David Barker

Functional Reactive Programming for Data Binding in C♯

Computer Science Tripos Part II

Jesus College

March 14, 2013

Proforma

Name:	**David Barker**
College:	**Jesus College**
Project Title:	**Functional Reactive Programming for Data Binding in C♯**
Examination:	**Computer Science Tripos Part II**
Word Count:	**TBC**
Project Originator:	Tomas Petricek
Supervisor:	Tomas Petricek

Original Aims of the Project

The project aimed to provide a general-purpose data binding framework for C♯ using concepts taken from functional reactive programming. Specifically, the goal was to implement a set of Haskell-style arrows and a binding framework which utilised these, and also to make invertible binding possible through the implementation of 'invertible arrows' – a two-way extension of normal arrows. A secondary aim was to make the framework (and arrow implementation) as easy to use as possible, by making the syntax concise and readable, eliminating boilerplate code and allowing easy integration with the existing WPF data binding framework.

Work Completed

All the original goals were met: a general-purpose data binding framework based on arrows has been implemented, and an extensive arrow implementation has been completed. As well as the standard operators, a series of more complex extra operators have also been added, and some additional arrow types have been included – for instance, 'list arrows' which map between enumerable data types.

The framework allows bindings in both directions, between multiple sources and multiple destinations, and the arrows can be used in conjunction with WPF data binding with reasonable ease.

[Continue this?]

Special Difficulties

None

Declaration

I, David Barker of Jesus College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Chapter 1

Introduction

It has become a firmly established practice when writing applications to separate the code managing the user interface from the business logic powering it. This is a key part of the principle of separation of concerns, and means that the interface can be safely changed without needing to modify the backend code (and vice versa, assuming the backend provides a consistent interface). Beginning with the traditional MVC architecture, this has led to the development of a family of system architectures such as MVPM and MVVM which enforce this principle.

1.1 Data binding

Data binding presents a mechanism for bridging the gap between the separated layers by allowing the developer to specify that some value in the user interface code should be bound to a property of the model. This usually simply means that whenever the value in the model changes, the data binding framework will ensure that the value in the user interface is also updated to the same value. However, bindings can often be more complex: for instance, there could be a text box in the user interface which reflects backend data but can also be used to change it (two-way binding). Alternatively, there might be a user interface component whose value is determined by some function of a variable in the model - consider a list view which displays a filtered and sorted version of a list stored in the model. There could even be values which depend on multiple sources, or more complex many-to-many bindings.

Many current languages and frameworks provide data binding features with varying levels of complexity. Java has a variety of extension libraries [examples] which allow the programmer to use data binding with Swing. Javascript too has a variety of possibilities, with a prominent example being the backbone.js MVC

framework. Unfortunately, many of these are quite limited and difficult to use - backbone.js, for example, only provides one-way binding and requires a lot of boilerplate code from the programmer.

1.2 Data binding in .NET

Microsofts .NET framework offers a particularly powerful example of data binding through Windows Presentation Foundation (WPF). Based on the MVVM architecture, it has many features to allow things like two-way binding, binding through functions and bindings based on list operations. One of its key advantages is that the user interface can be defined entirely in XAML [1] with bindings being specified through XAML parameters. The view logic is then specified in the ViewModel which in turn communicates with the model. This means user interface designers can work purely in XAML without concern for the logic or binding mechanisms in place behind the scenes.

However, WPF suffers from a similar problem to many other data binding frameworks: advanced data bindings can be very complex and difficult to manage, and setting up bindings in the first place requires quite a lot of boilerplate code in the model and view. Furthermore, binding through functions requires special value converter classes to be written. This is essentially an application of the Template pattern, and the value converters are not type safe - they take objects as input and return objects as output (and bindings with multiple inputs will simply take arrays of objects with no guarantee that the right number of values has been passed). [Maybe more disadvantages?] Clearly a more simple and general binding framework would definitely make application development simpler.

1.3 Project inspirations

Many useful ideas for data binding come from the area of functional programming. Functional reactive programming (FRP), for example, is a paradigm which nicely models the concept of data binding in a general way. Another useful concept is that of the 'arrows' implemented in Haskell. An 'arrow from type A to type B' essentially represents a process taking an input of type A and returning something of type B, and these can be arbitrarily combined in interesting ways to build up complex functions. More detail on both will be given in the next chapter.

[1] Extensible Application Markup Language, an XML-based markup language for defining user interfaces and simple behaviour

The FRP paradigm was the inspiration for the project, which sought to implement a general-purpose data binding framework in C♯ using concepts derived from functional programming. I successfully implemented a framework which provides data binding in both directions, through arbitrary functions with arbitrary numbers of inputs and outputs, all with a simple syntax based on lambda expressions. Minimal boilerplate code is required to set up bindable properties - the containing class should extend Bindable, and from there marking arbitrary member variables and properties with the [Bindable] tag will make it possible to create bindings to and from them with a single function call. I also implemented a large variety of arrow types and arrow operators along with a number of utility functions to make creating and combining them simple.

Chapter 2

Preparation

This chapter is empty still!

