

# An Aspect-Oriented Framework in F#

---

**Nitesh Chacowry**

**MSc Computer Science Final Year Project Report  
Department of Computer Science and Information Systems  
Birkbeck College  
University of London**

**17<sup>th</sup> September 2011**

## ***Declaration***

*I certify that I have accessed and understood the Blackboard Course BBK ITST009N 2006: Avoiding Plagiarism.*

*By submitting this assignment I confirm that the work is my own, with the work of others fully acknowledged. All quotations from the published or unpublished works of other persons are duly acknowledged. Where there is no such acknowledgement the work is my own.*

*The work presented here has not been presented, in whole or in part, for credit in any module in this or any other school.*

*Permission is granted to submit the work to an on-line database. It is understood that this work may be compared with other works in the database in order to detect plagiarism and that work submitted to the database will remain in the database.*

## *Abstract*

This dissertation presents the research, design and development of an aspect-oriented framework for F#, a functional programming language. Our framework allows one to insert *advices* before, after or around the call to a particular function. We provide two distinct approaches to *weaving* the advice to the source code: using a monad-based weaver, and using a weaver built on metaprogramming technologies.

Our weaver built using metaprogramming technologies translates a source program into a data structure which is amended as required to inject the advices. The weaver then returns another data structure representing the advised (or transformed) program. The weaver using metaprogramming technologies is judged to be more granular and less intrusive and hence is chosen as the weaver for our framework.

We use our framework to advise a program which estimates  $\pi$ . In addition, we provide some timing comparison between a program where we manually insert the advices and one advised by the framework. Our results show a performance penalty when using the framework as we are required to make expensive function calls to compile and execute the transformed program. This dissertation provides some concluding remarks and suggestions for future work.

# Contents

<i>Declaration</i> .....	2
<i>Abstract</i> .....	3
1. Introduction .....	6
1.1 Background.....	6
1.2 Motivation.....	6
1.3 Scope and objectives .....	7
1.4 Achievements .....	7
1.5 Overview of dissertation .....	7
2. Background on Aspect-Oriented Programming and Functional Languages .....	8
2.1 Aspect-oriented programming (AOP) .....	8
2.1.1 Join points.....	9
2.1.2 Pointcuts.....	9
2.1.3 Advices .....	9
2.1.4 Aspects .....	10
2.1.5 Weaving.....	10
2.2 Functional languages .....	11
2.2.1 Function signatures.....	12
2.2.2 F# Interactive (fsi) .....	14
2.2.3 Active patterns.....	14
2.3 Summary .....	15
3. Requirements and development approach .....	17
3.1 Requirements .....	17
3.2 Development tools and methodology .....	18
4. Framework design and implementation.....	19
4.1 Weaver design.....	19
4.2 Monad-based weavers.....	19
4.2.1 Theoretical background on monads .....	19
4.2.2 F# support for monads .....	21
4.2.3 Development of a monad-based weaver using computation expressions .....	23
4.2.4 Discussion on monads.....	27
4.2.5 Summary of a monad-based weaver .....	28

4.3 Code quotation-based weaver .....	29
4.3.1 Metaprogramming technologies in the .Net Framework .....	31
4.3.2 F# code quotations.....	33
4.3.3 F# abstract syntax tree nodes.....	37
4.3.4 Development of code quotation-based weaver .....	43
4.3.5 Before advice constraints .....	48
4.3.6 After advice constraints .....	49
4.3.7 Around advice constraints.....	51
4.3.8 Summary on the code quotation-based weaver .....	51
4.4 Pointcuts and aspects for the quotation-based weaver .....	51
4.4.1 Pointcut design .....	51
4.4.2 Aspect design.....	52
4.5 The AspectF Framework – a Visual Studio Solution.....	54
4.6 The F# compiler as a weaver .....	55
4.6.1 Building the compiler .....	55
4.6.2 Attaching a debugger to step through the compile process .....	55
4.6.3 Suggested strategy for building a compiler extension .....	56
4.7 Summary .....	56
5. Usage and instrumentation – advising a recursive function to estimate $\pi$ .....	58
5.1 Implementing Machin’s formula in F# .....	58
5.2 Estimating $\pi$ – Comparing speed of execution.....	60
5.2.1 Instrumentation of the non-advised code.....	61
5.2.2 Instrumentation of manually advised code.....	62
5.2.3 Instrumentation of weaved code.....	64
5.3 Summary .....	69
6. Conclusion.....	70
7. Future work.....	72
8. Bibliography .....	74
Appendix A – Unadvised and advised expression trees for the <i>arctan</i> function .....	81
Appendix B – Raw timing results for the weaved function.....	84
Appendix C – CD with source code.....	87

# 1. Introduction

## 1.1 Background

The dissertation presents the research, design and development of an *aspect-oriented* [1,2] framework for Microsoft's implementation of a functional language: *F#* [3]. Aspect-oriented programming (AOP) is a programming paradigm where functionalities which apply across different modules are cleanly encapsulated into separate components. For example, a program may require security checks to ensure that a particular user has access to certain functionalities only. In this case, AOP provides a solution to cleanly separate out the security checks from the flow of the program.

AOP is usually implemented via a dedicated framework. Such frameworks exist for object-oriented languages. Examples include AspectJ [4] for Java and Policy Injection Application Block [5,6] or Spring.Net [7] for .Net languages.

We noted in the project proposal [8], that within function languages, AOP constructs are implemented via a dedicated programming language. Example includes AspectML [9], AspectFun [10,11], Aspectual Caml [12] or MinAML [13]. The project proposal [8] of this dissertation provides further analysis of AOP frameworks targeting functional languages. However, no such frameworks or AOP programming languages currently exist for *F#* - Microsoft's implementation of a functional programming language [3,14].

## 1.2 Motivation

This project is motivated by the following factors:

- a. There are no dedicated AOP frameworks available for *F#*. *F#* is a functional language based on the .Net Framework, and is a core language shipped with Visual Studio [15] – Microsoft's integrated development environment. This is a strong indication of Microsoft's objective of getting its flavour of a functional language operating within an industrial setting. In such a setting, there are many use cases where AOP is required, for example: auditing or instrumentation of code performance [16].
- b. This project provides an opportunity to study functional languages. Functional languages have a long history [17], but there has been a relatively recent resurgence in their popularity. This can be attested by the release of functional languages such as *F#* or Clojure [18], and the integration of functional language constructs (such as lambda functions) in object-oriented languages such as *C#* and Groovy [19]. Polyglot languages [20] such as Scala [21] support both object-oriented and functional programming constructs.

- c. This project provides an opportunity to study AOP. In the project proposal [8], we provided a review of current AOP frameworks for functional languages and noted some of the issues encountered in existing implementations, such as the need to break *referential transparency* [11], or implement *runtime type analysis* [22]. This project therefore shows how F# functionalities such as the ability to define objects [23], or the ability to inspect the type of an object at runtime [24] can also address these implementation issues.

### 1.3 Scope and objectives

The objectives of this project are:

1. Research AOP and develop an understanding of this programming paradigm (Section 2.1 and Chapter 3).
2. Develop an understanding of F# (Section 2.2).
3. Design the AOP constructs for F# (Chapter 4) and develop these constructs.
4. Design a framework around the AOP constructs (Section 4.5) for reusability and distribution.
5. Perform some time measurements to compare the performance of our framework against *regular* code (Chapter 5).

### 1.4 Achievements

In this project, and supported by this dissertation, we show the design and development of the AOP framework in F# (Chapter 4). We also show a usage of this framework in Chapter 5. Currently, the framework is experimental and a greater amount of use cases would need to be implemented before general distribution.

### 1.5 Overview of dissertation

Objectives 1 and 2 above are covered in Chapter 2 while objectives 3, 4 and 5, the main outcome of the dissertation are presented in Chapters 3, 4 and 5.

In Chapter 2, we present a brief overview of aspect-oriented programming (AOP). This is followed by an overview of functional programming, with an emphasis on F# and its syntax. In Chapter 3, we detail the requirements for AOP framework we shall implement. Chapter 4 details the design choice and presents a technical presentation of our framework. In Chapter 5, we present metrics that contrasts the performance of a *regular* program with the one in which we have *woven* some additional functionalities. Finally, Chapters 6 and 7 presents some concluding remarks and suggests future work.

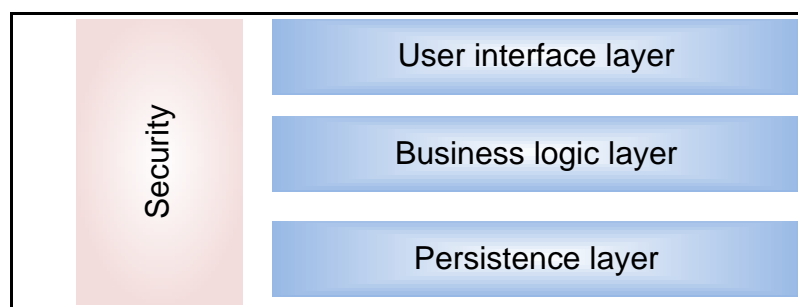
## 2. Background on Aspect-Oriented Programming and Functional Languages

This chapter provides a brief review of aspect-oriented programming (AOP) and functional languages - with an emphasis on F#. The project proposal [8] covers much of the theoretical background of AOP and functional languages, but we reiterate the important points within this chapter. The review helps in defining the issues that need to be addressed.

### 2.1 Aspect-oriented programming (AOP)

In the project proposal [8] for this dissertation, we reiterated the concept of *separation of concern* as an important concept in program design [25]. A common design pattern to achieve separation of concerns in large programs is to encapsulate the concerns into their own distinct layers, where each layer performs a specific functionality [5,26]. For example, an application might be split in three layers: a first layer which handles persistence to a database, a second layer to handle business objects and a third layer to display data to the user and handle user input. In Figure 2.1 these functionalities are represented by horizontal layers.

However, there may be additional requirements for including concerns which affect all layers of an application. Such concerns include security, error handling, performance monitoring, thread synchronisation and transactions [26]. These features are formally referred to as *cross-cutting concerns* [5,27]. In Figure 2.1 we have modelled the security concern as a vertical layer which affects all other layers.



**Figure 2.1– Horizontal and vertical (cross-cutting) layers in an application**

Cross-cutting concerns have the undesirable property that they cause clutter and noise on program code. For example, in the proposal [8], we present sample C# code whose cohesion is degraded by other cross-cutting concerns.



AOP is a programming paradigm which allows the encapsulation of these cross-cutting concerns and provides constructs to *weave* transparently these cross-cutting concerns into working code. In the project proposal [8], we noted the following definition for AOP from [28]. As this definition is referred within this dissertation and forms a thread for our weaver design, we reiterate the definition here:

*AOP is [...] the desire to make programming statements of the form:  
“In program P, whenever condition C arises, perform action A”*

**Definition 2.1 - A language-neutral definition of AOP**

From Definition 2.1, there is a requirement for *obliviousness* in that the program P should have no knowledge of the action A.

AOP consists of the following different components and these are explained in more details in the next sections:

- i. Join points
- ii. Pointcuts
- iii. Advices
- iv. Aspects

### 2.1.1 Join points

A multitude of events can arise during a program’s execution, such as method calls and exceptions. Join points are the set of events which can arise during the execution of a program. Referring back to Definition 2.1, the condition C is one such join point.

AOP frameworks expose *a join point model* [16,27], which specifies the events that can occur and allows a user to attach an action when the event fires. For our implementation, we propose to advise the join point set to function calls only – i.e. the execution of a particular function.

### 2.1.2 Pointcuts

Pointcuts are used to define the subset of join points on which a specific action should be taken [16,29,30].

### 2.1.3 Advices

Advices define the actions that should be taken when a particular join point has been reached during the execution of a program. Referring back to Definition 2.1, the advice is the action A to be performed. Within the scope of this project, we use the following definitions:

- We say that a program or module is *advised* when an advice is attached to a module (e.g. a function) via a join point/pointcut
- An *unadvised* program or module refers to one on which no advices have been attached.
- *Advising* refers to the process of injecting an advice into a source program.
- A *target function* is the function which is to be identified and advised.
- A *source program* is the original program
- A *computation* is a sequence of F# statements in a source program.

An advice can be primed to execute:

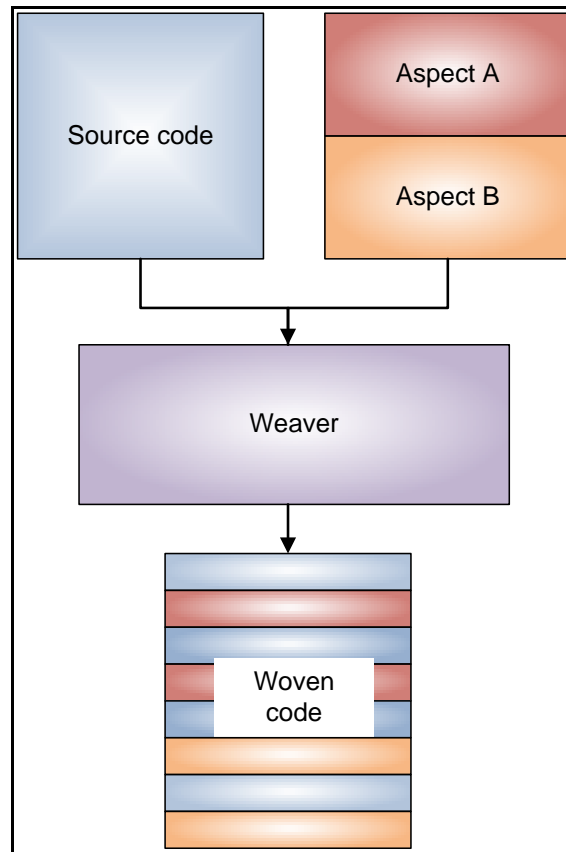
- Before the target function runs. We refer to these as **before advices**.
- After the target function runs. We refer to these as **after advices**.
- Before the target function runs, and also after the function runs. We refer to these as **around advices**.

#### 2.1.4 Aspects

Aspects encapsulate crosscutting concerns [16] [29]. Within an AOP framework, aspects store the pointcut and the advice information.

#### 2.1.5 Weaving

Referring back to Definition 2.1, a program  $P$  should have no knowledge of the action  $A$ . Clearly, there must be a mechanism to combine the source code of program  $P$  with the code defined in advice  $A$ . This mechanism is known as *weaving* [1,2,5,16,26] and is illustrated below, in Figure 2.2 (adapted from [5,31]). In Figure 2.2, there are two aspects, “Aspect A” and “Aspect B” which are weaved to a source program:



**Figure 2.2 – Weaving aspects to source code**

Weaving – the merging of source code with aspects, can be done *statically* or *dynamically*:

- *Static weaving* occurs at compile time. In this case the weaver modifies the source code by identifying the selected join points and injecting the advices.
- Conversely, *dynamic weaving* is a strategy where the weaver inspects running code (or code that is about to be loaded at runtime) and applies advices as specified in the pointcut [16,32].

## 2.2 Functional languages

In the project proposal [8] of the dissertation, we presented a background on functional languages and highlighted the following characteristics:

- Functional programming languages are a sub-set of *declarative programming languages* [17] where the result of a program is achieved through the combination and execution of functions. By comparison, in *imperative languages* the result of a program is achieved through the execution of commands [33].

- Formally, functional languages are those which abide by the rules of *lambda calculus*. In lambda calculus, every construct is a function and hence every construct returns a value [33].
- Functions can be passed in arguments to other functions, and functions can also be returned. Formally, we say that first-order functions can be passed as parameters to, or returned from, higher-order functions [33,34].
- An important concept for functional languages is that of *referential transparency*, i.e. “equals can be replaced by equals”. Referential transparency can assist in reasoning about a function’s output. As [11,17] highlights, the ability to reason about a program is reduced by AOP – as AOP transparently weaves additional functionalities and can alter a function’s output.
- Functional languages have a *type inference system* [33,35], where the types used in a computation do not have to be explicitly declared. The project proposal [8] provides more details on type inference and its impact on AOP.

In this section we introduce some additional characteristics of F# which are used extensively within the project. F# constructs which are specific to the topic being discussed (e.g. *computation expressions*) are introduced as we progress. This section does not aim to be a full introduction to F#. However for a good introduction into F# please refer to [15,33,36,37]. Furthermore, the project proposal [8] of the dissertation describes functional language constructs which we use and encounter during this project, namely *currying*, a more detailed treatment on *type inference* and monads. F# is a descendent of the Meta Language (ML) family of functional languages, and shares some syntax with OCaml [38], hence familiarity with these languages may be useful in comprehending the F# snippets we use in this dissertation.

### 2.2.1 Function signatures

This section on function signatures was not included in the proposal. We cover function signatures here as they are useful to gather information about a function’s purpose. Within F# (and functional languages in general), the function represents the underlying model of computation [17]. As mentioned previously, every function returns a value. As such, every function has an associated *function signature* which represents:

- The set of inputs to the function (*loosely* analogous to input parameters).
- The return type of the function.

A function signature for a function which accepts two inputs of type `string` and returns another `string` would be:

```
string -> string -> string
```

**Listing 2.1 – A sample function signature**

For example, a possible implementation for this function could be:

```
let concatenate (lhs:string) (rhs:string) = lhs + rhs
```

**Listing 2.2 – Sample function implementation**

In the listing above, the function name is “`concatenate`” and the inputs are `lhs` and `rhs`, both *annotated* with the type `string`. The type inference system of F# infers that the return type is also of type `string` [33].

In the function signature shown in Listing 2.1, we simplified by describing the two type parameters (`string -> string`) as the two input parameters of the function. More precisely, the function signature indicates a sequence of *transformation* from one type to another [33,37]. A more accurate reading of the function signature would be: “the function `concatenate` accepts one input of type `string` and returns another (*anonymous*) function. The new function accepts an input of type `string` as its parameter and returns an output of type `string`”. In effect this indicates that the following is possible:

```
let partialConcatenate = concatenate "foo"
```

**Listing 2.3 – Currying example**

This returns another function which is bound to the `partialConcatenate` identifier and has the function signature:

```
string -> string
```

**Listing 2.4 – Function signature for the curried function**

The signature above can be read as: “the function `partialConcatenate` transforms an input `string` into another `string`”. This is an example of *currying* [33].

Understanding function signatures enables one to gather information about the behaviour of a particular function. For example our weaver function (called `weave`) which we introduce in Section 4.3.4 has the signature:

```
aspectInformation -> Expr -> Expr
```

One can read this function as follows: “the function `weave` accepts an object which provides information on the aspect (`aspectInformation`), and returns another function. This new function accepts a representation of the source program (as an object of type `Expr`) and returns an object which represents the output (i.e. weaved) program as an object of type `Expr`”.

### 2.2.2 F# Interactive (fsi)

F# Interactive is a useful tool used in this project to develop the different components. F# Interactive provides a REPL (Read, Evaluate, Print, Loop) environment to test F# snippets [39]. This approach can assist the development process: once a fragment of F# produces the required output and is tested, the code is then copied into *F#* files with an extension “.fs” for later compilation into an assembly [36] (the project proposal [8] of the dissertation presents more information on .Net assemblies).

Most of the snippets in the code listings of this project can be run via fsi.

