Automasjon 2016

The analysis of C# to F#

Jostein Andreassen, Michael Blomli and Mikkel Eltervåg

Automation

12. May 2015

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# **Authors:** Jostein Andreassen Michael Blomli Mikkel Eltervåg	**Date:** May 12th **Classification:** Open **Pages:** 74 **Attachments:** 4

Mentor: Puneet Sharma	
Contracting entity: Serit IT Partner Tromsø	**Contact person:** Jonas Juselius

Summary:
In this report do we an analysis of a new programming language called F#. This analysis is written for Serit IT Partner to look at the possibility that F# can be a replacement for the programming language they are using today that is C#.

To help us write this analysis we have translated and optimized a program that Serit have written. This is to get an understanding of the benefits and drawbacks of changing from C# to F#. We have also looked at code examples that others have written and looked at what experiences that company's that already have changed language has.

Our conlution is that F# should not completely replace C# as their main programming language. F# should be used together with C# to get the benefits of both languages.

Keywords:
F#, C#, Functional programming, analysis, programming, SQL

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. Serit IT Partner wanted us to test out if a change of main programming language from C# to F# would have an impact on time spent on coding, reduce the number of lines of code and would improve stability.

We found that we could clearly see a difference regarding the numbers of lines of code and the performance in the application that was given to us. Indeed, F# is better (or perhaps, more convenient) than C# in some cases. F# is a programming language which is very good at handling large amounts of data, and is also good with concurrency (dealing with multiple things at the same time). Due to the nature of the language, large operations on data can be broken down into smaller tasks and calculations can be made without fear of unintended side effects.

We started out at thinking that F# was out there to replace C#, however that was not the case. F# is made as a tool that should work together with C# seamlessly to make great applications. Both programs are used in the .NET framework and by that they work very well together. F# doesn’t need to completely replace C#, instead it should be implemented there where it is suited best. Based on our results Serit should consider starting a small team and use F# where it’s best and easy to implement and use C# where its better. Then they could gradually expand the F# team if they see the need for it.

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 **6**
- 1.1 Background . . . 6
- 1.2 Problem for discussion . . . 7
- 1.3 Formulations of objectives . . . 8
- 1.4 Different programming paradigms . . . 9
- 1.5 F# programming language . . . 11
- 1.6 Project specification . . . 13
  - 1.6.1 Where Serit is now . . . 13
  - 1.6.2 What Serit wants . . . 14
  - 1.6.3 Method of translation . . . 15

2 sTranslate **16**
- 2.1 How it works . . . 17
  - 2.1.1 Inputs and Outputs . . . 18
- 2.2 Solution . . . 19
  - 2.2.1 Basic code examples . . . 20
  - 2.2.2 The GetTranslations function in 3 different ways . . . 21
  - 2.2.3 Detailed look at F# sTranslate . . . 24
  - 2.2.4 Parallel version of sTranslate . . . 26
- 2.3 Analysis . . . 28
  - 2.3.1 Performance . . . 28
  - 2.3.2 Experiences . . . 28
  - 2.3.3 Lines of code . . . 29

3 Additional research **30**
- 3.1 Testimonials . . . 30
- 3.2 Code examples . . . 32
  - 3.2.1 Basic from 1 to N square function . . . 32
- 3.3 Rotate a tuple . . . 35

4 Complete analysis **36**
- 4.1 Readability and clarity . . . 37
  - 4.1.1 Indentation and code structure . . . 38
  - 4.1.2 File structure . . . 39
  - 4.1.3 Similarity to other coding languages . . . 40
  - 4.1.4 Conclusion readability and clarity . . . 43
- 4.2 Debugging and error handling . . . 44
- 4.3 Performance . . . 44
- 4.4 Economic aspects . . . 45

4.4.1 Training and recruiting employees . . . 46
4.4.2 Development time . . . 47
4.4.3 Customer satisfaction . . . 48
4.4.4 Conclusion economic aspects . . . 49

5 Conclusion **50**

6 Reference list **51**

A Appendices **52**
A.1 sTranslate from Serit (C#) . . . 52
A.1.1 Enums.cs . . . 52
A.1.2 XltEnums.cs . . . 59
A.1.3 XltTool.cs . . . 60
A.2 sTranslate Direct (F#) . . . 66
A.2.1 Enums.fs . . . 66
A.2.2 XltEnums.fs . . . 67
A.2.3 XltTool.fs . . . 68
A.3 sTranslate (F#) . . . 70
A.3.1 XltEnums.fs . . . 70
A.3.2 XltTool.fs . . . 71
A.4 sTranslate Parallel (F#) . . . 73
A.4.1 XltTool.fs . . . 73

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 type. 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.

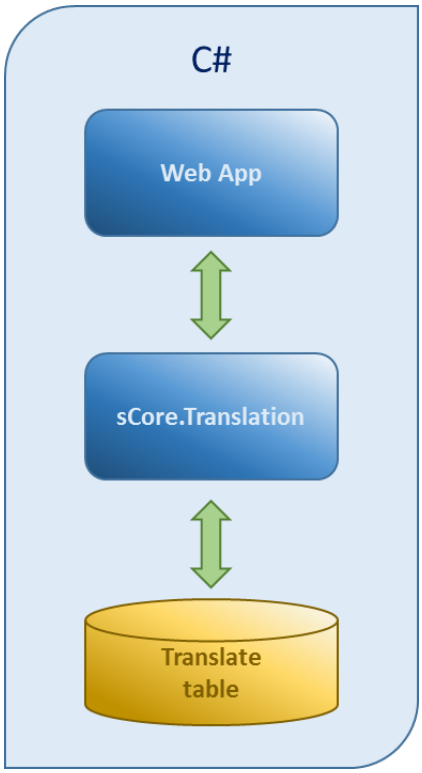

Figure 1: How the communication of Serit's sCore.Translation application looks now.

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.

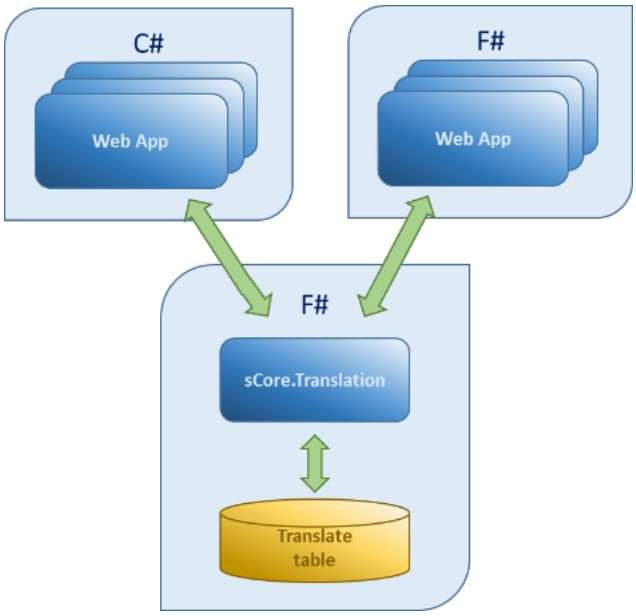

Figure 2: How they want the sCore.Translation application to communicate.

