minRx
A minimal implementation of Reactive Extensions in C#

Final Report
Fall 2012

Steinar Gislason
BSc in Computer Science
\textbf{minRx} is a minimal implementation of Microsoft Reactive Extensions library using functional programming methods in C#. The base for this project is another project called minLINQ created by \textit{Reactive Extensions} developer Bart De Smet. minLINQ provided a way to use functional methods to query interactive lists (Enumerables). \textbf{minRx} extends these methods so that they can also be used with reactive lists (Observables), similar to that which the Reactive Extensions library offers. The reader should have some familiarity with C#, the \textit{Reactive Extensions} library, threading issues, functional programming and LINQ.
# Contents

1 Introduction  
2 minLINQ  
3 Monads  
   3.1 Return  
   3.2 Bind  
   3.3 Axioms  
   3.4 Using monads  
4 Interactive v.s. Reactive  
   4.1 Enumerable v.s. Observable  
   4.2 Reactive Extensions library  
5 Schedulers  
   5.1 Disposable  
   5.2 ThreadPool  
   5.3 TPL: Task Parallel Library  
   5.4 SynchronizationContext  
6 Coding in minRx (The user perspective)  
   6.1 Query  
   6.2 Observable creation  
   6.3 Event handler  
   6.4 Asynchronous methods  
   6.5 Schedulers in minRx  
7 Conclusion  
   7.1 The research  
   7.2 Final verdict  

Appendices

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>4</td>
</tr>
<tr>
<td>2 minLINQ</td>
<td>4</td>
</tr>
<tr>
<td>3 Monads</td>
<td>5</td>
</tr>
<tr>
<td>3.1 Return</td>
<td>5</td>
</tr>
<tr>
<td>3.2 Bind</td>
<td>6</td>
</tr>
<tr>
<td>3.3 Axioms</td>
<td>7</td>
</tr>
<tr>
<td>3.4 Using monads</td>
<td>8</td>
</tr>
<tr>
<td>4 Interactive v.s. Reactive</td>
<td>9</td>
</tr>
<tr>
<td>4.1 Enumerable v.s. Observable</td>
<td>10</td>
</tr>
<tr>
<td>4.2 Reactive Extensions library</td>
<td>10</td>
</tr>
<tr>
<td>5 Schedulers</td>
<td>10</td>
</tr>
<tr>
<td>5.1 Disposable</td>
<td>11</td>
</tr>
<tr>
<td>5.2 ThreadPool</td>
<td>12</td>
</tr>
<tr>
<td>5.3 TPL: Task Parallel Library</td>
<td>12</td>
</tr>
<tr>
<td>5.4 SynchronizationContext</td>
<td>13</td>
</tr>
<tr>
<td>6 Coding in minRx (The user perspective)</td>
<td>14</td>
</tr>
<tr>
<td>6.1 Query</td>
<td>15</td>
</tr>
<tr>
<td>6.2 Observable creation</td>
<td>15</td>
</tr>
<tr>
<td>6.3 Event handler</td>
<td>16</td>
</tr>
<tr>
<td>6.4 Asynchronous methods</td>
<td>16</td>
</tr>
<tr>
<td>6.5 Schedulers in minRx</td>
<td>16</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>17</td>
</tr>
<tr>
<td>7.1 The research</td>
<td>17</td>
</tr>
<tr>
<td>7.2 Final verdict</td>
<td>18</td>
</tr>
<tr>
<td>Appendices</td>
<td>19</td>
</tr>
</tbody>
</table>
1 Introduction

What the f.. ? Is it dead? Hello? Wake up? Arrgg! These are common thoughts that many users have had the pleasure of having bursting into their heads and sometimes even out of their mouths.

While the program just unresponsively sits there one is desperately clicking on it trying to get any feedback. You are left in the dark ... waiting and waiting ... seems sometimes that you’ve been waiting a decade ... but when impatience gets the better of you, you just kill the program. This happens all the time.

Although there are many unavoidable reasons for this e.g. network latency, lack of memory, disk paging or the CPU is simply busy with something else, it is never a good experience when the program hangs. But to solve this, we the programmers must change the way we program and utilize the available multi-threaded environment we have at our disposal.

My operation system teacher said to our class: "Do not use multiple threads in your programs. Why? Because you will make mistakes." He is a very good teacher but obviously, in my opinion, completely wrong. His audience consisted mostly of novice programmers so the statement was somewhat understandable. But what was missing from that statement was maybe "... until you know what you are doing".

Programming in a multi-threaded environment is challenging. That is why the Reactive Extensions [Tea] library form Microsoft is very interesting. The Reactive Extensions aims to simplify the work needed in C# to implement concurrency and thus help us to avoid the hanging experience. Although they don’t solve every multi-threaded problem, it looks like a better way to approach it.

