FSharp Project Report

Course requirement for completion of INB309-1 & INB309-2

Student: Jessica Davis (N8855391)
Supervisor: Associate Professor James Hogan, PhD

Semester 2, 2016

Abstract

The aim of this project was an exploration of using the FSharp programming language with the .NET Bio Bioinformatics library, with the eventual development of a Type Provider for GenBank data. It is believed that the scripting capabilities of FSharp (FSharp Interactive), will enable users to perform quick and easy explorations of GenBank data. The output of this project is a workflow to identify and compare gene upstream regions, written in a functional style, and a small Type Provider for GenBank files, which highlights the possibilities that a more generalised Type Provider may offer.

Repository: view top level repository
Wiki and Tutorials: view wiki

Contents

Introduction

This chapter aims to introduce the fundamental concepts underpinning the motivation for- and execution of- this project. This will include a short discussion of Bioinformatics and its importance; a discussion of the existing Bioinformatics library for .NET- **.NET Bio**; a superficial discussion of type providers, and; the goals and objectives of the project.

1.1 Bioinformatics

Merriam Webster defines Bioinformatics as;

> "The collection, classification, storage, and analysis of biochemical and biological information using computers especially as applied to molecular genetics and genomics."

The key issue here is that the advent of next generation sequencing techniques has led to an abundance of genomic data, and new techniques must be developed to manage and utilize this data. Bioinformatics is important in and of itself for the management of biological and medical data. However, provided the proper analytical tools are created and distributed, Bioinformatics could have an even more important role in contributing to a functional understanding of the human genome, which could lead to enhanced discovery of drug targets and individualised therapy [1].

There exist several different tools and libraries, across a variety of languages and platforms, which seek to fill different needs. The work in this project makes use of several such tools:

- The National Center for Biotechnology Information (NCBI)'s GenBank database
- NCBI's Basic Local Alignment Search Tool (BLAST)
- NCBI's Entrez Programming Utilities (eutils)
- The open source .NET Bio .NET library.

1.2 GenBank

According to the National Center for Biotechnology Information (2016) , GenBank is a genetic sequence database - an annotated collection of all publicly available DNA sequences [5].

This is one of the most comprehensive sources of genomic data, all of which is made available in the GenBank flat file format via an anonymous ftp server (see here for format specifications).

This format is favoured over the FASTA file format in this particular project, as GenBank files contain metadata that is vital to have in order to offer much of the functionality provided by this project.

There are a number of different ways to retrieve GenBank records, and the format seems to undergo changes with some regularity, which was a pain point in early development. For example, the file extension seems to have been changed at some point between the release of .NET Bio and the development of this project, which caused minor hiccups.

1.3 Entrez Programming Utilities

Entrez Programming Utilities (eutils), are a set of programming utilities for searching and fetching data from NCBI's databases.

Requests can be made using HTTP GET and POST requests, with URL parameters that specify the required eutil service, the query term, and the output data format.

Efetch is the particular eutil service relied upon in this project, which allows genomic data records to be retrieved from a specified database based on the accession number of the record.

A sample query might be

https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=nuccore&id=U00096&rettype=gb&retmode=text

Where `https://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi` is the endpoint, `db=nuccore` specifies the database to retrieve the record from, `id=U00096` is the accession number of the record, `rettype=gb` specifies that the record should be returned in the GenBank flat file format, and `retmode=text` specifies the the result should be sent as text (this is often xml by default, which does not play well with .NET Bio parsers).

1.4 BLAST

The Basic Local Alignment Search Tool (BLAST) is a tool which allows users to find regions of similarity between biological sequences [8].

In the implementation of this project, BLAST requests were made using the BLAST API, accessed through a wrapper class in the .NET Bio library. Figure 1.2 is a graphical representation of the parameters that may be passed to a BLAST request. Figure 1.3 shows the default algorithm parameters, which may be modified as desired.

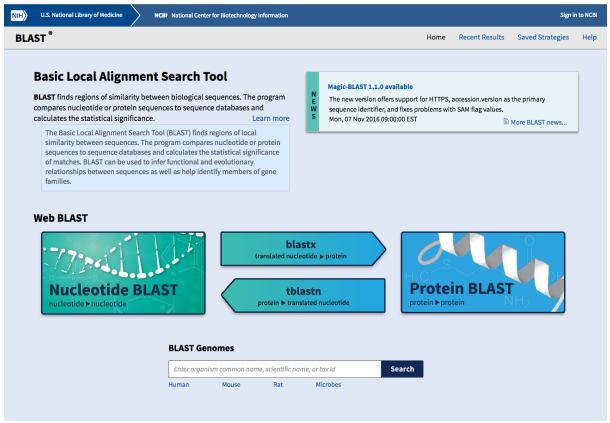

Figure 1.1: BLAST homepage.

There are several different kinds of BLAST requests which can be performed; the main focus of this project was on the Nucleotide BLAST option

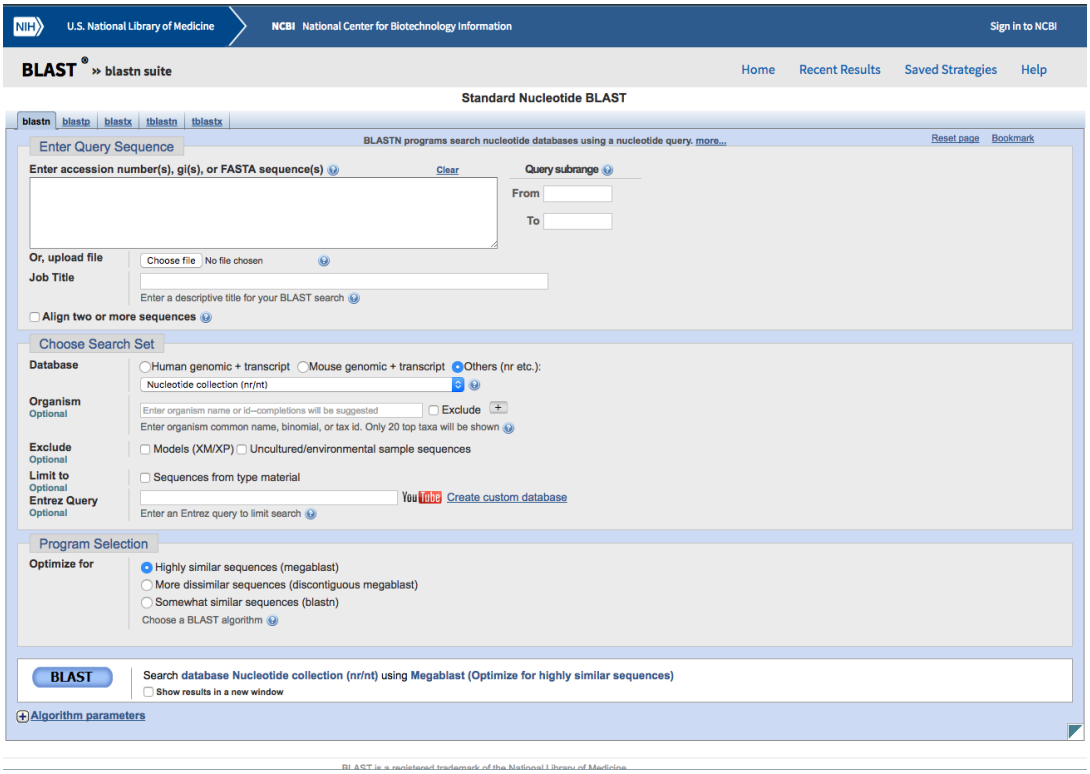

Figure 1.2: The standard BLAST query page for a Nucleotide BLAST

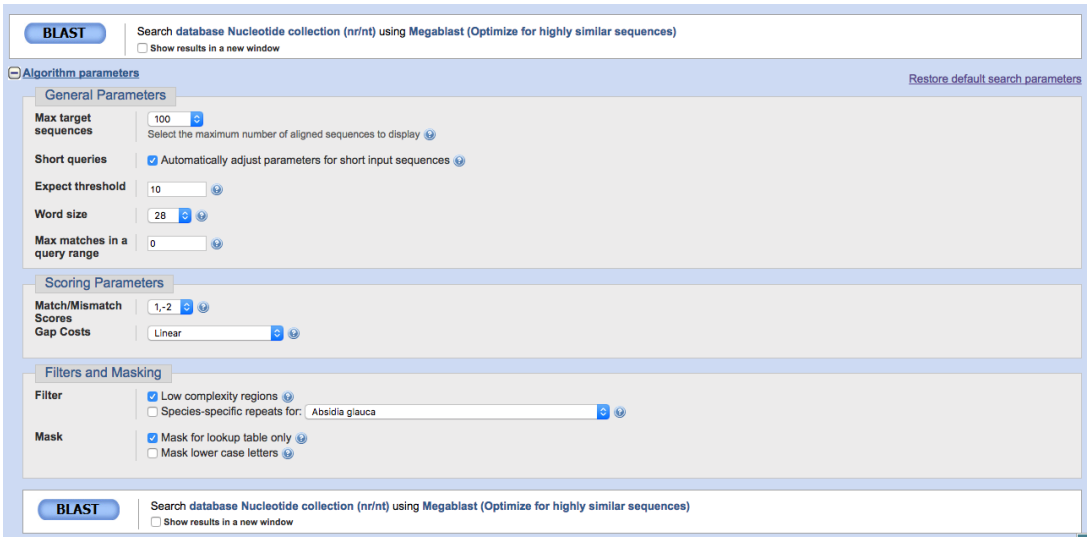

Figure 1.3: Algorithm parameters that can be modified in a Nucleotide BLAST

1.5 .NET Bio

According to the .NET Bio GitHub page (2016);

> .NET Bio is an open source library of common Bioinformatics functions, intended to simplify the creation of life science applications.
>
> The core library implements a range of file parsers and formatters for common file types, connectors to commonly-used web services such as NCBI BLAST, and standard algorithms for the comparison and assembly of DNA, RNA and protein sequences. Sample tools and code snippets are also included [3].

Much of the functionality provided by .NET Bio is utilized in this project. One of the key motivations for .NET Bio was to provide a set of generalised components, that are not dependant on a particular data format (i.e. they work equally well with FASTA and GenBank flat file sequences). However, by definition, this means that there are many layers of abstraction, and the process of reaching core data components can be quite convoluted. A fact which was a key motivation for the development of the type provider discussed in chapter 4.

1.6 FSharp and Type Providers

1.6.1 FSharp

Fox (2016) explains FSharp as;

> ...a cross-platform, open source programming language for .NET which provides first-class support for functional programming, along with support of object-oriented

and imperative programming. The Visual FSharp compiler and tooling are Microsoft's implementation and tooling for the FSharp programming language, making FSharp a first-class member of .NET. [6]

The fact that FSharp is a functional language supported by .NET offers the opportunity to work with the genomic data within the functional paradigm, while still being able to take advantage of the .NET Bio library.

1.6.2 Type Providers

Type providers are a component unique to FSharp, which provide types, properties, and methods for use within a program. FSharp type providers can provide types based on external information sources. This project seeks to explore the development of a type provider to work in conjunction with .NET Bio for the development of simplified, strongly typed workflows with genomic data.

An example of a well-established type provider is the JSON type provider. A type for a particular dataset is created by passing a URL to the JSONProvider, the type provider then dynamically generates types and methods for that dataset based on the JSON document structure. Once the type has been created, other documents with the same structure can be loaded using the Load method, and the JSON properties of that dataset can then be accessed using IntelliSense, as shown in Figure 1.4.

Figure 1.4: Using the FSharp.Data JSONProvider by Petricek & Guerra, 2016 [9]

It is easy to see the possibilities that such a construct provides, especially in the context of simplifying programmatic interactions with information sources.

1.7 Project Goals and Objectives

The initial motivation for this project was the development of a type provider for using GenBank data with .NET Bio, in order to take advantage of the scripting capabilities of FSharp, while also offering streamlined genomic data access. The path toward this goal occurred in several key stages:

1. Familiarization with the field of Bioinformatics
2. Familiarization .NET Bio Library
3. Setting up a development environment for using .NET Bio in FSharp
4. Explorations of .NET Bio with FSharp
5. Development of workflow in FSharp Interactive for making a BLAST request with a particular gene, and comparing upstream regions of the input gene with the upstream regions of the result genes.
6. Familiarization with Type Providers
7. Explorations with creating and using Type Providers
8. Development of a type provider for GenBank data using .NET Bio infrastructure

Items 5 and 7 represent the two main programmatic outputs of this project, both of which are accompanied by documentation and a tutorial wiki. The other items on the list represent stages of research and exploration, which required significant time input for no directly usable output.

