

David Barker

**Functional Reactive
Programming for Data Binding
in C#**

Computer Science Tripos Part II

Jesus College

April 29, 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.

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

Date

Contents

| | | |
|----------|--|----------|
| 1 | Introduction | 1 |
| 1.1 | Data binding | 1 |
| 1.2 | Data binding in .NET | 2 |
| 1.3 | Project inspirations | 2 |
| 2 | Preparation | 5 |
| 2.1 | Background theory | 5 |
| 2.1.1 | Functional reactive programming | 5 |
| 2.1.2 | Arrows | 5 |
| 2.2 | Requirements analysis | 6 |
| 2.3 | Software engineering approach | 7 |
| 3 | Implementation | 9 |
| 3.1 | Overview | 9 |
| 3.2 | Arrows | 9 |
| 3.2.1 | Overview | 9 |
| 3.2.2 | Simple arrows | 10 |
| | Challenges encountered | 12 |
| 3.2.3 | Invertible arrows | 14 |
| 3.2.4 | List arrows | 15 |
| 3.2.5 | Further utility arrows | 16 |
| 3.3 | Data binding | 16 |
| 3.3.1 | Overall architecture | 16 |
| 3.3.2 | Creating bindable sources and destinations | 18 |
| 3.3.3 | Creating bindings | 19 |
| | Syntax and usage | 20 |
| | Two-way binding | 20 |
| | Cycle and conflict detection | 21 |
| | Many-to-many bindings | 22 |
| | Problems encountered | 23 |

| | | |
|----------|---|-----------|
| | Type safety | 23 |
| | Binding to lists | 24 |
| 3.3.4 | Integration work with WPF | 24 |
| 4 | Evaluation | 27 |
| 4.1 | Correctness of arrow implementations | 27 |
| 4.1.1 | Automated testing | 27 |
| | Simple arrows | 28 |
| | Invertible arrows | 28 |
| 4.1.2 | Correctness proof by decomposing into lambda calculus . . | 29 |
| 4.2 | Syntax evaluation | 29 |
| 4.2.1 | Arrow syntax | 29 |
| | Comparison with Haskell | 30 |
| 4.2.2 | Binding syntax | 30 |
| | Username two-way binding | 31 |
| | List binding from a mock database | 32 |
| | Some other demo | 33 |
| 4.3 | Performance testing | 33 |
| 4.3.1 | Arrow performance | 33 |
| | Measuring technique | 34 |
| | Simple function results | 34 |
| | List function results | 36 |
| | Overhead due to arrow chaining | 36 |
| 4.3.2 | Binding performance | 37 |
| 5 | Conclusion | 39 |
| 5.1 | Future work | 39 |
| 5.1.1 | Performance enhancements | 39 |
| A | Arrow laws | 41 |
| A.1 | Normal arrow laws | 41 |
| A.2 | Invertible arrow laws | 42 |
| B | WPF integration demo – code listings | 43 |
| B.1 | Creating the bindable source | 43 |
| B.2 | The binding arrow | 43 |
| B.3 | Doing the binding | 43 |

| | | |
|----------|---|-----------|
| C | Code samples for case studies | 45 |
| C.1 | Username two-way binding | 45 |
| C.1.1 | Creating the bindable data source | 45 |
| C.1.2 | Creating the functionality for splitting the name | 46 |
| C.1.3 | Creating the binding | 47 |
| C.2 | List binding from a mock database | 49 |
| C.2.1 | Creating the bindable ‘database’ | 49 |
| C.2.2 | Filtering and mapping the list | 49 |
| C.2.3 | Creating the binding | 50 |
| C.3 | Some other demo | 51 |

List of Figures

| | | |
|-----|--|----|
| 3.1 | The Arr operator | 11 |
| 3.2 | The Combine operator | 11 |
| 3.3 | The First operator | 11 |
| 3.4 | The And operator | 12 |
| 3.5 | Correspondence between Haskell and C# arrow operators | 13 |
| 3.6 | Overview of the binding method | 17 |
| 3.7 | Marshalling a list of BindPoint sources to pass to an arrow and unmarshalling the result | 23 |
| 3.8 | WPF integration demo | 25 |
| 4.1 | The username two-way binding application | 31 |
| 4.2 | The list binding application | 33 |
| 4.3 | Performance of arrows, Funcs and normal functions in implement- ing simple functionality | 35 |
| 4.4 | Performance of arrows, Linq queries and normal (loop-based) func- tions in implementing simple list functionality | 37 |
| 4.5 | Execution times of chains of identity functions | 38 |

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.

¹Extensible 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

2.1 Background theory

2.1.1 Functional reactive programming

Functional reactive programming (FRP) was one of the initial inspirations for the project. A good introduction is given by [<http://conal.net/fran/tutorial.htm>], which demonstrates how the principles can be applied to animation. In essence, functional reactive programming is a declarative way of specifying the relationships between different values. 'Signals' which change over time are created and passed around as first-class values, and other values can then be defined as functions of these signals.

One of the main advantages of FRP is that it removes the need for explicitly defining how values are to be kept consistent and writing code to make this so. This is clearly solving a very similar problem to data binding, and so inspired the idea of making data binding more like FRP.

2.1.2 Arrows

Arrows were introduced by Hughes as a generalisation of `monads`. A monad takes the form `m a`, representing some computation which will give a result of type `a`. These are often useful in representing sequences of operations and those which involve side-effects. The basic monad type interface provides an identity function, which gives a monad returning whatever its input was, and a means of chaining monadic functions together. One way of viewing these is as a way of letting the programmer chain together functions in a way controlled by the monad – for instance, the monad could be written to ensure that some form of global state is passed through along with the function results, or it could be used to handle

error conditions without the programmer needing to write extra code for this. In all cases it is essentially handling some side-effect or background operation for the programmer, allowing them to chain various operations together without explicitly handling the side-effects.

Arrows represent a generalisation of monads, in the sense that any monad can be equivalently implemented as an arrow. However, an arrow is of the form `arrow b c`, taking an input of type `b` and transforming it into an output of type `c`. The basic operators are similar: an `arr` operator takes a function as an argument and returns the arrow representing that function, and the `>>>` operator chains two arrows together such that the output of the first is passed through the second. Further to these, the `first` operator takes an `arrow b c` and returns an `arrow (b, d) (c, d)` – an arrow taking a pair and applying the original arrow only to the first element of the pair. With these basic operators, a number of more complex standard operators can be built up. These are discussed in Section 3.2.2 on the implementation of arrows.

Arrows were selected for this project as the semantic nature of an `arrow a b` is simply a transformation from an input of type `a` to an output of type `b`, an interpretation which works well as a generalised model of the nature of data binding.

