FUNCTIONAL PROGRAMMING WITH A MAINSTREAM LANGUAGE

Sadek Drobi Consultant and technology evangelist sadache@twitter

How does FP look like from enterprise

Introduction to the project's context

- Media planning software: Indicators Calculations
- Performance constrains
- Flexibility needed: more calculations need to be added easily and declaratively
- Any latency will be apparent?
- Productivity and interactivity of the GUI are crucial
- A lot of data on the screen with a lot of simultaneous calculations on each user interaction

How to approach performance in this case?

- Optimization of objects creation?
  - Reuse Data Structures
  - Use low level looping constructs and mutable arrays
- Functional approach with laziness?
  - No mutation
  - Use Streams as delayed lists
  - Compose functions for more modularity

Do it the Lean way

- Started two different strategies of implementation:
  - imperative with excessive use of loops and mutation
  - functional

What I am and what I am not!

- I am a mainstream OOP and imperative programmer: Java, C#
- I am not a functional programming geek: at least I wasn't prior to this experience
- All my FP knowledge dates back to university and school time: knowledge of Lisp that most of us acquired and forgot before stepping into enterprise
- I like to search for applying the suitable programming paradigm to the problem at hand

Do it the Lean way

- Started two different strategies of implementation:
  - imperative with excessive use of loops and mutation
  - functional

What did functional programming buy me more in this experiment?

Talking Paradigms: OOP vs. Streams

Talking Paradigms: OOP vs. Streams

- Streams as delayed lists
  - Modularity
  - Performance
- Immutability
  - Cuts down complexity enhancing readability : no state tracking
  - Your OO favorite language is optimized for object creation, let the garbage collector do its job!
  - Code safety
- OOP is great for encapsulation

Streams as delayed lists: Modularity

- A program needs to change state to be useful
- In a classical imperative approach, state is all over the place and the program consists of sequential changes of it
- With Streams, state is taken outside the program and gets passed through compound (composed) functions that operate on the stream to produce the result
- With long lists, and when you do not want to iterate the lists twice, delayed lists with list comprehensions give the opportunity to express logic modularly (partitioned into semantically distinct units)

Streams as delayed lists: Modularity

```
IEnumerable<Performance> CalculatePerformanceFor(IEnumerable<RawObject> rawObjects)
{
    return from r in rawObjects select CalculatePerformanceFor(r);
}
```

And in some other module…

```
IEnumerable<Network> ConstructNetworks(IEnumerable<RawObject> rawObjects)
{
    return from p in CalculatePerformanceFor(rawObjects) select new Network(names[p.Id], p);
}
```

yield …

Streams as delayed lists: Performance

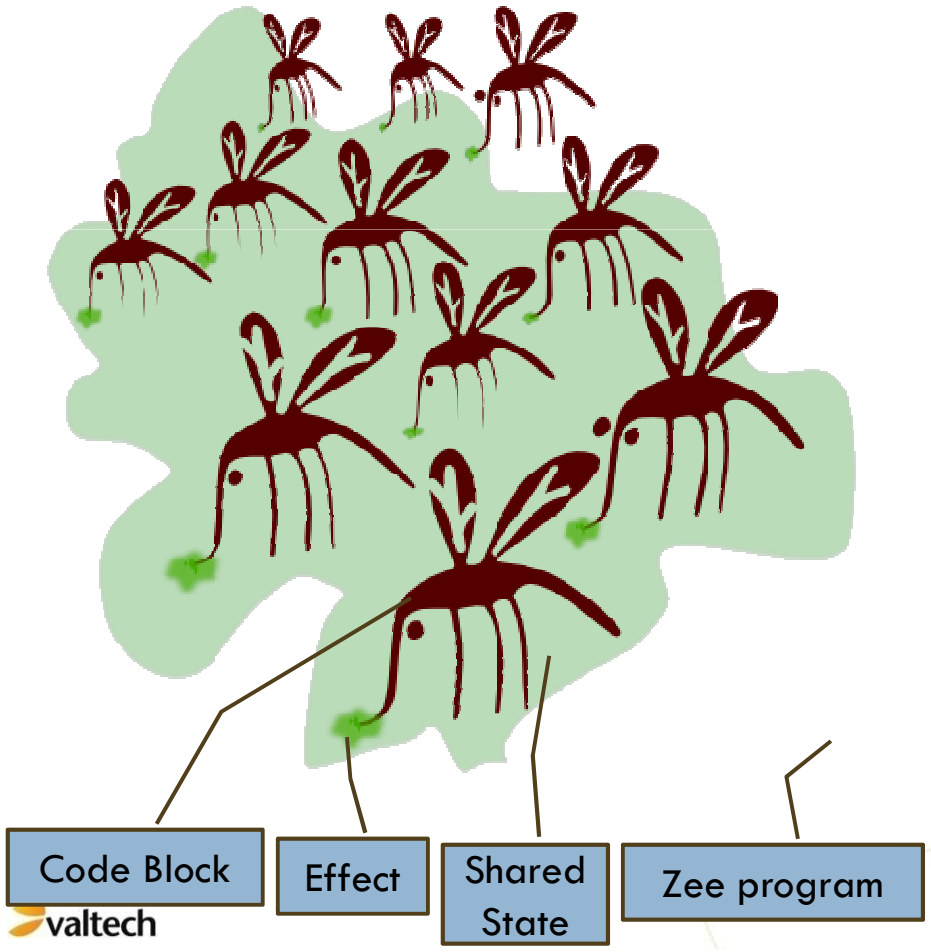
- No useless lists walkthroughs
- Quite tricky to choose where to be strict (.ToList())
- Yet it can be viewed as a decision that can be differed for later