Development with .NET Bio in FSharp

This chapter discusses the trials and tribulations of setting up .NET Bio to work with FSharp, and presents a tutorial for replicating this set-up.

2.1 General Comments

On reflection, it is rather laughable how long it took to get past this initial step of setting up a project to work with .NET Bio. The difficulties ultimately came down to a lack of experience with Visual/Xamarin Studio, and correct documentation being hard to find.

Initially, it was not known that .NET Bio was available as a Nuget package. It is quite likely that this is an obvious fact, which doesn't need to be explained. However, as a novice, it was difficult to understand how to use the library, and attempts were made to download the entire source and try to compile this for use in another project, which failed for various reasons. Eventually, while scouring the internet, an offhand remark in a BioStars forum thread mentioned that .NET Bio was distributed as a Nuget package, and the library was finally able to be added to a project... almost.

There were some issues arising from the fact that .NET Bio could not work on the default runtime selected when a new FSharp project is created. Again, this took longer than it should have to resolve due to a lack of experience with these issues, and little understanding of what it means to choose a different FSharp runtime or .NET Framework.

It has been found that the ideal configuration for an FSharp project working with .NET Bio is to use .NET Framework 4.5 or higher, and FSharp runtime 4.0.0.

Significant time was also invested in setting this up under mono on MacOS, which compounded the problem, but eventually projects were able to be run in both Xamarin Studio and Visual Studio.

2.2 Instructions

This section includes instructions for how to add the .NET Bio library to an FSharp project under .NET; instructions for Mono setup, and troubleshooting ideas, can be found in A.1.1. After the initial battle outlined above, this process now takes no longer than 1-2 minutes.

2.2.1 Windows - Visual Studio

The screenshots and instructions below outline how to set up a new visual studio project that makes use of .NET Bio.

Step 1: Create a new visual studio solution, as an FSharp console project. Your screen should then look as shown in figure 2.1.

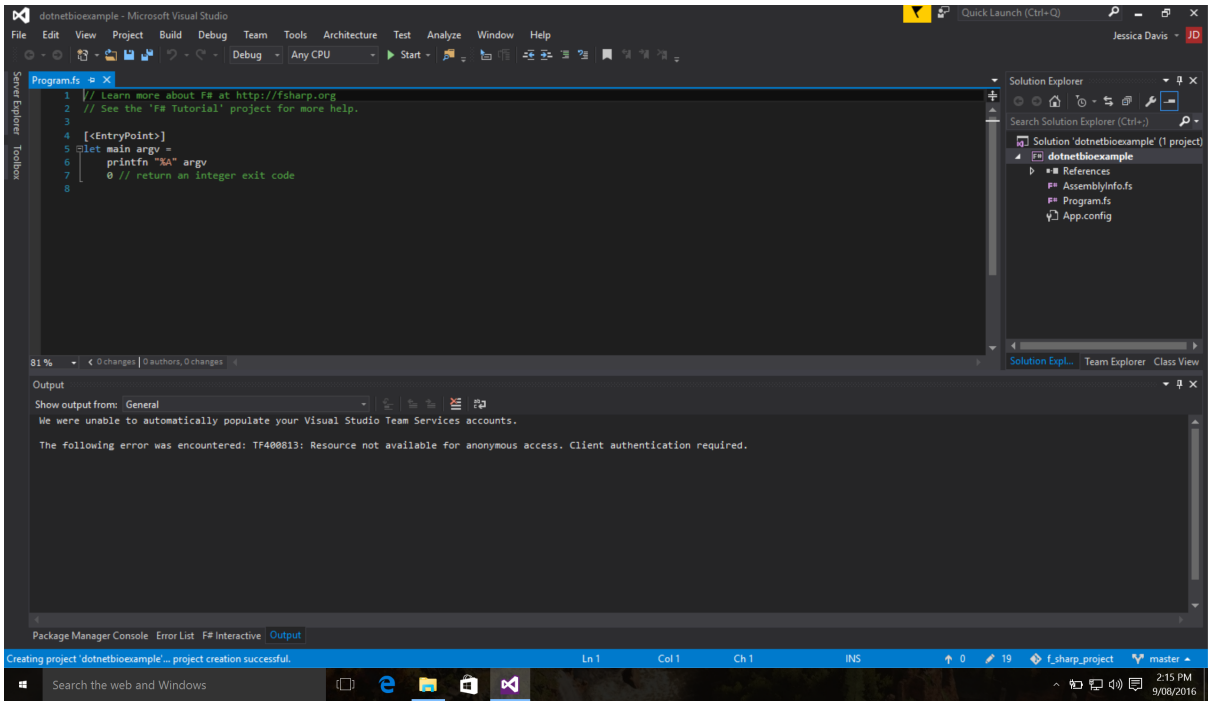

Figure 2.1: Step 1

Step 2: You will now need to add the .NET Bio package. To do this, right click on the project in solution explorer, and select manage NuGet packages, as shown in figure 2.2.

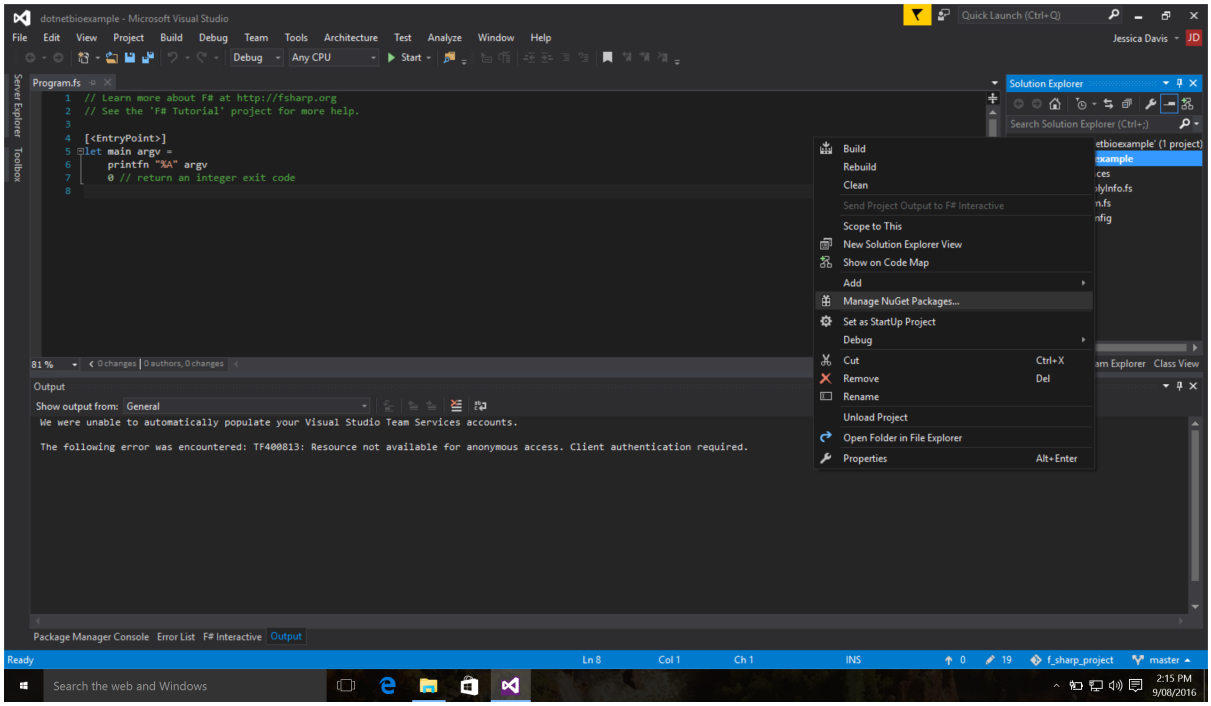

Figure 2.2: Step 2

Step 3: Search 'bio.core', or some variation thereof. The .NET Bio package should present itself as seen in figure 2.3.

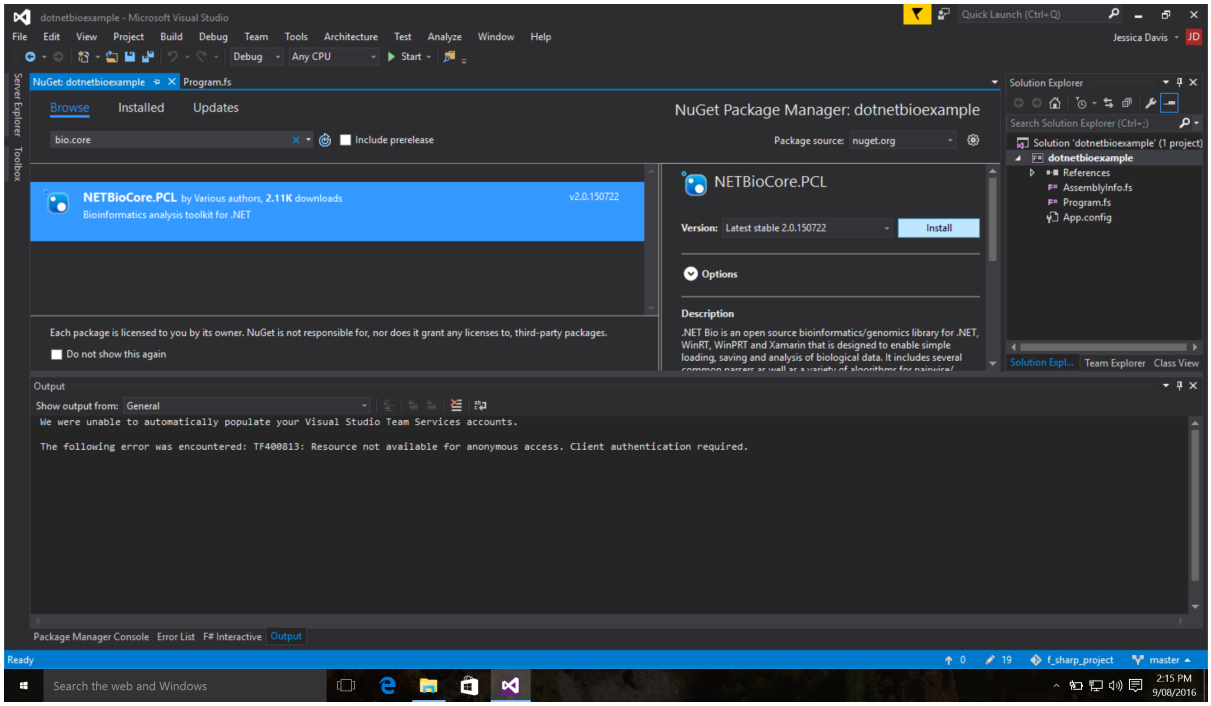

Figure 2.3: Step 3

Step 4: Once the package has installed, you should be able to open and use as required.