-Explain FRP here -Explain arrows here -Software engineering approach - iterations at first, issue-driven Agile

Chapter 3

Implementation

3.1 Overview

Give a general overview

3.2 Arrows

3.2.1 Overview

Implementing the highly functional idea of an arrow in C♯ posed an interesting challenge. Fundamentally, there were two main obstacles to overcome: C♯'s syntax, which is far clunkier than Haskell's; and the type system, which is far more restrictive and relies on static analysis at compile time. It was decided early on that arrows would be built on lambda expressions as these seemed the most natural way of expressing the desired functionality. C♯'s generic `Func<T1, T2>` type provided a good way of handling arbitrary lambda expressions between arbitrary types, and so this became the basis for the arrow type.

Tackling the type system was particularly difficult. I initially took the approach of writing arrow combinator functions so that they required no type parameters and instead inferred all their types by reflection. This made the syntax far neater and, using some run-time type checking, could plausibly be made type safe (though the compiler would have no way of enforcing it). However, after some experimenting it became clear that writing new operators and arrow types would be incredibly difficult using this approach, and the resulting code was very complicated and difficult to understand.

It was therefore decided that arrows and combinators should all be statically typed, as this would allow the compiler to do type checking and lead to much

cleaner code. However, this meant the issue of the programmer having to provide long lists of type parameters to every combinator was still there. I was eventually able to solve this problem by writing the combinators in such a way that the compiler could always infer types from the arguments without the programmer needing to supply them. Several more situations were fixed by including static extension methods which would use the source object to infer some parameters, and a series of static helper methods were included in the class `Op` to help fill in the gaps. This was easier said than done in several cases (which will be discussed later), but ultimately I managed to eliminate type parameters completely from the syntax.

A further problem was how best to allow arrows from multiple inputs to multiple outputs. Simply having a `Func` with multiple inputs would not have worked as the arrow would have had to be rewritten for every given number of inputs, and to make matters worse `Func` is limited to one output. For some time I debated writing a generic binary-tree-structured type[1] to handle this, but on discovering C♯'s built-in `Tuple<T1, T2>` type I felt this would be the simplest option. This ended up complicating the data binding (as I will discuss later), but was entirely sufficient for supporting the arrow implementation.

[Maybe mention arrow intercompatibility?]

3.2.2 Simple arrows

The first objective in implementing arrows was to get the simple function-based arrow working, as all the others derive from this. As mentioned earlier, it was implemented using the C♯ `Func<A, B>` class to hold its function. An `Invoke` method is then exposed for using the arrow.

Whilst arrows can be constructed using `new Arrow<A, B>(function)`, a simpler syntax is provided by an extension method:

```
var increment = ((int x) => x + 1).Arr();
```

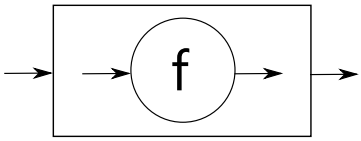

Figure 3.1: The Arr operator

The `Arr` operator effectively lifts a function to arrow status, in much the same way as the `arr` operator in Haskell. Arrow composition is also possible:

[1] All arrows work on nested pairs

```
var combinedArrow = arrow.Combine(anotherArrow);
```

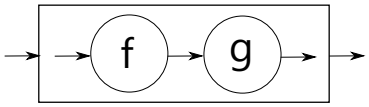

Figure 3.2: The Combine operator

This is equivalent to `arrow >>> anotherArrow` in the Haskell syntax. Implementing a `>>>` operator was unfortunately not possible in C♯ as it does not allow user-defined operators. The `first` operator was then implemented to complete the basic arrow:

```
var firstArrow = arrow.First(default(T));
```

Or, equally:[2]

```
var firstArrow = Op.First(arrow, default(T))
```

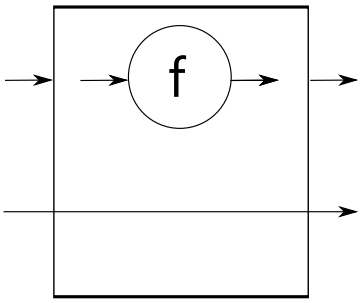

Figure 3.3: The First operator

The above syntax may not immediately make sense. Essentially, the `default(T)` passes in null if `T` is a reference type and some sensible default if it is a null type. The only purpose for this parameter is to allow the compiler to infer the type of the second argument to the resulting arrow, and so any value of the appropriate type would work – `default(T)` is used for simplicity. This is the one syntax 'glitch' which I was unable to remove, and is explained in more detail at the end of this subsection.

Going beyond the basic operators, many composite operators were also implemented based on these simple combinators. For instance, `Second` was implemented in terms of `First`. The standard parallel combinator was also implemented as `And`:

```
var parallelArrow = leftArrow.And(rightArrow);
```

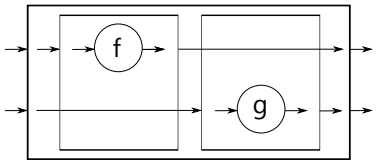

Figure 3.4: The And operator

This is equivalent to the `***` operator in Haskell, and executes the two arrows in parallel. This allows a number of complex operations to be conveniently parallelised in one arrow, which could likely be useful when working with data bindings between complex objects. The `Fanout` operator makes good use of this: given one input arrow of type `Arrow<A, B>` and two following arrows of types `Arrow<B, C>` and `Arrow<B, D>`[3], it returns an arrow which executes the input arrow, splits the result and then passes it through the two following arrows in parallel yielding a pair of results.

