Automasjon 2016

The analysis of C# to F#

Jostein Andreassen, Michael Blomli and Mikkel Eltervåg

Automation

12. May 2015

Prosjektrapport – Side 2

Universitetet i Tromsø
Institutt for ingeniørvitnskap og sikkerhet
9037 TROMSØ

Studieretning: Automasjon	**År:** 2016

Tittel: The analysis of C# to F#	Dato: 12. Mai Gradering:
Forfattere: • Jostein Andreassen • Michael Blomli • Mikkel Eltervåg	Antall sider: Vedlegg:
Fortrolighet:	
Veileder: **Puneet Sharma**	
Oppdragsgiver: **Serit IT Partner Tromsø**	Oppdragsgivers kontaktperson: Jonas Juselius

Sammendrag:
Stikkord: F#, C#, fungsjonell programmering, analyse, programmering

Project rapport - Page 2

Universitetet i Tromsø
Institutt for ingeniørvitenskap og sikkerhet
9037 TROMSØ

Study: Automation	**Year:** 2016

Title: The analysis of C# to F#	**Date:** May 12th
Authors: Jostein Andreassen Michael Blomli Mikkel Eltervåg	**Grading:** Open
	Pages: 45
	Attachments: 4

Summary

This report is written by 3 students from the final year in the Automation class at “UiT: University of Tromsø - Arctic university of Norway”.

The assignment is given by a company named Serit IT Partner from Tromsø. They wanted us to identify the benefits, flaws and our experience from learning a new programming language called F#. Today Serit mostly uses the famous programming language C# which is currently widely used around the world. The time spent writing code and debugging, and the stability in these languages is very valuable.

Preface

This thesis is written by graduates from the automation program at the department of engineering at UiT: University of Tromsø - The Arctic University of Norway. This thesis will be used by Serit to help them determine whether incorporating the F# programming language would be favourable for the company.

We chose this assignment because everyone in our group enjoys programming, the assignment had some database management and set up, and because it looked like an interesting project overall. Since we already had some experience in C#, we thought it would be fun to find the "pros and cons" of this relatively new programming language.

We have written the thesis in LaTeX which is a word processor and a document markup language. When we were working on the raw text we used Google documents so that we all could work together simultaneously on writing and editing the text for the thesis. On the main assignment we used Visual Studio (with C# and F# tools), Atlassian Sourcetree and Github to write and manage our code, and SQL Server Management Studio to handle the database. By using both C# and database we covered a wide area of our syllabus from the previous semester. We have learnt a lot and have had the privilege of testing out the new attractive programming language F# that may just be the future.

We want to give out a special thanks to Jonas Juselius and Jens Blomli at Serit for all the great support and advice given, on all the regular follow up meetings and by email. We also want to give a big thanks to our mentor Puneet Sharma for his great support and contribution to the assignment.

Contents

1 Introduction

1.1 Background

One of the biggest problems in modern application development is the rapidly growing complexity of all major software systems. This complexity makes it almost impossible to ensure quality and accuracy of the code. It also becomes harder and harder to make changes to existing code without introducing new errors. All these difficulties multiply when one attempts to utilize modern computers with many CPU (central processing unit) cores for increased performance.

The imperative object-oriented programming paradigm has been dominant in software development for over 20 years. In the imperative paradigm the state variables will be handled explicitly, which can quickly give too much complexity. The functional programming paradigm has been known since the 1930's, but has not been popular with professional developers because of the slightly lower (single core) performance and greater resource use. Today these obstacles are long gone, and functional programming is experiencing a new renaissance due to significantly better control over complexity and parallelism.

C# is meant to be a simple, modern, flexible and object oriented programming language. It is developed and maintained by Microsoft and is inspired by previous popular object-oriented languages like C++ and Java. F# is a hybrid language that supports both the familiar object-oriented method and functional programming. F# is also developed by Microsoft, and like C# also has access to Microsoft's .NET framework.

1.2 Problem for discussion

Programming in C# and F# works in different ways, they both have benefits and drawbacks. The main question is whether it is worth it for a company to change their main programming language, in other words if the benefits outweigh the drawbacks of implementing this new language. We have to look at what the company wants to achieve by making the change, and that boils down to making quality programs for a low price.

A modern IT company uses a lot of time developing, changing and fixing code. If we can use a programming language that takes less time to develop and at the same time works better without generating errors, cost saving could be achieved.

The programming language F# claims to be a solution to these problems by using fewer lines of code, be more simple to write, be easier to understand and have better error handling than other programming languages. Our task is to uncover whether those claims are true by answering the following questions:

- What are the benefits of switching from C# to F#?
- To what degree can we reduce the number of lines written in the program code?
- How much time is saved in the debugging stage?
- How much time is saved in the development of the code?

1.3 Formulations of objectives

Serit IT Partner want to find out if it is worth changing their main programming language from C# to F#. Our task is to do an analysis and make our own conclusion to help them make their decision. This is achieved by reaching the objectives presented below.