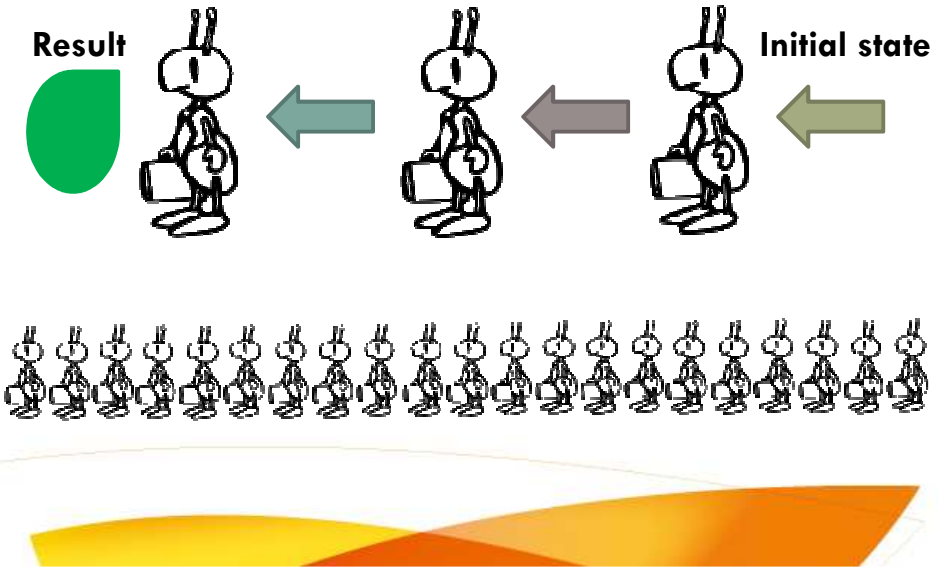
Immutability: Complexity Down, Enhance Readability

- No state tracking
  - Substitution model is far easier to reason about
    - Code turned to work correctly more often from the first time!
  - When correctly composing pure function, I can ignore completely semantics of both the functions and focus on semantics of the new function to go on. That never seemed to work when mutable objects are shared.
  - Shared mutable state cries for a debugger
  - State in not compositional

Immutability: Complexity Down, Enhance Readability

Mosquito Programming vs. Functional Programming





Immutability: Give Your GC Some Work

- Classes Vs. Objects : Procedural vs. OOP
  - In most E-Applications I see no OOP applied but procedural
- Share and Cache