### 2.2.3 Active patterns

This section covers pattern matching and active patterns. As we developed the weaver, we made extensive use of this construct, hence it is useful to provide a short explanation on active patterns. [36] provides a succinct introduction to active patterns and covers the *complete* and *incomplete patterns* constructs –this section focuses on *complete* patterns only.

Pattern matching is akin to a switch statement in C++ or C# [40], but with additional functionality such as the ability to analyse the input code and then “branch” into the appropriate path [36]. Pattern matching starts with the keyword `match`, followed by the identifier to be matched. The different possible matches are separated by pipes (|) [36]. A complete active pattern is a function which accepts an input and returns one *pattern* (also known as *case*) [36,37]. To illustrate, the example in Listing 2.5, below, shows an active pattern that analyses an input population value and returns a pattern indicating whether this population describes a *village*, *town*, *city* or *megalopolis* – obviously in real code, the program in Listing 2.5, below, would not have these *magic numbers* and error checking would be in place to validate the input (i.e. if population is negative):

```

let (|Village|Town|City|Megalopolis|) (population : int) =
  if population < 100 then
    Village
  else if population >= 100 && population < 10000 then
    Town
  else if population >= 10000 && population < 1000000 then
    City
  else
    Megalopolis

```

**Listing 2.5 - Simple active pattern**

We can call the active pattern through a pattern matching construct as shown below:

```

let eval population =
  match population with
  | Village -> printfn "Village"
  | Town -> printfn "Town"
  | City -> printfn "City"
  | Megalopolis -> printfn "Megalopolis"

```

**Listing 2.6 - Calling the active pattern**

In the `eval` function above, the integer value of `population` is passed to the active pattern. The active pattern behaves like a function, and based on the value of `population` returns one of the possible patterns (`Village` or `Town` or `City` or `Megalopolis`) back to the caller. The pattern matching construct then branches appropriately.

## 2.3 Summary

This Chapter has introduced aspect-oriented programming (AOP) and functional programming with an emphasis on F#. AOP is a programming paradigm which allows the clean encapsulation of concerns and the *weaving* of those concerns into an existing program.

We noted the following basic constructs of AOP:

- *Join points* define the set of events on which one can attach actions to. Examples of event include method executions or exception handling.
- *Pointcut* allows the selection of events. For example, it allows one to select a specific join point such as the execution of a particular method.
- *Advices* represent the action to perform when the event has occurred. Advices can be primed to execute before, after or “around” the event occurrence.
- *Aspects* represent the logical grouping of pointcuts and advices into one cross cutting concern –for example we use the term *security aspect* to define such a grouping.
- *Weaving* is the process which transparently merges an aspect into a source program.

We provided a short background on functional languages and on F#. Function signatures were introduced and we explained how they can be interpreted. We also covered active patterns - a construct which we use extensively within this project.



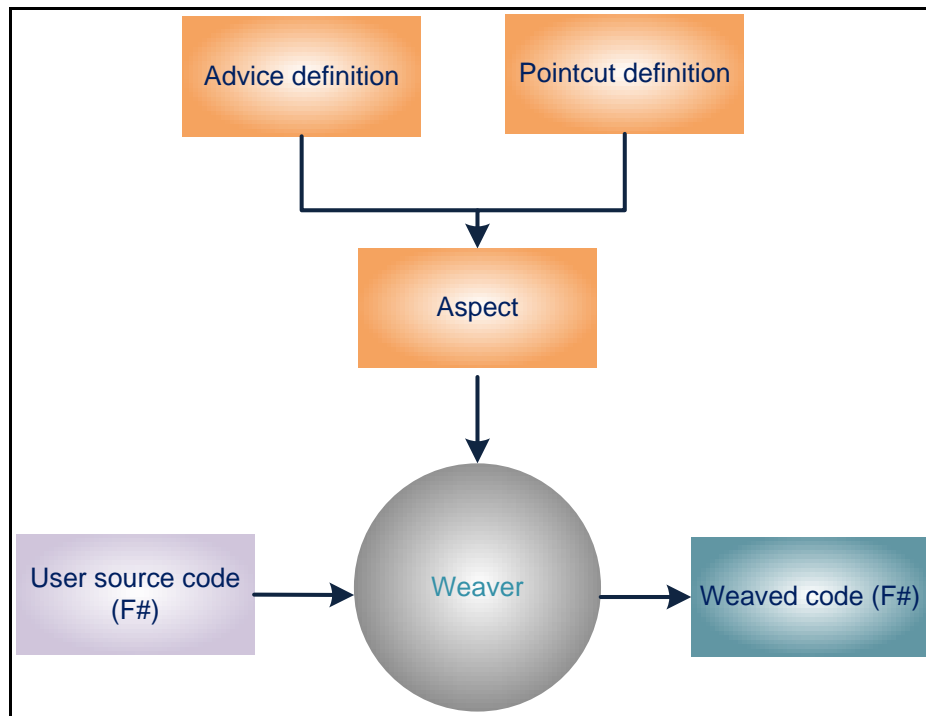
## 3. Requirements and development approach

This chapter presents the requirement for our framework and presents the development methodology used.

### 3.1 Requirements

- The framework will be developed in F# and advise other programs written in F#.
- *Join point*: Our join point model will be the execution of functions.
- *Pointcut*: The framework will provide the ability to select join points by allowing the user to specify the target function. In the project proposal of the dissertation, we highlighted how AspectJ allows user to name the pointcut for re-use [16]. Our framework shall allow *named* and *anonymous* pointcuts.
- *Advices*: the framework will allow the user to specify the additional modules to inject into a target. We propose to implement the ability to insert advices:
  - *Before* the execution of a target function.
  - *After* the execution of a target function.
  - *Around* the execution of a target function.
- Our framework shall implement a *weaver* to inject the advices into the target program. Within this project we implemented two weavers, one using *monads* (Section 4.2), and another using metaprogramming technologies (Section 4.3). As discussed in Section 2.1, an important design consideration will be to ensure our weaver satisfies the concept of obliviousness. We propose to develop a static weaver (Section 2.1.5).
- Our framework shall allow *aspects* to be defined such that pointcuts and advices can be grouped.

Figure 3.1 illustrates the different components of the framework and how they process some user source code (leftmost box “User source code (F#)”).



**Figure 3.1 – Framework components**

We can note that a central component of the framework is the weaver. The weaver design impacts the design of the aspect, which in turn impacts the pointcut and the advice design.

### 3.2 Development tools and methodology

The framework was developed using the following software:

- F# 2.0. [3,14].
- F# interactive (fsi) and Visual Studio 2010 [41].
- Version 4.0 of the .Net Framework [42]<sup>1</sup>.

A *test driven approach* was taken [43]: unit tests were built using xUnit [44] during the development phase of the project.

<sup>1</sup> As described in the project proposal [8], the .Net Framework provides a program runtime environment (the *Common Language Runtime* or CLR) and a set of libraries (the *Framework Class Libraries* or FCL) [59], which are analogous to Java's Virtual Machine [117] and the Java Class Library [118].

## 4. Framework design and implementation

This chapter details the implementation of the framework. We emphasise the weaver design and implementation, as its design impacts all other components of the framework (as shown in Figure 3.1, above).

### 4.1 Weaver design

A relatively large amount of design and development effort was focused on developing a weaver as it is a central component of the application. Recall from Section 2.1 that a successful weaving strategy abides by the rule of *obliviousness*.

Two weaving strategies were investigated and developed:

- A *monad-based* weaver (Section 4.2), where we shall see that the advice is encapsulated in what is termed a *monadic type* [33]. Monads provide constructs to weave these advices into a program.
- A weaver based on *metaprogramming technologies* (Section 4.3), where advices are implemented as regular F# functions. Special language functionalities are provided to weave the advice function into the source program.

### 4.2 Monad-based weavers

#### 4.2.1 Theoretical background on monads

The use of monads as a weaving strategy was postulated in [45]. As per the project proposal [8], our first approach for a weaver is the use of monads.

An introduction to monads is given in [46], and [47] provides more information on these programming construct. Within this section we provide a short introduction to monads, monads in F# and recap the suitability as a weaving strategy.

Monads originate from the branch of mathematics known as *group theory* [48]. In functional languages, a program is usually written such that for a known set of input(s), the program gives one output. This is a fundamental description of functional languages and allows one to *reason* about a program. However, there are often valid use cases which require a program to apply a side effect, for example printing to screen. Monads are used in functional programming languages to encapsulate these side effects [49], without affecting the original computation or logic of the program.

The core of the monad is the *monadic type* [33]. The purpose of the monadic type is to *augment or enhance* the current program, e.g. by applying some relevant side effect. A common notation for the monadic type is  $M\langle 'a \rangle$  [33,48] where  $'a$  is a generic type representing the *original* type of the computation, and the monadic type  $M\langle 'a \rangle$  represents the functionalities/behaviours added by the monad.

Monads further require the use of two operators, known within the literature as *return* and *bind* operators [33,45]:

- *Return* is a first order function which *lifts* the original type  $'a$  to the monadic type  $M\langle 'a \rangle$ , and therefore has the signature  $('a \rightarrow M\langle 'a \rangle)$ . This function is also known as *unit* [45].
- *Bind* is a higher-order function which allows the composition of monadic types together. In the project proposal, [8], we heuristically derived its signature:

$$M\langle 'a \rangle \rightarrow ('a \rightarrow M\langle 'b \rangle) \rightarrow M\langle 'b \rangle.$$

**Figure 4.1 – Function signature of the *bind* operator**

From the signature above, *bind* is a function which takes a monadic type, and a function which returns a monadic type  $( 'a \rightarrow M\langle 'b \rangle )$ . The function signature implies that *bind* has sufficient knowledge of the monadic type  $M\langle 'a \rangle$  to *unwrap* the original value (of type  $'a$ ) and pass in this value into the first order function (the second parameter).

Combined together, these operators allow functions to be chained together into a computation. To illustrate, consider the *forward pipe* operator  $|>$  (with signature  $'a \rightarrow ('a \rightarrow 'b) \rightarrow 'b$ ) [50]. We can construct the following computation, if  $f(x)$  has a signature  $(int \rightarrow int)$ :  $5 |> f$ . In this example, we are passing in the integer value  $5$  into the function  $f(x)$ . We expect the result to be of type  $int$ , given that  $f(x)$  accepts an integer and returns another integer. We assume now that there exists a requirement to augment the value  $5$  (of type  $integer$ ) to another type  $M\langle int \rangle$ . Clearly, we cannot reuse the forward pipe operator, as the function  $f(x)$  accepts an  $integer$ , not a value of the type  $M\langle int \rangle$ . However, the solution is to create another function which has the same signature as the *bind* function, and making the changes to the function  $f(x)$ :

- Let  $f(x)$  have the signature  $(int \rightarrow M\langle int \rangle)$
- Let *bind* be represented with the notation  $>>=$

We can reconstruct the original computation chain as follows:

```
return(5) >>= f
```

**Listing 4.1 – An example of monads**

From [45,49] and the discussion above, monads are suitable as a weaving strategy as we can use the monadic type  $M<'a>$  to encapsulate our advices and use the bind operator to recreate the original computation of the unadvised program.

#### 4.2.2 F# support for monads

Within F#, a construct known as *computation expression*<sup>2</sup>, provides syntactic support for monads. Smith [15] defines computation expressions as a construct which allows one to augment a certain computation with additional behaviour. Computation expressions consist of the following components:

- An F# *class*. In order to participate in a computation expression, the methods of the class are required to conform to specific function signature(s)<sup>3</sup>. For example, a member method called `Bind` will be automatically be used by the compiler as the monadic bind operator.
- A *computation expression builder* [15] .

Listing 4.2, below, illustrates a computation expression which applies a side effect (printing to screen). In the example, the original type (`'a`) and the monadic type ( $M<'a>$ ) are both integers (i.e. `'a` is of type `int`).

---

<sup>2</sup> Computation expressions are also known as *workflows* [15]. Within this document, we use the term computation expression when describing an implementation in F# and the term monad to describe the general concept.

<sup>3</sup> Please refer to Table 10.1 of [15] for a full list of methods which are given special treatment by the compiler and hence can participate in a computation expression.

```

1  type SideEffectBuilder(msg : string) =
2      let msg = msg
3
4      member this.Bind((x : int), (rest : int -> int)) =
5          printfn "The message: %s." msg
6          printfn "The result: %i." x
7          rest x
8
9      member this.Return (x : int) = x
10
11 let sideEffect msg = new SideEffectBuilder(msg)
12
13 let double x = x * 2
14
15 let printmsg = sideEffect "this is the injected test message"{
16     let! v1 = double 5
17     let! v2 = double 10
18     return 0
19 }
20
21 printmsg

```

**Listing 4.2 – Computation expressions in F#**

The computation expression starts by defining the type of the computation expression builder: `SideEffectBuilder` (lines 1 to 9). The class has a `Bind` method (line 4) which prints a custom message to the screen and returns a value which has the same type as the monadic type. As the original and monadic types are both integers, the `Return` method accepts an integer and returns the same integer. The `SideEffectBuilder` type has a constructor (line 1) which accepts a custom message.

On line 11 of Listing 4.2, we define the computation expression builder by creating an instance of the `SideEffectBuilder` class (line 11). We apply the side effect on a function called `double`, which doubles the input value (defined in line 13).

This is followed by the computation expression itself, which is bound to an identifier `printmsg`. The computation expression (lines 16 to 18) has the appearance of regular F# code, except for the presence of the “!” marker appended to the `let` expressions. The compiler translates marker this to a call to the computation expressions builder’s `Bind` method. The first parameter of the bind expression is the result of the first call of the `double` function (`double 5`). The second parameter of the computation is the function call in the following line (`double 10`).

To illustrate the action of the compiler, the lines:

```
let! v1 = double 5
let! v2 = double 10
return 0
```

**Listing 4.3 – Computation**

are expanded (also referred to as *de-sugared*) by the F# compiler into the following lines:

```
sideEffect.Bind(double 5,
    fun v1 -> sideEffect.Bind (double 10,
        fun v2 -> sideEffect.Return( 0 ) ) )
```

**Listing 4.4 – De-sugared form of listing 4.3**

We can draw parallels with the example given in Listing 4.2 with AOP. Namely, the Listing shows a weaver which applies *after advices* to target code. The after advice is the side effect of printing to the screen and the target code is the computation wrapped within the curly brackets. This example forms the basis for a more complex weaver implementation.

#### 4.2.3 Development of a monad-based weaver using computation expressions

This section discusses the implementation of a monad-based weaver using F# computation expressions. The source code is available in the CD in Appendix C and in the online repository (Section 4.5).

Our implementation uses three single case *discriminated unions* [51] to represent before, after and around advices. These discriminated unions are concrete implementation of the monadic types  $M\langle a \rangle$ . An F# discriminated union can be compared to the object-oriented construct of an abstract base class with a single level of inheritance [33]. A *single case* discriminated union is therefore analogous to an abstract base class with only one child. Listing 4.5 shows the definition of advice types:

```
type BeforeAspect<'T> =
    | Before of 'T

type AfterAspect<'T> =
    | After of 'T

type AroundAspect<'TBefore, 'TAfter> =
    | Around of 'TBefore * 'TAfter
```

**Listing 4.5 – Types of advices**

Single case discriminated unions are used as they have some advantages when performing pattern matching and result decomposition [33]. The discriminated unions are of a generic type `'T`, which we can use to encapsulate a *lambda* function (i.e. an anonymous function [33,52]) representing the function to execute. Note that the `AroundAspect` is different in that it requires two types: `'TBefore` and `'TAfter`. We expand on this difference in Section 4.2.3.4. In the example below, the advice is a lambda function which takes any input and returns `unit` [36] (`unit` is equivalent to `void` in C# or C++).

```
let beforeaspect = Before( fun _ -> printfn "running advice.")
```

**Listing 4.6 – Lambda representing a before advice**

We now cover the specific implementations of the before, after and around computation expression builder types.

#### 4.2.3.1 The *BeforeBuilder* computation expression builder

The computation expression builder type which applies a before advice is shown in Listing 4.7:

```
1 type BeforeBuilder() =
2
3     member this.Bind(Before(aspectfunc), func) =
4         aspectfunc() // execute the advice
5         func()
6
7     member this.Return(value) = Before(value)
```

**Listing 4.7 – Computation expression builder for the before advice**

The parameters of the `Bind` function are:

- A monadic type which is constrained to the `Before` case.
- A function with the signature `(unit -> BeforeAspect<'b>)`.

#### 4.2.3.2 *BeforeBuilder* Example

We demonstrate the use of the before computation expression builder by showing how a message is printed to screen before a target function `targetFunction()` executes. The function `targetFunction()` has the signature `(unit -> bool)`:

```
let targetFunction() = printfn "executing target function."
                       true
```

**Listing 4.8 – Sample target function**



`targetFunction()` does not do anything very interesting, except for printing a message to screen and constantly returning `true`. A computation expression which uses the computation expression builder is shown in Listing 4.9:

```
1 let before = BeforeBuilder()
2
3 let beforeTestRunner(targetfunc) =
4     before {
5         let! res = beforeaspect
6         return targetfunc()
7     }
8
9 // below is code to run the computation expression
10 beforeTestRunner(targetFunction) |> ignore
```

**Listing 4.9 – Example usage of the before computation expression builder**

The computation expression bound to the `beforeTestRunner` identifier (line 3) and accepts a function of signature `(unit -> 'a)`, which matches our concrete implementation `targetFunction()` - which has the more *constrained* type `(unit -> bool)`.

After running the computation expression, the following expected output is printed to screen:

```
running advice.
executing target function.
```

#### 4.2.3.3 The AfterBuilder computation expression builder

The builder for the after aspect is similar to the builder for the before aspect. The computation expression builder type is shown below:

```
1 type AfterBuilder() =
2
3     member this.Bind(After(aspectfunc), func) =
4         let (After(res)) = func() // call the target function and store the result
5         aspectfunc() // execute the advice
6         After(res) // return the result
7
8     member this.Return(value) = After(value)
```

**Listing 4.10 – Computation expression builder for the after advice**

The class is similar to the `BeforeBuilder` type and has a `Bind` and `Return` method. The main differences are:

- We now use the `After` discriminated union.
- Within the `Bind` method, we are required to re-order the sequence of calls such that we begin by calling the target function and store its result (line 3). We then execute the advice and return the results of the target function (lines 4 and 5).

An example usage is shown below:

```
let afterAspect = After( fun _ -> printfn "running advice.")

let afterTestRunner(targetFunc) =
    after {
        let! res = afterAspect
        return targetFunc()
    }

afterTestRunner(targetFunction) |> ignore // Running the computation expression.
```

**Listing 4.11 – Example usage of the after computation expression builder**

After running the computation expression, the following expected output is printed to screen:

```
executing target function.
running advice.
```

#### 4.2.3.4 The *AroundBuilder* computation expression builder

The around aspect must accommodate for the before and after advices being different. The around aspect discriminated union is repeated below:

```
type AroundAspect<'TBefore, 'TAfter> =
    | Around of 'TBefore * 'TAfter
```

The single case discriminated union requires a *tuple* [15], where `'TBefore` is a lambda function representing the before advice, and the `'TAfter` is a lambda function representing the after advice.