- Learn to program in F# to:
  - get a good enough understanding of how to write simple programs.
  - see how it is for a programmer to learn the new language.
  - get our own opinions about the language.
- Learn from translating a real program from Serit by:
  - finding the best translation method.
  - looking at the development process for F#.
  - learning how F# handles the databases.
  - looking at how F# deals with debugging and error handling.
- Learn from online research by looking at:
  - other people's opinions about F#.
  - how other companies take advantage of F#.
  - good code examples that others have written.
  - what are some good usages of F#.
- Write a complete analysis that contains information about:
  - development time.
  - readability and clarity.
  - debugging and error handling.
  - performance.
  - our experiences.
- Draw a conclusion to consider whether it is valuable for Serit to change their main programming language from C# to F#.

1.4 Different programming paradigms

There are many different ways of classifying programming languages and styles. One of the most common ways of classifying is imperative versus declarative programming. The distinction is that imperative programming states the order in which the operations execute, while declarative does not. Another important feature is if side effects are allowed or not. By this, we mean if a function or subroutine can change the state of other parts of the program. In imperative programming, this is allowed, while in functional programming, it is not. Finally, there are different ways of organizing programs and breaking down problems into smaller, reusable pieces. Procedural programming focuses on breaking down the program into functions (also commonly called subroutines). Object-oriented programming organizes the program by grouping together the functions (methods) with the state they are allowed to modify (properties).

Imperative Imperative programming is a paradigm where statements are used to tell the computer program what to do. It is the programmer's job to tell the computer how to solve the problem. If an imperative program were to make a person get a cup of coffee, it would have to list all the individual tasks they had to do, like stand up, walk to the coffee machine, place the cup, press the button and so on.

Declarative Declarative programming focuses instead on telling the computer program what the desired result is, and then the implementation of the programming language decides how to do it. A declarative program would simply state "Give me a cup of espresso". Examples of declarative programming include database query languages and functional programming.

Functional Functional programming is a form of declarative programming. Instead of using statements, like in imperative programming, functional programming uses expressions. An expression is a combination of one or more values, operators or functions that produce a resulting value when evaluated. A goal in functional programming is to limit side effects and mutable values. This means that if you call a function with the same arguments you should get the same result everytime. Thus, the output of a function should only rely on the inputs to that function, and not any other values or variables.

Procedural Procedural programming breaks programs down into subroutines or functions which take arguments as input, and produces an output or result that is sent back to the caller. Functions can be designed to be as generic as possible so they can be used again in many parts of the program, which means less repetition of code.

Object-oriented On the other hand, Object-oriented programming focuses on categorizing the entities in the program as different objects, that expose behavior and data using interfaces. Objects may contain both data and methods. Methods are functions that are only allowed to change the data of the object to which it belongs. This is a way of protecting against side effects that is called encapsulation.

1.5 F# programming language

F# is a modern multi-paradigm programming language with a focus on functional programming. It was originally invented as an implementation of the OCaml language (which comes from ML) in the .NET platform. The first stable version was F# 2.0 which came in 2010. F# is open source and cross platform. F# was influenced by a number of other languages. It gets it's object model from C#. Sequence expressions and computation expressions come from Haskell. Finally, the indentation-aware syntax was inspired by Python. F# began as a .NET language, working together with C# and .NET libraries. It can also be compiled to Android, Apple iOS, Linux, JavaScript, and GPU code.

F# is "functional first" which means that functional programming is the preferred go-to method for solving problems. However, unlike many other functional languages, F# works well with imperative and object-oriented programming where those styles are required. F# is statically typed and type-safe which means the compiler can optimize the code deeply to increase performance, which is not possible in dynamic languages where types do not need to be known during compilation. Despite this, since F# uses type inference (the compiler deduces what type a value has based on the context and data) it can be more expressive, like a dynamic language.

There are four main differences between F# and a standard imperative language such as C# or Java.

- Function-oriented rather than object-oriented
- Expressions rather than statements
- Algebraic types for creating domain models
- Pattern matching for control flow

Function oriented In F#, functions are First-Class Citizens. This means that functions can be passed as arguments, returned from functions, and assigned to values. Composition is the technique of making basic functions, then functions that use those functions, and so on. Functional code is by it's nature high level and declarative.

Expressions In functional languages, there are no statements, only expression. Every code chunk must always return a value. Rather than using a list of

statements that are executed in order, larger chunks of code are created by combining smaller chunks. There are no such thing as unassigned values, they must be assigned to when declared. Every possible branch of a control flow construct must be explicitly handled and must also return the same value. This makes code built from expressions safer than code using statements.

Algebraic types In F#, new compound types are built by combining existing types in two different ways. First, a combination of values from a set of types. These are called product types, and are similar to structs or record types from other languages. Second, a disjoint union, representing a choice of types. These are called sum types.

Pattern matching Imperative languages have many different constructs for control flow and looping, including if-then-else, switch and case, for, foreach and while. In F#, all of these can be replaced by pattern matching and the single keyword match. Conditionals such as if and switch are replaced by simple pattern matching, and loops are replaced by pattern matching and recursion. And finally, inheritance, a defining feature of object-oriented programming can be replaced by pattern matching on union types.

> “F# is unique amongst both imperative and declarative languages in that it is the golden middle road where these two extremes converge. F# takes the best features of both paradigms and tastefully combines them in a highly productive and elegant language that both scientists and developers identify with. F# makes programmers better mathematicians and mathematicians better programmers.”
>
> *From the foreword of the book Expert F#*

1.6 Project specification

1.6.1 Where Serit is now

- They have an ASP.NET Web application in C# where the user interface is based on ASP Web Forms. All code is written in English, as well as all the text in the user interface.
- Language support is dissolved in a separate library sCore.Translation which is called from the application and performs translation according to data recorded in a translation table.
- Translation tables are located in a SQL database.