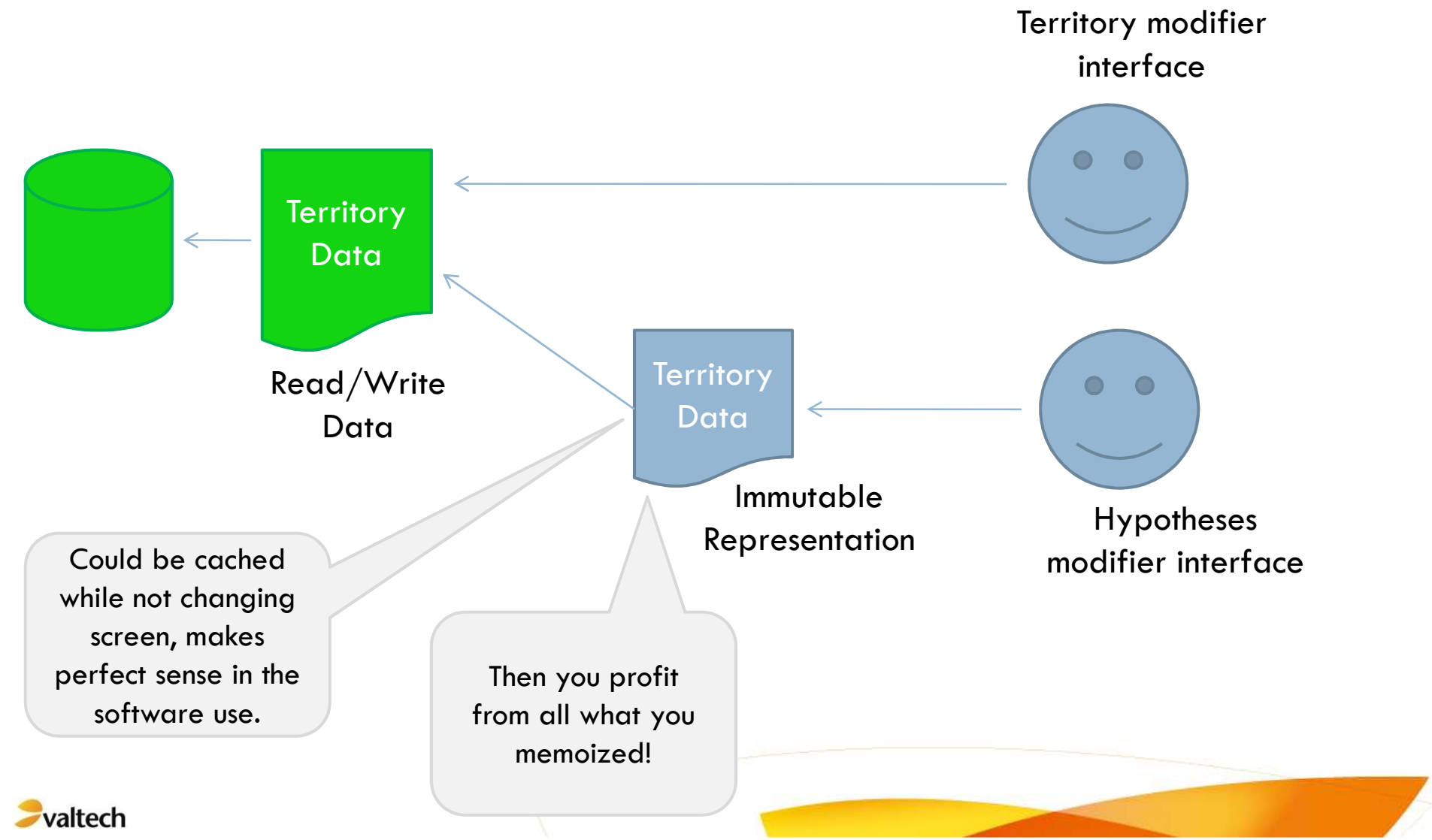
Immutability: Learnt to share

```
public class BaseNetwork
{
    public readonly IEnumerable<Repartition> Repartitions;
    public readonly double GrossRate;

    public BaseNetwork(IEnumerable<Repartition> repartitions, double grossRate)
    {
        Repartitions = repartitions;
        GrossRate = grossRate;
    }

    public readonly Func<int> FacesNumber;
    public readonly Func<double> RatePerFace;
    public BaseNetwork()
    {
        FacesNumber = F.Memorize(() => Repartitions.Sum(r=>r.FacesNumber));
        RatePerFace = F.Memorize(() => GrossRate / FacesNumber());
    }
}
```

Immutability: Caching, Finely optimized for context

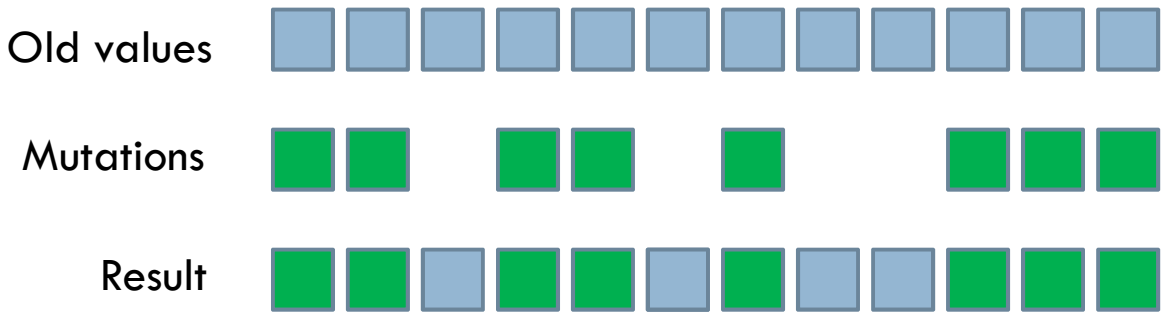
- Data retrieved from database don't need to be mutable all over the application even if they are modifiable in some contexts
- Data Views

Caching: Finely optimized for context



Immutability: Code safety

- Code Safety
  - Reusing mutable structure can have the effect of representing old obsolete values as current
  - WPF example