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#.

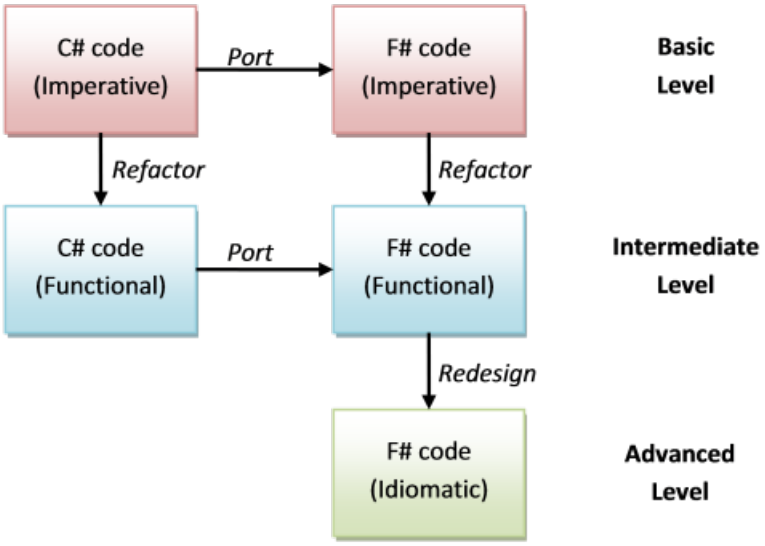

Figure 3: Method's of translating from C# to F#.

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.

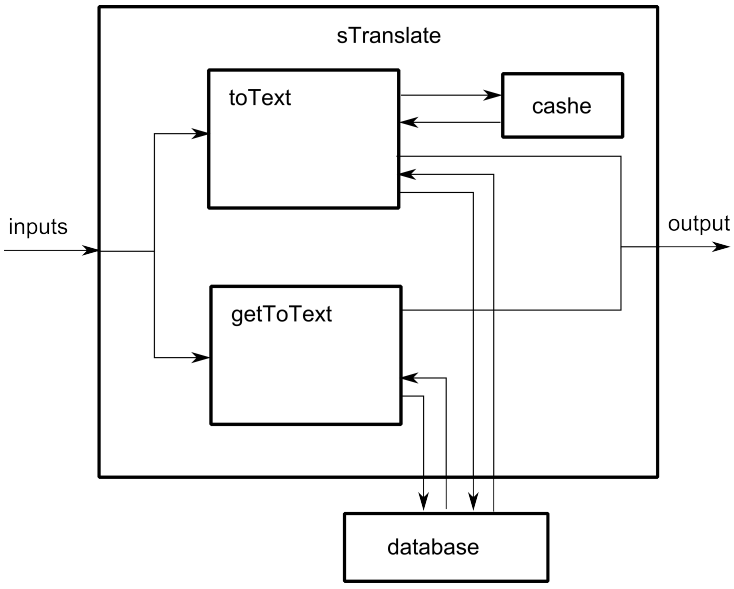

Figure 4: sTranslate library illustrated.

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.

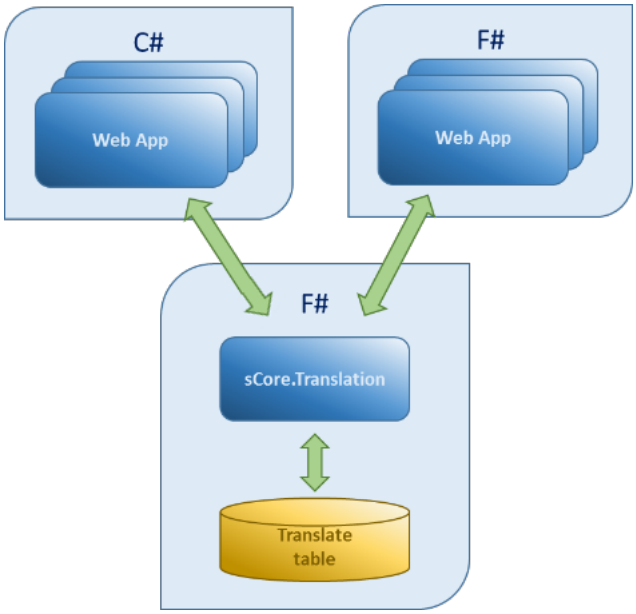

Figure 5: How Serit want the sCore.Translation application to communicate.

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.

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()`.

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 first be taking a closer look at one part of the program, which is the part that reads the database and provides a list that another function can use to perform the search. Then we will examine some parts of the final F# program and see how and why it differs from the C# program. Finally, we tried to go even further beyond, and made a version of the program that, while it doesn't quite interface with Serit's web application, does show off some of the power and elegance of the F# language. This version of the library uses high performance data structures and parallel programming to translate a large number of queries very quickly.

