# Introduction

For those that are not aware or familiar with Rx, I’ll repeat the quote from the MSDN site;

“The Reactive Extensions (Rx) is a library for composing asynchronous and event-based programs using observable sequences and LINQ-style query operators.”

Things to bear in mind there; asynchronous, observable, sequence and queryable.

Anyone not familiar with the Observer pattern? Its one of the Gang of Four’s patterns and as it turns out it is the dual of IEnumerable.

This talk is going to be in 3 parts.

* The first part will be re-examining the familiar IEnumerable interface and redefining it functionally.
* Part 2 will take this functional view of IEnumerable and mechanically invert it’s semantics to derive a functional view of IObservable. We will then discover its shortcoming and try and compensate.
* Part 3 will be the reverse of part 1, we will be rediscovering the interface based representation of the functional view. We will then go on and try to demonstrate the power of Rx.
* Part 4 a few demos to show the power of (our homebrew) Rx.

I hope that by the end of the talk you will see the ‘obviousness’ of Rx after seeing how it can be mechanically derived from very familiar ground. And assuming time permits I hope I can demonstrate how powerful it can be.

This talk is not intended to make you an Rx guru but really just try and explain some of the mechanics.

# Part 1. IEnumerable<T>… Functionally.

Decompile or something to get to this, removing Reset and non-generic stuff.

public interface IEnumerable2<out T>

{

IEnumerator2<T> GetEnumerator();

}

public interface IEnumerator2<out T> /\*: IDisposable\*/ //TODO come back to me!!!

{

T Current { get; }

bool MoveNext();

}

“Lets break this down into a functional representation. IEnumerable has a single method we can call which gives us back an IEnumerator.”

“That looks like Func<???> where the Func is representing GetEnumerator.”

“So looking at IEnumerator itself. What is it actually telling us? It has a method that you can call to see if there is another value for you, and a way to get to it. As we are working functionally right now we need to represent that as a single function. Kind of sounds like a something or nothing doesn’t it? Something or nothing is an example of a discriminated union type, C# doesn’t do a great job with that, unlike functional languages which usually have built in support. Let me just hack up something we’ll use in place”.

Add NotificationType with Next and Complete. Add Notification<T> with ctors.

“Now we can get back to our functional interface representation: “

Func<Func<Notification<T>>

“This signature says, Give me a function I can repeatedly call and get a value or completion”.

“GET something from which I can GET either values or completion”.

Lets’ go over the existing way you might iterate a sequence of integers:

var ints = Enumerable.Range(0, 10);

foreach (var i in ints)

{

Console.WriteLine(i);

}

var enumerator = ints.GetEnumerator();

while (enumerator.MoveNext())

{

Console.WriteLine(enumerator.Current);

}

“So now, what does this look like with our functional definition?”

When building out the following add the non-generic Notification class. Note that Notification.Return I type-mumbling.

public static Func<Func<Notification<int>>> Range(int start, int count)

{

return (()=>

{

var i = 0;

return () => i < count ? Notification.Next(start + i++) : Notification.Complete<int>();

});

}

And on the enumerating side:

var enumerable2 = Enumerable2.Range(0, 10);

var enumerator2 = enumerable2();

var r = enumerator2();

while (r.NotificationType != NotificationType.Complete)

{

Console.WriteLine(r.Next);

r = enumerator2();

}

“See how this sits together? First we grab the enumerable by calling range. We call GetEnumerator by invoking the outer Func, on the implementation side this allows us to set up any state for the enumeration, in this case that’s starting a counter. We then call MoveNext by invoking the enumerator and putting he value in r. r contains our something or nothing and we can use this to determine when we are done.”