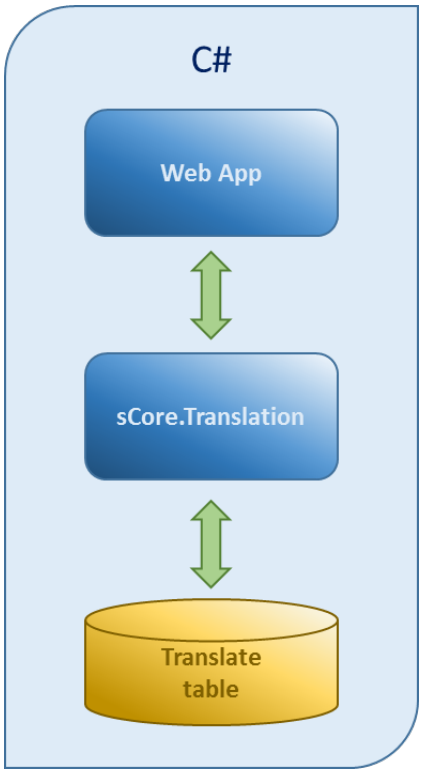

Figure 1: How the communication of Serit's sCore.Translation application looks now.

1.6.2 What Serit wants

- They want to have the existing translation library sCore.Translation developed as a separate library in the functional language F#. This should be able to be called from the present imperative program (C#) and from functional programs (F#).
- With the translation from C# to F# done, both languages and programming paradigms can be compared analytically. By this we can evaluate benefits (and possible disadvantages) with the functional paradigm in relation to an object-oriented imperative paradigm. The analysis will provide a better basis in the choice of programming language in future development projects.

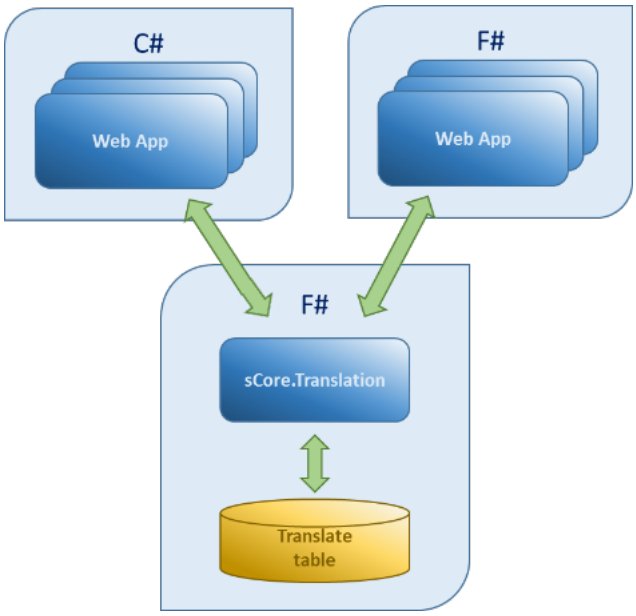

Figure 2: How they want the sCore.Translation application to communicate.

1.6.3 Method of translation

F# for fun and profit describes three levels of "sophistication" for porting code from C# to F#. The basic level is simply a direct port. Since F# supports imperative programming, it can be translated directly. At the intermediate level, the code is refactored to be fully functional. The advanced level takes advantage of F#'s data type system.

There are two paths to achieve this goal: Either by first porting to F# and then refactoring to functional code, or by converting to functional code in C# before porting that to F#.

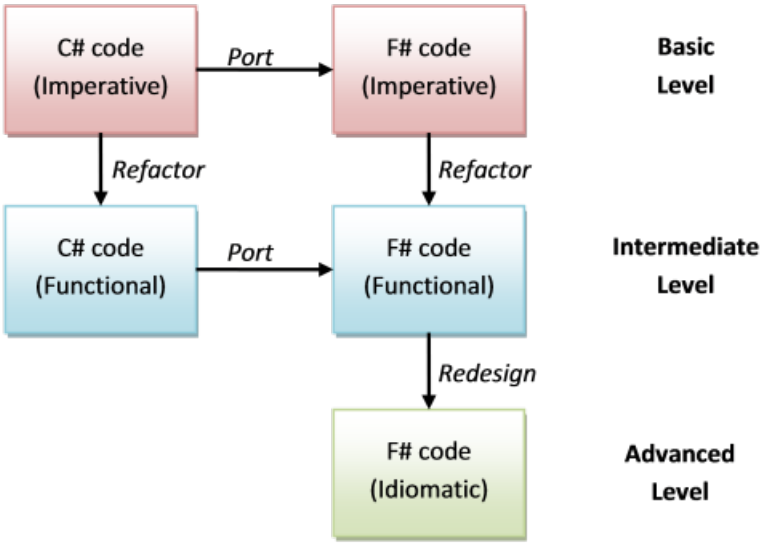

Figure 3: Method's of translating from C# to F#.

2 sTranslate

sTranslate is a library that is part of a bigger program. The purpose of the library is to take a phrase and some meta-data as arguments (inputs) and then use a language database to translate the phrase (output).