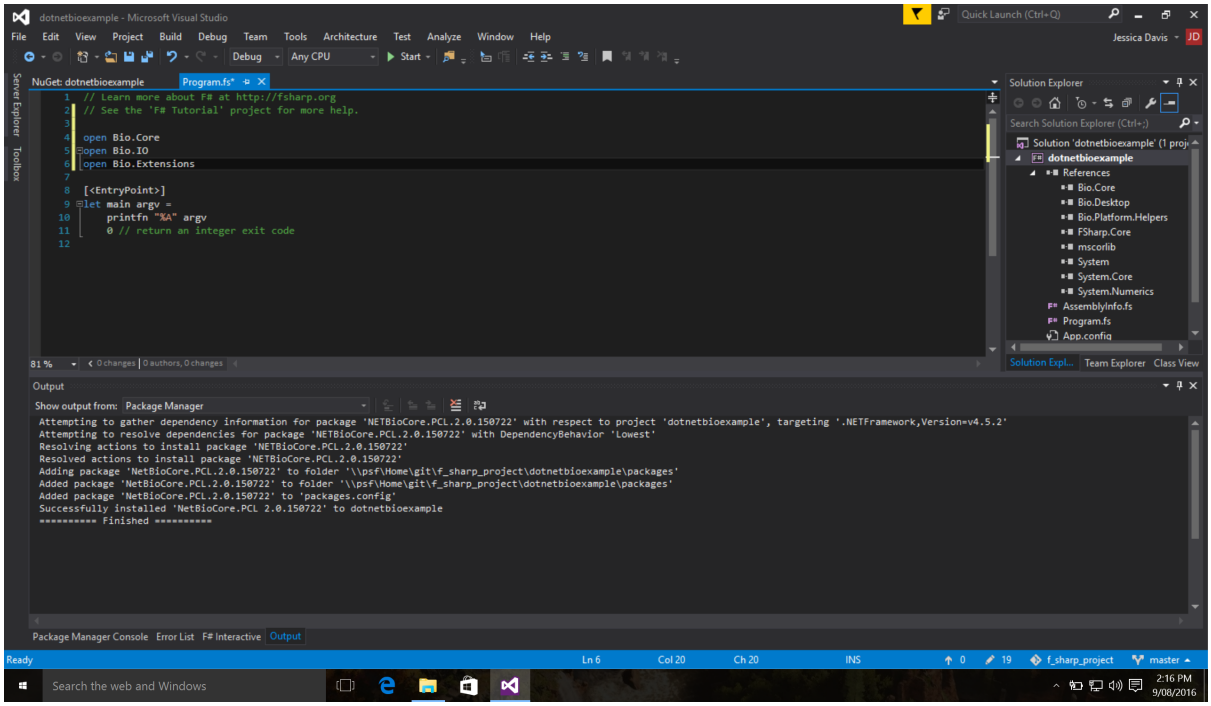

Figure 2.4: Step 4

Step 5: Interacting with .NET Bio in FSharp Interactive.

In a new FSharp console application, you will find two key file types `.fs` and `.fsx`. FSharp program files (.fs) are similar to typical CSharp files, and require a main method, with an `[<EntryPoint>]` annotation. Code written in these files can be run and debugged using the traditional methods.

FSharp interactive files (fsx) are typically used for writing scripts. FSharp interactive allows you to run code line-by-line and get immediate feedback, which is ideal for flexible interactions with complex data. There are a few steps to getting FSharp interactive correctly configured with .NET Bio.

1. In solution explorer, right click on the references for the project you are working on, and click 'send references to FSharp Interactive'. This will ensure that all the required assemblies are available in your interactive session.

2. At the very beginning of your fsx file, include the following:

```
#r "../packages/NetBioCore.PCL.2.0.150722/lib/net45/Bio.Core.dll"
#r "../packages/NetBioWeb.PCL.2.0.150722/lib/Bio.WebServices.dll"
```

These steps ensure that the .NET Bio types and methods will be available in the context of the FSharp Interactive session (be sure to execute the reference lines before anything else - right click on the line and select 'send to FSharp Interactive', or press alt-enter on Windows).

Upstream Analysis Workflow

This chapter walks through the steps required and challenges faced to create the workflow available in the Git Repository for this project, as well as instructions on how to use this workflow in FSharp Interactive.

3.1 Overview

Bioinformatics is seemingly riddled with conflicting data formats, database standards, and services that *don't quite* talk to each other. As a result, some common workflows can be quite tedious to complete, requiring heavy user engagement. This workflow aims to automate the majority of the work required to prepare a set of genes for upstream analysis.

3.2 Use Cases

Use Case 1 - Find orthologs

As a user, I have a transcription factor (e.g. Fur) and I know it regulates 80 genes in E.coli. I also have a list of 10 target genomes that I want to study. I want to find, of the 80 genes in E.coli, how many orthologs exists in the 10 target genomes using the reciprocal best hit method.

Use Case 2 - Find upstream regions

As a user, once the reciprocal best hits are found, I want to be able to examine both the upstream intergenic region, and the 1000bp upstream region, of a particular hit.

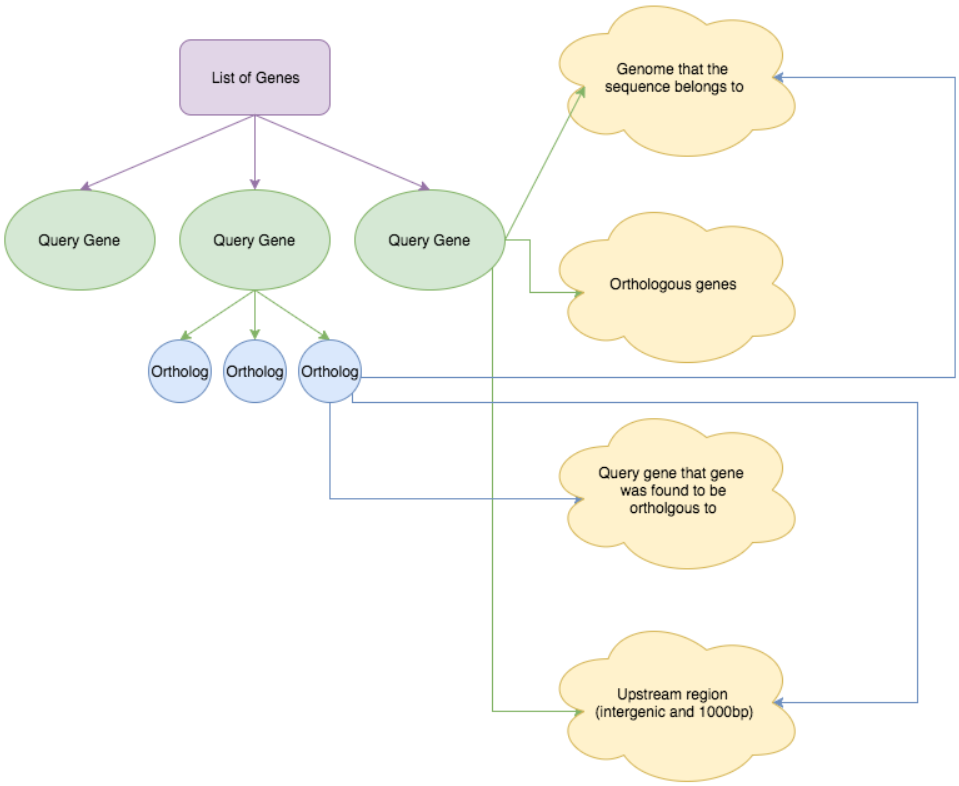

Figure 3.1: Abstract representation of data that should be associated with each object in the result set

Use Case 3 - Queries on results

As a user, I want to be able to perform queries on a result set, where a result set has the information as represented in figure 3.1.

The queries I want to be able to perform are:

- get the list of orthologs (the reciprocal hits) for query gene X
- get the list of orthologs for a specified query gene that all belong to the same genome G

Once I have this information, I want to be able to:

- get the upstream regions of the results in this list
- view the sequences of each of the results in this list
- filter out results which do not meet a specified evalue threshold

3.3 Development

3.3.1 Assumptions

There are several assumptions that were made in the development of this workflow. These assumptions were made in collaboration with an intended user.

- All queries will be within the Bacteria kingdom
- The user has a set of genes they wish to find orthologs for. These genes will be either:
    1. a string list of one or more gene locus tags
    2. a fasta file containing one or more genes, where the fasta file is strictly in the form shown in figure 3.2. That is, sequences are denoted by an id line followed by the sequence itself. The id line should contain the accession number of the genome to which the sequence belongs.
- The user has an internet connection
- The user has Visual Studio or Xamarin Studio installed, with FSharp runtime 4.0, and .NET 4.5 or higher.

```
>U00096
atggttaaagtttatgccccggcttccagtgccaatatgagcgtcgggtttgatgtgctcggggcggcggtgacacctgt
tgatggtgcattgctcggagatgtagtcacggttgaggcggcagagacattcagtctcaacaacctcggacgctttgccg
ataagctgccgtcagaaccacgggaaaatatcgtttatcagtgctgggagcgtttttgccaggaactgggtaagcaaatt
ccagtggcgatgaccctggaaaagaatatgccgatcggttcgggcttaggctccagtgcctgttcggtggtcgcggcgct
gatggcgatgaatgaacactgcggcaagccgcttaatgacactcgtttgctggctttgatgggcgagctggaaggccgta
tctccggcagcattcattacgacaacgtggcaccgtgttttctcggtggtatgcagttgatgatcgaagaaaacgacatc
atcagccagcaagtgccagggtttgatgagtggctgtgggtgctggcgtatccggggattaaagtctcgacggcagaagc
cagggctattttaccggcgcagtatcgccgccaggattgcattgcgcacgggcgacatctggcaggcttcattcacgcct
gctattcccgtcagcctgagcttgccgcgaagctgatgaaagatgttatcgctgaaccctaccgtgaacggttactgcca
ggcttccggcaggcgcggcaggcggtcgcggaaatcggcgcggtagcgagcggtatctccggctccggcccgaccttgtt
cgctctgtgtgacaagccggaaaccgcccagcgcgttgccgactggttgggtaagaactacctgcaaaatcaggaaggtt
ttgttcatatttgccggctggatacggcgggcgcacgagtactggaaaactaa
>U00096
atgaaactctacaatctgaaagatcacaacgagcaggtcagctttgcgcaagccgtaacccaggggttgggcaaaaatca
ggggctgtttttttccgcacgacctgccggaattcagcctgactgaaattgatgagatgctgaagctggattttgtcaccc
gcagtgcgaagatcctctcggcgtttattggtgatgaaatcccacaggaaatcctggaagagcgcgtgcgcgcggcgttt
gccttcccggctccggtcgccaatgttgaaagcgatgtcggttgtctggaattgttccacgggccaacgctggcatttaa
agatttcggcggtcgctttatggcacaaatgctgacccatattgcgggtgataagccagtgaccattctgaccgcgacct
ccggtgataccggagcggcagtggctcatgctttctacggtttaccgaatgtgaaagtggttatcctctatccacgaggc
aaaatcagtccactgcaagaaaaactgttctgtacattgggcggcaatatcgaaactgttgccatcgacggcgatttcga
tgcctgtcaggcgctggtgaagcaggcgtttgatgatgaagaactgaaagtggcgctagggttaaactcggctaactcga
ttaacatcagccgtttgctggcgcagatttgctactactttgaagctgttgcgcagctgccgcaggagacgcgcaaccag
ctggttgtctcggtgccaagcggaaacttcggcgatttgacggcgggtctgctggcgaagtcactcggtctgccggtgaa
acgttttattgctgcgaccaacgtgaacgataccgtgccacgtttcctgcacgacggtcagtggtcacccaaagcgactc
aggcgacgttatccaacgcgatggacgtgagtcagccgaacaactggccgcgtgtggaagagttgttccgccgcaaaatc
tggcaactgaaagagctgggttatgcagccgtggatgatgaaaccacgcaacagacaatgcgtgagttaaaagaactggg
ctacacttcggagccgcacgctgccgtagcttatcgtgcgctgcgtgatcagttgaatccaggcgaatatggcttgttcc
tcggcaccgcgcatccggcgaaatttaaagagagcgtggaagcgattctcggtgaaacgttggatctgccaaaagagctg
gcagaacgtgctgatttacccttgctttcacataatctgcccgccgattttgctgcgttgcgtaaattgatgatgaatca
tcagtaa
```

Figure 3.2: Fasta format for use with upstream analysis workflow

3.3.2 Process

Step 1: Issuing a BLAST Request

The first attempt at implementing this workflow made use of some GenBank flat files provided from an existing data store on a local machine. Significant work was done to become familiar with the .NET Bio codebase, and the Genbank flat file format. From here, code was written - making use of .NET Bio's file parsers - to read a GenBank flat file in to an ISequence object.

Several issues arose here, which took a significant time investment to understand, but are now known to be trivial. For example, incorrect use of IDisposable resources throwing exceptions that were unfamiliar; the nuances of FSharp sequences meaning that computations for each list element were not performed at the time the overall sequence was created, causing evaluations to occur involving resources that were no longer in scope; etc...

Several modules were created here to streamline data access and processes behind the scenes, but as familiarization with the .NET Bio library continue to increase, it was realized that much of this logic already existed within .NET Bio, and so significant refactoring occurred to make

better use of .NET Bio's offerings.

