Part II Project Proposal: Functional Reactive Programming for Data Binding in C♯

David Barker
Project supervisor: Tomas Petricek

October 18, 2012

1 Introduction and Description of the Work

Data binding is widely used throughout software design as a general way of linking two properties together such that their values are synchronised. This is useful in many areas – for instance, in user interface programming, data binding can be used to allow the interface to directly reflect the underlying model with minimal code on its part. As a result, data binding is now a standard feature in some widely-used languages and frameworks.[1] However, it is often fairly limited in that only direct bindings are available (where the destination's value is exactly that of the source) and they generally only work in one direction.

A feature some frameworks provide is *functional* data binding. This is where the value of a binding source is passed through a function to the binding destination,[2] and so a more complex binding can be established. This works well for many applications, but is still not as natural to use as it could be. It also still suffers from the limitation of only working in one direction, which prevents the programmer from having two mutable properties which are functionally linked.

One way of making this idea of functional data binding more general is through functional reactive programming (FRP). This is a paradigm in which behaviour is defined *reactively* on input signals which can vary either discretely or continuously. Elliott[1] provides a good example and introduction to the concept of FRP. In the context of data binding, this would involve treating the source of the binding as the input signal and setting up some framework with which the programmer could arrange the functions for the data to be passed through.

Haskell's *arrow* construct[2] provides a useful abstraction for functions along with a set of combinators which make it easy to combine several arrows and build up more complex functions.[7] The high-level nature of arrows makes them a good fit for modelling functional reactive programming, and the abstraction of an 'arrow from type a to type b' fits the basic

[1]For instance, Javascript[6] and .NET[5] both provide this functionality

[2]Microsoft Excel's formula cells are a widely-used example of this

concept of data binding nicely. As such, the project will use an implementation of Haskell-style arrows as the primary form of data binding. Arrows also open up the possibility of creating an *invertible* arrow which could be used for two-way binding.

The project's main deliverable will be a C♯ arrow implementation with all the standard combinators along with a library for setting up and managing data bindings using these arrows. Furthermore, a means of constructing invertible arrows will be developed and implemented as well, and two-way binding will be implemented in the data binding managers.

2 Starting Point

As a starting point, I have developed a good working knowledge of C♯ and am familiar with functional languages and ideas. I have also read some introductions to both functional reactive programming and arrows, and have implemented a very basic binding demo illustrating the technique outlined in the next section.

3 Substance and Structure

3.1 Data Binding

The basic data binding approach will most likely consist of four main object types: BindingSource<A>,[3] Binding<A, B>, Arrow<A, B> and BindingDestination<B>. Binding objects will hold references to a source, an arrow and a destination, and subscribe to a 'value changed' event[4] which will be published by the binding source (the source will override its value property's setter method to fire off this event). It will then call the arrow with the value of the source and set the destination's value to the result of this.

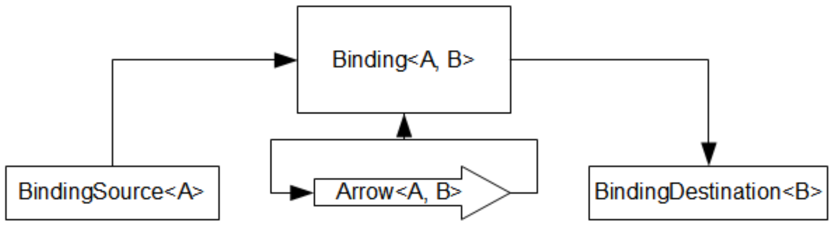

Figure 1: The binding setup

However, there are a number of possible variations on this and so the final approach may differ slightly if a better one is found.

[3]Where <A> denotes a parametric type A

[4]This may use the standard .NET INotifyPropertyChanged interface for simplicity[8]

3.2 Arrows

Arrows will be defined as objects containing C♯ lambda functions, with an 'execute' function which takes an input of the correct type and returns the result of the lambda function applied to it. This will be called by the Binding object which owns the arrow whenever an update to the bound variable is received. Arrow objects will take the input and output types as type parameters, and so the combinators will need to use some form of reflection to dynamically combine arrows of different types which will only be known at runtime. Establishing how exactly to implement this will likely form a significant portion of the initial implementation work. Further to this, allowing arrows to take pairs as inputs (which could potentially be pairs of pairs) will require some form of tree-structured generic type to be developed as an arrow input type.