2.2.1 Basic code examples

In the figures below are shown code chunks with identical functionality, one written in C# and two different ways of doing it in F#. This is not from the sTranslate program, just a simple demonstration example.

```
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

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

Figure 8: Functional approach code example

2.2.2 The GetTranslations function in 3 different ways

C# version from Serit 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

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

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

2.2.3 Detailed look at F# sTranslate

There are a few key points where the functional F# library differs from the C# version:

- The F# SQL Type Provider is used instead of Entity Framework.
- GetToText takes Option string arguments instead of Enums.
- GetTranslations produces an immutable list.
- The ToText function was removed as it was obsolete.

F# SQL Type Provider Type providers are the F# way of accessing external data. They provide types, properties and methods for use in programs. When introducing data sources, they need to be represented in a way that makes sense for the programming language. A common way is to use code generators, but the disadvantage of this is that the generated code must be replaced if the target changes (a column in a database table is removed, added or renamed, for instance). Type providers are written targeting a specific service, for example databases, web resources or spreadsheets. The SQL Type Provider for F# provides typed access to SQL databases, and when using Visual Studio, even connects to the database in real time when programming. This allows the programmer to explore the database structure by "dotting into" the database objects.

Option type arguments The C# version of the program used enumerations to enforce a set of correct arguments for `Property` and `Criteria`. In the F# version, the functions perform the same check, but allows the program to continue in the case that a wrong value slipped through the cracks. By using option types, the functions can return a `None` value if the enum is not found. It will be up to the caller of the function to figure out what to do with this value. This is checked in the `GetToText` function, and if either of the arguments have the None type, the search is cancelled.

Getting the database The `GetTranslations` function was changed to an expression that is evaluated exactly once during program execution. It opens a connection to the database and reads the contents of the Translation table. The relevant data is copied into a `Translation` data type, and a list is built of all the translations. The `reRead` functionality of the C# program is lost, but it can be safely assumed that the contents of the Translation table will not change during program execution. The data is required to be immutable for the pure functional approach to work.

Removal of obsolete function In the early versions of this program, we had both the `ToText` and the `GetToText` functions. However, they did the database searches in exactly the same amount of time, both when doing a single search, and doing many. This defeated the purpose of having one function that specialized in doing single searches. It was decided to cut the `ToText` function since it was obsolete.

2.2.4 Parallel version of sTranslate

We made this version of the program just to test out the possibilities of the language. Compared to the regular F# version of sTranslate, this program ran three times faster on our test computer. Parallel programs gain most speed when it can distribute very time-consuming calculations to multiple processing cores. Since the sTranslate library just performs a database search, a significant amount of the processing power is spent distributing tasks between the cores. Still, this program showcases how easy and streamlined parallel programming is in F#. We also refined the syntax and structure of the program and took away unnecessary code to make it more idiomatic F#.

The parallel version of sTranslate differs from the regular version in the following ways:

- All the search arguments were collected in a type.
- The function and value names were changed to `camelCase`.
- A high speed data structure to cache the database was used.
- The inputs and outputs were changed to accommodate parallel computing.
- The entire Enum system was removed.

Search type We replaced the five input arguments of the regular F# program with a record type that holds these arguments as members. The type is public so that users of the library can build lists of this type. This is used for the input to the parallel search function. This also helps the readability of the program, since the search function only needs to accept one argument, instead of five.

Function and value names The identifiers for functions and values were changed to `camelCase` to conform with the suggested naming conventions for F# code. Type names and members were left as `PascalCase`. The public function GetToTextAsync is also `PascalCase`.

FlatList data structure The database is copied into a very fast data structure called FlatList. The FlatList is implemented as an array, but unlike arrays, it is an immutable data structure (which we want, to make our program purely functional). Since it is implemented as an array, it is faster than a regular list, except when one needs to append an element to it. Since we always construct the FlatList from a sequence, we don't need fast appending. We only care about iterating through the list and reading the items really fast.

Input and output change As mentioned before, the input arguments to the search function were collected in a record type. The users of the library only have the Search type and the GetToTextAsync function visible to them. The user must aggregate a list of Search items and provide it as input to GetToTextAsync. The output is a list of string which are the translations. The elements of the output list is in exactly the same order as the corresponding searches in the input, even if they might have been computed in a different order.

Removal of Enum module The Enum checking that was part of the original C# program didn't feel like proper F# programming. An invalid search in the database will now simply return the untranslated phrase, just as in the original program. We decided to remove the Enum module and keep all the members of the Search type as strings (and not Enums or option strings) for the sake of simplicity, clarity and speed. If we were to implement a similar feature to the one in the C# program, we would use discriminated unions.

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 Additional 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, UK-based Power Company*

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, 15below*

Jan Erik Ekelof Jan Erik is Head IT-architect and lead developer at Handelsbanken. In 2009 he got the mission to implement a Real-time Counter-party Risk system. Their plan was to write the program in C#, but they did a small proof-of-concept with F#. When using F# they got a significant productivity boost and now they have moved almost entirely into F#.

In one example application they wrote contains 35 000 to 40 000 lines of F# code and an equal amount of C# code. They found that at least 80% of the functionality was in the F# part of the application.

This shows how much more compact an C# application can get when translated into F#. Less code tend to lead to shorter development time, less bugs and easier readability.

> “Our experience shows us that the number of code lines shrinks with a ratio of 1/2 to 1/4 by just porting functionality from C# to F#.”
>
> *Jan Erik Ekelof, Handelsbanken*

Employee at Bayard Rock Bayard Rock is a company where they are trying to finding new approaches towards anti-money-laundering. Before they adopted F# there where often months of turnaround time between development of an idea and actually testing. Now that they are using F# they can often see their ideas come to life in just days.

They find that this makes them really competitive in their field. Being one of the first to try out new technology can be risky but often with high reward. For Bayard Rock this have really beneficial, they can deliver projects faster and cheaper than their competitors and therefore corner the market.

> “The benefits of functional programming have given us a great advantage over our slow moving competitors. Little do they know that it’s largely thanks to our secret weapon, F#.”
>
> *Employee at Bayard Rock*

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

4.1 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?

4.1.1 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.

4.1.2 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.

4.1.3 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 4.1.1). 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.

4.1.4 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.

4.2 Debugging and error handling

4.3 Performance

4.4 Economic aspects

To be one of the first to try out new technology can be a high risk high reward scenario. Even though F# is a relativity new programming language it have been tried out with success multiple of times.

Changing programming language can be a costly a affair but in the long run it can be really beneficial and sometimes even necessary to survive in the marked. Here we look at the pros and cons of changing language in a economic aspect.

4.4.1 Training and recruiting employees

Training employees Learning a new programming language can take a lot of time from where the programmers could be making applications that generate income. This is especially true when it comes to F# if the programmers is not used to functionally programming beforehand. Programmers time is really valuable and shod therefore be used wisely.

Recruiting new employees In some parts of the word it can be hard to find and recruit new employees with programming background. In Tromsø, Norway where Serit IT Partner have their office is there specially hard to find good experienced programmers. Since F# is a new language, it will be very difficult to find new programmers that don't need training before they start working. This can make changing language costly especially if the company needs to expand.

4.4.2 Development time

Time equals money, if a programmer can write a application in less time that can help a company save a lot of money.

From what we have earlier discussed in Lines of code at page 29 and Testimonials at page 30 can we easy see that the code length in F# is superior in comparison to C#. The main reason for this is that F# offers a way to get around using loops. It does this by substituting the loops with sequence functions. One other reason for the shorter code is that F# uses a shorter and more simple syntax than C#. This can shorten down the development time significantly just because the programmer have less code to write before a project is finished.

The shorter development time is major selling point for F# as you can write applications with less code. This can save programmers huge amount of time and in extension save a company for a lot of money. But this shorter development time is only true if the company have good experienced F# developers employed. A new and inexperienced developer can take a long time getting up to speed in a new language, this can be exceptionally true in F# if the developer never have programmed functionally before.

4.4.3 Customer satisfaction

Costumers pay a high price for quality and shorter delivery time. In programming quality often refer the speed of the program and lack of bugs. From what we have found both from our code and research, can we see that F# does this exceptionally well. This and what we can see from development time can lead to a very high customer satisfaction.

4.4.4 Conclusion economic aspects

The two main economic problems of switching programming language from C# to F# is training employees and recruiting new employees. Both of those problems can be gone in a long term perspective if F# becomes popular.

F# does it really well in both saving money in development time and creating good customer satisfaction. So a company that have the experience in F# can have a great advantage over competitors that don't convert over to F#. This means that if F# becomes popular and more developers learn the language there can be a huge gain for a company that develop in F#.

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/

A Appendices

A.1 sTranslate from Serit (C#)

A.1.1 Enums.cs

