An Aspect Oriented Framework in F#

Nitesh Chacowry

MSc Computer Science project report, Department of Computer Science and Information Systems, Birkbeck College, University of London and the 2011

This report is substantially the result of my own work, expressed in my own words, except where explicitly indicated in the text. I give my permission for it to be submitted to the JISC Plagiarism Detection Service.

The report may be freely copied and distributed provided the source is explicitly acknowledged.

Table of Contents

1. Introduction 3

2. Motivation 4

3. Background 6

3.1 A Theoretical Introduction to Aspect Oriented Programming 6

3.2 A Theoretical Introduction to Functional Programming Languages 13

3.3 The .Net Framework 18

4. Current AOP Frameworks for Functional Languages 21

4.1 Complexities of implementing AOP Frameworks for Functional Languages 21

4.2 Rationale for an AOP Framework in F# 21

4.3 Review of Functional Languages Implementing AOP 23

4.3.1 AspectML 23

4.3.2 Aspectual Caml 24

4.3.3 AspectFun 25

5. Framework Specifications 25

5.1 Join points 25

5.2 Pointcuts 26

5.3 Advices 27

5.4 Aspects 28

5.5 Weaving 28

6. Implementation Plan 31

7. Conclusion 32

8. References 33

1. Introduction

This document presents a project proposal for the development of an aspect oriented framework for Microsoft's implementation of a functional language: F#.

- Section 2 presents the motivation for this project.
- Section 3 presents a background to the aspect oriented and functional programming paradigms.
- In section 4 we shall elaborate on the complexities which arise when designing an aspect oriented framework for a functional language. We shall also present a justification to explain why aspect oriented frameworks are applicable and useful to programs written in a functional language.
- We present a program design and task list for the project in sections 5 and 6.

2. Motivation

Within this section we will present an overview of aspect oriented framework (AOP), of F# [1] and discuss the motivations for this project. The terms and concepts introduced in this section are expanded in sections 3 and 4 of this document.

Aspect oriented programming (AOP) is a programming paradigm where functionalities which apply across many other modules are clearly encapsulated and expressed in a separate module [2,3]. AOP, its terminology and implementation mechanism (*weaving*) will be formally introduced in section 3.

Access to AOP functionalities is not usually done via the core programming language. Instead these are accessed through an AOP framework designed for that language. The most popular of such frameworks generally target object oriented languages such as Java or C#. These include AspectJ [4] and Spring for Java [5], Policy Injection Application Block [6][1], Spring.Net [7] and PostSharp [8] for .Net languages.

In the field of functional languages, there currently exist some industrial and academic AOP languages, namely AspectML [9], AspectFun [10], Aspectual Caml [11] or MinAML [12]. Clojure [13], a functional language implementation targeting the *Java Virtual Machine* (JVM) has some AOP implementations [14,15]. However, no such frameworks or language extensions are currently available for Microsoft's functional language: F#. Within this project, we therefore propose the specification, development and test of an AOP framework targeting F#.

We shall cover the background on AOP and F# in section 3. This will be followed by an explanation of the challenges of creating an AOP framework in a functional language. The main motivation behind this project is that it presents an opportunity to reconcile two different aspects of Computer Science (AOP and functional languages). Through this, we shall be covering different facets of Computer Science. Namely, this project presents an opportunity to:

1. Investigate, understand and use functional languages, specifically F#. F# targets the *Common Language Runtime* (CLR) and is a core language shipped with *Visual Studio* (VS) since VS2008 [16]. This is a strong indication of a drive by Microsoft to get its version of a functional language into an industrial setting.

2. Investigate current AOP framework trends for object oriented language and design an AOP framework which is suited for a functional language.

3. Understand the *type inference* system. As we will explain in sections 4 and 5, functional languages (including F#) have a type inference system where the types used in functions or returned from functions are resolved at compile time. We anticipate working with the type-inference system to be challenging, as an AOP framework should be designed to work with modules where the parameter type(s) used are not known in advance. To illustrate, consider the *signature* for a generic method in C#, shown in listing 2.1 below:

[1] However, the Microsoft website does not note consider the Policy Injection Application Block to be an AOP framework [59]

```
public void Execute<T, U>(T first, U second)
```

Listing 1.1: Generic method signature

An AOP framework should be able to inject code into this method, independent of the parameter types T and U. Therefore, we shall be required to apply generic programming techniques [17] or *monads* [18] to compile and run our AOP framework.

4. Investigate and develop a *dynamic weaving* functionality. These are fairly advanced concepts and can be challenging to implement. Solutions, which are discussed in section 5, range from using monads, injecting custom behaviour using the CLR's object lifecycle (i.e. *ContextBoundObject* [19]), or implementing a module to perform *Microsoft Intermediate Language (MSIL)* [20] bytecode inspection and modification.

We present an overview of the .Net Framework in Appendix A.

The framework will target modules written in F# and will be written in F#. Several factors motivate the choice of F# for the AOP framework:

1. Applications written in F# are *managed applications* which compiles to regular MSIL bytecode. It will therefore be possible to address complex framework implementation problems in an imperative .Net language (say C#), and return the result to the AOP framework written in F#.
2. Since F# allows access to the .Net Framework's extensive *Framework Class Libraries* (FCL), it is possible to implement dynamic weaving, say, through the use of the .Net Reflection libraries [21]. The dynamic weaving mechanism need not be implemented in F# as it might be more "natural" to implement such functionalities in an object-oriented language.
3. F# implements characteristics of functional languages, but is flexible enough to allow the developer to bypass the characteristics of *pure functional languages*. For example it is possible to introduce mutable data structures which allow one to carefully introduce side effects[2].
4. An important success factor for any framework is the modularisation of the compiled program to simplify distribution, versioning and patching. Programs written in F# can be packaged and distributed as regular .Net assemblies [22].

[2] This is not strictly a characteristic of F#. ML also allows *references* which permits *side effects* [30].

3. Background

This section will introduce and explain the different paradigms and technologies involved in this project.

We begin with an examination of aspect oriented programming. This will be followed by a discussion of functional languages, with examples in F#. When discussing F# functionalities, our discussion will emphasise the characteristics of F# which can be leveraged (or impacts) the design of our AOP framework, hence a working understanding of functional programming is required.

3.1 A Theoretical Introduction to Aspect Oriented Programming

3.1.1 The purpose aspect oriented programming

A key concept of program design is the one of *separation of concerns* [23], where the program is split into distinct and non-overlapping features. Separation of concerns aims to assist the programmer in achieving low coupling and high cohesion [6] between the program features. Low coupling and high cohesion also assists in program maintenance and in enhancing the program's clarity.

From a practical perspective, however, there are some features which *cross-cut* other features [2]; hence separation of concerns does not assist in modularising these features. These cross cutting features are usually non-functional requirements [24].

These cross-cutting features are formally known as *cross-cutting concerns*.

To illustrate a cross-cutting concern, a program use case could require that the user credentials are checked before business logic is allowed to execute. We say that the program has a *security aspect*, where the credential checking functionality cross-cuts the business logic concern.

A naïve (or direct) approach would be to repeat the credential checking code at *all points* where sensitive business logic is about to be executed. An example is given in C# below in listing 3.1:

```
public void DebitCustomerAccount(decimal amount)
{
    if (User.HasAccess("DebitCustomerAccount ")) // Security aspect
    {
        /*
         *  Business logic {. . .}
         */
    }
}
```

Listing 3.1: "Naïve" approach at creating a security aspect

Repeating the credential checking code yields the following issues, mostly around code maintenance:

1. Programmers will have to remember to add the credential checking code at every critical point where business logic is accessed [25].

2. Manually copying code may lead to errors / typos in the replicated code which could lead to the program getting compiled but displaying incorrect behaviour.