2.2 Requirements analysis

From the outset, the main objectives of the project were to make the data binding framework syntactically clear and free from boilerplate, and to make the arrows as powerful and easy to use as possible. Therefore, some time was spent at the beginning of the project establishing what could be done to achieve this and just what sort of functionality would likely be needed.

To start off, I spent some time understanding how data binding is already done in .NET's WPF framework. It was clear that the existing approach involved a lot of repetitive coding from the programmer to set up bindable sources and destinations, and there were several different conflicting ways of actually creating the binding – either in the XAML code describing the user interface or in the code-behind. Thus I decided a good first objective would be to find a way of making the framework completely general, with one consistent way of creating bindings which handles all the situation-specific details itself. I also focussed on ways of removing the usual boilerplate code for creating sources and destinations, replacing it with a base class which manages events and interactions and an attribute which can be placed on properties the programmer wishes to bind.

I also investigated what sort of features modern data binding frameworks implement, and which would be most useful for an arrow-based one. Binding through functions is not a new concept by any means, and indeed two-way binding is already implemented in .NET. However, in most cases these are fairly messy to set up – in .NET, for instance, large `ValueConverter` classes have to be written to perform the transformation. It was decided that the cleanest way of handling this would be simply requiring that all bindings have an arrow attached. This way, the syntax is consistent whether the binding is direct or functional, and of course it is easy to pass in an identity arrow for a non-functional binding.

Exploring existing libraries highlighted a couple of extra features which I had not previously thought would be necessary: arbitrary many-to-many bindings, which are useful in a variety of situations, and list bindings. The latter refers to the situation where the data source is actually a list of data, and would have been possible with normal arrows. However, it would have been fairly messy to do, and simple tasks like filtering and mapping would have required large loops. Given list binding is done fairly often in practice, I decided it would make sense to add an arrow type for handling them and a variety of list processing methods.

For the arrows, the requirements were taken from the existing, successful implementation in Haskell. Essentially, I sought to implement all the main combinators and make the syntax as clear as possible. The main obstacle to keeping the syntax clean was the type system, and so much of the preparation time for arrows was spent devising the best approach for minimising type parameters and making the operators compatible. This stage of the project is described in greater detail in Section 3.2.1.

2.3 Software engineering approach

The implementation work was roughly done in two phases. In the early stages, a spiral model was used as it was unclear what the best way of implementing the arrows and binding framework would be. The first iteration was completed before the beginning of the project, and featured a very limited arrow class which worked on integers and supported only the basic combinators, and a binding framework which couldn't yet use arrows and required the sources and destinations to be special `BindingSource` and `BindingDestination` objects. Over several iterations the approach was refined until I found a suitable general and extensible implementation.

From there, development was issue-driven Agile with sprints defined to be the work packages designated at the start of the project (though many of these work

packages were altered and expanded slightly). The FogBugz¹ issue tracker was used to manage this and keep track of the various milestones. The project was tested throughout development with a series of unit tests, and I frequently tried implementing complex data bindings to ensure that new features being added were compatible with what was already there and the syntax was still reasonably clean.

The entire project (dissertation included) was stored in a Git repository which was backed up on external storage and online with BitBucket².

¹www.fogcreek.com/fogbugz

²www.bitbucket.org

Chapter 3

Implementation

3.1 Overview

The implementation work consisted of two main parts: the arrows and the binding framework. As mentioned, these were both initially developed through repeated iterations as different strategies were being investigated and new options discovered. This chapter describes the main development work and highlights the various difficulties encountered along the way.

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 type¹ 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.

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();
```

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:

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

This is equivalent to `arrow >>> anotherArrow` in the Haskell syntax. Implementing a `>>>` operator was unfortunately not possible in C# as it does not allow

¹All arrows work on nested pairs

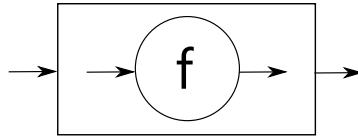


Figure 3.1: The Arr operator

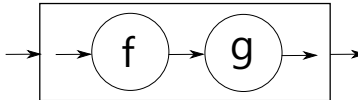


Figure 3.2: The Combine operator

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))
```

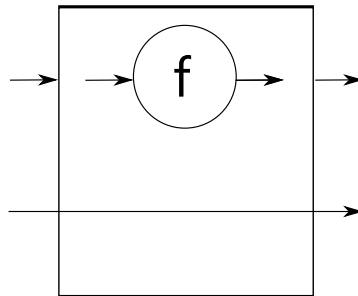


Figure 3.3: The First operator

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 imple-

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

mented in terms of `First`. The standard parallel combinator was also implemented as `And`:

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

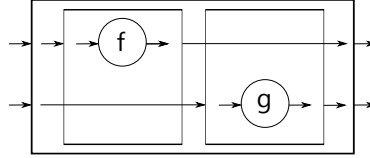


Figure 3.4: The `And` operator

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 with a binary operator (passed as a `Func<A, B, C>`) 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. A correspondence between Haskell arrow operators and the C# equivalents is given in Figure 3.5.

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

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

| Haskell | C# |
|-------------------------------|---------------------------------------|
| <code>arr f</code> | <code>f.Arr()</code> |
| <code>a >>> b</code> | <code>a.Combine(b)</code> |
| <code>first arrow</code> | <code>arrow.First(default(T))</code> |
| <code>second arrow</code> | <code>arrow.Second(default(T))</code> |
| <code>a *** b</code> | <code>a.And(b)</code> |

Figure 3.5: Correspondence between Haskell and C# arrow operators

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).

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 polymor-

phic 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 3.2.5.

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

3.2.5 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 **cosssa**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.

3.3 Data binding

3.3.1 Overall architecture

The basis for the data binding system is a publish/subscribe network, where bound values publish events when they are updated and bindings update on receiving these events. A couple of alternatives were considered for this: the first idea was to have binding sources hold a reference to their target and a copy of the arrow associated with the binding, allowing them to simply update the referenced value via the arrow when needed. However, **C#** lacks the ability to hold references as member variables in this way. The simple approach of simply

⁴**assoc** takes a tuple ((a, b), c) and returns (a, (b, c)), whilst **cosssa** does the opposite

periodically polling all bound variables to check for updates was also considered, but this was clearly no good: polling is system-intensive and wasteful with many bindings using the same source, a binding target's value would potentially be 'invalid' between polling updates, and the bindings manager would need to store a copy of every bound variable.

Conveniently, C# provides simple syntax for event-based programming. Events can be created for a class and thrown with a method call, and other objects can subscribe and unsubscribe simply by adding or removing their handler methods from the event. The clearest approach was therefore to attach events to bindable objects and have them trigger whenever a bound variable is updated. The bindings manager, when creating a binding, need only tell the binding to subscribe to the source object's event and it will then be notified of all updates to variables. Information stored in the event arguments is used to filter out events from variables the binding is not watching, and the updated value is passed through the binding's arrow and then assigned to the destination variable. This approach is shown in Figure 3.6.

