Introduction to Monads

Your Everyday Chainable Decorators

2013-10-03

sergiotaborda@zbra.com.br

Contents

- Brief historical introduction
- What are Monads ?
  - Monadic Laws
- What are Monads in OO ?
- Monads and LINQ
- Fundamental Monads
  - Identity Monad
  - Collection Monad
  - Maybe Monad
  - Writer Monad
- Other Monads

Brief historical introduction

- Philosophy (a.c.) – The Unit. The One "essence' that generates all others. Belongs to a category of "generators" like the Dyad and Triad, etc..
- Algebra (1964)- A 3-set construct with mappings between them (A → B → C).
- Category Theory – The Composition of two adjoint *Functors (GoF)*
- Computer Science (1980)- Created in Opal and then used in Haskell (~1990) that where functional languages. Nowadays used in OO languages with functional support like Scala and C#

What Are Monads

- 3 = 1 + 2
- Box(3) = Box (1) + Box (2)
- Box(3) = Box (1) (+) Box (2)
- Box(3) = ● (Box (1) , Box (2) , (x, y) => x+y)
- Box (3) = ● (Box (1) , Box (2) , Func (x,y))

- Unit(T obj) : Box<T>
- Fmap(Func<T,U,R> g) : Func<Box<T>,Box<U>,Box<R>>

What are Monads

- Are Types with
  - A method, Unit, to augment other objects.
  - A method, Fmap, to transform a function to another function (Fmap is the equivalent of a Functor)
- Or
  - A method, Fmap, to transform a function to another function
  - A method, Join , to de-augment a double augmented type
- Or
  - A method, Unit, to augment other objects in it.
  - A method, Bind, to operate between two object of the same monad

What are Monads

- Fmap(f)(m) ⬄ Bind(m, f)
- Join(m) ⬄ Bind(m, I)
- Bind(m, f) ⬄ Join(Fmap(f)(m))

- where I(y) = y
- f is a function over x
- m the monad value

Monadic Laws

1. Left Identity

   Bind(Unit(x), f) ⇔ f (x)

2. Right Identity

   Bind(m , Unit) ⇔ m

3. Associativity

   m = Unit (x)

   Bind(Bind(m , f), g) ⇔ g(f(x))

That is a Monad !?

What is a Decorator

- It’s a design pattern
- Provides an object type with new methods
- Normally wraps the original object and even operates over it.

What is a Chainable Decorator

- It’s a Decorator with operations that return the same Decorator type allowing to invoke it again

```
decorator.x().y().z()
```

- Each invocation returns a new, immutable object

What are Monads in OO

- Are Chainable Decorator Types with
  - A method, Unit, to wrap (augment) other objects.
  - A method, Bind to operate over other augmented types
- And
  - Additional methods to operate with regardless of the augmented object

Monads and C#

- C# Constructor and/or Extensions features support the implementation of the Unit Monad method.

```
public static Box<T> ToBox<T>(this T obj)
{
        return new Box<T>(obj);
}
```

- This is the expected construction for a decorator factory method
- Typing the extension method allows for polymorphic monad construction

Monads and C#

- Typing the extension method allows for polymorphic monad construction

```
public static Maybe<S> ToMaybe<S>(this Nullable<S> value)
where S : struct
{
        return !value.HasValue
                 ? Maybe<S>.Nothing
                 : new Maybe<S>(value.Value);
}

public static Maybe<string> ToMaybe(this string value)
{
        return string.IsNullOrEmpty(value)
                 ? Maybe<string>.Nothing
                 : new Maybe<string>(value);

}
```

Monad and LINQ

- LINQ (Language Integrated Query) is a general, agnostic mechanics to support the Monad concept
- Any type that implements (directly or by extension) the SelectMany method can be used with LINQ. SelectMany is the Monad Bind operation

  ```
  Box<T> SelectMany<T,U,V> (this Box<T> m , Func<T, M<T>> k,
  Func<T,U,V> s)
  ```

- Then you may write:

```
var result = from x in 1.ToBox()
             from y in 2.ToBox()
             select (x + y);
```

This is called do-notation *(or* computation expression *in F#)*

Fundamental Monads

- Identity – just augments a value. No special operations.
- Maybe – keeps track of the absence of value. A maybe object can be a Nothing or a Just. It allows to compute complex chained expressions that can handle Nothing (null) automatically without exceptions.
- Collection – keeps track of a set of objects. It allows for bulk chained operations like adding , removing, filtering, mapping, etc…
- Writer– allows to write to another object in the “background” while executing a complex chained expression.

Identity Monad

- Simply augments a type to another
- The Box type used in the examples is really and implementation of the Identity Monad
- Using a computation expression is possible to operate with the augment objects.

