<tr><td><expr> + <expr>,
<expr> - <expr></td><td>Left</td><td>Addition and subtraction binary operators</td></tr>
<tr><td><expr> ^^^ <expr></td><td>Right</td><td>bitwise exclusive or</td></tr>
<tr><td><expr> < <expr>,
<expr> <= <expr>,
<expr> > <expr>,
<expr> >= <expr>,
<expr> = <expr>,
<expr> <> <expr>,
<expr> <<< <expr>,
<expr> >>> <expr>,
<expr> &&& <expr>,
<expr> ||| <expr> ,</td><td>Left</td><td>Comparison operators, bitwise shift, and bitwise 'and' and 'or'.</td></tr>
<tr><td><expr> && <expr></td><td>Left</td><td>Boolean and</td></tr>
<tr><td><expr> || <expr></td><td>Left</td><td>Boolean or</td></tr>
</table>

Types

It is easy in F # to give a name to a type to make code easier to read.

```
type department = string
type costs = (department * float) list
let total (costs:costs) : float =
        List.fold (fun acc (_,f) -> acc+f) 0.0 costs
```

Note:

The type of department is just a synonym for the type of string.

The total function can therefore be used on all values of the type (string * float) list.

Type-generic type abbreviations

Type abbreviations can be generic so that it is possible to write generic code there refers to a type abbreviation:

```
// association lists mapping strings to values of type 'a
type 'a alist = (string * 'a) list
let add (m:'a alist) (s:string) (v:'a) : 'a alist = (s,v)::m
let rec look (m:'a alist) (s:string) : 'a option =
        if List.isEmpty m then None
        else    if fst(List.head m) = s then Some(snd(List.head m))
                else look (List.tail m) s
let empty () : 'a alist = []
```

Note:

We can use the empty list [] to represent the empty association list. We will later see how we with modules can ensure that the type 'a alist becomes "fully abstract" so that only the mentioned functions can be used to operate on the constructed association lists.

Record types

Records in F # allow you to name items in a tuple. The syntax for defining a record type is quite simple:

```
type person = {first:string; last:string; age:int}
let xs = [{first="Lene"; last="Andersen"; age=56};
          {last="Hansen"; first="Jens"; age=39}]
let name (p:person) : string = p.first + " " + p.last
let incr_age (p:person) : person = {p with age=p.age+1}
let ys = List.map incr_age xs
```

Note:

- When designing a record, the field order is insignificant.
- Items in a record can be extracted using the dot notation (p.first).
- A new record can be constructed (with an updated item) using *with*-construct.

Pattern matching

Generally, pattern recognition allows to examine and break down a value in its constituents.

We will look at pattern recognition based on the type of values we examine.

In F #, pattern recognition can occur in several different program constructions:

1. In simple let bindings.
2. In match-with constructions.
3. In function parameters.

Pattern matching

Pattern matching in tuples

```
let x = (34,"hej",2.3) // construct triple
let (_,b,f) = x // use of wildcard (_)
do printfn "%s:%f" b f // b and f are available here
```

Pattern matching in records

```
type person = {first:string; last:string; age:int}
let name ({first=f;last=l}:person) = f + " " + l
```

Note:

1. Matching on records only requires that a selection of field names be mentioned.
2. If multiple record types use the same field names, it may be necessary type annotations.
3. Pattern recognition for tuples and records is also widely used in function parameters: `let swap (x:'a,y:'b) : 'b * 'a = (y,x)`

Pattern matching on

The type int option is an example of a simple so-called "sum type", also called ”discriminated union", which represents values that are either None or one value Some (v), where v is a value of type int.

Here is a function that "lifts" addition to values of the type int option:

```
let add_opt (a:int option) (b:int option) : int option =
match a, b with
        | Some a, Some b -> Some(a+b)
        | _ -> None
```

Note:

1. Here, a form of "nested pattern matching" is used in pairs of values of the type int option.
2. When designing and matching tuples one can often do without the use of parentheses.
3. Variables can be bound in a match case and refer to the right side of a match where they will handle the matched values.

Simple sum types

It is easy in F # to declare a sum type consisting of a smaller amount of "tokens".

Example:

```
type country = DK | UK | DE | SE | NO
type currency = DKK | EUR | USD | CHF | GBP
type direction = North | South | East | West
```

Pattern matching:

```
let opposite (dir:direction) : direction =
        match dir with
        | North -> South // A constructor can appear
        | South -> North // both as a pattern and as
        | East -> West // an expression (just as
        | West -> East // integers can)
```

Note:

It may be advantageous to use simple sum types instead of eg. string or integer representations of data.

Sum types with argument-bearing constructors

Sum types can have constructors taking arguments.

Example: `type object = Pnt | Circ of float | Rect of float * float`

```
let rec area (obj:object) : float =
match obj with
        | Pnt -> 0.0
        | Circ r -> System.Math.PI * r * r
        | Rect (a,b) -> a * b
```

Note:

It is not a requirement that all designers take arguments; Pnt does not take one argument.