This report and minRx is aimed for the C# developer who wants to understand the Reactive Extensions environment and dig a little deeper into the functional programming potentials of C#. The report will also explain how the dreaded monads[Dye08] can be created in C#. C# is not a functional programming language like F#, but one can utilize high order functions (send functions as parameters) and other interesting stuff like currying[Pop12]. Learning these concepts may help you create more loosely coupled code and find new ways of solving hard to fix problems in your C# code.

2 minLINQ

Our first stop must be minLINQ because it is the base project for the minRx library. Bart De Smet, the author of minLINQ and the current developer of Reactive Extensions wanted to demonstrate how LINQ can be built using only three powerful operators that he used to manipulate a monad construct. He called these operators Ana, Bind and Cata which he got from Erik Meijers[wika] thesis Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire[MFP91].

Ana, sometimes also called return or unit, is used as the operator to get into the monad. Cata is a supplementary operator not really needed to support the monad, but helps with many aggregate functions. Bind is the most powerful operator in the monad because it can make your monad function compositions look like they are on steroids.
Bind never leaves the monad but one can make any functional change to the value as long as the function lets you stay in the monad (explained later).

Bart stated that from these operators it is possible to create about 80-90% of all LINQ methods that are available today. He built many of LINQs methods using this technique in his minLINQ project[1]. However minLINQ only uses these monad operators to iterate over interactive lists, known as an enumerable collection. But in his blog[2] Bart showed that it could also be used to iterate over reactive lists, known as an observable collection. This iteration over observables is what minRx is built upon.

3 Monads

Then what is a monad that minLINQ so powerfully utilizes? Well, monads come from category theory and were introduced to computer science to aid the semantics of computation by Eugenio Moggi[3]. Monads are often thought of as only useful in pure lazy functional languages like Haskell, but their application can obviously be extended to other languages as well (Scala, Ocaml, F# etc.). One thing they can do really well is to control complexity of functional composition.

How then does a monad help with complexity? Especially as they are hard to understand so one would think that they add to the complexity rather than reducing it. Also, learning about monads requires a paradigm shift in problem solving which is not always easy to grasp. However, one successful implementation of a monad in C# is LINQ. LINQ is an extremely powerful, compositional way to handle and query any type of lists. LINQ is widely used in the C# community and has become an essential construct to many C# developers. Another type of monad is a maybe monad that could be used to implement nullables to any type without explicit language support. Many other monad concepts are available[4, 5].

To be a monad the monad constructs must have a particular type, a function to get into the monad and a bind function so it can compose functions of that type together. Also the monad must obey three axioms, left identity, right identity and must be associative.

3.1 Return

The Return function is a simple gateway function that lets us enter the world of the monad. It wraps any value as a monadic type. Wadler in his report ”Monads for functional programming”[6] named this function unit that had this type structure.
In C# the function (unit function has been renamed as Return, to match this picture of the monad in figure 1) can be defined as:

```csharp
public static M<T> Return<T>(T value)
{
    return new M<T>(value);
}
```

Where M is any monad class that should have at least this supplementary class structure:

```csharp
class M<T>
{
    private T _value;
    public M(T value)
    {
        _value = value;
    }
}
```

By calling Return with any value we have wrapped that value inside a monad class.

### 3.2 Bind

The most confusing aspect of the monad is probably the Bind function. The purpose of the Bind function is to bind the result of a computation to a variable which in turn can be used in the succeeding computation.

Walder uses the \(\star\) to represent the Bind as an infix operator and explains the bind operator as having the following type signature and implementation:

\[
(\star) :: M_a \rightarrow (a \rightarrow M_b) \rightarrow M_b
\]

\[m \star \lambda a.n\]

He gives this explanation for the function:

The above can be read as follows: perform computation \(m\), bind \(a\) to the resulting value and then perform computation \(n\). Types provide a useful guide. From the type of \(\star\) we can see that expression \(m\) has type \(M_a\), variable \(a\) has type \(a\), expression \(n\) has type \(M_b\), lambda expression \(\lambda a.n\) has type \(a \rightarrow M_b\) and the whole expression has type \(M_b\). [Wad92]

Although this is a good explanation it is a little bit vague, atleast from a C# perspective. Converting this operator to a C# function one can start by making the \(\star\) a prefix function instead of infix. This will result in

\[\star(m, \lambda a.n)\]

where the computation \(m\) is the first parameter to the function and the \(\lambda a.n\) is the second. We will then exchange the \(\star\) operator with the function name \(Bind\).

\[Bind(m, \lambda a.n)\]
\[ \text{Bind}(M<A> m, \lambda a. n) \]

\[ \lambda a. n \] is converted to a function that takes the variable \( a \) (which confusingly is of type \( A \)) and will return the monad \( M<B> \). This means that the \text{Bind} function can return a monad containing a different type than the type that was in the first parameter, so the function \( \lambda a. n \) can do a type conversion.

\[ \text{Bind}(M<A> m, \text{Func}<A, M<B>> n) \]

The whole expression should then return the type of \( M<B> \). So the complete \( \ast \) function would look something like this in C#:

\[ M<B> \text{Bind}(M<A> m, \text{Func}<A, M<B>> n) \]

where \( M \) can be any monad class.