Given a pair result, it is then possible to recombine the two elements using a `Func<A, B, C>` operator using `Unsplit`. The following example recombines the tuple by adding its elements together:

```
var unsplitArrow = arrowOnPairs.Unsplit((x, y) => x + y);
```

An example of a fairly elaborate composite operator is `LiftA2`. This essentially performs the same function as `Fanout`, but also takes a function for recombining the two results and so is effectively a `Fanout` arrow with an `Unsplit` operation at the end.

Figures 3.1, 3.2, 3.3 and 3.4 give a visual representation of some of the arrow operators, to make their effects more clear.

Include table showing correspondence between C♯ arrow operators and Haskell ones?

Challenges encountered

As mentioned in Section 3.2.1, one of the main challenges in implementing usable arrows was eliminating the need for the programmer to specify large lists of type parameters when using arrow combinators. This was achieved by writing the combinators in such a way as to allow the compiler to infer the parameters, and in a few cases this required the use of extension methods (so that the type of the arrow it would be used with could be inferred).

[2] `Op` is the static class containing all arrow operators and extension methods

[3] Note that the input types of the two arrows have to match the output type of the first arrow

The most difficult operator to implement cleanly was `First`. The problem here is that it takes an arrow from type `A` to type `B` and turns it into an arrow on tuples, applying the arrow to one input and leaving the other untouched. In Haskell, the type of the other input needn't be specified as the function doesn't make use of it. However, C♯'s type system is far more restrictive, and in constructing an arrow one must pass in the full types of its inputs and outputs. As a result, however the operator worked it would require the programmer to specify this third type.

The first implementation simply required the programmer to specify type parameters. However, there is no way in C♯ to specify the one unknown parameter and let the compiler infer the other two, and so the programmer would have to pass in all three types. This often became very unwieldy when the types of the original arrow were complex. For instance, consider the following example where an arrow from types `A` and `B` to types `C` and `D` is used with the First operator to allow an `int` to pass alongside as a third parameter:

```
Arrow<Tuple<A, B>, Tuple<C, D>> originalArrow = [...];

var firstArrow = originalArrow
                            .First<Tuple<A, B>, Tuple<C, D>, int>();
```

It was decided that this was too cumbersome to be used in practice. The next attempt used a supplementary `First` method taking only the one unknown parameter, and using reflection to get the type of the arrow and invoke the original `First` method with its type. This works reasonably well and so was kept in the final version, but the code is incredibly messy due to all the reflection.

The final version was a modification of the first which allowed the compiler to infer the third type parameter by requiring that the user pass in a value of that type (hence the syntax shown in the last section). Although this isn't perfect, it is by far the cleanest workaround.

Another problem which was discovered was implementing standard polymorphic arrows, like an identity or one for swapping two inputs. This was also a result of the type system being too strict: to create an identity arrow, one had to specify in the definition what type it would work on. Several workarounds were considered for this, the most plausible being 'generic' arrows whose `Invoke` methods took a type parameter. However, the problem with this is that the resulting arrows would be very difficult to combine with non-generic arrows (and would most likely involve a lot of reflection). I eventually settled for implementing actual arrow subtypes for identity and other common functions, which take type parameters on construction – these are discussed further in Section 3.2.6.

3.2.3 Invertible arrows

At the time of writing the proposal, I was still unsure of the best way of implementing invertible arrows. There were two main strategies I was considering: simply requiring the user to supply an arrow for each direction, or providing a basic set of invertible functions and allowing the user to compose these together to build up more complex functions. The former wouldn't have worked very well as part of the advantage of using arrows in data binding is that simple ones can be re-used in building up multiple different bindings, and having to build up two arrows independently would be very messy and prone to error. The latter also has several problems. For instance, what functions should be made available to prevent the system being too restrictive? Also, if the functions were too simple then many would need to be combined in most cases, and this can cause a lot of unnecessary overhead (as explored in Section 4.3.1).

Ultimately, a solution roughly combining the two approaches was found. This was largely inspired by [REF], and is based on an `Arr` operator which takes *two* functions rather than one – one function for each direction. The arrows can then be combined using all the same combinators as are available for simple arrows (bar those which don't have inverses, such as `Unsplit`). As a result, it gives the same flexibility as allowing the user to simply define two arrows, but saves time by retaining the composability that makes arrows useful. An example of how an invertible arrow can be constructed using the `Arr` extension method is given below:

```
var invertibleIncrement = new Func((int x) => x + 1)
                                  .Arr((int x) => x - 1);
```

An invertible arrow can be used in the other direction by calling `Invert()` to get a new invertible arrow which is the original one reversed.

To make implementation simpler and reduce code duplication, many of the invertible arrow combinators make use of the standard arrow combinators in their implementations. The basic operators, `Arr`, `Combine` and `First`, are all overloaded so that any composite operators which use them will work for invertible arrows for free.