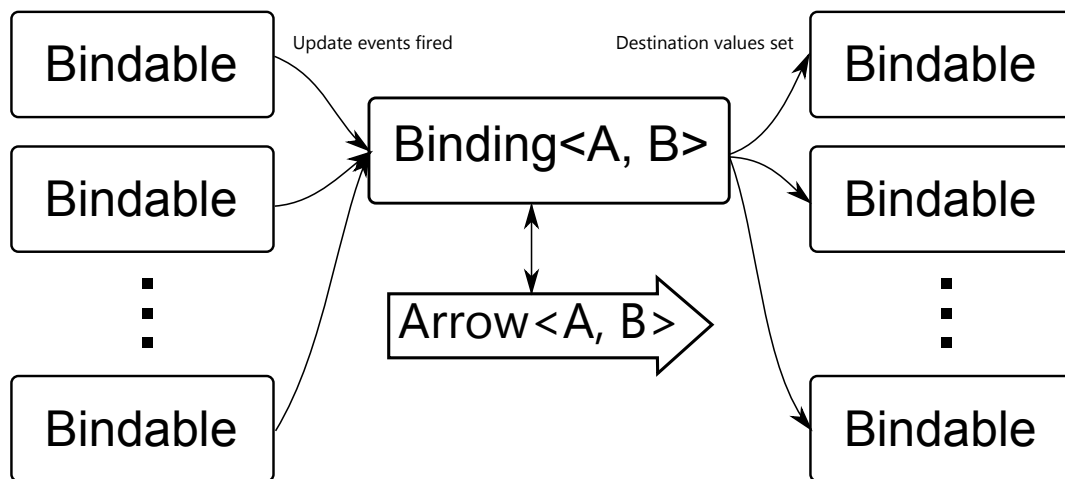


Figure 3.6: Overview of the binding method

An advantage of this technique is that sources and destinations can be treated in exactly the same way: both extend the `Bindable` base class, which handles events for binding updates and allows `Binding` objects to set the values of its bindable variables.

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 variables⁵.

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; }
}
```

PostSharp

PostSharp is a C# extension which provides aspect-oriented programming (AOP) functionality. Essentially, AOP is a means of separating cross-cutting concerns

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

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 case⁶.

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.

⁶The '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

For simplicity, two-way bindings are inferred based on the type of arrow passed in; the function for creating bindings is overloaded such that passing in an invertible arrow will automatically set the binding up as two-directional. Two-way bindings are accomplished in much the same way as one-way ones, with the difference being that the binding subscribes to events from both sides and decides which way to update based on the binding point which triggered the update event.

The main difficulty encountered here was that these bindings initially led to infinite loops. If we consider a two-way binding between 'A' and 'B', whenever 'A' is updated the binding will then update 'B'. However, the setter for 'B' has been

modified to throw an event when it's updated signalling the binding to update, and so the binding is then forced to update 'A' again. This continues back and forth in the main thread causing the whole application to hang.

The solution to this was to allow variables to be *locked* when they are first updated. Essentially, the **Bindable** base class maintains a list of all the bindable variables it manages and holds a 'locked' flag for each. Setting a variable through the methods supplied by **Bindable** will have no effect when the variable is locked, and so loops can be prevented simply by modifying the variable setter to lock, set and unlock the variable. Considering the previous example with this addition, when 'A' changes it will first be locked and then set. 'B' will then be updated by the binding, and will throw an event signalling the binding to update 'A'. However, the binding's attempt to update 'A' will fail and the loop will stop, unwinding back to 'A' which is then unlocked.

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 unpredictable⁷. 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

⁷This 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

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)
```

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 Tuple⁸. 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.

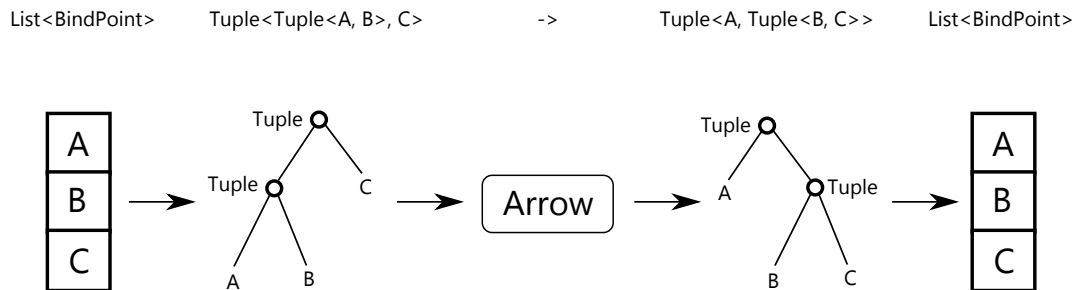


Figure 3.7: Marshalling a list of BindPoint sources to pass to an arrow and unmarshalling the result

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 3.7, where a set of inputs of types A, 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 The binding framework at the moment is unfortunately not statically type safe: binding sources and destinations are of type BindPoint, which doesn't say anything about the actual type of the variable it refers to. An obvious solution would be to simply make the BindPoint class parameterised by the type of the variable it refers to (e.g. BindPoint<int> would refer to an integer variable), but this approach removed the ability to treat sets of bind points

⁸This 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

polymorphically which is used heavily by the methods for creating many-to-many bindings. As a result, it was decided that ensuring the types are correct should be left to the programmer. However, I was able to implement dynamic type checking at runtime so the framework will throw explanatory exceptions whenever the programmer has made a mistake rather than simply failing unexpectedly.

Binding to lists Unfortunately, the technique of intercepting assignments to trigger binding update events doesn't easily work for lists. The vast majority of list operations occurring in practice involve changing elements of the list rather than reassigning the list variable itself, and so the list's overridden assignment method will not be called. This means that list bindings will not update unless the programmer explicitly assigns a modified version of the list to itself:

Listing 3.4: Explicitly reassigning a list to trigger a binding update

```
var tempList = listToBeUpdated;
tempList[3] = newValue;
listToBeUpdated = tempList; // Triggers the binding update
```

This can be overcome by using an `ObservableCollection` instead of a list. An `ObservableCollection` is essentially the same as a normal list but provides a `CollectionChanged` event to which listeners can subscribe for notifications whenever the contents of the collection change, and so it would be fairly simple to have list bindings subscribe to these events and update in that way. However, this would require that all bound collections be of type `ObservableCollection` – far more limiting than the current system which only requires that they be subtypes of `IEnumerable`, the base type of all collections.

