David Barker

Functional Reactive Programming for Data Binding in C♯

Computer Science Tripos Part II

Jesus College

February 26, 2013

Proforma

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

Original Aims of the Project

Aims of the project will go here.

Work Completed

Work completed will go here.

Special Difficulties

None

Declaration

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

Signed [signature]

Date [date]

Contents

List of Figures

Chapter 1

Introduction

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

1.1 Data binding

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

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

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

1.2 Data binding in .NET

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

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

1.3 Project inspirations

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

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

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

Chapter 2

Preparation

This chapter is empty still!

Chapter 3

Implementation

3.1 Overview

Give a general overview

3.2 Arrows

3.2.1 Overview

Overview of how arrows worked

- Type inference chosen instead of reflection
- Techniques for allowing type inference - static constructor methods, extension methods etc.
- Arrow intercompatibility?
- Multiple inputs/outputs and the decision to use C♯ Tuple class

3.2.2 Simple arrows

Summarise functionality, combinators and syntactic efforts made here, mention things like the parallelism of the And operator as well.

Challenges encountered

A few - mention type inference challenges, overcoming type system whilst maintaining type safety and difficulties with the And operator (also alternatives considered for this)

3.2.3 Invertible arrows

Discuss the alternative ways of implementing this - a set of preset arrows designed to form a Turing-complete basis, forcing the user to specify an arrow for each direction etc. Final decision to simply construct using a function for each direction then allow arbitrary combination, as inspired by the paper on invertible arrows.

3.2.4 List arrows

- Wrapping Linq
- SQL-style syntax - helped by various extension methods
- Functional-style operations - foldl and foldr

3.2.5 Choice arrows

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

3.2.6 Further utility arrows

Identity arrows Exist for normal arrows and invertible ones; mention the reason for the class - has to be type parameterised

Tuple reassociation arrows Useful for stuff like arrow law tests, explain here...

Others Any more to mention?

3.2.7 Feedback in arrows

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

3.3 Data binding

3.3.1 Overall architecture

Use diagrams and stuff to explain it.

3.3.2 Creating bindable sources and destinations

Explain it with a bit of code here and there.

PostSharp

Explain AOP and how this was useful in fixing up the syntax.

3.3.3 Creating bindings

Syntax

Show what it looks like.

Cycle and conflict detection

Techniques used and all that.

Two-way binding

Problems encountered - need for variable locking system to avoid infinite loop

Many-to-many bindings

Mention:

- Difficulties with how to set it up; ultimate decision to pass in lists of sources and destinations
- Tuple marshalling/unmarshalling:
  - Arrows use binary tree structured tuples due to the way they are constructed, so had to match this structure rather than just going to a flat tuple
- Test

Problems encountered

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

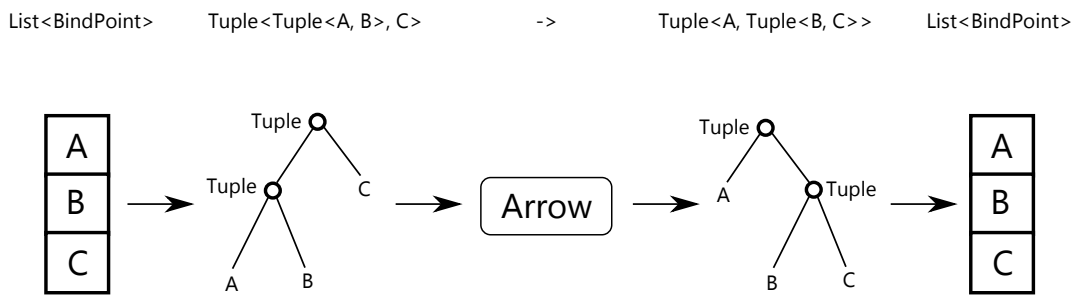

Figure 3.1: Marshalling a list of BindPoint sources to pass to an arrow and unmarshalling the result

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

3.3.4 Integration work with WPF

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

Chapter 4

Evaluation

4.1 Correctness of arrow implementations

Mention here that the main device for determining their correctness has been the arrow laws, and possibly list them with explanations? Also mention that 'special' arrows like ListArrows are correct because they are based on simple arrows and invertible arrows. Also, maybe say that ArrowChoice was ignored here as it's slightly extraneous and its correctness probably follows from the correctness of simple arrows anyway.

4.1.1 Automated testing

Simple arrows

Give tests and results for arrows.

Invertible arrows

Same for invertible arrows.

4.1.2 Correctness proof by decomposing into lambda calculus

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

4.2 Syntax evaluation

4.2.1 Arrow syntax

Arrow syntaxy stuff.

Comparison with Haskell

Might not be great, but why not?