OOP: Encapsulation

- Immutability doesn't mean abandoning object orientation
- OOP encapsulation for better code organization
- With immutability, all object methods can be memoized if needed, this is interesting especially when sharing instances

OOP: Encapsulation

```
IEnumerable<KeyValuePair<NetworkIdentifier,double>> Calculate1(IEnumerable<BaseNetwork> networks,double some_n)
{
    return from n in networks
           let totalFacesNumber = n.Repartitions.Sum(r => r.FacesNumber)
           let ratePerFace = n.GrossRate/totalFacesNumber
           select new KeyValuePair<NetworkIdentifier, double>(n.Id, ratePerFace * some_n);
}
IEnumerable<KeyValuePair<NetworkIdentifier, double>> Calculate2(IEnumerable<BaseNetwork> networks, double some_n)
{
    return from n in networks
           select new KeyValuePair<NetworkIdentifier, double>(n.Id, n.RatePerFace() * some_n);
}
```

Functions as First Class Values

Functions as First Class Values

- Mutable State Vs. Closures and Partial Application
- Presenter return actions to be executed on the view
- Continuation monad
  - More interface responsiveness
  - Less apparent latency
- AOP with no framework (Memoize)
- With Functions as First Class Values a lot of Design Patterns become obsolete

Closures and Partial Functions Application

- Being immutable everywhere, you will be faced sometimes a situation where you have different parameters values of a function in different scopes
- Yet you want to stay immutable and modular!
- Partial application supports your modular design

Closures and Partial Functions Application

In some module we have

```
SomeResult GetSomeResult(int totalFacesNumberOnT,NetworkIdentifier nId, int networkFacesNumberOnT)...
```

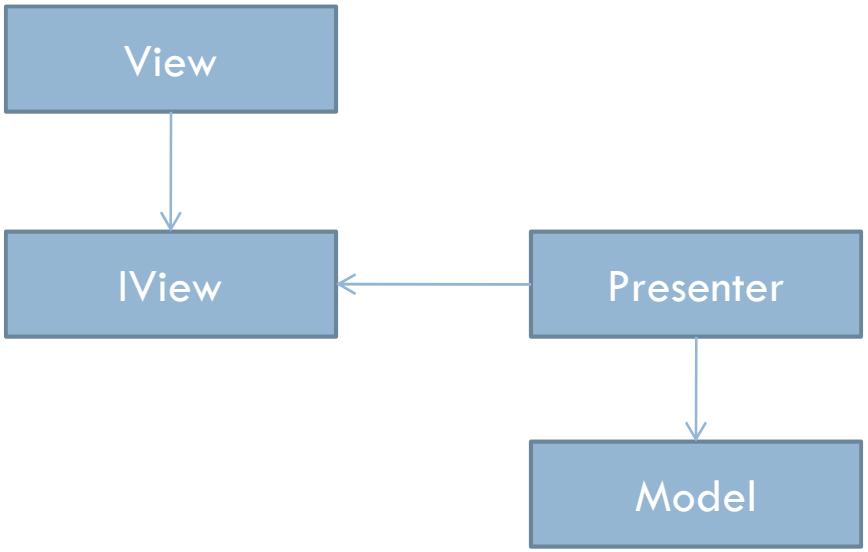
And in some other

```
SomeOtherResult DoSomeCalculations(int totalFacesNumberOnT)
{
    Func<NetworkIdentifier, int, SomeResult> calculateP = (id, nfOnT) => GetSomeResult(totalFacesNumberOnT, id, nfOnT);
```

Then this could be passed to yet another!

MVP the Functional way

- Model View Presenter



MVP the Functional way

- Presenter as a service

```
Action<IShowAvailableNetworksView> ChangeTarifNetOfNetworks(IEnumerable<Network> localNetworks, IEnumerable<Networ
{
    var changedLocalNetworks = service.ChangeTarifNetOfNetworks(tauxNego, localNetworks);
    var changedNationalNetworks = service.ChangeTarifNetOfNetworks(tauxNego, nationalNetworks);

    return v =>
    {
        v.NationalAvailableNetworks = changedNationalNetworks;
        v.LocalAvailableNetworks = changedLocalNetworks;
    };
}
```

```
this.ExecuteOrScheduleOnceForWhenVisible("ChangeTarifNetData",
    () =>
    Presenter.ChangeTarifNetOfNetworks(localNetworks, nationalNetworks, tauxNego));
```

WPF Monad

- WPF and threads
  - Graphical components are not truthful about their types
  - When threads are engaged, they no longer present their type
  - View<ContractType>
  - from v in view …
  - Threads logic and freezing is done by the monad
  - Unified syntax vs. Special syntax or DSLs
  - Same could be done for exceptions