After verifying the functionality of this starting point, code was written - making use of .NET Bio's BlastWebHandler - to perform a BLAST query with the genome that had been parsed. This code failed tremendously, for the following reasons:

1. Lack of understanding of threading and asynchronous programming in CSharp. Once familiarization was gained with this, it was found not to translate to FSharp as well as other language elements, as FSharp does not implement Async in the same way as CSharp. As a result, further time was spent learning the FSharp logic and patterns for asynchronous workflows.

2. Due to a lack of clear use cases at the time of developing this component, the kinds of sequences that users would be making BLAST requests with was not entirely understood. As a result, in early development, the sequence of an entire genome was being submitted to a BLAST request, which was resulting in HTTP status `414 Request-URI Too Long`.

   The HTTP specification states that a 414 status should be returned when

   > "The server is refusing to service the request because the Request-URI is longer than the server is willing to interpret." [4]

   So the best guess here is that NCBI has some kind of character limit on the length of a BLAST request, and that the sequence of an entire genome exceeds this limit.

   Passing a single gene to the NcbiWebHandler (mostly) resolved this issue.

3. Lack of understanding of NCBI's database structure, the difference between a protein and a nucleotide, and what, exactly, BLAST is. This meant that query parameters were not ideal e.g. wrong database and BLAST program for the chosen gene. Once this was better understood, queries were organized so that actual results would come back.

4. NCBI was in the process of changing over to HTTPS; changing the defaultEndpoint property of the NcbiWebHandler fixed this issue.

Step 1.1: Getting data from the NCBI FTP server

It took some weeks for the aforementioned issues to be overcome. In the meantime, the initial data source was modified so that genomes could be loaded in directly from NCBI's FTP server, instead of the user having to have files downloaded before using the workflow. This exercise presented its own challenges, as NCBI had overhauled their old FTP server structure, so most documentation was obsolete. Eventually, a method was developed to give the user a list of all available genomes in the `/Bacteria/Escheria_coli` folder, as this was mentioned as an organism of particular interest. The functionality to provide this directory listing and then download, save, and read a chosen genome in as an ISequence is hidden away in modules, so all the user needs to do is execute the lines seen in figure 3.3 in FSharp interactive.

```
#r "../packages/NetBioCore.PCL.2.0.150722/lib/net45/Bio.Core.dll"
#r "../packages/NetBioWeb.PCL.2.0.150722/lib/Bio.WebServices.dll"
#load "BLAST.fs"
#load "DownloadGenBankFiles.fs"
#load "GenomeHelperMethods.fs"
let dataDir = System.IO.Directory.GetCurrentDirectory() + "/Data/"
// 1. Chose a genome from the list of available genomes
let availableGenomes = Seq.toList DownloadGenBankFiles.
    getListOfEcoliFileNamesAsSequence
let chosenGenome = availableGenomes.[0]
// 2. Download the .gbff.gz file as .gbk from ncbi and save locally for use
DownloadGenBankFiles.downloadAndSaveGenbankFile chosenGenome
// 3. Read the file in as an ISequence
let chosenGenomeISeq = GenomeHelperMethods.getGenomeSeqFromFile (dataDir +
    chosenGenome + ".gbk")
```

Figure 3.3: Choosing a file from the FTP server, downloading, and reading in as an ISequence

However, the way that NCBI organises folders on the FTP server means that there is little correlation between, say, a genome's accession number and the name of the folder that contains it. Folders are given names based on when that particular piece of data was added to the server. As such, it was difficult to design the workflow in a way that would give the user the desired level of automation, as well as the flexibility to choose a variety of different genomes from different species or organisms. As a result, this may have been a largely wasted endeavour; it may be more prudent to make use of the NCBI eutils for data selection and download. However, this too, has proven difficult due to missing documentation and changed formats.

Step 1.1.2: BLAST, revisited

Eventually (over 5 weeks later), a result stream was able to be successfully returned from a BLAST web request, using the following method:

```
let getBlastResultStream sequences =
        let doBlastAsync dsequences = async {
                let bp = new Blast.BlastRequestParameters(sequences, [new
                    KeyValuePair<string, string>("evalue","1")])
                bp.Database <- "nr"
                bp.Program <- BlastProgram.Blastn

                // set up web handler
                let ncbiWebHandler = new NcbiBlastWebHandler(
                        EndPoint = @"https://www.ncbi.nlm.nih.gov/blast/Blast.cgi
                            ",
                        LogOutput = fun str -> Console.WriteLine(str)
                )

                use task = ncbiWebHandler.ExecuteAsync(bp, System.Threading.
                    CancellationToken.None)
                // wait for response to come back before passing along
                Async.AwaitIAsyncResult task
                return task.Result
        }

        // Make to call to BLAST synchronously
        Async.RunSynchronously (doBlastAsync sequences)
```

Figure 3.4: Method to get BLAST results

From here, the next step was the parse the result stream from the method in Figure 3.4 using the .NET Bio BlastParser, which transforms the BLAST XML output into a BlastResult object.

A non-trivial issue was encountered here, which has been worked around, but not solved. It seems that if the BLAST result stream from the above method is passed directly to a BlastParser, the stream is partially cut off, causing the parser to throw an unexpected End Of File exception. It was discovered that if the stream was logged to the console, the stream terminated correctly, but if it was written to a file, then the stream terminated before all data had been written. However, if stream was read in by a StreamReader, then written to a file from the StreamReader, all xml was written correctly. This workaround method can be seen in figure B.1.

As mentioned above, this workflow aims to get the bidirectional best hit, which means that a new BLAST request must be constructed with the output of the initial request used as the input

to the second request. Since the .NET Bio NcbiBlastWebHandler takes a sequence of ISequence objects as input, a method was written to transform BlastHit objects into ISequences, as seen in figure 3.5. This method ensures that the BlastHits can be consumed by the NcbiBlastWebHandler, and takes advantage of the generic Metadata dictionary to associate the original BlastHit object with it's new ISequence object, so that the user can refer back to the result statistics as desired.

```
let blastResultAsISeqs (blastResult:BlastResult) =
       blastResult.Records.[0].Hits
       |> Seq.map (fun ht ->
              let seq = new Sequence(Alphabets.DNA, ht.Hsps.[0].QuerySequence)
                   :> ISequence
              // Associate the BlastHit with the ISequence so data is not lost
              seq.Metadata.Add("BlastHit", ht)
              seq
       )
```

Figure 3.5: Method to transform BlastHit objects to ISequence objects

Step 1.1.3: Data from eutils

Upon completion of the above steps, a discussion was had with a potential user, which motivated significant refactoring, and an overhaul of the way data is provided.

Users may now input a file which is in the format from figure 3.2, and the genome will be retrieved from the web using NCBI's eutils; the genome for each blast hit on each input gene is also retrieved in this way. The methods in section 3.6 allow multiple requests to occur in parallel, and wait for all requests to come back before proceeding.

In figure 3.6, the web request is configured to always download from the nucleotide (nuccore) database, in the GenBank data format; this will eventually be modified to work with proteins was well, but data must always be in the GenBank format, or there is simply not enough information to produce results with any particular confidence.

```
 // download the genome corresponding to an accession
let getGenomeWithEfetch alignmentAccession = async {
        Console.WriteLine("Requesting genome for " + alignmentAccession)
        let request =   WebRequest.Create("https://eutils.ncbi.nlm.nih.
            gov/entrez/eutils/efetch.fcgi?db=nuccore&id="+
            alignmentAccession+"&rettype=gb&retmode=text") :?>
            HttpWebRequest
        // these requests will take a while...
        request.KeepAlive <- false
        request.Timeout <- System.Threading.Timeout.Infinite
        request.ProtocolVersion <- HttpVersion.Version10
        request.AllowWriteStreamBuffering <- false
        try
                use! response = request.AsyncGetResponse()
                use stream = response.GetResponseStream()
                let gbParser = new GenBankParser()
                return gbParser.ParseOne(stream)
        with
                | _ ->
                        Console.WriteLine("Unable to get the genbank
                            record for {0}", alignmentAccession)
                        return null
}

// do all requests in parallel to download genomes based on list of
    accessions
let allGenomesFromAcessions accessionList =
        accessionList
        |> Seq.map (fun ac -> getGenomeWithEfetch ac)
        |> Async.Parallel
        |> Async.RunSynchronously
        |> Seq.filter (fun x ->
                match x with
                | null -> false
                | _ -> true
        )
        |> Seq.toList
```

Figure 3.6: Two methods to facilitate genome file download from eutils

Step 2: Verifying the bidirectional best hit

Verifying the bidirectional best hits presents one of the most challenging aspects of this workflow. To begin, let us outline *exactly* what a bidirectional best hit is, and how it is obtained. This was initially not well understood in development, which led to the creation of interesting- but not entirely useful- algorithms.

According to Dalquen & Dessimoz (2012), finding bidirectional best hits

> "...entails identifying the pairs of genes in two different genomes that are more similar to each other than either is to any other gene in the other genome" [2]

Thus, the steps for determining the bidirectional best hits are:

1. Perform a BLAST request with an initial gene sequence (figure 3.7, layer 2), and get the results (layer 3)
2. For each result of the initial BLAST request (layer 3), perform a new BLAST request using the layer 3 result as the input sequence, and query only against the genome to which the initial gene (layer 2) belongs
3. Determine if any of the hits (layer 4) for a BLAST result (layer 3) are equal to the original gene (layer 1)

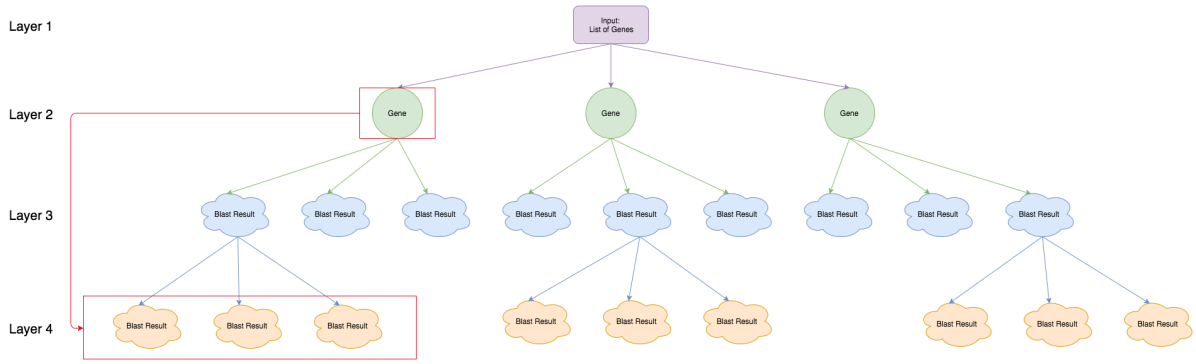

Figure 3.7: Layers of BLAST requests performed

A key issue here is that the sequences returned as hits from a BLAST request are not necessarily matches for an entire genomic feature. As such, when the BLAST1 (layer 3) requests come back, some further processing must occur:

1. For each layer 3 result, get the accession number of the genome to which the result sequence belongs, and use the algorithm in figure 3.6 to download those genomes and store them in memory. These genomes need to be downloaded in the GenBank file format, as information about sequence features and their locations is essential.
2. For each layer 3 result, search its genome file for a feature which encompasses the layer 3

result sequence. If a matching feature is found, associate the data for that feature with the layer 3 result. If a matching feature is not found, discard the layer 3 result.

Each layer 2 gene now has a sequence of BLAST results associated with it, where each BLAST result was found to be part of a known genomic feature, and that BLAST result has the full sequence of its matched feature associated with it in memory.

Once the layer 3 results have been associated with their full feature sequences, the BLAST2 request can be performed. This is much the same as BLAST1, but the input is now the full feature sequence for a layer 3 result. Thus, a BLAST request must be performed for each layer 3 result under each layer 3 gene, only against the genome which the layer 2 gene belongs. This is achieved by adding an 'extra parameter' to the .NET Bio NcbiWebHandler request: `ENTEREZ_QUERY=query_gene_genome_accession[Accession]`. This tells BLAST to do alignments only against the specified genome record.