3. Changes in the credential checking code will need to be propagated to every existing implementation, which may lead to additional mistakes in replicating the changes.
4. In some cases, the business logic may be obscured by the credential checking code, or code from other cross cutting concerns. The cohesion of the method is degraded by the introduction of the cross cutting concern and we have also introduced coupling between the two different concerns.

Aspect oriented programming has been formulated as a programming paradigm to address these cross-cutting concerns by providing constructs and patterns which assist in encapsulating cross-cutting concerns into their own modules. Once the cross-cutting concern has been encapsulated, a mechanism is required to combine or *weave* it back to the module - in order to satisfy program use-cases.

A succinct, language-neutral definition of aspect oriented programming is provided by [25], which we quote here in Definition 3.1:

AOP is […] the desire to make programming statements of the form:

In programs P, whenever condition C arises, perform action A

Definition 3.1

From Definition 3.1, an important requirement for *true* AOP is that the program *P* should have no knowledge of the actions *A* which can modify it, otherwise Definition 3.1 is simply the definition for regular procedural programming. To illustrate, a simple *if statement* also fulfils the requirements of Definition 3.1, as we showed in Listing 3.1. This important concept where P has no knowledge of the existence of *A* is formally known as *obliviousness* [25].

To re-emphasise the importance of obliviousness, we return to the security aspect example presented in Listing 3.1. If the code base was maintained by a team of developers, every member would have to remember to add the credential checking code before any business logic executes. Unless strictly enforced, there is no guarantee that this manual step will be done which may result in an application failing to comply with its use cases. In an ideal scenario, developers should only focus on developing the business logic, delegating to the AOP framework the task of adding the security aspect at the right point of execution.

We now introduce some definitions used to describe AOP constructs.

3.1.2 Join Points

There are different types of conditions *C* which may arise during the execution of a program for which we would require action *A* to be executed. *Quantification* is the process of specifying the set of conditions C.

Following the discussion from [26], within a program execution, we are generally interested in a well-defined and clearly identifiable sub-set of conditions $\{P_1...P_n\}$. Examples of such conditions are *method calls* or *method execution*.

Formally, these well-defined and identifiable conditions are known as *join points.* Definition 3.1 can thus be expanded as follows [26]:

In programs P, whenever execution reaches one of the points in {P_1,…,P_n} perform action A

Definition 3.2

AOP frameworks usually expose a point model [27,28], which explicitly states the join points available. Table 3.1, below, presents the joint point model in AspectJ – an AOP framework for programs written in Java. This table is adapted from [27]:

Join point type	Purpose
Method call	Exposes the point in a program when a method is called.
Method execution	Exposes and encompasses the execution of a method.
Constructor call	Exposes the point in a program when an object is created.
Constructor execution	Exposes and encompasses the execution of an object constructor.
Field Access	Exposes the access to an object's field for read (or write) access.
Exception handling	Exposes the point at which the exception handling code in a catch block is executing.

Table 3.1: Joint points exposed by the AspectJ framework

The set of join points is highly dependent on the type of language used. For example, from table 3.1, it can be noted that constructor and field access join points are relevant to object oriented languages.

We defer an example of join points after discussing pointcuts.

3.1.3 Pointcuts

From Definition 3.2, we require the ability to select join points, for example a user might be interested in capturing the join point only when the method "`ExecuteBusinessLogic`" is called.

Formally, the predicate construct which allows the selection of join point(s) is known as a *pointcut*.

AOP frameworks usually have a "sub-language" to define pointcuts [11]. The purpose of the sub-language is twofold:

1. Provide the ability to select join points based on a predicate. For example the predicate may be the method name when selecting over method call join points.
2. Provide the ability to collect the join point *context*. The context is a rather fuzzy concept and varies greatly based on the type of join point. For example the context for a pointcut over a method call join point may include the method parameters [27].

[28] highlights three categories of such sub-languages. We discuss these in table 3.2:

Pointcut language type	Description
Specification based pointcut language	The selection of join points is done through a lexical analysis of the source code – if available. The selection of the join points does not refer to the program context.
State based pointcut language	The selection of join points is based on the next executing statement and the context. Despite providing access to the context, pointcuts of this type does not allow access to the local variables of the next executing statement.
Progress based pointcut language	Allows the inspection of the current call stack and give the pointcut the ability to select join points based on **both** the *context* and the *loci of execution* within the program.

Table 3.2: Pointcut language categorisation

An example of a pointcut selecting over method calls with signatures matching the name "`ExecuteBusinessLogic`" and returns *void*, adapted from AspectJ [27], is shown below in listing 3.1:

```
public pointcut matchBusinessLogicOperation() : call ( void ExecuteBusinessLogic )
```

Listing 3.1: A pointcut to capture calls to `void ExecuteBusinessLogic` in AspectJ

Where:

- `public` is an access modifier.
- `pointcut` is a keyword.
- `matchBusinessLogicOperation` is the name of the pointcut.
- `call` represents the join point.
- `void ExecuteBusinessLogic` represents the method to be wrapped.

3.1.4 Advices

Referring back to Definition 3.1, we have so far seen how C, a program condition, is defined in AOP through join points and pointcut constructs. We now discuss the action A which is executed when the condition C is triggered. The action A to be executed is known formally as an *advice*. An advice encapsulates the logic required to be "injected" at the join points. We say that a program module is *advised* when an advice is attached to the module via a join point/pointcut.

Table 3.3, below, presents the different ways advices can execute. Table 3.3 uses an advised method as example, but advices can execute over any types of join points:

Advice Type	Description
Before advice	These execute before the advised module runs. For example, the advice can implement a caching mechanism such that the method parameters are saved before the advised method executes.
After advice	These execute once the advised module has completed execution. For example a method might require some additional clean up actions to be carried out after execution e.g. disposing of resources. The use of an advice with appropriately set up pointcuts ensure that the clean-up executes obliviously to the programmer.
Around advice	These types of advices have a pre-execution step, following by the execution of the advised module and finally a post-execution step. By convention, most AOP frameworks have a `proceed` method to indicates the point in the around advice where the advised module is allowed to run. For example an advice for a method call might inspect the method parameter to detect parameters requiring resource clean up. After execution of the method, the clean-up operation then runs.
Instead advices	These advices can substitute the advised module, i.e. they allow a form of re-direction [26]. For example, through the use of instead advices, we may dynamically patch a method which is exhibiting poor performance in a production environment. Through the use of a suitable pointcut, any call to the method can be intercepted and re-directed to a patched version of the method. Instead advices are synthetic as they can be constructed from a before or around advice.

Table 3.3: Advice categories

An *aspect* is a construct to encapsulate pointcut declarations (*when to inject the new behaviour*) with the advice (*what the new behaviour should be*). Aspects therefore help to realise the modularisation of a cross-cutting concern.

An example of an aspect implementing and around advice, adapted from AspectJ [27], is shown in listing 3.2:

```
public aspect BusinessLogicAspect
{
    public pointcut matchBusinessLogicOperation() : call( void ExecuteBusinessLogic );

    void around() : matchBusinessLogicOperation()
    {
        /*Apply advice*/
    }
}
```

Listing 3.2: Defining an aspect in AspectJ

Where:

- The advice and pointcut is wrapped in an aspect called "BusinessLogicAspect".
- The around advice is introduced with the AspectJ keyword `around` and is injected into the program flow at the points defined by the `matchBusinessLogicOperation`.

3.1.5 Weaving

Referring back to Definition 3.1, there is an implicit requirement for a mechanism to combine the aspect to the program code, or to the thread of execution.

This is achieved through a technique known as *weaving* and the module which accomplishes the combination of the aspect to the program is known as the *weaver* [6,27].

Careful design of a weaver can help satisfy the obliviousness requirements. Several types of weaving strategy exist:

- *Static weaving* occurs at build time. In this case the weaver modifies the source code (or bytecode if the source is not available) by identifying the join points and injecting the advices. Done carefully, the result can be highly optimised code.

