Using Functional Languages and Declarative Programming to Analyze Large Datasets: LINQtoROOT

G. Watts
Box 351560
Department of Physics, University of Washington
Seattle, WA, 98195
USA
gwatts@uw.edu

Abstract. Modern HEP analysis requires multiple passes over large datasets. For example, one has to first reweight the jet energy spectrum in Monte Carlo to match data before you can make plots of any other jet related variable. This requires a pass over the Monte Carlo and the Data to derive the reweighting, and then another pass over the Monte Carlo to plot the variables you are really interested in. With most modern ROOT based tools this requires separate analysis loops for each pass, and script files to glue to the two analysis loops together. A prototype framework has been developed that uses the functional and declarative features of C# and LINQ to specify the analysis. The framework uses language tools to convert the analysis into C++ and runs ROOT or PROOF as a backend to get the results. This gives the analyzer the full power of an object-oriented programming language to put together the analysis and at the same time the speed of C++ for the analysis loop. The tool allows one to incorporate C++ algorithms written for ROOT by others. The code is mature enough to have been used in ATLAS analyses. The package is open source and available on the open source site CodePlex.

1. Introduction
Functional and declarative programming has been around for many years – especially in academic language research. Only recently elements of this programming style have started entering mainstream programming languages – for example, lambda functions in C++11. The functional style of programming tends to be more expressive than the standard Object Oriented style. Another approach to specifying an algorithm - declarative programming – has also started to receive mainstream attention. In declarative programming one expresses the desired result, not how the result should be computed. The details of the computation are abstracted away – it is left to the runtime and the compiler to affect the programmer’s desire. Many declarative programs use a functional approach to express the desired calculation. This paper discusses an experimental framework that applies these two styles to processing large particle physics datasets stored in the ROOT [1] data format.

This paper discusses the motivation, some history, and the overall guiding principles and design decisions in Section 2. In Section 3 some short examples are shown and explained to set the context for the following section, Section 4. a quick introduction to C# and Language Integrated Query (LINQ) which this work uses as its declarative language. Following that is Section 5. which describes the implementation of the backend in detail. Finally, Section 6. gives a short summary of future work.

2. Background and Motivation
This project started because of the author’s need to simplify analysis – even the process of making a single plot. A modern analysis in a large LHC experiment, like ATLAS, requires many steps on potentially very large datasets. For example, if one wished to make a distribution of jet transverse momentum ($p_T$) for jets within a specific pseudo-rapidity ($\eta$) range in Monte Carlo and compare it to data, one had to do the following steps:

1. Run over all data to generate a template jet $p_T$ spectra over the full $\eta$ range.
2. Run over all Monte Carlo to generate a similar template jet $p_T$ spectra.
3. Calculate a weighting function to correct the Monte Carlo jet $p_T$ spectra as a function of jet $p_T$ by dividing the results from steps 1 and 2.
4. Apply the weighting function to the Monte Carlo sample, keeping only the jets in the desired $\eta$ range, and finally plot the sought after jet $p_T$.

Steps 1, 2, and 3 require separate runs over large datasets. Steps 1 and 2 need be done once to calculate the weighting function, and likely step 3 will be done repeatedly as selection cuts are refined, etc. This is a large amount of accounting and jobs that must be kept track of. Jobs submitted, intermediate histograms (like the weighting function) tracked, etc. The job specification, C++ code to generate the code, etc., will be spread out over a number of files and programming languages.

The idea for LINQtoROOT is to put all of this back into one source code file, keeping the individual steps local so that at a glance the analyzer can see what the analysis steps are. Being able to see the logical flow of the analysis quickly should have a number of benefits, including maintaining the analysis, reducing bugs, and helping those analyzers that can spend time only intermittently on the analysis. The code is a series of declarations of intent: make this plot on this dataset with these cuts. The compiler and the run-time libraries should take care of all the mechanics of running locally in ROOT, or perhaps on PROOF (a distributed version of ROOT, [2]).

