Abstract Data Types

Lecture 13

Modules

(part 1)

Chapter 7 of HR book

Data Abstraction

- **Data abstraction** is on the most important technique 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 its 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
  - ***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 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 summary

- They work as ordinary built-in types – actually, **int** is an ADT …
- Can be efficiently implemented
- Have a nice type-theory (existential types), which is the dual of polymorphism (universal types)
- Solid connections with algebra and algebraic specifications
- Hence, a very good candidate for the notion of **data abstraction**

ADT in F#

- In F# this can be 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 compiles a DLL, i.e. a *dynamic link library* (**file.dll**)
- Then, the dll is linked at run-time, possibly interactively
- This allows to have **one** ADT and **multiple** reps, but only one will eventually be used

More info

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

  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

Howto: using **fsharpc/fsc**

- Open a terminal, go to the directory containing your files
- For our working example: run
  - `fsharpc –a set.fsi listFS.fs`
- This will produce a library file listFS.dll. To use it you can run F# interactive from the shell like that:
  - `fsharpi –r listFS.dll,` or
  - `#r "ListFs"` inside the IDE
- Now open the module (or use qualified names) and use it in your script file

Howto: **MonoDeveop**

- "*Open*" a new project **(solution)** and choose "*F# library*" under **Miscellaneous**. Choose the name and location of the *dll*
- Go to Solution Explorer **(View >> visual design),** remove for hygiene reasons the *fs and *fsx that are generated
- "*Add existing files*" (right click) namely the *fsi and *fs. Move the *fsi to be first. "*Build*" the project (F8)
- To use the *dll*, you need to reference it in your *.fsx file
  - `#r "directory\name.dll"`
- Note that the dll will be under bin\debug in the folder that the IDE builds for you. Or just move it where you prefer

Lecture plan

- Today: an example ADT: sets of integers
  - A naive rep as lists w/o repetitions
  - A better one using binary search trees

- Friday:
  - Polymorphic ADTs: queues/stacks
  - Type augmentations and relations with OO classes