The library is written in C# and contains two major functions, one is designed to use when you are only going to translate one word or phrase, and the other one is designed to be better (faster) at doing multiple translations in succession. Even though the function works differently they have the same inputs and outputs, that means that they do the same thing if you look at it from the outside. You can see an illustration of the library in figure 4.

Our task from Serit is to take this C# code and translate it into F# with two different approaches. The first one we translate line for line and program in an imperative style, the same way that the C# program is coded. The other one is to refactor the program to a more functional approach, the way F# is meant to be written. After we have translated it we have to do look at how we can optimize the code to improve performance and readability while reducing the number of lines of code. Finally we will compare and analyze it against C# and look at performance, difficulty of programming and all other obvious differences.

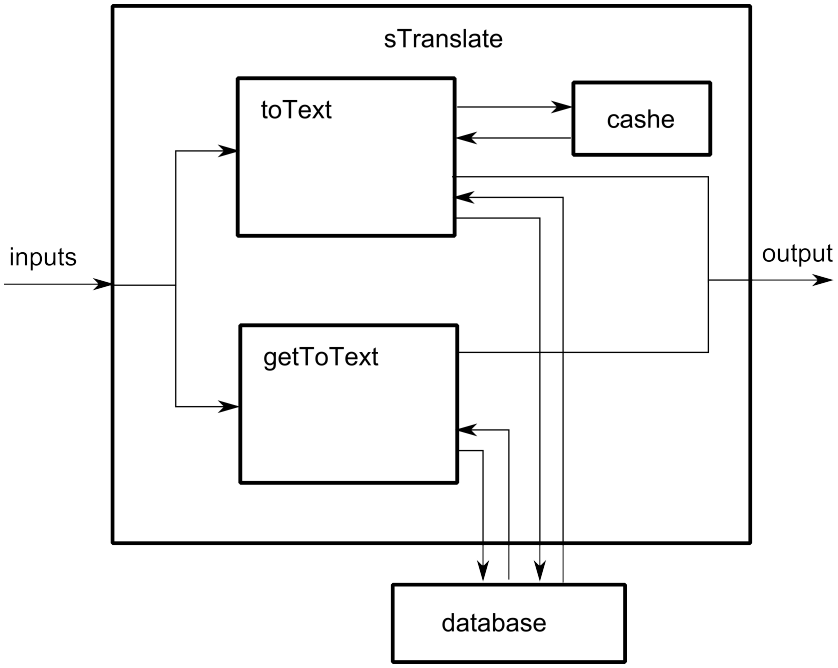

Figure 4: sTranslate library illustrated.

2.1 How it works

The sTranslate library is supposed to be used with both C# and F# programs. It is important that the library takes the same inputs and outputs in both languages. Figure 5 show this communication.
The main part of the library is the following functions: toText and getToText, as shown in figure 4. Both functions do the same thing but there are one major difference between them. In getToText you have to open the language database each time you want a translation, but in toText the database is cached (stored in program memory) so it does not need to reopen the database for each search.

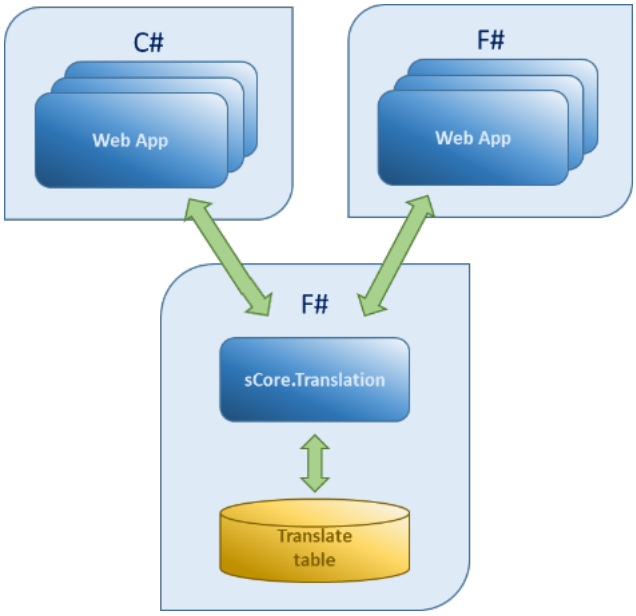

Figure 5: How Serit want the sCore.Translation application to communicate.

2.1.1 Inputs and Outputs

The sTranslate library should in theory work well for a pure functional approach. The output of the functions should only depend on the input, which will be the phrase to translate, it's metadata, and the contents of the database itself (which we can assume to no change while the search is happening).

A key part of the viability of F# will be to see how easy it is to compile a library that is compatible with C# programs. This should be trivial due to the close integration with the .NET system. A C# program should easily give inputs or take outputs from F# and the other way around. How we are going to use this feature can be seen in figure 5.

sTranslate is taking in 5 inputs:

- **fromText**, the phrase in English that the library is translating.
- **context**, information over how the phrase is used, examples: string, title, lable.
- **property**, category information about the phrase, examples: id, text, tooltlip...
- **criteria**, position of the phrase in a sentence, examples: startWith, endWith, Contains.

- **toLang**, the target language of the translation, examples: no, ge.

sTranslate gives only one output, which is the translated phrase (a string). All the inputs have to match up to the row in the database to send the output. In there is no match, fromText (the English phrase) will be sent back as output and if the search algorithm finds duplicate rows (multiple matches), only the first one will be selected.

