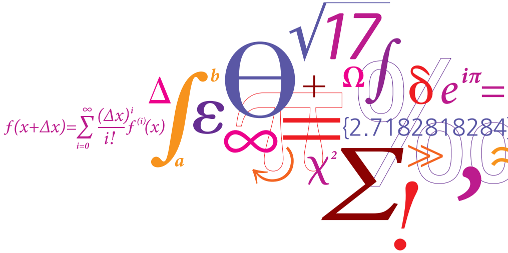
# 02157 Functional Programming

Lecture 9: Module System – briefly

Michael R. Hansen



DTU Informatics
Department of Informatics and Mathematical Modelling

## Overview

- Supports 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. (A fast reading suffices.)

## Overview

- Supports 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. (A fast reading suffices.)

## Overview

- Supports 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. (A fast reading suffices.)

## Overview

- Supports 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. (A fast reading suffices.)

## 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
```

Solution: Hide the internal structure of search trees.

## 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
```

Solution: Hide the internal structure of search trees.

## 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
```

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.

## 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 Vector
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: Signature

The signature specifies one type and eight values:

```
// Vector signature
module Vector
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: Signature

The signature specifies one type and eight values:

```
// Vector signature
module Vector
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 Vector
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 (2): Simple implementation

An implementation must declare each specification of the signature:

```
// Vector implementation
module Vector
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 ’Vector.fsi’
- the implementation is in a file ’Vector.fs’

A library file ’Vector.dll’ is constructed by the following command:

```
C:\mrh\Kurser\02157-11\Week 10\fsc -a Vector.fsi Vector.fs
```

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 @"c:\mrh\Kurser\02157-11\Week 10\Vector.dll";;
--> Referenced 'c:\mrh\Kurser\02157-11\Week 10\Vector.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 * to also denote vector operations.
- overload * is even overloaded to denote two different operations on vectors.

## 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 `*` to also denote vector operations.
- overload `*` is even overloaded to denote two different operations on vectors.

## Type augmentation – signature

```
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 – signature

```
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 – signature

```
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 – signature

```
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

```
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 augmentation – implementation and use

```
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
```

## 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)"
```

## 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)"
```

## 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, ...
- ...