Designers who take arguments act as expressions of expression; eg. has Circ type float -> object.

Sum types can be generic

Sum-type definitions may be generic so that they are parameterized over one or more several types.

This option can be useful for creating reusable constructions.

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

let valOf (def:’a) (obj:’a option) :
’a =
match obj with
        | None -> def
        | Some v -> v
```

Sum-types are powerful

Sum-types allow for a lot of possible applications:

- Sum-types can be combined with tuples, and allow for a more precise specification of data relationships
- Sum-types are the foundation of ”embedded domain-specific languages” (EDSLs)

Recursively defined sum-types

Lists can be understood as generic recursively defined sum-type with two constructors [] and ::

```
type 'a list = [] | (::) of 'a * 'a list
```

Sum-types allow us to define advanced generic data-structures.

Example Turtle Graphics DSL

The example demonstrate how to develop a DSL for Turtle Graphics (similar to the PL Logo)

Idea:

Define a language which can specify the movement of a turtle (move forward by n steps, rotate for m degrees)

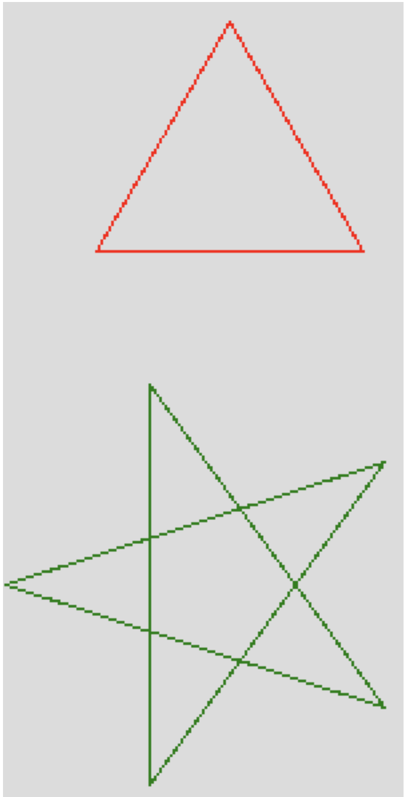
It gives the power to define programs moving a pen over a screen and draw.

```
type cmd = // turtle command
        | SetColor of color // change pen color
        | Turn of int // degrees right (0-360)
        | Move of int // 1 unit = 1 pixel
        | PenUp // avoid drawing when moving
        | PenDown // draw when moving
```

Example for Turtle Graphics

A turtle graphic program in DSL and its graphic generated

```
// Draw equal-sided triangle
let triangle x =
        [Turn 30; Move x; Turn 120; Move x;
        Turn 120; Move x; Turn 90]
// Define helper command to repeat
// turtle commands
let rec repeat n cmds =
        if n <= 0 then []
        else cmds @ repeat (n-1) cmds
// Draw a star using 5 lines
let star sz =
        repeat 5 [Move sz; Turn 144]
```



Tuples (product types)

Product type

Indexing functions for pairs

Parenthes are unnecessary but recommended

```
$fsharpi
...
> let a = (1, 1.0);;
val a : int * float = (1, 1.0)

> printfn "%A %A" (fst a) (snd a);;
1 1.0
val it : unit = ()

> let b = 1, "en", '\049'
val b : int * string * char = (1, "en", '1')
```

Pattern matching

```
> let (b1, b2, b3) = b;;
val b3 : char = '1'
val b2 : string = "en"
val b1 : int = 1
```

A tuple can be mutable, not its elements

```
> let mutable c = (1,2)
- c <- (2,3)
- printfn "%A" c;;
(2, 3)
val mutable c : int * int = (2, 3)
val it : unit = ()
```

Lambda calculus AST

Example: Lambda calculus DSL

We represent the Lambda calculus as DSL modeling its abstract syntax tree

```
type VarType = string
type LambdaTerm=
    | Var of VarType
    | App of LambdaTerm * LambdaTerm
    | Abs of VarType * LambdaTerm
```

```
let term0= Var("z")
```

$$z$$

```
let term1= Abs("x", Var("x"))
```

$$\lambda x.x = I$$

```
let term2= App(Abs("x", Var("x")), Var("z"))
```

$$(\lambda x.x)z$$

Calculator example

Questions?

Prof. Dr. Boris Düdder
University of Copenhagen
Copenhagen
Denmark

Email: boris.d@di.ku.dk
Wechat: BorisDuedder
Mobile: +45 93565748

Information:

diku.dk
ebcc.eu
blockchainschool.eu

Boris Düdder
Denmark

Scan the QR code to add me on WeChat

Co-funded by the Erasmus+ Programme of the European Union

BlockNet

Project: BlockChain Network Online Education for interdisciplinary European Competence Transfer
Project No: 2018-1-LT01-KA203-047044