  As [29] points out, a downside of this strategy is that static weaving leads to opaque code where it becomes difficult to separate aspect specific statement from business logic code. AspectJ provides the Aspect J compiler (*ajc*) to enable static weaving [27].

- Conversely, *dynamic weaving* is a strategy where the weaver inspects running code (or code that is about to run) and applies advices as specified in the pointcut [27]. Dynamic weaving is further sub-categorised as shown in table 3.4:

Weaving technique	Description	Example
Proxy-based.	A proxy is used between the client and the advised module.	Microsoft's Policy Injection Application Block, Spring.Net.
Load time static weaving	Aspect weaving is applied just before the assembly is loaded in memory.	See example given below from AspectJ.

Table 3.4: Some dynamic weaving techniques

We now illustrate the load time weaving strategy present in AspectJ [27]. In section 5 we shall consider a similar strategy for our AOP framework. In AspectJ, load time static weaving is applied as follows:

1. The user defined the aspects to be weaved in a separate file (*aop.xml*). This is required to prevent all aspects found on the *classpath* to be weaved, which in turn might cause degradation in performance.
2. The aop.xml file is loaded via a Java agent. In turn, the agent is registered with the JVM.
3. When the JVM is about to load a class, it sends an event to the Java agent.
4. The Java agent inspects the class and weaves in the aspect if required.
5. The JVM then executes the woven bytecode. Figure 3.1 from [27] illustrates this process:

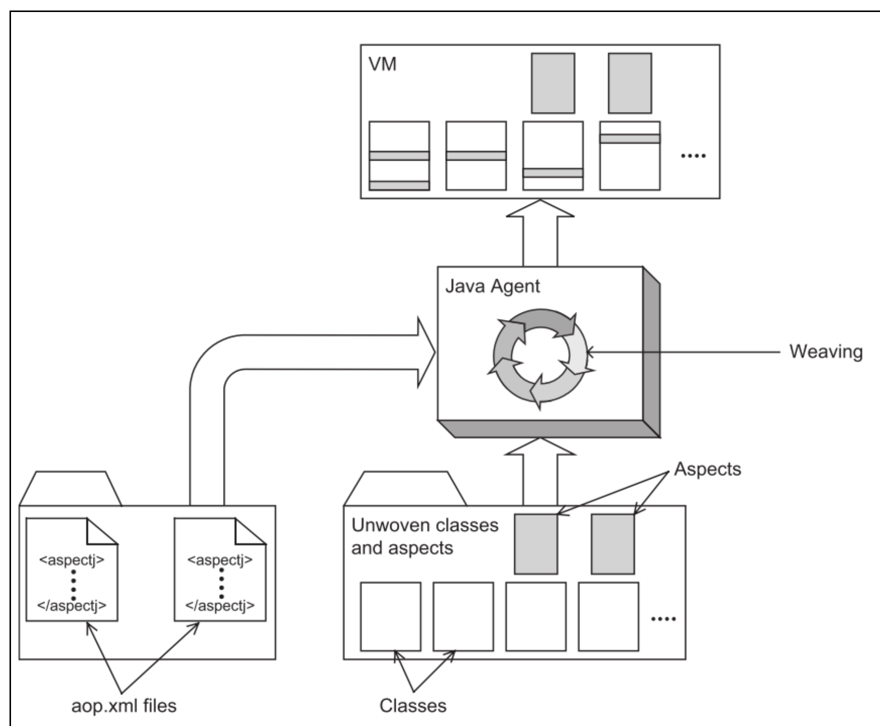

Figure 3.1: AspectJ load time static weaving strategy [27]

From the above, we note the following:

- The `aop.xml` file which specifies which aspects are to be loaded.
- The "regular" Java classes and aspects classes are both compiled to Java bytecode, as represented by the .class files.
- The virtual machine (VM) "box" at the top of the figure represents the woven classes which are executed.

3.1.6 Specifications for a Language-Neutral AOP Framework

Within this section we have presented an overview of aspect oriented programming. Aspect oriented programming is a fairly large subject[3], but we have now identified some of the key design decisions that the design of a generic (language neutral) aspect oriented framework needs to include.

Starting with requirements for the lowest level of our framework (join point) to the highest level (weaving strategy):

1. A set of join points is required to be defined, over which pointcuts can operate.
2. A join point selection syntax is required to be defined such that pointcuts can be specified.
3. Pointcuts should be designed such that they can capture the execution context.
4. The framework shall allow before, after and around advices.
5. The framework shall allow the encapsulation of cross cutting concerns into aspects
6. The framework shall allow aspect weaving in order to satisfy the concept of obliviousness.

Although the specifications above define a language-neutral AOP framework, the realisation of the framework is largely dependent on the program language. We now introduce functional languages in general and F# in particular in order to further develop our specification.

3.2 A Theoretical Introduction to Functional Programming Languages

Functional programming languages are a sub-set of declarative programming languages [30] where the result of a program is achieved through the execution and combination of functions. This can be contrasted to imperative languages where the result of a program is achieved through the execution of commands [31].

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. In addition, functions are allowed to be 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.

Functional languages provide several desirable properties “out of the box”, such as immutability[4], referential transparency[5], pattern matching or type inference [30,31]. These properties have sparked a resurgence of interest in these languages within the IT industry. Some well-known functional languages include Haskell [32], Scheme, Objective Caml [33] or Clojure [13,34]. In addition, some

---

[3] For example, *introductions* are AOP constructs which allows one to change the structure of classes, interfaces of aspects in the program, through the addition of methods or fields [27]. Introductions shall kept out of scope for this project.

[4] This is strictly true pure functional languages. Some functional languages such as ML and F# allow mutable values for additional flexibility.

[5] As for (1), pure functional languages are designed to obey referential transparency rules, however F# allows functions to apply side effects, for example through the use of mutable values.

popular languages now encompass both imperative and functional paradigms such Python [35], Scala [36] or the Linq functionality in C# or VB.Net [37].

This project focuses on F# which is Microsoft's implementation of a functional language [1]. F# takes its roots from ML (Meta Language). ML was developed in 1979 as language to control a theorem prover called *Logic of Computable Functions* (LCF) [30]. ML evolved into CAML (Categorical Abstract Machine Language) and one of its descendants is Objective Caml (OCaml) [38]. F# was developed by Don Syme and his team at Microsoft Research Cambridge in 2002 and draws some of its syntax and concepts from Caml [16]. F# became one of the core languages shipped with the Microsoft Visual Studio 2008 *integrated development environment* (IDE) [31].

We now discuss some of the characteristics of functional languages which will allow us to further the AOP Framework specifications. Please note that standard functional programming concepts, such as pattern matching, first-class/higher-order and lambdas functions are not covered in this section. Further information on F# can be obtained from [31,39,40].

3.2.1 Free Variables and Closures

A *free* variable is one within a particular expression which is not bound. A free variable is therefore a variable which is not passed into the function as a parameter or created locally, yet there is a reference to that variable within the function. A *closure* is a function where all free variables have been bound.

A simple example from F# is shown below, in listing 3.2:

```
let closeme (unbound:int) =
    fun (parameter:int) -> unbound + parameter
```

Listing 3.2: Closure example in F#

In the above, the `closeme` function returns a *lambda function* [41], namely `fun(parameter:int) -> unbound + parameter`. This lambda uses two parameters: `parameter` and `unbound` both of integer types. However `unbound` is handed from the enclosing function – we say this is an *outer variable* [42]. Within the context of the lambda `unbound` is the free variable. Conversely `parameter` is specified as the lambda parameter hence this variable is bound. The following F# fragment will bind the `unbound` parameter to return a *closure* – we say that the outer variable has been *captured* [42].

```
let value = 6
let closure = closeme value
```

An interesting property of closure is that closures keep a reference to the captured variable event if the original context where the outer variable was defined is out of scope [42].