3.3 Invertible Arrows

Currently, two possible approaches to implementing invertible arrows are known. One possibility is to require the user to provide both a 'set' and a 'get' arrow for the binding source (with the assumption being that they will be inverses of each other). The binding would then simply use the 'get' when passing a change forward from source to destination, and 'set' when passing back.

Another possibility is to limit the range of possible functions to those which are known to be invertible. That is, a set of 'basic' arrows would be provided (ideally Turing complete) and the programmer would then build up their desired invertible functions by combining these using the arrow combinators.

The viability of each approach will be explored over the course of the project in order to determine which is the best option, and the reasoning will be discussed in the evaluation.

3.4 Possible Extensions

There are a number of possible extensions to the project which will be worked on once the main framework is complete:

- Unfortunately, C♯ does not allow the programmer to create custom operators. This means that a Haskell-like syntax for arrows is not possible in the C♯ implementation. However, the Roslyn framework allows direct access to C♯'s parser, and allows the programmer to manipulate the result of parsing. This could potentially be used to extend the C♯ syntax to allow for more natural arrow construction and combination.

- In a similar vein to the above, the strictness of C♯'s type system means that the system will make heavy use of generics, and so programmer code will likely have to be littered with type parameters. The goal of this extension would be to minimise these wherever possible, most likely by making liberal use of reflection to derive types.

- A side-effect of the above is that using reflection to derive types at runtime will most likely have a significant impact on type safety, and potentially lead to runtime type errors and bad conversions where the programmer has made typing mistakes. To make the framework more useful, a good extension would involve adding type checks to the code to ensure that incorrect types aren't being used and throw type exceptions where mistakes are found.
- Arrows as originally defined by Hughes[3] must conform to a series of 'arrow laws'. These are outlined and discussed in detail by Lindley et al[4]. Invertible arrows may well introduce new laws or require modification of the existing ones, and so an extension to the project would be to explore these and derive a new set of laws for invertible arrows.
- The PostSharp[9] framework allows the programmer to define custom attributes and intercept variables being set and read. This could be used to simplify the creation of binding sources, as the programmer implementing the source could simply place a 'Bindable' attribute above the variable they want to bind to and I could set up an Aspect which would handle the generation of events whenever they assigned to the variable.
- Another possible use of the Roslyn framework for simplifying syntax would be to allow pair arguments to arrows to be passed using simple (v1, v2) syntax rather than by creating a new pair object first.

4 Success Criteria

The goal of the project is to implement a C♯ library for one- and two-way data binding, using an implementation of Haskell arrows with special 'invertible arrows' for two-way binding. To evaluate the end result, a number of techniques will be used:

- As mentioned earlier, C♯ already has a working data binding system. Thus, a good way of evaluating the implementation would be writing some application which uses data binding non-trivially[5] and comparing the code required by normal data binding and by FRP-based binding for readability, simplicity and elegance.
- It could also make sense to compare the C♯ implementation with Haskell arrows (on similar metrics), as a large portion of the work will involve trying to minimise the messy type parameter syntax which will be required to implement general arrow combinators in C♯.

[5] One possible application would be a simple demo in which the user can drag around four line end-points and the point of intersection is automatically drawn. This would be implemented with arrows by having point values in the model which are bound to those in the view by an invertible arrow which translates screen co-ordinates to real co-ordinates, and then a binding from the points in the model to the point of intersection on the screen by an arrow which takes two lines defined as point tuples and outputs the point of intersection.

- As Hughes' original definition of arrows requires them to follow a set of laws, a successful C♯ implementation would naturally require that these laws are followed, and so part of the evaluation will consist of verifying this. The main approach for this will be automated testing on random data, but an extension to this would be to attempt a form of semi-formal verification using a simplified mathematical model of the program syntax to prove equalities.
- Tying in with the third point in 'Possible Extensions', if the laws for invertible arrows differ from the standard arrow laws it would also be necessary to test that these are also upheld.
- It may also make sense to compare the performance of the resulting data binding framework with equivalent bindings expressed in standard .NET binding, though this will effectively be an extension to the evaluation as it will involve working out how exactly to go about this.

