Functional Programming Using F#
1 of n

Quest Integrity Boulder
March 30, 2016

Agenda

- Introduction
- Functional Data Types
- Recursion
- Lazy Evaluation
- Currying: Partial Function Application

Literature

- Programming F# 3.0, C. Smith, O'Reilly 2012
- F# Deep Dives, T. Petricek, 2014
- Real-World Functional Programming Using F#, T. Petricek, J. Skeet, 2010
- Dr. Erik Meijer, Functional Programming Fundamentals, Channel9
- Real World Haskell, O'Sullivan, Goerzen, Stewart, O'Reilly 2008
- https://en.wikibooks.org/wiki/F_Sharp_Programming
- http://fsharpforfunandprofit.com/

Early Functional Programming Languages

- Programming style that has been around since 1970s
- ML (meta-language) created by Robin Milner in the early 1970s
  - Type Inference (Hindley–Milner type system)
  - Static Typing
  - Parametric Polymorphism
  - Algebraic Data Types
  - Pattern matching
  - Impure
  - Garbage collected
  - Eager evaluation
- Miranda (1985, commercial). Similar to ML, but
  - Lazy
  - pure
- Haskell (1990). Based heavily on Miranda
  - Pure
  - Lazy
  - Type classes
  - Monads :-)
- Caml is a dialect of ML, and OCaml adds OO elements
- F# (2005), ML-dialect with heavy influence from Haskell and OCaml and other languages

F#

- Based on OCaml (looks almost the same)
- Multi-Paradigm
- Fully integrated into the .NET ecosystem
- Inter-operates with C# and VB.NET (with some pitfalls)
- Fully integrated into VS
- FSI - REPL

- Type inference
- Staticly typed
- Strongly typed
- Expression-based
- Eager Evaluation
- Pattern Matching
- Algebraic Data Types
- Units of measure
- Impure
- OO, imperative and FP
- Computational Expressions (aka Monads)

Functional vs Other Styles

Functional	**Imperative**	**OO**
Functions and Higher-Order Functions	Procedures	Objects
Declarative, i.e. what not how	Mutability	Encapsulation
Immutability	Loops	Inheritance
Algebraic Data Types	Side-effects	Polymorphism
Explicit about side-effects		
DSL		
Pattern matching and deconstruction		
Recursion		
Currying		

Evolution of C# w.r.t. FP

Version	Element
2	Generics (List<T>, Tuple<T>, …) Delegates
3	LINQ Lambda Expressions
5	IReadOnlyCollection (.NET 4.5) Immutable Collections
6	Null Propagator
7	Pattern Matching ADT

FP: Why Should we care?

- emphasizes a higher level of abstraction when thinking about the solution to a problem (declarative vs imperative)
- tends to be safer due to its immutable nature. Avoids **side-effects**.
- sees a resurgence due to limitations of traditional programming models based on imperative and OO styles (sharing, multi-threading, mutation, …)
- OO often awkward when expressing simple ideas (command pattern)
- Null considered harmful
- Following many recommended principles like loose coupling and the like often means code bloat (interfaces, ...)
- Brian Will
  - Object-Oriented Programming is Embarrassing: 4 Short Examples, *https://www.youtube.com/watch?v=IRTfhkiAqPw*
  - Object-Oriented Programming is Bad, *https://www.youtube.com/watch?v=QM1iUe6IofM*

Immutability

- Functional programming philosophy is about making side-effects explicit by using immutable data structures
- Some languages like Haskell are pure, i.e. they do not even allow side-effects
- Why are side-effects so important to control?

  A central idea is to make complicated from simple via function composition. If functions were allowed side-effects, it becomes impossible to reason about their results. This is a major source of pain in imperative programming styles...

Functional Data Types

- Tuple
- List
- Discriminated Unions
- Records

Tuple

Ordered, heterogeneous collection of **immutable** data

```
let person : string * int = ("John", 45)

let children : string * int * ((string * int) * (string * int)) = ("Father", 42, (("Daugther", 12), ("Son", 9)))
```

Common Operations:

- Extract 1st element: `let first_name = fst person`
- Extract 2nd element: `let age = snd person`
- Generally: `let first_name, age = person`

Tuples are an integral part in many functional languages and are therefore convenient to use (pattern matching, currying, ...).