3.2.4 List arrows

Whilst exploring existing uses of data binding in WPF applications, it was discovered that a fairly common use case involves binding some form of list display to an enumerable data structure in the model. WPF provides support for this already, but trying to do it with arrows would be syntactically clunky – for

one thing, the arrow types would all be of the form `Arrow<IEnumerable<T1>, IEnumerable<T2>>`. To simplify this I decided it would make sense to implement an 'arrow on lists' abstracting away the actual list processing details and exposing a simple set of common list operators.

The result is a set of simple arrows implementing SQL-like functionality on enumerable data sources. List arrows are all of the type `ListArrow<T1, T2>`, which extends `Arrow<IEnumerable<T1>, IEnumerable<T2>>` for compatibility with existing arrows. There are a variety of simple list arrows which can be combined to build up complex functionality:

`FilterArrow<A>` Accepts a function from `A` to `bool`, which it uses to filter the list to those matching the predicate

`MapArrow<A, B>` Accepts a function from `A` to `B` which it uses to map all items in the list to a list of `B`s

`OrderByArrow<A>` Takes a function from two elements of type `A` to `int` and uses it to sort the list

`ReverseArrow<A>` Simply reverses the input list

For simplicity, these arrows all essentially wrap Linq queries. As well as these operators, the standard functional operations `foldl` and `foldr` were implemented to make gathering a list into a single result easier.

The list arrows are all supported by a simple syntax which has been made possible using several extension methods. For instance, given a particular `ListArrow<int, int>`, one can quickly attach a filter to it as follows:

```
var filtered = listArrow.Filter(x => x > 3);
```

In this example, the Filter extension method takes `listArrow` and the function `(int x) => x > 3`, creates a `FilterArrow<int>` out of the function, and returns a list arrow which is the original arrow and the filter arrow combined. Whole chains of list arrows can be quickly set up in this way:

```
var filtered = listArrow
                    .OrderBy((x, y) => y - x)
                    .Filter(x => x > 3)
                    .Map(x => "Number"+x)
                    .Reverse();
```

[More on foldl and foldr?]

3.2.5 Choice arrows

- Not deeply implemented or tested but could be handy for some things, most likely convenient exception handling
- This wouldn't be much good for bindings though

3.2.6 Further utility arrows

To simplify various common tasks, a number of simple utility arrows were written. These simply inherited from an arrow on the appropriate types, and initialised the arrow to some particular function in the constructor.

Identity arrows The identity arrow, though very rarely used in practice, plays a key part in several of the arrow laws. Creating it from a `Func` every time made the tests a lot messier, so a simple `IDArrow<T>` class was written which takes a type parameter and returns an identity arrow on that type. An invertible version of this was also written for the invertible arrow laws.

Swap arrow This fulfilled a need which came up surprisingly often: given two types `A` and `B`, create an arrow on a `Tuple<A, B>` which swaps them around and outputs a `Tuple<B, A>`.

Tuple reassociation arrows The tuple operations `assoc` and `cossa`[4] turn up reasonably frequently in functional programming, and feature in several of the arrow laws. As these are fairly fiddly to implement I decided it would make sense to create utility arrows for both these functions.

3.2.7 Feedback in arrows

Discuss whether this would be useful or not, reasons for it not being there (lack of real-world use cases?) and how one might implement it.

3.3 Data binding

3.3.1 Overall architecture

Use diagrams and stuff to explain it.

[4] `assoc` takes a tuple ((a, b), c) and returns (a, (b, c)), whilst `cossa` does the opposite

3.3.2 Creating bindable sources and destinations

One of the main problems with WPF data binding is the complexity of making sources 'bindable'. This usually requires that the programmer manually implement the `INotifyPropertyChanged` interface, creating appropriate events and overriding the set methods of all their properties such that they throw the right events. This often becomes a case of copying and pasting the code used for other data sources as it is almost always identical and includes boilerplate tasks like ensuring a variable's new value is different from its old one before throwing the event, and checking that the event isn't null.

For this project I decided to abstract these details away into a base class, called `Bindable`. This provides events for binding and a set of methods used by binding classes to manage data binding, many of which will be discussed later in this section. The most important are those which allow getting and setting of arbitrary variables by name using reflection, as these are the main interface used by the bindings manager. These have been designed to be dynamically type safe (that is, throwing exceptions at run time where type errors are encountered), to ensure the properties being accessed exist and to abstract away the differences between properties and simple public member variables[5].

The result of this is that instead of writing all the boilerplate code one would usually write, the programmer need only make their sources and destinations extend Bindable and all the usual things will be handled elsewhere leading to considerably less code clutter. A representative example of the final syntax is given in Listing 3.1.

Another addition that can be seen in the listing is the `[Bindable]` attribute which has been used above the `value` property. This uses a PostSharp aspect, explained in the next section, to intercept the setter for throwing events and so relieve the programmer of having to do this themselves.

Listing 3.1: Creating a bindable source class

```
public class DataSource : Bindable
{
        [Bindable]
        public int value { get; set; }
}
```

[5]WPF does not allow binding to member variables, but for this project I decided there was no obvious need to distinguish between the two

PostSharp

PostSharp is a C♯ extension which provides aspect-oriented programming (AOP) functionality. Essentially, AOP is a means of separating cross-cutting concerns from the code they affect, allowing 'aspects' to be written providing this common functionality. Markers can then be placed in the appropriate points, and the aspects will generate and inject their code into these locations at compile time.

