Introduction to Modules in F#

Torben Mogensen

November 22, 2016

These notes is a brief introduction to modules to supplement the textbook. Read this first, and then Chapter 7 in the textbook.

1 Simple Modules

Modules are a way to split a program into multiple files. A program is built from a number of modules and a single `.fsx` file that binds these modules together to an executable program. A module is in itself not executable, but functions declared in a module can be called from either the interactive system (`fsharpi`), from other modules or from a `.fsx` file.

A simple module is a file *`filename`*`.fs` which starts with a line of the form `module` *`filename`* followed by declarations. Note that the filename specified in the `module` line must be the same as the name of the file (sans extension). As an example, we can define the following module in the file `Fib.fs`:

```
module Fib

let rec fib n = if n<2 then n else fib(n-1) + fib(n-2)
```

If we start the interactive system with the command `fsharpi Fib.fs`, we get the following output:

```
F# Interactive for F# 4.0 (Open Source Edition)
Freely distributed under the Apache 2.0 Open Source License

For help type #help;;

[Loading /home/torbenm/DIKU/home/whm779/vile-backup/PoP2016/Fib.fs]

namespace FSI_0002
  val fib : n:int -> int

>
```

This shows that we have defined a `namespace` containing a declaration of a value `fib` of type `int -> int`. A namespace is a set of declarations that do not conflict with or replace declarations in other namespaces.

Note that `fsharpi` does not terminate (as it would if given a `.fsx` file), but shows a prompt where you can type expressions and declarations.

To call `fib`, we must prefix the call with the the module name, so we can at the prompt write `Fib.fib 7;;` followed by newline (which we from now on will assume implicit after a double semicolon) and get `13` as a result.

It is possible to omit the module prefix when using names declared in the module by *opening the module*. If we at the prompt write `open Fib;;`, we can afterwards simply write `fib 8` to get `21` as a result. Opening a module will, however, replace any current declarations that overlap with declarations in the module, so the namespaces are no longer separate. So it should be done with caution and is generally not recommended. Namespaces can be nested, which is common for library modules, so file operations are found in the module / namespace `System.IO.File`. To avoid writing the full prefix, it is common to partly open this namespace by writing `open System.IO`, after which it is sufficient to write only the `File` prefix, e.g, `File.OpenText`. Partly opening a namespace (mostly) prevents names from the current namespace from being overwritten. So a general advise is not to fully open namespaces / modules, but partially opening them is common practice.

One of the advantages of using modules is that modules can be compiled separately. To compile the module above, we write `fsharpc -a Fib.fs`. This produces a file `Fib.dll`, where the `dll` extension is used for libraries and modules that can be shared between multiple programs. We can now start the interactive system using the command `fsharpi -r Fib.dll`, which makes the module available in the same way as when writing `fsharpi Fib.fs`, but it is faster to do so, since the module is already compiled. If we use the `Fib` module in an executable file, say, `FunnyFunctions.fsx`, we can do so by using the prefix, e.g, `Fib.fib` or by opening the `Fib` module before using the `fib` function. If `FunnyFunctions.fsx` is executed using `fsharpi`, we must use the command

```
fsharpi -r Fib.dll FunnyFunctions.fsx
```

If we, instead, compile `FunnyFunctions.fsx` to make a `.exe` file, we must use the command

```
fsharpc -r Fib.dll FunnyFunctions.fsx
```

which produces the executable `FunnyFunctions.exe`, which can be executed using `mono`.

2 Signatures

We have, so far, seen the following motivations for using modules:

- We can reuse a module in several programs without having to copy the text.
- We can compile a module separately from the program that uses the module.
- We get separate namespaces, so we can have multiple declarations using the same name, as long as they are in different namespaces. We have seen examples of this in the `List` and `Array` modules that both define functions called `length`, `map`, and so on.