List

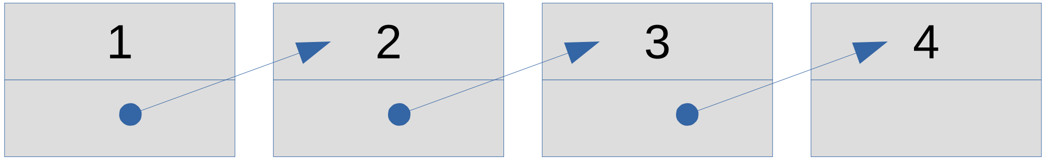
Ordered, homogenous collection of **immutable** data

```
let some_numbers = [1; 2; 3; 4]

let some_strings = ["C#"; "F#"; "VB"; "Rust"; "C"; "C++"]

let list_of_lists = [[1; 2]; [3; 4; 5]]
```

- F# lists are modeled as singly-linked lists:



List

Generate elements of a list

- Generator form:

```
[ for a in 1 .. 10 do yield (a * a) ]

val it : int list = [1; 4; 9; 16; 25; 36; 49; 64; 81; 100]
```

- Range form:

```
[1 .. 2 .. 10]

val it : int list = [1; 3; 5; 7; 9]
```

List

Common Operations

- List.Head returns the 1st element
- List.Tail returns all but the 1st element
- But generally: `let head::tail = list_of_lists`

Cons Operator: (::) : 'T → 'T list → 'T list, prepends an item to an existing list

```
1 :: 2 :: 3 :: []
```

```
val it : int list = [1; 2; 3]
```

Append Operator: (@) : 'T list → 'T list → 'T list, prepends the 1st list to the 2nd

```
let first      = [1; 2; 3;]
let second     = [4; 5; 6;]
let combined1  = first @ second
```

```
val combined1 : int list = [1; 2; 3; 4; 5; 6]
```

Note that a copy of the entire 1st list is made as its last element is modified to point to the 1st element of the 2nd list!

List.Map

F# (same in OCaml): List.map : ('a → 'b) → 'a list → 'b list

Haskell: list.map :: (a → b) → [a] → [b]

```
let lst = [1; 2; 3;]
let inc x = x + 1
List.map inc lst
```

```
val it : int list = [2; 3; 4]
```

map f [x1, x2, ..., xn] == [f x1, f x2, ..., f xn]

LINQ Select:

IEnumerable<TResult> **Select**<TSource, TResult>(this IEnumerable<TSource> source, Func<TSource, TResult> selector)

Note: More generally, we can write this as

List.map : ('a → 'b) → f 'a → f 'b

Later, we will recognize this as what is called a Functor.

List.Filter

List.filter : ('a → bool) → 'a list → 'a list

```
let lst = [ 1; 2; 3; 4;
5; 6; 7; 8 ]
let even x = x%2=0
List.filter even lst
```

```
val it : int list = [2; 4; 6; 8]
```

The same as `let lst_even = [ for a in lst do if a % 2 = 0 then yield a]`

In Haskell, `filter p xs = [ x | x <- xs, p x]`

LINQ Where:

IEnumerable<TSource> **Where**<TSource>(this IEnumerable<TSource> source, Predicate<TSource> predicate)

List.Fold

List.fold : ('a → 'b → 'a) → 'a → 'b list → 'a

```
List.fold (fun a b -> a + b) 0 [0; 1; 2; 3; 4; 5; 6]
```

```
val it : int = 21
```

Accumulator		1	2	3	4	5	6
0							
1							
3							
6							
10							
15							
21							

Every time you want to reduce a list of items to one item, think fold!

List.Fold

List.fold : ('a → 'b → 'a) → 'a → 'b list → 'a

The fold operation reflects the current state in the iteration using the accumulator. We will later see that this is a very common pattern to simulate state by passing the state that changes as a function argument recursively.

Imperative Implementation

```
public static T Imperative<T>(this IEnumerable<T> items, T accumulator, Func<T, T, T> func)
{
    foreach (var item in items)
    {
        accumulator = func(accumulator, item);
    }
    return accumulator;
}
```

Functional Implementation

```
public static T Functional<T>(this IEnumerable<T> items, T accumulator, Func<T, T, T> func)
{
    return items.Any() == false ? accumulator : Functional<T>(items.Skip(1), func(accumulator, items.First()), func);
}
```

List.Fold

Fold or Aggregate is a very common pattern in data parallel implementations used for example on GPUs, as they expose fine-grained data parallelism.