```
1  using System;
2  using System.Configuration;
3
4  namespace sTranslate.Tools
5  {
6    public sealed class Enums
7    {
8          /// ------------------------------------------------
9          /// Description for struct.
10         /// ------------------------------------------------
11
12         public struct FileStruct
13         {
14             public string Flags;
15             public string Owner;
16             public string Group;
17             public bool IsDirectory;
18             public string CreateTime;
19             public string ParentDirectory;
20             public string Name;
21         }
22
23         /// ------------------------------------------------
24         /// Description for Enums.
25         /// ------------------------------------------------
26
27         /// <summary>
28         /// Insert placements
29         /// </summary>
30         public enum InsertPlacements
31         {
32             Before = 1,
33             After
34         }
35         public static InsertPlacements ToInsertPlacement(string value)
36         {
37             return (InsertPlacements)Enums.GetEnumState(typeof(
                   InsertPlacements), value);
38         }
39
40         /// <summary>
41         /// FileListStyle
42         /// </summary>
43         public enum FileListStyles
44         {
45             UnixStyle,
46             WindowsStyle,
47             Unknown
48         }
49         public static FileListStyles ToFileListStyle(string value)
50         {
51             return (FileListStyles)GetEnumState(typeof(FileListStyles), value
                   );
52         }
53
```

```
54
55          /// <summary>
56          /// Archive directory types
57          /// </summary>
58          public enum ArchiveDirectoryTypes
59          {
60              Archive = 0,
61              Skip,
62              Error
63          }
64          public static ArchiveDirectoryTypes ToArchiveDirectoryType(string
                value)
65          {
66              return (ArchiveDirectoryTypes)GetEnumState(typeof(
                    ArchiveDirectoryTypes), value);
67          }
68
69
70          /// <summary>
71          /// Debug view levels
72          /// </summary>
73          public enum DebugViewLevels
74          {
75              None = 0,
76              All,
77              Copy,
78              Exclude
79          }
80          public static DebugViewLevels ToViewLevel(string value)
81          {
82              return (DebugViewLevels)GetEnumState(typeof(DebugViewLevels),
                    value);
83          }
84
85          /// <summary>
86          ///
87          /// </summary>
88          public enum FilterTypes
89          {
90              Exclude = 0,
91              Select
92          }
93          public static FilterTypes ToFilterType(string value)
94          {
95              return (FilterTypes)GetEnumState(typeof(FilterTypes), value);
96          }
97
98          /// <summary>
99          /// Confiuration policys
100         /// </summary>
101         public enum ConfigPolicys
102         {
103             Undefined = 0,
104             File,
105             Database
106         }
107         public static ConfigPolicys ToConfigPolicy(string value)
108         {
109             return (ConfigPolicys)GetEnumState(typeof(ConfigPolicys), value);
110         }
111
112         /// <summary>
```

```
113         ///
114         /// </summary>
115         public enum AddinTypes
116         {
117             Undefined = 0,
118             Agent,
119             Guard
120         }
121         public static AddinTypes ToAddinType(string value)
122         {
123             return (AddinTypes)GetEnumState(typeof(AddinTypes), value);
124         }
125 
126         /// <summary>
127         /// Thread startup types
128         /// </summary>
129         public enum StartupTypes
130         {
131             Manual = 0,
132             Automatic,
133             Disabled
134         }
135         public static StartupTypes ToStartupType(string value)
136         {
137             return (StartupTypes)GetEnumState(typeof(StartupTypes), value);
138         }
139 
140         /// <summary>
141         ///
142         /// </summary>
143         public enum HexValueLength
144         {
145             TwoBytes = 2,
146             TreeBytes = 3
147         }
148         public static HexValueLength ToHexValueLengt(string value)
149         {
150             return (HexValueLength)GetEnumState(typeof(HexValueLength), value
                    );
151         }
152 
153         /// <summary>
154         ///
155         /// </summary>
156         public enum BitStates
157         {
158             Low = 0,
159             High = 1
160         }
161         public static BitStates ToBitState(string value)
162         {
163             return (BitStates)GetEnumState(typeof(BitStates), value);
164         }
165 
166         /// <summary>
167         ///
168         /// </summary>
169         public enum ByteOrderInDataWords
170         {
171             HighLow,
172             LowHigh
173         }
```

```
174         public static ByteOrderInDataWords ToByteOrderInDataWord(string value
                )
175         {
176             return (ByteOrderInDataWords)GetEnumState(typeof(
                    ByteOrderInDataWords), value);
177         }
178 
179 
180         /// <summary>
181         ///
182         /// </summary>
183         public enum MessageFormats
184         {
185             Fixed = 0,
186             FieldSeparated,
187             FixedByteArray,
188             FieldSeparatedByteArray,
189             Xml
190         }
191         public static MessageFormats ToMessageFormat(string value)
192         {
193             return (MessageFormats)GetEnumState(typeof(MessageFormats), value
                    );
194         }
195 
196 
197         /// <summary>
198         /// The state of a message
199         /// </summary>
200         public enum MessageStates
201         {
202             Disabled = 0,   // The message is disabled (Will not be handled
                    at all)
203             Created,        // The message is crested and not processed
204             Processed,      // The message is processed successfully
205             Delivered,      // The message is sent successfully to the
                    receiver node (communication acknowledgment).
206             Acknowledged,   // The message is acknowledged from the remote
                    application (remote application acknowledgment).
207             Failed,         // The message processing is failed
208             Skipped         // The message prosessing is skipped
209         }
210         public static MessageStates ToMessageState(string value)
211         {
212             return (MessageStates)GetEnumState(typeof(MessageStates), value);
213         }
214 
215         /// <summary>
216         /// The states of a thread
217         /// </summary>
218         public enum ProcessingStates
219         {
220             Initializing = 0,
221             Idle,           // The thread is in idel (wait) state.
222             Processing,     // The thread is processing some task.
223             Shutdown        // The thread is in shutdown state.
224         }
225         public static ProcessingStates ToProcessingState(string value)
226         {
227             return (ProcessingStates)GetEnumState(typeof(ProcessingStates),
                    value);
228         }
```

```
229
230         /// <summary>
231         ///
232         /// </summary>
233         public enum ObjectLoadStates
234         {
235             UnLoaded = 0,
236             Loaded
237         }
238         public static ObjectLoadStates ToObjectLoadState(string value)
239         {
240             return (ObjectLoadStates)GetEnumState(typeof(ObjectLoadStates),
                        value);
241         }
242
243
244         /// <summary>
245         ///
246         /// </summary>
247         public enum DriverRunStates
248         {
249             Normal = 0,
250             SlowMotion
251         }
252         public static DriverRunStates ToDriverRunState(string value)
253         {
254             return (DriverRunStates)GetEnumState(typeof(DriverRunStates),
                        value);
255         }
256
257
258         /// <summary>
259         ///
260         /// </summary>
261         public enum LogFilePolicies
262         {
263             Service = 0,
264             Component
265         }
266         public static LogFilePolicies ToLogFilePolicy(string value)
267         {
268             return (LogFilePolicies)GetEnumState(typeof(LogFilePolicies),
                        value);
269         }
270
271
272         /// <summary>
273         ///
274         /// </summary>
275         public enum SortOrders
276         {
277             Ascending = 0,
278             Descending
279         }
280         public static SortOrders ToSortOrder(string value)
281         {
282             return (SortOrders)GetEnumState(typeof(SortOrders), value);
283         }
284
285
286         /// <summary>
287         ///
```