2.2 Solution

When porting code from C# to F# there are two ways of refactoring the code into being functional. The first is to translate the C# code directly to F#, almost line by line, and then refactor the F# imperative code into functional code. The second way is to use the functional programming tools in C# to refactor the program, and then translating that to F#. We chose to use the first method.

In the figures below, we show an extremely simple code example of translating a piece of C# code to F# using an iterative approach. First the code is translated directly, then the `if-then-else` construct is replaced by pattern matching to make it more functional. It is important to remember that for such a simple code, the `if` construct would actually work fine in F# (see figure 7). This is because `if` constructs work a little different in F#: The entire if is itself an expression, and both (or all) conditional branches must evaluate to the same type. In this case, writing to the console evaluates to `unit()`. Since both branches evaluate to `unit()`, the expression is valid. Similarly, when using pattern matching for control flow as in figure 8, all patterns must evaluate to the same type, and just like `Console.WriteLine`, the `printfn` function evaluates to `unit()`.

```
1 if (a == b){
2   Console.WriteLine("equal");
3 }
4 else{
5   Console.WriteLine("not equal");
6 }
```

Figure 6: C# code example

```
1 if a = b then
2   Console.WriteLine("equal")
3 else
4   Console.WriteLine("not equal")
```

Figure 7: Direct translation code example

For a complete study of the different versions of the sTranslate program, the complete libraries with comments are documented in attachments (insert ref later). In this chapter, we will be taking a closer look at one part of the program, which

```
1 match a with
2       | b -> printfn "equal"
3       | _ -> printfn "not equal"
```

Figure 8: Functional approach code example

is the part that reads the database and provides a list that another function can use to perform the search.

2.2.1 C# code

A private variable is declared whose purpose it is to hold the contents of the database when the user calls the `toText` function. This variable is visible to all functions inside the `XltTool` class, but not to external users of the class. The `GetTranslations` function both mutates the `_translateColl` variable, and has it as a return value. Since the function is public, this makes the contents of `_translateColl` accessible to users. The `GetTranslations` function will open and read the database if either the `_translateColl` variable is empty, or the user has specified that it wants to read the database again.

```
1  private static List<Model.Translation>
       _translateColl = null;
2  public static List<Model.Translation>
       GetTranslations(bool reRead = false)
3  {
4      if (_translateColl == null || reRead == true)
5      {
6          using (var ctx = new Model.
               TranslationEntities())
7          {
8              _translateColl = (from xl in ctx.
                   Translation select xl).ToList();
9              if (_translateColl == null)
10                 _translateColl = new List<Model.
                       Translation>();
11             return _translateColl;
12         }
13     }
14     return _translateColl;
15 }
```

Figure 9: C# version of the GetTranslations function

2.2.2 F# imperative version

The imperative version retained the functionality of the C# version. The function is called and makes changes to a mutable value, and also returns that value to the caller. Note that we are still using mutable values even if we are working with F#.

```
1 let mutable _translateColl : List<Translation> = []
2 let GetTranslations (reRead : bool) =
3     if _translateColl = [] || reRead = true then
4         use db = dbSchema.GetDataContext(Settings.
              ConnectionStrings.DbConnectionString)
5         _translateColl <-
6         query {
7             for xl in db.Translation do
8                 select xl
9         } |> Seq.toList |> List.map toTranslation
```

Figure 10: Direct translation of the code

2.2.3 F# functional version

The functional version gets rid of the mutable value alltogether. The `GetTranslations` function uses the `FSharp.Data.TypeProviders` library, which gives typed access to the tables of the database. The Type Provider handles caching of the database automatically, so we can get rid of the "storage variable" that was used in the imperative version. The `GetTranslations` expression is evaluated once when it is needed, and then the data is there, ready to be used until the program ends.

```
1 let GetTranslations =
2     use db = dbSchema.GetDataContext(Settings.
        ConnectionStrings.DbConnectionString)
3     query {
4         for row in db.Translation do
5             select row
6     } |> Seq.toList |> List.map toTranslation
```

Figure 11: Refactored to functional code

2.3 Analysis

In this chapter we will take a look at the analysis of the module sTranslate. Looking back on how we used the two different methods of writing the F# code, the line by line translation and the functional approach, we can say that the direct translation was clearly not the way to go. It became much harder to do the actual programming in an object-oriented way, so much that we even ended up doing the direct translation without using classes at all (while the C# code of course did have classes). We found that by using the direct translation, we didn't really save many lines of code. We could also notice in the result a clear difference in the actual speed of the program, where the direct translation was inferior to the functional code. We get into this a little more later.

2.3.1 Performance

This chapter is not a demand from Serit, instead this was something we wanted to test out for ourselves, and is just a bonus for Serit if it proves to be better performance in F# vs C#.

2.3.2 Experiences

The experiences that we can draw from this assignment is that in F#, it's very easy to read the code and what it does, and that you almost don't need to comment the code. But the thing we appreciated the most was the direct response from the compiler and Microsoft's Intellisense technology in real time while writing the code. At first we thought that this was a bit unfavourable since you always saw errors made in code with the red lines everywhere. However, as the language is very strict about types and such, one is forced to root out many errors before the program even is able to run. Because programming with pure functions eleminates side effects and state changes, the programmer won't have to deal with unexpected errors later while running the program, which could make it misbehave or even crash. Hence we grew to really enjoy this feature. It makes the programmer put a little more time in thinking about what they want the code to do, but they will spend less time dealing with problems later.