Identity Monad and C#

```
public class Box<T> {
         public property Value {get; private set;}
         protected Box(T x)
         {
                 this.Value = x;
         }
}
public static class BoxExtentions{
         // Unit
         public static Box<T> ToBox (this T x ){
                 return new Box(x);
         }
         // Bind
         public static Box<R> SelectMany(this Box<T> box, Func<T, Box<R>> f)
         {
                 return f(box.Value)
          }
         …
}
```

Identity Monad em C#

```
public static class BoxExtentions{
           // Unit
           public static Box<T> ToBox (this T x ){
                   return new Box(x);
           }
          // Bind
          public static Box<R> SelectMany(this Box<T> box, Func<T, Box<R>> f)
          {
                  return f(box.Value)
           }
          // Bind
          public static Box<R> SelectMany(this Box<T> m, Func<T, Box<V>> f,
                                 Func < T, R, V> s
          {
                     return m.SelectMany(x => k(x)
                               .SelectMany(y => s(x, y).ToBox()));
          }
}
```

Identity Monad em C#

```
Box<int> three = from x in 1.ToBox()
                 from y in 2.ToBox()
                 select (x + y);
// Or
Box<int> three =  1.ToBox().SelectMany( x => 2.ToBox(), (x,y) =>x+y );
```

Collection Monad

- Binds together objects in a set
- Allows for bulk operations on every element of the set (foreach)
- Allows for operations on the set like, filtering, ordering, mapping, etc…
- IEnumerable<T> is an implementation of the Collection Monad
- Concat is the special operation that adds another element to the collection

Collection Monad

```
public static class CollectionExtentions{
        // Unit – any object
        public static Bag<T> ToBag(this T x ){
              return new Bag().Add(x);
        }
        // add another element
        public static  Bag<T> Concat(this Bag<T> bag, T x)
       {
            return bag.Add(x);
       }

}
```

Maybe Monad

- Allows for chaining complex expressions always keeping track of absent values.
  - The absent value is named Nothing.
  - The not absente value is named Just
- In practice allows to execute always valid operations even in the presence of absent values.
- Special methods are:
  - `Or` - Translates the Maybe to its value , or a default value
  - `Select` – Operates over the inner object and returns a Maybe
- Nullable<S> in C# is an implementation of the Maybe Monad with special support in the compiler. But only can be used for struts. No special methods are provided

Maybe Monad

```
public static class MaybeExtentions{
        // Unit – any object
        public static Maybe<T> ToMaybe (this T value ){
                return value == null ? Maybe<T>.Nothing : new Maybe<T>(value);
        }
        // Unit - string
        public static Maybe<string> ToMaybe(this string value)
        {
                return string.IsNullOrEmpty(value)
                ? Maybe<string>.Nothing
                : new Maybe<string>(value);
        }
        // Unit - Nullable
        public static Maybe<S> ToMaybe<S>(this Nullable<S> value)
        where S : struct
        {
                return !value.HasValue
                ? Maybe<S>.Nothing
                : new Maybe<S>(value.Value);
        }
        …
}
```

Maybe Monad

```
public static class MaybeExtentions{
        …
    // Or - Reduce to value
    public static T Or (this Maybe<T> x , T defaultValue)
    {
        return x.HasValue ? x.Value : defaultValue;
    }
    // Select
    public static Maybe<V> Select<T,V>(this Maybe<T> x, Func<T,V> k)
    {
        return  !x.HasValue
        ? Maybe<V>.Nothing
        : k(m.Value).ToMaybe()
    }
    …
}
```

Nullable Extentions

```
public static class NullableExtentions{
        …
    // Or - Reduce to value
    public static S Or (this Nullable<S> x , S defaultValue)
        where S : struct
    {
        return x.HasValue ? x.Value : defaultValue;
    }
    // Select
    public static Nullable<V> Select<T,V>(this  Nullable<S> x,
                Func<S,Nullable<V>> k )
        where S : struct
        where V : struct
    {
        return  !x.HasValue
        ? (Nullable<V>)null
        : k(m.Value)
    }
    …
}
```

To Maybe or not To Maybe

```
public InitView(){
  int? id= View.GetId();
  var client;
  if ( id.HasValue)
  {
       client = ServiceSearch(id);
  }
  else
  {
      client = new Client();
   }
   View.Show(client);
}
```

```
public InitView(){
  var client = View.GetId().Select( id =>ServiceSearch(id)).Or(new Client());
   View.Show(client);
}
```

```
public InitView(){
  var client = View.GetId().AlsoNothing( id => id <= 0)
       .Select( id =>ServiceSearch(id)).Or(new Client());
   View.Show(client);
}
```

To Maybe or not to Maybe

```
 public bool IsReadOnly()
{
      var isReadOnlyString = Request["isReadOnly"];
      if (string.IsNullOrEmpty(isReadOnlyString))
          return false;
      bool isReadOnly;

return bool.TryParse(isReadOnlyString, out isReadOnly) ? isReadOnly : false;

 }

public bool IsReadOnly()
{
      return  Request["isReadOnly"].ToMaybe().Convert<bool>().Or(false);
}
```

To Maybe or not to Maybe

```
public  Money  Tax( Money base,  Fraction taxInterest)
{
          if (base != null && taxInterest != null)
         {
               return money * taxInterest;
         }
         else
        {
              return  ??????
        }
 }
```