WPF Monad

```
public interface ISayHello
{
    Unit SayHello(string Name);
    string GetName();
}

View<ISayHello> view = this.AsView<Window1, ISayHello>();

from v in view
let name = v.GetName()
select v.SayHello(name)).Do();
```

Quite convenient to use several views in one expression:

```
from v1 in view1
from v2 in view2
```

WPF Monad: Implementation

```
public delegate R View<R>();

public static View<TView> AsView<TWPF, TView>(this TWPF value) where TWPF : UIElement, TView
{
    return value.ToWpfMonad<TWPF, TView>();
}

public static View<Answer> ToWpfMonad<T, Answer>(this T value)
   where T : UIElement, Answer
{
    return () =>
    {
        Answer a = default(Answer);
        value.Dispatcher.Invoke(DispatcherPriority.Normal, (EventHandler)((sender, e) =>
        {
            a = value;
            if (a is Freezable)
            {
                var result = ((Freezable)(object)a).Clone();
                a = (Answer)(object)result;
                result.Freeze();
            }
        }), null, null);

        return a;
    };
}
```

WPF Monad: Implementation

```
public static View<U> SelectMany<T, U>(this View<T> m, Func<T, View<U>> k)
{
    return () => k(m())();
}
public static View<U> Select<T, U>(this View<T> m, Func<T, U> selector)
{
    return () => selector(m());
}
public static View<V> SelectMany<T, U, V>(this View<T> source, Func<T, View<U>> kSelector, Func<T, U, V> resultSelector)
{
    return () =>
            {
                var t = source();
                var u = kSelector(t)();
                return resultSelector(t, u);
            };
}
```

WPF Monad

- WPF and threads
  - Graphical components are not truthful about their types
  - When threads are engaged, they no longer present their type
  - View<ContractType>
  - from v in view …
  - Threads logic and freezing is done by the monad
  - Unified syntax vs. Special syntax or DSLs
  - Same could be done for exceptions

Design Patterns

- Most GOF Design Patterns are not of a great values with the existence of a higher order functions (closure)
- Lambda expressions are very easy to create at call site and are quite expressive
- Polymorphism is hard to reason about
- Functions are compositional

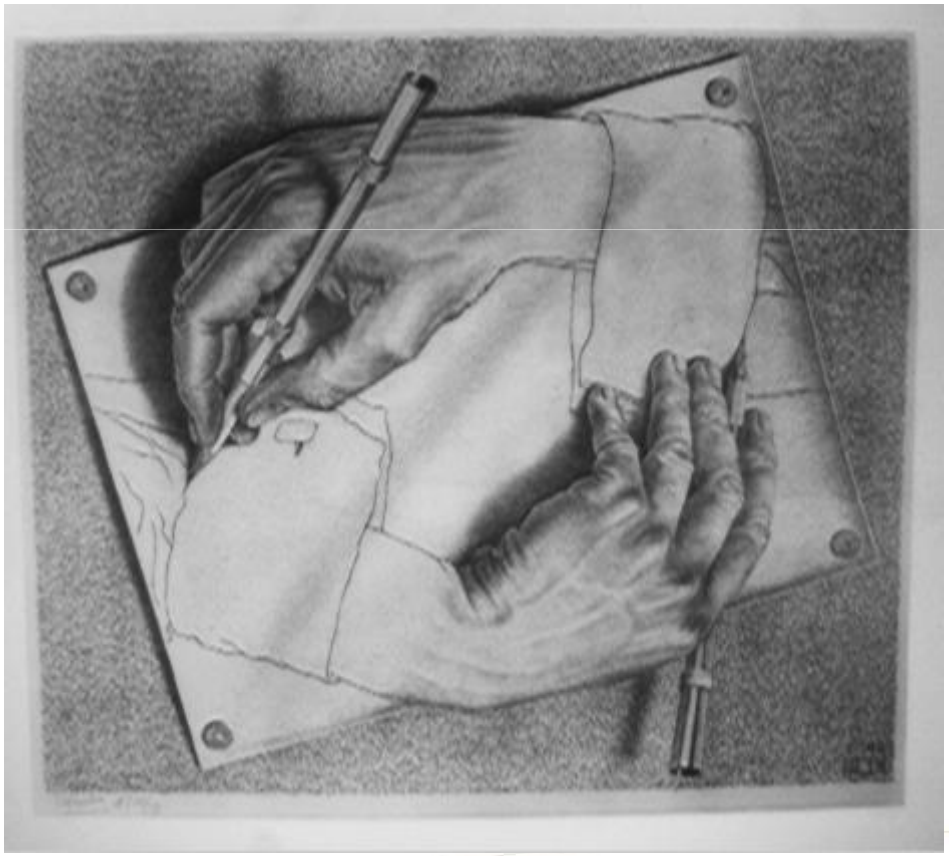
Recursion, costly but clearer and more readable?

Recursion, costly but clearer and more readable?