This project’s initial goals have evolved since it was started two years ago, mostly driven by lessons learned by the author. The major goal is to make analysis easy. Specifically, this translates to several technical goals: being able to run multiple passes on dataset(s) in a single source file, strongly support the iterative approach to developing an analysis, and a definite absence of boilerplate code. The analysis code it produces must be optimized. Note that this project, even in its first version, was never geared towards implementing the next generating tracking algorithm, or even being able to do something like one of the modern very complex analyses (Higgs searches).

The list of goals (plus the author’s personal prejudices) results in a system that assumed all data is stored in the ROOT TTree format, that all scanning of the TTree’s should be by compiled C++, and that results should be cached so repeated requests for the same plot or data return the cached result. Further, the interface to ROOT and to PROOF as a back end needed to be basically transparent.

2.1. Code is the User Interface
Before arriving at this solution the author make three aborted attempts: A graphical based approach and a two workflow based approaches. It is possible to quickly summarize the reasons for failure of these approaches. Graphic approaches failed first because information density and scaling was not good enough. A full analysis, even a medium sized one, could not be represented readably on a single screen, and the simple parts of an analysis were a chore to code. More traditional work flow solutions failed because several languages were usually required: C++ for the running on the data, the workflow language (or GUI) for putting the steps together. Further, the analysis often ended up being distributed through several files.

The unsatisfactory results of these attempts are what drove the author to look for a single-file, single-language approach to doing multi-pass analyses of data. The same code could describe the actual data analysis as well as the manipulation of the results – fitting, plot manipulation, etc. The GUI for this system is the code. This has the added benefit that most websites and source control systems need no extra support for detailed dealing with the files. However, the development environment does need some significant modification.
ROOT actually has the seeds of this solution built into its TTree::Draw method. The user declares to TTree::Draw the plot they wish to make:

\[
\text{root [19] CollectionTree->Draw("AntiKt4TopoJets_pt/1000.0", "abs(AntiKt4TopoJets_eta)<1.0")}
\]

Note that in this code there is no information about how to run over the TTree – just a declaration that the jet $p_T$ in units of GeV should be made for jets with $|\eta| < 1.0$. Obviously this isn’t very composable and works only for the most simplistic of plots. But this is the kernel of the solution presented here.

3. Usage

Figure 1 shows the complete code to generate a single plot of jet $p_T$. This is the simplest usage of this tool.

![Figure 1: Code to generate a single plot of jet $p_T$ from a file called sample.root. The plot is shown in the inset.](image)

Lines 1-14 are mostly setup and standard C# code for program header. Line 15 creates the data source from the TTree. Passed in is a FileInfo object which points to the source file. One can pass an array of files, or PROOF URL’s here. The data variable should be considered an iterator of events. Lines 16-18 convert the stream of events into a stream of jets. Finally, line 19 makes one entry per jet in a new histogram. Loosely in this paper this is often referred to as a query. For each entry it plots the jet $p_T$ in units of GeV (note the automatic conversion of the string pt to $p_T$ in the plot title). The variable plot holds a TH1F ROOT histogram object, and lines 20-23 do the usual work of storing the histogram in a ROOT output file.

