The F# Programming Language CEFP2009 Warm-Up Session

Páli Gábor János
E-mail: pgj@elte.hu

Eötvös Loránd University, Faculty of Informatics,
Department of Programming Languages and Compilers

May 23, 2009

Quick Overview

1. Forward Pipe Operator
2. Sequences
3. Lazy Values
4. Objects

Links for the Session

- You can find a draft version for the session slides at `http://people.inf.elte.hu/pgj/fsharp/s3d.pdf`
- Please upload your solutions for the exercises at `https://pnyf.inf.elte.hu/cefp-es/`

Forward Pipe Operator

- Perhaps the most important operator.
- Defined as follows:

```
let (|>) f x = x f
```

- Operator `|>` is just "function application in reverse".
- Advantages:

Clarity Allows to perform data transformation and iterations in a forward-chaining, pipelined style.

Type inference Allows type information to be flowed from input objects to the functions manipulating objects.

A Practical Example

- Take a number, double it, then convert it to a string, then reverse the string.

```
let double x = 2 * x
let toStr (x : int) = x.ToString ()
let rev   (x : string)
  = new String(Array.rev (x.ToCharArray ()))
```

- $512 \rightarrow 1024 \rightarrow$ "1024" $\rightarrow$ "4201"

```
let result = rev (toStr (double 512))
```

The Pipeline Syntax

The code is straightforward, but it has a complicated syntax. We simply want to take the result of one computation and pass that to the next computation.

```
let step1 = double 512
let step2 = toStr step1
let result = rev step2
```

Let's eliminate the temporary variables, and forward the values to a function by the |> operator. It is essentially allows to specify the parameter of a function before the call.

```
let result = 512 |> double |> toStr |> rev
```

Exercise

By using pipeline syntax, write an expression that determines the sum of the first ten even square numbers.

Sequences

- Many programming tasks require iteration, aggregation, and transformation of data steamed from various sources.
- `System.Collections.Generic.IEnumerable<type>` aka. `seq<type>`.
- `seq<type>` can be iterated, producing results of *type* on demand.
- Sequences can specify infinite series.

Sequence Expressions

```
> Seq.init_infinite (fun x -> x);;
val it : seq<int> = seq [0; 1; 2; 3; ...]

> seq { for x in 0 .. 10 -> (x, x * x) };;
val it : seq<int * int> = seq [(0,0); (1,1); ...]

> seq { for x in 0 .. 10 do
          if x % 3 = 0 then yield (x, x / 3) };;
val it : seq<int * int> = seq [(0,0); (3,1); ...]

let fileInfo dir =
 seq { for file in Directory.GetFiles(dir) do
  let creationTime = File.GetCreationTime(file)
  let lastAccessTime = File.GetLastAccessTime(file)
  yield (file,creationTime,lastAccessTime) }
```

Exercise

Write a function that calculates coordinates for checkerboards of size *n*: a coordinate should be yielded if the sum of row and column is even.

```
checkerboardCoordinates : int -> seq<(int * int)>
```

A Case Study: Rewriting the Factorial Function

```
let fi =
  (1,1)
    |> Seq.unfold (fun (i, x) ->
         Some (x, (i + 1, i * x)))

let seqFactorial n = Seq.nth n fi
```

`Seq.unfold`:

- Return a sequence that contains the elements generated by the given computation.
- An initial “state” is passed to the element generator.
- For each `IEnumerator` elements in the stream are generated on-demand until a `None` element is returned.

Lazy Values

- A memoizing function is one that avoids recomputing its results by keeping an internal table, called a lookaside table.
- Memoization is a form of caching.
- Another important variation on caching is a *lazy* value.
- A lazy value is a delayed computation of type `Microsoft.FSharp.Control.Lazy<'a>`.

Lazy Values Continued

- Lazy values are explicitly formed by using the keyword `lazy`.
- Lazy values are implemented as thunks holding either a function value that will compute the result or the actual computed result.
- Lazy values will be computed only once, and thus its effects are executed only once.
- Lazy values are implemented by a simple data structure containing a mutable reference cell.

Creating and Evaluating Lazy Values

```
> let x = lazy (printfn "Computed."; 42);;
val x : Lazy<int>


> let listOfX = [x; x; x];;
val listOfX : Lazy<int> list


> x.Force();;
Computed.
val it : int = 42
```

Getting Started with Objects

- Static type of a values can be explicitly altered by either throwing information away, upcasting, or rediscovering it, downcasting.
- The type hierarchy start with `obj` (`System.Object`) at the top and all its descendants below.
- An upcast (`:>`) moves a type up the hierarchy, and a downcast moves a type the hierarchy.
- Upcasts are type safe operations since the compiler always knows all the ancestors of a type through static analysis.
- Upcasts are required when defining collections that contain disparate types.
- Upcast means automical boxing of any value type, so they can be passed around by reference.