“There’s a lot of code to enumerate our new definition. We can do better by introducing a ForEach method to do this leg work for us, much like the compiler does for us over IEnumerable”

public static void ForEach<T>(this Func<Func<Notification<T>>> enumerable2, Action<T> action)

{

var enumerator2 = enumerable2();

var r = enumerator2();

while (r.NotificationType != NotificationType.Complete)

{

action(r.Next);

r = enumerator2();

}

}

“This brings us back to”

Enumerable2.Range(0, 10).ForEach(Console.WriteLine);

“Just as an aside, I don’t know if you know this but types in C# don’t actually have to implement IEnumerable in order to be able to work with the foreach keyword, you simply need a method called GetEnumerator which returns something that has MoveNext and Current. Its an example of what is known as duck typing by the compiler. Given this we should be able to write an extension method on our functional definition called GetEnumerator right?? Unfortunately not. From the original source, the C# compiler takes several normalizing passes over that code to implement language features before, at last emitting IL. It seems like the pass which looks for GetEnumerator occurs BEFORE the pass that resolves Extension Methods. ”

“OK, before we move on build out a couple more enumerables”

public static Func<Func<Notification<char>>> Keys()

{

return (() =>

{

Console.WriteLine("Press esc to exit");

return () =>

{

var k = Console.ReadKey(true);

return k.Key == ConsoleKey.Escape ? Notification.Complete<char>() : Notification.Next(k.KeyChar);

};

});

}

public static Func<Func<Notification<T>>> Empty<T>()

{

return () => () => Notification.Complete<T>();

}

public static Func<Func<Notification<T>>> Return<T>(T value)

{

return () =>

{

var first = true;

return () =>

{

if (first)

{

first = false;

return Notification.Next(value);

}

return Notification.Complete<T>();

};

};

}

“And to enumerate it: “

Enumerable2.Keys().ForEach(Console.WriteLine);

“So we can do enumeration right now. Big deal, to be useful we probably want to implement Where and Select right?”

public static Func<Func<Notification<T>>> Where<T>(this Func<Func<Notification<T>>> source, Func<T, bool> predicate)

{

return () =>

{

var enumerator2 = source();

return () =>

{

while (true)

{

var r = enumerator2();

if (r.NotificationType == NotificationType.Complete)

return r;

if (predicate(r.Next))

return Notification.Next(r.Next);

}

};

};

}

public static Func<Func<Notification<TO>>> Select<TI, TO>(this Func<Func<Notification<TI>>> source, Func<TI, TO> projection)

{

return () =>

{

var enumerator2 = source();

return () =>

{

var r = enumerator2();

return r.NotificationType == NotificationType.Complete ? Notification.Complete<TO>() : Notification.Next(projection(r.Next));

};

};

}

“Now we can do”

Enumerable2.Range(0, 10).Where(i => i % 2 == 0).Select(i => string.Format("str:{0}", i)).ForEach(Console.WriteLine);

“Or”

var q2 = from i in Enumerable2.Range(0, 10)

where i%2 == 0

select string.Format("str:{0}", i);

q2.ForEach(Console.WriteLine);

“So this is Linq in action folks. I mentioned compiler passes before, well this is a feature called query comprehensions, essentially there is a compiler pass to transform the latter into the former. So lets do something cooler:”

var q = from i in Enumerable2.Range(0, 10)

from j in Enumerable2.Range(0, 2)

where i%2 == 0

select i + j;

q.ForEach(Console.WriteLine);

“This should print the numbers 1-> 10 by selecting all the even numbers along the way, then adding 0 and 1 to each of them. Theres clearly a problem though, its doesn’t compile. For me, they way I like to think about SelectMany is that it is shorthand for “for every one thing, give me many more things”, or “for every item in the outer collection, build me another collection then return each item in it and repeat with the every value in the outer collection”. I am not even going to attempt to type this one……”

public static Func<Func<Notification<TResult>>> SelectMany<TOuter, TInner, TResult>(this Func<Func<Notification<TOuter>>> source, Func<TOuter, Func<Func<Notification<TInner>>>> innerFactory, Func<TOuter, TInner, TResult> resultSelector)

{

return () =>

{

var enumerator2 = source();

var stepOuter = true;