The computation expression builder is shown in the Listing below:

```
1 type AroundBuilder() =
2
3     member x.Bind(Around(beforeFunc, afterFunc), func) =
4         beforeFunc() // call the before aspect
5         let res = func() // call the target function and store the result
6         afterFunc() // call the after aspect
7         res // return the result
8
9     member x.Return(value) = Around(value)
```

**Listing 4.12 - Computation expression builder for the around advice**

The Bind method performs *sequential* calls [53] to the before advice (line 4), the target function (line 5) and finally the after advice (line 6). The result of calling the target function is stored in a temporary variable `res` which is returned at the end of the computation. A sample usage is shown below:

```
1 // Create the advice to execute and the aspect
2 let beforeF = fun _ -> printfn "running before advice."
3 let afterF = fun _ -> printfn "running after advice."
4 let aroundaspect = Around( beforeF, afterF)
5
6 let aroundTestRunner(targetfunc) =
7     around {
8         let! res = aroundaspect
9         return (targetfunc(), None)
10    }
11
12 aroundTestRunner(targetFunction) |> ignore // Running the computation expression.
```

**Listing 4.13 – Sample usage of the around computation expression builder**

In the Listing shown above, we begin by specifying two lambda functions which represent the before and after functions. We then create the around aspect which is bound to an identifier called `aroundaspect`. When the computation expression is run, the following expected results are printed to screen:

```
running before advice.
executing target function.
running after advice.
```

#### 4.2.4 Discussion on monads

From the implementation provided, we note the following advantages when using a monad-based weaver:

- Monads provide clean access to function boundaries.
- Weaving is trivial – the `Bind` method does most of the required weaving without changes to the original computation order.
- Monads allow the encapsulation of any advices - the monadic type  $M<'a>$  can encapsulate a wide range of advices.
- Monads are well documented.

We note the following disadvantages when using a monad-based weaving :

- Constructing a pointcut selection language is complex and not intuitive.

- Monads do not easily comply with the important concept of obliviousness. To illustrate, we refer back to Listing 4.1 shown previously, which we repeat below:

```
return(5) >>= f
```

Implementing a monad-based weaving required changes to the function  $f$  as we need to change its signature from  $(int \rightarrow int)$  to  $(int \rightarrow M<int>)$ . Arguably this can be achieved via a higher order function having the signature  $(int \rightarrow int) \rightarrow (int \rightarrow M<int>)$ .

- A monad-based approach lacks granularity. To illustrate, consider the following computation which achieves the business requirements of committing some data into a database:

```
let save() = preptxn()
           let transactionResult = executetxn()
           transactionResult
```

**Listing 4.14 – A computation representing a trivial business logic**

Using the monadic constructs shown previously, it is trivial to attach before, after or around advices to *the top level function* `save()`. However, we cannot trivially attach an advice to the inner functions called by `save()`. As such, we cannot address a requirement to inject an advice to, say the `executetxn()` function without substantially re-engineering the `save()` function.

#### 4.2.5 Summary of a monad-based weaver

This section has provided a background on monads and explained their usage within functional programming. Parallels between monads and AOP were drawn to provide justification to use monads as a weaving strategy. F# provides some (admittedly fairly unintuitive) language constructs for monads called *computation expressions*. Our implementation of monads which apply advices before, after and around a target function was implemented using computation expressions.

We next highlighted some advantages and disadvantages of a weaver built using monads. The main disadvantage of this approach is that monads do not provide a satisfactory level of granularity – namely it is trivial to advise *top level function* (functions calling other functions), but difficult to advise functions called by the top level function. The disadvantages of a monadic approach lead us to design and develop a finer grained yet more complex weaving strategy through the use of *metaprogramming* technologies.

### 4.3 Code quotation-based weaver

A definition [34] of metaprogramming is presented below:

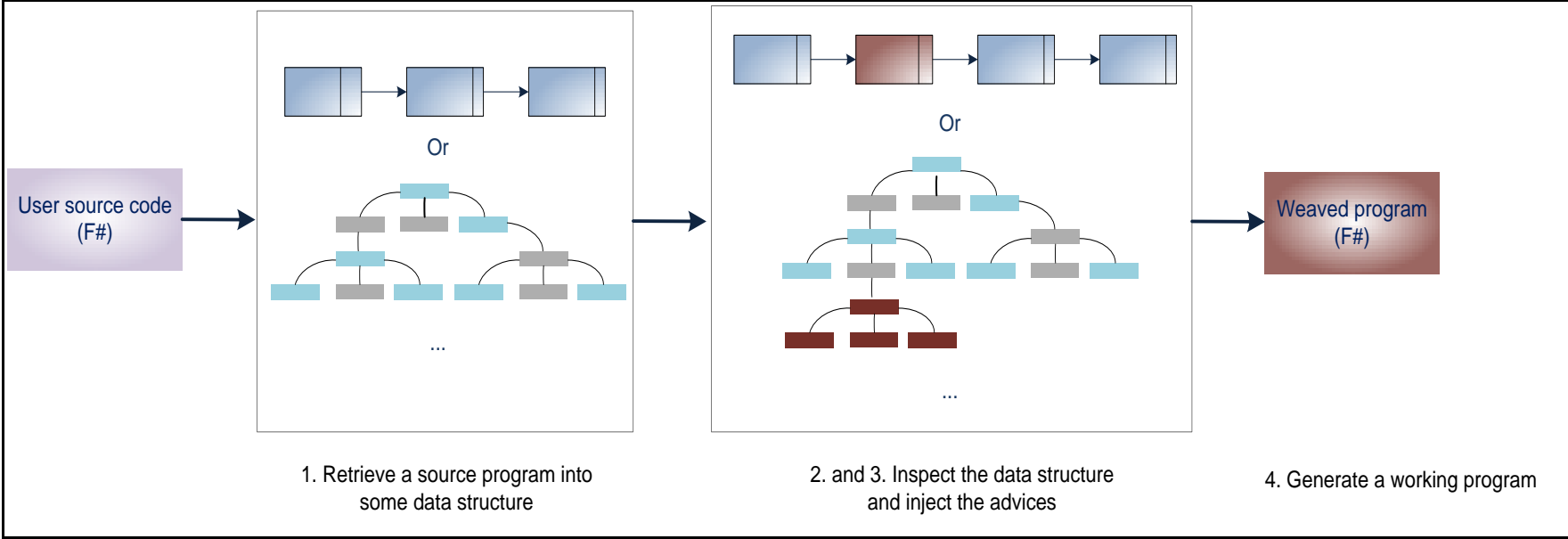
Metaprogramming is a term which refers to a computer program which *transforms* a source program into another program.

#### **Definition 4.1 - A definition for metaprogramming**

This is achieved by treating the source program as input data. Compilers are classical examples of programs that carry out such transformations [34,54]. Conceptually, a weaver performs the same transformation – namely the weaver takes a source, un-advised program, and manipulates it to inject the relevant advices. From Definition 4.1, the following steps are required to implement a weaver-based on metaprogramming:

1. Retrieve a source program and convert it to some data structure.
2. Analyse the data structure and detect the join points defined in the pointcut.
3. Amend the data structure and inject the advices.
4. Generate a working program from the amended data structure.

Figure 4.2, on the following page, illustrates the weaving strategy.



**Figure 4.2 - Weaving strategy using metaprogramming technologies**

In Figure 4.2, the data structure for the source and advised program is illustrated as a *linked list* or a *tree* – the actual data structure is not important, as long we can traverse and amend it.

There are two general metaprogramming techniques for transforming a source program into another [31,55]:

- *Source transformation*: is the act of transforming the source code of the base program before execution. This can be achieved through compiler switches or compiler extensions. In Section 4.6 we present some analysis on the F# compiler. In all cases, source transformation technologies involve traversing an input data structure representing the source code and applying changes where required.
- *Dynamic reflection*: can be loosely described as manipulating the source program at runtime. This can be achieved by using a *meta-representation* [31] of the programming constructs in the base code. Examples of these meta-representations include *meta-objects* [56,57]. Within F#, we can use objects from the `System.Reflection` [58] library to perform runtime analysis of the source program.

These two technologies are not mutually exclusive, for example [55] presents a weaver which is a hybrid between source transformation and dynamic reflection.

The .Net Framework [42,59] provides several libraries which expose metaprogramming functionalities. These libraries provide access to both source transformation and dynamic reflection techniques. Table 4.1, below, describes these libraries and their comparative advantage. Based on the information in Table 4.1 we discuss our chosen technology to develop our weaver.

#### **4.3.1 Metaprogramming technologies in the .Net Framework**

Table 4.1, below, shows the different libraries available in the .Net Framework which can be used for metaprogramming. This Table also includes a discussion on *code quotations* [15,36,60,61] which is a metaprogramming construct specific to F#.

**Table 4.1 – Summary of metaprogramming technologies in the .Net Framework**

Reflection	
Description	The methods within the .Net <code>System.Reflection</code> namespace [58] provide functionalities to inspect and instantiate types from compiled or executing code.
Source code information returned	Information on compiled / executing code (e.g. functions, properties) are made available as collections of objects (e.g <code>MethodInfo</code> [62]).
Advantages	Mature API (available since .Net 1.0 [34]).
Disadvantages	Does not give any indication as to how the program operates [15].
CodeDOM	
Description	Provides a document-oriented approach to code generation [34,63]. CodeDOM is mainly focused on generating a code graph code which can be compiled immediately or saved to file for later compilation [34,64].
Source code information returned	Graph of objects within the <code>System.CodeDOM</code> namespace [63].
Advantages	Allows generation of source code in a variety of .Net languages [34]. This <i>might</i> be useful for debugging purposes, however as noted in the disadvantages column, generating F# code is not natively supported.
Disadvantages	<p>Not actively developed by Microsoft in recent years [34].</p> <p>No functionalities exist to convert a string (i.e. an input source file) into an object graph which is critical for further manipulation.</p> <p>As of the time of writing, there are no native <i>providers</i> [34,65] to generate code into F# CodeDOM. Instead the F# CodeDOM provider is part of a separate experimental module – the <i>F# PowerPack</i> [66].</p>
LINQ Expression Trees	
Description	Presents code as a data structure for analysis, transformation and compilation [67,68].
Source code information returned	Source code is expressed as a data structure where each node is an object deriving from the <code>Expression</code> class [69]. Collectively, the structure represents an abstract syntax tree (AST) [70].
Advantages	Relatively mature technology – present in .Net since the .Net Framework 3.5 [71].
Disadvantages	Support for expression trees is not available for F# in the core.Net 4.0 libraries, but available as part of the experimental <i>F# PowerPack</i> [66].



**Table 4.1 (continued)**

Code Quotations	
Description	Allows to retrieval of F# code into a data structure through the use of keywords [15,36,72].
Source code information returned	The data structure returned is an abstract syntax trees (AST) where every node of the tree is an object deriving from <code>Quotations.Expr</code> [73].
Advantages	<p>Allows the generation of source code AST via F# keywords (i.e. <i>quotation markers</i>).</p> <p>Dedicated functionalities are available to traverse and modify the AST – which we shall cover in later sections of this document.</p> <p>Provides a mechanism where it is possible to define <i>holes</i> in the AST. At a later phase, these holes can be filled in by an actual sub tree –the act of defining holes and then filling them is known as <i>splicing</i> [15,72].</p> <p>Moreover, methods exist to convert a <code>Quotations.Expr</code> object into an <code>Expression</code> object [66] –where the latter object is of a type used in Linq. This allows for additional work to be carried out in C#, where expression trees are implemented as part of the core .Net libraries.</p>
Disadvantages	<p>Documentation available but generally quite brief.</p> <p>Some additional functionalities is provided in the experimental F# PowerPack [66].</p>

### 4.3.2 F# code quotations

Code quotations are our preferred choice for a weaver. This choice is motivated by the fact that it is trivial to return a fragment of F# code into a data structure. This data structure (as noted in Table 4.1) is known as an *abstract syntax tree* (AST) [61,70,74]. Within this dissertation we refer to this data structure as an AST or an *expression tree*. As mentioned in Table 4.1, every node of the expression tree derives from the type `Quotations.Expr` [73].

Listing 4.15 shows a simple use of code quotations where we use *quotation markers* [15,72] `<@ @>` to wrap F# code and get a data structure representing the wrapped code. The example below is a string replace function which is a façade over the .Net Framework’s `String.Replace()` method [75]. This method replaces a character in a string with another character. The implementation is trivial:

```

let stringReplace ( source : string ) ( oldChar : char ) ( newChar : char ) =
    source.Replace(oldChar, newChar)

let quote = <@ stringReplace "test" 't' 'e' @> // retrieving the AST.

```

**Listing 4.15 – Implementation of the `stringReplace` function.**

The Listing above outputs the following in *F# Interactive (fsi)* [39]. The abstract syntax tree is shown in italics below:

```

val stringReplace : string -> char -> char -> string

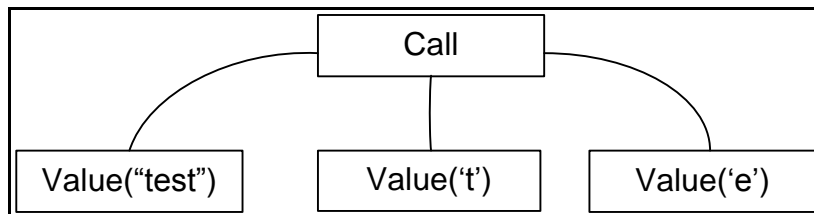
val quote : Quotations.Expr<string> =
    Call (None, System.String stringReplace(System.String, Char, Char),
        [Value ("test"), Value ('t'), Value ('e')])

```

**Listing 4.16 – fsi output**

The lines in italics indicate that the abstract syntax tree is an object of *generic* type [76] `Quotations.Expr<'a>`, where `<'a>` is the return type of the expression [15] – here of type `string`. Wrapping F# code with the `<@ @>` quotation markers returns an object of the generic type: `Quotations.Expr<'a>`, whereas wrapping F# code with the `<@@ @>` quotation markers returns object of the non-generic type: `Quotations.Expr` [15].

In Listing 4.16 above, we have four nodes in our AST, which we illustrate in the figure below:



**Figure 4.3 – Abstract syntax tree for the `stringReplace` function**

The parent node is a node of type *Call*, which represents a function call [15,77]:

```

Call (none, System.String stringReplace (System.String, Char, Char), [ Value
("test"), Value('t'), Value('e') ] )

```

The leaf nodes are of type *Value* which represents the literal values `"test"`, `'t'` and `'e'` [15,78]. Many different types of nodes exist, representing other F# constructs such as *let bindings*, *tuples*, *curried*

functions – more information is available from [73],- and we cover some of the common ones in Section 4.3.3. As the AST is a recursive data structure, each node may represent another object of type `Quotations.Expr`.

We focus on the Call nodes as these are the nodes on which we need to attach before, after or around advices.

From Listing 4.16 we can *decompose* the Call node into three parameters:

1. The object that the function is being called on, which in the example shown in Listing 4.16 case is `none`.
2. The name and signature of the function. More precisely, the second parameter is an object of type `MethodInfo` which contains *metadata* about the method (or function) being called [62].
3. The last parameter is a list of the function parameters. In this example, the three parameters are bound to the string `"test"` and the characters `'t'` and `'c'`.

Using *pattern matching* [15,36] it is possible to recursively traverse an AST with the objective of identifying and inspecting each node. Each node can be inspected and decomposed into its constituent parts – e.g. retrieve the mutable `MethodInfo` object for inspection and modification.

The general approach to traversing the input AST (and the one generally described in literature [15,36,79]) is to write a recursive function which carries out pattern matching using *active patterns* [15,36,80]. There are many active patterns available to decompose an AST, and these are documented in [81]. For ease of use, however, Microsoft has provided three *general* patterns which cover all other (more fine grained) patterns [15,82]:

1. `ShapeVar` matches a value.
2. `ShapeLambda` matches a function value.
3. `ShapeCombination` matches “anything else”, for example a combination of other nodes – a simple example is the Call node which combines other nodes. This pattern allows one to “drill” further into sub-expressions.

We can develop a simple *non-tail recursive* function `qa` (quotation analyser) to traverse an input AST and output the same AST. The Listing below shows the function `qa`:

```

let rec qa expr =
  match expr with
  | ShapeVar(var) -> Expr.Var(var)
  | ShapeLambda(var, lambdaBody) -> Expr.Lambda(var, qa lambdaBody)
  | ShapeCombination(h, exprs) -> RebuildShapeCombination(h, exprs |> List.map(qa))

let result = qa quote

```

**Listing 4.17 – A general recursive function to traverse an expression tree**

The Listing below shows the (unexciting!) fsi outputs when the input is the AST from the `stringReplace` function.

```

val qa : Expr -> Expr
val result : Expr =
  Call (None, System.String stringReplace(System.String, Char, Char),
    [Value ("test"), Value ('t'), Value ('e')])

```

**Listing 4.18 – Output when parsing the input expression tree via the quotation analyser in Listing 4.17**

We can now modify the `qa` function, such that it transforms the input AST and swaps the new character with the old character (i.e. the last two parameters of `stringReplace` are swapped). In Listing 4.19, below, the changes are highlighted in red.

```

let rec qaReverse expr =
  match expr with
  | Call(obj, methBody, args) ->
    let firstParameter = args.Item 1
    let secondParameter = args.Item 2

    let reversedArgs = [args.Item 0; secondParameter; firstParameter]
    Expr.Call(methBody, reversedArgs)
  | ShapeVar(var) -> Expr.Var(var)
  | ShapeLambda(var, lambdaBody) -> Expr.Lambda(var, qaReverse lambdaBody)
  | ShapeCombination(h, exprs) -> RebuildShapeCombination(h, exprs |>
List.map(qaReverse))
let resultReverse = qaReverse quote

```

**Listing 4.19– A recursive function to traverse and apply transformations to a quotation tree**

In the body of the `qaReverse` function, we have added a finer grained match to the `Call` pattern. When we have matched the (only) call to `String.Replace`, we swap the second and third parameters of the call to `String.Replace`. Obviously, `qaReverse` is not very generic, but serves to illustrate the general concept of pattern matching and AST transformation.

The results from F# Interactive are shown below, where the inversion of parameters is in bold and underlined:

```
val qaReverse : Expr -> Expr
val resultReverse : Expr =
  Call (None, System.String stringReplace(System.String, Char, Char),
    [Value ("test"), Value ('e'), Value ('t')])
```

**Listing 4.20 – Transformed AST**

It is possible to compile and execute the ASTs bound to the `result` and `resultReverse` identifiers using methods available in the F# PowerPack experimental library [15,66]. In this case, `result` and `resultReverse` evaluate to “eese” and “ttst”, respectively.

It is important to note an important drawback when attempting to retrieve the AST of an F# computation (e.g. a sequence of function calls) through quotation markers, as we have done so far in Listing 4.15. Namely, it is not possible to retrieve the AST of any functions called within the computation – i.e. we have the same issue as for monads and can only retrieve the AST of the top level function and not any function called by the computation.

One F# *attribute* does allow function bodies to be included in the returned quotation: the `[<ReflectedDefinition>]` attribute [15,36,83]. This attribute needs to be applied to the *top level function*. The `MethodWithReflectedDefinition` active pattern [84] allows one to inspect the functions called by the top level function.

In this section, we have provided some information on metaprogramming techniques and focused on F# code quotations. Code quotations allow the retrieval of the AST of a fragment of F# code. We followed this discussion with examples where we recursively traversed and manipulated the AST of a trivial `stringReplace` function.