Before one can write the code above there is some one-time setup that must be done. First, one must use the Nuget [3] tool to add the LINTToTTree-v5.30.01.win32.vc10 package to the project. Nuget is an open-source package manager for Windows that gives the user easy access to over 8000 open source libraries. Note that you must match the version number to your local version of ROOT. Before you do this you must configure Nuget to fetch packages from the ROOT.NET package feed: http://deeptalk.phys.washington.edu/rootNuGet/nuget. Once the library has been added to your project in Visual Studio 2010 you can then use the command Write-TTree-MetaData to scan and parse the TTree. Finally, some modifications were made to the metadata generated by this command so we could use simple names like Jets and pt rather than the true names in the TTree. Some of this will be described in further detail later on. The configuration of Nuget is done once on a machine, the libraries must be added to each project once, and finally, you need to run the Write-TTree-MetaData command only once. The author has found that the metadata is continuously modified during analysis development and those modifications are automatically compiled as part of the build process.
Figure 2 shows one additional, major feature: the batching of queries.

```csharp
static void Main(string[] args)
{
    var data = ROOTLinqQueryableCollection.CreateQueryable(new FileInfo("/.../..sample.root"));
    var outf = new System.FutureFile(new FileInfo("results.root"));
    var jets = from j in data
                select j;
    var pl = jets
        .FuturePlot("allJets", "jet pt for all jets; pt [GeV]; 50, 0.0, 100.0, j => j.pt / 1000
                        .Save(outf);
    var pl2 = jets
        .Where(j => Math.Abs(j.eta) < 0.5)
        .FuturePlot("centralJets", "jet pt for central jets; pt [GeV]; 50, 0.0, 100.0, j => j.pt / 1000
                        .Save(outf);
    pl2.ExtractValue((p1, p2) => (p22=p2.Divide(p1)); p2.SetName("ratio"); return p22; )
    .Save(outf);
    outf.Close();
}
```

Figure 2: Code that makes two plots and calculates the ratio in a single pass over the data.

Line 15 is actually creating a TFile for output – but is wrapping it in a helper object, as will shortly be explained. Line 19 generates a future plot of the jet $p_T$. The FuturePlot method returns a future, or promise, of the plot. The runtime doesn’t actually start generating the plot. Line 21 associates that future plot with the output file – so when the plot is actually done, it will be stored in the file. Line 22 does the same thing for jets that have $|\eta| < 0.5$. Line 28 causes the TFile helper object to attempt to dereference every future value that has been saved, and this, in turn, triggers a run on all the data. The advantage of this is both plots are generated in a single pass over the data. When the data is a terabyte or more, doing a single pass is crucial to going fast. Line 26 just shows how one can manipulate the future results without triggering a run over the data. Manipulating the futures is cumbersome, as can be seen here. Referencing the actual object in the future will cause the analysis to run, perhaps prematurely. Thus all references and histogram manipulations must be protected in these function calls. Extensive use of C# language features – lambda functions in particular – are made to ease this non-intuitive aspect.

The expressions are composable. It is possible to build the cuts, one line at a time, in response to an if statement, for example. You can store the cuts in a table, and apply them one after the other trivially by simply appending them. In fact, it is so easy to compose the cuts that one must be careful not to over-do it: some simple analyses jobs have quickly ballooned to 1000’s of plots because the composable capabilities of this system are so good.

Plots are not the only thing that can be returned by the infrastructure. It is possible to count the number of events, for example. Arbitrarily complex data structures can be generated, though there is no current support for them in LINQtoROOT.

Further documentation, along with bug reports, etc., can be found on the CodePlex open source site: http://linqtoroot.codeplex.com/.

4. Quick Introduction to LINQ and C#

C# and LINQ were the technologies chosen to implement this project. C# is a language much like Java – a reduced set of C++ [4]. However, unlike Java, and until recently C++, it includes a number of modern features – like lambda functions (inline function definitions).

C# 3.0 added another key feature: Language Integrated Query (LINQ). While the syntax is reminiscent of the database language SQL, it is actually a rather deep language extension based on research in functional languages. Figure 3 shows an example of a simple query that counts all elements of an array that are larger than 10. Lines 12-14 show the simple LINQ syntax – the query syntax built directly into the language. This syntax is just syntactic sugar for the functional form, shown on line 17.
Figure 3: Examples of LINQ in C#.

What makes LINQ work and this project possible can be seen in the signature of the Where method:

```csharp
public static IQueryable<TSource> Where<TSource>(this IQueryable<TSource> source, Expression<Func<TSource, bool>> predicate);
```

The IQueryable generic object represents the stream of data – the array. It is, for all intents and purposes, a fancy iterator. The Where method takes a source stream of data and produces a result stream of data, and if the element passes the lambda function predicate, it will copy it onto the result data stream. Note the declaration of predicate: Expression<Func<TSource, bool>>. In short, a function that accepts the element type (integer in line 17 of Figure 3) and returns a bool – the very definition of a filter function. Except for the Expression template surrounding it all. This tells the compiler that the argument isn’t a lambda function, but rather an Abstract Expression Tree (AST) that describes the lambda function. In this example, the AST represents the “>” (greater-than) operation, with an argument on one side of it and the 10 on the other. This is a data structure, not code, and can be manipulated and interpreted by Where method. It could, for example, be translated into C++ code.