It is not important to keep the layer 4 BLAST results, it is important only to verify that of the layer 4 results returned (limited only to the genome which the layer 2 gene belongs), the **best** hit (typically the lowest EValue) **is the exact** layer 2 input gene. This is achieved by using an algorithm which ranks the layer 4 results by EValue, extracts the result with the lowest EValue, and compares this to the layer 2 gene. If they match, then the layer 3 result to which the layer 4 results belong is considered to be a bidirectional best hit for the layer 2 gene.

This is shown in listing B.1. The algorithm takes all the layer 2 genes associated with their layer 3 results as a 'BlastOutputRecord', and returns a new sequence of all layer 2 genes associated only with the layer 3 results that were determined to be bidirectional best hits.

Once all best hits are found and returned as a sequence of BlastOutputRecords (layer 2 gene associated with sequence of layer 3 results that are bidirectional best hits), then the upstream regions of all features may be retrieved.

Step 3: Retrieving 1000bp upstream- and intergenic- regions

There are two different types of upstream regions which may be of interest. The first is the 1000bp upstream region, which is simply the sequence of 1000 base pairs upstream of the specified gene start location. The second is the intergenic upstream region, which is the sequence of base pairs between the start of the specified gene, and the end of the previous gene.

As mentioned above, each BLAST result has feature data associated with it, and this data contains start and end location information. Therefore, to extract a particular sequence from the overall genome, the start and end location of a feature is used, and the base pairs at those locations can then be extracted from the genome sequence. Figure 3.8 shows the algorithm to retrieve the 1000bp upstream region for a given feature.

```
let getUpstreamRegion (genomeSeq:ISequence) (gene:FeatureItem) =
        let getUpstreamSeq1000bp (genomeSeq:Bio.ISequence) upstreamStartPoint =
                let numBPToGet = int64(1000)
                let genomeStart = int64(0)
                if (upstreamStartPoint <= genomeStart) then
                        genomeSeq.GetSubSequence(genomeStart, numBPToGet) else
                                genomeSeq.GetSubSequence(upstreamStartPoint,
                                    numBPToGet)
        // if the feature is on the other strand make sure to take upstream of
            reversed complemented sequence
        let upstreamRegion =
                match gene.Location.Operator.ToString() with
                        |"Complement" -> int64(gene.Location.LocationEnd)
                        | _ -> int64(gene.Location.LocationStart - 2)
                |> getUpstreamSeq1000bp genomeSeq
        match gene.Location.Operator.ToString() with
                |"Complement" -> upstreamRegion.GetReverseComplementedSequence()
                | _ -> upstreamRegion
```

Figure 3.8: Algorithm to retrieve 1000bp upstream region of a given feature

Step 4: Querying results

Most of the logic outlined above is hidden away in an FSharp library file (but not a portable library, as .NET Bio does not appear to be compatible with the target framework and profile for portable FSharp libraries at this stage). The user can then interact with the data with a few simple commands within an FSharp script file, executing things in FSharp Interactive as they go. Listing B.2 shows how a user would go through and complete each stage as discussed above.

3.3.3 Comments and Known Issues

At this stage, the upstream workflow is very much in the experimental stage, and still needs rigorous testing, as well as the addition of some further 'nice to have' features before it is released. Some known issues which are currently being addressed are:

- In some cases, if a query gene or the feature associated with a BLAST hit has a particularly long sequence, HTTP 414 errors are still being thrown, which means that sometimes not all the orthologs can be verified. This is an issue that needs further investigation. For example, if the sequence is sufficiently long, would it be okay to trim the query to within

the URI length limitations, as the chances of random matches at these kind of lengths is extremely unlikely?

- BLAST requests take a long time, and since the number of BLAST2 requests is equal to the number of query genes multiplied by the number of hits, there can be quite a number of BLAST2 requests being sent. This also poses a problem as these large numbers of BLAST requests are in violation of the BLAST terms of use. As such, it might be worth looking in to porting some kind of local BLAST to .NET, so the alignments for BLAST2 requests can be run locally. This makes sense given that BLAST2 requests are run against a single genome which is already stored in memory.

- Given that there can be a large number of hits on a query gene (though this can be limited, it is likely that users will want to choose how many hits they consider), there are often a large number of efetch requests that need to be sent to get the genomes for each of these hits. This has already been optimised as much as possible by ensuing that the accession numbers sent off are a distinct list, but it can still be quite large. This could lead to violations of the efetch terms of use.

- In some cases, requests to retreive genomes using efetch were timing out due to a lack of a timely response from the server. This appears to have been mostly resolved by increasing the timeout of the request locally to wait indefinitely, and modifying the asynchronous parallel algorithms. However, this needs more rigorous testing before anything can be said with confidence.

Some further features to be included are:

- Making the system more flexible, so that users can make BLASTP requests if desired. This presents some challenges around getting sequence translations, and the fact that GenBank associates data with protein sequences in different ways to the nucleotide sequences. Some infrastructure has been worked on to this effect, but it needs refinement before it is integrated with the workflow.

- An algorithm to retrieve the intergenic upstream region and not just the 1000bp upstream region needs to be developed.

GenBank Type Provider

This chapter explores the method used to develop the simple type provider for using .NET Bio with GenBank data, made available in the Git Repository for this project. Also included is a discussion of the significant challenges faced - those borne both of inexperience and otherwise.

4.1 Motivation and Goals

As mentioned in the introduction, type providers can enable simplified interactions with information sources. It can be seen - from viewing the code snippets in section 3.3 of this report - that a significant amount of work must be done to deal with the .NET Bio types and the morphic nature of sequence data. The amount of times an object had to be retrieved from a generic dictionary and cast to a specific .NET Bio type is laughable. The development of a type provider for using .NET Bio with GenBank data aims to ease some of these difficulties, as well as make GenBank information sources more accessible to non-programmers through the use of well documented types and methods.

4.2 Requirements

The GenBank type provider developed in the scope of this project is intended as an experimental prototype, and does not represent a definitive model of such a type provider. From project meeting discussions as well as developing the workflow, a list of initial requirements was agreed.

The type provider should:

- allow strongly typed access to the sequence
- allow strongly typed access to genome metadata
- allow streamlined access to genes and coding sequences
- allow streamlined access to the sequence identifier
- have all data stored in memory

4.3 Development

As with the workflow, progress was inexorably slow due to inexperience and lacking documentation.

It is important to note here that almost all existing tutorials and documentation on type provider development are based on the use of the Type Providers Starter Pack, and the type provider for this project made use of this provided infrastructure.

Before proceeding, let us outline the basic structure of a type provider, as created with the support of the starter pack. Figure 4.1 neatly outlines the basic building blocks for any type provider.

```
module Mavnn.Blog.TypeProvider

open ProviderImplementation.ProvidedTypes
open Microsoft.FSharp.Core.CompilerServices
open System.Reflection

[<TypeProvider>]
type MavnnProvider (config : TypeProviderConfig) as this =
    inherit TypeProviderForNamespaces ()

    let ns = "Mavnn.Blog.TypeProvider.Provided"
    let asm = Assembly.GetExecutingAssembly()

    let createTypes () =
        let myType = ProvidedTypeDefinition(asm, ns, "MyType", Some typeof<obj>)
        let myProp = ProvidedProperty("MyProperty", typeof<string>, IsStatic = true,
                                      GetterCode = (fun args -> <@@ "Hello world" @@>))
        myType.AddMember(myProp)
        [myType]

    do
        this.AddNamespace(ns, createTypes())

[<assembly:TypeProviderAssembly>]
do ()
```

Figure 4.1: MyFirstTypeProvider by Michael Newton (2013) [7]

The important things to note here are the `[<TypeProvider>]` annotation, the adding of the created types to the namespace, and the `[<assembly:TypeProviderAssembly>]` annotation. These elements are key to ensuring that the type provider is correctly interpreted by the compiler. Another thing to note is the use of the `<@@ @@>` syntax. This is known as an FSharp quotation, the simplified explanation being that code inside a quotation is compiled in the context where the overarching feature is being used, not when the type provider itself is compiled.

However, the example in figure 4.1 is extremely limited, and does not make use of many of the features that make type providers so attractive, such as being backed by a provided static

parameter, or having constructors which allow new objects of the provided type to be created, with different data being provided.

Some of the core elements that a type provider can have are:

- Provided Constructors - these are constructors that can be passed arguments to support instantiation of new objects of the provided type, but backed by different data. They have an instantiation function which is an FSharp quotation, and the return value of this instantiation function can be passed as an input to a provided property.
- Provided Properties - these are properties of the provided type that have a name and GetterCode, where the GetterCode is an FSharp quotation that can compute the value to be provided based on the value passed to it from the constructor.
- Provided Methods - these are similar to provided properties, but instead provide methods which can do some given computation.
- Provided Static Parameters - this is a parameter that is passed in to the type definition, and is accessible to all internal components of the type provider, but is not able to be changed.
- Provided Type Definitions - these are definitions of types which can be added as nested types to the overall type provider. These can be created based on the provided static parameter of their top level type, but cannot- and are not- recomputed every time a new object of the top type is created using the constructor.

As ever, the first stage of this development process was to create a simple type provider, as an exercise in familiarization. After a significant amount of reading, and experimenting with code in the Starter Pack, a proof-of-concept type provider was developed. This type provider simply took a file path as a static parameter. This static parameter was then passed to a few different provided properties, which would perform some basic operation like return the locus tag of the gene as a string. At this stage, no data about the sequence was being stored in memory, only the file name.

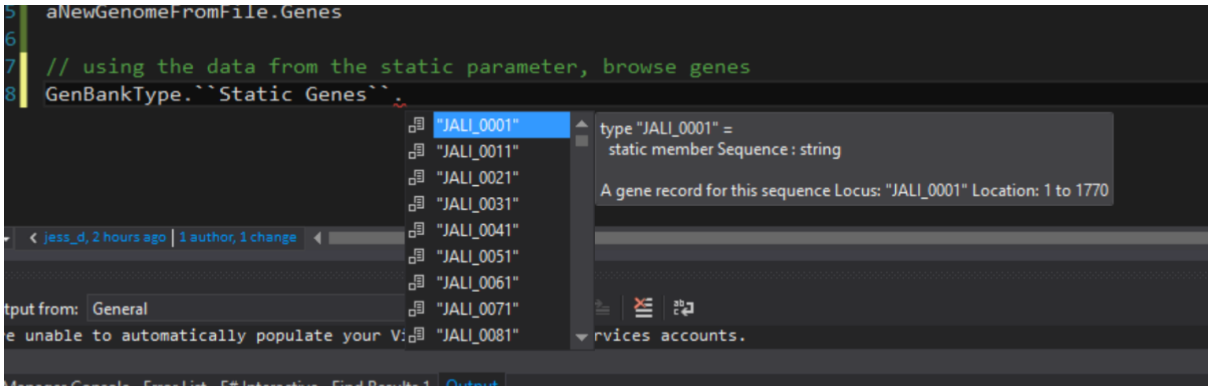

Figure 4.2: Browsing through all the genes belonging to a genome by their locus tag

From here, the next step was to work out how to get the sequence from the file name, and then store the sequence internally so that it can be passed as input to the other elements in the type provider. This proved quite challenging, as the .NET Bio types were available to the type provider internally, but not within the context of the quotations. The solution to this eventually boiled down to making sure that references in an FSharp script file are included in a very particular order, and ensuring that the script file making use of the type provider is open in a separate instance of visual studio to that which the type provider is open in.

Once this step was overcome, it was relatively straightforward to create a series of provided properties which fostered more streamlined interactions with GenBank data. Some of these properties simply offer easier access to existing .NET Bio properties, and some provide additional functionality (such as getting the sequence of a particular feature).