3.3.4 Integration work with WPF

As the standard WPF binding method is fairly widely-used and naturally compatible with WPF, it was decided that arrows should be made compatible with it. WPF's standard approach to binding through a function is by creating a class extending the `IValueConverter` interface which implements the desired functionality through its `Convert` and `ConvertBack` methods. It was therefore reasonably simple to implement a value converter which allowed an arrow to be used for the conversion. However, this cannot be used when defining bindings in XAML as it does not allow constructor parameters to be passed in (it is possible to *set* parameters in XAML, but this is a clunky workaround and the application would crash if the arrow was left unset). Creating WPF bindings with arrows is therefore limited to the code-behind.

One area where WPF integration went particularly well is in removing the boilerplate usually required to make a source bindable. In general, the programmer has to set up an event to be triggered when the bound variable changes and override the setters of all bound variables so that they throw this event. The method throwing the event also has to pass in the name of the variable triggering it (for disambiguation) and ensure that the event is not null (which occurs when it has no subscribers). This is usually combined with a method which stores variables' old values and checks that they have actually changed to avoid redundant updates. I was able to extract all this functionality into the `Bindable` base class and the `PostSharp` aspect, thus removing all the standard boilerplate. Variables can now be made bindable simply by making their class extend `Bindable` and placing the `[Bindable]` tag above them.

To demonstrate the WPF integration, a simple demo application was made in which a constantly incrementing 'time' variable is bound through a sin function to a progress bar. This is pictured in Figure 3.8, and the main source code listings can be found in Appendix B.



Figure 3.8: The WPF integration demo – the progress bar goes sinusoidally up and down

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.

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 of 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. They also required random arrows with inverses provided as well. However, aside from being slightly longer to write the laws were generally fairly simple to test. As a small example, the test for double inversion is given below:

```
public static bool TestDoubleInversion()
{
    var arr1 = Op.Arr((int x) => x * 7, (int y) => y / 7);
    var arr2 = arr1.Invert().Invert();

    return ArrowTestUtils.AssertInvertibleArrowsGiveSameOutput(arr1, arr2);
}
```

Including minor variations from different sources, 24 different arrow laws were tested in total, and all passed on large quantities of random input data. This gives a reasonably good indication that the arrow implementation is correct.

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

Syntax is a difficult thing to give an objective evaluation of without extensive user trials and a variety of different opinions, and there are many conflicting metrics which could be used to judge its quality. However, the way this project was originally set out allows for a certain degree of objective syntax evaluation: brevity was a key goal, along with removing the need for type parameters and providing syntax for all the standard arrow operators. This section will therefore evaluate the syntax of the arrow implementation in terms of these key features, and in doing so highlight how well the project's original goals were met. A comparison with Haskell's syntax is also provided, taking into account the limitations inherent in C# for fairness.

One of the main syntactic achievements of the project was finding ways of writing all the arrow combinators such that under normal circumstances the user need never provide explicit type parameters. Given a typed lambda expression, for instance `(int x) => x + 1`, arrow construction is done with `Op.Arr((int x) => x + 1)`. From there, the type of this arrow is used to infer the types required for all the combinators – composition, for example, is simply `arrow.Combine(otherArrow)`.

Less successful was the `First` operator. As outlined in Section 3.2.2, many approaches were attempted for this, but all turned out slightly messy. The current preferred approach is `arrow.First(default(T))`, where `T` is the type parameter which cannot be inferred. However, `arrow.First<T>()` is arguably slightly better as it is clearer what it does (passing `default(T)` definitely qualifies as a ‘hack’), and so it is used in the arrow law listings of Appendix A for clarity. The main problems are that it takes an extra type parameter and relies heavily on reflection, leading to slower execution. This unfortunately leaves the `First` operator very inconvenient to use (so much so that the arrow law listings define an abbreviation for it). However, this was fortunately the only bad example and could probably be improved by using a slightly different operator in place of it¹.

¹For instance, consider an operator `FirstThrough` which takes the arrow and then an arrow

A slightly inconvenient (if sensible) feature of C# is the restriction that, unlike in many other similar languages, all methods have to be called on an object or class. Whilst in Java one could import all static methods from a class and use them as though they were defined locally, in C# they need to be prefixed with the class name (i.e. `StaticClass.Method()`). This of course presented a slight obstacle to making arrow code concise, as chains of operators would build up into messy expressions like `Combinators.And(Combinators.Arr(f1), Combinators.First(Combinators.Arr(f2)))`. By writing all combinators to be usable as extension methods, writing utility extension methods (like `Func.Arr()` to make a `Func` into an arrow) and giving the class a short name for the worst case, I was able to keep this to a minimum and so keep combinator-heavy code relatively clean. The `Op.Combinator()` syntax is still fairly clumsy, but in many cases there was no easy way around this.

Comparison with Haskell

Firstly, mention difference in specifying lambdas and shizz (C sharp stunted from the start). From there, show how C sharp arrows create arrows from them (need types specified somewhere is an issue (no type inference like Haskell gets), C sharp also has a strict function calling syntax which doesn't allow Haskell's pretty currying syntax). Go on to give a few examples of complicated-ish arrows expressed in both? Also, maaaybe mention the shiny new syntax Haskell has for arrows, but the C sharp implementation can't really battle that so it's not worth going into much detail about.

4.2.2 Binding syntax

The goals for binding syntax were similar to those for arrows: fundamentally, the objective was to make the syntax brief and clear. Specifically this meant removing all the boilerplate associated with WPF data binding and similar frameworks, whilst making it easy to understand what's going on. To evaluate how concise and readable the binding syntax is, a few case study applications were written using both normal WPF binding and arrow-based binding. The key code snippets from the resulting programs are listed in Appendix C, and this section explores how the samples differ and how successful the attempt to simplify binding syntax and reduce boilerplate code was.

on pairs to pass the result through. The type of the arrow on pairs could be used by the compiler to infer the type missing from the `First` arrow preceding it, and the programmer could easily just supply an identity arrow of the appropriate type if they wanted a pure `First` operator

Username two-way binding

The first case study was relatively simple: given a data source containing a full name (forename and surname separated by a space), bind this information to two text boxes in the interface such that one shows the forename and the other shows the surname. To demonstrate that the binding was working, I also added a button which triggers a change in the name stored in the data source. A screenshot of the resulting application is given in Figure 4.1, and code snippets are given in Appendix C.1.