Our approach for a weaver using metaprogramming consists of the following steps:

1. Retrieve the AST of the target code.
2. Detect the calls to the target function.
3. Manipulate the AST to inject advices.
4. Return the modified AST.

### 4.3.3 F# abstract syntax tree nodes

In Section 4.3.2, we noted the `Call` node and we introduced the notion that within F# different nodes exist. For example, nodes exist to represent a `let` binding, or a `curried` function, an `if else`

statement etc... Table 4.2, below, shows the different types of node, and Table 4.3 presents the F# snippets which generate the corresponding node.

Note that our F# snippets use a `commit` function with the following signature – i.e. a function which accepts a username and a date and returns a Boolean value:

```
commit : string -> System.DateTime -> bool
```

and the following function, which accepts a string and returns a unit:

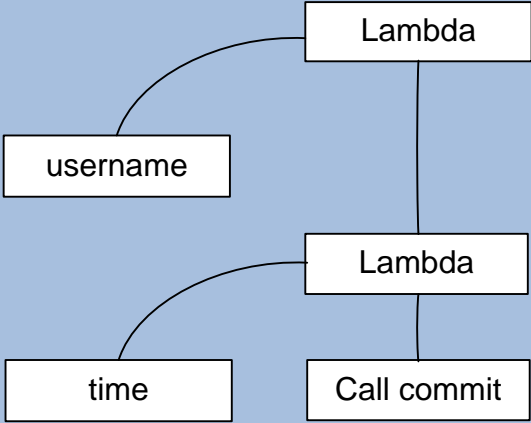
```
logmsg: string -> unit
```

**Table 4.2 – Main types of nodes in F# code quotations**


#	Node	Description
1	<code>Expr.Call</code> [77][85]	Represents a call to a function, when all its parameters are bound – i.e. this is a <i>direct</i> function call.
2	<code>Expr.Let</code> [86][87]	Represents the execution of a function call, with its result bound to an identifier. This node also holds the <i>continuation</i> of the expression – i.e. the rest of the F# code within which the identifier is in scope.
3	<code>Expr.Lambda</code> [88][89]	Represents an anonymous function.
4	<code>Expr.Sequential</code> [90,91]	Represents the execution of an expression (which might be a function call) followed by another.
5	<code>Expr.Application</code> [92][93]	Represents the partial application of a value to a function – i.e. a curried function [33].
6	<code>Expr.IfThenElse</code> [94,95]	Represents an if/then/else statement. The if/then branches may be a call to a function or another expression.

**Table 4.3 – Demonstration of generating F# nodes**

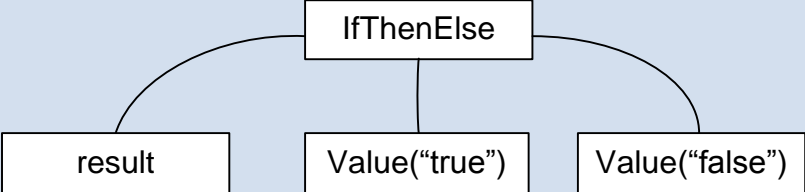
#	Sample Output
1	<p>F# snippet to generate an <code>Expr.Call</code> node:</p> <pre data-bbox="320 439 887 465">&lt;@ commit "User" System.DateTime.Now @&gt;</pre> <p>Output from fsi:</p> <pre data-bbox="320 618 1302 685"><b>Call</b> (None, Boolean commit(System.String, System.DateTime),       [Value ("User"), PropertyGet (None, System.DateTime Now, [])])</pre>
2	<p>F# snippet to generate an <code>Expr.Let</code> node:</p> <pre data-bbox="320 801 1018 949">&lt;@   let res = commit "User" System.DateTime.Now   res @&gt;</pre> <p>Output from fsi:</p> <pre data-bbox="320 1106 1286 1254"><b>Let</b> (res,       Call (None, Boolean commit(System.String, System.DateTime),             [Value ("User"), PropertyGet (None, System.DateTime Now,             [])]), res)</pre> <p>Figure 4.4 shows the AST of the Let node:</p> <div data-bbox="440 1375 1251 1572" data-label="Diagram"> <pre data-bbox="440 1375 1251 1572"> graph TD     Let[Let] --- res1[res]     Let --- Call[Call commit]     Let --- res2[res]   </pre> </div> <p>Figure 4.4 – AST for the Let node</p> <p>Referring to Figure 4.4:</p> <ol data-bbox="368 1733 1355 1980" style="list-style-type: none"> <li>The left child node is a variable which holds the result of the <code>let</code> expression. The left child is an object of type <code>Var</code>.</li> <li>The middle child node represents the body of the <code>let</code> expression. The middle node is of type <code>Quotations.Expr</code>.</li> <li>The right child node represents the <i>continuation</i> of the <code>let</code> expression – more</li> </ol>

#	Sample Output
	<p>precisely the rest of the computation where the variable (defined in the left most node) is in scope. The right child node is an object of type <code>Quotations.Expr</code>.</p>
3	<p>F# snippet to generate an <code>Expr.Lambda</code> node:</p> <pre>&lt;@ commit @&gt;</pre> <p>Output from fsi:</p> <pre><b>Lambda</b> (username,         <b>Lambda</b> (time,                 Call (None, Boolean commit(System.String,                 System.DateTime),                 [username, time])))</pre> <p>Figure 4.5 shows the AST of the lambda node:</p>  <pre> graph TD     Lambda1[Lambda] --- username[username]     Lambda1 --- Lambda2[Lambda]     Lambda2 --- time[time]     Lambda2 --- Call[Call commit]   </pre> <p>Figure 4.5 – AST for the Lambda nodes</p> <p>Referring to Figure 4.5:</p> <ol style="list-style-type: none"> <li>1. Each Lambda function node consists of a left child node and a right child node.</li> <li>2. The left child is a single argument of type <code>Var</code>.</li> <li>3. The right child is an object of type <code>Quotations.Expr</code> which represents the body of the lambda function.</li> </ol>
4	<p>F# snippet to generate an <code>Expr.Sequential</code> node:</p> <pre>&lt;@ logmsg "msg 1"   logmsg "msg 2"</pre>



#	Sample Output
	<pre>@&gt;</pre> <p>Output from fsi:</p> <pre><b>Sequential</b> (Call (None, Void logmsg(System.String), [Value ("msg 1")]),              Call (None, Void logmsg(System.String), [Value ("msg 2")]))</pre> <p>Figure 4.6 shows the AST of the Sequential node:</p>  <pre> graph TD     Sequential[Sequential] --- Call1[Call logmsg]     Sequential --- Call2[Call logmsg]   </pre> <p><b>Figure 4.6– the Sequential node</b></p> <p>Referring to Figure 4.6:</p> <ol style="list-style-type: none"> <li>The leftmost child node represents the first expression to be evaluated, in this case a call to the <code>logmsg</code> function. The node is of type <code>Quotations.Expr</code>.</li> <li>The rightmost child node represents the second expression to be evaluated. The node is of type <code>Quotations.Expr</code>.</li> </ol>
5	<p><b>F# snippet to generate an <code>Expr.Application</code> node:</b></p> <pre>&lt;@ let transactionResult = commit     let carried = transactionResult "User"     carried System.DateTime.Now @&gt;</pre> <p>Output from fsi:</p> <pre>Let (transactionResult,      Lambda (username,             Lambda (time,                     Call (None, Boolean commit(System.String, System.DateTime),                         [username, time]))),      Let (carried, <b>Application</b> (transactionResult, Value ("User")),          <b>Application</b> (carried, PropertyGet (None, System.DateTime Now, []))))</pre>

#	Sample Output
---	---------------

6	<p><b>F# snippet to generate an <code>Expr.IfThenElse</code> node:</b></p> <pre>&lt;@ let result = commit "User" System.DateTime.Now match result with   true -&gt; "true"   false -&gt; "false" @&gt;</pre> <p>Output from fsi – the <code>IfThenElse</code> node is underlined:</p> <pre>Let (result,     Call (None, Boolean commit(System.String, System.DateTime),           [Value ("User"), PropertyGet (None, System.DateTime Now, [])]),     <u>IfThenElse (result, Value ("true"), Value ("false"))</u>)</pre> <p>Figure 4.7 illustrates the AST of the <code>IfThenElse</code> node:</p>  <pre>graph TD     A[IfThenElse] --- B[result]     A --- C[Value("true")]     A --- D[Value("false")]</pre> <p><b>Figure 4.7 – the <code>IfThenElse</code> node</b></p> <p>Referring to Figure 4.7:</p> <ol style="list-style-type: none"> <li>1. The leftmost child node is the <i>guard</i> condition and is of type <code>Quotations.Expr</code>.</li> <li>2. The central child node is evaluated if the guard condition evaluates to true. This node is of type <code>Quotations.Expr</code>.</li> <li>3. The rightmost child node is evaluated if the guard condition is false. This node is of type <code>Quotations.Expr</code>.</li> </ol>
---	---

From this discussion, it follows that a function may be called within a `Let` or `Sequential` or `IfThenElse` node – i.e. a “parent node”. Clearly our weaver must therefore be able to gather information about the parent node such that:

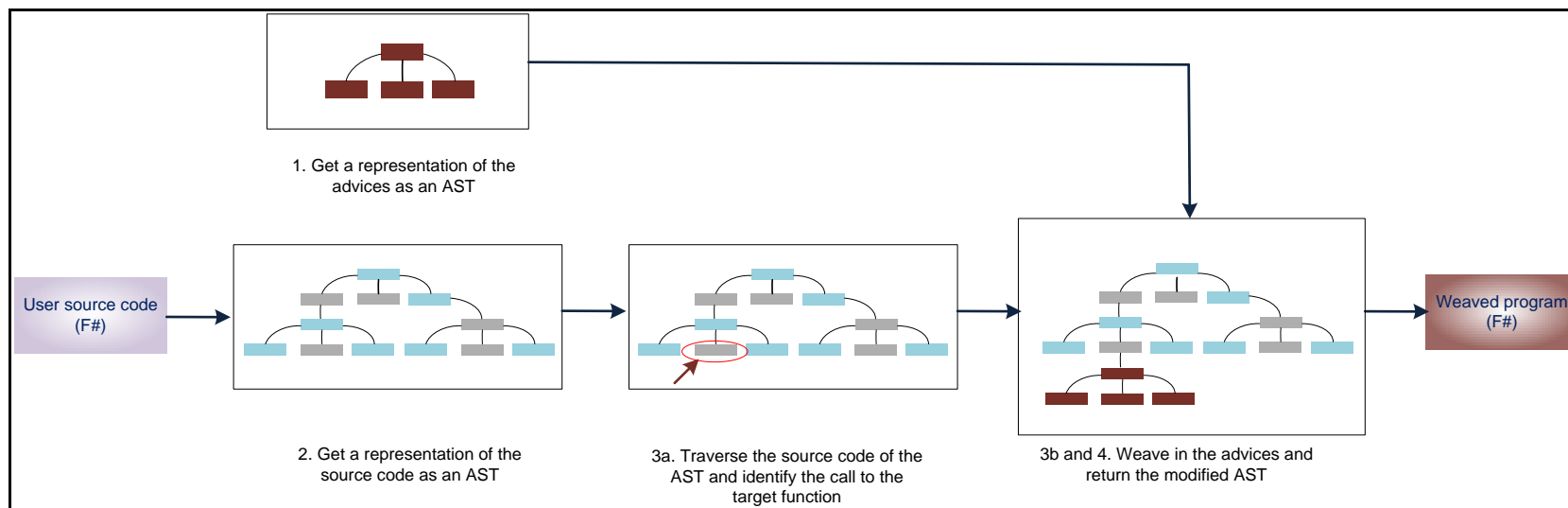
1. The weaver can advise the target function within the parent node.
2. The weaver can then recreate the parent node and insert back into its original location.

#### 4.3.4 Development of code quotation-based weaver

This section presents our weaver built using code quotations. The weaver carries out the following steps, which is illustrated in Figure 4.8 and Figure 4.9 (on the following page):

1. Get an internal representation of the advices such that they can easily be added to the AST data structure.
2. Get an internal representation of the source code as an AST.
3. Traverse the source code AST:
  - a. Identify the node(s) which represent the target function.
  - b. Weave the advices, i.e. replace the node identified in step 2(a) with the structure created in step 1 – the weaver must ensure that the right parent node is created.
4. Return the modified AST.

**Figure 4.8 – Processing steps for the weaver**



**Figure 4.9 – Schematic representation of the weaving process**

Our weaver is implemented via the “weave” function and has the following function signature:

```
weave: aspectSpecification -> Expr -> Expr
```

**Listing 4.21 – Function signature for the weave function**

The weaver parameters are:

1. An object of type `aspectSpecification` which represents an aspect – we cover this type in more details in Section 4.4.2.
2. A `Quotations.Expr` object representing the source program.

The weaver returns another expression which is the AST of the advised function. It is left to the caller code to decide whether to compile the AST into a function for immediate execution, or, *walk* the returned AST to carry out further custom actions [15].

The weaver code is shown below – this is a fairly simple function where most of the functionality to traverse the AST and weave advices is delegated to the `quotationAnalyser` function:

```
/// Weaves aspects defined by the aspectSpecification object into the source AST.
/// Returns an AST containing the weaved program.
let weave ( aspect : aspectSpecification ) ( sourceExpr : Expr ) =

    // Scans the input AST to check if we can handle the parent node of the target
function
    // throws an exception if we cannot handle the parent type
    scanAST sourceExpr aspect.TargetFunctionName |> ignore

    // Retrieve a quotation analyser and execute it
    let qa = quotationAnalyser (sourceExpr) (aspect)
    qa
```

**Listing 4.22 – The structure of the weave function**

The full listing of the `quotationAnalyser` function is available in the CD attached, and via the online repository. Section 4.5 presents the Visual Studio solution for the framework and provides details on accessing the online repository. The `quotationAnalyser` carries out the following tasks:

1. Parse the `aspectSpecification` object to retrieve information about the target function and the advice.
2. Traverse the expression tree recursively.
3. Identify one of the parent nodes discussed in Section 4.3.3.
4. Check the parent node to see if there is a call to the target function.
5. If there is a match, call supporting functions to handle the insertion of the advice.

6. Recreate the parent node and return the advised parent node.

Currently, the framework has been tested against Call, IfThenElse and Let parent nodes.

Listing 4.23 shows a section of the `quotationAnalyser` function where we match a function call whose parent node is a `let` expression, i.e. a function of the general form `let identifier = f()`:

```
1 match expr with
2 | Let(var, expr, continuation) ->
3     match checkMethBody expr targetFunctionName with
4     | true ->
5         // found a match
6         let advised_continuation = qa continuation
7
8         // inject the advices
9         inject aspect.Advices.Application (beforeMethodInfoList,
10 afterMethodInfoList) advised_continuation expr var acceptedContextNodes.LetExpr
11
12     | false -> // continue with the evaluation
13         Expr.Let(var, expr, qa continuation)
```

**Listing 4.23– Matching a target function wrapped within a let expression**

In line 3, we check the method body of the `let` expression (c.f. with the representation shown in Table 4.3) to detect if there is a function call to the target function by making a call to the `checkMethBody` function, which has the following signature:

```
checkMethBody: Expr -> string -> bool
```

**Listing 4.24 - Function signature for the `checkMethBody` function**

The `checkMethBody` function takes an expression and a string which represents the name of the target function. The function returns a `Boolean` indicating a match (or not).

On matching the target function, the weaver creates an expression representing the advised function. This is achieved by a call to the `inject` function. The function has the format and signature shown in the Listing below:

```
inject adviceType (beforeAdviceList, afterAdviceList) continuationExpr targetExpr
var contextNode

inject: apply -> MethodInfo list * MethodInfo list -> Expr -> Expr -> Var ->
acceptedContextNodes -> Expr
```

**Listing 4.25 – Function declaration and signature of the `inject` function**

The rather large method signature contains the information required to construct the expression for the advised function, namely:

1. The `adviceType` variable (of type `apply`) is a discriminated union which specifies whether the advice is to be applied, before, after or around the target function. The implementation of the discriminated union is as follows:

```
type apply =  
  | After_function_call  
  | Before_function_call  
  | Around_function_call
```

**Listing 4.26 - Implementation of the `apply` discriminated union**

2. The second parameter is an F# tuple [96] containing two elements which are lists of `MethodInfo` objects. This tuple represents the list of advices to inject before or after the target function. This allows us to insert different advices before and after the target function. The advices are passed in as `MethodInfo` objects, which are used within the .Net Framework to represent information on a method, such as its name, its return parameter and input parameters [62].
3. `ContinuationExpr` represents the expression which is to be added after the advised function, for example this could contain the expression required to reconstitute the parent node.
4. `targetExpr` represents the expression of the original target function.
5. `var` is an object of type `Var` [97,98] and can be used to represent the variable used within a `let` parent node.
6. `contextNode` is a discriminated union of type `acceptedContextNodes` which is used internally to create an appropriate parent node. The discriminated union is shown below:

```
type acceptedContextNodes =  
  | LetExpr  
  | CallExpr
```

**Listing 4.27 - Implementation of the `acceptedContextNodes` discriminated union**

We can use these to advise other type of parent node – for example, without any changes, the framework supports weaving an advice within an `IfThenElse` statement.

The `inject` function uses other helper functions to create the appropriate expression tree. For example, for example, the helper function below shows how we traverse the list of before advices

(which recall is a list of `MethodInfo` objects) and create *sequential* calls to the advices. Finally, we insert the call to the target function and return the expression tree.

```
1 let mergeNodesForCallExprBefore (adviceList : MethodInfo list) targetExpr =
2     let rec constructSequence advList =
3         match advList with
4         | head :: tail -> Expr.Sequential(Expr.Call(head, []),
5     constructSequence tail)
6         | [] -> targetExpr
7     constructSequence adviceList
```

**Listing 4.28 – Creating the before advice**

The function in Listing 4.28 performs the following steps:

1. Traverse the list of advices.
2. Create, for each advice create a `Sequential` node<sup>4</sup>, where the left expression is a call to the advice, and the right expression is either:
  - a. Another advice – if we have not reached the end of the list of advices.

**or**

  - b. The target expression – if we have reached the end of the list of advices.

### 4.3.5 Before advice constraints

This section discusses some of the constraints on advices which arise due to the nature of functional languages. Recall from Section 2.2 that in functional languages all functions must return a value. In F# this means that within an F# computation the result of the last expression is also the returned value. For example, in Listing 4.29, the function returns the string “returned value”, hence F# *infers* that the return type is string:

```
Let f() = printfn "returned value"
```

**Listing 4.29 – F# infers a string return type**

However, changing Listing 4.29 to the one shown in Listing 4.30 where we add the Boolean `true` just before the return type causes the compiler warning shown in Figure 4.10, below:

```
1 let f() = true
2     "returned value"
```

**Listing 4.30 – F# infers a string return type but with a warning**

<sup>4</sup> c.f. Table 4.2 for an explanation of the left and right nodes



```
warning FS0020: This expression should have type 'unit', but has type 'bool'. Use 'ignore' to discard the result of the expression, or 'let' to bind the result to a name.
```

**Figure 4.10– Warning displayed when compiling the function shown in Listing 4.30**

The warning indicates that F# is expecting a function which returns `unit` on line 1 of Listing 4.30. However we have inserted an expression which returns the Boolean `true`. Intuitively, this warning makes sense, as we have lost information (the Boolean value) in our computation. It is therefore a requirement that our before advices should return `unit`.