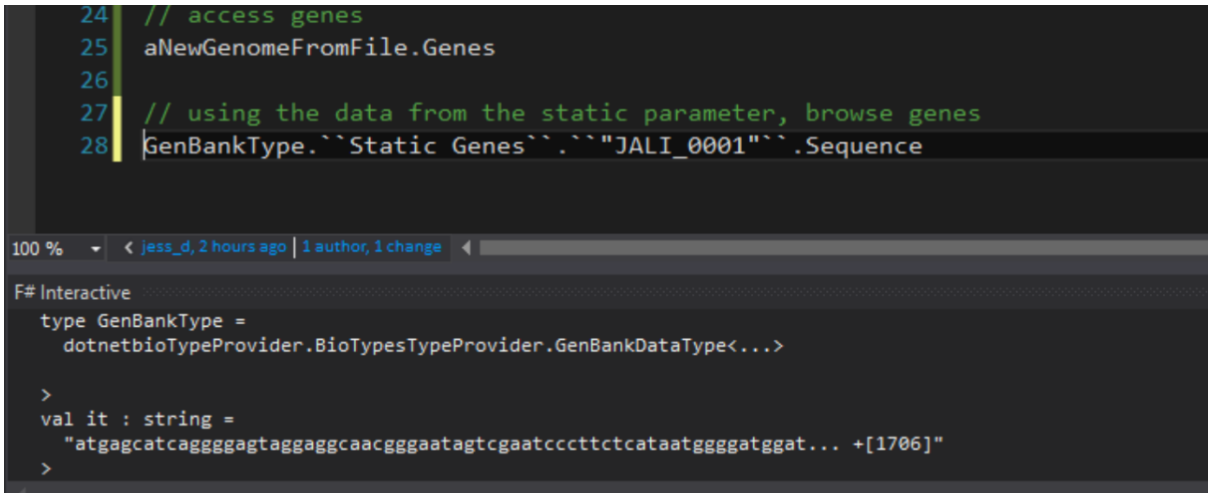

Figure 4.3: Accessing the sequence of a gene selected by locus tag

As an exploration, two ProvidedTypeDefinitions have also been added to the overall type provider - one of genes and one for coding sequences. These ProvidedTypeDefinitions generate a nested type for every recorded gene or coding sequence recorded in the file which was passed to the top level type as a static parameter. Using Intellisense, these features show up with their locus tag as the type name, and their sequences are associated with their type as a ProvidedProperty. This means that interactions can be such as that seen in figures 4.2 and 4.3. This is an interesting feature, as it provides quick and easy access to any desired feature in a genome, as opposed to having to iterate over a list. However, this is limited, in that it is only possible to provide this feature to the top level type. That is, these genes will not be updated every time a new object is created, as it is impossible to get the argument from a constructor (an FSharp quotation) and then use this to generate new types. At this stage, these features have been included with the overall type provider, but it is likely that these will be pulled out and made part of a separate provider offering.

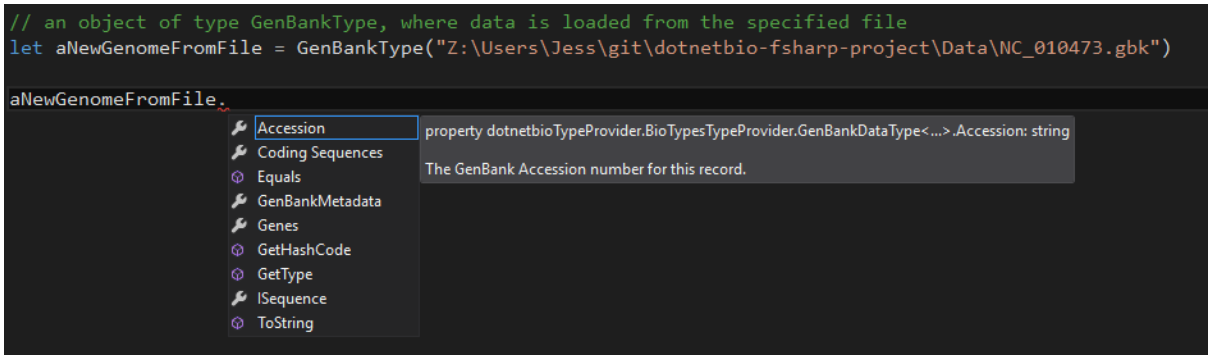

Figure 4.4: Intellisense showing some of the properties provided

The current form of the type provider can be seen in C.1. To best understand the provided properties offered, however, it is prudent to examine how a user might interact with it. Listing 4.1 outlines some key interactions that can be made with the type provider, figure 4.4 show how the xmlDoc feature of a provided type's members can help users gain context and understanding of the data they are working with, and figure 4.5 shows how easy it is for a user to print out metadata.

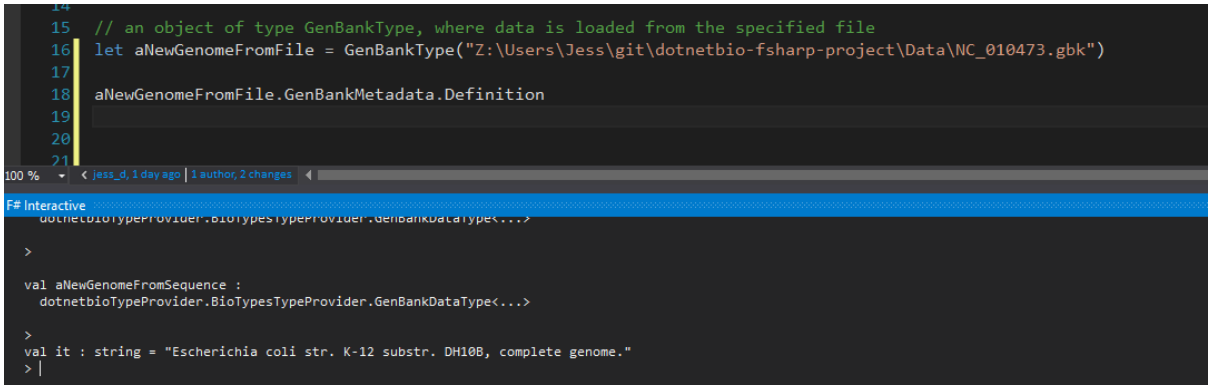

Figure 4.5: Showing how a user can easily print metadata for their genome

```
#r "System.IO"
#r "System.Collections"
#r "lib\Bio.Core.dll"
#r "lib\Bio.Desktop.dll"
#r "lib\Bio.Platform.Helpers.dll"
#r "lib\FSharp.Management.dll"
#r "lib\BioTypesTypeProvider.dll"

// creates the type using data from specified file as the template
type GenBankType = dotnetbioTypeProvider.BioTypesTypeProvider.
```

```
    GenBankDataType<"Z:\Users\Jess\git\dotnetbio-fsharp-project\Data\
    NC_012686-Chlamydia␣trachomatis.gbk">

// gives an object of the type GenBankType, where the data is the
    default data loaded from the filename above
let defaultGenome = GenBankType()

// an object of type GenBankType, where data is loaded from the
    specified file
let aNewGenomeFromFile = GenBankType("Z:\Users\Jess\git\dotnetbio-
    fsharp-project\Data\NC_010473.gbk")

// an object of type GenBankType, where data is loaded from an existing
     ISequence
let aNewGenomeFromSequence = GenBankType(defaultGenome.ISequence)

// access identifer for genome
aNewGenomeFromFile.Accession

// access genes
aNewGenomeFromFile.Genes

// using the data from the static parameter, browse genes
GenBankType.``Static Genes``.``"JALI_0001"``.Sequence
```

Listing 4.1: Showing how a user can interact with type provider

4.3.1 Comments and Known Issues

At this stage there do not appear to be any significant bugs with the type provider, though this also needs proper testing. There are a lot of features that could be integrated into this. For example, it would be nice to add method properties to the provided type to allow simplified calls to get the upstream region of a given gene. It might even be possibly to incorporate rich pattern matching into the type provider - these are all possibilities that can be explored.

Recommendations and Future Directions

There is a lot of scope to what can be done from here. The first stage is to get all components into a more robust and presentable form, before being released as a version 1.0, as well as rebuilding the workflow with the type provider as a demonstration. From here, there are many different features that could be integrated, and it would good to conduct interviews with more people working in this space to get a sense of things that can make their lives easier. Having to sift through all the different services, scattered documentation and web pages that say 'sorry, this page no longer exists', has only served as further motivation to develop tools for people in this space.

Appendices

Appendix: A

Appendix for Development with .NET Bio in FSharp

A.1 Setup Guide and Troubleshooting

A.1.1 MacOS - Mono

The screenshots and instructions below outline how to set up a new Xamarin studio project which makes use of .NET Bio.

Step 1: Create a new Xamarin Studio solution, as an FSharp console project. Your screen should then look as shown in figure A.1.

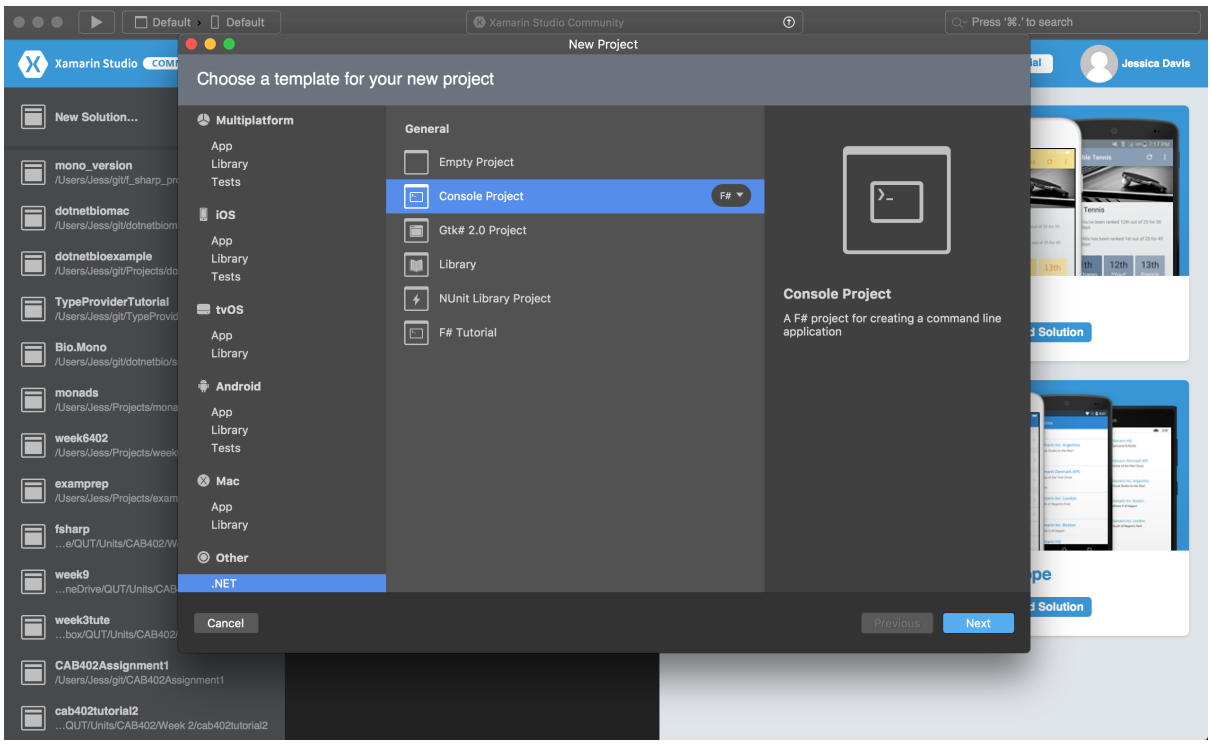

Figure A.1: Step 1

Step 2: Add the .NET Bio package, as shown in figure A.2.

