BFNP – Functional Programming

Lecture 6: Modules

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.

Overview

- Modular program design including
  - encapsulation
  - abstraction and
  - reuse of software components.
- A module is characterized by:
  - a *signature* – an interface specifications and
  - a matching *implementation* – containing declarations of the interface specifications.
- Example based (incomplete) presentation to give the flavor.

Sources:

- Chapter 7: Modules.

An example: Search trees

Consider the following implementation of search trees:

```
   type Tree = Lf
             | Br of Tree*int*Tree;;

let rec insert i = function
  | Lf                -> Br(Lf,i,Lf)
  | Br(t1,j,t2) as tr ->
     match compare i j with
     | 0          -> tr
     | n when n<0 -> Br(insert i t1 , j, t2)
     | _          -> Br(t1,j, insert i t2);;


let rec memberOf  i = function
  | Lf          -> false
  | Br(t1,j,t2) -> match compare i j with
                   | 0    -> true
                   | n when n<0 -> memberOf i t1
                   | _          -> memberOf i t2;;
```

Example cont'd

Is this implementation adequate?

No. Search tree property can be violated by a programmer:

```
toList(insert 2 (Br(Br(Lf,3,Lf), 1, Br(Lf,0,Lf))));;
> val it = [3;1;0;2]: int list
```

Problem: *The tree argument to `toList` is not balanced.*

Solution: Hide the internal structure of search trees.

Module

A module is a combination of a

- signature, which is a specification of an interface to the module (the user's view), and an
- implementation, which provides declarations for the specifications in the signature.

Geometric vectors: Signature

The signature specifies one type and eight values:

```
// Vector signature
module VectorSimple
type vector
val ( ~-. ) : vector -> vector                    // Vector sign change
val ( +. )  : vector -> vector -> vector          // Vector sum
val ( -. )  : vector -> vector -> vector          // Vector difference
val ( *. )  : float  -> vector -> vector          // Product with number
val ( &. )  : vector -> vector -> float           // Dot product
val norm    : vector -> float                     // Length of vector
val make    : float * float -> vector             // Make vector
val coord   : vector -> float * float             // Get coordinates
```

The specification 'vector' does not reveal the implementation

- Why is make and coord introduced?

Geometric vectors (2): Simple implementation

An implementation must declare each specification of the signature:

```
// Vector implementation
module VectorSimple
type vector = V of float * float
let (~-.) (V(x,y))                = V(-x,-y)
let (+.) (V(x1,y1)) (V(x2,y2)) = V(x1+x2,y1+y2)
let (-.)  v1         v2          = v1 +. -. v2
let ( *.) a          (V(x1,y1)) = V(a*x1,a*y1)
let (&.) (V(x1,y1)) (V(x2,y2)) = x1*x2 + y1*y2
let norm  (V(x1,y1))             = sqrt(x1*x1+y1*y1)
let make  (x,y)                  = V(x,y)
let coord (V(x,y))                = (x,y)
```

- Since the representation of 'vector' is hidden in the signature, the type must be implemented by either a tagged value or a record.

Geometric vectors (3): Compilation

Suppose

- the signature is in a file 'VectorSimple.fsi'
- the implementation is in a file 'VectorSimple.fs'

A library file 'VectorSimple.dll' is constructed by the following command:

```
fsc -a VectorSimple.fsi VectorSimple.fs
```

On my MacBook installation I have to use `fsharpc` instead of `fsc`.

The library 'Vector' can now be used just like other libraries, such as 'Set' or 'Map'.

Geometric vectors (4): Use of library

A library must be referenced before it can be used.

```
 #r @"/Users/.../VectorSimple.dll";;
--> Referenced '/Users/.../VectorSimple.dll'
open Vector ;;

let a = make(1.0,-2.0);;
val a : vector
let b = make(3.0,4.0);;
val b : vector
let c = 2.0 *. a -. b;;
val c : vector

coord c ;;
val it : float * float = (-1.0, -8.0)

let d = c &. a;;
val d : float = 15.0

let e = norm b;;
val e : float = 5.0
```

Notice: the implementation of `vector` is not visible and it cannot be exploited.

Type augmentation

A *type augmentation*

- adds declarations to the definition of a tagged type or a record type
- allows declaration of (overloaded) operators.

In the 'Vector' module we would like to

- overload $+$, $-$ and $\star$ to also denote vector operations.
- overload $\star$ is even overloaded to denote two different operations on vectors.

