Using Functional Languages and Declarative Programming to analyze ROOT data: LINQtoROOT

Gordon Watts

Professor of Physics
Box 351560
Department of Physics, University of Washington
Seattle, WA, 98195-1650
USA

E-mail: gwatts@uw.edu

Abstract. Modern high energy physics analysis is complex. It typically requires multiple passes over different datasets, and is often held together with a series of scripts and programs. For example, one has to first reweight the jet energy spectrum in Monte Carlo to match data before plots of any other jet related variable can be made. 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 the analyser is really interested in. With most modern ROOT based tools this requires separate analysis loops for each pass, and script files to glue to the results of the two analysis loops together. A framework has been developed that uses the functional and declarative features of the C# language and its Language Integrated Query (LINQ) extensions to declare 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 analyser 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. A by-product of the design is the ability to cache results between runs, dramatically reducing the cost of adding one-more-plot and also to keep a complete record associated with each plot for data preservation reasons. 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

The LINQtoROOT project is a small piece of a new way of performing physics analysis, though it is technically the most difficult and central to realizing the vision. LINQtoROOT, an open source project, has been around for a few years undergoing sporadic development. This paper gives a brief overview of the LINQtoROOT project, recent developments motivated by its use in an ATLAS analysis, and finally a look towards the future of the project.

Physics analysis is still performed much the same way it was 15 years ago, though with larger datasets, better networking, and faster computers. The recent trend in software engineering has emphasised continuous integration, testability, keeping performance records over time, and repeatability. Physics analysis could benefit from this. If everything needed for a physics analysis were stored in a source code repository it could be automatically re-run each time a check-in occurred. The results could be automatically cached. There are a number of things that make this vision impractical – in particular the many hour run-times often required, the TB sized datasets that must be accessed, the

complexity of running all parts of the analysis. Running in this mode was not an initial goal of the LINQtoROOT project, but it has turned out to be a powerful side-effect of it.

2. The LINQtoROOT Project

The project grew out of the author's general frustration with the steps required in modern collider physics analysis. Multi-pass runs on data files and Monte Carlo (MC) to derive scale factors that are later applied are not uncommon. Any change in selections cuts in an analysis will often require carefully re-running all the steps that use those selection cuts; especially tricky in a multi-dataset analysis.

The modern toolset to accomplish this requires familiarity with several different languages: bash and python to control the analysis, C++ to loop over the ROOT data files and generate plots, and CINT macros to produce the final files. LINQtoROOT attempts all of this with a single language.

Along with the goal of making analysis simpler, there were several requirements – mostly for practical reasons. First, all data files had to be in the ROOT format – no translation. Second, the processing had to be fast – the code must be C++. Third, it must be possible to run locally or remotely (e.g. PROOF).

This tool, written in Microsoft's C# language, was first presented at the CHEP 2012 conference [1]. The tool has developed quite a bit since then, however the conference paper detailing the inner workings of the tool is still accurate. An introduction to the tool is perhaps best served by a quick look at a small two pass analysis file:

```
var output = new FutureTFile(@"..\..\..\..\MCJ1-MCQCDStudyResults.root"));
var eventsMC = MCQCD.QueryableCollectionTree.CreateQueriable(dsMC);
var ptMC = eventsMC.SelectMany(e => e.Jets).FuturePlot("RawPtMC", "Raw Pt MC; pT",
     100, 0.0, 200.0, j => j.pt()/1000.0).Save(output);
eventsMC.SelectMany(e => e.Jets).FuturePlot("RawPtMC", "Raw eta MC; eta",
     100, 0.0, 200.0, j => j.eta()).Save(output);

var eventsDATA = MCQCD.QueryableCollectionTree.CreateQueriable(dsDATA);
var ptDATA = eventsDATA.SelectMany(e => e.Jets).FuturePlot("RawPtData", "Raw Pt Data”,
     100, 0.0, 200.0, j => j.pt()/1000.0).Save(output);
eventsDATA.SelectMany(e => e.Jets).FuturePlot("RawPtData", "Raw eta Data; eta",
     100, 0.0, 200.0, j => j.eta()).Save(output);

Helper.Divide(ptMC.Value, ptDATA.Value).Save(output);
```