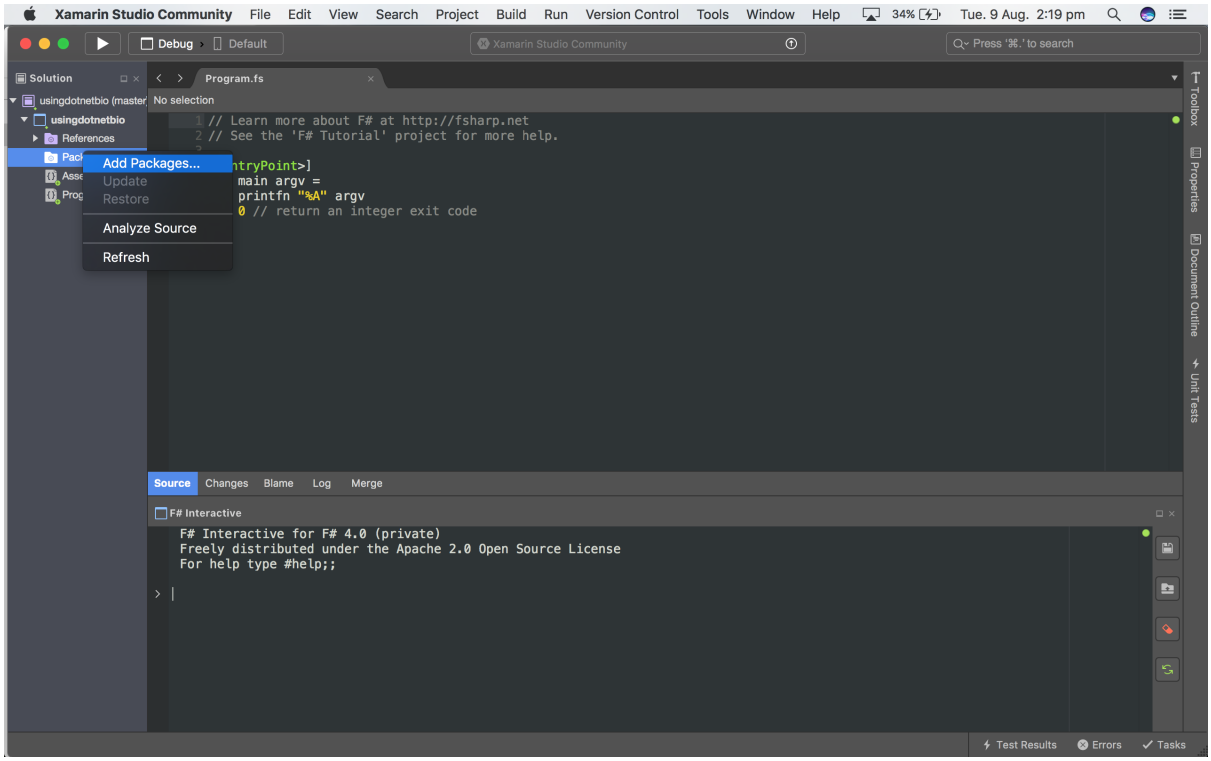

Figure A.2: Step 2

Step 3: You should then be able to use the package as required

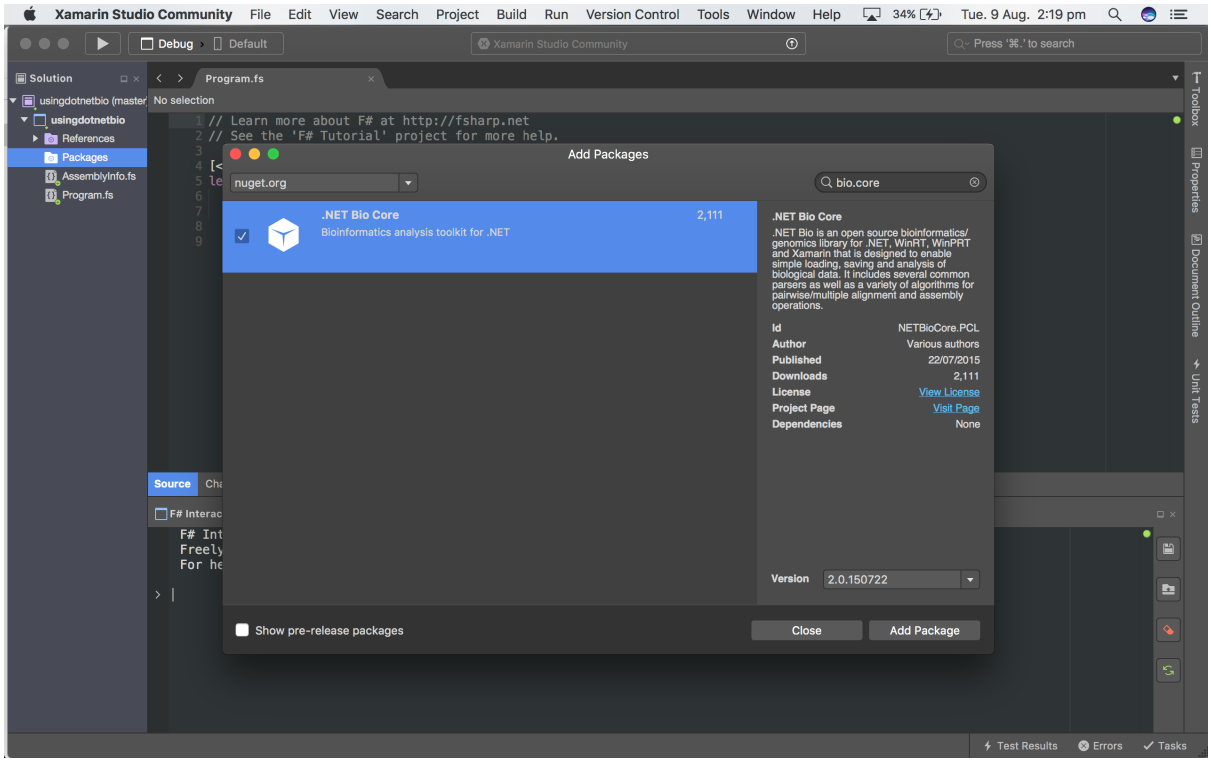

Figure A.3: Step 3

A.1.2 Troubleshooting

The solution to some common issues which arose in the setup process.

Note: All screenshots here are taken in Visual Studio, but the same general steps also apply in Xamarin Studio

Issue: The package will not work, with an error like "the package does not work with .NET Framework v x.x.x".

Solution: Open the project options, and ensure that the target framework is .NET Framework v4.5 or higher.

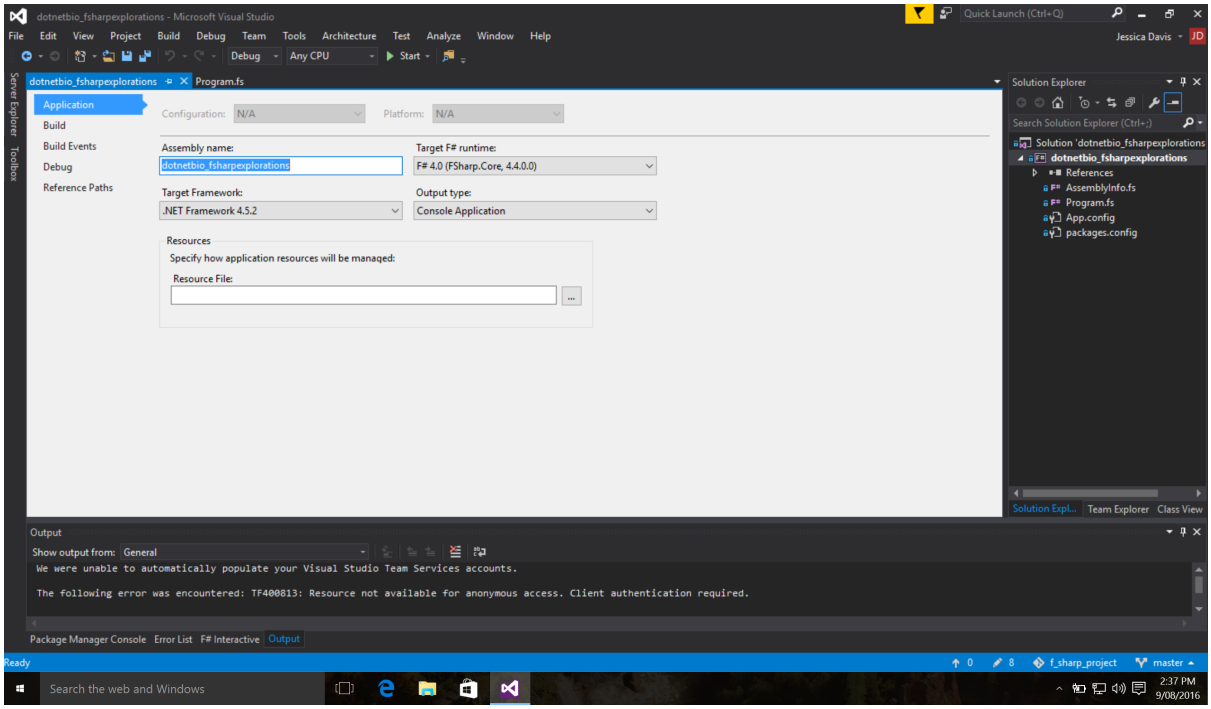

Figure A.4: Troubleshooting - package does not work with target framework

Issue: The code will not work in FSharp interactive

Solution: When using FSharp interactive a reference must be provided to the library being used.

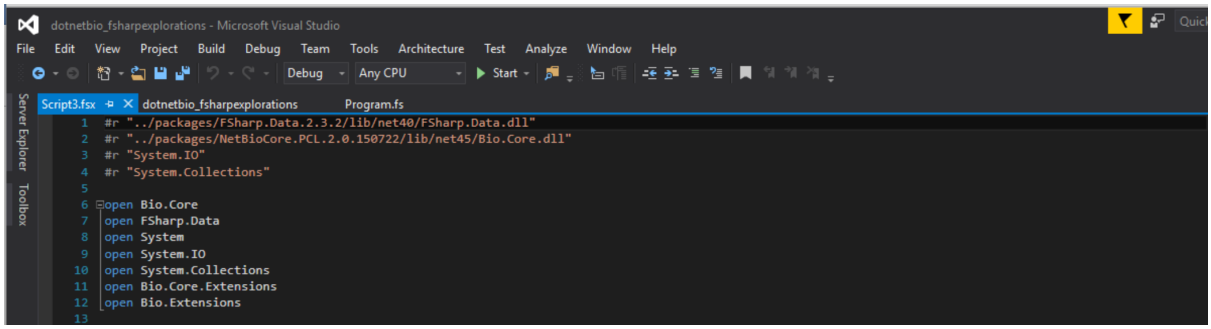

Figure A.5: Troubleshooting - code does not work in FSharp interactive

It must be ensured that these lines are executed in FSharp interactive before trying to execute any code or use any classes or methods from the library.

Appendix: B

Appendix for Upstream Analysis Workflow

B.1 Workflow Code Snippets

B.2 BLAST Result Stream Workaround

```
let getParsedBlastResult (blastResponseStream:Stream) =
// need to write to file, then read from file
let resultStr = blastResponseStream |>
(fun brs ->
use sr = new StreamReader(brs)
sr.ReadToEnd()
)

// blastResponseStream no longer required; close
blastResponseStream.Close()
blastResponseStream.Dispose()

File.CreateText(Directory.GetCurrentDirectory() + "/Data/blastresult.
    txt") |>
(fun (rs:string) (f:StreamWriter) ->
f.AutoFlush <- true
f.Write(rs)
// make sure file is appropriately terminated, otherwise all xml will
    not be written from stream to file
f.Close()
f.Dispose()
) resultStr

new FileStream(Directory.GetCurrentDirectory() + "/Data/blastresult.txt
    ", FileMode.Open, FileAccess.Read) |>
(fun fs ->
let bParser = new Blast.BlastXmlParser()
// should only be one result set in stream
bParser.ParseOne(fs)
)
```

Figure B.1: Workaround method get a parsed BLAST result

B.3 Bidirectional best hit algorithm

```
let getBlast2BestHitsForAllGenes filePath database program genome (
    allGenes:seq<BlastOutputRecord>) =
allGenes
|> Seq.map (fun singleInitialBlastOutputRecord ->
let ``original gene`` = singleInitialBlastOutputRecord.QueryGene
let blast2extraparams = [new KeyValuePair<string, string>("ENTREZ_QUERY
    ", ``original gene``.ID + "[Accession]")]

let isBiDir =
singleInitialBlastOutputRecord.BlastResultSequences
|> Seq.map(fun ibrs -> ibrs.FeatureSequence)
|> getBlastResultsForAllGenes filePath database program
    blast2extraparams
|> fullSeqForHits genome
|> Seq.map(fun blastOrForHit ->
let bestHitInGenome =
blastOrForHit.BlastResultSequences
|> Seq.sortBy (fun brs2 -> brs2.HitEval)
|> Seq.toList
|> (fun x -> x.[0])

if (bestHitInGenome.FeatureSequence.ToString() = ``original gene``.
    ToString()) then
// associate the now known location data with the query gene for later
    use
let fmd = bestHitInGenome.FeatureSequence.Metadata.["Feature"] :?> Bio.
    IO.GenBank.FeatureItem
``original gene``.Metadata.Add("Feature", fmd)
true
else
false
)

let bidirbesthits =
Seq.map2 (fun isBiDirel blastrsel ->
if isBiDirel then
blastrsel
else
{FeatureSequence=null; HitSequence=null; Hit=null; HitEval=0.0}
) isBiDir singleInitialBlastOutputRecord.BlastResultSequences
```

```
|> Seq.filter (fun x ->
match x.FeatureSequence with
| null -> false
| _ -> true
)

{QueryGene=``original gene``; BlastResultSequences=bidirbesthits}
)
```