One of the AOP framework specifications presented in section 3.1.5 was that our pointcut should be designed such that it captures the context. Since closure is a characteristic of functional languages, our AOP framework will therefore be required to include any captured variables when retrieving the context.

3.2.2 Currying

In functional programming, c*urrying* is the mechanism where a function which takes multiple arguments can be transformed into a function taking a single argument. If not all parameters are

closed, another function is returned through the currying process. The function evaluates only when all parameters have been defined.

An example in F# is shown below, in listing 3.3:

```
let curryme (parameter1:int) (parameter2:int) = parameter1 + parameter2

let curry = curryme 10 // set parameter1 to 10 and return another function

let result = curry 25 // set parameter 2 to 25 and return 35
```

Listing 3.3: Currying example

In the example, the function `curryme` expects two parameters of type `integer`. The example then defines another value `curry`, which is the `curryme` function with its `parameter1` set to 10. The function evaluates after `parameter2` is set to 25 on the last line of the example.

Currying is an important characteristic of functional languages, the pointcuts in the proposed AOP framework shall be required to be able to capture function even when curried. This is further discussed in section 5.2.

3.2.3 Type Inference

Type inference is an important characteristic of functional programming languages [31,40]. This functionality allows the compiler to infer the type of the parameters and return type of a function without requiring the programmer to explicitly refer to the data type(s) involved.

We may re-write example 4.2 by dropping the explicit type declarations for the function definition. These type declarations are known as *type annotations.* Example 4.2 can therefore be simplified to:

```
let curryme parameter1 parameter2 = parameter1 + parameter2
```

The *function signature* represents the desired intent – i.e. that `curryme` is a function taking two integers and *transforms* it into another integer [31]:

```
val curryme : int -> int -> int
```

This is possible because the *default behaviour* of the F# type inference system is to infer that the (+) operator uses integers as operands and outputs an integer [40].

Note that parameter or return types may be *generic* – which signifies that a function may be designed to accept parameters of any type. If the function body does not contain sufficient information to assist the type inference system to identify the types used, F# infers that the function uses generic types. This process is known as *automatic generalisation*. An example is shown below, adapted from [43]. In this example we are constructing a *tuple* [44] from the parameters to the `tuplefactory` function.

```
let tuplefactory parameter1 parameter2 = (parameter1, parameter2)
// inferred function signature: val tuplefactory : 'a -> 'b -> 'a * 'b
```

Listing 3.4: Example of automatic generalisation

The generic parameters are prefixed with a single quote `'a` and `'b` and allows us to construct tuples consisting of two integers, a string and an integer, a custom made type and a string etc…

More formally, the ability to use generics means that functions in F# (and some other functional languages) support *parametric polymorphism*. Functions implementing polymorphic parameters are known as p*olymorphic functions*.

In section 5.2, we shall explain design considerations to ensure that our AOP framework is compatible with polymorphic functions.

3.2.4 Monads

A monad can be loosely explained as functions which operate over data. The monad is usually accompanied by other constructs which allows small functions to be “bound” together into a working program [45]. Monads are formally defined in a branch of mathematics known as *category theory*. Within this section we define the term monad and derive its accompanying constructs. We consider monads here, as these have characteristics which may enable us to create AOP behaviour in our framework [45]. A further introduction to monads is given in [18], [46] and [43]. Within the context of F#, monads are known as *computation expressions* [18,47], but in this document we shall exclusively use the term monad.

As explained earlier, a loose definition of a monad is a function which operates over data. The operations carried out can be IO or exception handling or security checking. The nature of the operation is un-important, solely the fact that there exists such operation which augments and enriches the data.

We now derive the monad and its accompanying constructs by introducing a function *f* which operates over a type *a* and has the simple function signature:

```
f : a -> a
```

Listing 3.5: A general function f

Namely, this function takes a parameter of type *a* and returns a value of type *a*. Now if we introduce a monad *M*, we can amend the expression for *f* such that *f* returns a function *M a*:

```
f : a -> M a
```

Listing 3.6: Unit declaration and introduction of monads

Formally, an expression which carries out the above operation is known as a *unit*. In its current form, however, the monad has broken the compositionality characteristic of function *f*. To elaborate on this point further we now introduce another function *g*:

```
g : a -> a
```

Listing 3.7: A general function g

Clearly, from listings 3.5 and 3.7, the following function compositions are possible:

$f \circ g : a \to a$ (where the result of *g* is input into the function *f*)

-and-

$g \circ f : a \to a$ (where the result of f is input into the function g)

Definition 3.3: Allowable function compositions without monads

Where the ○ operator is the "composition" operator. Assume now that there is a requirement to apply *M* to the output of *g*, i.e. we now declare the function *g* as a unit, but also apply the same function *M* to the output of *f*:

```
g : a -> M a
```

Listing 3.8: g as a unit

By applying M to the output of *f* and *g*, it is not possible to create the same compositions: *f* ○ *g* or *g* ○ *f.* Hence we have gained some program functionality through the introduction of the operation *M*, but we have lost the desirable characteristic of being able to compose functions. In order to re-introduce this characteristic, we define a new operator called *bind*, which we shall represent by the symbol ! (bang). The bind operator aims to replace the composition operator ○ introduced previously. We shall now derive its function signature.

From Definition 3.3, the function *g* ○ *f* has the signature *g* ○ *f: a -> a*, clearly the ! operator should have a similar signature, namely *a -> M a.* We introduce two *lambda* functions which operate over type *a* and return *f a* and *g a*, namely *(a -> f a)* and *(a -> g a).*The requirement for the ! operator is that it should take the output of *(a -> f a)* and input it to the lambda *(a -> g a).*This is shown in listing 3.9:

```
( a -> f a ) ! ( a -> g a )
```

Listing 3.9: The ! operator

From the above, the ! operator should be designed to accept:

- An input *f a* (which is by definition equal to *M a*),
- A lambda *(a -> g a)*, which is equivalent to (*a -> M a),*
- And return the monad *M a* (which recall is the output of *(a -> g a*))

From this discussion, if we design the ! operator to have the following signature, we can re-instate function composition:

```
! : M a -> (a -> M a) -> M a
```

Listing 3.10: Function signature for the bind operator

We now introduce a more general signature for the ! operator to define a composition *g* ○ *f* (more precisely *f ! g*) where *g* and *f* have the following definitions:

```
f : a -> M b

g : b -> M c

! : M b -> ( b -> M c ) -> M c
```

Definition 3.4: Generalisation of the bind operator

To explain further the importance of definition 3.4, we have essentially shown that it is possible, through a carefully crafted bind operator (!), to combine two (or more) functions with different

input and output parameters. In effect the ! operator allows us to create the composition $g \circ f$. More precisely, this composition has the form *f ! g* and returns a general function with the signature *a -> M c.*

We return to the concept of monads in section 5.4 when we discuss the weaving strategy. The parallel of monads to AOP programming are:

- Monads can be used to encapsulate crosscutting concerns.
- The unit operator can assist in the generation of monads, hence allowing one to "wrap" a function output into a monad.
- We can use the bind operator ! to compose our programs, such that the calculation flow and order of the calculation is un-changed from the original program design.

3.3 The .Net Framework

This section presents a high level introduction to the .Net Framework. This section aims to explain how interoperation between C# and F# can be achieved. For further reading, please consult [48] or the introduction in [49].

The .Net Framework consists of a program runtime (the *Common Language Runtime* or CLR) and a set of libraries known as the *Framework Class Libraries* (FCL). In order to develop an application targeting the CLR, a developer must use a compiler which translates the source code (written in C#, VB.Net, F# or any other) into legal *Microsoft Intermediate Language* (MSIL). In addition to MSIL, the compiler is required to generate *metadata* describing the types and members used in the program. The MSIL and metadata are encapsulated in an *assembly*, usually a file with an exe or dll extension.