Func<Notification<TInner>> inner = null;

Notification<TOuter> currentOuter = null;

return () =>

{

while (true)

{

if (stepOuter)

{

currentOuter = enumerator2();

if (currentOuter.NotificationType == NotificationType.Complete)

return Notification.Complete<TResult>();

inner = innerFactory(currentOuter.Next)();

stepOuter = false;

}

else

{

var r = inner();

if (r.NotificationType == NotificationType.Complete)

{

stepOuter = true;

continue;

}

return Notification.Next(resultSelector(currentOuter.Next, r.Next));

}

}

};

};

}

“And with this our linq expression compiles and prints the values we expect. Theres probably one or two language geeks in the audience that are thinking, there’s more I can do with the SelectMany, those would be the ones that recognise it as a version of monadic Bind. Bind is extremely powerful just as a quick example lets look again at Where, we can definitely rewrite it in terms of SalectMany”

public static Func<Func<Notification<T>>> Where<T>(this Func<Func<Notification<T>>> source, Func<T, bool> predicate)

{

return source.SelectMany(v => predicate(v) ? Return(v) : Empty<T>(), (outer, inner) => inner);

}

“Ok, that’s about it for part 1. I hope I have proved that we can pretty accurately reproduce IEnumerable functionally”

# Part 2. IObservable<T>…. Functionally

So we have seen IEnumerable functionally

/\*

\* IEnumerable = Func<Func<Notification<T>>>

\* IObservable = ????

\* \*/

“The IEnumerable definition says, GET me something from which I can GET a something or nothing”

”Whats the opposite of GET?”

“GIVE!”

“So we want GIVE something to which something or nothing can be GIVEN.”

“If Func defines a value returning function, or GETTER, what defines a SETTER in functional terms?”

“Action”

“So we can dumbly apply this inversion to produce”

/\*

\* IEnumerable = Func<Func<Notification<T>>>

\* IObservable = Action<Action<Notification<T>>>

\* \*/

“So lets define the old favorite, Range”

public static Action<Action<Notification<int>>> Range(int start, int count)

{

Action<Action<Notification<int>>> observable = observer =>

{

for (var i = 0; i < count; i++)

observer(Notification.Next(start + i));

observer(Notification.Complete<int>());

};

return observable;

}

“Does everyone follow? We don’t really need the intermediate variable but I think it makes it a little clearer. We can read it as, “when someone passes in an observer (which is an action over notification), use the SETTER to set the values from the loop followed by SETTING the completion.” It will help to see it used.”

Action<Notification<int>> observer = n =>

{

if(n.NotificationType == NotificationType.Next)

Console.WriteLine(n.Next);

else

Console.WriteLine("done");

};

Observable2.Range(0, 10)(observer);

“The variable observer is an Action which defines what to do when the notification comes in. Action<Notification>. Range returns an Action<Action<Notification>>, in other words something we can pass the Action<Notification > to.”

“Lets do Keys again”

public static Action<Action<Notification<char>>> Keys()

{

Action<Action<Notification<char>>> observable = observer =>

{

Console.WriteLine("Press esc to exit");

while (true)

{

var k = Console.ReadKey(true);

if (k.Key == ConsoleKey.Escape)

break;

observer(Notification.Next(k.KeyChar));

}

observer(Notification.Complete<char>());

};

return observable;

}

And on the call side:

Action<Notification<char>> observer2 = n =>

{

if (n.NotificationType == NotificationType.Next)

Console.WriteLine(n.Next);

else

Console.WriteLine("done");

};

Observable2.Keys()(observer2);

“We are looking a lot like before we defined ForEach on our function enumerable. So lets do that now.”

public static void ForEach<T>(this Action<Action<Notification<T>>> source, Action<T> onNext)

{

Action<Notification<T>> observer2 = n =>

{

if (n.NotificationType == NotificationType.Next)

onNext(n.Next);

else

Console.WriteLine("done");

};

source(observer2);

}

And on the call side:

Observable2.Keys().ForEach(Console.WriteLine);

“Now we have moved to callbacks we have something VERY different to IEnumerable. IEnumerable is a PULL based idiom; calls to MoveNext and Current are synchronous; they complete on the same thread that initiated the call. With IObservable and callbacks we have a PUSH based sequence. The callbacks may be called by any thread at pretty much ANY TIME.”