Since all the group members have object-oriented programming background, the main struggle we experienced was our tendency to get back to old habits and thinking in the object-oriented way of programming. The new thinking in functional programming was harder than we imagined. We acknowlegded that it's probably easier for a person who have never done programming before to learn F# then a person coming from a background with object-oriented programming at least in some aspects. Of course, that is not completely true because a skilled

programmer can quickly find patterns in the code and structure, this varies from programmer to programmer. We had to let go of a lot of hard learned coding and try and think “out of the box” to be able to write “decent” F# code.

We realised that everytime we went ack to the written code, improvements could be made both in terms of code length but also with regards to quality. F# is mainly a function based programming but it actually allows imperative and object-oriented programming in some way. Since it’s integrated into Visual Studio and the .NET framework there are a lot of features inherited from C#. That will allow the user to use for loops, if-else constructs, mutable values, etc. Still, we found that all of these thing were replacable when programming functionally.

For Michael, the biggest struggle of this experience was that he kept finding himself getting back in think in object-oriented “thinking”. It’s hard to think in a more functional approach.

2.3.3 Lines of code

When it breaks down to how many lines of code there is in each project we can right away see big differences.

Here is a short outline on how many files and lines each project got:

Short outline			
Language	Lines of code	Files	Projects
C#	280	3	2
F# imperative style	170	4	1
F# functional style	145	2	1

3 Online research

3.1 Testimonials

To get a better understanding of how companies could benefit using F# do we have to look at what experiences programmers and companies have with F#. The internet contains a lot of testimonials where people write their personal experiences with F#. It is important to note that those companies have already done the choice of using F# in their applications. This means that they have found F# useful and therefore more likely to talk possessively about the language. Here we have taken some of those testimonials to take a closer look.

Simon Cousins Simon is a developer at UK-based Power Company. He uses both C# and F# to write his applications. What we can learn from what he have written is that F# and C# can be used together. He uses C# to write the client and server components and F# to write the calculation engine. This is a really smart usage of both programming languages as he uses the best language to each task.

He does also write that he is still waiting for the first bug from a bug to came in from a delivered F# project and that this is not the case with C#. Here we can see the effect of the strong type system in the F# compiler.

> “I have now delivered three business critical projects written in F#. I am still waiting for the first bug to come in.”
>
> *Simon Cousins*

Michael Newton Michael is a Senior Developer at a development company named 15below. Historically they have written their code in a mix of C# and VB.NET. They uses F# for components where the language strength’s shine, without discarding or rewrite the existing code.

What we can learn from this is that is really well integrated with the .NET framework. This makes it really easy to use with the code that the company already have written. That can be really beneficial for Serit IT Partner as their code today is written in C# that supports .NET.

> “We would recommend F# as an additional tool in the kit of any company building software on the .NET stack.”
>
> *Michael Newton*

3.2 Code examples

We have done some research on the web to find some concrete code examples to highlight some differences in these two coding languages.

3.2.1 Basic from 1 to N square function

F# A lot of our research comes from the web page fsharpforfunandprofit, and there they highlight a simple task where they sum the squares from 1 to N. We can see how the code could look like if it was written in F# in figure 12.

```
1 // define the square function
2 let square x = x * x
3
4 // define the sumOfSquares function
5 let sumOfSquares n =
6     [1..n] |> List.map square |> List.sum
```

Figure 12: Basic from 1 to N square function made in F#.

Here they made 2 functions, where one uses the other function as an output on it's own. The |> is used to send the output from one function and send it as an input to the other. If we break this down to what it does it's basically, first an function is made that is called square, it simply takes itself and multiply it with itself. Then an other function is made sumOfSquares, it makes a loop from 1 to n, then puts them into a list and each element is used with the square function. Then it sums the list up and sends it back as the output. We didn't need to do any type declarations, because the compiler figured out that it has to be an integer. That is because the input in sumOfSquares is used to create an array, and that needs an integer. If we now try to write a piece of code that would send for example a floating point number to that function, the program wouldn't even compile, and you would save yourself some debugging time.

C# Now if we take a look at how the code would look like in classic C# (non-functional) style of a C-based language. First we take a look at the code seen in figure 13 below.

```
1  public static class SumOfSquaresHelper
2  {
3      public static int Square(int i)
4      {
5          return i * i;
6      }
7
8      public static int SumOfSquares(int n)
9      {
10         int sum = 0;
11         for (int i = 1; i <= n; i++)
12         {
13             sum += Square(i);
14         }
15         return sum;
16     }
17 }
```

Figure 13: Basic from 1 to N square function made in C#.

So this code basically does the same thing. The interesting part here is that what F# could to with 2-3 lines of code, the basic C# would to in about 16 (to be fair most of the lines is just and) that is pretty impressive. However you could make this much more compact but that would make it much harder to read, since this is the standard that is mostly used.

So to highlight the differences of these two code examples. First it is that the F# code is much more compact and still is very readable (almost self explained). The F# code didn't have any type declarations (f.ex C# have to do int sum = 0, before sum is even used). The last thing is that F# can be developed interactively. With that we mean that we can test the functions right away and check if we get back what we want or not without having to go in and out of functions with the use on the interactive window that F# has.