An assembly containing MSIL and metadata can be loaded and executed by the CLR. The CLR executes a multitude of tasks, but the most important is that it further compiles the MSIL into native CPU instructions. This is an on demand process known as *just in time* compilation. The process from compilation to execution is shown below (adapted from [48]):

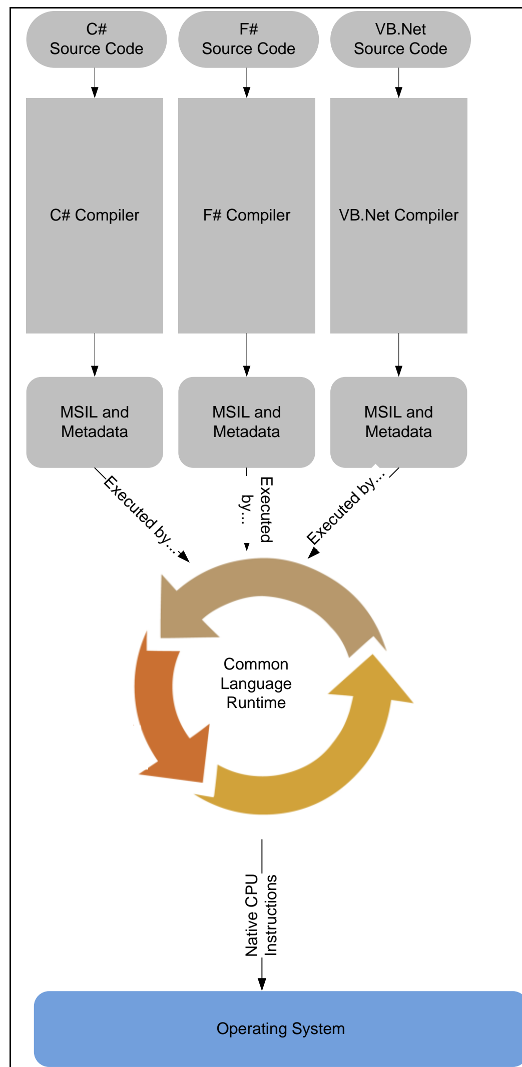

Figure 7.1: Illustration how the compilation and execution process

The CLR also handles unloading the assembly. Since the CLR manages the whole lifecycle of the application, we say that the application is *managed* [48]. Clearly, since the CLR only processes MSIL and metadata, the actual high level language used during the development process is ultimately unimportant. In this project we leverage this functionality as we are able to develop code in F# but can use an imperative language to resolve a particular problem where more natural to do so.

The Framework Class Library is an extensive library of types which provide an access to standard functionalities, such as handling IO, accessing metadata, performing operations on XML data etc… The relationship between the FCL and the CLR can be logically represented as follows, figure 7.2, adapted from [49]:

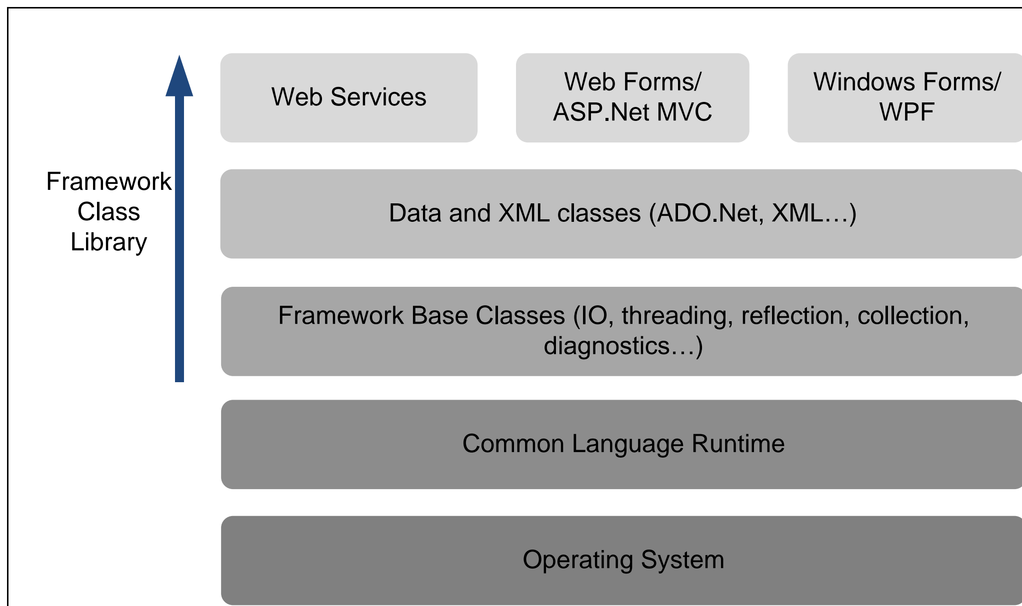

Figure 7.2: Framework Class Library and Common Language Runtime

Being a .Net language, F# can access these libraries. In addition, we can define *interfaces* in C# and implement these in an F# class. In effect, this allows us to pass information between C# and F# modules via an interface. It is also possible to develop an AOP framework in F# to advise modules written in a different language which targets the CLR.

4. Current AOP Frameworks for Functional Languages

Sections 3 presented the theoretical background of AOP and functional languages. We will now discuss the motivations and challenges of creating an AOP framework for F#. We begin by discussing the complexities which need to be addressed when developing our framework for F#.

4.1 Complexities of implementing AOP Frameworks for Functional Languages

As noted in section 1 and re-iterated by [10], AOP is not in common usage within the functional programming community. An important reason is that AOP breaks to the concept of *referential transparency*.

Referential transparency is often summarised as "equals can be replaced by equals", where if a function evaluates to a result for a known input, it can be replaced by its result in any calling code [50]. Referential transparency assists in reasoning about function output and is an important concept within the functional programming community [30]. Clearly, through the use of an AOP framework, one can inject an advice with side effects into a function call, which breaks the concept of referential transparency.

Furthermore, the concept of obliviousness complicates the ability to reason about a function. To illustrate, assume there exists a function *f* called by client. The client might reasonably expect a particular calculation to be executed, but given that the function call may be intercepted by pointcut and the result amended, we cannot guarantee to the client that the expected result will be returned.

Another reason for the low uptake of AOP within the functional programming community is that AOP framework can allow one to break *parametricity* [10][51]. Parametricity states that for a given a generic function *f* which can accept parameters of **any** type, the function is not **naturally** able to assert any information about the type. Theoretically, therefore, the only useful work this function can do is to return a constant.

However, through reflection or other programming technique, we can design our AOP framework to apply runtime type analysis or type casting to gather information about the parameter types – in effect breaking the rule of parametricity. As mentioned by [52] when developing AspectML, it was judged that runtime type analysis was required. As a result we can anticipate that our AOP framework might need to break polymorphic parametricity.

4.2 Rationale for an AOP Framework in F#

Despite these reservations, there are several reasons to consider an AOP framework for a functional language:

1. Given that functional languages allow for declarative syntax, the framework can be designed such that the function calls to the AOP framework resembles a *domain specific language* [53].

We elaborate further here by referring to the AspectJ aspect example given in listing 3.2, which is repeated below:

```
public aspect BusinessLogicAspect
{
    public pointcut matchBusinessLogicOperation() : call( void ExecuteBusinessLogic );

    void around() : matchBusinessLogicOperation()
    {
        /*Apply advice*/
    }
}
```

The following snippet, shown in listing 4.1 expresses the same aspect albeit using a more *fluent* syntax.

```
let aspectapply advice apply (onpointcut:string) = { /*Apply advice */ }
```

Listing 4.1: Aspect construct in F#

Where:

- `aspectapply` is the function name.
- `advice` is the function to inject.
- `apply` specifies whether the advice shall be applied before, after or around the join point.
- `onpointcut` is the name of the pointcut.