“Lets demonstrate that by building a time-based sequence; Interval”

public static Action<Action<Notification<int>>> Interval(TimeSpan timeout)

{

Action<Action<Notification<int>>> observable = observer =>

{

var i = 0;

var t = new Timer(\_ => observer(Notification.Next(i++)), null, (long)timeout.TotalMilliseconds, (long)timeout.TotalMilliseconds);

};

return observable;

}

Observable2.Interval(TimeSpan.FromSeconds(1)).ForEach(Console.WriteLine);

With that ticking away

“Can anyone think of shortcomings here?”

We want:

* Exceptions are no longer on the stack. SEH falls down, we need to encode them in some other way
* How do we know when we are done? Terminating vs non-terminating
* Unsubscribing?.... we will come back to that, lets deal with errors and completion first.

Add Error to NotificationType and fix up the Notification<T> and Notification types.

Add new operator Subscribe.

public static void Subscribe<T>(this Action<Action<Notification<T>>> source, Action<T> onNext, Action<Exception> onError, Action onComplete)

{

Action<Notification<T>> observer = n =>

{

switch (n.NotificationType)

{

case NotificationType.Next:

onNext(n.Next);

break;

case NotificationType.Complete:

onComplete();

break;

case NotificationType.Error:

onError(n.Exception);

break;

default:

throw new ArgumentOutOfRangeException();

}

};

source(observer);

}

“Lets add an Interval overload that shows this off a little”

public static Action<Action<Notification<int>>> Interval(TimeSpan timeout, int count)

{

Action<Action<Notification<int>>> observable = observer =>

{

var i = 0;

Timer t = null;

t = new Timer(\_ =>

{

observer(Notification.Next(i++));

if (i != count) return;

if (t != null) t.Dispose();

observer(Notification.Complete<int>());

}, null, (long)timeout.TotalMilliseconds, (long)timeout.TotalMilliseconds);

};

return observable;

}

And call it:

Observable2.Interval(TimeSpan.FromSeconds(1), 3)

.Subscribe(Console.WriteLine, e=>Console.WriteLine(e.Message), ()=>Console.WriteLine("done"));

“There is still another problem lurking here. I alluded to it earlier with the fact that IEnumerable<T> derives IDisposable…. This is for cancellation and resource deallocation. Just imagine that we have a UI subscribed to notifications of stock ticks and that these occur very fast and update a grid, if the user navigates away from that screen, you’d want to cancel the subscription to save system resources right? Well this is the role of the IDisposable which I have ignored upto now because it complicates the functional representations (IObservable<T> would have been Func<Action<Notification<T>>, IDisposable> and its not even possible to represent functionally for IEnumerable because of the way the enumerator derives it). In section 3 at going from the purely functional view back into the nice safe world of interfaces.”

# Part 3. IObservable<T>… with Interfaces

“Lets do the reverse of the very first thing we did, here we will go from our functional representation to an interface based one.“

Action<Action<Notification<T>>>

Starting with the outer Action this represents something we can pass a RECIEVER of notifications e.g it is the IObservable<T>. We have seen that it should probably be called Subscribe and take an IObserver<T>

public interface IObservable3<out T>

{

void Subscribe(IObserver3<T> observer);

}

“And the IObserver”

public interface IObserver3<in T>

{

void OnNext(T value);

void OnError(Exception e);

void OnComplete();

}

“So here we have broken out the notification types. Whilst we *could* have a single notification type, its probably more naturally C# to pull them out given that lack of support for discriminated unions.”