#### 4.3.6 After advice constraints

After advices are those which run after a target function executes. This provides the possibility to *intercept* any intermediate result for custom processing.

To illustrate, consider the function below, which returns the result of another function `hasAccess`. `hasAccess` returns `true` or `false`, depending on whether the user can be granted access. The signature of the function is `string -> bool`.

```
let checkAccess (user: string) = hasAccess user
```

**Listing 4.31 – Function to check a user’s access**

It is possible to insert after advices to record the fact that this user was granted (or refused) access. Assume that the after advice is called `recordAccessCheck` and accepts the user name and the result of the check. Logically this is equivalent to re-engineering the code in Listing 4.31 as follows in Listing 4.32:

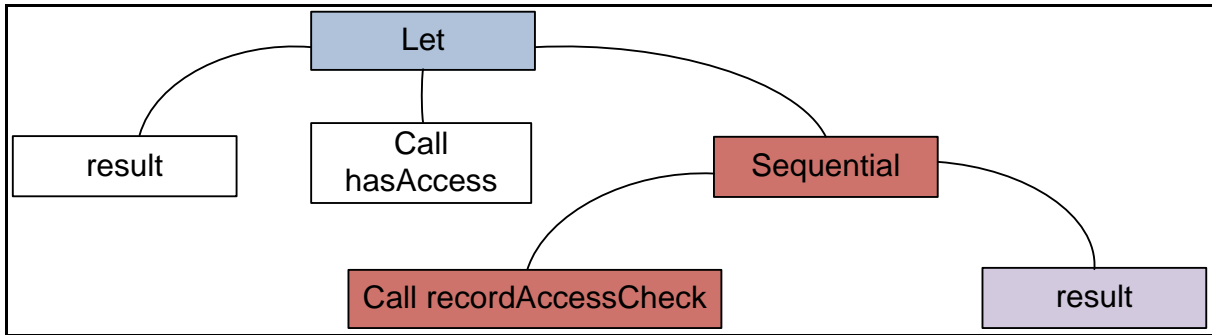
```
1 let checkAccess (user: string) = let result = hasAccess user
2                               recordAccessCheck user result
3                               result
```

**Listing 4.32 – Manually advising a function with after advices**

The following steps were carried out to manually advise the function:

1. Change the call to the `hasAccess` function such that its result is bound to a variable: `result` – on line 1.
2. Call the advice and pass in the relevant arguments – on line 2.
3. On line 3, we return `result`, to preserve the function signature of the `checkAccess` function, i.e. `string -> bool`.

Steps 1 to 3 construct a Let expression which has the format shown in Figure 4.11 (we have omitted the leave nodes of the Call nodes for brevity):



**Figure 4.11 – AST when capturing the intermediate result of a function call**

In our weaver we therefore need to fit a Let expression to capture the intermediate result, as illustrated by the top node in blue in Figure 4.11. We then process the result (maroon nodes in Figure 4.11), and finally return a *result of the same type* as the result node (rightmost purple node in Figure 4.11).

The function, below, shown in Listing 4.33 implements this AST transformation and is called by the inject method:

```

1 let constructAfterAdvicesForCallExpr (adviceList : MethodInfo list) var =
2   let rec constructSequence advList =
3     match advList with
4     | head :: tail ->
5       Expr.Sequential(Expr.Call(head, [Expr.Var(var)]), constructSequence tail)
6     | [] -> Expr.Var(var)
  
```

**Listing 4.33 – Injecting after advices, and capturing the intermediate result**

The function shown in Listing 4.33 has the same format as the one shown in Listing 4.28: in lines 2 to 6 we traverse the list of advices and create sequential calls to the advices. The input to the advices is the result of the function call – as underlined in Listing 4.33. In Section 5, we present a more complex example where we capture the intermediate values of an execution in an after advice.

This design also leads to some constraints on the after advices:

1. After advices must also return unit, so that no warnings are generated when calling the after advices and returning the intermediate result from the computation.
2. As a consequence of point 1, above, an after advice can therefore only *consume* an input value. Within our weaver, our advices parameters are not allowed to be of a generic type - i.e. they must be concrete types (e.g. primitives such as strings or more complex objects). This is to ensure that there is no input mismatch when passing in the intermediate result to the advice. We therefore apply some runtime type analysis to verify that the return type of the target function matches the input type of the advice.

### 4.3.7 Around advice constraints

The around advices are regular before and after advices, and hence have the same constraints:

1. The before and after advice must return a unit.
2. Similar to after advices, the around advice can have either unit or a non-generic type as an input parameter.

### 4.3.8 Summary on the code quotation-based weaver

This section covered our weaver built using metaprogramming technologies. We explain how such a weaver can transform the AST of a source program into another AST representing the advised program. We began by providing a background on metaprogramming and presented the technologies currently available in the .Net Framework. We explained that the preferred technology was the use of *code quotations* which allows the retrieval of the AST with minimal effort – indeed the AST can be retrieved through language keywords.

Our discussion also introduced the rationale and the requirement to decorate the top level function definitions in the source program with the [`<ReflectedDefinition>`] attribute. Contrasting the requirement to decorate top level function with this attribute with the obliviousness requirements for AOP (Section 2.1), it can be argued that this approach is minimally intrusive and a weaver based on metaprogramming constructs is much less intrusive than a monad-based weaver.

We highlighted how the parent node affects our weaver design – as we need to recreate the original context of the function call after injecting the advices. Our weaver based on code quotations allows the intermediate result when executing the function call to be passed on to the after advices.

## 4.4 Pointcuts and aspects for the quotation-based weaver

Our `weave` function described in Section 4.3 depends on an object of type `aspectSpecification`. This type encapsulates the pointcut and advice information. This section covers the design of the pointcut and advice constructs.

### 4.4.1 Pointcut design

A pointcut is a predicate used to specify the join points over which to apply the advices. From our discussion in Section 2.1.1, the join points under consideration in this project are function calls only. Our pointcut structure is required to provide the following functionalities:

1. Allow the user to create named or anonymous pointcuts [16].
2. The user must specify the name of the target function. It is not possible to overload a function in F#, therefore the possibility of weaving the wrong advice is minimal.

A suitable data structure to hold this information is an F# class [15,23,33,36], where the constructor enforces the constraints listed above (Listing 4.34):

```
type pointcutSpecs(pointcutName : string, targetFunctionName : string) =  
  
    member this.PointcutType =  
        if pointcutName = "" then  
            Anonymous_pointcut  
        else  
            Named_pointcut  
  
    member this.PointcutName = pointcutName  
  
    member this.TargetFunctionName = targetFunctionName
```

**Listing 4.34 – the pointcutSpecs class**

It is sufficient to pass in an empty string "" to the constructor of `pointcutSpecs` to set its internal state to an `Anonymous_pointcut`.

#### 4.4.2 Aspect design

Within AspectJ, aspects are Java constructs which encapsulates a crosscutting concern [16] [29]. Within this project, aspects are data structures which have a similar role. Within our framework, we attempt to provide a function to simplify the process of populating this data structure. Different strategies were investigated in order to craft a simple interface over the data structure:

1. Develop a fluent interface [99] using F# constructs.
2. Develop a custom language with its own syntax – effectively an external *domain specific language* [99].

It was judged that proposal (2) was quite onerous, especially in light of the fact that we would need to define our own grammar and implement a parser for the custom language. Tools such as *Fslex* and *Fsyacc* [36,66] can be used to develop our *tokeniser* and *parser* [36], respectively. Another interesting approach to parsing is *FParse*, a *monadic parser combinator* [100]. There is a non-negligible learning curve incurred to use these.

As such, we follow proposal (1), and suggest the function `createAspect`, below, to populate the aspect data structure (Listing 4.35), below:

```

let createAspect ( pointcutInformation : pointcutSpecs) application (beforeAdvices,
afterAdvices)=
    let spec = {   Name = pointcutInformation.PointcutName
                  ; Application = application
                  ; TargetFunctionName = pointcutInformation.TargetFunctionName
                  ; PointcutType = pointcutInformation.PointcutType
                  ; AfterAdvices = afterAdvices
                  ; BeforeAdvices = beforeAdvices
                  }
    Spec

```

**Listing 4.35 – Function to populate an aspectSpecification type**

An appropriate data structure to hold the information related to the aspect is an F# record [36,101] (Listing 4.36):

```

/// Supporting data structure to hold the aspect
type aspectSpecification = {
    Name : string
    Application : apply
    TargetFunctionName : string
    PointcutType : pointcut_type
    Advices : AdvicesInformation
    BeforeAdvices : AdvicesInformation
}

```

**Listing 4.36 – The aspectSpecification type**

The record values model the following information:

3. *Name*: The name of the pointcut.
4. *Application*: whether the advice is to be applied before, after or around the target function.
5. *TargetFunctionName*: the name of the function to advise.
6. *PointcutType*: indicates whether the pointcut is anonymous or named.
7. *Advices*: an object of type `AdvicesInformation`, which encapsulates the advices to weave after the target function.
8. *BeforeAdvices* : an object of type `AdvicesInformation` which encapsulates the advices to weave before the target function.

The `AdvicesInformation` type is shown below:

```

type AdvicesInformation(funcLibrary, adviceNames:string list, application: apply) =
    member this.FuncLibrary = funcLibrary
    member this.AdviceNames = adviceNames
    member this.Application = application

```

**Listing 4.37 – The advicesInformation type**

The advices are *static methods* on an object (the `funcLibrary` parameter) and the names of the member which are to be weaved are passed in as a list of string (the `adviceNames` parameter). The `application` parameter specifies whether the advices are to be weaved before, after or around the target function.

## 4.5 The AspectF Framework – a Visual Studio Solution

The framework developed in Section 4.3 and 4.4 is contained in a Visual Studio solution called *AspectF*. The solution is available in the CD attached in Appendix C and via the online repository on Atlassian Bitbucket (<https://bitbucket.org/>). The username and password are as follows:

1. Username `Nitesh_bbk`
2. Password: `aspectffinal`

The BitBucket documentation provides further information on how to import the source code to a local machine [102].

The AspectF solution is organised in the following projects [103]:

- *AspectF.Shared* – which is an assembly [104] containing some shared types which do not have any business logic.
- *AspectF.Pointcuts* – which is an assembly containing the pointcuts, advices and the aspect types.
- *AspectF.Weaver* – which is an assembly containing the weaver.
- *Tests* – contains the xUnit [44] tests. The assembly generated by this project (Tests.dll) can be opened in the xUnit Test Runner.

Please also note the following project:

- *AspectF.Monads* – contains the source code for the before, after and around monad-based weaver.

The “Readme.txt” file contains system requirements and instruction on running the unit tests.

To use the framework, client code must include the “AspectF.Shared”, “AspectF.Pointcuts” and “AspectF.Weaver” assemblies in their project. A sample usage of weaving a target program is shown in Chapter 5.

## 4.6 The F# compiler as a weaver

During the design of a weaver based on metaprogramming technologies, we investigated and analysed the possibility of building a compiler extension to the F# compiler (`fsc.exe`). The design was to extend the F# compiler to add another compiler switch [105]. With this compiler switch, the compiler would do some pre-processing on the source code file and add advices. The output of the pre-processing step would be new source code containing the advices. The regular F# compilation process would then continue and generate an assembly or executable file. Similar approaches to weaving currently exist in .Net, such as the *Eos* compiler [106].

The F# compiler source code is freely available for download [66][107]. Extending the F# compiler was not one of the main weaving strategies proposed.

This section details some of the more interesting finds when analysing the F# compiler source code. Please note that there exists very little information on the structure of the F# compiler. However, some of the tools used are standard (e.g. `Fslex` [36,66] and `Fsyacc` [36,66]). The information in this section is relevant for *changeset* number 64420 [108].

### 4.6.1 Building the compiler

The `readme.html` file available as part of the F# compiler download provide detailed information on building the F# compiler. The F# compiler is itself written in F#. To compile this F# code into another compiler, a “proto” version of the F# compiler is built from the current version of the F# compiler. The “proto” compiler is then used to build the F# assemblies and executable files (such as `FSharp.Core.dll`, `FSharp.Compiler.dll`, `fsc.exe`) using the new source code.

### 4.6.2 Attaching a debugger to step through the compile process

[109] highlights the process used to attach a debugger to step through the compile process in Visual Studio. When stepping through the F# code of the compiler, the entry point is the `main(argv)` function located in the `fscmain.fs` file.

The compiler is fairly complex, but the target source code (i.e. the code which we want to compile) is tokenised by a call to `Fslex` and the sequence of tokens is processed by the compiler. A parser built using `Fsyacc` parses the sequence of tokens to check that the source code is valid F#. The *configuration* files for `Fslex` and `Fsyacc` – i.e. the definitions of what constitutes valid F# constructs and terms [36] are located in the directories `\src\fsharp\lex.fsl` and `\src\fsharp\pars.fsy`, respectively.

We can note the following functions and files:

1. The function `ParseOneInputLexBuf` handles the source code tokens returned from `Fslex`.
2. The file `prim-types.fs` contained all the core types of F#.
3. The file type definition `TcConfigBuilder` located in the `build.fs` file contains all compiler switches – including those not “officially” documented.

#### 4.6.3 Suggested strategy for building a compiler extension

The proposed strategy is to make changes to the F# compiler in order to define a new compiler switch, for example `--weave`, followed by the file name containing the aspects. When the compiling process starts, the compiler would inspect the aspects file and the source code file and merge the two codes as defined by the pointcuts. The regular compilation would then be executed on the resultant advised source code.

This approach is feasible as the compiler itself is written in F# which is a high level, declarative language, hence is probably more comprehensive than a compiler written in, say, C. Generally, this weaving strategy would provide a very high level of granularity during the weaving process.

Major drawbacks on this approach include the fact that the F# compiler remains a complex system, and due to time constraints, this approach was not investigated thoroughly. Manipulating the F# compiler is a static weaving strategy, which may not be the most suitable for all uses – i.e. dynamic weaving might be preferable in some use cases. In addition, this approach would require the use a specific compiler, maintained separately from the release cycle of the official F# compiler. As such, there would be a maintenance overhead involved in regression testing our compiler extension with every new releases of the official F# compiler.

#### 4.7 Summary

This chapter detailed the technical implementation of our framework. We focused on the weaver implementation as it is the central component of the framework (c.f. Figure 3.1). We showed the implementation two types of weavers:

- A monad-based weaver implemented using computation expressions (Section 4.2).
- A code-quotation based weaver (Section 4.3).

Monads makes weaving a trivial activity but they were judged to lack granularity and required too many changes to client code – in effect our implementation of the monad-based weaver breaks the concept of obliviousness. We therefore consider a weaver built using metaprogramming technologies, namely through the use of code quotations. Code quotations provide us with a



representation of source program as an abstract syntax tree (AST). Our code quotation-based weaver consumes the AST of the source program, together with an `aspectSpecification` object which encapsulates the pointcut and the advice, and returns an AST representing the advised program. The advices can be made to execute, before, after or around the target function. We noted some constraints on the advice constructs: before and after advices must return `unit` and after advices are allowed one input parameter, which we can use to consume the result of the target function. The same constraints apply to around advices.

Our research into a weaver leads us to consider developing a new compiler extension which would weave the aspects into a source code file (Section 4.6). This strategy provides a lot of flexibility at the expense of added complexity and maintenance.

## 5. Usage and instrumentation – advising a recursive function to estimate $\pi$

This section shows a usage of the weaver built using code quotations and performance results. Our running example is a function which estimates  $\pi$ .

### 5.1 Implementing Machin's formula in F#

Machin's formula, shown below, is one algorithm used to estimate  $\pi$  [110]:

$$\frac{1}{4}\pi = 4 \arctan \frac{1}{5} - \arctan \frac{1}{239}$$

**Formula 5.1 – Machin's formula**

Where:

$$\arctan x = \sum_{k=0}^{\infty} \frac{(-1)^k x^{2k+1}}{2k+1}$$

**Formula 5.2 – An expansion of  $\arctan x$**

A trivial implementation in F# is shown in Listing 5.1 below, which accepts an upper limit  $k$  for the number of iterations to use when calculating the term  $\arctan x$ . We shall use this implementation to get some performance metrics.

```
let calcTerm x k =
    System.Math.Pow(-1.0, k) * System.Math.Pow(x, 2.0 * k + 1.0) / (2.0 * k + 1.0)

let (|Reachlim|_|) x lim =
    if x = lim then
        Some(x)
    else
        None

let arctan x k =
    let rec calc pow =
        match pow with
        | Reachlim k r -> calcTerm 0.0 pow
        | _ -> (calcTerm x pow) + calc (pow + 1.0)
    calc 0.0

let term1 = arctan (1 / 5.0) 10000.0
let term2 = arctan (1.0 / 239.0) 10000.0
let piEstimate = 4.0 * ( (4.0 * term1) - term2 )
```

**Listing 5.1– Implementation of the Machin formula in F#**

In Listing 5.1 the `calcTerm` function evaluates one term of the summation shown in Formula 5.2. The `Reachlim` active pattern returns an F# option [111] which indicates whether the evaluation has reached the allowed limit. The `arctan` function computes  $\arctan x$  up to a certain number of terms. We make a call to the recursive `arctan` F# function twice, for the two terms in Formula 5.1. Finally we estimate  $\pi$  and the result is bound to the `piEstimate` identifier.

We can now use the code quotation weaver (detailed in Section 4.3) to inject advices. Let's assume that the advices must capture the intermediate results of the `calcTerm` function for further processing. The intermediate results are of type `float` [112], hence the advices have the following function signatures, which matches the requirement for after advices (Section 4.3.6):

```
float -> unit
```

For illustrative purposes, let us assume that the advice prints out to screen, and an advice has the following format:

```
let printparams f = printfn "the intermediate value: %f" (f)
```

**Listing 5.2 – Example after advice**

Our objective is therefore to change the implementation of the `arctan` function from the one shown in Listing 5.1 to the one shown below – where the changes are in red:

```
let arctan x lim =
  let rec calc pow =
    match pow with
    | Reachlim lim r -> let res = calcTerm 0.0 pow // capture the intermediate result
                       printparams res // prints out the result
                       res // return the result
    | _ -> let res = (calcTerm x pow) + calc ( pow + 1.0 ) // capture the intermediate result
          printparams res // prints out the result
          res // return the result
  calc 0.0
```

**Listing 5.3 – An advised implementation of Machin's Formula**

We construct the following advice/pointcut/aspect triplet to insert our *after* advice after the `calcTerm` function:

```
let afterAdvices = new AdvicesInformation(new al(), ["printparams"],
apply.After_function_call)

let pointcutSpecs = (new pointcutSpecs("", "calcTerm"))

let aspect = createAspect pointcutSpecs apply.After_function_call (afterAdvices,
afterAdvices)
```

**Listing 5.4 – Creating our advice/pointcut and aspect triplet**

We then *decorate* the `arctan` function with the `ReflectedDefinition` attribute and generate its expression using quotation markers `<@ @>` - a requirement described in Section 4.3.2. The aspect object and the expression are then transformed by the weaver:

```
let transformedAST = weave (aspect) (sourceExpr)
```

Appendix A shows the original unadvised tree, and the advised tree – i.e. the result bound to the `transformedAST` identifier

## 5.2 Estimating $\pi$ – Comparing speed of execution

Using the `CompiledUntyped` method of the `Microsoft.FSharp.Linq` namespace of the F# PowerPack [66], it is possible to compile the expression tree of the advised function and execute it. This allows us to instrument the speed of execution of the advised function.