But there is a further motivation for using modules: The user of the module need not worry about *how* the functions defined in the module are defined, only their type and behaviour. So we can hide details of the implementation from the user of the module, and we can change these details without invalidating the programs that use the module (though these may have to be recompiled). Hiding details of implementation in this way is called *abstraction.*

To increase abstraction, we can separate a module into a specification part and an implementation part. The specification part is called a *signature* or *interface* and contains declarations of the types of the names defined in the module, while the implementation part has the full declarations of these names. The signature is written in a separate file *`filename`*`.fsi`, where the file name is the module name. For example, we can make a signature for the `Fib` module by writing a file `Fib.fsi` with the content

```
module Fib

val fib : int -> int
```

Note that only the type of `fib` is specified. The `val` keyword specifies that `fib` is a value (as opposed to a type).

We use the signature file when compiling a module:

```
fsharpc -a Fib.fsi Fib.fs
```

will compile the module `Fib.fs` to a library file `Fib.dll`, but it will also verify that `Fib.fs` defines all the names specified in the interface and with the specified types. `Fib.fs` can define more names than specified in the interface, but not less, and the types can not differ. Furthermore, only the names specified in the interface can be used outside the module. For example, if `Fib.fs` in addition to declaring `fib` declares a function `fac`, `fac` is hidden and can not be accessed by programs using the module. This allows auxiliary functions used in the implementation to be hidden from users of the module.

3 Overloaded operators

Some operators in F# are *overloaded*, which means that the operator works on several different types. For example, `+` works on both integers, floats and strings.

We can define overloaded operators on our own types, as long as they are enumerated types or sum types. We do this by adding*member functions* to the type declaration. For example, we can specify a number type with a signature `Number.fsi` containing

```
module Number

[<Sealed>]
type Number =
  static member ( + ) : Number * Number -> Number
  static member ( - ) : Number * Number -> Number
  static member ( * ) : Number * Number -> Number
  static member ( / ) : Number * Number -> Number

val ofInt : int -> Number
val show : Number -> string
```

The `[<Sealed>]` attribute is required for adding member functions to a type in a signature, unless the type is specified to be a sum type (discriminated union type) in the signature. The `[<Sealed>]` is an attrubute of the type and not the module, so if several types in a module have member functions, they must all be sealed.

This signature says that the module has a type called `Number` that overloads the four arithmetic operators. Additionally, the module has a function for converting an integer to a number and for converting a number to a string.

We can make an implementation `Number.fs` of this signature where the `Number` type is a sum type having both integers and reals:

```
module Number

type Number =
  | I of int | F of float
  static member ( + ) (a, b) =
    match (a,b ) with
    | (I m, I n) -> I (m + n)
    | (I m, F y) -> F (float m + y)
    | (F x, I n) -> F (x + float n)
    | (F x, F y) -> F (x + y)

  static member ( - ) (a, b) =
    match (a,b ) with
    | (I m, I n) -> I (m - n)
    | (I m, F y) -> F (float m - y)
    | (F x, I n) -> F (x - float n)
    | (F x, F y) -> F (x - y)

  static member ( * ) (a, b) =
    match (a,b ) with
    | (I m, I n) -> I (m * n)
    | (I m, F y) -> F (float m * y)
    | (F x, I n) -> F (x * float n)
    | (F x, F y) -> F (x * y)

  static member ( / ) (a, b) =
    match (a,b ) with
    | (I m, I n) -> if m % n = 0 then I (m / n)
                    else F (float m / float n)
    | (I m, F y) -> F (float m / y)
    | (F x, I n) -> F (x / float n)
    | (F x, F y) -> F (x / y)

let ofInt n = I n

let show = function
  | I m -> sprintf "%d" m
  | F x -> sprintf "%g" x
```

Note that the keyword `static` must be aligned with the first `|` in the type declaration.

If we compile this module using the command

```
fsharpc -a Number.fsi Number.fs
```

and start the interactive system using the command `fsharpi -r Number.dll`, we can write expressions like `Number.show (Number.ofInt 7 / Number.ofInt 5);;` and get the result `"1.4"`. Note that we do not need to specify a prefix for the `/` operator.