In this case, the `Bindable` aspect will intercept the setter of any property or member variable it is placed above and insert code to check whether the value has changed and throw appropriate events if it has. This does not interfere with any setters the programmer may already have specified, as it defers to the programmer's setter before throwing the event for the value changing (thus ensuring the event is only thrown once all values are updated and any side-effects have occurred).

Unfortunately, the project budget only extended as far as the free version of PostSharp. Syntax for bindings could likely have been improved even further using the method and property introduction features provided by the full version, as this would have removed the need to extend Bindable. The unfortunate side-effect of the Bindable base class is that the class can no longer extend any other base class, as C♯ does not allow multiple class inheritance. However, in working with the binding framework it was found that this rarely caused problems that couldn't be worked around, and in any case patterns exist to overcome the lack of multiple inheritance in the general case[6].

3.3.3 Creating bindings

It was decided that it would make sense to manage all bindings in a given project centrally, so that constraints on cycles and conflicts can be enforced and bindings can be dynamically added and removed. Therefore, the `BindingsManager` class was implemented to create and maintain bindings. This is a static class handling all bindings for the project it is used in and exposing methods for creating and removing particular bindings. The programmer can call it passing in the sources, destinations and arrow they wish to use, and it will return a binding handle which they can use to remove the binding later if needed. The bindings manager also handles cycle and conflict detection, automatic creation of two-way bindings based on the arrow type and support for many-to-many bindings, all of which will be explained in this section.

[6]The 'composition over inheritance' pattern is one such technique which uses a shared base interface and a proxy object to 'inherit' from a class without using actual inheritance

Syntax and usage

In order to abstract away details like types and specific object references, the bindings manager operates on 'bind points' – essentially, structures containing a reference to the `Bindable` source and a string representation of the property or member variable being bound. An extension method was written to make creating these simpler, as calling the constructor with the object and variable name turned out to make the overall syntax fairly cluttered. The standard way of making a bind point for an object 'obj' with property 'value' is thus `obj.GetBindPoint("value")`, which is a lot simpler to write.

Basic bindings (that is, from one source to one destination in one direction) can be set up by passing in a source bind point, and arrow to use for the binding and a destination bind point. An example of this is given in Listing 3.2.

With the binding created, the bindings manager subscribes to the update event exposed by the `Bindable` base class and will thus be notified of any updates. The event is set up to contain the name of the property or variable which has changed so that multiple values from the same object can be bound to without any additional complexity, and as mentioned the `Bindable` class provides an interface allowing the bindings manager to freely access and modify those variables which have been marked as bindable. Although this largely precludes static type safety, dynamic type safety is provided wherever possible by using reflection to check that types match up and throwing explanatory exceptions where something has gone wrong.

Listing 3.2: Creating a binding between two properties

```
BindingsManager.CreateBinding(
        source.GetBindPoint("value"),
        arrow,
        destination.GetBindPoint("result"));
```

Two-way binding

Mention inference of two-way bindings by reflecting on the type of the arrow passed in. Also mention the problems encountered - need for variable locking system to avoid infinite loop

Cycle and conflict detection

When setting up a large collection of arbitrary data bindings in a complex application, there is often the risk that some of the bindings could interfere with each other. This generally occurs in one of two ways:

- A collection of bindings ends up forming a cycle, with every update to the first source property ultimately leading to an update of that same property
- One property is directly or indirectly bound to more than one source through functions which will not necessarily yield the same result

The first scenario is especially problematic for this binding framework as binding updates are performed in the same thread, so a binding cycle will lead to an infinite loop of updates causing the application to freeze. The second scenario doesn't necessarily break the application, but the resulting behaviour will be entirely unpredictable[7]. Naturally, a useful data binding framework would need to handle these situations properly.

To tackle this problem, I implemented a `BindingGraph` class which is used by the `BindingsManager` and keeps track of the topology of all the existing bindings (leaving the details of actual bind points to the bindings manager). Whenever a new binding is added or an old binding is removed, the binding graph is notified and it updates its map of the bindings appropriately. It then provides methods which allow the bindings manager to query whether a new binding will introduce a cycle or conflict, and if so the bindings manager will know to abort the binding with an exception.

Both cycles and conflicts can be reduced to the same question: for each node in the binding graph, is there another node which is reachable from it via more than one path? Both situations are therefore handled by the same algorithm, which proceeds by depth-first searching from each node and maintaining a set of nodes which have been seen along the way. Rough pseudocode for this method is given in Listing 3.3.

Listing 3.3: Pseudocode for cycle and conflict detection

```
for each node in the graph:
  seen = {}
  checkCycles(node, seen)

function checkCycles(node, seen)
  if node in seen:
    return 'cycle'
  else:
    add node to seen
    for each child node:
      checkCycles(child, seen)
```

[7]This is because the order in which bindings are updated is governed by the order in which C♯ delivers events, and this is outwith the programmer's control

Further to detecting these situations, another issue which had to be resolved was what action to take on finding a problem. Were the programmer to create a new binding between two objects which were already bound, a possible response would be to simply overwrite the old binding. This would prevent errors from the framework disrupting the user, and is also a fairly reasonable behaviour to expect. However, it was decided that this made the binding process too opaque and could lead to confusing behaviour if the programmer had accidentally overwritten a binding – for instance, by mistyping the name of one of the objects they intended to bind. Therefore, whenever the programmer attempts to create a binding which would lead to a cycle or conflict, the framework throws an exception explaining the error. The behaviour of 'overwriting' an old binding is still possible as the `BindingsManager` provides a method for removing specific bindings, but now the programmer has to do the overwriting explicitly.