What is not clear only by looking at this prototype is how the result of the computation \( m \) gets bound as the variable to function \( n \). This is because it depends on the monad and so implementations can differ, but all implementations must adhere to the prototype. One easy implementation could be the implementation of the identity monad which could look something like this:

```csharp
public M<B> Bind<A,B>(M<A> a, Func<A, M<B>> k)
{
    return k(unwrap(a));
}
```

where \( \text{unwrap} \) is a function that retrieves the value that is contained in the monad. A more C# like way would be to implement a property for the monad and then use a getter method to retrieve the value. That is left as an exercise for the reader.

### 3.3 Axioms

As noted before, the monad must obey three axioms: left identity, right identity and should be associative. Just as in functional composition, this is necessary so the monad composition is well-behaved.

#### Left identity

\[ \text{Bind}(\text{Return}(a), a => k(a)) \equiv k(a) \]

**Example:** Given that the variable \( a \) has the value of 1. \( \text{Return}(a) \) should give us the monadic value \( M<int>(1) \). This means that \( M \) is a monad that contains the integer 1. The \( \text{Return} \) function does not do any computation so the return value from \( \text{Return} \) is bound to the variable sent to \( k \), is the same value given to \( \text{Return} \). Which is the same as sending the value \( a \) directly to \( k \).

#### Right identity

\[ \text{Bind}(m, a => \text{Return}(a)) \equiv m \]
**Example:** Let the variable $m$ be any computation in the monad. Say that the computation of $m$ returns the value $M<int>(42)$. The value 42 is then bound to the variable $a$. Return does not do any computation. It will only return the same value it received but wrapped in a monad, namely $M<int>42$ which is the same as the computation $m$ so the axiom holds.

**Associative**

$$\text{Bind}(m, x => \text{Bind}(k(x), y => h(y))) \equiv \text{Bind}(\text{Bind}(m, x => k(x)), y => h(y))$$

**Example:** To help us examine this we need to define the functions $k$ and $h$.

```csharp
Func<int, M<A>> k = x => new M<A>(x + 100);
Func<int, M<A>> h = x => new M<A>(x / 2);
```

The function $k$ takes in an integer value and adds the value 100 to it. The function $h$ takes in an integer value and divides it in half. Both functions return a new monad. Also let the computation $m$ return the monad $M<int>(18)$.

Let's first evaluate the left expression:

\[
\text{Bind}(m, x => \text{Bind}(k(x), y => h(y))) = \\
\text{Bind}(M<int>(18), 18 => \text{Bind}(k(18), y => h(y))) = (m \text{ evaluated}) \\
\text{Bind}(M<int>(18), 18 => \text{Bind}(M<int>(118), 118 => h(118))) = (k \text{ applied}) \\
\text{Bind}(M<int>(18), 18 => \text{Bind}(M<int>(118), 118 => M<int>(59))) = (h \text{ applied}) \\
\text{Bind}(M<int>(18), 18 => M<int>(59)) = (\text{reduce}) \\
= M<int>(59)
\]

Let's now evaluate the right expression

\[
\text{Bind}(\text{Bind}(m, x => k(x)), y => h(y)) = \\
\text{Bind}(\text{Bind}(M<int>(18), 18 => k(18)), y => h(y)) = (m \text{ evaluated}) \\
\text{Bind}(\text{Bind}(M<int>(18), 18 => M<int>(118)), y => h(y)) = (k \text{ applied}) \\
\text{Bind}(M<int>(118), 118 => h(118)) = (\text{reduce}) \\
\text{Bind}(M<int>(118), 118 => M<int>(59)) = (h \text{ applied}) \\
= M<int>(59)
\]

As seen by this example both equations return the same value so the axiom holds.

### 3.4 Using monads

Hopefully one can now understand how monads are created, but their usefulness is probably still a bit obscure. I would say that their strongest feature is their ability to be compositional. The simplest of all monad types is a type called the *Identity* monad. It would be possible in C# to create something like this using the identity monad:
1. ToIdentity().Add(5).Minus(3).Print(WriteLine).Mul(6)
   .Wait(TimeSpan.FromSeconds(2)).Print(WriteLine);

Many will see the similarity between this code and LINQ. The function ToIdentity() is of course the return function that encapsulates the integer 1 inside the monad. All other methods use the Bind function in some way behind the scenes. Although this could be implemented, perhaps more simply, using standard procedural or object oriented structure; clearly the compositional nature of the code makes it very appealing. If one looks at monads like this, I think one begins to grasp their powerful nature. Both minLINQ and minRx use monads like this to be able to compose their functions together. Take a look at appendix A to see how this identity monad could be created.