5 Work plan

The core implementation work consists of nine work packages. There are then four extension sections, and three for evaluation, with the remaining time allocated to writing the dissertation and fixing any bugs and problems which arise.
Core work packages:

5.1 Work package 1 (22nd October, one week)

- Setting up Git, backup, C♯ projects, log etc.
- Establishing overall system structure

5.2 Work package 2 (29th October, two weeks)

Simple demos to test key functionality:

- Get basic (combinator-free) arrows working for arbitrary types
- Make a function to combine two arbitrary lambda expressions. Will involve investigating how best to do this, eg. converting into expression trees first, or perhaps storing as expression trees the whole time?
- Investigate minimising type parameters

5.3 Work package 3 (12th November, two weeks)

More simple demos and tests:

- Get a simple two-way binding working and establish the general approach and semantics
- Investigate invertible arrows (combinators, implementation techniques etc.) – may well involve several small demos

Milestone: Various working demos of framework functionality complete

5.4 Work package 4 (26th November, two weeks)

- Implement the basic binding framework
- Implement basic arrows (still combinator-free and probably using lots of type parameters)

5.5 Work package 5 (3rd December, one week)

- Arrow combinators – get all the standard ones working, probably with type parameters for now

Milestone: Primitive version of the framework is functional

5.6 Work package 6 (10th December, two weeks)

- Eliminate as many type parameters as possible from arrow combinators using reflection, ExpressionTree constructs etc. – will require some research into possibilities

5.7 Work package 7 (24th December, two weeks)

- Implement two-way binding

5.8 Work package 8 (7th January, two weeks)

- Investigate and implement invertible arrows (including combinators)

5.9 Work package 9 (21st January, one week)

- Pair arguments – implement a generic tree-structured arrow argument type to allow for pairs (and pairs of pairs etc.) to be passed as input arguments to arrows
- Write progress report

Milestone: Core project is complete, progress report is underway
Extension work packages:

5.10 Work package 10 (28th January, one week)

- Complete and submit progress report
- Continue work on simplifying syntax – removing type parameters, making operations more natural etc.

5.11 Work package 11 (4th February, one week)

- Explore possible changes to the arrow laws for invertible arrows
- Implement more careful type-checking for generic and reflective code to restore type safety

5.12 Work package 12 (11th February, one week)

- Investigate Rosalyn framework; attempt various small demos

5.13 Work package 13 (18th February, one week)

- Work on simplifying syntax for arrow creation and combination, using Rosalyn or otherwise
- Work on simplifying syntax for pair creation

Milestone: End of time allocated for extension work – as many as time allows will have been implemented
Evaluation work packages:

5.14 Work package 14 (25th February, two weeks)

- Evaluate syntax – compare with Haskell and standard .NET binding via sample apps
- Start dissertation

5.15 Work package 15 (11th March, two weeks)

Evaluate arrow laws

- Implement and run automated testing
- Attempt to construct a simplified mathematical model and use to prove some of the equalities
- Continue work on dissertation

5.16 Work package 16 (25th March, one week)

(This one will be an extension.)

- Performance comparison between FRP-based data binding and standard .NET equivalent for some non-trivial purpose
- Continue work on dissertation

Milestone: Evaluation work complete, dissertation well underway
Time left from this point will be allocated to writing the dissertation and fixing any bugs.

6 Resource Declaration

I will be using my own laptop for the project, or in the event of failure the MCS machines available in the Computer Laboratory. I will ensure all data is backed up online – code will be hosted on Github, and other data will be hosted elsewhere. In the event of one of these sites going down I will also ensure I have a copy of the entire project on a USB drive.

References

[1] Conal Elliott, *Composing Reactive Animations*, 1998

[2] Ross Paterson, *Arrows: A General Interface to Computation* retrieved 08/10/2012

[3] John Hughes, *Generalising Monads to Arrows*, 1998

[4] Sam Lindley, Philip Wadler, Jeremy Yallop, *The Arrow Calculus (Functional Pearl)*, 2008

[5] MSDN, *Data Binding Overview*, retrieved 08/10/2012

[6] Greg Weber, *Javascript Frameworks and Data Binding*, 2012

[7] Ross Paterson, *Arrows and Computation*, The Fun of Programming, 2003

[8] MSDN, *INotifyPropertyChanged Interface*, retrieved 08/10/2012

[9] Code Project, *Aspect Oriented Programming using C♯ and PostSharp*, retrieved 17/10/2012