C# LINQ Today many C# programmers uses LINQ, a much more modern tool with C# that in someway lets C# be more functional.

So let's take a look at how the code would look like using the C# LINQ extension, we can see this in figure 14 below.

```
1 public static class FunctionalSumOfSquaresHelper
2 {
3     public static int SumOfSquares(int n)
4     {
5         return Enumerable.Range(1, n)
6             .Select(i => i * i)
7             .Sum();
8     }
9 }
```

Figure 14: Basic from 1 to N square function made in C# LINQ.

We took the basic C# code from 16 lines down to 9 lines. This almost took down the C# code to half, but still not near the F# code. Also here we can see that the C# code is bound to use the curly braces and periods and semicolons. The code also still needs to be declared to be able to use variables, that we don't need to do in F#.
What we can conclude with this short example is that F# in this case was superior in many ways, more easy to read, more compact and you can easily check the gained outputs through the interactive window.

3.3 Rotate a tuple

Here we are gonna take a look at one more example, this time very short and easy function. The function simply is used to change the order of a tuple (a tuple is set of values, f.ex. cordinates in an aksis).

F# First lets take a look on how we would to this in F#, we can se this in figure below.

```
1 let rotate (x,y,z) = (z,x,y)
```

Figure 15: Rotate a tuple example made in F#

C# Since we can't just rearrange the order easily on a tuple in C#, we have to do it in a little different approach. This is shown in figure below.

```
1 Tuple<V,T,U> Rotate(Tuple<T,U,V> t)
2 {
3     return new Tuple<V,T,U>(t.Item3, t.Item1, t.Item2
          );
4 }
```

Figure 16: Rotate a tuple example made in C#

So firstly here we have to define that its a tuple first, and then give it a name and in this case it's Rotate. When we are returning the new values we have to tell the compiler what tuple we are gonna return as well as ref to each item in the new tuple.

This is a good example to show how easy things can be done in F#.

4 Complete analysis

In this chapter we will take a look at the complete analysis of the previous chapters.

4.1 F# compared to C#

4.2 Development time

4.3 Readability and clarity

Readability and clarity is really important when writing code. The reason for this is that it is much easier to find and remove bugs and add future content. If the programmer does not need to use a lot of time simply to understand what the code is supposed to do. Poorly written code can cause a programmer to spend a lot of time familiarizing themselves with the code before they can do changes to it.

For the most part it is up to the programmer to write readable and clear code, but the programming language can have a lot to do with helping the programmer in this matter.

So the question is how does F# do in this matter? How difficult is it to write readable and clear code. How would a bad or new programmer's code look like? And how readable is a perfect written code?

4.3.1 Indentation and code structure

F# is a whitespace sensitive programming language, this means that indentation do have a meaning and will be read when you compile the code. When you are using indentation the compiler will read the indented lines as a code block that belonging to the code above. You can see this illustrated by code in figure 17.

```
1 let f =    // Line 2-5 belongs to this block
2   let x = // Line 3 is the sub code of this line
3     5+5
4   let y=1
5     x+y
6 let a = f+2 // On the same indentation as line 1,
    starting a new block
```

Figure 17: Example of how indentation works in F#.

That means that to some extent the F# language force the programmer to write code that have good structure in the code. This can make it easier to understand and read the code even if you don't know it beforehand.

For an inexperienced programmer this can be a little confusing at fist. But as this is a really good practice to learn early on it can help the inexperienced programmer write better code.

In indentation and code structure, F# really does a good jobb when it comes down to readability and clarity.

4.3.2 File structure

In most programming languages the file structure have nothing to do in how the program is compiled or run. F# does this a little different, all the files have to be in a specific order to run the program correct.

Whenever you call a function you have to make sure that the function is written before the call in the code is made. And since all the files are read in order you have to make sure that if the function is in another file it has to be placed above the current file in the file structure.

The most common way to organize the files in a project is to have all the files in the same folder. This makes it more transparent and easy to order the code. This way of organization can work really well for small and medium sized projects, but for really big projects it may be better to organize in multiple folders.

A good practice when organising the files is to only have one module per file and make the functions inside correspond to the file name.

4.3.3 Similarity to other coding languages

ML Programming Language Even though F# has taken inspiration from a lot of different modern programming languages it still have a lot of roots to ML. This is a old and not that popular programming language. This makes the syntax very different to how syntaxes normally looks like today, and can make the syntax hard to read and write to any programmer that are not used to ML or other similar programming languages.

If we take a close look at figure 18 and 19 we see a lot of similarity in how the programs work. In the F# example we can read match, that is a way of doing pattern matching. The ML code does also use pattern matching with a small change in the syntax. Line 1 in the ML code we can see that if the function gets a 0 as an input the output will be 1, the same thing happens at line 3 in the F# example. Line 2 in the ML code tells us that if the input is an int the output will be `n * factorial (n-1)` and the same thing happens in F# line 4 if the input is not 0.

```
1 fun factorial 0 = 1
2   | factorial (n:int) = n * factorial (n-1)
```

Figure 18: ML code example of a recursion function (function that's call itself) that are calculating factorials. Example from Carnegie Mellon University[3]

```
1 let rec factorial n =
2     match n with
3     | 0 -> 1
4     | _ -> n * factorial (n - 1)
```

Figure 19: F# code example of a recursion function (function that's call itself) that are calculating factorials.