4 Interactive v.s. Reactive

Now that we have some understanding of monads and their role, it is time to look at the difference between working with interactive lists, as done in minLINQ and reactive lists, as done in minRx and Rx. To understand this one must understand the difference between interactive and reactive programming.

The difference is quite distinct when one looks at how it communicates with its environment. When an interactive call is made to its environment, e.g. waiting for user input, the program blocks all further work and waits for the environment to signal that its work is done and then the program will continue executing.

Heavy use of interactive programming leads to unresponsive programs and bad user experience as was explained in the introduction. The benefit of using interactive programming is that it is simple and the program knows its state at all time. No need for threads so the program avoids the risk of containing thread specific bugs.

The reactive programming paradigm however does not wait for the environment. It only signals to the environment that it may send its information when it is ready to do so. So the environment can then push new values at the program at any time. This usually leads to a better user experience, when it is done correctly, but managing the
state of the program can become very problematic. First of all the program cannot assert what state it will be in when it receives the values from the environment. Also the values can affect and change the state, sometimes even in a non-sequential order. Also to support this the programmer has to introduce threads to the program which can lead to thread specific bugs.

Programmers often shy away from this reactive programming style because of its complexity and are often discouraged to write multi-threaded applications. However this is getting harder to avoid because of the demand for a better user experience and improved use of the increasing number of CPUs in our computers.

4.1 Enumerable v.s. Observable

The way to represent an interactive list in C# has been around since the first version of C# and is known as an enumerable list. Whenever you want to receive a blocking list, e.g. query a database, that list implements the IEnumerable interface. This is an interactive call where the call waits for the database to give us the next value before continuing. It can be thought of as a pull based method, where you are pulling the data out of the database.

Erik Meijer and Wes Dyer released in July 2009 the first version of the Reactive Extensions library. They were of the view that it would be possible to create a dualistic form of the interactive list to represent a reactive list. By studying the interface of the enumerable and doing some mathematical observation they succeeded in creating a new interface for these reactive lists which they named IObservable (see figure 3).

These reactive lists can represent all kinds of reactive data such as events, asynchronous function calls or any stream of data that can be tapped into without blocking the programs execution. This can be thought of as a push based method, where the data gets pushed at the program at any time. This in itself was a great discovery and as Erik Meijer himself said:

"I can never in my life time invent anything this beautiful any more". [MD09]

4.2 Reactive Extensions library

So then, what is this Reactive Extensions library? Reactive Extensions is a library that tries to simplify this reactive style of programming. It contains the interface for reactive lists (IObservable), and it supports querying these list using LINQ (this is why we introduced monads) and schedulers which we will discuss next.

5 Schedulers

The scheduler in the Reactive Extensions library has three main concepts. It controls where the work item will be executed, it controls the priority of the items being executed and when it should be executed. Although the Observable interface was designed to work as a reactive list running on a different thread, the concept of it doing so was abstracted away into this scheduler structure meaning you can use Observable without ever using any concurrency in your program. Also Reactive Extensions was designed
to be free-threaded. This means that it could run on any thread and it should not to
be confused with multi-threaded constructs, which are supposed to be run on multiple
parallel threads.

Although a simple scheduler is in itself not too hard to implement, I found that it
had the most problematic areas for the minRx implementation. This is mainly due to
non-standard design patterns for threads and strange quirks in the C# API which I will
touch upon in the next sections.

The Reactive Extensions library supports an immediate scheduler, newthread sched-
ulator and a threadpool scheduler. The immediate scheduler does the work on the current
thread and is a blocking scheduler. The new thread scheduler does the work on a new
thread and the threadpool scheduler lets the threadpool handle the work. The immediate
and new thread scheduler implementation was manageable but the thread pool imple-
mentation was a bit risky as it could not be disposed of without the use of a cancellation
token. This token would have to be parameterized to all calls which would break the
current simple API of minRx.

Besides the scheduler, the Reactive Extensions library also supports the Syncroniza-
tionContext class. I found out that this support could also lead to strange bugs. Next
I will describe some findings that I gathered when I was trying to implement these
schedulers.

5.1 Disposable
To show more clearly the duality of Enumerable and Observables in minLINQ, Bart
De Smet, the author of minLINQ, purposefully ignored the disposable part of these
expressions. To implement the disposable for the Observables a change in the sequence signature had to be made. The signature was:

\[
(() \rightarrow \text{Maybe}<T>) \rightarrow ()
\]

\[
\text{Action}<\text{Action}<\text{Maybe}<T>>>
\]

But now we want the expression to return a disposable method. The disposable method is a function that takes in nothing and returns noting, which we can define in C# with the `Action` declaration. So the signature for the sequence becomes:

\[
() \rightarrow ((() \rightarrow \text{Maybe}<T>) \rightarrow ()
\]

\[
\text{Func}<\text{Action}<\text{Maybe}<T>>, \text{Action}>
\]

This allows us keep a handle on the sequence so it is for example possible to stop an infinite running sequence that is running in a new thread by calling dispose. To support this disposable option in `minRx` the signature of all functions was changed to this disposable signature.