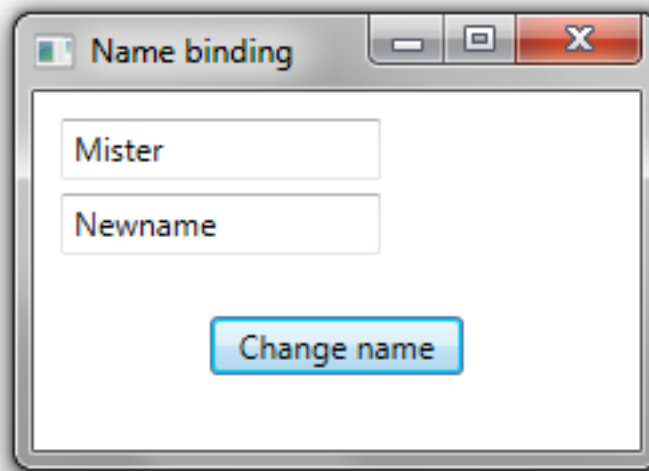


Figure 4.1: The username two-way binding application

This case study revealed a weakness in WPF’s data binding, as it turned out to be impossible to do an invertible one-to-many binding linking the name data in the data source with the text box in the user interface². It was possible to do the binding with two one-way bindings, one for each name, but reversing the binding – taking the names, concatenating them and assigning this to the variable in the data source – proved impossible without hacking around the inability to create invertible one-to-many bindings.

Looking at the code for creating the data source, it is clear that the arrow-based framework has considerably less boilerplate – setting the name string up as bindable is a simple case of putting the `[Bindable]` tag above it, whereas the WPF approach involves creating an event, writing a method to throw that event if it exists and overriding the variable’s setter to trigger that method. A

²The `IValueConverter` interface’s method for updating the reverse binding can take only one input, and so bindings have to be many-to-one

real-world application would probably also have code to ensure the variable had actually changed value before calling the method.

Setting up the arrow is then a simple method call with two lambda functions defining the function either way, whilst the WPF version has two large `ValueConverter` classes for the forename and the surname. The arrow syntax is denser, however, which may make it harder to read – it would perhaps be beneficial to allow invertible arrows to be constructed from two normal arrows so that the code can be more neatly separated.

For creating the actual binding, it's not clear which is better. The arrow-based code is shorter, but also more complicated due to the need to construct arrays of sources and destinations. However, the comparison is not entirely fair – the WPF code is simply creating two one-to-one bindings, and if the same was being done with the arrow code it would be a lot neater than it is as the bind points wouldn't need to be put into arrays.

List binding from a mock database

For a second case study, a simple 'database' (a `List` of data for simplicity) containing information on orders placed for certain products was created. The objective for this was to bind this data to a list view, filtering it to only the orders with volume greater than one and mapping the result to a list of user-friendly strings describing the order (of the form "Order from [name] in [location] for [volume] '[product]' from [supplier]"). The simple application is pictured in Figure 4.2, and its source code is given in Appendix C.2.

As with the last example, the arrow syntax is a lot more concise than the value converter. Filtering and mapping is done by calling `Filter` and `Map` on a base list arrow and passing in the appropriate lambda functions for each. The value converter, meanwhile, has to convert its input to the correct type and then manually cycle through each element, adding a string version of it to the result if its volume exceeds one. However, it could be argued that with more complex bindings like this the value converter is easier to debug than arrows: breakpoints can be conveniently set up inside the filtering loop and the code can be stepped through, whereas if the arrow gives unexpected results a lot of the work will be done by 'behind the scenes' methods which the user cannot step through (this is more of an issue for list arrows as they are built up with the map, filter and sort operators).

The code for creating the binding is much simpler this time, and the arrow-based code is very short: only a single line calling the `GetBindPoint` methods of the source and destination and passing in the arrow. The WPF binding is again

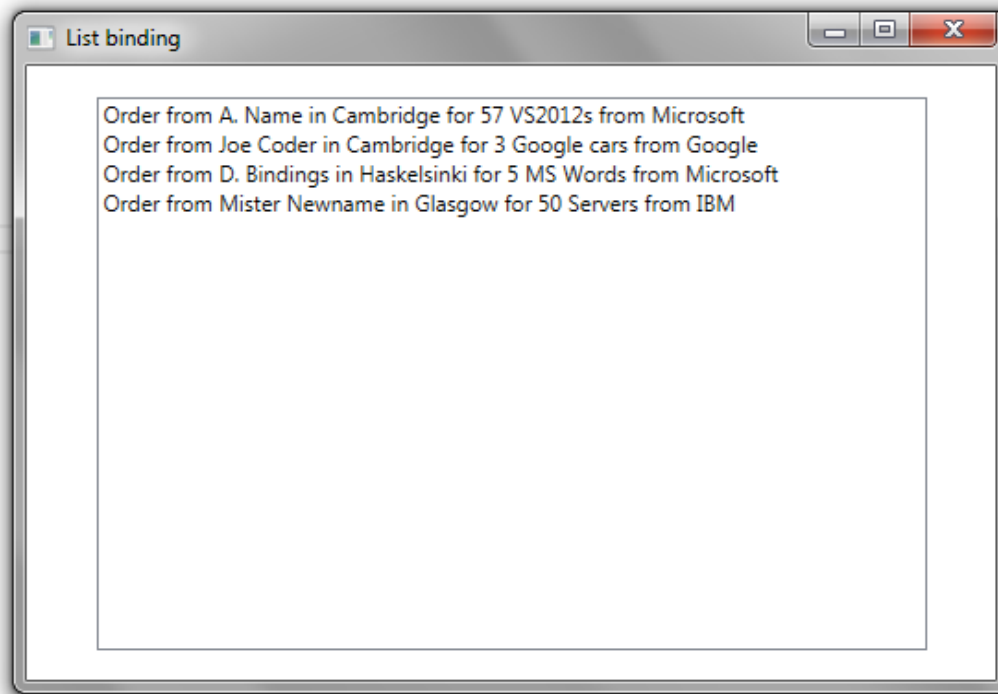


Figure 4.2: The list binding application

more verbose, requiring the programmer to create a new `Binding`; individually set the `Source`, `Path` and `Converter` properties³ and attach it to the list view. The main advantage of arrow data binding here is the `GetBindPoint` method provided by `Bindable` objects, which simplifies specifying the bound variable to one call and means the variable name is 'attached' to the object it belongs to.

Some other demo

Todo

4.3 Performance testing

4.3.1 Arrow performance

Naturally, the C# arrow implementation comes with some overhead. Being built on `Func` objects, arrows are slower than normal functions and slightly slower

³`Source` is the object to bind to, `Path` is the name of the property being bound and `Converter` is the `IValueConverter` used to convert the source value into the destination value

than plain `Func` objects. However, the difference isn't very large in most cases – depending on how arrows are constructed, their performance is often very similar to using a plain `Func`. A number of performance tests were conducted to explore how using arrows affects performance, and the results are given in the next few sub-sections.