```
288         /// </summary>
289         public enum CleanActions
290         {
291             Copy = 0,
292             Delete,
293             Move,
294         }
295         public static CleanActions ToCleanAction(string value)
296         {
297             return (CleanActions)GetEnumState(typeof(CleanActions), value);
298         }
299 
300         /// <summary>
301         /// UserActions
302         /// </summary>
303         public enum UserActions
304         {
305             Undefined = 0,
306             Add,
307             Change,
308             Delete
309         }
310         public static UserActions ToUserAction(string value)
311         {
312             return (UserActions)GetEnumState(typeof(UserActions), value);
313         }
314 
315         /// <summary>
316         /// Options for Tool.Time() methode
317         /// </summary>
318         public enum TimeOptions
319         {
320             Now = 0,
321             LastDayOfThisMonth,
322             FirstDayOfPreviousMonth
323         }
324         public static TimeOptions ToTimeOption(string value)
325         {
326             return (TimeOptions)GetEnumState(typeof(TimeOptions), value);
327         }
328 
329 
330         /// <summary>
331         /// ------------------------------------------------
332         ///
333         /// Get enumeration state
334         ///
335         /// ------------------------------------------------
336         /// </summary>
337         /// <param name="type"></param>
338         /// <param name="value"></param>
339         /// <returns></returns>
340         public static Object GetEnumState(System.Type type, string value)
341         {
342             if (value == null || value == "")
343                 throw new Exception(string.Format("{0}:GetEnumState:
                        Mandatory value parameter must be entered in call", type.
                        ToString()));
344 
345             foreach (string s in Enum.GetNames(type))
346             {
347                 if (value.ToLower() == s.ToLower())
```

```
348                         return Convert.ChangeType(Enum.Parse(type, s), type);
349                 }
350
351                 throw new Exception(string.Format("{0}:GetEnumState: Enumeration
                        don't contain value '{1}'", type.ToString(), value));
352             }
353         }
354     }
```

A.1.2 XltEnums.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace sTranslate.Tools
8  {
9      public class EnumsXlt
10     {
11         public enum PropertyTypes
12         {
13             Id = 1,
14             Text,
15             ToolTip,
16             Page
17         }
18         public static PropertyTypes ToPropertyType(string value)
19         {
20             return (PropertyTypes)Enums.GetEnumState(typeof(PropertyTypes),
                   value);
21         }
22
23         public enum Criterias
24         {
25             None = 0,
26             Exact,
27             StartWith,
28             EndWith,
29             Contains
30         }
31         public static Criterias ToCriteria(string value)
32         {
33             return (Criterias)Enums.GetEnumState(typeof(Criterias), value);
34         }
35
36         /// <summary>
37         /// This is only to use as helper to get type checked result.
38         /// Ex: This will return exact "HtmlAnchor" and you avoids typing
                   careless mistakes.
39         /// Ex: string str = Contexts.HtmlAnchor.ToString();
40         /// </summary>
41         public enum Contexts
42         {
43             HtmlAnchor = 1,
44             String,
45             Title
46         }
47         public static Contexts ToContext(string value)
48         {
49             return (Contexts)Enums.GetEnumState(typeof(Contexts), value);
50         }
51     }
52 }
```

A.1.3 XltTool.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6  using Model = sTranslate.Model;
7
8  namespace sTranslate.Tools
9  {
10     /// <summary>
11     /// Translate :
12     ///     This class contains functionality for translation from one
           language to another.
13     ///     The translation definitions are read from the Translate table.
14     ///     The default source language code (FromLang) is 'en' for English.
15     /// </summary>
16     public class XltTool
17     {
18         private static List<Model.Translation> _translateColl = null;
19         public static string FromLanguageCode = "en";
20
21         /// <summary>
22         /// Get all translation entities into collection
23         /// </summary>
24         /// <param name="reRead">ReRead from database</param>
25         /// <returns>Translation collection</returns>
26         public static List<Model.Translation> GetTranslations(bool reRead =
               false)
27         {
28             if (_translateColl == null || reRead == true)
29             {
30                 using (var ctx = new Model.TranslationEntities())
31                 {
32                     _translateColl = (from xl in ctx.Translation select xl).
                           ToList();
33                     if (_translateColl == null)
34                         _translateColl = new List<Model.Translation>();
35                     return _translateColl;
36                 }
37             }
38             return _translateColl;
39         }
40
41         /// <summary>
42         ///
43         ///     GetToText methode returen the translated string, if defined
               int the Translate table.
44         ///     If the fromText is not found, the value of fromText is
               returned unchanged.
45         ///     Multiple definitions for the same source string can be
               registered, and in case,
46         ///     the property and context fields must be used to separate them
               .
47         ///
48         /// </summary>
49         /// <param name="fromText"> The string to be translated </param>
50         /// <param name="context"> The context the definition is aimed for.
               </param>
51         /// <param name="propType"> The row type. This is to be used to
               pinpoint the search string if multiple definitions exists </param
```

```
        >
52      /// <returns> The translated string contained in the ToText field </
            returns>
53      public static string GetToText(EnumsXlt.Criterias criteria, string
            fromText, EnumsXlt.PropertyTypes property, string context, string
             toLanguageCode = "no")
54      {
55          if (fromText.Trim() == "")
56              return "";
57
58          List<Model.Translation> coll = new List<Model.Translation>();
59          using (var ctx = new Model.TranslationEntities())
60          {
61              if (string.IsNullOrEmpty(toLanguageCode))
62                  toLanguageCode = "no";
63
64              // Do search by criteria
65              coll = (from xl in ctx.Translation
66                      where xl.Criteria.ToLower() == criteria.ToString().
                            ToLower() &&
67                            xl.FromLang == FromLanguageCode &&
68                            xl.FromText == fromText &&
69                            xl.Property.ToLower() == property.ToString().
                                ToLower() &&
70                            xl.Context.ToLower() == context.ToLower() &&
71                            xl.ToLang == toLanguageCode
72                      select xl).ToList();
73
74              return (coll != null && coll.Count > 0) ? coll.First().ToText
                    : fromText;
75          }
76      }
77
78      /// <summary>
79      ///
80      ///     ToText methode returen the translated string, if defined int
            the Translate table.
81      ///     If the fromText is not found, the value of fromText is
            returned unchanged.
82      ///     Multiple definitions for the same source string can be
            registered, and in case,
83      ///     the context and rowtype fields must be used to separate them.
84      ///
85      /// </summary>
86      /// <param name="fromText"> The string to be translated </param>
87      /// <param name="context"> The context the definition is aimed for.
            </param>
88      /// <param name="propType"> The row type. This is to be used to
            pinpoint the search string if multiple definitions exists </param
            >
89      /// <returns> The translated string contained in the ToText field </
            returns>
90      public static string ToText(EnumsXlt.Criterias criteria, string
            fromText, EnumsXlt.PropertyTypes property, string context, string
             toLanguageCode = "no")
91      {
92          if (fromText.Trim() == "")
93              return "";
94
95          GetTranslations();
96
97          if (string.IsNullOrEmpty(toLanguageCode))
```