This feature of the language is crucial to this project. A complete AST describing exactly what should be plotted or counted or calculated is passed to the LINQtoROOT back-end and translated into C++ code. The C++ code is then compiled in ROOT using ACLiC or passed over the wire to a PROOF server and compiled there. It is then run against the full TTree and the result returned. Any language besides C# could have been used for this as long as it supported some way of treating code as data.

The new C++, C++11, includes lambda functions. It would have been possible to construct a framework like this in C++11 as long as the jobs always ran locally. As soon as code has to be shipped across the wire, where different versions of ROOT, the OS, etc., come into play, one needs the capability of shipping source code, not binary code.

4.1. Other Helpful Language Features

There were several reasons that the C# language was chosen. First, it is C++-like (similar to Java), and a full blown no-compromise programming language. This means any plot manipulations, etc., required after running a query can be done by the programming language.

Second, C# has a fairly good interface to ROOT. This is a result of some previous work by the author, ROOT.NET [5]. ROOT.NET gives any .NET language access to most of ROOT: all objects, global variables, enums, etc., through a reasonably fast static translation layer. The LINQtoROOT library makes heavy use of this to access basic ROOT objects, trees, and PROOF. Though the details of how this access is accomplished may be of interest, for LINQtoROOT it is just as a simple pass-through library: the calls to ROOT in C# look almost identical to C++. The only difference is the sometimes different scoping life-times for .NET garbage collected objects and ROOT’s complex object ownership rules.

There are at least two other features of C# which make this project simpler. Extension methods allow the programmer to add a new method to an already defined object or interface. For example the
Plot and FuturePlot demonstrated above calls are actually extension methods. Extension methods do not allow internal access to the object’s private data – so they are simply syntactic sugar. However, it allows one to write code that looks like Figure 2, which would not be possible otherwise without subclassing a number of objects unnecessarily.

Attribute programming is also heavily used in this project. A method or class or property or field can have user defined attributes attached to it. These attributes can contain arbitrary information, but do not compile to executable code. At runtime it is then possible to query for attached attributes. This is used, for example, to tell the translation system that a field name called pt is actually referencing a TTree leaf called AntiKt4TopoJets_pt.

5. Implementation
The ideas behind this work are general, and almost any implementation language that has a good interface to ROOT and has the ability to treat code as data should be able to support approach to analysis. Other languages that do not have this capability built-in will have to generate a good deal more of the infrastructure, of course.

5.1. Architecture
Figure 4 shows the flow of control and data in the system when a query is coded that contains a directive like Plot. Much of the work is done by the C# compiler and C# run-time libraries. Towards the end there is a handoff to the LINQtoROOT library and an open source library used to simplify the LINQ integration.

When the compiler compiles a query statement, like the one shown in Figure 1, it recodes all the Where, etc. calls as data – a large expression tree: an AST.

The C# run-time library is passed this AST along with some data that represents the ROOT TTree that user wants to process the query against. Unfortunately, the AST is in a rather raw form. For example, any constants that are in line will be represented symbolically rather than just by their value. Any function calls that can be made before the query is run have not been called. In short, this is the most general form of the expression tree, but there is some significant optimization that can be done before it is translated to C++.

This is where the re-linq library comes in [6]. An open source library, hosted on CodePlex, it makes many optimizations to the initial AST provided by the C# compiler and run-time. Further, it provides a number of very useful AST traversal tools that alleviate the need of writing the boilerplate that is required to drive the translation.
After the re-linq library finishes optimizing the expression tree it is handed off to the LINQtoROOT runtime library. This library will check to see if the plot has been previously made (and cached). If not, it will generate the C++ code required to make the plot.