LINQ Aggregate:

TSource **Aggregate**<TSource>(this IEnumerable<TSource> source, Func<TSource, TSource, TSource> func)

Example from LQP: Reduce error messages using string concatenation

```
string errorExample = string.Empty;
List<string> errors = ...
errorExample = errors.Aggregate(errorExample, (current, error) => current +
error + "\r\n");
```

Piping or |>

The piping operator, |>, allows chaining operations on lists:

```
let lst = [1..100]
let square x = x * x
let positive x = x >= 0
let result = List.map square lst
             |> List.map (fun x -> x - 5000)
             |> List.filter positive
```

```
Syntactic sugar


      let (|>) x f = f x


Thanks, John :-)
```

In C#:

```
var lst = Enumerable.Range(1, 100);
var result = lst.Select(i => i * i)
                .Select(i => i - 5000)
                .Where(i => i >= 0);
```

A common pattern in F# is for a function to accept as last element the list to operate on. This enables piping!

Seq

Both C# and F# are eager languages, hence we cannot easily define infinite lists for example.

Consider an “infinite” list implementation in C#:

```
public class InfiniteList
{
    private int _counter;

    public InfiniteList(int startIndex)
    {
        _counter = startIndex;
    }

    public IEnumerable<int> Next()
    {
        while (true)
        {
            yield return _counter;
            ++_counter;
        }
    }
}
```

```
var il = new InfiniteList(10);
foreach (var i in il.Next())
{
    Console.WriteLine(i);
}
```

This works because C# asks for new elements on demand or lazily!

As a matter of fact, the C# compiler will create a state machine internally to implement this.

This is how it remembers its state in Next().

Sequence Expressions

Seq is F#'s equivalent to IEnumerable<T> in C#. Seq allows for lazy collections of elements that are evaluated when needed.

Consider an infinite list of natural numbers:

Haskell:

```
seqn = [1..]
```

F#:

```
let inf_nat_numbers =
    let rec loop x = seq { yield x; yield! loop (x + 1); }
    loop 1

let seqn = inf_nat_numbers
```

Sequence Expressions

Consider a non-trivial example:

F#:

```
let tens_with_yield = seq { for i in 0..10..100 do
                                // yield the subsequence i, i+1, .., i+9
                                yield seq { i..1..i+9 } }
```

```
val tens_with_yield : seq<seq<int>>
```

```
let tens_with_yield_bang = seq { for i in 0..10..100 do
                                     // yield the subsequence i, i+1, .., i+9
                                     yield! seq { i..1..i+9 } }
```

```
val tens_with_yield_bang : seq<int>
```

C#:

```
// [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
var seq2 = Enumerable.Range(0, 10).Select(i => i * 10)
// [[0, 1, .., 9], [10, 11, .., 19], .., [90, .., 99]]
                     .Select(i => Enumerable.Range(i, 10).ToList());
```

```
// [0, 10, 20, 30, 40, 50, 60, 70, 80, 90]
var seq1 = Enumerable.Range(0, 10).Select(i => i * 10)
// [0, 1, .., 9, 10, 11, ..., 189]
                     .SelectMany(i => Enumerable.Range(i, 10));
```

We will learn later that IEnumerable<T> together with SelectMany is essentially the List monad.

Discriminated Unions

- Tagged union
- Type-safe variant of union

Example: **Option** type (Maybe in Haskell)

Type Constructor

Value Constructor

```
type Option<'a> =          // use a generic definition
   | Some of 'a            // valid value
   | None                  // missing
```

Option in LQP

```
public class Option<T>
{
    public enum Type
    {
        Some,
        None
    }

    #region Fields and Constants

    private readonly Type _type;

    private readonly T _value;

    #endregion

    public static Option<T> None()
    {
        return new Option<T>();
    }

    public static Option<T> Some(T value)
    {
        return new Option<T>(value);
    }

    #region Constructors

    private Option(T value)
    {
        _value = value;
        _type = Type.Some;
    }

    #endregion
```

```
    #region Public Properties

    public bool IsSome => _type == Type.Some;

    public T Unwrap
    {
        get
        {
            if (_type == Type.None)
            {
                throw new
InvalidOperationException();
            }
            return _value;
        }
    }

    #endregion
```

Option in LQP

Before:

```
public Config CurrentConfiguration
      {
          get
          {
              RegistryKey key = Registry.LocalMachine.OpenSubKey(WindowManager.Instance.App.RegistryPath);
              if (key != null)
              {
                  string val = key.GetValue("Configuration").ToString();
                  val = ConvertToProperCase(val);
                  if (val != string.Empty)
                  {
                      return (Config)(Enum.Parse(typeof(Config), val));
                  }
              }
              return Config.Customer;
          }
      }
```