Many-to-many bindings

Many-to-many bindings presented an additional challenge for the binding system. It was decided that the best approach in terms of syntax would be to have the user pass in a list of source bind points, an arrow and a list of destination bind points, so as to mirror the syntax used in simple bindings. However, this simple approach is out of step with the way parameters are passed to arrows: by construction, all arrows taking multiple inputs will take them in the form of a binary-tree-structured Tuple[8]. As there didn't appear to be any C♯ syntax allowing the user to pass in binding sources in a similar fashion (such that the structure could be understood by the binding), it was decided that an argument marshaller/unmarshaller would be needed to handle the translation between lists of bind points and tree-structured tuples.

By using reflection on the type of the arrow, I was able to convert a list of bind points to a binary-tree-structured Tuple. The code essentially does a recursive depth-first search through the input type of the arrow (obtained via reflection), simultaneously constructing a Tuple of the same structure and inserting values from the bind points wherever a 'leaf' is encountered in the type being searched. At the other end, a similar process is used to extract the values from the returned tuple and put them into a list which is then used to assign to the destination bind points. The process is illustrated in Figure 3.5, where a set of inputs of types A,

[8]This can be verified by looking at the implementations of the operators; for instance, And converts two arrows (which may be on tuples) into an arrow on two-tuples containing the types of the original arrows, thus building up a binary tree structure

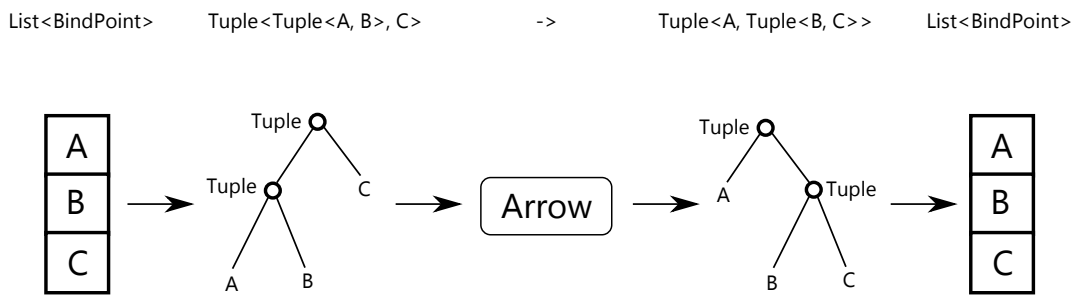

Figure 3.5: Marshalling a list of BindPoint sources to pass to an arrow and unmarshalling the result

B and C are being passed into an arrow from type `Tuple<Tuple<A, B>, C>` to type `Tuple<A, Tuple<B, C>>`.

Problems encountered

Type safety All sort of problems here; difficulties of type checking arrows should also be mentioned.

Binding to lists They don't throw events! Binding to lists needs the list to be assigned to for it to update. Possible solutions like ObservableEnumerable (or whatever it was called), potentially more boilerplate, future work n shit.

3.3.4 Integration work with WPF

Mention difficulties with the value converter and inability to properly integrate it into XAML and stuff.

Chapter 4

Evaluation

4.1 Correctness of arrow implementations

The key requirement for an arrow type satisfying the original definition by Hughes is that it conforms to a set of 'arrow laws' defining how the various basic combinators should work. While there are numerous variations on the set of arrow laws, the standard ones used by Haskell are given in [REF]. These are listed in Appendix A, along with their equivalent definitions using the C♯ syntax.

As invertible arrows conform to a slightly different set of arrow laws (which put extra requirements on their inverses), both one-way arrows and invertible arrows had to be tested separately. List and choice arrows were not separately tested as they are entirely built upon normal one-way arrows, and so will conform to the arrow laws for free if simple arrows do.

[Mention correctness proof by decomposition into lambda calculus?]

4.1.1 Automated testing

The main method of testing the arrow laws was using a set of unit tests, each testing one of the arrow laws using a large set of randomly-generated inputs. These were in most cases a simple case of generating the appropriate arrows for each side of the equality, pushing the set of inputs through both and checking that the outputs matched up. A few sample cases are given in the following subsections to illustrate the techniques used.

The tests are all supported by a large set of utility methods – for instance, `AssertArrowsGiveSameOutput` and `AssertInvertibleArrowsGiveSameOutput` run the arrow comparison algorithms, whilst `AssertArrowEqualsFunc` checks that an arrow performs the same operation as a supplied C♯ `Func`.

Simple arrows

The laws for simple arrows can be found in Appendix A.1. The first few of these are fairly simple, but there were some which were more complicated to test – indeed, the `SwapArrow` and `AssocArrow` arrows were mainly added to make these tests more readable and simpler to write.