```
98                  toLanguageCode = "no";
99
100             // Serach collection
101             foreach (Model.Translation xl in _translateColl)
102             {
103                 if (xl.Criteria.ToLower() == criteria.ToString().ToLower() &&
104                     xl.FromLang == FromLanguageCode &&
105                     xl.FromText == fromText &&
106                     xl.Property.ToLower() == property.ToString().ToLower() &&
107                     xl.Context.ToLower() == context.ToLower() &&
108                     xl.ToLang == toLanguageCode)
109                     return xl.ToText;
110             }
111             return fromText;
112         }
113
114         public static string ToText(string fromText, EnumsXlt.PropertyTypes
                property, string context, string toLanguageCode = "no")
115         {
116             if (fromText.Trim() == "")
117                 return "";
118             string toText = fromText;
119             List<Model.Translation> coll;
120             coll = XltTool.GetXltByKeys(property, context, EnumsXlt.Criterias
                .None);
121             foreach (Model.Translation tr in coll)
122             {
123                 switch (tr.Criteria.ToString().ToLower())
124                 {
125                     case "startwith":
126                         if (toText.ToLower().StartsWith(tr.FromText.ToLower()
                            ))
127                             toText = toText.Replace(tr.FromText, tr.ToText);
128                         break;
129                     case "endwith":
130                         if (toText.ToLower().EndsWith(tr.FromText.ToLower()))
131                             toText = toText.Replace(tr.FromText, tr.ToText);
132                         break;
133                     case "exact":
134                         if (toText.ToLower() == tr.FromText.ToLower())
135                             toText = toText.Replace(tr.FromText, tr.ToText);
136                         break;
137                     case "contains":
138                         if (toText.ToLower().Contains(tr.FromText.ToLower()))
139                             toText = toText.Replace(tr.FromText, tr.ToText);
140                         break;
141                     case "none":
142                         if (tr.FromText == "*")
143                             toText = tr.ToText;
144                         else
145                             if (toText.ToLower().Contains(tr.FromText.ToLower
                                ()))
146                                 toText = toText.Replace(tr.FromText, tr.
                                    ToText);
147                         break;
148                     default:
149                         // No translation
150                         break;
151                 }
152             }
153
154             return toText;
```

```
155         }
156
157         /// <summary>
158         /// Get Translation entity collection by keys
159         /// </summary>
160         /// <param name="propertyType"></param>
161         /// <param name="context"></param>
162         /// <param name="toLanguageCode"></param>
163         /// <returns></returns>
164         public static List<Model.Translation> GetTranslationByKeys(EnumsXlt.
                Criterias criteria, EnumsXlt.PropertyTypes property, string
                context = null, string toLanguageCode = "no")
165         {
166             List<Model.Translation> coll = new List<Model.Translation>();
167             using (var ctx = new Model.TranslationEntities())
168             {
169                 if (string.IsNullOrEmpty(toLanguageCode))
170                     toLanguageCode = "no";
171
172
173                 coll = (from xl in ctx.Translation
174                         where xl.FromLang == FromLanguageCode && xl.ToLang ==
                                toLanguageCode &&
175                               xl.Criteria.ToLower() == criteria.ToString().
                                    ToLower() &&
176                               xl.Property.ToLower() == property.ToString().
                                    ToLower() &&
177                               (context == null || (context != null && xl.
                                    Context.ToLower() == context.ToLower()))
178                         select xl).ToList();
179
180                 return (coll != null) ? coll : new List<Model.Translation>();
181             }
182         }
183
184         /// <summary>
185         ///
186         /// </summary>
187         /// <param name="criteria"></param>
188         /// <param name="propertyType"></param>
189         /// <param name="context"></param>
190         /// <param name="toLanguageCode"></param>
191         /// <returns></returns>
192         public static List<Model.Translation> GetXltByKeys(EnumsXlt.
                PropertyTypes property, string context, EnumsXlt.Criterias
                criteria = EnumsXlt.Criterias.None, string toLanguageCode = "no")
193         {
194             List<Model.Translation> newColl = new List<Model.Translation>();
195             if (context == null)
196                 return newColl;
197
198             GetTranslations();
199             using (var ctx = new Model.TranslationEntities())
200             {
201                 if (string.IsNullOrEmpty(toLanguageCode))
202                     toLanguageCode = "no";
203
204                 // Serach matching entities
205                 foreach (Model.Translation xl in _translateColl)
206                 {
207                     if (((criteria == EnumsXlt.Criterias.None) || (criteria
                            != EnumsXlt.Criterias.None && xl.Criteria.ToLower()
```

```
                        == criteria.ToString().ToLower())) &&
208                     xl.FromLang == FromLanguageCode &&
209                     xl.Property.ToLower() == property.ToString().ToLower
                            () &&
210                     xl.Context.ToLower() == context.ToLower() &&
211                     xl.ToLang == toLanguageCode)
212                 {
213                     newColl.Add(xl);
214                 }
215             }
216         }
217         return newColl;
218     }
219 
220     /// <summary>
221     ///
222     /// </summary>
223     /// <param name="text"></param>
224     /// <param name="tr"></param>
225     /// <returns></returns>
226     public static bool IsCriteriaMet(string text, Model.Translation tr)
227     {
228         // string[] arr = Tool.RxReplace(c.Text, "(<.*?>)|([\n]+)|( 
                ;)", "").Split(new string[] { "\r", ":", "  " },
                StringSplitOptions.RemoveEmptyEntries);
229         switch (EnumsXlt.ToCriteria(tr.Criteria))
230         {
231             case EnumsXlt.Criterias.Exact:
232                 if (text.IndexOf(">" + tr.FromText) >= 0 && text.IndexOf(
                        tr.FromText + "<") >= 0)
233                     return true;
234                 break;
235             case EnumsXlt.Criterias.StartWith:
236                 if (text.IndexOf(">" + tr.FromText) >= 0)
237                     return true;
238                 break;
239             case EnumsXlt.Criterias.EndWith:
240                 if (text.IndexOf(tr.FromText + "<") >= 0)
241                     return true;
242                 break;
243             case EnumsXlt.Criterias.Contains:
244                 if (text.IndexOf(tr.FromText) >= 0)
245                     return true;
246                 break;
247         }
248         return false;
249     }
250 
251     /// <summary>
252     /// Get nestled exception message
253     /// </summary>
254     /// <param name="prompt">Optionaly, the message prompt</param>
255     /// <param name="ex"></param>
256     /// <returns></returns>
257     public static string ExceptionMsg(Exception ex, bool InnerOnly =
            false)
258     {
259         if (ex == null)
260             return "";
261 
262         Exception e = ex;
263         if (InnerOnly)
```