The LINQtoROOT runtime library, finally, decides if the code should be run on a PROOF server or locally in-process depending on the original specification of the dataset: a file or proof URI.

If a FuturePlot is requested, LINQtoROOT gets called the same way as above, but instead of generating a Plot, it returns a token and queues the query. The first time a request is made to generate a real plot all queued queries are rendered to C++, combined and then run.

5.2. The Cache
The cache is very simple. The architecture and the nature of a LINQ query means that a complete definition of the requested plot (all cuts, etc.) and input constants or data are all in one place before the LINQtoROOT library is invoked. This includes the query itself, like cuts and track-jet matching code, variables that are to be cut on, etc. It also includes any data – for example if a histogram or a TFunction is being passed in to apply a weighting value. A hash is calculated by translating the query to text and calculating a hash for any ROOT objects and input files.

The query is trivially translated to text using some debugging infrastructure in the re-linq library. Variable names that are internal to the query (implied loop indices) are normalized. The hash for ROOT objects is calculated by serializing the objects to an in-memory buffer and then calculating a hash for the resulting string of bytes.

Everything is written out to disk in a well-known, per-program computer-wide file database. This includes the query itself, a list of the input datasets, the hash of the ROOT objects, and, finally, the resulting plot. If there is a hash key hit, all of this is read in and tested before the object is returned to the user.

One of the trickiest parts of the cache code is understanding object ownership and the different ownership chains that seem to exist in a local run and a PROOF based run. However, coding ROOT is full of dealing with object ownership issues.

It is possible to disable the cache for a particular data source to force a complete run.

5.3. Translating the Query
The query is translated from the AST to C++ in a two of steps. The first step is the translation from the AST to a series of execution objects that are finally optimized, combined with other pending queries, and translated to C++ by the backend. This C++ is compiled by ROOT’s built-in ACLiC or by PROOF and run.

Many TTree’s are not in a structure that is suitable for easy use, especially in a functional environment. For example, the \( p_x, p_y, p_z \) and \( p_T \) are all in different leaves – there is no jet object. The LINQtoROOT allows one to simulate an object structure as a convenience (in database parlance this is often called changing the shape of the data). These “conveniences” are translated during the conversion of the AST to the series of execution object. In order for this to happen, and for just straight up processing, the data structure of the TTree must be known ahead of time. This requires a pre-processing step that must be done once.

5.3.1. Tree Preparation
A standalone program, the metadata scanner, scans a TTree and generates several XML files that describe the tree’s structure. The user can modify some of these XML files to alter the shape of the TTree, including grouping values together, making pointers, and renaming leaves. The output is an XML file, a portion of which is shown in Figure 5.
These extra XML files are added to the MSBUILD file (an advanced version of the Linux makefile) and the Visual Studio 2010 project. The build actions/commands associated with these files produce C# objects. These C# objects are what the user codes against when writing their queries. These objects are attributed to guide the run-time translation how to do map between objects and renamed variables and the actual TTree leaf names.

All of the mapping from the C# objects to the actual TTree leaves takes place during query translation: no extra C++ code is emitted to deal with this translation. In short, this translation comes for free when it comes to running over the TTree.

Renaming a TTree leaf is a very simple operation. This is particularly useful when creating objects, like a Jet object. For example, in Figure 5 the TTree leaf called antiKt4TopoJets_pt to pt. This feature is particularly useful along with the object layout feature.

The TTree scanning code can take a very good guess at the object layout of a TTree. Each set of leaves that are an array that has the same number of entries in every event in the TTree is grouped together. Figure 5 shows an example of this – the leaves associated with a jet have been grouped together. The user can then modify the XML file and name the group appropriately (i.e. Jets). The user can also remove leaves from a grouping or add other ones into it. However, the translation code makes an implicit assumption that the array length of every single leaf in a group is the same. If the user includes a leaf that doesn’t follow that rule it can lead to a crash during run.