“Right, so this is more or less the translation of out functional representation, but what about that IDisposable. People who have not used Rx before… Where do you think it should be?”

Make Subscribe return IDisposable. Reiterate Func<Action<Notification<T>>, IDisposable>

“See how this is break from the mechanical inversion? It’s the calling side that always needs to IDisposable, with IEnumerable, this can be the IEnumerable itself (remember our comment/TODO?), with Observable, despite the fact that we are passing IN an observer, we require the Disposable to be returned. ”

“So lets get started with good old fashioned Range. As you can see we have a bit of a problem already! In C# we cant anonymously implement interfaces (wish we could), so lets create AnonObservable and accept Resharper’s help.”

public static IObservable3<int> Range (int start, int count)

{

return new AnonObservable2<int>();

}

public class AnonObservable<T> : IObservable3<int>

{

public IDisposable Subscribe(IObserver3<int> observer)

{

throw new NotImplementedException();

}

}

It’s clear we need something to represent what to do with a new Observer, furthermore that thing needs to return a disposable now to allow us to unsubscribe that guy. So we need a function that takes an Observer and returns an IDisposable, so that’s Func<IObserver<T>, IDisposable>, lets push it into the ctor.

public class AnonObservable<T> : IObservable3<T>

{

private readonly Func<IObserver3<T>, IDisposable> \_factory;

public AnonObservable(Func<IObserver3<T>, IDisposable> factory)

{

\_factory = factory;

}

public IDisposable Subscribe(IObserver3<T> observer)

{

return \_factory(observer);

}

}

“Lets build out Range again”. Build Range adding Disposable as you go

public static IObservable3<int> Range(int start, int count)

{

return new AnonObservable<int>(obs =>

{

var isDisposed = false;

for (var i = 0; i < count; i++)

{

if (!isDisposed)

obs.OnNext(i + start);

}

if (!isDisposed)

obs.OnComplete();

return Disposable.Create(() => isDisposed = true);

});

}

“Ok, just one more little refactor, lets create a generic factory called Create because in the full Rx library, Create is the primary way to Create Observable sequences and Operators, so it will pay to get accustomed to it“

public static IObservable3<T> Create<T>(Func<IObserver3<T>, IDisposable> factory)

{

return new AnonObservable<T>(factory);

}

“Create adds checks around OnComplete only being called one, no OnNext after and OnComplete and OnError. As a placeholder for that safety, we will just add the following.”

public static IObservable3<T> Create<T>(Func<IObserver3<T>, IDisposable> factory)

{

Func<IObserver3<T>, IDisposable> safe = o =>

{

//The real Create adds more safety

try

{

return factory(o);

}

catch (Exception e)

{

o.OnError(e);

}

return Disposable.Create(() => { });

};

return new AnonObservable<T>(safe);

}

“Now range can become”

public static IObservable3<int> Range(int start, int count)

{

return Create<int>(o =>

{

var isDisposed = false;

for (var i = 0; i < count; i++)

{

if (isDisposed) break;

o.OnNext(i + start);

}

if(!isDisposed)

o.OnComplete();

return Disposable.Create(() => isDisposed = true);

});

}

“See whats happening here? We are passed and observer and we can start pushing values to him, however if the Subscription is disposed we stop. This example is super trivial but imagine these values are coming back over a network, we’d really want to close down that connection as soon as our observer is not interested anymore.”

“OK, so lets Subscribe to it.”

Observable3.Range(0, 10).Subscribe();

“Intellisense is telling us we need to subscribe with an IObserver which is kind of sucky so we’ll do the Anon implementation trick again and start by building an extension method”

public static IDisposable Subscribe<T>(this IObservable3<T> source, Action<T> onNext, Action<Exception> onError, Action onComplete)

{

return source.Subscribe(new AnonObserver<T>(onNext, onError, onCOmplete));

}

Build out AnonObserver and complete the subscription code. Run it.