Our instrumentation relies on the `System.Diagnostics.Stopwatch` [113] object from the .Net Framework. The `stopWatch` object is used to evaluate the time elapsed across different points of the application. We measure the time elapsed across the following sub-processes:

1. Time spent executing the non-advised code.
2. Time spent executing “manually advised” code – i.e. where we manually place call to the advice.
3. Time spent weaving the code.
4. Time spent compiling the advised expression tree and executing the returned function.

Our test setup is as follows:

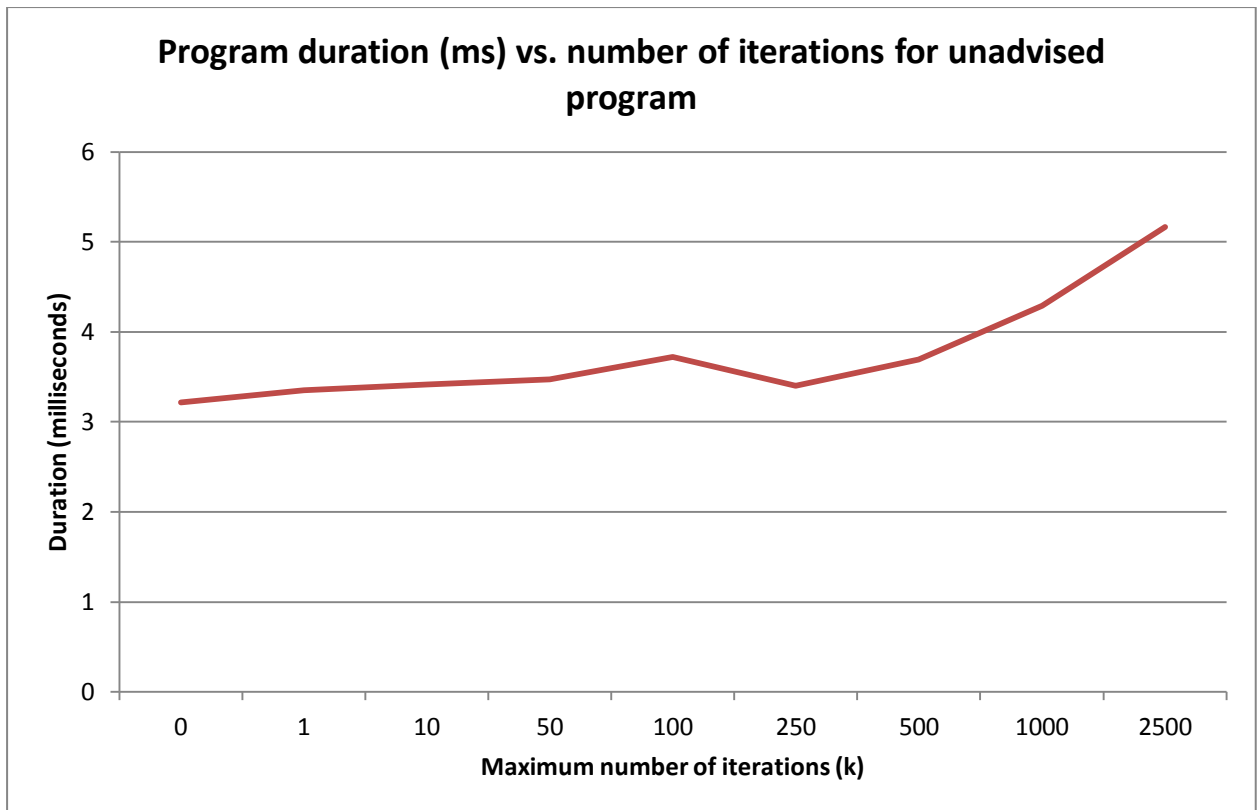
- The timing operation is done using the default *release* build configuration in Visual Studio 2010. This has the minor effects of turning on the optimisation switch on the F# compiler. We are using the default implementation of the release build configuration – i.e. the one included with Visual Studio 2010 Ultimate [114].
- The application is compiled using the .Net 4.0 libraries and the default F# 2.0 included with Visual Studio 2010.
- The hardware used is a standard desktop computer with an Intel i3 Core CPU (quad core). However we do not take advantage of any parallelism.
- The time is initially measured in *ticks*, where the ratio of ticks per second is dependent on whether the system has a *high resolution system counter*.
  - a. If the system does have such a timer, then the number of ticks per second is called the *frequency* and changes every time the machine is rebooted [115]. The machine used within the test runs has a high resolution system counter.
  - b. Otherwise, the value of a tick defaults to 1 tick for every 100 nanoseconds [116].

### 5.2.1 Instrumentation of the non-advised code

We construct an F# program which evaluates unadvised code – i. e. the program shown in Listing 5.1. We take 5 timing samples for each value of  $k$  and take an average the time taken to estimate  $\pi$  with the given value of  $k$ . The results are shown in Table 5.1 and are plotted in Figure 5.1:

**Table 5.1 – Time to execute an unadvised program for varying number of iterations**

Maximum numbers of iterations (k)	Duration (seconds)
0	0.003213636
1	0.003350539
10	0.003414003
50	0.00347489
100	0.003718269
250	0.003403091
500	0.003696043
1000	0.00429219
2500	0.005168637



**Figure 5.1 – Plot of Table 5.1**

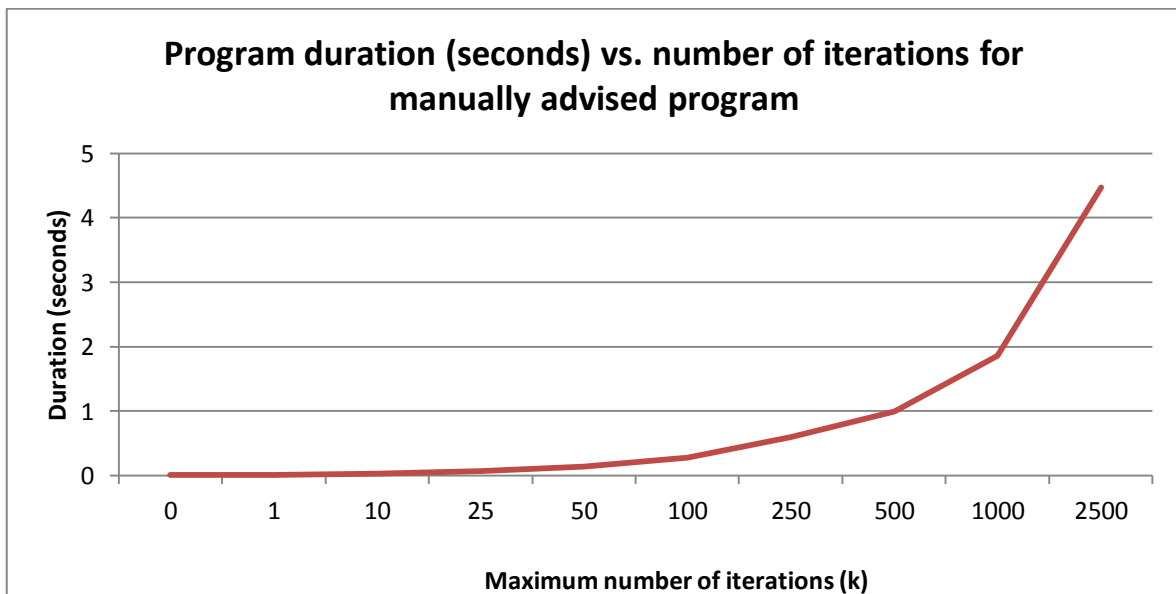
We note that the curve shown in Figure 5.1 is upward sloping. This makes sense intuitively, since the larger the number of iterations  $k$ , the more CPU operations are required and hence the longer will the operation take. We can note that the curve descends slightly between  $k = 100$  and  $k = 250$  before rising again, perhaps due to some optimisation carried out transparently by the compiler. To investigate this further a greater number of samples could have been taken.

### 5.2.2 Instrumentation of manually advised code

We construct an F# program where we manually insert the advices shown – c.f. Listing 5.3. The results of executing the program for different values of iteration  $k$ , are shown in Table 5.2 below:

**Table 5.2 – Time to execute a manually advised program for varying number of iterations**

Maximum numbers of iterations (k)	Duration (seconds)
0	0.011418237
1	0.013885502
10	0.02844561
25	0.067860322
50	0.134745453
100	0.274505423
250	0.594982392
500	0.996590817
1000	1.861183008
2500	4.467141598



**Figure 5.2 – Plot of Table 5.2**

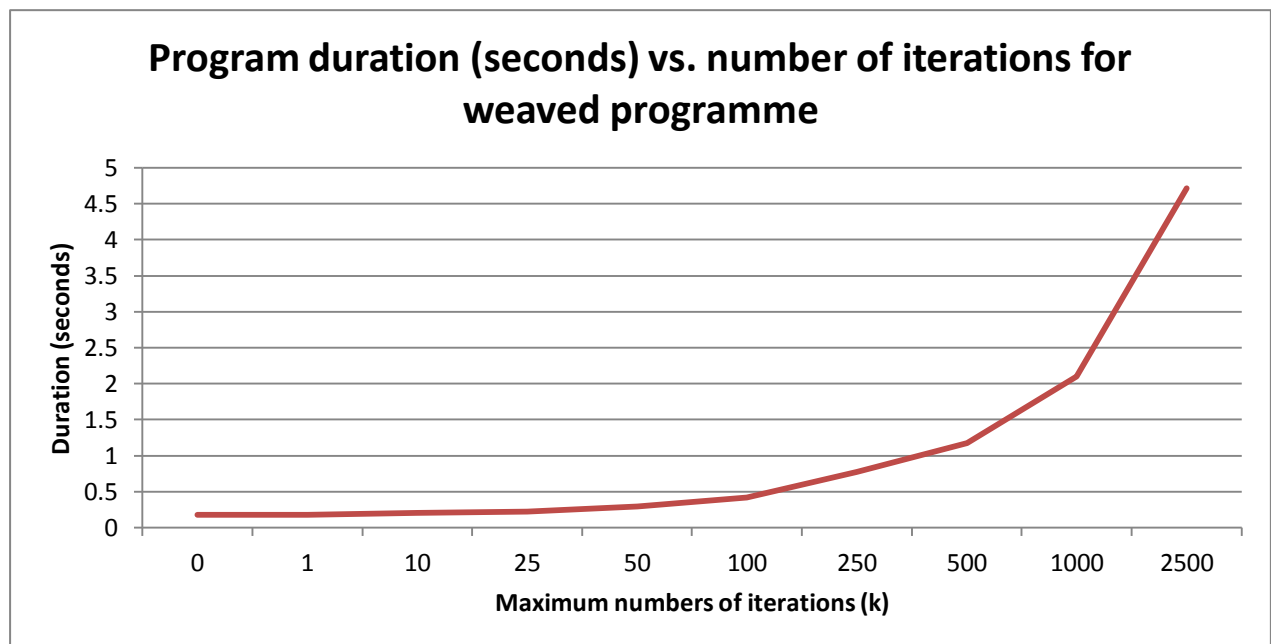
The time to execute the manually advised function also increases as the number of iterations increases. In this case, the execution is not only a CPU operation but also an IO operation as we are printing to screen. The additional CPU operations and IO operation cause a degradation of performance. For the simple case where  $k = 0$ , the performance is degraded by a factor of 255%.

### 5.2.3 Instrumentation of weaved code

We construct an F# program which uses the weaver detailed in Section 4.3 to automatically advise the program showing in Listing 5.1. The results of running the program for different values of  $k$  are summarised in Table 5.3 and the results are plotted in Figure 5.3 below:

**Table 5.3 – Time to execute a weaved program for varying number of iterations**

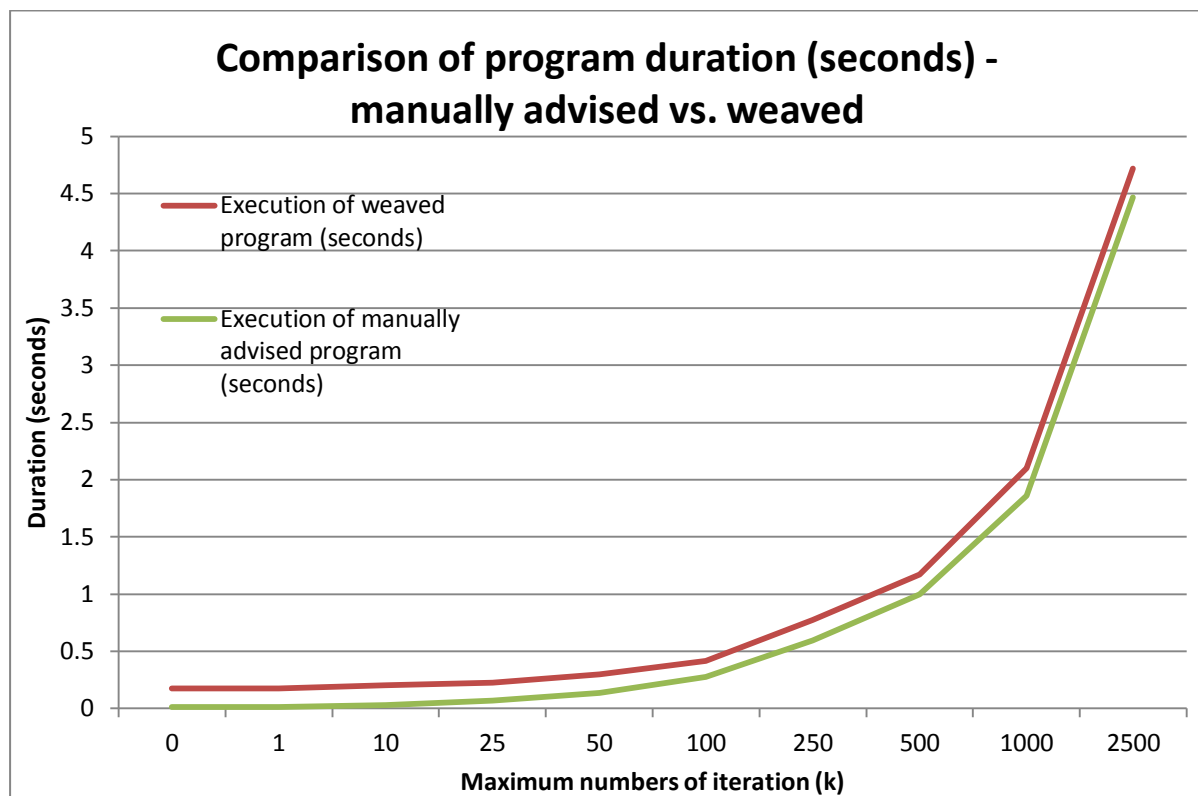
Maximum numbers of iterations (k)	Duration (seconds)
0	0.177468194
1	0.174280733
10	0.205149825
25	0.223888317
50	0.29621107
100	0.414801312
250	0.775149882
500	1.170822077
1000	2.099559668
2500	4.716360577



**Figure 5.3 – Plot of Table 5.3**

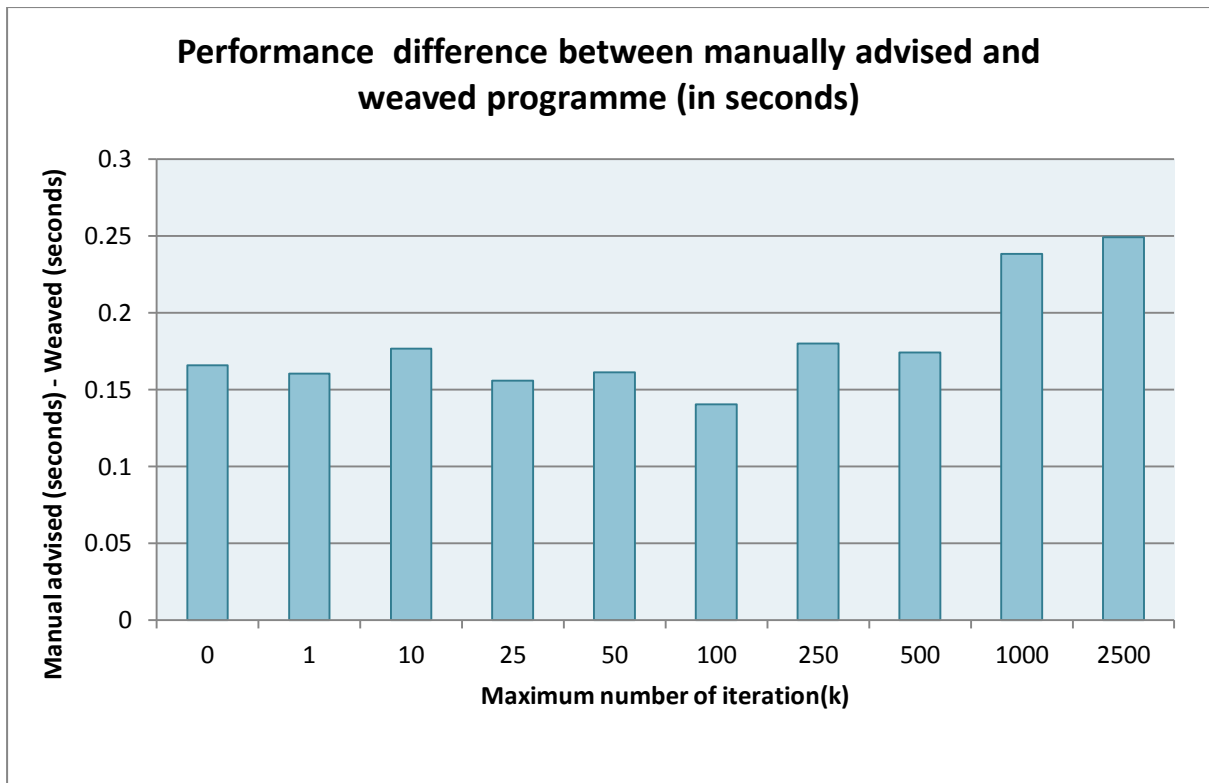


Figure 5.4 superposes Figure 5.2 and 5.3:



**Figure 5.4 – Superposition of manually advised program vs. weaved program**

From Figure 5.4, we can note that for small number of iterations, it appears that there is an almost constant difference between running the manually advised program and the weaved program. This difference is still present for larger values of k. Figure 5.5, below, shows a histogram of the time difference between the manually advised program and the weaved program:



**Figure 5.5 – Difference in execution time**

We note that the time difference between running the manually advised program and the weaved program is around ~0.17 seconds. Notable exceptions include the cases where  $k = 100$  and  $k > 1,000$ . These are possibly due to compiler optimisations. This difference may have been smoothed out by taking a larger number of samples.

When  $k = 0$ , the manually advised program take  $1.14 \times 10^{-2}$  seconds to complete (Table 5.2), whereas the weaved program takes  $1.775 \times 10^{-1}$  seconds to complete (Table 5.3). Generally, the effects of the IO operations (printing to screen) dominates over the effect caused by the CPU bound operations (evaluating the terms of the Machin's formula). This effect of printing a single line to screen (akin to executing the advice once) was estimated to be around 1ms - however this measure was taken a few days after the testing run with a different Stopwatch *frequency* value, hence this number is an approximation.

