Abstract Data Types

Lecture 09

Modules

Chapter 7 of HR book

Modules, in general

Modules support:

- *Namespaces:* a basic scoping mechanism to avoid name clashes
- *Code reuse:* no need to duplicate code which can be packaged and called when needed
- *Abstraction:* choose what to hide or reveal (information hiding/encapsulation)

F# does **not** have a first-class notion of modules such as Ocaml, so we concentrate on the latter

Data Abstraction

- **Data abstraction** is one of the most important techniques for structuring programs.
- Provides an ***interface*** that serves as a **contract** between the ***client*** and the ***implementor*** of an abstract type.
  - The interface specifies what the client may rely on for her own work, and, simultaneously, what the implementor must provide to satisfy the contract.

Data Abstraction 2

- The interface **isolates** the client from the implementor so that each may be developed in isolation from the other
  - ***Information (data) hiding***
- In particular, one implementation may be **replaced** by another without affecting the behavior of the client, provided that the two implementations meet the same interface.

ADT

- An **abstract data type** (ADT) is a type with a **public** name equipped with a **public** set of operations for creating/combining/observing values of that type.
- ADT is implemented by providing a **representation** type for the values of the ADT and an **implementation** for the operations defined on values of the representation type.
- What makes an ADT **abstract** is that the representation type is **hidden** from clients of the ADT. Consequently, the only operations that may be performed on a value of the ADT are the exposed ones.

ADTs in practice

- They can be used as ordinary built-in types – actually, **int** is an ADT …
- They have a nice type-theory (existential types), which is the dual of polymorphism (universal types)
- Solid connections with algebra and algebraic specifications
- Your language **must** be strongly-typed.

More info

- A very readable essay on ADT and their difference with object-orientation:

  “On Understanding Data Abstraction, Revisited” by William R. Cook.

  OOPSLA '09, Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications

ADT in F#

- In F# this can be (partially) achieved via the use of *signatures* and *modules*
  - sig files (**file.fsi**) specify the interface/API
  - module declarations (**file.fs**) represent the implementors side
- They are "matched" by the compiler, which creates a DLL, i.e. a *dynamic linked library* (**file.dll**)
- Then, the DLL is linked at run-time, possibly interactively
- This allows to have **one** ADT and **multiple** representations, but only **one** can be used at any time

If a module M matches a sig T

- *Signature matching:* every name declared in **T** is defined in **M** at the same or a more general type.
- *Opacity:* any name defined in **M** that does not appear in **T** is not visible to code outside of **M**.
  - We say that the sig **seals** the module
  - Compare to **visibility modifier** in say Java

How to build a project/dll within code

- Create a new project
  - `dotnet new console -lang "F#" -o MyPj`
- cd there and open code
  - `cd MyP; code .`
- assuming your code is in *Library.fs* and your interface in *Library.fsi*, **add them by right-clicking on the left window (***add file***)**
  - be sure the *.fsi is first
- Or you can edit the MyPy.proj file
- build the dll by clicking the (|>) button on the left
  - Or run “dotnet build” in the terminal
- if compilation is OK, **MyPj.dll** will be under “bin/Debug/net6.0”.
  - Using “dotnet build -o .” will put the dll in the current dir

Using a dll

You know it already:

- you can run F# interactive from the shell like that, assuming it is in the path:

```
dotnet fsi –r MyPj.dll
```

Or inside the IDE

```
#r "MyPl"
```

- Now open the module (or use qualified names) and use it in your script file

How to build a project at the command line

- create a new solution file for your project MyP
  - dotnet new sln -o MyP
- create a class library project in the src folder named say MyPLibrary
  - cd MyP; dotnet new classlib -lang "F#" -o src/MyPLibrary
- put your code in (say) Library.fs and your interface in Library.fsi under src
- at the top level add the project to the MyP solution using the “dotnet sln add” command
  - dotnet sln add src/MyPLibrary/yPLibrary.fsproj
- Run “dotnet build” to build the project

Howto: using **fsharpc/fsc.exe**

If you’re lucky enough to be able to access the compiler directly (if you do, let us know how):

- Open a terminal, go to the directory containing your files
- Run

```
fsharpc -a Library.fsi Library.fs
```

- This will produce the library file `Library.dll`

Lecture plan

- An example ADT: sets of integers
  - A naive rep as list w/o repetitions
  - A better one using binary search trees
  - Polymorphic ADTs: queues
  - PBT over ADT (bonus lecture)