Line 1 creates a ROOT output file where results will be written. Lines 2 and 5 open the MC and DATA datasets. These datasets can be multiple ROOT files chained together, or a dataset name on a PROOF server. Lines 3, 4, and 6 and 7 request that plots be made, and when they are done, that they be saved to the file. Note the `SelectMany(e => e.Jets)`: for each event this unrolls the jets, so it is the jet that is handed to the FuturePlot method. In FuturePlot's arguments, note the `j => j.pt()`. This is a C# lambda function that converts the jet into the jet's p_T which is then plotted. The final line, 8, divides the histograms to create a ratio. The ratio could be used as input to another plot below if desired.

There is a subtly in this code that will be necessary to understand for discussion in later sections. The concept of futures, or promises, must be introduced to keep the analysis efficient [3]. Futures allow the system to accumulate many requests for plots without actually running a single one. When the actual value is requested, the LINQtoROOT library will process all known plots. This is key because almost all jobs are I/O bound, not CPU bound. When one considers a single pass through the file takes only 10 minutes, but one would need 500 passes to make 500 plots, it is easy to see why this some sort of facility like this must exist. Liens 3, 4, and 6, 7 do not make actual plots – just a promise to make a plot at some later time. Line 8's reference to the future's Value tells the backend it must now run over both datasets and produce an actual plot that can be used to calculate the ratio.

3. Using LINQtoROOT in a real analysis

This tool received extensive testing in an ATLAS physics analysis [4]. This tool was used by the author to accomplish a number of QCD background studies and cross checks in this analysis. None of the public plots were directly produced by this tool, but a good number appear it into the 200 page support note. Development since CHEP and the experiences described below are all from this experience.

4. General usage and performance

The performance of LINQtoROOT during the analysis is compared to the initial goals of the project. A good deal of further development was motivated by this analysis and that is described where appropriate.

4.1. Simplifying the boilerplate code in an analysis

There are several sources of boilerplate when building a physics analysis. At the lowest level there is the boilerplate that is required by the language: all the extra characters required to define functions, methods, classes, etc. Functional languages are perhaps the tersest, but this project required two language based features only available in C# when it was started (compile-time expression tree's and LINQ). The original paper contains extensive discussion over the language choice.

The second source of boilerplate is often required by the language and libraries themselves. For example, to declare, fill, and save a histogram using ROOT one ends up with source code spread throughout the analysis code: initialization to declare the histogram, code to fill it, and code to save it to the file. The design of LINQtoROOT means that all three occur in one line:

```
var allPlot = allJets
    .Select(j => j.Jet)
    .FuturePlot("ptAll", "AllMatchedJets", 100, 0.0, 200.0, j => j.pT/1000.0)
    .Save(output);
```

The *FuturePlot* method declares, books, and fills the histogram with the p_T of each jet. Note that there is support to make a template histogram so every jet p_T plot will have the same binning and limits. This was very successful and the code is much simpler than the standard C++ counterpart.

Efforts were also made to streamline the handling of input files. A separate text file lists the files and groups them under a single name – *test*, for example, or *mc* and *data*. That single name can be used to open the files in the code. By specifying a prefix like *proof* on the filename the LINQtoROOT backend would invoke the PROOF runner.

There are also numerous helper functions to help manipulate and post-process plots. This is necessary due to the Future mechanism. If one wants to manipulate a plot without forcing an actual run, some infrastructure is required. Languages that support monads natively will have the language syntax built in. As an example of what has to be done in C# to manipulate a plot without triggering evaluation:

```
FutureHelpers.Apply(gPTPlot, allPlot,
    (num, denom) =>
    {
        var h = num.Clone(num.Name + "_fraction") as ROOTNET.NTH1;
        h.Divide(denom);
        return h;
    })
    .Save(output);
```

Two future references to plots, gPTPlot and allPlot, will be passed to the lambda function that divides the plots, and saves the result to the output file. The lambda function will be executed when both gPTPlot and allPlot are already run, and not before. Though these functions and routines do remove a bunch of boilerplate, the difficulty, coined "monad hell", of manipulating these futures in a non-monadic way remains awkward and can clutter the code.