After:

```
public Config CurrentConfiguration
{
    get
    {
        var option =
            Option.Lift(Registry.LocalMachine.OpenSubKey(WindowManager.Instance.App.RegistryPath))
                    .Map(key => key.GetValue("Configuration").ToString())
                    .Map(ConvertToProperCase)
                    .AndThen(s =>
                    {
                        Config type;
                        bool result = Enum.TryParse(s, true, out type);
                        return result ? Option.Lift(type) : Option<Config>.None();
                    });
        return option.IsSome ? option.Unwrap : DefaultUpdateTarget;
    }
}
```

Option in LQP

The type signature for Option.Map is interesting:

```
Option<TB> Map<TA, TB>(this Option<TA> option,
                         Func<TA, TB> action)
```

or decluttered:

$$Map :: F\ a \rightarrow (a \rightarrow b) \rightarrow F\ b$$

Or

$$Map :: (a \rightarrow b) \rightarrow F\ a \rightarrow F\ b$$

This means that Option is a **Functor** w.r.t. Map.

The type signature for Option.AndThen is also interesting:

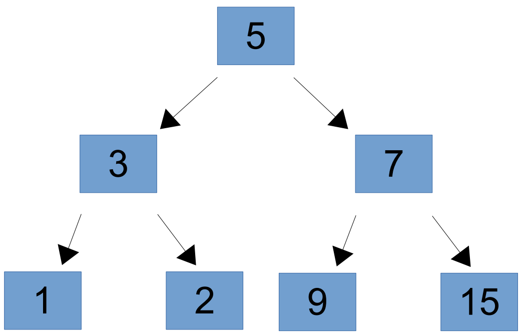
```
Option<TB> AndThen<TA, TB>(this Option<TA> option,
     Func<TA, Option<TB>> action)
```

Essentially, it is:

$$AndThen :: m\ a \rightarrow (a \rightarrow m\ b) \rightarrow m\ B$$

Later, we will learn that **Monads** have a special function called bind with that very type signature!

Example 1: Flatten a Tree




```
val it : int list = [1; 2; 3; 9; 15; 7; 5]
```

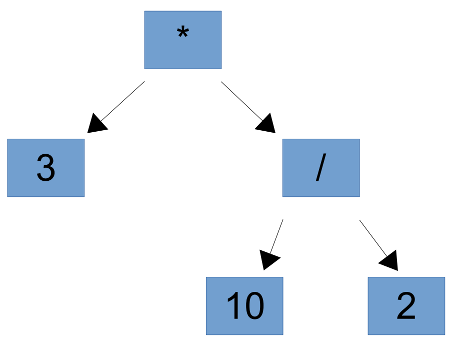
```
type Tree<'a> =
    | Leaf of 'a
    | Branch of Tree<'a> * 'a * Tree<'a>

let l3 = Branch (Leaf 1, 3, Leaf 2)
let r3 = Branch (Leaf 9, 7, Leaf 15)
let top = Branch (l3, 5, r3)

let fridge (t : Tree<'a>) : 'a list =
    let rec fr (lst : 'a list) (t : Tree<'a>) : 'a list =
        match t with
            | Leaf c -> c :: lst
            | Branch (l, c, r) ->
                let llst = fr lst l
                let rlst = fr lst r
                llst @ rlst @ [c]
    fr [] t

fridge top
```

Example 2: Evaluate Expressions




```
type Operation =
    | Add
    | Sub
    | Mul
    | Div

    member self.Eval (e1 : Expression) (e2 : Expression) =
        match self with
            | Add -> e1.Eval + e2.Eval
            | Sub -> e1.Eval - e2.Eval
            | Mul -> e1.Eval * e2.Eval
            | Div -> e1.Eval / e2.Eval

and Expression =
    | BinaryExpression of Expression * Operation * Expression
    | Constant of float

    member self.Eval : float =
        match self with
            | BinaryExpression(e1, op, e2) -> op.Eval e1 e2
            | Constant(c)                  -> c

[<EntryPoint>]
let main argv =
    (*
             *
            / \
           /   \
          3     :
               / \
              /   \
             10    2
    *)
    let expr = BinaryExpression(Constant(3.0), Mul, BinaryExpression(Constant(10.0), Div, Constant(2.0)))
    printfn "%A" expr.Eval

    0 // return an integer exit code
```

Compare this to the typical OO approach of using interfaces, derived classes and the like.
=> bloat!

Records

Recursion

- Basic Recursion
- Stack Records
- Stack Overflow
- Tail-Recursion Elimination
  - Accumulator pattern
  - Continuation pattern

Currying: Partial Function Application

- Eager
- Lazy