Finally it is possible to make an index a real pointer by forwarding. It is common to have a list of tracks associated with a Jet written out to the TTree. The way the data is stored, however, makes it painful to access in any framework. The most common way this is done is each jet has a linear array of integers associated with it: jet_index[njet][ntracksinjet]. Each integer is an index into a master array of tracks also stored in the TTree. To access the all track parameters associated with a jet one must first loop over the jets, get each track index, and then lookup the track parameter in the track parameter array: track_pt[jet_index[i_jet][i_trackinjet]]. Forwarding just tells the translation infrastructure that a particular integer array is really an index so that one can write: jet[i_jet].tracks[i_trackinjet].pt – and it will be correctly translated into the more complex double array indirect lookup. This new form also allows the user to take advantage of all the LINQ implied looping constructs, which is much more difficult in the double array indirect lookup form.

There are times when a leaf is actually an index into another array. Thus if an array that is associated with a Jet contains pointers to a list of tracks, it is possible to refer to the tracks directly.

The metadata scanner tool also generates a template TSelector source file. This file is used to plug in the query code at a later time. The TTree::MakeProxy method is used to generate this file. The proxies are used to optimize access to a TTree’s leaves, reading only the relevant leaves. No tests have been done to understand if TTree::MakeSelector, with appropriate leaves removed, would render a faster run.

5.3.2. Supported Leaf Types

For a leaf type to be supported by LINQtoROOT it must be able to translate it, sensibly, between the C++ world and the C# world. Most common types are supported. Any ROOT object that inherits from TObject is supported (i.e. TLorentzVector) – anything that is supported by ROOT.NET. All the
primitive types, like int, double, float, etc., are supported. Standard Template Library (STL) vector arrays of the primitive types (and ROOT objects) are supported. 2D arrays (vector<vector<int>> for example) are also supported. C-style arrays are also supported, including multi-dimensional arrays, and C-style arrays that are indexed by another leaf in the TTree (though object grouping is not done properly in this case).

String’s and char* variables are not supported. Nor are other STL containers, like map’s supported. Finally, there is some attempt to support custom objects that derive from TObject, however earlier versions of root have trouble parsing their streamer information, and the LINQtoROOT tools, which depend on this parsing, also have trouble.

5.3.3. Translating the query
The metadata scanner generates XML files that, in turn, generate a C# model that the user can query against. Actually, it generates two C# models. One is the high level one with objects like jets and tracks that the user queries against. The second is a model that represents the actual TTree leaf layout.

The first step in translating the query, then, is to convert from the high level C# model to the actual TTree layout. A series of translation algorithms in the back-end takes care of this fairly complex task. It is basically a pattern-matching problem over ASTs. The result is a query against the actual TTree object model.

The second step takes this AST and turns it into a series of statement objects and variable declarations that can be directly translated to C++ to run on the TTree as a TSelector. For example, there are representations for loops, filtering statements, etc. This intermediate representation is used to aid in optimization and in combining multiple queries.

5.4. The C++ Backend
The C++ backend combines queries and optimizes the resulting C++ as best it can. Each statement object knows how to combine itself with another statement object. In some cases the statements will be identical. The statement objects may be semantically identical, but not syntactically. For example, two loops over all jets in the event, these are identical even though the loop variable probably has a different name. The statement object combining code knows how to account for this. In the case that the statement objects are semantically identical, one set of statement objects will be eliminated. Current optimizations are very simple. For example, if the user tries to look at the highest p_T track associated with a jet, the code knows enough to cache the index into the track arrays for that particular track. However, there is a lot of room for improvements for the optimizer.

Once the final set of statement objects is combined and optimized, they are very directly translated into C++ fragments. This is a very simple process. That done, a template TSelector is used to generate the full C++ code. One very nice side effect of this process is that the code infrastructure knows exactly which leaves of the tree are accessed. It tracks this and it removes the unused leaves from the TSelector template. In a query that has over 1000 leaves, but with selection code that accesses only 30, this can make a dramatic difference in compilation speed.