We can therefore create an aspect to apply an around advice called `custombehaviour` which executes on pointcuts called `EveryBusinessLogicCalls`, as shown in listing 4.2 below:

```
let woven = aspectapply custombehaviour around “EveryBusinessLogicCalls”
```

Listing 4.2: Realisation of the aspect construct

The F# version can be judged to be more concise, simpler and expresses intent more clearly. More complicated aspects may be achieved through function composition or currying.

2. Similarly, functional languages can be used to express rich aspect policies [54].
3. From our discussion of functional languages and currying, we noted that functional languages allow the combination of first-order functions into higher-order functions. We can leverage this functionality to compose our aspects in a more dynamic fashion. This also allows for de-coupling between the different components of the framework, for example, the pointcut can resolved and declared separately from the advice, and the result composed into an aspect much later in the program.

   From the AspectJ example given in listing 3.2, the advice body and the pointcut are declared within the same aspect. Although it might be possible to construct a pointcut which can be globally re-used, a natural approach is to declare and use the pointcut within the same aspect. Through the use of currying and function composition, we can re-use the same pointcut in multiple aspects. A similar argument can be made for the ability to re-use advices.

4. There is a recent push by language designers to foster a wider adoption of functional languages within the IT industry, as noted by the introduction of F# as a core language of Visual Studio 2008. AOP encapsulate concerns which appear in many industrial projects, such as profiling, caching and security. It is likely that there will be a demand to implement these aspects in a non-intrusive way across code written in a functional language.

In section 1, we elaborated further on the choice of F# for this project, some of the key reasons include the fact that F# compiles to MSIL and is integrated with the .Net framework.

4.3 Review of Functional Languages Implementing AOP

This section highlights the features of some existing functional languages which have AOP constructs.

4.3.1 AspectML

AspectML [9], [52] is a functional programming language which provides standard AOP constructs such as pointcuts, advices (before, after, around). Aspect ML re-uses the syntax from Standard ML.

In order to capture the context of the advised function, AspectML makes use of run-time type analysis through a call to a dedicated function called `case-advice`.

AspectML supports *monomorphic* and *polymorphic pointcuts*. A monomorphic pointcut is one which selects the advised function based on statically defined parameter types. A polymorphic pointcut will select the advised function based a set of valid parameter types.

The snippet below, adapted from [9], shows an example of an *around* advice implements a caching aspect:

```
val fCache  = (* Create a new global cache *)

advice around (| #execute# |) (arg, _, _) =
    case (cacheGet (fCache, arg))
        of Some res => res
        | None => let
                    val res = proceed arg
                    val _ = cachePut (fCache, arg, res)
                    in
                        res
                    end
```

Listing 4.3: An caching aspect in AspectML from [9]

Stepping through the example:

1. A new cache is generated called `fCache`.
2. We then define the around advice. The notation `(| #execute# |)` applies the advice around any function with **execute** in its name.
3. The parameter `(arg, _, _)` is triple where:

- The first item captures the argument passed to the advised function.
- The second item in the triple (unspecified in listing 4.3) is bound to the current call stack.

- The last item in the triple (unspecified in listing 4.3) provides metadata about the advised function.

4. The function begins with a call to `cacheGet` which returns an *option.* The option assists in deciding whether the argument value exists in the cache.

5. If the argument is found, it is returned from the cache (`Some res => res`).

6. Otherwise the original `execute` function is called (`val res = proceed arg`) and the output is bound to the `res` local variable.

7. Finally, the value of `res` is saved to the cache.

We can note the following from listing 4.3:

- The brevity of the advice declaration and its resemblance to a domain specific language.
- The absence of the aspect construct, which is superfluous.
- The mechanism to capture the context is achieved through a triple (*argument*, *stack context*, *advised function metadata*).
- The join point is implicitly declared, i.e. we do not specifically mention that the join point is a function call.

4.3.2 Aspectual Caml

Aspectual Caml [11] is a functional language based on Objective Caml which supports standard AOP constructs, such as pointcuts, advices (before, after, around). As for AspectML, there exists support for monomorphic and polymorphic pointcuts.

An example of an advice declaration adapted from [11] is given below which implements an advice which implements tracing:

```
Advice tracing = [ around call eval env; t ]

 (* Advice code *)
```

Listing 4.4: Tracing advice in Aspectual Caml

In the code above, an around advice called tracing is declared. The "`call eval env; t`" is the pointcut where:

- The join point is defined as being a `call` join point.
- The pointcut advises the `eval` function.
- `env` and `t` are the parameters to the `eval` function and are handles to the function context.

From listing 4.4, we may note the following characteristics of Aspectual Caml:

- The aspect construct is not presented here, again it appears superfluous.

- The join point type is explicitly declared.

4.3.3 AspectFun

AspectFun ([10], [55] and [56]) is an AOP language which employs a Haskell-like syntax. AspectFun employs type-directed programming in order to allow pointcut selection based on specific parameter types.

To illustrate a usage of AspectFun, we can assume the existence of a polymorphic function called `sort` which operates sort functionality over a list. Assume that there exists a requirement such that the function `sort` is not called for lists which are nearly sorted. This requirement can be implemented using an around advice (`checkifsorted`) which is shown in listing 4.5 (adapted from [55]):

```
checkifsorted@advice around {sort} (arg) = if isSorted arg then arg else proceed arg
```

Listing 4.5: An around advice in AspectFun.

5. Framework Specifications

Within this section we now augment the AOP framework specifications and present further technical considerations.

We propose to generally follow the terminology and patterns in use in current AOP frameworks. These include: the ability to define join points, pointcuts, and advices and combine these in an aspect. However, we will not port a particular functionality if it does not “naturally” apply to a functional language. For example, AspectML and Aspectual Caml do not include an explicit *aspect* construct.

Unlike AspectML, Aspectual Caml and AspectFun, we shall not build a language to implement AOP in F#. Instead we shall define a framework which exploits existing .Net Framework functionalities. We expect the framework to be distributed as a separate assembly.

Within the scope of this section we refer to “client code” as any user code which uses the framework.

5.1 Join points

Referring back to the language neutral framework specification introduced in section 3.1.6:

A set of join points is required to be defined, over which pointcuts can operate.

Since every construct in functional programming is a function, important join points are *function boundaries* and the execution of functions. For this project, we propose to limit the scope of join points to function calls, more precisely, the point before a function executes.

F# does have a concept of classes, allowing one to build object oriented constructs. We propose not to implement OOP-specific join points (e.g. field accesses) within the scope of this project and focus on core functional constructs.

5.2 Pointcuts

A join point selection syntax is required to be defined such that pointcuts can be specified.

As mentioned in our join point requirement, we propose to restrict the scope of join points to function calls.

- The framework shall allow named pointcuts only, such that they can be re-used throughout client code.
- Pointcuts shall select join points by referring to their names in a **case-sensitive manner**.
- Pointcuts shall allow simple wildcards (e.g. *) when specifying the function names. If we extend our framework to allow the use of regular expressions we anticipate using the `System.Text.RegularExpressions` [57] types in the .Net framework.

Using the requirements above, a proposed syntax for a pointcut is presented below:

```
let transactionpointcut = pointcut "CheckAccessToBusinessLogic"

                          onfunctioncall "Execute*Transaction"
```

Where:

- `pointcut "CheckAccessToBusinessLogic"`: establishes a *named pointcut*.
- `onfunctioncall`: establishes the join point over which to operate. In our framework the join point will always be `onfunctioncall`.
- `"Execute*Transaction"`: establishes the function(s) to be advised. Note the wildcard (*) which would match functions such as `"ExecuteAccountDebitTransaction"`, `"ExecuteaccountcreditTransaction"` but not `"executeaccountcredittransaction"` as join point selection is case sensitive.

- At this stage, we will be excluding lambdas (anonymous functions) from the pointcut mechanism – i.e. the pointcut shall not be able to capture lambda functions.