Upcasting Objects

- Converting a `string` to an `obj` by upcasting:

```
> let anObject = ("This is a string" :> obj);;
val anObject : obj
```

- Objects can be represented as strings as usual:

```
> anObject.ToString ();;
val it : string = "This is a string"
```

- Adding different type of objects to the same list:

```
> let moreObjects = [ ("A string" :> obj)
                    ; (2.0 :> obj)
                    ; ('Z' :> obj) ];;
val moreObjects : obj list
```

Downcasting Objects

- Downcast `:?>` change a value's static type to one of its descendant types, thus recovers information hidden by an upcast.
- Downcasting is not safe since the compiler does not have any way to statically determine compatibility relations between types.
- If downcasting does not succeed, it will cause an invalid cast exception (`System.InvalidCastException`) to be issued at runtime.
- Because of dangers of downcasting, it is often preferred to match patterns over .NET types.

Examples on Downcasting

- Some examples of downcasting:

```
> let boxedObject = box "abc";;
val boxedObject : obj


> let downcastString = (boxedObject :?> string);;
val downcastString : string = "abc"


> let invalidCast = (boxedObject :?> float);;
```

Exercise

Determine the types of objects in a list via pattern matching.
The type of the function to be written is as follows:

```
typeOfObjects : obj list -> string list
```

Comparison Example Revisited

Previously, we have defined an `Ordering` type that represents results for comparison:

```
type Ordering = LT | EQ | GT
```

We also added a `compare` function to its definition to create values of such type:

```
compare : 'a -> 'a -> Ordering
```

An Enhanced Version of compare

But because `compare` uses comparison operators to compare values, it requires to restrict the type of the arguments to ones that implement the `System.IComparable` interface:

```
let compare (x: #System.IComparable)
            (y: #System.IComparable) =
  match () with
    | _ when x > y -> GT
    | _ when x = y -> EQ
    | _ when x < y -> LT
```

Otherwise it will result in an exception when values of a type without comparison operators are used.

Records As Objects

- It is possible to simulate object-like behaviour by using record types, because record can have fields that are functions.
- Sometimes it is comfortable to use, because only the function's type is given in the record definition, so the implementation can be changed without having to define a derived class.
- Create multifunctional records without having to worry about any unwanted features we might also be inheriting.

Exercise

- Create a record `Shape` with fields `reposition` and `draw` with the following types (note that they are functions):
  - `reposition : Point -> unit`
  - `draw : unit -> unit`
- Write a `makeShape` function that receives an initial position (a `Point`) and a drawing function (`unit -> unit`) and creates a record of type `Point`.
- Use the `makeShape` function to create a circle (`circle`), and a square (`square`). Let the draw function for both of them is a textual representation, like:

```
Circle, with x = 33 and y = 66.
```

F# Types with Members

- A function added to a record can be called using dot notation, just like a member of a class from a library not written in F#.
- This provides a convenient way of working with records with mutable state, and it is also useful when exposing type in F# to other .NET languages.
- To include member definitions in a record, one should add them to the end of the definition, between `with` and `end` keywords.
- The definition of the members start with the keyword `member`, followed by:
  - an identifier that represents the parameter of type,
  - a dot,
  - a function name,
  - any other function parameters.

Exercise

- Create a record of type `Point` with the following *mutable* members:
  - `top` with type of `int`
  - `left` with type of `left`
- Add a member called `Swap` to this record that implements swapping of the values of the `top` and `left` fields.
- Create a simple main program to test the implemented functionality.

Object Expressions

- Heart of succint object-oriented programming in F#, they provide a concise syntax to create an object that inherits from an existing type.
- This is useful is we want to provide a short implementation of an abstract class or an interface or want to tweak an existing class definition.
- Object expressions allows to provide an implementation of a class or interface while at the same time creating a new instance of it.
- The syntax is similar to the syntax for creating new instance of record types.

Exercise

- Create a simple comparer object implementing the interface `System.Collections.Generic.IComparer`. It compares strings (`IComparer<string>`) by their reverse.
- Create a simple program to the implemented functionality: create an array (the same way as a list but enclosed in `[|` and `|]` symbols), then call the `Array.Sort` function with the comparer instance.
  Some sample data:

```
[| "Sandie Shaw"
 ; "Bucks Fizz"
 ; "Dana International"; "Abba"
 ; "Lordi" |]
```