### 5.2 ThreadPool

Note however that not all schedulers can return a disposable method[Mic]. For example the thread pool scheduler is a queue based scheduler, so whenever a scheduled work has been queued there is no way of disposing of that work.

Microsoft’s recommendation for the thread pool is that it should only be used for short tasks, but one can easily put an infinite loop on the thread pool and that thread would run until the process of the program terminates. Jon Skeet explains it like this:

"By default, the thread pool has 25 threads per processor. Note that the thread pool isn’t just used for whatever asynchronous calls you make - the .NET framework libraries use it as well, and things can go badly wrong (usually resulting in a deadlock) if all the threads are used and some of them depend on other tasks which are scheduled. (For instance, if one thread is waiting for the results of another work item, but that work item is never run because there are no free threads.) This is a good reason to avoid using the thread pool for particularly long-running tasks.”[Ske]

### 5.3 TPL: Task Parallel Library

The TPL default behavior is to use the thread pool when it schedules work. One should then use the same design principles as when one is working with thread pools. One other thing to bear in mind when using TPL is that the work done after the `async` keyword is by default pushed to a thread pool which then communicates with the calling thread using the `SynchronizationContext` class. This means that TPL only works with UI threads out of the box, because only WinForms, WPF and Silverlight have a working implementation of the `SynchronizationContext` class.
5.4 SynchronizationContext

The purpose the `SynchronizationContext` class in C# is to allow threads to communicate. If you give another thread your threads current `SynchronizationContext`, then the other thread is supposed to be able to post back messages onto your thread through the `SynchronizationContext`. While this is true in some cases, it is only true in those scenarios where `SynchronizationContext` has been inherited and fully implemented, for example the `WindowsFormsSynchronizationContext`.

The `WindowsFormsSynchronizationContext` is used to synchronize with the windows form UI thread but what is strange is that the context is secretly created for you whenever you create windows control in your program [Per09]. It is kind of awkward that it magically and unknowingly appears as a side-effect of doing something that is somewhat related.

Running the code below in a console application (see code for the `SynchronizationContextTester` in Appendix B) would give us an error message saying that no `SynchronizationContext` is available.

```csharp
new SynchronizationContextTester(Console.WriteLine).RunTest();
```

However, if we were to run a similar code in a windows application where a form has been created then we would not get this error e.g.:

```csharp
new SynchronizationContextTester(x => MessageBox.Show(x)).RunTest();
```

What is also rather annoying is that the `SynchronizationContext` does not work out of the box. This is because the main methods that should be used to communicate between threads, the `Send` and `Post` methods, have a misleading implementation which do not allow for cross-thread communication. These methods have to be overridden in an inherited class to support this and this can complicate the code [Per08a].

Not all `SynchronizationContext` implementations guarantee the order of delegate execution or synchronization of delegates. The UI-based `SynchronizationContext` implementations do satisfy these conditions, but the ASP.NET `SynchronizationContext` only provides synchronization. The default `SynchronizationContext` does not guarantee either order of execution or synchronization. [Cle11]

Therefore in a console application we have to create a new `SynchronizationContext` because none is created for us.

```csharp
var ctx = new SynchronizationContext();
SynchronizationContext.SetSynchronizationContext(ctx);
```

And then we call:

```csharp
new SynchronizationContextTester(Console.WriteLine).RunTest();
```

But now the code will not behave as one expects as the output will be:

```
UI thread: 1
Run thread: 3
Send thread id: 3
Post thread id: 4
```
One expects that the methods Send and Post will do their work on the same thread as the UI thread, but they do not. That is because the default implementation of these methods does not marshal code between threads, as the purpose of this class implies.

Below are the default implementations of these methods taken from the mono class library [MCL05] and as one can see, no specific thread communication code is present. The Send method is just blocking on the same thread that is calling and the Post method sends the delegate to the thread pool as demonstrated above with the SynchronizationContextTester.

```csharp
public virtual void Send(SendOrPostCallback d, Object state)
{
    d(state);
}

public virtual void Post(SendOrPostCallback d, Object state)
{
    ThreadPool.QueueUserWorkItem(new WaitCallback(d), state);
}
```

However, the most compelling reason to create your own SynchronizationContext class would be if you wanted thread communication with WCF (Windows Communication Foundation), because their interface supports the SynchronizationContext [Per08b].