- Recursive calls are often more expressive
- Not optimized C# (tail recursion)
  - Use *fold* and *map*
  - Memoize it because you are pure!

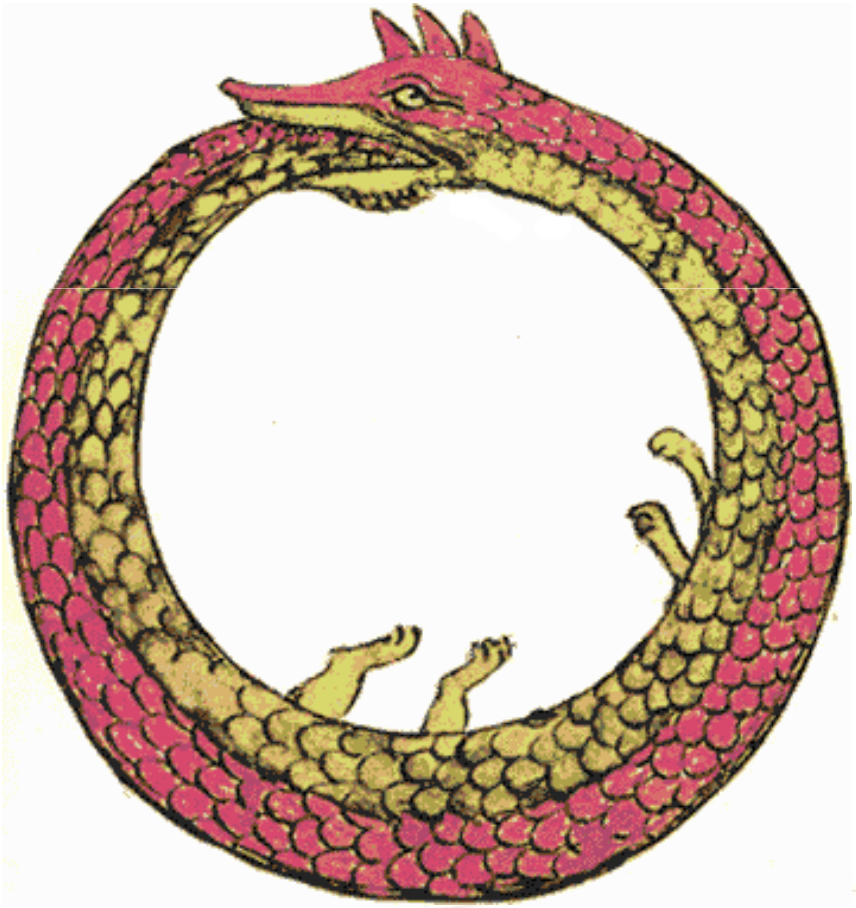
Expressiveness of recursive calls

- They are often more expressive



Not optimized in C#

- Tail recursion



Not optimized in mainstream languages

- Use fold and map
  - Select and Aggregate
  - Abstractions of some recursive forms that help being declarative without sacrificing performance

Not optimized in mainstream languages

- Memoize it because you are pure!
  - Memoize can be introduced as an aspect
    - Interchange MemoizeFix and Y for performance tuning

Another side effect of purity: order does not matter

Another side effect of purity: order does not matter

- Future<T>
- More processors? No problem!

Purity: Future<T>

```
var populationTotal=
        Future<int>.Create(
            ()=>(from t in territoryBaseGeos.Distinct()
                 select t.GeoPopulation).Sum());

some other work and calculations

var pnxGFTotalTerritoire =
        Future.Create(
            ()=>(from g in repartitionsThatMatchTheTerritory
                 from m in g
                 select m.Left)
                    .Where(r=>r.Network.SousUniverId == NetworkReferentiel.PosterSousUnivers.GRAND_FORMAT)
                    .Sum(r=>r.FacesNumberByBaseGeography));

var offreTotalTerritoire =
        Future<double>.Create(
            ()=>(from gm in repartitionsThatMatchTheTerritory
                 select CalculateOffreReseauxOnTerritory(gm.Key, gm.Select(m => m.Left))).Sum());

var globals = new GlobalHypotheseValues(populationTotal.Value,
                                        pnxGFTotalTerritoire.Value,
                                        offreTotalTerritoire.Value);
```