Measuring technique

A modular test framework was set up in which performance tests could be added by simply extending a base class and implementing methods for creating the arrow, the `Func` and the normal method. Timing was done by taking measurements of the total CPU time utilised by the current process, as follows:

```
TimeSpan start = Process.GetCurrentProcess().TotalProcessorTime;
RunTest();
TimeSpan end = Process.GetCurrentProcess().TotalProcessorTime;

double timeTaken = (end - start).TotalMilliseconds;
```

Each method was run for several million iterations and the resulting execution times were recorded and graphed to show relative performance.

Simple function results

To test how arrows perform implementing simple functions (on primitive types), a selection of simple arrows was implemented along with `Func` objects and normal functions implementing the same functionality. The functions chosen are given below, along with the number of iterations used for each⁴:

Arctan This function involved computing the `arctan` function on its input and converting the result to degrees. This was run for 5,000,000 iterations.

Quadratic Computed the simple quadratic function $x^2 + 2x - 5$, and was run for 10,000,000 iterations.

Increment Simply incremented its input, and was run for 10,000,000 iterations.

Pythagoras Computed the function $\sqrt{2x^2}$, as though finding the hypotenuse of a right-angled triangle with two sides of length x . For this one, the arrow was created by combining several sub-arrows, and so the overhead was considerably greater. 3,000,000 iterations were used here.

⁴Different numbers of iterations were used to normalise the resulting graph

The results of these tests are given in Figure 4.3. There are a few key observations which can be made here. Firstly, for simple functions like incrementing, arrows fare reasonably well with only slightly worsened performance. This is because these functions were implemented directly as one uncombined arrow, and so the overhead above a simple `Func` is minimal. Indeed, in the case of `arctan`, the performance difference is almost non-existent, as here function is sufficiently complex that the time taken to compute it becomes the main factor and makes the arrow overhead negligible. However, with the `Pythagoras` function, the arrow performs significantly worse than the other two. This is most likely due to the way the arrow is composed of many sub-arrows, as shown in the following code:

```
Func<int, int, int> add = ((int x, int y) => x + y);

arrow = Op.Arr((int x) => Tuple.Create(x * x, x * x))
        .Unsplit(add)
        .Combine(Op.Arr((int x) => (int)Math.Sqrt(x)));
```

Arrow chaining is the main performance issue arrows have, as increasingly complex combinations of arrows lead to a linear performance overhead. This issue is explored in more detail at the end of this sub-section. However, this is to be expected as a result of the implementation technique, and could perhaps be mitigated in future by optimising combined arrows as they are built up (discussed further in Section 5.1.1).

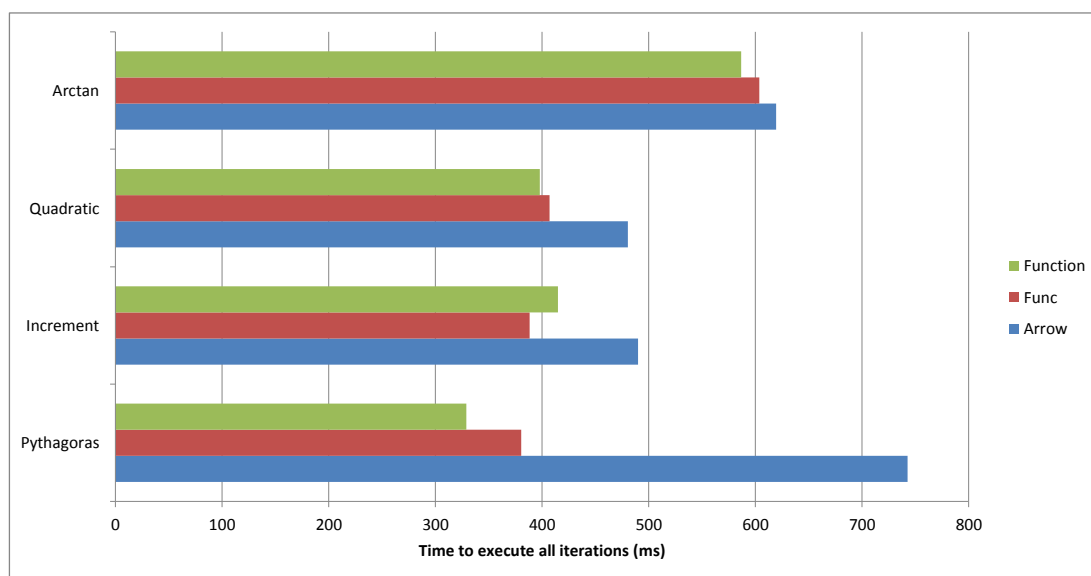


Figure 4.3: Performance of arrows, Funcs and normal functions in implementing simple functionality

List function results

List arrows were also evaluated, this time in comparison with Linq queries and simple functions based on loops. For this, a simple mock database was created of **Person** objects, each having a name, age and reference to an **Employer** object. The three functions tested follow:

Filter The task here was to filter the input to the list of people over ten years old. This was run for 1,000,000 iterations.

Foldl This tested the performance of the `foldl` method implemented for list arrows. The method had to combine all the **Person** objects into a single **Person** whose age was the sum of all the others and whose name was a concatenation of every name interspersed with ‘and’ (“Alice and Bob and Eve and...”). This was run for 300,000 iterations.

Order by then map This function involved ordering the list by age and then mapping each person such that their name was followed by “ mapped”. This was also run for 300,000 iterations.

The results can be seen in Figure 4.4. Interestingly, arrows performed fairly well on the first two tests. The loop-based function was fastest for filtering, with arrows following close behind – presumably as a result of their filter method also being a simple loop. The Linq expression was slowest as it seems the Linq overhead ended up dominating. For `foldl`, the results are too similar to draw much from, but the arrow implementation appears to have been slightly more efficient. Again, the difference is likely to have been influenced by the overhead associated with Linq queries. Finally, filter then map shows the effect of arrow concatenation slowing things down – in the case of list arrows, this overhead is greater than normal because the **IEnumerable** is converted to a list before being passed on to the next arrow⁵.