One interesting case was the identity law – in C♯ notation:

```
new IDArrow<T>() ≈ id
```

This differs from the other laws because it has to be proven true over all *types* rather than simply for all inputs. As such, the test proceeds by obtaining a representative list of types and using reflection to build an identity arrow for each, then uses the standard technique of firing lots or random inputs at it and asserting that the arrow does indeed represent the identity function. The set of types used was just the set of primitive types, obtained by querying the current assembly for all available types and filtering the non-primitive ones (as non-primitive types would have to be initialised to null, which defeats the purpose of testing correctness over all types).

To reduce complexity, 'random' arrows were produced by randomly selecting a function from a set of pre-defined functions and constructing an arrow with it. It seems reasonable to assume that if an arrow law were to hold for some combination of these functions, but fail on another, then the problem is more likely with the C♯ compiler than with the arrow implementation.

Invertible arrows

Invertible arrow laws are listed in Appendix A.2, again with their C♯ equivalents. These were slightly more complicated to test than simple arrows as the laws have to hold for inverses as well.

[TBContinued]

4.1.2 Correctness proof by decomposing into lambda calculus

Explain the technique used to translate into lambda calculus and the likely correctness of this, and give a few sample proofs for some non-trivial arrow laws.

4.2 Syntax evaluation

4.2.1 Arrow syntax

Arrow syntaxy stuff.

Comparison with Haskell

Might not be great, but why not?

4.2.2 Binding syntax

Pretty good. Say something about the demo apps and stuff:

Username two-way binding

Todo

List binding from a mock database

Todo

Some other demo

Todo

4.3 Performance testing

4.3.1 Arrow performance

Tests indicating how arrows perform vs. functions and funcs.

Measuring technique

Explain how the results were obtained

Simple function results

The simple function results can be seen in the graph.

List function results

Results for list-based functions

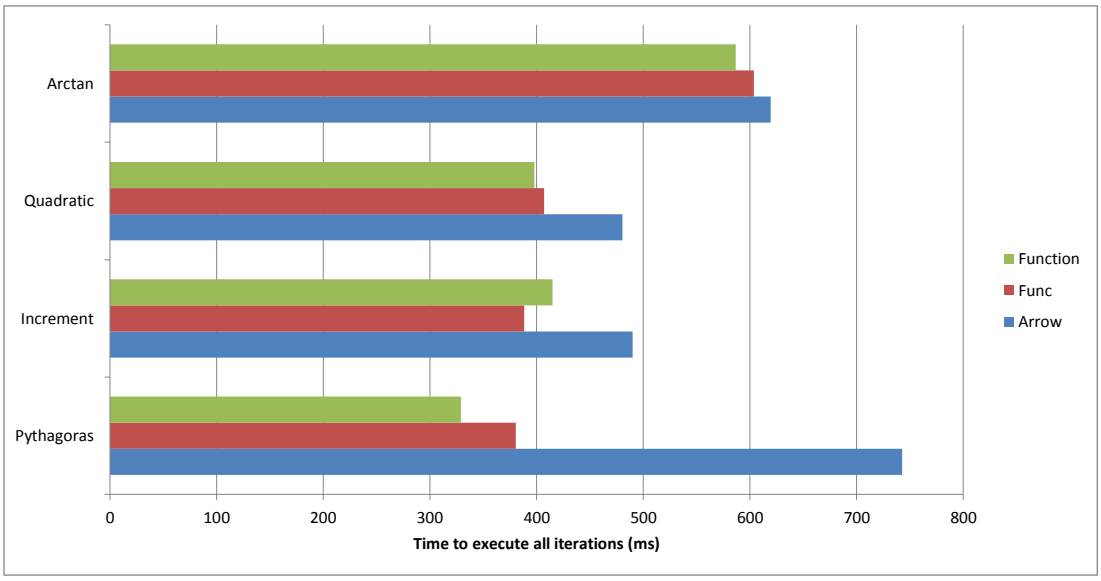

Figure 4.1: Performance of arrows, Funcs and normal functions in implementing simple functionality