We now discuss the requirements to capture the execution context, namely:

Pointcuts should be designed such that they can capture the execution context

- The pointcut syntax will allow the capture of the function parameters.
- The pointcut syntax shall be able to capture the call to curried functions.

Using the above requirements, a proposed use case is as follows:

Assume that there exists a function which debits an amount from a customer's account. This function will be curried because the customer name is known early in the program flow, but the exact amount is known at a later stage of the program flow.

```
let ExecuteAccountDebitTransaction customername amount = printfn "Debiting £%d for
%s" debitamount customername

let customer = ExecuteAccountDebitTransaction "John Smith"

let customer 20 // Debiting £20 for John Smith
```

Now assuming that we define our pointcut as follows – we are using *type annotations* here to assist the type inference system from resolving the function types at compile time:

```
let transactionpointcut = pointcut "CheckAccessToBusinessLogic" onfunctioncall
"Execute*Transaction" (customername:string) (amount:int)
```

At the point of execution, the pointcut should be able to capture the context of the curried function, namely that `customerName` is bound to "John Smith" and `amount` is bound to 20.

Note that there is no need for the parameter names in the pointcut (`customername` and `amount`) to match the parameter names of the function to be advised, merely that they match in terms of parameters type and number.

To elaborate further, the following pointcut will only match functions with names like `Execute*Transaction` and a single parameter of type integer:

```
let transactionpointcut = pointcut "CheckAccessToBusinessLogic" onfunctioncall
"Execute*Transaction" (amount:int)
```

* In the pointcut examples given so far, we have focused on advising a *monomorphic* function. Since polymorphic functions are an important aspect of functional languages and F#, we require extending the pointcut syntax in order to advise polymorphic functions. In this case we drop the type annotations from the pointcut declaration and allow the type inference system to automatically generalise the types used. We anticipate the use a run-time type analysis strategy to be able to resolve the scope of operations which can be carried out on the parameters, once the client code has defined the *type arguments* [42].

5.3 Advices

The framework shall allow before, after and around advices.

As described in section 2, advices are code items which are injected at selected join points. The framework will allow client code to select whether the advice should run before, after or around the advised function.

A prototype signature to define the advice is given below:

```
let advice apply (onpointcut:string) = {…}
```

Where:

- `apply` is a *discriminated union* specifying a before, after or around advice.
- `onpointcut` is the pointcut name.

In the case of an around advice, the framework will expose a polymorphic function called `proceed`. The purpose of `proceed` is to call the advised function and get its return value. An issue to be addressed is that the advised function might have an arbitrary number of parameters of any type hence `proceed` should also be able to handle situations where the number of parameters and their types are known only at runtime.

F# does not support the C# *parameter array* construct – known as *params* in C# [40], [58]. This constructs could have been sufficient to specify the `proceed` function. Therefore the following strategy is considered:

1.1 Inspect the pointcut declaration via reflection, say, and determine the input parameters to the advised function.

1.2 Using the pointcut context, build an ordered list, where every node of the list will contain the parameter and the *boxed* parameter value.

1.3 The proceed function shall therefore have at least the following two parameters:

1.3.1 A parameter indicating the advised function name.

1.3.2 A parameter accepting the ordered list of function parameters.

1.4 Return values shall be boxed.

Please note that there will be a performance overhead in boxing/un-boxing parameters and return values. Within the scope of this project we will defer performance considerations.

5.4 Aspects

The framework shall allow the encapsulation of cross cutting concerns into aspects.

Within the scope of this project, we will focus mostly on join point, pointcut and advices. Aspects are a method to encapsulate functionalities (for example a security aspect, a caching aspect etc…). Aspects also assist in testing [27]. We propose that the aspects in the framework shall be `advice` functions in curried form:

```
let aspectapply advice apply (onpointcut:string) = advice apply onpointcut
```

5.5 Weaving

The framework shall allow aspect weaving in order to satisfy the concept of obliviousness.

We propose to use dynamic rather than static weaving in this project.

Recall that the sole join point exposed by the framework will be the function call join point. In order to satisfy the requirement for obliviousness, this project will aim to avoid using attributes in the target code, as this would create coupling to the AOP framework.

To achieve low coupling, there are three strategies which could be used to weave the aspect to the target code. These strategies are currently under investigation and a decision will be taken after

further experiments, we list them here in order of increasing complexity (conversely it could be argued that these are presented in order of decreasing elegance).

5.5.1 Monads as a weaving strategy

At the simplest level, the dynamic weaving strategy could be applied using monads [45]. We list the parallel between AOP and monads:

- As per our definition in section 5.1, join points will cover function calls only. As monads essentially work over function call boundaries, these join points will be exposed for "free".
- At this stage, further investigation is required to identify the application of pointcuts within an AOP framework:
- We might be required to re-define our join point selection syntax such it looks less like a predicate but is a *composition root*, which allows the user of the framework to orchestrate smaller functions into a larger working program. The composition of these functions into a larger program shall be done through the bind operator.
- The bind operator can have access to the context, making the capture of the program context a simpler exercise.
- Further investigation is required to evaluate the behaviour of the pointcut under a simple currying scenario shown in figure 5.2.
- Aspects are equivalent to monads and advices are the function within the bind operator.

Our first attempt to implement a dynamic weaving strategy will be to use monads.

5.5.2 Common Language Runtime Strategy

A more complex solution is to monitor the Common Language Runtime to register for events which are generated when a function is about to be called. An event handler can then capture this event.

When the function call event is triggered, we shall inspect the function and decide whether to apply advices based on the pointcut definition. A technical solution using an existing .Net construct would be the use of *ContextBoundObject* [19]. We expect to develop the weaver in C# should we decide to use this particular strategy.

If we complete the implementation of a dynamic weaving strategy using monads, an attempt will be made to implement a dynamic weaver which uses the Common Language Runtime.

5.5.3 .Net Agent Strategy

Using a similar strategy as AspectJ, we can apply load time weaving. For this strategy, we shall investigate whether an item similar to a Java agent can be created to listen for events when the CLR is about to load an assembly into memory. When the event is triggered, the agent can then inspect the bytecode to identify join points and inject the advice where required. New bytecode would then be emitted. Issues with this approach are:

- It will be difficult to apply this functionality to obfuscated code.
- Should there be a system crash, additional effort is required to “disentangle” advice code and source code from the stack trace.
- Performance degradation can be anticipated when the agent first runs and injects the advice into the bytecode. However once the advice has been woven and loaded into the CLR, the code can execute at normal speed [27,48].

We anticipate this strategy to be the most complex to implement, and this will be attempted if time permits.

6. Implementation Plan

This section provides a summary of the tasks required to complete this project:

- Join points: Investigate and develop a functionality to enable advices to be executed on function calls.
- Develop the pointcut constructs. Based on the research, we will begin by implementing a weaver through monads. In this case, the pointcut is essentially a language to define how the application is composed. A plan to develop the pointcut is as follows:
  - We shall begin by defining and creating polymorphic pointcuts, as it will then be simpler to narrow this to monomorphic pointcuts.
  - Develop unit tests to verify that polymorphic pointcuts and monomorphic pointcuts are activated when un-curried functions execute.
  - Develop unit tests to verify that polymorphic pointcuts and monomorphic pointcuts are activated when curried functions execute.
- Develop the AOP advice functionality. We shall build unit tests to verify the following:
  - Ensure that before advices execute before the advised function executes.
  - Ensure that after advices executes after the advised function executes.
  - Develop and test the `proceed` function.
  - Ensure that around advices execute partially before and partially after the advised function.
- Dynamic weaver:
  - Develop a weaver based on monadic constructs.
  - If time permits, develop a weaver based on common language runtime events.
  - If time permits, develop a weaver based a Java agent like mechanism.