Overhead due to arrow chaining

As was discovered earlier on, complex arrow chaining imposes a performance overhead. To establish how severe the effects of this are, a test was set up in which the execution times of chains of identity arrows were compared. In an ideal world, fifty identity arrows chained end-to-end would run in the same time as a single one, but in reality the implementation leads to longer chains taking

⁵This was necessary to make some of the list arrows compatible, as various type errors were being encountered previously

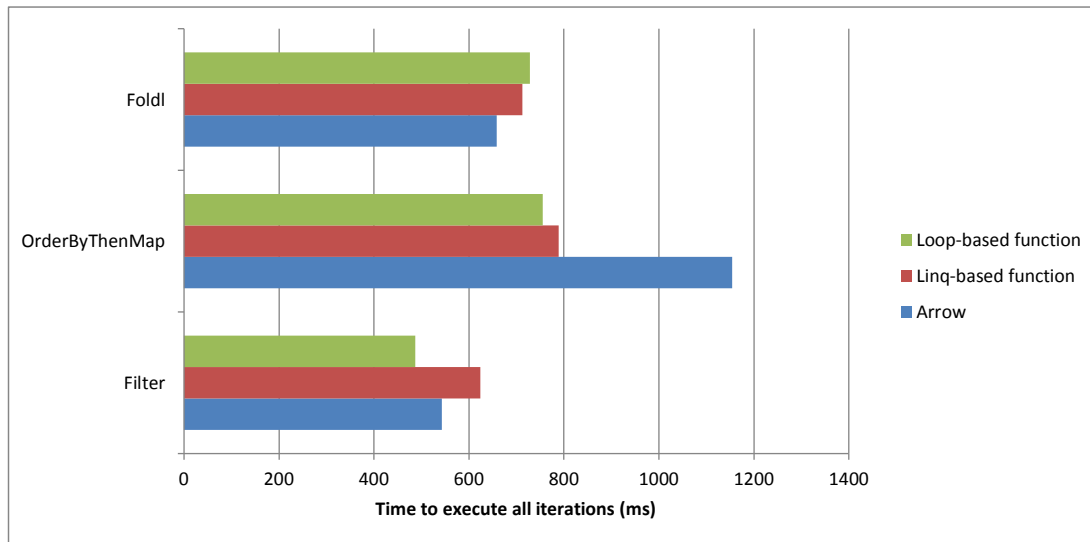


Figure 4.4: Performance of arrows, Linq queries and normal (loop-based) functions in implementing simple list functionality

considerably longer to execute. A set of identity chains with lengths ranging from one to twenty was created, and each one was timed over 1,000,000 executions.

Fortunately, it was found that the execution time is linear in the length of the chain – that is, each arrow combination adds only a constant extra overhead. The results can be seen in Figure 4.5. Aside from small random variations, the timings clearly follow a linear trend. As noted earlier on, the overhead is far less pronounced when the function implemented is more complex than identity or incrementing, and so it is likely that for the majority of use cases combination overhead won't be significant enough to noticeably harm performance.

4.3.2 Binding performance

Todo...

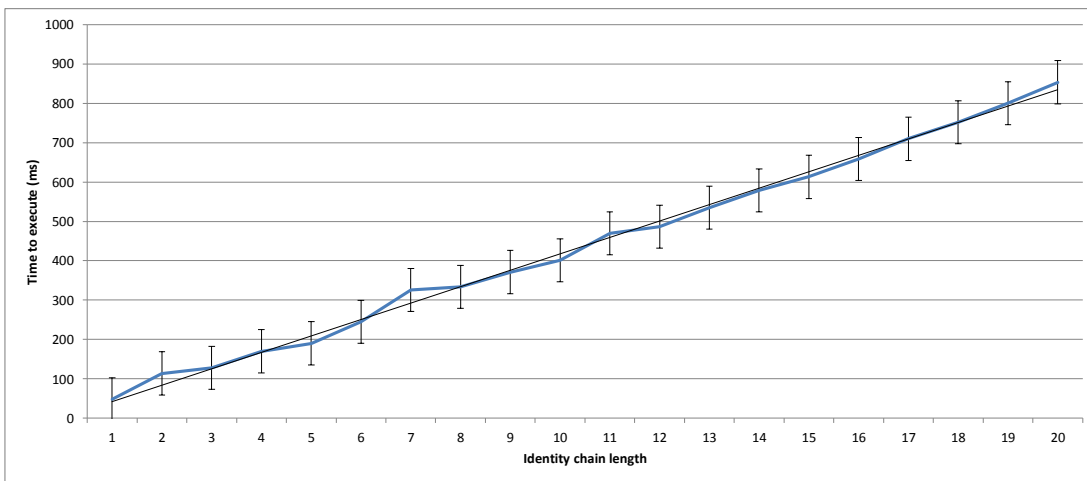


Figure 4.5: Execution times of chains of identity functions

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
- Syntax enhancements, perhaps through Roslyn

5.1.1 Performance enhancements

Blah

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.

The relation ' \approx ' is used throughout to relate effectively equivalent C# expressions.

A.1 Normal arrow laws

```
Identity  
arr id = id  
new IDArrow<T>()  $\approx$  id
```

```
Arr operator distributivity  
arr (f >>> g) = arr f >>> arr g  
Op.Arr(x => g(f(x)))  $\approx$  Op.Arr(f) .Combine( Op.Arr(g) )
```

```
First operator distributivity  
first (f >>> g) = first f >>> first g  
f .Combine(g) .First<T>()  $\approx$  f .First<T>() .Combine( g.First<T>() )
```

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

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

Composition associativity

```
f >>> (g >>> h) = (f >>> g) >>> h
f.Combine(g.Combine(h)) ≈ (f.Combine(g)).Combine(h)
```

Composition distributivity

```
f1 ↔ g2 >>> g1 ↔ f2 = (f1 >>> g1) ↔ (f2 >>> g2)
f1.Arr(g2).Combine(g1.Arr(f2)) ≈ Op.Arr(x => g1(f1(x)), y => g2(f2(x)))
```

Composition with identity

```
idA >>> f = f = f >>> idA (where idA = id ↔ id)
new IDArrow<T>().Combine(f) ≈ f ≈ f.Combine(new IDArrow<T>())
```

‘First’ operator distributivity

```
first (f >>> g) = first f >>> first g
(f.Combine(g)).First<T>() ≈ first_f.Combine(first_g)
```

‘First’ operator commutativity

```
first (f ↔ g) = (f * id) ↔ (g * id)
(f.Arr(g)).First<T>() ≈ f.Mult(id).Arr(g.Mult(id))
```

Double inversion

```
inv (inv f) = f
f.Invert().Invert() ≈ f
```

Distributivity of inversion

```
inf (f >>> g) = inv g >>> inv f
f.Combine(g).Invert() ≈ g.Invert().Combine(f.Invert())
```

Inversion correctness

```
inv (f ↔ g) = g ↔ f
f.Arr(g).Invert() ≈ g.Arr(f)
```

‘First’ operator under inversion

```
inv (first f) = first (inv f)
f.First<T>().Invert() ≈ f.Invert().First<T>()
```

Appendix B

WPF integration demo – code listings

B.1 Creating the bindable source

```
public class Time : Bindable
{
    [Bindable]
    public int CurrentTime { get; set; }

    private Timer timer;

    public Time()
    {
        CurrentTime = 0;
        // Make the CurrentTime variable increment every 25 milliseconds
        timer = new Timer(state => CurrentTime++, null, 25, 25);
    }
}
```