6 Coding in minRx (The user perspective)

How does one then code in minRx. Well, on the user side the code is very similar to Rx. All basic structures have been prefixed with an F and there are almost no properties but only functions. Let’s give a little example. Here is a piece of code in minRx that will iterate over the range 0..99 in a new thread:

```csharp
var xs = FObservable.Range(0, 100, FNewThreadScheduler.Default());
```

And here is the same code in Rx:

```csharp
var xs = Observable.Range(0, 100, NewThreadScheduler.Default);
```

As you can see the difference is subtle. Notice the prefix, but also that in Rx the NewThreadScheduler returns a property but minRx returns a function. However a big difference is hidden here by the var keyword. var is not a dynamic assignment but means that the type is determined at compile time. If we explicitly specify the type we will see the real difference.

```csharp
Func<Action<Maybe<T>>, Action> xs = FObservable.Range(...);

IObservable<int> xs = Observable.Range(...);
```

Well, this is one of the beauties of var. Sometimes it is more aesthetic to let the compiler infer the type for the programmer.
6.1 Query

Querying the list is basically the same, but minRx does not currently support as many query operators as Rx. But that is just a matter of time and implementation.

```csharp
var xs = FObservable.Range(0, 3).Where(x => x % 2 == 0).Select(x => x + 10);
var xs = Observable.Range(0, 3).Where(x => x%2 == 0).Select(x => x + 10);
```

6.2 Observable creation

One can also create an Observable in minRx.

```csharp
var xs = FObservable.Create<int>(obs => FImmediateScheduler.Schedule(() =>
{
    obs.OnNext(42);
    obs.OnNext(43);
    obs.OnCompleted();
}));
var xs = Observable.Create<int>(obs => Scheduler.Immediate.Schedule(() =>
{
    obs.OnNext(42);
    obs.OnNext(43);
    obs.OnCompleted();
}));
```

The difference between the schedulers is just syntactic sugar. However what is not obvious here is that the FObservable.Create method in minRx is also syntactic sugar. It was created just so minRx would adhere to the Rx API. The method in minRx looks like this:

```csharp
static Func<Action<Maybe<T>>, Action> Create<T>(
    Func<Action<Maybe<T>>, Action> observable)
{
    return observable;
}
```

It does nothing but return the parameter that was given to it. So the above Observer could have been written like this instead:

```csharp
Func<Action<Maybe<T>>, Action> xs = obs => FImmediateScheduler.Schedule(() =>
{
    obs.onNext(42);
    obs.onNext(43);
    obs.onCompleted();
});
```

However, because of the object oriented nature of Rx the same conversion is not possible. This shows that minRx functional implementation is closer to the root behavior of the inner workings of the Observable structure.
6.3 Event handler

\texttt{minRx} can also convert basic events as Observable lists.

\begin{verbatim}
var e = FObservable.FromEventPattern<MouseEventArgs>(sender, "MouseMove");

Here is the similar \textit{Reactive Extensions} code.

var e = Observable.FromEventPattern<MouseEventArgs>(sender, "MouseMove");

Both also subscribe the same way. They return an \textit{EventPattern} class that contains
the event arguments and the sender.

\begin{verbatim}
e.Subscribe(x => System.Diagnostics.Debug.WriteLine(x.EventArgs.Location));
e.Subscribe(x => System.Diagnostics.Debug.WriteLine(x.Sender));
\end{verbatim}
\end{verbatim}

6.4 Asynchronous methods

Every method in C\# can be called asynchronously and by using the \textit{FromAsyncPattern}
available in both \texttt{minRx} and \texttt{Rx} one can convert these methods to \textit{Observables}. Suppose
one creates a function like this:

\begin{verbatim}
Func<int, int> f = x => x + 1;
\end{verbatim}

This is a function called \textit{f} that takes in one integer argument and adds one to it. Converting this
function to an \textit{Observable} would look like this:

\begin{verbatim}
var fs = FObservable.FromAsyncPattern<int, int>(f.BeginInvoke, f.EndInvoke);
\end{verbatim}

or

\begin{verbatim}
var fs = Observable.FromAsyncPattern<int, int>(f.BeginInvoke, f.EndInvoke);
\end{verbatim}

depending on whether one uses \texttt{minRx} or \texttt{Rx}. However, \textit{Observables} were created
to work with lists, but functions like this are called \textit{one shot observables} as they contain only
one item. In order to call a function like this repeatedly, another function was created
to wrap this function call in a loop. This can be achieved by the \textit{While} function. That
takes in a predicate and \textit{Observable} and calls the \textit{Observable} while the predicate returns
true:

\begin{verbatim}
FObservable.While(() => true, fs(10)).Subscribe(x => WriteLine(x));
\end{verbatim}

This little code will infinitely call the function \textit{f} and writes the result.