Type augmentation – signature (`Vector.fsi`)

```
module Vector

[<Sealed>]
type vector =
  static member ( ~- ) : vector -> vector
  static member ( + )  : vector * vector -> vector
  static member ( - )  : vector * vector -> vector
  static member ( * )  : float  * vector -> vector
  static member ( * )  : vector * vector -> float
val make : float * float -> vector
val coord: vector -> float * float
val norm : vector -> float
```

- The *attribute* `[<Sealed>]` is mandatory when a type augmentation is used.
- The "`member`" specification and declaration of an infix operator (e.g. $+$) correspond to a type of form *type*$_1$ `*` *type*$_2$ `->` *type*$_3$
- The operators can still be used on numbers.

Type augmentation – implementation and use (`Vector.fs`)

```
module Vector

type vector =
  | V of float * float
  static member (~-) (V(x,y))                 = V(-x,-y)
  static member (+) (V(x1,y1),V(x2,y2)) = V(x1+x2,y1+y2)
  static member (-) (V(x1,y1),V(x2,y2)) = V(x1-x2,y1-y2)
  static member (*) (a, V(x,y))               = V(a*x,a*y)
  static member (*) (V(x1,y1),V(x2,y2)) = x1*x2 + y1*y2
let make (x,y)      = V(x,y)
let coord (V(x,y)) = (x,y)
let norm (V(x,y))  = sqrt(x*x + y*y)
```

The operators +, −, ⋆ are available on vectors even without opening:

```
let a = Vector.make(1.0,-2.0);;
val a : Vector.vector

let b = Vector.make(3.0,4.0);;
val b : Vector.vector

let c = 2.0 * a - b;;
val c : Vector.vector
```

Type Extension

You can insert function declarations between the type defintion and member declarations using *type extension*

```
module Vector
type Vector =
  | V of float * float
let make(x,y)      = V(x,y)
let coord(V(x,y)) = (x,y)
type Vector with
     static member (~-) (V(x,y))              = V(-x,-y)
     static member (+) (V(x1,y1),V(x2,y2)) = V(x1+x2,y1+y2)
     static member (-) (V(x1,y1),V(x2,y2)) = V(x1-x2,y1-y2)
     static member (*) (a, V(x,y))           = V(a*x,a*y)
     static member (*) (V(x1,y1),V(x2,y2)) = x1*x2 + y1*y2
let norm(V(x,y))  = sqrt(x*x + y*y)
```

A library file 'VectorTypeExtension.dll' is constructed by the following command:

```
fsc -a Vector.fsi VectorTypeExtension.fs
```

On my MacBook installation I have to use `fsharpc` instead of `fsc`.

Customizing the string function

```
module Vector
type vector =
    | V of float * float
    override v.ToString() =
            match v with | V(x,y) -> string(x,y)

let make (x,y)       = V(x,y)
    ...
type vector with
  static member (~-) (V(x,y))             = V(-x,-y)
   ...
```

- The default ToString function that do not reveal a meaningful value is overridden to give a string for the pair of coordinates.
- A type extension is used.

Example:

```
let a = Vector.make(1.0,2.0);;
val a : Vector.vector = (1, 2)

string(a+a);;
val it : string = "(2, 4)"
```

F# has OO capabilities which is heavily used when integrating to the .NET library.

A class can be defined as follows:

```
type ObjVector(X: float, Y: float) =
    member v.x = X
    member v.y = Y
    member v.coord() = (v.x, v.y)
    member v.norm() = sqrt(v.x * v.x + v.y * v.y)
    static member (~-) (v: ObjVector) = ObjVector(- v.x,- v.y)
    static member (+)  (v1: ObjVector, v2:ObjVector)
        = ObjVector(v1.x + v2.x, v1.y + v2.y)
    static member (-)  (v1: ObjVector, v2:ObjVector)
        = ObjVector(v1.x - v2.x, v1.y - v2.y)
    static member (*) (a,v:ObjVector) = ObjVector(a*v.x,a*v.y)
    static member (*)  (v1: ObjVector, v2:ObjVector)
        = v1.x * v2.x + v1.y * v2.y
```

Notice the syntactic resemblance with *Type Augmentation*.