Purity: More Processors

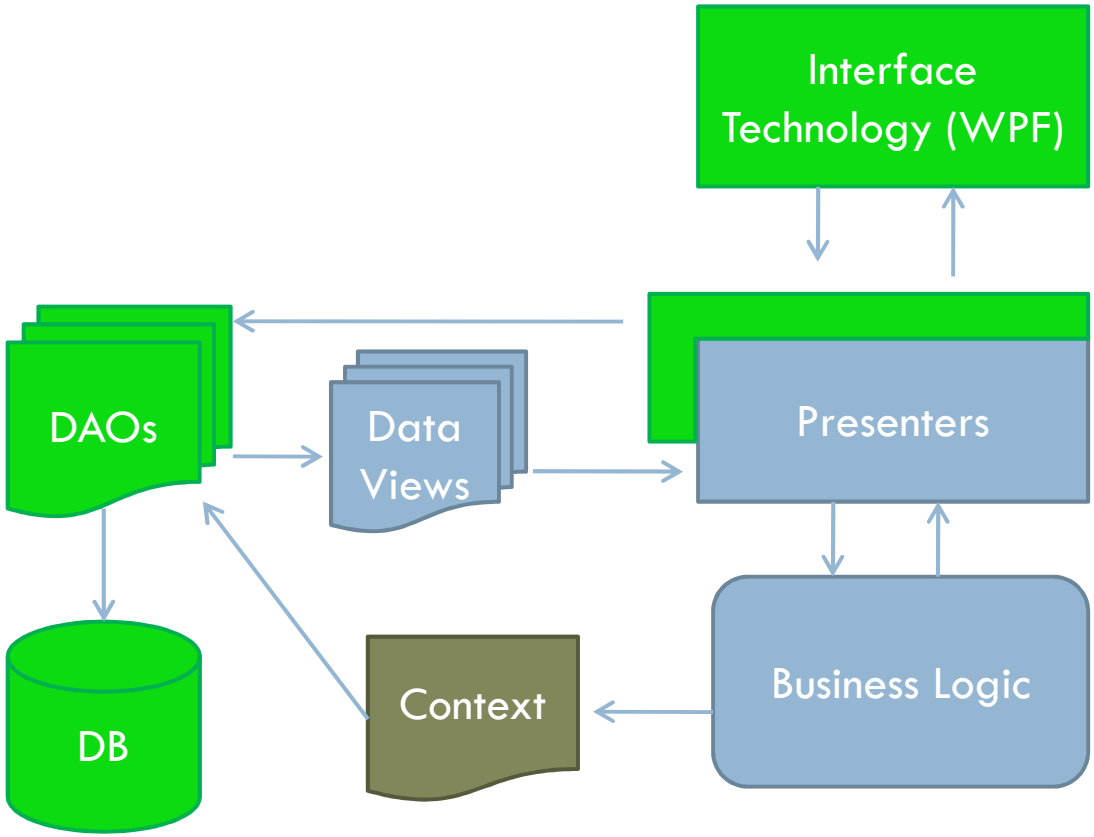
- Parallelize it, you are pure
  - LinQ .AsParallel()
- Or are you?
  - Failed on first shared mutable state
  - Need locks in Memoize

Purity: More Processors

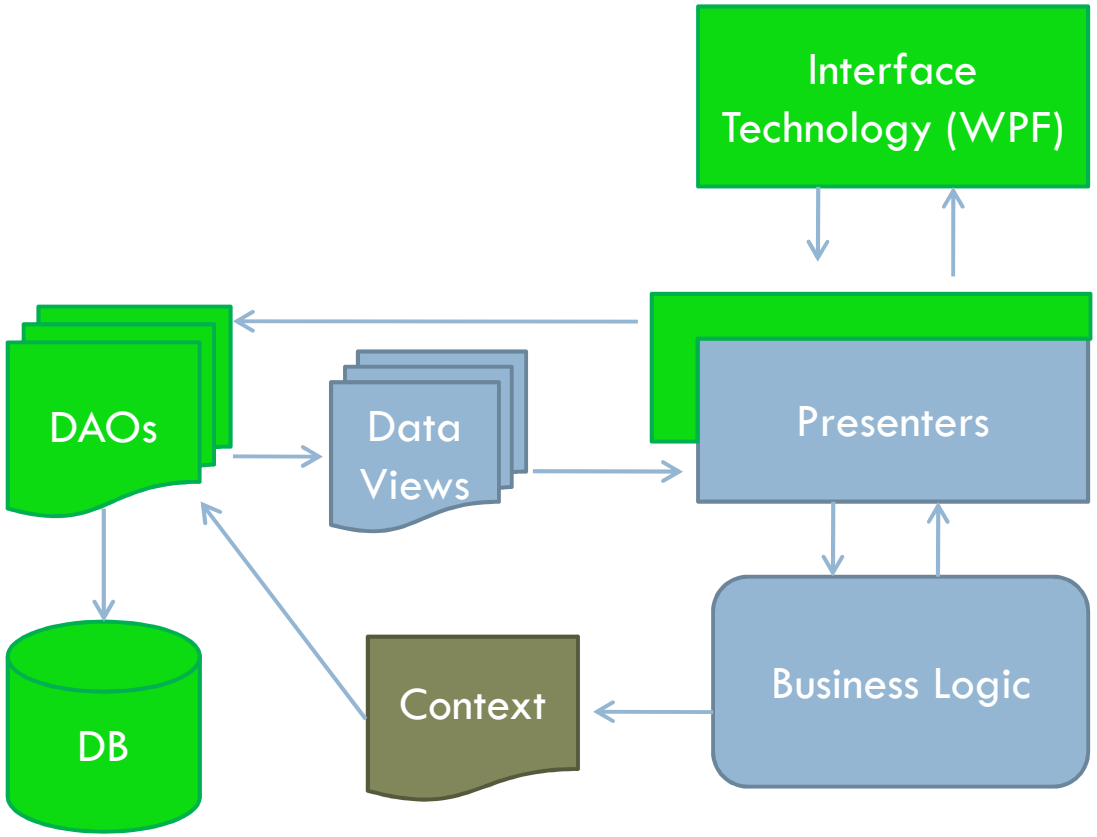
```
public static Func<A,U, R> Memorize<A,U, R>(this Func<A,U, R> f)
{
    var map = DictionaryHelper<R>.CreateDictionary(new { a = default(A), u = default(U) });
    return (a,u) =>
    {
        R value;
        if (map.TryGetValue(new {a = a,u = u},out value))
            return value;
        else lock(map)
        {
            if (map.TryGetValue(new { a = a, u = u }, out value))
                return value;
            else
            {
                value = f(a,u);
                map.Add(new {a = a,u = u}, value);
            }
        }

        return value;
    };
}
```