6.5 Schedulers in \texttt{minRx}

A note about the schedulers is important. One reason is the challenge that Bart, author
of \texttt{minLINQ}, gave in his video when he asked how would one implement a scheduler
on top of \texttt{minLINQ}. In \texttt{Rx} all schedulers return a disposable, but in \texttt{minLINQ}
the disposable part is ignored (see Disposable section in the report) but \texttt{minRx} has
support for it. So to implement a scheduler in \texttt{minRx} that would schedule work on
another thread (ignoring time and priority) would be as easy as doing:
public static class FNewThreadScheduler
{
    public static Action Schedule(Action action)
    {
        var t = new Thread(() => action());
        t.Start();
        return () => { // The disposable action
            t.Abort();
        };
    }
    public static Func<Action, Action> Default()
    {
        return Schedule;
    }
}

Look how the Default function just returns the Schedule function. In the Reactive Extensions library one calls the Default method to obtain the common scheduler. Also we can see here that the prototype for the scheduler is a function that takes in a function and returns a handle to the disposable method, in order to give us the opportunity to dispose of the thread before it terminates. We are using closures here on the variable t so even though t is a local variable it will not be disposed of after the function ends.

7 Conclusion

When I decided to work on this project I didn’t have much knowledge of what Reactive Extensions were all about, nor had I done much functional work in C#. The whole subject was a big black box to me.

I had watched the minLINQ video[Sme11] and was very intrigued how Bart, the author, had implemented LINQ and there was a dare from him to implement a scheduler on top of his minLINQ code. But I wanted to do more! I wanted to make a minimal implementation of the Reactive Extensions library in this functional dimension that he had shown. I wanted to learn more deeply about the Reactive Extensions library, but also to learn more about functional programming methods in C#, the programming language I thought I knew so well. As a bonus I got to learn about monads.

7.1 The research

One of the questions that one should try to answer after doing this kind of research is, what approach is best suited for a project like this? What structure and what software methodology? How do you organize something when you have no idea how big it is or what you will be doing?

I tried to use small (one week) Spiral iterations. While it gave a little bit of discipline, it mostly got in the way. Researching the best approach to do research assignments could be an interesting research assignment in itself. However I can only give my own assessment of what I found was the best approach for me.

The best way for me was to be extremely adaptable and always work in the area where there was the highest risk of project failure. Every week a new focus area and steps were
decided by evaluating the current highest risk areas. The evaluation did not need to be formal, like in the Spiral methodology, as I used a conversational style assessment with my instructor. Interestingly we never disagreed on what the next logical step would be and there was always the feeling that we were on the right path.

The basic strategy was to implement a project that one could demonstrate. When that was possible, the next step then was to slowly build on top of that. Another big decision was to try to have an implementation similar to the Reactive Extensions library and lastly that the code was simple and well written. If something felt overly complicated, cumbersome or not understandable, then it was re-studied and re-written until the code fitted and had that nice flow. This made the process a bit slower, but will make the code easier to build upon. Like art it is hard to judge whether a piece of code is good, although coding standards give guidelines. My guideline was that the code had to be short, have a specific role but most of all that it improved the current API and played well with other functions. I feel that I accomplished that goal.

This was a good project to be a time-framed research project because it scales well. The Reactive Extensions library has a huge API that could be implemented in the minRx library. Accordingly when the crucial methods had been implemented, it was just a matter of how much time was left to implement other interesting features. I would say in about week 4 most of all the crucial methods had been implemented. Thankfully, this gave time to create a proof of concept application to demonstrate the minRx library.

7.2 Final verdict

I think that the minRx library, although not complete, can be a great starting point for other C# programmers to study the inner workings of Reactive Extensions and functional programming. It can be built upon to support more of the features of Reactive Extensions library and maybe even find better solutions than are currently available in Rx. It could also be ported to other languages e.g. Java and Python.

For me this was a great experience. Not only did I get an in-depth vision of how Reactive Extensions works but also strengthened my functional programming skills. I also got to experience the quirks and pitfalls of threading in C# for example the misconception of SynchronizationContext, the misuse of thread pools and a better understanding of the TPL. My knowledge of writing multi-threaded applications has increased and has become more comfortable in my mind. I am very pleased to have been given the chance to create this project and highly recommend this approach to other students.