B.2 The binding arrow

```
progressArrow = Op.Arr(
    (int x) => 50.0 + (50.0*Math.Sin(x / 30.0))
);
```

B.3 Doing the binding

```
Binding binding = new Binding();
binding.Source = time;
binding.Path = new PropertyPath("CurrentTime");
// Make a value converter using the arrow
binding.Converter = new ArrowValueConverter(progressArrow);
Progress.SetBinding(ProgressBar.ValueProperty, binding);
```


Appendix C

Code samples for case studies

C.1 Username two-way binding

C.1.1 Creating the bindable data source

```
public class NameData :
    INotifyPropertyChanged
{
    public event
        PropertyChangedEventHandler
        PropertyChanged;

    private String fullName;
    public String Name
    {
        get
        {
            return fullName;
        }

        set
        {
            fullName = value;
            RaisePropertyChanged("Name");
        }
    }
}
```

```
public class NameData : Bindable
{
    [Bindable]
    public string Name { get; set; }

    public NameData(string name)
    {
        Name = name;
    }
}
```

```

public NameData(string name)
{
    Name = name;
}

public void RaisePropertyChanged(
    string name)
{
    if (PropertyChanged != null)
    {
        PropertyChanged(this, new
            PropertyChangedEventArgs(
                name));
    }
}
}

```

C.1.2 Creating the functionality for splitting the name

```

public class ForenameValueConverter :      nameArrow = Op.Arr(
    IValueConverter                        (string x) => Tuple.Create(
{
    public object Convert(object value,      x.Split()[0],
        Type targetType, object           x.Split()[1]),
        parameter, CultureInfo culture)    (Tuple<string, string> splitName) =>
    {
        return ((string)value).Split(' ')  String.Format("{0}_{1}", splitName
        [0];                               .Item1, splitName.Item2));
    }
}

public object ConvertBack(object
    value, Type targetType, object
    parameter, CultureInfo culture)
{
    throw new NotImplementedException
        ();
}
}

```



```

public class SurnameValueConverter :
    IValueConverter
{
    public object Convert(object value,
        Type targetType, object
        parameter, CultureInfo culture)
    {
        return ((string)value).Split(' ')[1];
    }

    public object ConvertBack(object
        value, Type targetType, object
        parameter, CultureInfo culture)
    {
        throw new NotImplementedException
            ();
    }
}

```

C.1.3 Creating the binding

```

public void InitialiseBindings()
{
    InitialiseNameBinding(ForenameBox,
        new ForenameValueConverter());
    InitialiseNameBinding(SurnameBox,
        new SurnameValueConverter());
}

public void InitialiseNameBinding(
    TextBox textBox, IValueConverter
    converter)
{
    Binding bind = new Binding();
    bind.Source = name;
    bind.Path = new PropertyPath("Name")

    BindingsManager.CreateBinding(
        BindingsManager.Sources(name.
            GetBindPoint("Name")),
        nameArrow,
        BindingsManager.Destinations(
            splitName.GetBindPoint("Forename")
            ,
            splitName.GetBindPoint("Surname"))
        );
}

```

```
        ;  
        bind.Converter = converter;  
        textBox.SetBinding(TextBox.  
            TextProperty, bind);  
    }
```

C.2 List binding from a mock database

C.2.1 Creating the bindable ‘database’

This was much the same as creating the data source for the username binding case study, with the main additional difference between the two being that the WPF version used an `ObservableCollection` rather than a plain `List`.

C.2.2 Filtering and mapping the list

```
public class ListFilterConverter :
    IValueConverter
{
    public object Convert(object value,
        Type targetType, object
        parameter, CultureInfo culture)
    {
        ObservableCollection<Order> list =
            (ObservableCollection<Order>)
            value;

        List<string> resultsList = new
            List<string>();

        foreach (Order order in list)
        {
            if (order.Volume > 1)
            {
                resultsList.Add(String.Format(
                    "Order_{0}in_{1}for
                    _{2}_{3}'_from_{4}",
                    order.Customer.Name, order
                    .Customer.Location, order.
                    Volume, order.Product,
                    order.Supplier.Name));
            }
        }

        return resultsList;
    }
}
```

```
arrow = ListArrow.Filter<Order>
    ((Order o) => o.Volume > 1)
    .Map((Order o) =>
        String.Format("Order_{0}in_{
            {1}_for_{2}_{3}s_from_{4}",
            o.Customer.Name, o.Customer.
            Location, o.Volume, o.Product,
            o.Supplier.Name));
```

```
}

public object ConvertBack(object
    value, Type targetType, object
    parameter, CultureInfo culture)
{
    throw new NotImplementedException
        ();
}
}
```

C.2.3 Creating the binding

```
Binding listBinding = new Binding();
listBinding.Source = ordersDB;
listBinding.Path = new PropertyPath("
    Orders");
listBinding.Converter = new
    ListFilterConverter();
BoundListBox.SetBinding(ListBox.
    ItemsSourceProperty, listBinding);

BindingsManager.CreateBinding(
    ordersDB.GetBindPoint("Orders"),
    arrow,
    arrowResult.GetBindPoint("Result"));
```

C.3 Some other demo

Todo

Bibliography

- [1] Artem Alimarine, Sjaak Smetsers, Arjen Weelden, Marko Eekelen, and Rinus Plasmeijer. There and back again: arrows for invertible programming. In *In Proceedings of the 2005 ACM SIGPLAN workshop on Haskell*, pages 86–97. ACM Press, 2006.
- [2] Conal Elliott. Composing reactive animations. *Dr. Dobbs' Journal*, 1998. Also in expanded animated form at <http://conal.net/fran/tutorial.htm>.
- [3] Paul Hudak. Functional reactive programming. In S.Doaitse Swierstra, editor, *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*, pages 1–1. Springer Berlin Heidelberg, 1999.
- [4] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming, 4th International School, volume 2638 of LNCS*, pages 159–187. Springer-Verlag, 2002.
- [5] John Hughes. Generalising monads to arrows. 1998.
- [6] John Hughes. Programming with arrows. In *Advanced Functional Programming*, pages 73–129. Springer, 2005.
- [7] Sam Lindley, Philip Wadler, and Jeremy Yallop. The arrow calculus (functional pearl). 2008.
- [8] MSDN. Data binding overview. <http://msdn.microsoft.com/en-us/library/ms752347.aspx> (accessed on 24/04/2013).
- [9] Ross Paterson. Arrows: A general interface to computation. <http://www.haskell.org/arrows> (accessed on 24/04/2013).
- [10] Ross Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

- [11] Ross Paterson. Arrows and computation. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*, pages 201–222. Palgrave, 2003.
- [12] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *In ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 242–252, 2000.