Listing B.1: Bidirectional best hit algorithm

B.4 User interactions with workflow

```
// step 1 - define query genes and download genome for query genes
let queryGenes = ourSeq |> Seq.take 1
let ``accession number of the gene which your sequences are from`` = "
    U00096" // change this to the accession of the genome you are
    working with
let originGenome = allGenomesFromAcessions [``accession number of the
    gene which your sequences are from``]

// step 2 - perform first blast with this queryseq
let database = "nr"
let program = Bio.Web.Blast.BlastProgram.Blastn
let ``maxiumum number of results`` = "2"

let ``entrez query for limiting initial blast search`` =
"txid1423␣[ORGN]␣OR␣txid1392␣[ORGN]␣OR␣txid632␣[ORGN]␣OR␣txid1301␣[ORGN
    ]␣OR␣txid813␣[ORGN]␣OR␣txid1313␣[ORGN]␣OR␣txid1280␣[ORGN]"
|> (fun s -> s.Replace(' ', '+'))
let blast1extraparams = [new System.Collections.Generic.KeyValuePair<
    string, string>("MAX_NUM_SEQ", ``maxiumum number of results``)]
// you can use the entrez query to limit the search to specific
    organisms, at the moment, this is set to only BLAST against
// Bacillus subtilis (taxid:1423)
// Bacillus anthrasis (taxid:1392)
// Yersinia pestis (taxid:632)
// Streptococcus (taxid:1301)
// Chlamydia trachomastis (taxid:813)
// streptococcus pneumonia (taxid:1313)
// staphylococcus aureus (taxid:1280)
let blast1 = getBlastResultsForAllGenes filePath database program
```

```
    blast1extraparams queryGenes

// step 3 - download genomes for the results
let listOfAccessions = uniqueAccessions blast1
let genomes = allGenomesFromAcessions listOfAccessions

// step 4 - get the full sequences for each of the hits
let fullSeqsForHits = fullSeqForHits genomes blast1 |> Seq.toList

// step 5 - get only bidirectional best hits
let biDirectionalBestHits = getBlast2BestHitsForAllGenes filePath
    database program originGenome fullSeqsForHits |> Seq.toList

// step 6 - analyse!
// each object in the biDirectionalBestHits is a "BlastOutputRecord"
    which has two properties
// 1. QueryGene (the gene that was sent as input to BLAST)
// 2. BlastResultSequences (a sequence of SingleBlastHit objects which
    contains only the bidirectional best hits for the QueryGene)
// SingleBlastHit objects have 4 properties:
// 1. FeatureSequence (an ISequence corresponding to the full sequence
    matching the BLAST hit object)
// 2. HitSequence (an ISequence representation of the hit alignment
    sequence returned from BLAST, usually a subsequence of some other
    feature)
// 3. Hit (Bio.Web.Blast.Hit object, contains stats about Blast result)
//4. HitEval (The Evalue associated with this hit)

// you can get the upstream region of your input sequence like this:
let upstreamOfInputSeq =
biDirectionalBestHits.[0]
|> (fun blastoutputrecord ->
blastoutputrecord.QueryGene.Metadata.["Feature"] :?> Bio.IO.GenBank.
    FeatureItem
)
|> getUpstreamRegion genomes.[0]

// you can get an upstream region for a particular hit on your input
    sequence like this:
let upstreamSelectedHit =
biDirectionalBestHits.[0]
|> (fun blastoutputrecord ->
```

```
blastoutputrecord.BlastResultSequences |> Seq.toList
|> (fun x -> x.[0].FeatureSequence.Metadata.["Feature"] :?> Bio.IO.
    GenBank.FeatureItem)
)
|> getUpstreamRegion genomes.[0]
```

Listing B.2: Showing how a user might perform the entire workflow

Appendix: C

Appendix for GenBank Type Provider

C.1 Type Provider Code Snippets

```
namespace dotnetbioTypeProvider.BioTypesTypeProvider

open System
open System.Reflection
open System.IO
open System.Collections.Generic
open Samples.FSharp.ProvidedTypes
open Microsoft.FSharp.Core.CompilerServices
open Microsoft.FSharp.Quotations
open Bio
open Bio.IO
open Bio.Core
open Bio.IO.GenBank
open BioMethods

[<TypeProvider>]
type public BioTypesTypeProvider(config: TypeProviderConfig) as this =
inherit TypeProviderForNamespaces()

let ns = "dotnetbioTypeProvider.BioTypesTypeProvider"
let asm = Assembly.GetExecutingAssembly()

// Type to export from type provider
let genBankTy = ProvidedTypeDefinition(asm, ns, "GenBankDataType", Some
    (typeof<obj>), HideObjectMethods=true)

// paramaterise type by file to use as template
let filename = ProvidedStaticParameter("filename", typeof<string>)

// Create the type
do genBankTy.DefineStaticParameters(
[filename],
(fun typeName [| :? string as filename |] ->
```

```
// resolve filename relative to resolution folder
let resolvedFilename = Path.Combine(config.ResolutionFolder, filename)

let geneTopType = ProvidedTypeDefinition("Static Genes", Some typeof<
    obj>)
let cdsTopType = ProvidedTypeDefinition("Static Coding Sequences", Some
     typeof<obj>)

// Define the provided type
let ty = ProvidedTypeDefinition(
asm,
ns,
typeName,
Some(typeof<obj>)
)

// Parameterless constructor that loads file used to define schema
let ctor0 = ProvidedConstructor(
[],
InvokeCode = fun [] -> <@@ getGenomeSeq resolvedFilename @@>
)

ctor0.AddXmlDoc "Parameterless constructor"
ty.AddMember ctor0

// Constructor that takes file name to load
let ctor1 = ProvidedConstructor(
[ProvidedParameter("filename", typeof<string>)],
InvokeCode = fun [filename] -> <@@ getGenomeSeq %%filename @@>
)

ctor1.AddXmlDoc "Constructor that takes filename"
ty.AddMember ctor1

// Constructor that takes an existing ISequence
let ctor2 = ProvidedConstructor(
[ProvidedParameter("sequence", typeof<ISequence>)],
InvokeCode = fun [sequence] -> <@@ (%%(sequence):ISequence) @@>
)

ctor2.AddXmlDoc "Constructor that takes an existing ISequence"
```

```
    ty.AddMember ctor2

    let originalISeq = ProvidedProperty(
    propertyName = "ISequence",
    propertyType = typeof<ISequence>,
    GetterCode = (fun args ->
    <@@
    ((%%(args.[0]) : obj) :?> ISequence)
    @@>
    )
    )

    originalISeq.AddXmlDoc "This␣is␣the␣original␣.NET␣Bio␣ISequence␣object␣
        representing␣this␣sequence"
    ty.AddMember originalISeq

    let genbankmetadata = ProvidedProperty(
    propertyName = "GenBankMetadata",
    propertyType = typeof<GenBankMetadata>,
    GetterCode = (fun args ->
    <@@
    ((%%(args.[0]) : obj) :?> ISequence)
    |> getMetadata
    @@>
    )
    )

    genbankmetadata.AddXmlDoc "The␣.NET␣Bio␣GenBankMetadata␣object␣for␣this
        ␣sequence"
    ty.AddMember genbankmetadata

    let accession = ProvidedProperty(
    propertyName = "Accession",
    propertyType = typeof<string>,
    GetterCode = (fun args ->
    <@@
    ((%%(args.[0]) : obj) :?> ISequence)
    |> getMetadata
    |> (fun y ->
    match y.Accession.Primary with
    | "" | null -> "No␣accession␣number␣for␣this␣record"
    | _ -> y.Accession.Primary
```

```
)
@@>
)
)

accession.AddXmlDoc "The␣GenBank␣Accession␣number␣for␣this␣record."
ty.AddMember accession

let genes = ProvidedProperty(
propertyName = "Genes",
propertyType = typeof<list<FeatureItem>>,
GetterCode = (fun args ->
<@@
let sequence = ((%%(args.[0]) : obj) :?> ISequence)
sequence
|> getMetadata
|> (fun md -> md.Features.Genes)
|> Seq.map(fun gene ->
findFeatureSequence sequence gene
)
|> Seq.toList
@@>
)
)
genes.AddXmlDoc "All␣the␣recorded␣genes␣in␣this␣sequence,␣with␣their␣
    associated␣sequences␣from␣the␣genome."
ty.AddMember genes

let codingSequences = ProvidedProperty(
propertyName = "Coding␣Sequences",
propertyType = typeof<list<FeatureItem>>,
GetterCode = (fun args ->
<@@
let sequence = ((%%(args.[0]) : obj) :?> ISequence)
sequence
|> getMetadata
|> (fun md -> md.Features.CodingSequences)
|> Seq.map(fun cds ->
findFeatureSequence sequence cds
)
|> Seq.toList
@@>
```

```
    )
    )
    codingSequences.AddXmlDoc "All␣the␣recorded␣coding␣sequences␣in␣this␣
        sequence,␣with␣their␣associated␣sequences␣from␣the␣genome."
    ty.AddMember codingSequences

    let genesAsTypes =
    let genomeSeq =
    resolvedFilename
    |> getGenomeSeq
    genomeSeq
    |> getMetadata
    |> (fun md -> md.Features.Genes)
    |> Seq.map (fun gene ->
    let lt = if gene.LocusTag.Count >= 0 then gene.LocusTag.[0] else ""
    let geneDesc =
    "A␣gene␣record␣for␣this␣sequence␣\n␣Locus:␣" + lt + "\n␣Location:␣" +
        gene.Location.StartData + "␣to␣" + gene.Location.EndData
    let geneSeq = findFeatureSequence genomeSeq gene |> (fun x -> x.
        ToString())
    [lt; geneDesc; geneSeq]
    )
    |> Seq.map (fun info ->
    // add each gene as a member
    let geneTp =
    let geneTp = ProvidedTypeDefinition(info.[0], Some typeof<obj>)
    let genseq = info.[2]
    let geneProp = ProvidedProperty("Sequence", typeof<string>, IsStatic =
        true, GetterCode=(fun args -> <@@ genseq @@>))
    geneProp.AddXmlDoc "A␣string␣containing␣the␣base␣pairs␣which␣make␣up␣
        this␣genomic␣feature."
    geneTp.AddXmlDoc info.[1]
    geneTp.AddMember geneProp
    geneTp
    geneTp
    )
    |> Seq.toList

    geneTopType.AddMembers genesAsTypes

    ty.AddMember geneTopType
```

```
        // the type
        ty
        )

        )

        // add type to the namespace
        do this.AddNamespace(ns, [genBankTy])

        // tell compiler that this assembly has type providers
        [<assembly:TypeProviderAssembly>]
        do ()
```

Listing C.1: Current state of the type provider

Bibliography

[1] Ardeshir Bayat. Bioinformatics. 324(7344), 04 2002.

[2] Daniel A. Dalquen and Christophe Dessimoz. Bidirectional best hits miss many orthologs in duplication-rich clades such as plants and animals. 5(10), 09 2013.

[3] dotnetbio. Dotnetbio/bio, 10 2016.

[4] R Fielding, UC Irvine, J Gettys, J Mogul, H Frystyk, and T Berners-Lee. Hypertext transfer protocol – http/1.1, 01 1997.

[5] National Center for Biotechnology Information. Genbank overview, 03 2016.

[6] Jack Fox. F# guide, 06 2016.

[7] Michael Newton. Type providers from the ground up, 12 2013.

[8] National Library of Medicine. Blast: Basic local alignment search tool, 2016.

[9] Tomas Petricek and Gustavo Guerra. F# data: Library for data access, 11 2016.