To Maybe or not to Maybe

```
public  Maybe<Money>  Tax( Maybe<Money> base,
Maybe<Fraction> taxInterest)
{    // do -notation
    return  from x in base
               from y in taxInterest
               select ( x * y);
 }
public  Maybe<Money>  Tax( Maybe<Money> base,
Maybe<Fraction> taxInterest)
{    //  explict linq
    return  base.SelectMany (  base => taxInterest ,  (base,
taxInterest) => base * taxInterest );
 }
```

Writer Monad

- Allows to write to a “background” object while executing chained operations
- Originally used for file writing I/O operations, logging or debugging (as functions cannot have secondary effects)
- Today is the basis for a robust implementations of the Builder pattern (specially when fluent interface is used) and simple Domains Specific Languages (DSL)
- LINQ to Relational Data uses this monad to write to Expression objects (with the help of the compiler) that are then run thought an Interpreter (the LINQ Provider)

Writer Monad

```
public class Client {

    public string Name {get;set;}
    public string Addres {get;set;}

    public override int GetHashCode()
    {
        return Name.GetHashCode()* 17
                    + Address.GetHashCode();
    }
}
```

Writer Monad

```
public class Hash {
     private int hash;
     public Hash  (int hash){
          this.hash = hash;
     }
     // get value
     public override int GetHashCode()
     {
       return hash;
     }
     … // equals
}
```

Writer Monad

```
public static class HashExtentions{
         private static readonly int prime = 17

         // Unit - any object
         public static Hash ToHash (this object value ){
                return new Hash(value.GetHashCode())
         }

         // Add another object to the hash
         public static Hash Concat(this  Hash  value, object other )
         {
              return other == null
                        ? value
                        : new Hash( // the composite hash rule
                                    value.GetHashCode() * prime + other.GetHashCode()
                              );
         }

         // Add a collection of objects
         public static Hash Concat(this Hash value, IEnumerable<object> others)
         {
              var result = value;
              foreach ( T element in others)
               {
                result = result.Concat(element);
              }
             return result;
         }
}
```

Writer Monad

```
public class Client {

   public string Name {get;set;}
   public string Addres {get;set;}

   public override int GetHashCode()
   {
       return Name.ToHash()
                   .Concat(Address)
                   .GetHashCode();
   }
}
```

Writer Monad

```
public class Client {

   public string Name {get;set;}
   public string Addres {get;set;}

   public override int GetHashCode()
   {
      return from x in Name.ToHash()
             from y in Address.ToHash()
             select (x * 17 + y)
   }
}
```

Other Monads

- Reader (aka Environment) – The counterpart of the Writer Monad. Allows for accessing values in the chain from outside the original augmented value

```
string userName = user.ToReader()
         .Select ( (ctx, user) => ctx.IsUserAutenticated(user)
                          ? user.Name
                          : “Guest” )
         .RunIn(appcontext);
```

- State – Allows for tracking state in a chained operation.

```
 var state = microwave.ToState(State.Closed)
          .Open()
          .Put(meal)
          .SetTimer(2, Time.Minutes);
 // changes the door state from Closed to Open
// changes the timer state from NotSet to Set
if (state.IsDoorClosed() && state.IsTimerSet()){){
    microwave.Start();
}
```

- Continuation* – Allows for postponing operations. Useful in the presence of distributed/multithreaded operations.

```
list.Do(list => list.Sum()).ContinueWith( sum => label.Text = sum)
```

Use Monads

- They are a special type of Decorator that
  - augment code and types capabilities
  - simplify code by encapsulating rules
  - reduce code an increase readability as they are chainable and strong typed
- Use Maybe Monad
  - to remove `null` checks and `NullReferenceException`
  - to simplify chained calculations that compute correctly even in the absence of some values
  - Reduces decisions around `null` (use of `if` and ternary conditions) increasing test coverage
- Use Write
  - to construct fluent powerful Builders
  - connect with LINQ and Expression
- Use Collection
  - to operate in bulk (for each, filter, map)
  - to do aggregations (sum , avg, reduce)
- Mix them up
  - A collection of maybe
  - Filtering a collections with a writer

References

- Code Rant – Monads in C#
  http://mikehadlow.blogspot.co.uk/2011/01/monads-in-c1-introduction.html
- A Fistfull of Monads
  http://learnyouahaskell.com/a-fistful-of-monads
- Monad (functional programming)
  http://en.wikipedia.org/wiki/Monad_(functional_programming)
- LINQ to Relational Data
  http://msdn.microsoft.com/en-us/library/cc161164.aspx
- Sprache – C# parser toolkit
  https://github.com/sprache/Sprache
- Monads for normal Programmers
  http://blog.jorgenschaefer.de/2013/01/monads-for-normal-programmers.html

References

- Reader Monad in C#
  https://gist.github.com/vmarquez/4640678
- Continuations by example: Exceptions, time-traveling search, generators, threads, and co-routines
- http://matt.might.net/articles/programming-with-continuations--exceptions-backtracking-search-threads-generators-coroutines/