```
264             {
265                 while (e.InnerException != null)
266                     e = e.InnerException;
267                 return e.Message;
268             }
269             else
270             {
271                 string msg = "";
272                 while (e != null)
273                 {
274                     msg += (msg != "") ? ";" + e.Message : e.Message;
275                     e = e.InnerException;
276                 }
277                 return msg;
278             }
279         }
280     }
281 }
```

A.2 sTranslate Direct (F#)

A.2.1 Enums.fs

```
1  namespace sTranslate_direct
2  module Enums =
3      open System
4
5      //////////////////////////
6      // Get enumeration state
7      let GetEnumState myType (value : string) =
8
9          // Exits the function if the input is empty or null
10         if value = null || value = "" then
11             eprintf "%s:GetEnumState: Mandatory value parameter must be
                   entered in call" (myType.ToString()) |> ignore
12             exit 1
13
14         try
15             // Filters a string array and finds the correct Enumeration
16             let enumName = Enum.GetNames(myType) |> Seq.filter (fun x -> x.
                   ToLower() = value.ToLower()) |> Seq.head
17             // Returns the object of the specified type with the correct
                   state
18             Enum.Parse(myType, enumName)
19         with _ ->
20             // Exits if there was no match
21             eprintf "%s:GetEnumState: Enumeration don't contain value '%s'" (
                   myType.ToString()) value |> ignore
22             exit 1
```

A.2.2 XltEnums.fs

```
1  namespace sTranslate_direct
2  module XltEnums =
3      open Enums
4
5      // Defines the types
6      type PropertyTypes =
7          | Id      = 1
8          | Text    = 2
9          | ToolTip = 3
10         | Page    = 4
11     type Criterias =
12         | None      = 0
13         | Exact     = 1
14         | StartWith = 2
15         | EndWith   = 3
16         | Contains  = 4
17
18     // Creates an object of type PropertyType from the input string
19     let ToPropertyType value =
20         let myType = typeof<PropertyTypes>
21         GetEnumState myType value
22
23     // Creates an object of type Criterias from the input string
24     let ToCriteria value =
25         let myType = typeof<Criterias>
26         GetEnumState myType value
```

A.2.3 XltTool.fs

```
1  namespace sTranslate_direct
2  module XltTool =
3      open System
4      open System.Data
5      open System.Data.Linq
6      open FSharp.Data.TypeProviders
7      open Microsoft.FSharp.Linq
8      open FSharp.Configuration
9      open XltEnums
10     open Model
11
12     // Sets the database connection string
13     type Settings = AppSettings<"App.config">
14     type dbSchema = SqlDataConnection<ConnectionStringName = "
           dbConnectionString">
15
16     // Language to translate from
17     let FromLanguageCode = "en"
18
19     // Copy the contents of a database row into a record type
20     let toTranslation (xlt : dbSchema.ServiceTypes.Translation) =
21         {
22             Id = xlt.Id
23             FromText = xlt.FromText
24             ToText = xlt.ToText
25             Context = xlt.Context
26             Property = xlt.Property
27             Criteria = xlt.Criteria
28             FromLang = xlt.FromLang
29             ToLang = xlt.ToLang
30         }
31
32     let mutable _translateColl : List<Translation> = []
33
34     let GetTranslations (reRead : bool) =
35         if _translateColl = [] || reRead = true then
36             use db = dbSchema.GetDataContext(Settings.ConnectionStrings.
                   DbConnectionString)
37             _translateColl <-
38             query {
39                 for xl in db.Translation do
40                     select xl
41             } |> Seq.toList |> List.map toTranslation
42         _translateColl
43
44     ////////////////////////////////////////
45     //     GetToText function returns the translated string, if defined inn
           the Translate table.
46     //     If the fromText is not found, the value of fromText is returned
           unchanged.
47     //     Multiple definitions for the same source string can be registered,
            and in case,
48     //     the property and context fields must be used to separate them.
49     let GetToText criteria (fromText : string) property (context : string)
           toLanguageCode =
50         if fromText.Trim() = "" then ""
51         else
52
53         // If toLanguageCode is not valid, sets it to default "no"
54         let toLang =
```

```
55              match toLanguageCode with
56              | null | "" -> "no"
57              | _ -> toLanguageCode
58
59          // Open a connection to the database
60          use db = dbSchema.GetDataContext(Settings.ConnectionStrings.
                DbConnectionString)
61
62          // Do search by criteria
63          let coll =
64              query {
65                  for xl in db.Translation do
66                  where (xl.Criteria.ToLower() = criteria.ToString().
                        ToLower() &&
67                               xl.FromLang = FromLanguageCode
                                                    &&
68                               xl.FromText = fromText
                                                          &&
69                               xl.Property.ToLower() = property.ToString().
                                    ToLower() &&
70                               xl.Context.ToLower() = context.ToLower()
                                                &&
71                               xl.ToLang = toLang)
72                  select xl }
73
74          // Returns the ToText field from the first row in coll, if empty it
                returns the original text
75          try
76              let result = Seq.head coll
77              result.ToText
78          with _ -> fromText
79
80      ////////////////////////////////////////
81      let ToText criteria (fromText : string) property (context : string)
            toLanguageCode =
82          if fromText.Trim() = "" then ""
83          else
84
85          // If toLanguageCode is not valid, sets it to default "no"
86          let toLang =
87              match toLanguageCode with
88              | null | "" -> "no"
89              | _ -> toLanguageCode
90
91          GetTranslations false |> ignore
92
93          let mutable result = None
94          let mutable i = 0
95          while result = None && i < _translateColl.Length do
96              let xl = _translateColl.[i]
97              if  xl.Criteria.ToLower() = criteria.ToString().ToLower() &&
98                  xl.FromLang = FromLanguageCode && xl.FromText = fromText &&
99                  xl.Property.ToLower() = property.ToString().ToLower() &&
100                 xl.Context.ToLower() = context.ToLower() &&
101                 xl.ToLang = toLanguageCode
102             then
103                 result <- Some xl.ToText
104             i <- i + 1
105
106         match result with
107             | Some string -> result.Value
108             | None -> fromText
```

A.3 sTranslate (F#)

A.3.1 XltEnums.fs

```
1  namespace sTranslate_fs
2
3  module XltEnums =
4      open System
5
6      type PropertyTypes =
7          | Id       = 1
8          | Text     = 2
9          | ToolTip  = 3
10         | Page     = 4
11     type CriteriaTypes =
12         | None       = 0
13         | Exact      = 1
14         | StartWith  = 2
15         | EndWith    = 3
16         | Contains   = 4
17
18     // Helper function to give the first element of a sequence, if it
           contains something
19     let checkHead (s : seq<'a>) =
20         match Seq.isEmpty s with
21         | true -> None
22         | false -> Some <| Seq.head s
23
24     // Get enumeration state
25     let getEnumState myType (value : string) =
26
27         // Filters a string array and finds the correct Enumeration
28         Enum.GetNames(myType)
29         |> Seq.filter (fun x -> x.ToLower() = value.ToLower())
30         |> checkHead
31
32     // Creates an object of type PropertyType from the input string
33     let toProperty value =
34         getEnumState typeof<PropertyTypes> value
35
36     // Creates an object of type Criterias from the input string
37     let toCriteria value =
38         getEnumState typeof<CriteriaTypes> value
```

A.3.2 XltTool.fs