Python One programming language a lot of programmers can agree on being easy to read and understand is Python. Python is a language with a high level of abstraction, which means it acts as a "middle man" between the nitty gritty stuff that happens inside the computer, and abstract ideas. The syntax of Python has been designed to remove curly braces and semicolons that are used in languages like C and Java, using indentations instead to separate code blocks. Example of Python code can bee seen in figure 20.

```
1 def factorial(n):
2     if n == 0:
3         return 1
4     else:
5         return n * factorial(n-1)
```

Figure 20: Python code example of a recursive function (a function that calls itself) calculating factorials. Python also has a factorial function built in to the Math library. The equivalent F# version of this program can be seen in figue 19.

At a first glance Python and F# can look similar because both languages is a white space sensitive programming language (Explained on page 4.3.1). But if you take a closer look at the programming languages they really works different, the main reason for this is that F# is meant to be a functional programming language and Python is not, even though it have support for it.

Python is one of the programming languages that the creators of F# have been using as a inspiration when making F#. It is not hard to see the reason for this, because Python has an easy readability as an programming language. Both of them can be really self explanatory and forces the programmer to write the code in a good structured way. This have made F# more easy read and understand for programmers that looks at programs others have written.

Assembly One commonly used example for a programming language that is not easy to read or understand is Assembly. The reason that Assembly can be hard to read and understand for some people is that the programmer is working directly with CPU instructions and memory addressing. Even mundane tasks like moving a value to a registry must be explicitly programmed. In figure 21 you can see an example of a "Hello world!" program. Most of the other programming languages does this in only one line . You can see an example of this in the F# example in figure 22.

```
1  section .text
2       global _start    ;must be declared for linker (
           ld)
3  _start:               ;tells linker entry point
4       mov edx,len      ;message length
5       mov ecx,msg      ;message to write
6       mov ebx,1        ;file descriptor (stdout)
7       mov eax,4        ;system call number (sys_write)
8       int 0x80         ;call kernel
9
10      mov eax,1        ;system call number (sys_exit)
11      int 0x80         ;call kernel
12
13 section .data
14 msg db 'Hello, world!', 0xa  ;string to be printed
15 len equ $ - msg       ;length of the string
```

Figure 21: Assembly "Hello world!" code example from tutorialspoint.com [2].

```
1 printfn "Hello, world!"
```

Figure 22: F# "Hello world!" code example.

It is maybe a little unfair to compare F# with Assembly because Assembly is a really low level programming language. But there are two points that we can get out from this comparison and that is firstly that F# is a easier language to write and understand, but this comes with the cost of less control over very basic computing that is important for really good optimization.

4.3.4 Conclusion readability and clarity

After analysing the readability and clarity of the F# programming language we can divide how easy it would be for a programmer to learn F# into 4 groups.

Inexperienced For a person that is new to programming it will take some time to learn programming regardless of the programming language. But F# can be a good place to start. It can be easy to read and understand simple code from only looking at the code and you don't already have any mindset that can stand in your way of understanding.

Object-oriented A programmer that have background from object-oriented programming languages like C++, C#, Java and Visual Basic can have a problem getting used to F#. F# is a functional programming language and works very different to object-oriented language.

Dynamical typed Python and Ruby are examples of dynamical typed languages. Those languages have a similar syntax to F# but they are works a little different in how it is typed. It can be easier for programmers that are used to dynamical typed programming languages to learn F# than it is for programmers with object-oriented background.

Functional Programmers that have background from functional languages like Haskell, OCaml or Standard ML for them F# will be really familiar. They will probably find that F# to be really easy to read, understand and write. The reason for this is that F# is a functional language itself.

4.4 Debugging and error handling

4.5 Performance

5 Conclusion

The main goal of this project was to give Serit IT Partner a insight in the programming language F#, to help them decide if they are going to start using F# as an replacement for what they are using today that is C#. To let them get this insight we have done a lot of research and written a thorough analysis.

For the research we have translated a concrete example of a program that Serit IT Partner have written in C#. This program have helped us understand functional programming and get a good understanding of the good and bad sides of F# in comparison to C#.

The 4 months that we been working with F# is not long enough to understand and analyse every aspect of the language. Therefore we have used a lot of online resources like code examples and testimonials from both programmers and companies that have been using F# to get to our conclusion.

Our final conclusion is that F# should not completely replace C# as their main programming language today.

Both C# and F# is integrated with the .net framework, this makes communication between the languages really good. C# and F# can be used to do the same things, but both languages have their own areas they are superior in. We think that programs should be written in the programming language that is best suitable to the task.

Serit IT Partner should at first start up one small team that specialise in programming in F#. This team is going to write programs where F# is superior. The team have to work close with the C# teams to create good seamless programs. This can later be expanded, if they have success with it.

F# should be used as extra tool for creating better content and not as a replacement for the language they use today.

6 Reference list

References

[1] F# for fun and profit
https://fsharpforfunandprofit.com

[2] Tutorialspoint
http://www.tutorialspoint.com/assembly_programming/

[3] Carnegie Mellon School of Computer Science
https://www.cs.cmu.edu/~rwh/introsml/core/recfns.htm

[4] Antonio Cisternino, Adam Granicz and Don Syme, *Expert F*, 2007

[5] F# Software Foundation
http://fsharp.org/testimonials/

7 Attachments