The timing results of the weaved program for the case where  $k = 0$  indicates that another effect dominates over the IO and the CPU bound operations for small values of  $k$ . This effect can traced to the time taken to generate the AST of the weaved program, compile it and execute it. We can use the data presented in Table 5.4, below, to estimate the time taken by this combined effect. In Table 5.4, we have broken down the timing of the execution into different sub-processes: time take to

weave each term – i.e. `term1` and `term2` in Listing 5.1, and the time taken to compile and execute each term. Figure 5.6 is a plot of the data in Table 5.4

**Table 5.4 – Breakdown of the time to execute the weaved program, for varying number of iterations**

Maximum number of iterations	Average time to weave term 1 (seconds)	Average time to weave term 2 (seconds)	Average time to compile and execute term 1 (seconds)	Average time to compile and execute term 2 (seconds)
<b>0</b>	0.01970853	0.00068619	0.14925207	0.00782141
<b>1</b>	0.01973095	0.00063678	0.14509156	0.00882143
<b>10</b>	0.02158513	0.00068974	0.1611892	0.02168575
<b>25</b>	0.01975305	0.00064502	0.16353188	0.03995837
<b>50</b>	0.01917812	0.00061395	0.20689174	0.06952726
<b>100</b>	0.01891215	0.00058544	0.25661259	0.13869113
<b>250</b>	0.02020881	0.00064013	0.50377266	0.25052828
<b>500</b>	0.01929527	0.0006319	0.68902591	0.461869
<b>1000</b>	0.01914358	0.00061683	1.17893074	0.90086851
<b>2500</b>	0.01996913	0.00072714	2.46331973	2.23234458

From Table 5.4, the total time (`term1` + `term2`) to compile and execute the program for the case where `k = 0` is 0.157 seconds. This effect dominates over the total time taken to weave the advice into the code (0.020 seconds).

Using the complete set of data (shown in Appendix B) we further note that the average time taken to weave the first term is  $1.975 \times 10^{-2}$  seconds, with a standard deviation of  $1.833 \times 10^{-3}$  seconds. The average time taken to weave the second term is  $6.489 \times 10^{-4}$  seconds with a rather large standard deviation of  $1.279 \times 10^{-4}$  seconds.

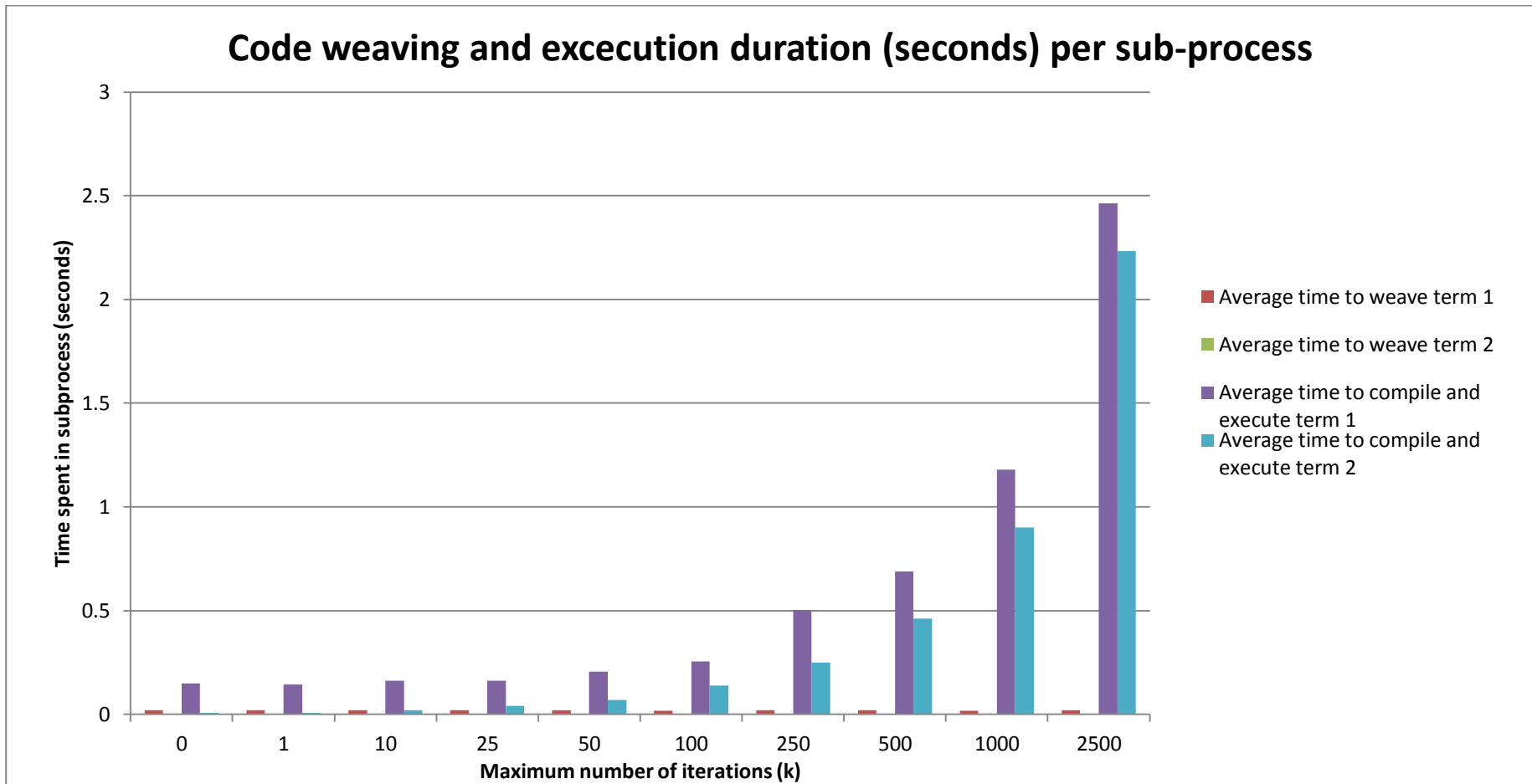


Figure 5.6

### 5.3 Summary

In this Chapter we have presented some performance comparison between an unadvised program, a manually advised program and weaved program. The program used was an implementation of Machin's algorithm which is used to estimate  $\pi$ . In the simple example of a single iteration, our manually advised program degrades the speed of the program by a factor of approximately 2.5, mostly due to the IO operation of printing to screen. However the weaved program causes a speed degradation by a factor of approximately 54 (for a single iteration)!

During our analysis in Section 5.2.3, we note that this degradation is largely caused by the compilation of the quotation expression into a working program. Our original program in Listing 5.1, is CPU bound, but becomes IO bound as our advices are printing to screen on every recursive pass. As a consequence the compilation delay becomes negligible as the number of iteration increases. If our program was CPU bound, this delay would be non-negligible. It is important to note that we are not strictly required to compile the code, and execute it immediately. It is possible as [15] notes, to *walk* the expression tree and interpret each node.

This chapter considers program performance to be a function of execution time only. Another description of performance – which was not considered in these experiments, would be to measure the size of the weaved program on disk and attempt to identify any code bloat the weaver introduces compared to a manually advised program.

## 6. Conclusion

This dissertation provides the background, design, development and demonstration of an AOP framework for F#. We showed how we implemented the components of an AOP framework:

1. Join points are restricted to function calls.
2. Pointcuts are objects which allow the user to specify which function to advise.
3. Advices are functions to execute either before, after or around the target function.
4. Aspects are an encapsulation of advices and pointcuts.

A key component of our framework is the weaver. We considered two different designs for our weaver: one using monads (Section 4.2.3) and other using metaprogramming technologies – namely code quotations (Section 4.2.4). The monad-based weaver is constructed using computation expressions which are unique to F# and provides syntactic support for monad constructs. After experimenting with monads, we decided that the weaver developed using metaprogramming technologies is more appropriate as it allow much more granularity.

After some research into the available metaprogramming techniques in the .Net Framework (Section 4.3.1), we opted to use code quotation, which is a technology specific to F#. Code quotation allows one to retrieve a data structure which represents F# code. This data structure is known as an abstract syntax tree where every node represents a construct in the F# language, such as a function call, an if/else statement. F# also exposes some functionalities to simplify traversing the abstract syntax tree and analyse each node of the tree.

Code quotation leads us to construct a static weaver [31], which inserts the advices at compile time and returns an abstract syntax tree representing the advised program. We noted in Sections 4.3.6 and 4.3.7 that there are constraints on the types of function which we can use as advices.

This project delivers the framework in an F# solution, which we detail in Section 4.5. In Section 5, we demonstrate the use of our framework to advise a CPU bound program. Some measurements we taken to contrast the time taken to execute an unadvised program with an advised program. Our approach to instrumentation was to compile and execute the abstract syntax tree representing the weaved program. We noted that there was a penalty of about 0.157 seconds to compile and execute a program, when the program only did a single iteration. Clearly, compilation and execution is an expensive operation, at least with the libraries used from the F# PowerPack.

We can make the following remarks on our code quotation based weaver:

1. The use of code quotations to generate a weaved program requires a substantial amount of analysis to implement correctly. This is important not only to ensure that the original program continues to function correctly but also to ensure that the weaved AST can be compiled.
2. There is not a lot of documentation available on code quotations, hence a lot of work was focused on experimenting with the available libraries.
3. Some of the libraries used in this project are from the F# PowerPack [66] library. These are experimental libraries, developed by Microsoft but outside of the release cycle of F# or of the .Net Framework.

In addition to developing an AOP framework, this project has required learning F#. Being a functional language, F# allows the development of programs which are free of side effects [33]. Interestingly, AOP opposes this notion as it precisely applies side effects to existing programs. There are, however, many use cases for AOP, such as instrumentation, transaction management and caching. By carefully crafting a weaver which guarantees that the original computations are unaltered, we obtain a clean separation of concerns, and it remains possible to develop side effect free programs, which can then be extended by transparently weaving in advices.

## 7. Future work

We note the following possible future work:

- We have only provided *kinded* pointcuts which allows one to specify the function name and signature to be advised. AspectJ also provides *non-kinded* pointcuts [16] which inject advices based on the current program *flow*, *lexical context* or *execution*. The code quotation-based weaver can be extended to accommodate for non-kinded pointcuts as we can reason about the program (e.g. its flow) as we traverse the abstract syntax tree representing the source program.
- Our join point model only covers function calls. Many more join points exists within F#, such as exception handling, binding to a particular identifier, calling constructors or utilising *sequence expressions* [33]. Further development work would be required to implement these and provide a richer join point model.
- Following on the point above and as described in Section 4.3.4, the framework has been tested against Call, IfThenElse and Let parent nodes. Further development work is required to extend the set of nodes the framework can advise.
- Currently our pointcut structure requires a function name (Section 4.4.1) to identify a target function. The weaver uses this function name to identify which function we should advise. Although function overloading is not possible in F#, a better approach would be to use the signature of the target function as a means of identification.
- We have provided support for named pointcuts, but at the moment these are not used within the framework - further development is required to make greater use of named pointcuts.
- An alternative to our weavers is to extend the F# compiler. Section 4.5 provided an overview of the F# compiler. Extending the compiler provides a lot of flexibility at the expense of added complexity.
- An objective to this project, but not fully met, was to package the framework as a set of libraries for distribution. At the moment, this framework is experimental (but functional).



Additional use cases are required to verify the suitability of the framework in its current state and hence to be more confident in distributing it.

- Performance tuning could be done to improve the performance of the code quotation-based weaver.
- In Chapter 5, we have used the time to compile and execute the weaved code as an indicator of performance. Other metrics, such as the size of the compiled weaved code on disk warrants investigation.
- In Figure 5.1 of Chapter 5, we have noted a dip around the values where  $k = 100$  and  $k = 250$ , further investigation would be required to ascertain the reasons for this dip.
- Despite lacking granularity, the monad-based weaver is a very clean solution and a good solution drawn from functional programming principles. In addition, its performance might be better than our code quotation-based weaver as it does not require any additional compilation stage. Such possibilities warrant further investigation.
- Our weaver is currently a static weaver and works at the compilation stage. *Dynamic weaving* – where code is weaved into a program at runtime can be more suitable in some situations [32]. Our project proposal [8] provides some suggestions on the form dynamic weaving could take. For example, we could inspect the program in its compiled form and inject advices as it is about to be loaded into for execution.

## 8. Bibliography

- [1] G. Kiczales, J. Lamping, and A. Mendhekar, "Aspect-oriented programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.
- [2] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys*, vol. 1241, Dec. 1997, pp. 220-242.
- [3] "F# - Microsoft Research," <http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/>.
- [4] "The AspectJ Project," <http://www.eclipse.org/aspectj/>.
- [5] D. Esposito and A. Saltarello, *Microsoft .Net: Architecting Applications for the Enterprise*, Microsoft Press, 2009.
- [6] "Microsoft Developer Network - Policy Injection Application Block," <Http://msdn.microsoft.com/en-us/library/ff650672.aspx>.
- [7] "Spring.NET - Application Framework," <http://www.springframework.net/>.
- [8] N. Chacowry, "An Aspect Oriented Framework in F #. Project proposal submitted as prerequisite for the MSc in Computer Science, Birkbeck University of London," 2011.
- [9] D. Dantas, D. Walker, and G. Washburn, "AspectML: A polymorphic aspect-oriented functional programming language," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, 2008.
- [10] <http://www.openfoundry.org/of/projects/801>, "AspectFun at the OpenFoundry," <http://www.openfoundry.org/of/projects/801>.
- [11] M. Wang and B.C.D.S. Oliveira, "What does aspect-oriented programming mean for functional programmers?," *Proceedings of the 2009 ACM SIGPLAN workshop on Generic programming - WGP '09*, 2009, p. 37.
- [12] H. Masuhara, H. Tatsuzawa, and A. Yonezawa, "Aspectual Caml: an aspect-oriented functional language," *Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, ACM, 2005, p. 320–330.
- [13] D. Walker, S. Zdancewic, and J. Ligatti, "A Theory of Aspects," *Proceeding ICFP '03 Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, 2003.
- [14] "The F # 2 . 0 Language Specification," *Microsoft Corporation*, 2010.
- [15] C. Smith, *Programming F#*, O'Reilly Media, Inc., Sebastopol CA, 2009.
- [16] R. Laddad, *AspectJ in action*, Manning Publications Co. Greenwich, CT, USA, 2003.
- [17] P. Hudak, "Languages Evolution , and Application of Functional Programming," *Computing*, vol. 21, 1989.

- [18] M. Fogus and C. Houser, *The Joy of Clojure*, Manning Publications Co. Greenwich, CT, USA, 2011.
- [19] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in action*, Manning Publications Co. Greenwich, CT, USA, 20011.
- [20] "Polyglot Programming," <http://polyglotprogramming.com/>.
- [21] "The Scala Programming Language," <http://www.scala-lang.org/>.
- [22] G. Washburn and S. Weirich, "Good advice for type-directed programming aspect-oriented programming and extensible generic functions," *Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming - WGP '06*, 2006, p. 33.
- [23] "Microsoft Developer Network - Classes (F#)," <http://msdn.microsoft.com/en-us/library/dd233205.aspx>.
- [24] "Microsoft Developer Network - Object.GetType Method," <http://msdn.microsoft.com/en-us/library/system.object.gettype.aspx>.
- [25] E.W. Dijkstra, "On the Role of Scientific Thought," *Selected Writings on Computing: A Personal Perspective*, 1982.
- [26] K. Baley and D. Belcham, *Brownfield Application Development in .Net*, Manning Publications Co. Greenwich, CT, USA, 2010.
- [27] M. Marin, A.V. Deursen, and L. Moonen, "A Classification of Crosscutting Concerns," *Proceedings of the 21st IEEE International Conference on Software Maintenance*, 2005.
- [28] R.E. Filman and D.P. Friedman, "Aspect-oriented programming is quantification and obliviousness," *Workshop on Advanced Separation of Concerns*, 2000.
- [29] D.B. Tucker and S. Krishnamurthi, "Pointcuts and advice in higher-order languages," *Proceedings of the 2nd international conference on Aspect-oriented software development - AOSD '03*, 2003, pp. 158-167.
- [30] "Aspect Oriented Programming with Spring," <http://static.springsource.org/spring/docs/2.5.0/reference/aop.html>.
- [31] K. Czarnecki, *Generative Programming - Principles and Techniques of Software Engineering Base on Automated Configuration and Fragment-Based Component Models*, Department of Computer Science and Automation, Technical University of Ilmenau: Dissertation submitted in partial fulfillment of the requirements for the degree of Doktor-Ingenieur, 1998.
- [32] "PostSharp - AOP on .Net - Run Time Weaving," <http://www.sharpcrafters.com/aop.net/runtime-weaving>.
- [33] T. Petricek and J. Skeet, *Real World Functional Programming*, Manning Publications Co. Greenwich, CT, USA, 2010.

- [34] K. Hazzard and J. Bock, *Metaprogramming in .NET*, MEAP Edition Manning Early Access Program, Manning Publications Co. Greenwich, CT, USA, 2011.
- [35] "Microsoft Developer Network - Type Inference (F#)," <http://msdn.microsoft.com/en-us/library/dd233180.aspx>.
- [36] R. Pickering, *Beginning F#*, Apress, 2009.
- [37] "The CTO Corner - The F# Survival Guide ebook," *The CTO Corner*: <http://www.ctocorner.com/fsharp/book/>.
- [38] J. Harrop, *F# for Scientists*, Wiley-Interscience, 2008.
- [39] "Microsoft Developer Network - F# Interactive (fsi.exe) Reference," <http://msdn.microsoft.com/en-us/library/dd233175.aspx>.
- [40] "Microsoft Developer Network - switch (C# Reference)," <http://msdn.microsoft.com/en-us/library/06tc147t.aspx>.
- [41] "Microsoft Visual Studio," <http://www.microsoft.com/visualstudio/en-us>.
- [42] ".NET Downloads, Developer Resources and Case Studies," <http://www.microsoft.com/net>.
- [43] R. Osherove, *The Art of Unit Testing*, Manning Publications Co. Greenwich, CT, USA, 2009.
- [44] "xUnit - Unit testing framework for C# and .Net (a successor to NUnit)," <http://xunit.codeplex.com/>.
- [45] W.D. Meuter, "Monads as a theoretical foundation for AOP," *Technology*, 1997, pp. 1-6.
- [46] P. Wadler, "The essence of functional programming," *Proceedings of the 19th ACM SIGPLAN SIGACT symposium on Principles of programming languages*, ACM Press, 1992, pp. 1-14.
- [47] "Haskell Glossary - Monads," [http://www.haskell.org/haskellwiki/Monad\\_%28sans\\_metaphors%29](http://www.haskell.org/haskellwiki/Monad_%28sans_metaphors%29).
- [48] B. Beckman, "Channel9 : Don't fear the Monad," <http://channel9.msdn.com/Shows/Going+Deep/Brian-Beckman-Dont-fear-the-Monads>.
- [49] S. Jones, "Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell," *Engineering theories of software construction*, 2001, pp. 47-96.
- [50] "Microsoft Developer Network - Symbol and Operator Reference (F#)," <http://msdn.microsoft.com/en-us/library/dd233228.aspx>.
- [51] "Microsoft Developer Network - Discriminated Unions (F#)," <http://msdn.microsoft.com/en-us/library/dd233226.aspx>.