4.2.2 Binding syntax

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

Username two-way binding

Todo

List binding from a mock database

Todo

Some other demo

Todo

4.3 Performance testing

4.3.1 Arrow performance

Tests indicating how arrows perform vs. functions and funcs.

Measuring technique

Explain how the results were obtained

Simple function results

The simple function results can be seen in the graph.

List function results

Results for list-based functions

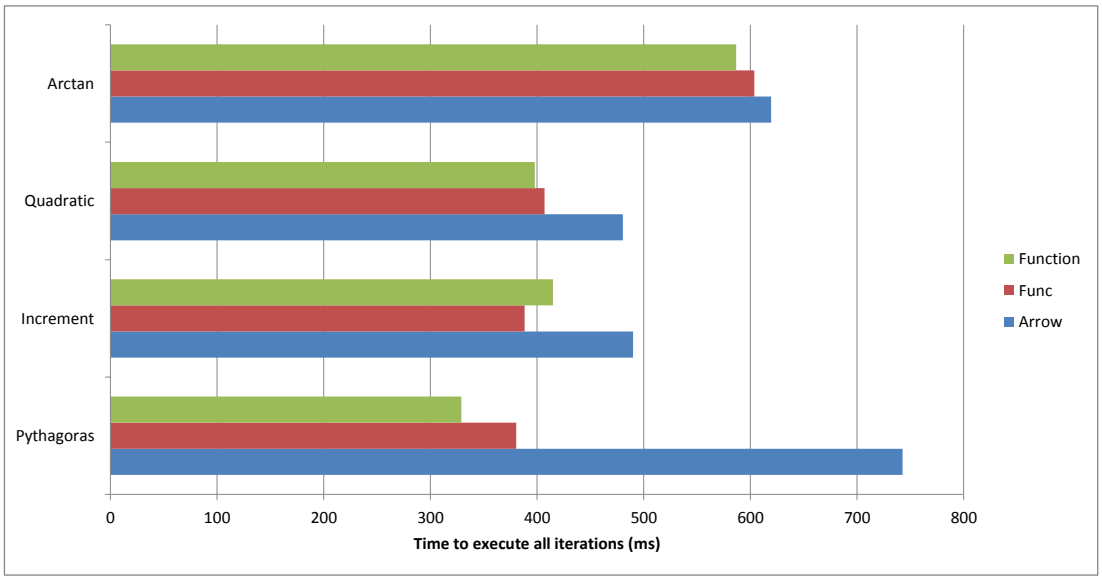

Figure 4.1: Performance of arrows, Funcs and normal functions in implementing simple functionality

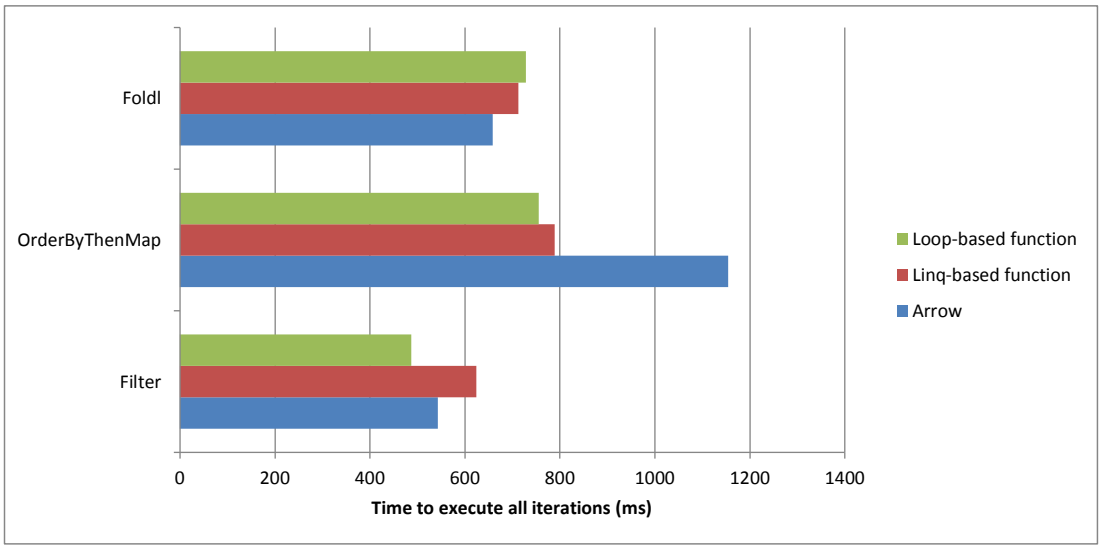

Figure 4.2: Performance of arrows, Linq queries and normal (loop-based) functions in implementing simple list functionality

Overhead due to arrow chaining

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

later?)

4.3.2 Binding performance

Todo...

Chapter 5

Conclusion

Conclusion goes here!