Architecture View

Architectural View



Architectural View



In FP Terms:

Is IO or "Effect"

Is pure

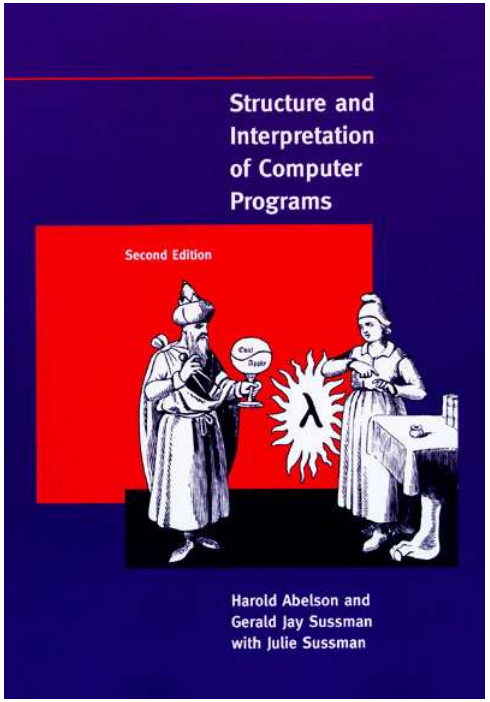
Overview

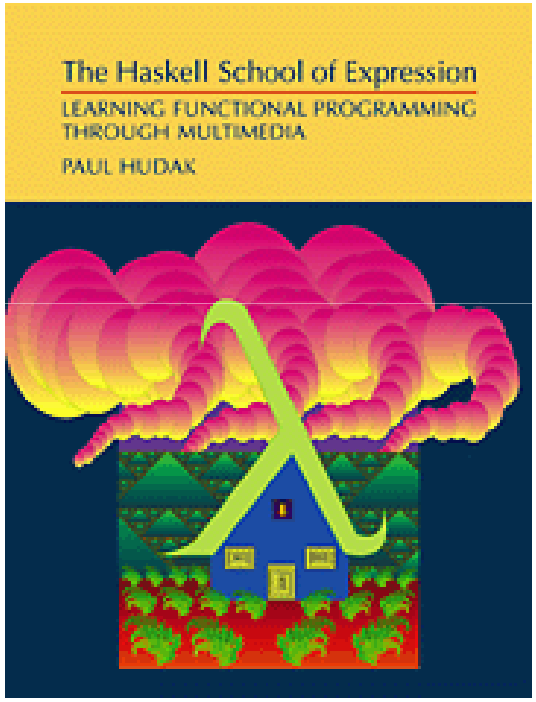
- Mutability is addictive and effects are like cancer
- Imperative programming is very tempting, only discipline can help in a mixed paradigm environment
- Less is more, FP simplicity is key to productivity

Regrets

- Function types are ugly without type inference
- No generic local values and partial type constructor application
- Laziness unleashes evil! Keep attention
  - No checked exception
  - Effects not expressed in the type system
- Null everywhere is a big source of bugs
- We might be not doing too bad about Structure Abstraction (especially hierarchical) but we do no Computation Abstraction
  - Null, exceptions, gui main thread, delays…

Inspiration





See you around

- Sadache@Twitter
- www.sadekdrobi.com
- contact@sadekdrobi.com