Acknowledgements

I want to thank many people for advice, suggestions and criticisms, but in particular: Birgir Kaldal Kristjansson for all his tremendous help and support through this project, Arni Hermann Reynisson for his input and his enthusiastic interest in this subject, Hallgrimur Arnalds for supporting me, Bjorn Thor Jonsson for allowing me to proceed with this idea, Willard Thor Rafnsson for his crazy functional programming teaching, Ulfar Erlingsson for his deep knowledge of OS and threading issues, Valdimar Kristjansson for suggesting Reactive Extensions and Katherine Nichols for her help in this report.
Appendices

Appendix A

Identity monad example: Here is an example code on how an identity monad could be implemented in C#.

```csharp
using System;
using System.Threading;

namespace IdentityMonad
{
    public class Identity<T>
    {
        public T Value { get; set; }
        public Identity(T value) { Value = value; }
    }

    public static class IdentityExtensions
    {
        public static Identity<T> ToIdentity<T>(this T value)
        {
            return new Identity<T>(value);
        }

        public static Identity<int> Add(this Identity<int> identity, int value)
        {
            return Bind(identity, x => new Identity<int>(x + value));
        }

        public static Identity<int> Minus(this Identity<int> identity, int value)
        {
            return Bind(identity, x => new Identity<int>(x - value));
        }

        public static Identity<int> Mul(this Identity<int> identity, int value)
        {
            return Bind(identity, x => new Identity<int>(x * value));
        }

        public static Identity<T> Print<T>(this Identity<T> identity, Action<T> action)
        {
            action(identity.Value); //not pure functional code
            return identity;
        }

        public static Identity<T> Wait<T>(this Identity<T> identity, TimeSpan timeSpan)
        {
            Thread.Sleep(timeSpan); //not pure functional code
            return identity;
        }

        private static Identity<U> Bind<T, U>(Identity<T> a, Func<T, Identity<U>> k)
        {
            // Implementation
        }
    }
}
```
class Program
{
    static void Main()
    {
        2.ToIdentity().Add(5).Minus(3).Print(Console.WriteLine).
        Mul(6).Wait(TimeSpan.FromSeconds(2)).Print(Console.WriteLine);
        Console.WriteLine((2 + 5 - 3)*6);
    }
}
Appendix B

The below code is a helper class to test the behavior of the \textit{SynchronizationContext} class in .Net. This class was referenced in section 5.4 to demonstrate how \textit{SynchronizationContext} behaves differently in different contexts. This code was built using Visual Studio 2010 and .Net framework 4.0.

```csharp
public class SynchronizationContextTester
{
    private readonly Action<string> _writeLine;
    public SynchronizationContextTester(Action<string> writeLine)
    {
        _writeLine = writeLine;
    }

    public void RunTest()
    {
        _writeLine("UI thread: " + Thread.CurrentThread.ManagedThreadId);

        SynchronizationContext uiContext = SynchronizationContext.Current;
        if (uiContext == null)
        {
            _writeLine("No SynchronizationContext");
            return;
        }

        // create a thread and associate it to the run method
        var thread = new Thread(Run);
        thread.Start(uiContext);
    }

    private void Run(object state)
    {
        // lets see the thread id
        _writeLine("Run thread: " + Thread.CurrentThread.ManagedThreadId);

        // grab the context from the state
        var uiContext = state as SynchronizationContext;
        uiContext.Send(UpdateUI, "Send");
        uiContext.Post(UpdateUI, "Post");
    }

    private void UpdateUI(object state)
    {
        // what thread are we on?
        _writeLine(state + " thread id: " + Thread.CurrentThread.ManagedThreadId);
    }
}
```
Appendix C

Let’s look at the time spent doing this project. The total time was about 355 hours. I am going to unscientifically break down the project into four main categories: School, Research, Coding, and Report.

The School category which I spent about 61 hours and 30 minutes (this is not hard core science but an approximation) I classified all the work I had to do that I would not normally spend time doing if this project was not a school project. E.g. time spent preparing demonstrations, working on special school requirements, meetings etc. This time took about 17% of my 355 hours spent on this project.

The next three categories are amazingly even, but the next category is the Research category. About 28% or 99 hours went into researching this Reactive Extensions subject. Some of my research was interesting but did not fit into this project. While other research material fitted perfectly. However, all the time I spent researching I, personally, will never regret. They often shed a new light on unknown or known concept which in turn expanded my overall knowledge.

Coding, my favorite category is next in line. I would have spent all of my 355 hours coding if I were allowed too. Coding took about 97 and 30 minutes or 27% of the total time, which is less than one third. That is also less than two and a half weeks work, if you are working 40 hours per week. But here is where the real progress was made.

The last category is the time spent directly or indirectly working on this Report. I must admit that reports are not my forte; I like coding. Creating a report is more mentally exhausting to me then writing code and for that reason the process was much slower. Fortunately I started by spending a lot of time coding, so I could take it a bit slower when I was writing this report. When I examined the data I saw that I nearly spent the same time coding as in writing this report, or about 97 hours (27%), which probably just means that I currently explain things slower then I code. Unless you think this report is more interesting than the minRx code, then it’s the other way around.

I have shown no data to back up these findings in this report, although the data is available, it is not highly scientific. The reason is that if I have known I was going to categories my time before I started this project I would have been more thorough, but then the minRx project could have suffered. But I hope this could at least be a starting point for someone else who is interested in examining this area.
References