```
1  namespace sTranslate_fs
2
3  module XltTool =
4      open System
5      open System.Data
6      open System.Data.Linq
7      open FSharp.Data.TypeProviders
8      open Microsoft.FSharp.Linq
9      open FSharp.Configuration
10     open XltEnums
11
12     // Get typed access to App.config to fetch the connection string later
13     type Settings = AppSettings<"App.config">
14
15     // SQL Type Provider. Give it a dummy connection string to satisfy the
           compiler.
16     type dbSchema = SqlDataConnection<ConnectionStringName = "
           dbConnectionString">
17
18     // Record type so that we can cache the database to memory and not have
           the data context go out of scope
19     type Translation = {
20             Id : int
21             FromText : string
22             ToText : string
23             Context : string
24             Property : string
25             Criteria : string
26             FromLang : string
27             ToLang : string
28         }
29
30     // Copy the contents of a database row into a record type
31     let toTranslation (xlt : dbSchema.ServiceTypes.Translation) = {
32             Id = xlt.Id
33             FromText = xlt.FromText
34             ToText = xlt.ToText
35             Context = xlt.Context
36             Property = xlt.Property
37             Criteria = xlt.Criteria
38             FromLang = xlt.FromLang
39             ToLang = xlt.ToLang
40         }
41
42     // Database only supports translating from english for now
43     let FromLanguageCode = "en"
44
45     // Copies the database to memory
46     let GetTranslations =
47         use db = dbSchema.GetDataContext(Settings.ConnectionStrings.
               DbConnectionString)
48         query {
49             for row in db.Translation do
50                 select row
51         } |> Seq.toList |> List.map toTranslation
52
53     let FindTranslation collectionFunction (criteria : Option<string>) (
           fromText : string) (property : Option<string>) (context : string)
           toLanguageCode =
54         // If fromtext does not contain a word, return an empty string
```

```
55          match fromText.Trim() with
56          | "" -> ""
57          | _ ->
58
59              // If criteria or property was not found in enums, return the
                    original string
60              match criteria, property with
61              | None,_ | _,None -> fromText
62              | Some criteria, Some property ->
63
64                  // If toLanguageCode is not valid, sets it to default "no"
65                  let toLang =
66                      match toLanguageCode with
67                      | null | "" -> "no"
68                      | _ -> toLanguageCode
69
70                  // Search for a valid translation
71                  let result =
72                      collectionFunction
73                      |> List.tryFind ( fun row ->
74                          row.Criteria.ToLower() = criteria.ToLower() &&
75                          row.FromLang = FromLanguageCode &&
76                          row.FromText = fromText &&
77                          row.Property.ToLower() = property.ToLower() &&
78                          row.Context.ToLower() = context.ToLower() &&
79                          row.ToLang = toLang )
80
81                  match result with
82                      | Some x -> x.ToText
83                      | None -> fromText
84
85      ////////////////////////////////////////
86      //      GetToText function returns the translated string, if defined in
            the Translate table.
87      //      If the fromText is not found, the value of fromText is returned
            unchanged.
88      let GetToText (criteria : Option<string>) (fromText : string) (property :
             Option<string>) (context : string) toLanguageCode =
89          FindTranslation GetTranslations criteria fromText property context
                toLanguageCode
```

A.4 sTranslate Parallel (F#)

A.4.1 XltTool.fs

```
1  namespace sTranslate_parallel
2
3  module XltTool =
4      open System
5      open System.Data
6      open System.Data.Linq
7      open FSharp.Data.TypeProviders
8      open Microsoft.FSharp.Linq
9      open FSharp.Configuration
10
11     // Get typed access to App.config to fetch the connection string later
12     type Settings = AppSettings<"App.config">
13
14     // SQL Type Provider. Give it a dummy connection string for now to
           satisfy the compiler.
15     type dbSchema = SqlDataConnection<ConnectionStringName = "
           dbConnectionString">
16
17     // Record type containing a search to the database
18     type Search = {
19         Criteria : string
20         FromText : string
21         Property : string
22         Context : string
23         ToLang : string
24     }
25
26     // Record type so that we can cache the database to memory and not have
           the data context go out of scope
27     type Translation = {
28         FromText : string
29         ToText : string
30         Context : string
31         Property : string
32         Criteria : string
33         FromLang : string
34         ToLang : string
35     }
36
37     let fromLang = "en"
38
39     // Copies the contents of a database row into a record type
40     let toTranslation (xlt : dbSchema.ServiceTypes.Translation) = {
41         FromText = xlt.FromText
42         ToText = xlt.ToText
43         Context = xlt.Context
44         Property = xlt.Property
45         Criteria = xlt.Criteria
46         FromLang = xlt.FromLang
47         ToLang = xlt.ToLang
48     }
49
50     // Copies the database to memory
51     let getTranslations =
52         let db = dbSchema.GetDataContext(Settings.ConnectionStrings.
               DbConnectionString)
53         query {
54             for row in db.Translation do
```

```
55                  select row
56              } |> Seq.toArray |> Array.map toTranslation
57
58          // Returns the correct translation for the given search
59          let findTranslation (s : Search) =
60              // If fromtext does not contain a word, return an empty string
61              match s.FromText.Trim() with
62              | "" -> ""
63              | _ ->
64
65                  // If toLanguageCode is not valid, sets it to default "no"
66                  let toLang =
67                      match s.ToLang with
68                      | null | "" -> "no"
69                      | _ -> s.ToLang
70
71                  // Search for a valid translation
72                  let result =
73                      getTranslations
74                      |> Array.tryFind ( fun row ->
75                          row.Criteria.ToLower() = s.Criteria.ToLower() &&
76                          row.FromLang = fromLang &&
77                          row.FromText = s.FromText &&
78                          row.Property.ToLower() = s.Property.ToLower() &&
79                          row.Context.ToLower() = s.Context.ToLower() &&
80                          row.ToLang = toLang )
81
82                  match result with
83                      | Some x -> x.ToText
84                      | None -> s.FromText
85
86          // Wraps FindTranslation inside an async
87          let findTranslationAsync s =
88              async { return findTranslation s }
89
90          // Takes a list of database searches, makes asyncs out of them, and
                combines them into a single parallel task.
91          // Returns a list of results, that are in the same order as the searches.
92          let getToTextAsync (searchList : Search list) =
93              searchList
94              |> List.map findTranslationAsync
95              |> Async.Parallel
96              |> Async.RunSynchronously
```