4.2. Translating C# to C++

This requirement was one of the main reasons C# was chosen: all manipulation of the TTree data from the file to the filling of the plot is given to the LINQtoROOT library in the form of an Abstract Syntax Tree (AST) [5]. The AST is data, and can be manipulated, or optimized. An AST is a typical intermediate representation in a compiler like gcc, for example. The AST can be translated into C++ code, or even shipped to a different machine with a different version of ROOT to run on a PROOF cluster.

A great deal of work has been done to optimize multi-plot runs. For example, after a complex selection of jets, one often wants to plot several quantities of those same jets (e.g. p_T and η). Hand coding one would make sure to select the jets once, and then plot all the variables. The LINQtoROOT library gets separate AST's for each plot and must intelligently combine them. Often it isn't obvious how to combine and optimize them. For example, expensive intermediate expressions (e.g. calculating η from $p_x, p_y,$ and p_z) can be lifted out of inner loops; complex sub-expressions are often calculated the same way in different parts of a plot's AST that can't obviously be combined – unless they are pulled out and the results are cached. This is by far the most complex part of the library. The author has gained new respect for compiler-optimizer writers. This is an ongoing area of work.

The AST used by LINQtoROOT is custom written by the author. However, the .NET framework contains a rather sophisticated and extensible AST. Utilities are starting to appear to help with optimizing these .NET AST's: a long term project is to replace the custom AST with the .NET one to take advantage of other people's optimization efforts.

4.3. Iterative analysis

Iterative analysis means being able to quickly add a plot to the analysis without being forced to wait hours for the new result. The final analysis, with 500 or so plots, takes close to 6 hours to run from scratch. A single additional plot often took less than 10 minutes to run on a local machine accessing the files across a fast network. Iterative means the ability to add a new plot to the existing 500 and have the results in about 10 minutes.

The AST, as described above, enables this feature. The AST plus any external input data (e.g. the input files), is turned into a cache key, with the file system as the cache. This has been remarkably successful. In no case has the cache-lookup time ever approached a typical run-time on the large files that are typical in an analysis. This enables numerous other scenarios. Changing a cut early in the analysis cut-flow will automatically re-run any dependent plots. If one then changes the cut back, the old results are still available in the cache. If a dataset doesn't need to be touched, then nothing will be run.

The caching hasn't been completely straight forward. For example, there is special code to prevent a re-run if one changes the title of a plot, though binning and limits remain the same. And while the look up is very fast compared to running over data, it isn't fast compared to some other operations, as described below. There is undoubtedly room for improvement: no effort has been made to optimize this yet.

4.4. A single programming language

This has been remarkably successful in eliminating the polyglot world physics analysis is performed in. One of the biggest aggravation points, having to run on multiple datasets with different script files, and then further scripts to glue the results together, is greatly mitigated, if not eliminated, by LINQtoROOT. Running over two large datasets is as simple as making plots from a single dataset. This plus the caching mechanism has eliminated a great deal of the "mess" involved in a traditional analysis.

The caching mechanism, however, is not as fast as it needs to be for the final stage of the analysis. The way the code is written and the design of C# leads to a very modular and composable layout of the code. Normally considered a good thing, but it makes running over matrices of cuts very easy, which can quickly lead to generating 500 or 1000 plots. The caching mechanism needs a finite amount of time to determine if each plot has been previously run. Somewhere between 500 and 1000 plots this is long enough that an analyser's attention will start to drift. This is problematic in the sense that near the end of an analysis one re-runs the analysis many times changing only plot formatting. With a normal

dedicated script accessing ROOT files containing histograms from a big run, one can do this in about a second or so.

The second problem turns out to be the code itself. As mentioned in the previous section, monad hell makes writing the final plot mechanisms awkward (especially in combination with the non-functional design of the ROOT histogram library).

As a result the PlotLingo language was designed, as briefly described in Section 6.1.

5. Enabling a continuously updating analysis

Software developers over the last decade have discovered continuous integration. Each time there is a check-in to the source code repository a build of the software is done and tests are run. Why not with an analysis? One of the main reasons not to, up to now, has been the complexity and length of time it takes to run an analysis – many hours. In the software industry tests are designed to run in seconds or minutes. So the analyser keeps very careful control over when and how an analysis is re-run.