“Lets get back to time based sequences with an Interval implementation, and while we are at it we will add the Where and Select operators”

public static IObservable3<int> Interval(TimeSpan timeout)

{

return Create<int>(o =>

{

var i = 0;

var t = new Timer(\_ => o.OnNext(i++), null, (long)timeout.TotalMilliseconds, (long)timeout.TotalMilliseconds);

return Disposable.Create(t.Dispose);

});

}

public static IObservable3<T> Where<T>(this IObservable3<T> source, Func<T, bool> predicate)

{

return Create<T>(o => source.Subscribe(n =>

{

if (predicate(n))

o.OnNext(n);

}, o.OnError, o.OnComplete));

}

public static IObservable3<T2> Select<T, T2>(this IObservable3<T> source, Func<T, T2> projection)

{

return Create<T2>(o => source.Subscribe(n => o.OnNext(projection(n)), o.OnError, o.OnComplete));

}

“See how we cancel the timer when interval is unsubscribed? Also, see how simple Select and Where were. So now it starts to get interesting because we can do things like the following”

var q = from i in Observable3.Interval(TimeSpan.FromSeconds(1))

where i % 2 == 0

select "oooh: " + i;

var d = q.Subscribe(Console.WriteLine, e => Console.WriteLine(e.Message), () => Console.WriteLine("done"));

Console.ReadKey(true);

Console.WriteLine("disposing");

d.Dispose();

“Lets do a couple more Operators. Lets do Take”

public static IObservable3<T> Take<T>(this IObservable3<T> source, int count)

{

return Create<T>(o =>

{

var i = 0;

IDisposable t = null;

t = source.Subscribe(n =>

{

i++;

o.OnNext(n);

if (i == count)

{

t.Dispose();

o.OnComplete();

}

}, o.OnError, o.OnComplete);

return t;

});

}

“So now we can hook that up to the Interval we had before:”

q.Take(3).Subscribe(Console.WriteLine, e => Console.WriteLine(e.Message), () => Console.WriteLine("done"));

“OK, that’s what it is. But you are probably thinking you could do the same with a timer. The point is we now have a generic mechanism for defining and controlling things that happen over time. We’ll go deeper into how that is cool in the demo section. But there one piece that’s missing…. SelectMany”

public static IObservable3<TResult> SelectMany<TOuter, TInner, TResult>(this IObservable3<TOuter> source,

Func<TOuter, IObservable3<TInner>>

innerFactory,

Func<TOuter, TInner, TResult>

resultSelector)

{

return Create<TResult>(o =>

{

var innerSubscription = Disposable.Empty;

var sync = new object();

var outerSubscription = source.Subscribe(n =>

{

lock (sync)

{

innerSubscription.Dispose();

innerSubscription = innerFactory(n)

.Subscribe(v => o.OnNext(resultSelector(n, v)), o.OnError, () => { });

}

}, o.OnError, o.OnComplete);

return Disposable.Create(() =>

{

outerSubscription.Dispose();

lock (sync)

{

innerSubscription.Dispose();

}

});

});

}

“And something simple to use it:”

var q1 = from outer in Observable3.Interval(TimeSpan.FromSeconds(1))

from inner in Observable3.Interval(TimeSpan.FromMilliseconds(1)).Take(3)

select inner;

var d1 = q1.Subscribe(Console.WriteLine, e => Console.WriteLine(e.Message), () => Console.WriteLine("done"));

Console.ReadKey(true);

d1.Dispose();

Console.WriteLine("disposed");

Console.ReadKey(true);

“I should just mention that all the code samples I have given are very naïve. In the full Rx framework there are a lot of checks and balances to help you keep behaviours consistent. There are also design guidlelines to help you keep whatever you produce idiomatic”

# Demo

Walkthough

# Resources

<http://msdn.microsoft.com/en-gb/data/gg577609.aspx>

<http://www.introtorx.com/>

<http://weareadaptive.com/blog/>

Channel9!!