7. Conclusion

In this document, we have explained the motivations to develop an aspect oriented framework in F#. We can summarise these motivations as follows: AOP is not a popular paradigm within the functional programming community, as it breaks some important characteristics of functional programming, such as referential transparency. However, we believe that there is a push within the IT industry to make functional languages more popular, as can be seen by the recent promotion of Microsoft's F# as a core language in the Visual Studio IDE. Given that AOP defines patterns and constructs to encapsulate cross-cutting concerns, we believe that an investigation of AOP for functional languages is warranted.

Current AOP implementations for functional languages have focused on developing a new language to provide the user access to AOP constructs. These languages re-use syntax from existing languages. For example AspectFun re-uses Haskell syntax; AspectML and Aspectual Caml both use the language from the ML family of functional languages. Our approach is different. We propose to build a framework using core characteristics of functional languages together with the .Net Framework libraries, without having to implement custom made parsers and compilers.

We defined a language-neutral AOP framework to contain at least five major components, namely join points, pointcuts, advices, aspects and weaving, and propose to implement these components in our framework. Our primary attempt will be to use monads – a powerful and versatile functional programming construct. If time permits, we shall investigate the possibility to use reflection and bytecode inspection as more complex weaving strategies.

From this project we aim to demonstrate a viable design and implementation for an AOP framework for a functional language. We also aim to show how functional languages can assist in in designing a fluent interface to the framework.

8. References

[1] "F# - Microsoft Research," *http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/*.

[2] G. Kiczales, "Aspect-oriented programming," *ACM Computing Surveys*, vol. 28, Dec. 1996, p. 154-es.

[3] G. Kiczales, J. Lamping, and A. Mendhekar, "Aspect-oriented programming," *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1997.

[4] "The AspectJ Project," *http://www.eclipse.org/aspectj/*.

[5] "Aspect Oriented Programming with Spring," *http://static.springsource.org/spring/docs/2.5.0/reference/aop.html*.

[6] D. Esposito and A. Saltarello, *Microsoft .Net: Architecting Applications for the Enterprise*, Microsoft Press, 2009.

[7] "Spring.NET - Application Framework," *http://www.springframework.net/*.

[8] "PostSharp," *http://www.sharpcrafters.com/*.

[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] 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.

[11] 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.

[12] 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.

[13] "Clojure," *http://clojure.org/*.

[14] "Robert Hooke - An AOP Framework for Clojure," *https://github.com/technomancy/robert-hooke*.

[15] "Ring," *https://github.com/mmcgrana/ring*.

[16] J. Harrop, *F# for Scientists*, Wiley-Interscience, 2008.

[17] "Microsoft Developer Network - Generics (F#)," *http://msdn.microsoft.com/en-us/library/dd233215.aspx*.

[18] D. Syme, "Some Details on F# Computation Expression," *http://blogs.msdn.com/b/dsyme/archive/2007/09/22/some-details-on-f-computation-expressions-aka-monadic-or-workflow-syntax.aspx*.

[19] "Microsoft Developer Network - ContextBoundObject Class," *http://msdn.microsoft.com/en-us/library/system.contextboundobject.aspx*.

[20] "Microsoft Developer Network - Compiling to MSIL," *http://msdn.microsoft.com/en-us/library/c5tkafs1%28v=VS.85%29.aspx*.

[21] "Microsoft Developer Network - System.Reflection Namespace," *http://msdn.microsoft.com/en-us/library/system.reflection.aspx*.

[22] "Microsoft Developer Network - Assemblies," *http://msdn.microsoft.com/en-us/library/hk5f40ct%28v=vs.71%29.aspx*.

[23] E.W. Dijkstra, "On the Role of Scientific Thought," *Selected Writings on Computing: A Personal Perspective*, 1982.

[24] W. Schult and a Polze, "Aspect-oriented programming with C# and .NET," *Proceedings Fifth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. ISIRC 2002*, pp. 241-248.

[25] R.E. Filman and D.P. Friedman, "Aspect-oriented programming is quantification and obliviousness," *Workshop on Advanced Separation of Concerns*, Citeseer, 2000.

[26] F. Steimann, "The paradoxical success of aspect-oriented programming," *ACM SIGPLAN Notices*, vol. 41, Oct. 2006, p. 481.

[27] R. Laddad, *AspectJ in action*, Manning, 2003.

[28] M. Stoerzer and S. Hanenberg, "A Classification of Pointcut Language Constructs," *SPLAT'05: Workshop on Software-Engineering Properties of Languages and Aspect Technologies*, 2005.

[29] K. Böllert, "On weaving aspects," *Proceedings of the Workshop on Object-Oriented Technology*, Springer-Verlag, 1999, p. 301–302.

[30] P. Hudak, "Languages Evolution , and Application of Functional Programming," *Computing*, vol. 21, 1989.

[31] T. Petricek and J. Skeet, *Real World Functional Programming*, 2010.

[32] "The Haskell Programming Language," *http://www.haskell.org/haskellwiki/Haskell*.

[33] "The Caml language," *http://caml.inria.fr/index.en.html*.

[34] M. Fogus and C. Houser, *The Joy of Clojure*, Manning Publications Co. Greenwich, CT, USA, .

[35] “Python Programming Language – Official Website,” *http://www.python.org/*.

[36] “The Scala Programming Language,” *http://www.scala-lang.org/*.

[37] “Microsoft Developer Network - LINQ,” *http://msdn.microsoft.com/en-us/library/bb308959.aspx*.

[38] “The Caml Language,” *http://caml.inria.fr/index.en.html*.

[39] “The CTO Corner - The F# Survival Guide ebook,” *The CTO Corner: http://www.ctocorner.com/fsharp/book/*.

[40] “The F # 2 . 0 Language Specification,” *Microsoft Corporation*, 2010.

[41] “Microsoft Developer Network - Lambda Expressions: The fun keyword,” *http://msdn.microsoft.com/en-us/library/dd233201.aspx*.

[42] J. Skeet, *C# In Depth*, Manning, 2008.

[43] “Microsoft Developer Network - Type Inference (F#),” *http://msdn.microsoft.com/en-us/library/dd233180.aspx*.

[44] “Microsoft Developer Network - Tuples,” *http://msdn.microsoft.com/en-us/library/dd233200.aspx*.

[45] W.D. Meuter, “Monads as a theoretical foundation for AOP,” *Technology*, pp. 1-6.

[46] B. Beckman, *Channel9 : Don’t fear the Monad*.

[47] “Microsoft Developer Network - Computation Expressions,” *http://msdn.microsoft.com/en-us/library/dd233182.aspx*.

[48] J. Richter, *CLR via C# 3*, Microsoft Press, 2010.

[49] J. Liberty, *Programming C#*, O’Reilly, 2005.

[50] B. Goldberg, “Functional programming languages,” *ACM Computing Surveys*, vol. 28, Mar. 1996, pp. 249-251.

[51] P. Wadler, “Theorems for free!,” *Proceedings of the fourth international conference on Functional programming languages and computer architecture*, ACM, 1989, p. 347–359.

[52] 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.

[53] M. Fowler, “Domain Specific Languages,” *http://www.martinfowler.com/bliki/DomainSpecificLanguage.html*.

[54] 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.

[55] K. Chen, S.-C. Weng, M. Wang, S.-C. Khoo, and C.-H. Chen, “Type-directed weaving of aspects for polymorphically typed functional languages,” *Science of Computer Programming*, vol. 75, Nov. 2010, pp. 1048-1076.

[56] http://www.openfoundry.org/of/projects/801, “AspectFun at the OpenFoundry,” *http://www.openfoundry.org/of/projects/801*.

[57] “Microsoft Developer Network - System.Text.RegularExpressions Namespace,” *http://msdn.microsoft.com/en-us/library/system.text.regularexpressions.aspx*.

[58] “C# Language Specification v4.0,” *Microsoft Corporation*, 2010.

[59] Microsoft Corporation, “The Policy Injection Application Block,” *http://msdn.microsoft.com/en-us/library/ff647463.aspx*.