- [52] "Microsoft Developer Network - Lambda Expressions: The fun keyword," <http://msdn.microsoft.com/en-us/library/dd233201.aspx>.
- [53] "Microsoft Developer Network - Expr.Sequential Method (F#)," <http://msdn.microsoft.com/en-us/library/ee353459.aspx>.
- [54] "Meta Programming - Online definition from Cunningham & Cunningham," <Http://c2.com/cgi/wiki?MetaProgramming>.
- [55] J. Baker and W. Hsieh, "Runtime aspect weaving through metaprogramming," *Proceedings of the 1st international conference on Aspect-oriented software development*, ACM, 2002, p. 86–95.
- [56] É. Tanter, R. Toledo, G. Pothier, and J. Noyé, "Flexible metaprogramming and AOP in Java," *Science of Computer Programming*, vol. 72, Jun. 2008, pp. 22-30.
- [57] G. Kiczales, J. des Rivieres, and D. Bobrow, *The Art of the MetaObject Protocol*, MIT Press, 1991.
- [58] "Microsoft Developer Network - System.Reflection Namespace," <http://msdn.microsoft.com/en-us/library/system.reflection.aspx>.
- [59] J. Liberty, *Programming C#*, O'Reilly Media, Inc., Sebastopol CA, 2005.
- [60] T. Petricek, "F # metaprogramming and classes," <http://tomasp.net/blog/fsclassmeta.aspx>.
- [61] T. Petricek and D. Syme, "F # Web Tools : Rich client / server web applications in F #," *Unpublished draft*, available from <http://fswebtools.codeplex.com/>.
- [62] "Microsoft Developer Network - MethodInfo Class," <http://msdn.microsoft.com/en-us/library/system.reflection.methodinfo.aspx>.
- [63] "Microsoft Developer Network - System.CodeDOM Namespace," <http://msdn.microsoft.com/en-us/library/system.codedom.aspx>.
- [64] "Microsoft Developer Network - Using the CodeDOM," <http://msdn.microsoft.com/en-us/library/y2k85ax6.aspx>.
- [65] "Microsoft Developer Network - CodeDomProvider Class," <http://msdn.microsoft.com/en-us/library/system.codedom.compiler.codedomprovider.aspx>.
- [66] "F# PowerPack, with F# Compiler Source Drops," <http://fsharppowerpack.codeplex.com/>.
- [67] "Microsoft Developer Network - Expression Trees (C# and Visual Basic)," <http://msdn.microsoft.com/en-us/library/bb397951.aspx>.
- [68] "Microsoft Developer Network - How to: Modify Expression Trees (C# and Visual Basic)," <http://msdn.microsoft.com/en-us/library/bb546136.aspx>.

- [69] "Microsoft Developer Network - Expression Class," <http://msdn.microsoft.com/en-us/library/system.linq.expressions.expression.aspx>.
- [70] F. Marguerie, S. Eichert, and J. Wooley, *Linq in Action*, Manning Publications Co. Greenwich, CT, USA, 2008.
- [71] J. Skeet, *C# In Depth*, Manning Publications Co. Greenwich, CT, USA, 2008.
- [72] "Microsoft Developer Network - Code Quotations (F#)," <http://msdn.microsoft.com/en-us/library/dd233212.aspx#Y432>.
- [73] "Microsoft Developer Network - Quotations.Expr Class (F#)," <http://msdn.microsoft.com/en-us/library/ee370577.aspx>.
- [74] "F# Quotations Samples on CodePlex - Tomas Petricek," <http://tomasp.net/blog/fsharp-quotation-samples.aspx>.
- [75] "Microsoft Developer Network - String.Replace Method," <http://msdn.microsoft.com/en-us/library/system.string.replace.aspx>.
- [76] "Microsoft Developer Network - Generics (F#)," <http://msdn.microsoft.com/en-us/library/dd233215.aspx>.
- [77] "Microsoft Developer Network - Expr.Call Method (F#)," <http://msdn.microsoft.com/en-us/library/ee370395.aspx>.
- [78] "Microsoft Developer Network - Expr.Value Method (F#)," <http://msdn.microsoft.com/en-us/library/ee340519.aspx>.
- [79] "Forty Six and Two: Traversing and transforming F# quotations: A guided tour," <http://fortysix-and-two.blogspot.com/2009/06/traversing-and-transforming-f.html>.
- [80] "Microsoft Developer Network - Active Patterns (F#)," <http://msdn.microsoft.com/en-us/library/dd233248.aspx>.
- [81] "Microsoft Developer Network - Quotations.DerivedPatterns Module (F#)," <http://msdn.microsoft.com/en-us/library/ee370434.aspx>.
- [82] "Microsoft Developer Network - ExprShape.ShapeVar|ShapeLambda|ShapeCombination Active Pattern (F#)," <http://msdn.microsoft.com/en-us/library/ee370517.aspx>.
- [83] T. Petricek, "F# quotations visualizer - reloaded!," <http://tomasp.net/blog/quotvis-reloaded.aspx>.
- [84] "Microsoft Developer Network - DerivedPatterns.MethodWithReflectedDefinition Active Pattern (F#)," <http://msdn.microsoft.com/en-us/library/ee353670.aspx>.
- [85] "Microsoft Developer Network - Patterns.Call Active Pattern (F#)," <http://msdn.microsoft.com/en-us/library/ee353474.aspx>.
- [86] "Microsoft Developer Network - Expr.Let Method (F#)," <http://msdn.microsoft.com/en-us/library/ee370277.aspx>.

- [87] "Microsoft Developer Network - Patterns.Let Active Patterns (F#)," <http://msdn.microsoft.com/en-us/library/ee340284.aspx>.
- [88] "Microsoft Developer Network - Expr.Lambda Method (F#)," <http://msdn.microsoft.com/en-us/library/ee340341.aspx>.
- [89] "Microsoft Developer Network - Patterns.Lambda Active Pattern (F#)," <http://msdn.microsoft.com/en-us/library/ee340233.aspx>.
- [90] "Microsoft Developer Network - Expr.Sequential Method (F#)," <http://msdn.microsoft.com/en-us/library/ee353459.aspx>.
- [91] "Microsoft Developer Network - Patterns.Sequential Active Patterns (F#)," <http://msdn.microsoft.com/en-us/library/ee353710.aspx>.
- [92] "Microsoft Developer Network - Expr.Application Method (F#)," <http://msdn.microsoft.com/en-us/library/ee340386.aspx>.
- [93] "Microsoft Developer Network - Patterns.Application Active Pattern (F#)," <http://msdn.microsoft.com/en-us/library/ee353646.aspx>.
- [94] "Microsoft Developer Network - Expr.IfThenElse Method (F#)," <http://msdn.microsoft.com/en-us/library/ee370408.aspx>.
- [95] "Microsoft Developer Network - Patterns.IfThenElse Active Pattern (F#)," <http://msdn.microsoft.com/en-us/library/ee340456.aspx>.
- [96] "Microsoft Developer Network - Tuples," <http://msdn.microsoft.com/en-us/library/dd233200.aspx>.
- [97] "Microsoft Developer Network - Expr.Var Method (F#)," <http://msdn.microsoft.com/en-us/library/ee353514.aspx>.
- [98] "Microsoft Developer Network - Quotations.Var Class (F#)," <http://msdn.microsoft.com/en-us/library/ee353442.aspx>.
- [99] A. Rahien, *DSLs in Boo: Domain Specific Languages in .NET*, Manning Publications Co. Greenwich, CT, USA, 2010.
- [100] "FParsec - A Parser Combinator Library for F#," <http://www.quanttec.com/fparsec/>.
- [101] "Microsoft Developer Network - Records (F#)," <http://msdn.microsoft.com/en-us/library/dd233184.aspx>.
- [102] "Bitbucket Documentation," <http://confluence.atlassian.com/display/BITBUCKET/Bitbucket+Documentation+Home;jsessionid=3B3900105B398F1A31AA1130FA1815ED>.
- [103] "Microsoft Developer Network - Using Visual Studio to Write F# Programs," <http://msdn.microsoft.com/en-us/library/dd233169.aspx>.
- [104] "Microsoft Developer Network - Assemblies," <http://msdn.microsoft.com/en-us/library/hk5f40ct%28v=vs.71%29.aspx>.

- [105] "Microsoft Developer Network - Compiler Options (F#)," <http://msdn.microsoft.com/en-us/library/dd233171.aspx>.
- [106] "Eos: Aspect-oriented extension for C#," <http://www.cs.iastate.edu/~eos/>.
- [107] "Stack Overflow - how to build the f# compiler from source," <http://stackoverflow.com/questions/4104480/how-to-build-the-f-compiler-from-source>.
- [108] "CodePlex - F# Compiler Source Changesets," <http://fsharpowerpack.codeplex.com/SourceControl/list/changesets>.
- [109] "Occasional notes: F#: Building compiler from sources.," <http://v2matveev.blogspot.com/2010/08/f-building-compiler-from-sources.html>.
- [110] E.W. Weisstein, "Machin's Formula - from Wolfram MathWorld," <http://mathworld.wolfram.com/MachinsFormula.html>.
- [111] "Microsoft Developer Network - Options (F#)," <http://msdn.microsoft.com/en-us/library/dd233245.aspx>.
- [112] "Microsoft Developer Network - Double Structure (System)," <http://msdn.microsoft.com/en-us/library/system.double.aspx>.
- [113] "Microsoft Developer Network - Stopwatch Class," <http://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.aspx>.
- [114] "Microsoft Developer Network - How to: Set Debug and Release Configurations," [http://msdn.microsoft.com/en-us/library/wx0123s5\(v=VS.100\).aspx](http://msdn.microsoft.com/en-us/library/wx0123s5(v=VS.100).aspx).
- [115] "Microsoft Developer Network - Stopwatch.Frequency Field," <http://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.frequency.aspx>.
- [116] "Microsoft Developer Network - Stopwatch.ElapsedTicks Property," <http://msdn.microsoft.com/en-us/library/system.diagnostics.stopwatch.elapsedticks.aspx#Y216>.
- [117] "The Java Virtual Machine Specification," <http://java.sun.com/docs/books/vmspec/>.
- [118] "Java API Specifications," <http://java.sun.com/reference/api/index.html>.





Listing A.2, below shows the Listing for the advised program, where the calls to the advices are in red.

```

Lambda (k,
  LetRecursive (((calc,Lambda (pow,
    Let (activePatternResult,
      Call (None,
        Microsoft.FSharp.Core.FSharpOption`1[System.Double] | Reachlim|_| [Double](Double, Double),
        [k, pow]),
      IfThenElse (UnionCaseTest (activePatternResult,
        FSharpOption`1.Some),
        Let (r,
          PropertyGet (Some (activePatternResult),
            Double Value,
            []),
          Let (res,
            Call (None,
              Double calcTerm(Double, Double),
              [Value (0.0),
              pow]),
            Sequential (Call (None,
              Void printparams(Double),
              [res]),
              res))),
          Let (res,
            Call (None,
              Double op_Addition[Double,Double,Double](Double, Double),
              [Call (None,
                Double calcTerm(Double, Double),
                [x, pow]),
              Application (calc,
                Call (None,
                  Double op_Addition[Double,Double,Double](Double, Double),
                  [pow,
                  Value (1.0)])))]),
            Sequential (Call (None,
              Void printparams(Double),
              [res]),
              res)))))),
    Application (calc, x))

```

**Listing A.2 – Expression tree for the unadvised program**

The expression for the weaved program indicates that we are capturing the result of a call to `calcTerm` and storing that result in an identifier called `res`. The value bound to `res` is then passed in to the advice – the `printParams` function.

```
Let (res,  
    Call (None,  
          Double calcTerm(Double, Double),  
          [Value (0.0),  
            pow]),  
    Sequential (Call (None,  
                    Void printparams(Double),  
                    [res]),  
                res)))
```

**Figure A3 – Storing the result of `calcTerm` in a `res` variable**

## Appendix B – Raw timing results for the weaved function

Number of iterations	Ticks taken to evaluate term 1	Time taken to evaluate term 1	Ticks taken to evaluate term 2	Time taken to evaluate term 2	Ticks taken to compile term 1	Time taken to compile term 1	Ticks taken to compile term 2	Time taken to compile term 2	Frequency (Ticks per second)
2,500	56,414	0.018883227	2,117	0.000708615	7,823,838	2.618841253	7,143,961	2.391268809	2,987,519
2,500	70,265	0.023519516	3,000	0.001004178	7,158,631	2.396179238	6,443,711	2.156876994	2,987,519
2,500	56,337	0.018857453	2,139	0.000715979	7,646,805	2.559583721	7,162,011	2.397310611	2,987,519
2,500	57,977	0.019406404	2,279	0.00076284	7,127,308	2.385694618	6,434,294	2.15372488	2,987,519
2,500	59,138	0.019795021	1,783	0.000596816	7,269,256	2.433208291	6,463,744	2.163582558	2,987,519
2,500	57,818	0.019353182	1,716	0.00057439	7,129,449	2.386411266	6,367,310	2.1313036	2,987,519
1,000	54,996	0.018408586	1,737	0.000581419	3,300,692	1.104827116	2,578,147	0.862972587	2,987,519
1,000	56,557	0.018931093	1,686	0.000564348	3,378,451	1.130855067	2,820,422	0.944068306	2,987,519
1,000	57,545	0.019261802	2,426	0.000812045	3,873,583	1.296588574	2,937,325	0.983198768	2,987,519
1,000	60,061	0.020103973	1,706	0.000571042	3,673,287	1.229544314	2,556,573	0.85575121	2,987,519
1,000	56,800	0.019012431	1,659	0.00055531	3,384,377	1.132838653	2,564,342	0.858351696	2,987,519
500	54,868	0.018365741	1,638	0.000548281	2,009,659	0.672684927	1,309,097	0.438188678	2,987,519
500	60,024	0.020091588	2,200	0.000736397	2,150,115	0.719699189	1,674,510	0.560501875	2,987,519
500	57,190	0.019142974	1,867	0.000624933	2,025,528	0.677996692	1,341,900	0.449168691	2,987,519
500	57,517	0.01925243	1,710	0.000572381	2,048,307	0.685621414	1,284,395	0.429920278	2,987,519
500	58,626	0.019623641	2,024	0.000677485	2,058,781	0.689127333	1,289,310	0.431565456	2,987,519
250	55,442	0.018557874	1,904	0.000637318	1,376,445	0.460731798	703,494	0.235477666	2,987,519
250	55,460	0.018563899	1,719	0.000575394	1,622,013	0.542929769	876,185	0.293281817	2,987,519
250	77,199	0.025840505	2,603	0.000871292	1,613,377	0.540039076	705,114	0.236019922	2,987,519
250	58,694	0.019646402	1,728	0.000578406	1,467,032	0.491053613	725,649	0.242893518	2,987,519
250	55,076	0.018435364	1,608	0.000538239	1,446,285	0.484109055	731,848	0.244968484	2,987,519
100	57,314	0.019184481	1,661	0.00055598	770,730	0.257983297	392,936	0.131525858	2,987,519

Number of iterations	Ticks taken to evaluate term 1	Time taken to evaluate term 1	Ticks taken to evaluate term 2	Time taken to evaluate term 2	Ticks taken to compile term 1	Time taken to compile term 1	Ticks taken to compile term 2	Time taken to compile term 2	Frequency (Ticks per second)
100	56,204	0.018812935	1,649	0.000551963	768,279	0.257162883	429,272	0.143688459	2,987,519
100	56,593	0.018943143	1,734	0.000580415	764,408	0.255867159	460,108	0.154010067	2,987,519
100	55,071	0.01843369	1,710	0.000572381	758,817	0.253995707	398,225	0.133296223	2,987,519
100	57,320	0.019186489	1,991	0.000666439	770,941	0.258053924	391,171	0.130935067	2,987,519
50	61,837	0.020698446	1,869	0.000625603	689,658	0.230846398	211,995	0.070960218	2,987,519
50	56,268	0.018834357	1,804	0.000603846	608,530	0.203690755	204,202	0.068351699	2,987,519
50	54,970	0.018399883	1,699	0.000568699	577,755	0.193389565	207,595	0.069487424	2,987,519
50	56,139	0.018791178	1,635	0.000547277	624,627	0.209078838	208,996	0.069956375	2,987,519
50	57,261	0.01916674	2,164	0.000724347	589,895	0.197453138	205,782	0.068880566	2,987,519
25	60,278	0.020176608	1,920	0.000642674	503,044	0.168381858	117,350	0.039280085	2,987,519
25	56,762	0.018999712	1,798	0.000601837	475,705	0.159230786	116,976	0.039154897	2,987,519
25	62,541	0.020934093	2,431	0.000813719	481,181	0.161063746	120,969	0.040491458	2,987,519
25	57,627	0.01928925	1,718	0.000575059	490,403	0.164150588	120,592	0.040365266	2,987,519
25	57,855	0.019365567	1,768	0.000591795	492,440	0.164832424	120,995	0.040500161	2,987,519
10	55,781	0.018671346	1,663	0.000556649	534,389	0.178873841	63,242	0.021168736	2,987,519
10	56,242	0.018825654	1,688	0.000565017	461,169	0.154365211	77,181	0.02583448	2,987,519
10	56,186	0.01880691	1,696	0.000567695	473,119	0.158365185	78,640	0.026322845	2,987,519
10	76,280	0.025532892	2,196	0.000735058	465,918	0.155954824	54,021	0.018082228	2,987,519
10	77,941	0.026088872	3,060	0.001024261	473,184	0.158386942	50,849	0.017020478	2,987,519
1	66,086	0.022120696	2,141	0.000716648	422,936	0.141567635	24,810	0.00830455	2,987,519
1	56,134	0.018789504	1,594	0.000533553	426,243	0.142674574	26,637	0.008916094	2,987,519
1	59,465	0.019904476	1,897	0.000634975	441,538	0.147794206	27,050	0.009054336	2,987,519
1	56,645	0.018960549	2,177	0.000728698	426,540	0.142773987	25,518	0.008541536	2,987,519
1	56,403	0.018879545	1,703	0.000570038	450,062	0.15064741	27,756	0.009290652	2,987,519
0	56,220	0.01881829	1,731	0.000579411	419,884	0.140546052	20,464	0.006849831	2,987,519

Number of iterations	Ticks taken to evaluate term 1	Time taken to evaluate term 1	Ticks taken to evaluate term 2	Time taken to evaluate term 2	Ticks taken to compile term 1	Time taken to compile term 1	Ticks taken to compile term 2	Time taken to compile term 2	Frequency (Ticks per second)
0	67,948	0.022743956	3,324	0.001112629	437,024	0.146283254	24,335	0.008145555	2,987,519
0	58,026	0.019422805	1,671	0.000559327	431,587	0.144463349	21,778	0.007289661	2,987,519
0	55,213	0.018481221	1,688	0.000565017	443,546	0.148466336	23,841	0.0079802	2,987,519
0	56,991	0.019076364	1,836	0.000614557	497,426	0.166501368	26,415	0.008841785	2,987,519

## Appendix C – CD with source code

{ CD containing source code is attached here.

Please refer to Section 4.5 for more information. }