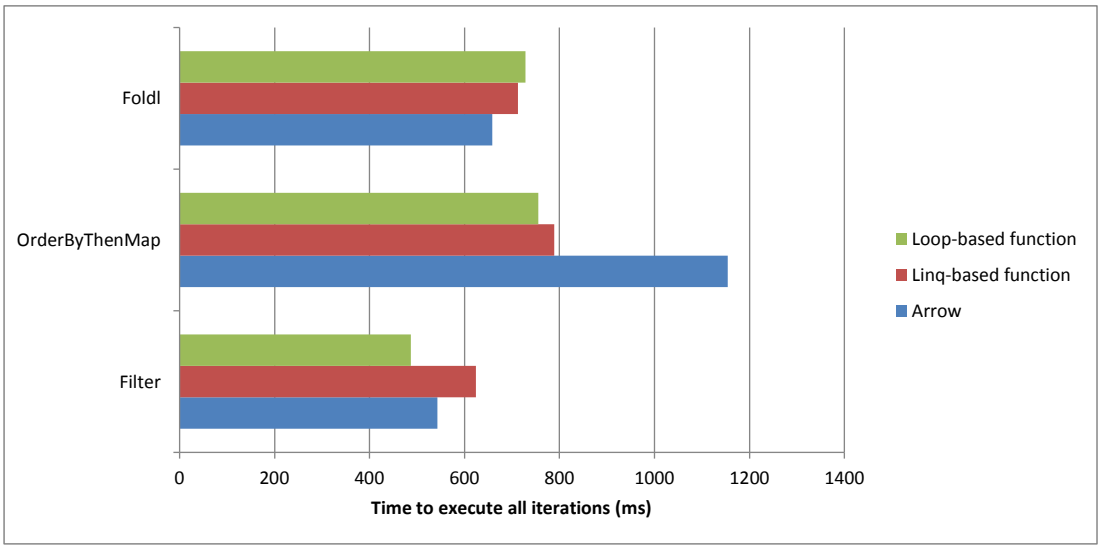

Figure 4.2: Performance of arrows, Linq queries and normal (loop-based) functions in implementing simple list functionality

Overhead due to arrow chaining

Explanation of the problem (which will have been highlighted in earlier results) along with test results for increasingly long chains of identity functions. Suggest reasons for this and potential solutions (or maybe the solutions should come in

later?)

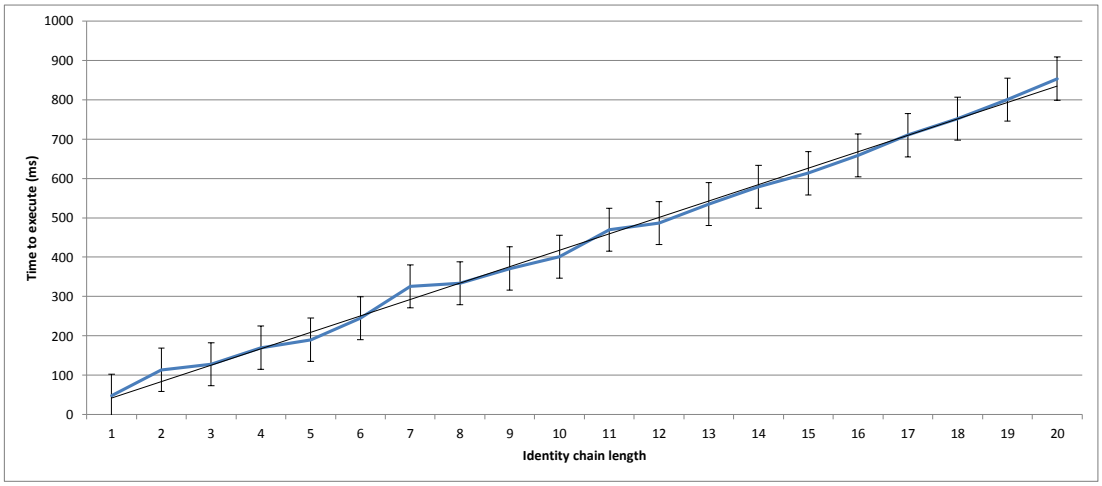

Figure 4.3: Execution times of chains of identity functions

4.3.2 Binding performance

Todo...

Chapter 5

Conclusion

Conclusion goes here!

5.1 Future work

- Performance issues - optimisations possible with dynamic expression trees?
- Messiness caused by use of Tuples - would be better with a custom tree type?
- Feedback arrows
- Better integration with WPF
- etc...

Appendix A

Arrow laws

Here the arrow laws used in testing are presented both in Haskell syntax and as they were translated into C♯. For convenience, the following definitions are used:

```
id = new Func<T, T>(x => x)
```

That is, id represents a C♯ identity function.

```
first_f = new Func<Tuple<T, T>, Tuple<T, T>>(
  tuple => Tuple.Create(f(tuple.Item1), tuple.Item2))
```

This defines a `Func` which invokes a function 'f' on the first element of a tuple.

A.1 Normal arrow laws

```
Identity
arr id = id
new IDArrow<T>() ≈ id

Arr operator distributivity
arr (f >>> g) = arr f >>> arr g
Op.Arr(x => g(f(x))) ≈ Op.Arr(f) .Combine( Op.Arr(g) )

First operator distributivity
first (f >>> g) = first f >>> first g
f .Combine(g) .First<T>() ≈ f .First<T>() .Combine( g.First<T>()
   )

Associativity of arr and first operators
first (arr f) = arr (first f)
Op.Arr(f).First<T>() ≈ Op.Arr ( first_f )
```

```
Piping commutativity
first f >>> arr (id *** g) = arr (id *** g) >>> first f
f.First<T>() .Combine( id.And(g) ) ≈ id.And(g) .Combine( f.First<
   T>() )

Piping simplification
first f >>> arr fst = arr fst >>> f
f.First<T>() .Combine( Op.Arr(tuple => tuple.Item1) ) ≈ Op.Arr(
   tuple => Tuple.Item1) .Combine( f )

Piping reassociation
first (first f) >>> arr assoc = arr assoc >>> first f
f .First<T>() .First<T>() .Combine( new AssocArrow<T>() ) ≈ new
   AssocArrow<T>() .Combine( f .First<T>() )
```

A.2 Invertible arrow laws

Todo