5.5. Leaky Abstractions & Extension Points
There are several places where the LINQ model needs some extension and the translation engine needs some help. The LINQtoROOT project provides a number of places where the C# model can be extended. These extension points are also designed to help with places where the LINQ functional model can’t quite capture all of the capabilities of the underlying C++ system required to write an effective query.

It is possible to specify in a small configuration file function substitution. For example, the .NET Math.Sin function should map to the std::sin function located in the cmath header. Any number of these configuration files can be specified, and are picked up from a fairly simple search path. They declare the types and any extra headers that need to be included. The system comes with most .NET Math functions already declared. Figure 6 shows a portion of the standard file.
It is also possible to inject straight C++ code into the translation. This is particularly useful when attempting to smooth over the non-functional aspects of C++ and ROOT. For example, it is often the case that one needs to calculate $\Delta R$ between tracks and jets when associating tracks to jets. It is simplest to use a TLorentzVector. However, there is no functional way to create a TLorentzVector from $\eta, \phi, p_T$, and $E$. This requires creating the TLorentzVector and then calling the method TLorentzVector::SetPtEtaPhiE to initialize it. LINQ is functional – so to get around this one can define a snippet of C++ code that is injected straight into the translation. Figure 7 shows an example. The C++ code to be inserted is a series of strings. Variable names contains special wording such that the LINQToROOT backend can insert the snippet multiple times without there being a name collision (e.g. the Unique as part of the variable name tlzUnique in the figure). There is also a convention for creating objects such that there are no memory leaks: all objects must be created on the stack.

Convention requires a pointer to the object be created, however.
and some processing in C#). Finally, there is no real reason why complete functions written in C# couldn’t be translated to C++ and run, thought that requires significant effort. A helper library is slowly being built. One can imagine it includes a sophisticated cut-flow table tracker and generator – relatively simple with this functional form of implementing selection cuts. Sophisticated plots could also be built in – for example, a plot with multiple distributions shown, with a sub-plot shown below with the differences of each distribution from the expected value.

There is infrastructure work to be done as well. For example, the custom Nuget server takes much too long to wake up if no one has downloaded a package recently – sometime 3 minutes. It would be very nice to be able to shrink the packages that are needed enough so that they could be stored in the Nuget organization’s master server – which is a high speed production server. This is the biggest thing preventing the 2 minute idea-to-plot goal.

In general, most outstanding bugs and feature requests can be viewed on the CodePlex website for this project. The author uses its task system to track all bugs and features that go into the project.

7. Conclusions
The LINQtoROOT tool is available for use now by anyone who can use the Nuget package manager. The author has used it to generate plots and calculate the first version of the QCD backgrounds for a published ATLAS analysis [7]. The tool has accomplished most of its initial design goals – making analysis easy, composable, and iterative, and quick. There is lots of room to expand the tools use case and the next set of features will be driven by trying to perform a full, simple, analysis.

References
[1] ROOT - An Object Oriented Data Analysis Framework, Proceedings AIHENP'96 Workshop, Lausanne, Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.
[2] PROOF, a distributed version of ROOT. Documentation and downloads can be found on the ROOT website: http://root.cern.ch/drupal/category/package-context/proof
[3] NUGET is an open source package management tool. Libraries uploaded to repositories can be easily installed by users. http://nuget.org
[4] The C# Language Specification 4.0, Microsoft, April, 2010, http://www.microsoft.com/en-us/download/details.aspx?id=7029
[5] Watts, G, May 2012, ROOT.NET: Using ROOT from .NET languages like C# and F#, J/. Phys.: Conference Series (CHEP 2012), At Press
[6] re-linq - A General Purpose LINQ Foundation, Fabian Schmied, September 2009, https://www.re-motion.org/download/re-linq.pdf
[7] Search for a light Higgs boson decaying to long-lived weakly-interacting particles in proton-proton collisions at $\sqrt{s} = 7\,\text{TeV}$ with the ATLAS detector, The ATLAS Collaboration, Phys.Rev.Lett. 108 (2012) 251801, arXiv: 1203.1303