The use of a class is straight forward:

```
let a = ObjVector(1.0, -2.0)
> val a : ObjVector
let b = ObjVector(Y=4.0, X=3.0)
> val b : ObjVector
b.coord()
> val it : float * float = (3.0, 4.0)
let c = 2.0 * a - b
> val c : ObjVector
c.coord()
> val it : float * float = (-1.0, -8.0)
b.x
> val it : float = 3.0
let d = c * a
> val d : float = 15.0
let e = b.norm()
> val e : float = 5.0
let g = (+) a b
> val g : ObjVector
g.coord()
> val it : float * float = (4.0, 2.0)
```

Notice the use of *Named Arguments*.

Parametrized modules (`QueueSimple.fsi`)
Type variables in signatures

Example to make a Queue class where objects in the queue can be of any type τ as long as they are all of the same type τ.

```
module Queue
type Queue<'a>
val empty : Queue<'a>
val put   : 'a -> Queue<'a> -> Queue<'a>
val get   : Queue<'a> -> 'a * Queue<'a>
exception EmptyQueue
```

Notice you can write

`type 'a Queue`

instead of

`type Queue<'a>`

Parametrized modules (`QueueSimple.fs`)
Queue Implementation

The Queue is implemented using a front list and a rear list supporting either constant time or linear time `get` functionality.

The “hope” is that in many cases it will be constant time.

```
module Queue
exception EmptyQueue
type Queue<'a> = {front: 'a list; rear: 'a list}
let empty = {front = []; rear = []}
let put y {front = xs; rear = ys} = {front = xs; rear = y::ys}
let rec get = function
              | {front = x::xs; rear = ys} ->
                    (x,{front = xs; rear = ys})
              | {front = []; rear = []} -> raise EmptyQueue
              | {front = []; rear = ys} ->
                    get {front = List.rev ys; rear = []}
```

If the `front` list is empty, then the `rear` list is reversed and used as `front` list.

Parametrized modules
Queue Example

Notice the most generic type for `q0` and then afterwards a non polymorphic `int` queue.

```
\#r @"/Users/.../QueueSimple.dll"
let q0 = Queue.empty
> val q0 : Queue.Queue<'a>
let q0 = Queue.empty : Queue.Queue<int>
> val q0 : Queue.Queue<int>
let q1 = Queue.put 1 q0
> val q1 : Queue.Queue<int>
let q2 = Queue.put 2 q1
> val q2 : Queue.Queue<int>
let (x,q3) = Queue.get q2
> val x : int = 1
  val q3 : Queue.Queue<int>
let q4 = Queue.put 4 q3
> val x2 : int = 2
  val q5 : Queue.Queue<int>
let (x2,q5) = Queue.get q4
> val x2 : int = 2
  val q5 : Queue.Queue<int>
```

Customizing Equality

Structural Equality does not work for the data representation chosen for Queue:

```
> let qnew = Queue.put 2 q0 ;;
val qnew : Queue.Queue<int>

> qnew = q3 ;;
val it : bool = false
```

The representation of

`qnew`, `{front=[]; rear=[2]}` and `q3`, `{front=[2]; rear=[]}`

is different.

Customizing Equality

We can override the default equality function with type augmentation.
An equality constraint is added to the element type in the signature:

```
type Queue<'a when 'a : equality>
```

You cannot override in a separate type extension.
You also have to re-define the hash function.
Below we also override `ToString` at the same time.

```
module Queue
exception EmptyQueue
[<CustomEquality;NoComparison>]
type Queue<'a when 'a : equality> =
     {front: 'a list; rear: 'a list}
     member q.list() = q.front @ (List.rev q.rear)
     override q1.Equals qobj =
         match qobj with
         | :? Queue<'a> as q2 -> q1.list() = q2.list()
         | _ -> false
     override q.GetHashCode() = hash (q.list())
     override q.ToString() = string (q.list())
```

```
qnew = q3
> val it : bool = true
```

Customizing Ordering

You can also change the ordering and indexing on values on a defined type.

```
module Queue
exception EmptyQueue
[<CustomEquality;CustomComparison>]
type Queue<'a when 'a : comparison> =
     {front: 'a list; rear: 'a list}
     member q.list() = q.front @ (List.rev q.rear)
...
     interface System.IComparable with
       member q1.CompareTo qobj =
         match qobj with
         | :? Queue<'a> as q2 -> compare (q1.list()) (q2.list())
         | _ ->
           invalidArg "qobj"
                      "cannot compare values of different types"
...
     member q.Item
       with get n = (q.list()).[n]
...
```

Summary

Modular program development

- program libraries using signatures and structures
- type augmentation, overloaded operators, customizing string (and other) functions
- Encapsulation, abstraction, reuse of components, division of concerns, ...
- ...