The LINQtoROOT library can potentially change this equation by virtue of its caching mechanism. Re-running the analysis with one new plot can be quite fast. The author used the TeamCity build robot to monitor check-ins to the ATLAS svn analysis repository, and re-run the analysis each time one occurred. This gave a complete record of the source code changes, captured the resulting plots, etc. As a result if a plot changed it was easy to go back in time and figure out exactly where the plot had changed. It also forces the analysis to be self-contained: there can be no hand-art involved if the build robot is to run the analysis. Given how successful this experience was, the author questions his ability to return to the old style of running the analysis.

6. Can a plot be self-describing?

A plot that sits in an electronic logbook or a computer desktop for a month often has questionable heritage. Exactly what selection cuts were made? What files is that actually from? Make a plot self-describing was not an original goal of this project, but was stumbled on as work continued, especially as the PlotLingo language developed.

6.1. The PlotLingo language

The PlotLingo package was designed to be a high speed plotting language to make the last step – the tedious manipulation of plots – fast. Second, it was designed to be intuitive and straight-forward. This was a result of two failings of the LINQtoROOT library: the time to determine 500 plots have been cached was long enough to slow down the process and the monad hell the future mechanism induced.

The reason this is a custom language and not just ROOT C++ code directly is because the ROOT plotting API's are, for the most part, very low level. For a plot to be useful even in an internal meeting in a collaboration like ATLAS it needs a lot of things – a legend, properly named axis, etc.

The core of the language itself is very simple. It is functional in nature, and meant to be very terse and single-minded – perhaps one plot per file. The language contains statements and functions and methods. The *for* loop and a *map* operation are implemented as a function or method call, and arguments can be code (e.g. lambda functions). It also contains a number of extensibility points – similar to Aspect Oriented Programming [6]. For example, the first time a plot is created, code is attached to the plot to automatically generate the legend, or perhaps normalize the histograms. There is a utility that will automatically assign plot colour depending on key words in the tittle to help maintain uniform colour assignment across multiple plots. Operators like plus, multiply, etc., can be overridden. A simple example PlotLingo script is shown here:

```
f = teamcity ("http://tc-higgs.XXX/repository/download/XXX/2250:id/dataJZ2W-
FirstJetStudies.root");
h1 = f.Get("EMFL05/JetPtFirstJetAll");
h2 = f.Get("EMFL05/JetPtSecondJetAll");
[h1,h2].plot().title("Jet p_T");
```

A few things to note about the script. The first line loads a file from a build robot that is running the analysis, the second and third lines will fetch two histograms. The fourth line plots them together with a plot title of jet p_T. Adding the legend is as simple as adding a call to the Legend method.

6.2. Plot history

All of the pieces are now in place to maintain a complete plot history of how the plot was made. The LINQtoROOT library keeps as part of its cache the AST that generates the plot, the input files, and the input data. This can be attached as meta-data to the histogram. Plot lingo is an interpreted script – there is no reason not to store the script along with the meta-data with the resulting plot. Finally, a small program can be written that will extract the information from the plot and dump it to a nicely formatted file.

The cache key, which is the AST, is generated as a by-product of the LINQtoROOT plot generation. It is currently text and quite ugly. Further, it does not have enough information to be compiled. The text, even for a fairly complex plot, is relatively short: 2 or 3 KB. There is an issue of how to attach the code to the histogram as it is written out to the ROOT file. The simplest thing has been to write an extra TObjString object alongside the histogram, but the ideal case would be to extend the histogram object itself. This isn't possible without losing compatibility with other programs that may want to use the result.

PlotLingo produces PDF or jpg output files. Each format is extensible: arbitrary blocks of data can be added to the image. PlotLingo thus needs only to add the plot meta-data and the PlotLingo script into a block of data inserted in the final image. A proof of concept has been added to the PlotLingo language for jpg files.

If one can achieve code quality of the data stored in the plot image, the plot image becomes more than just an image. Many cloud storage systems maintain versions of old files – it would now be possible to look back at the plot from a few days ago and see what selections changed. One could upload the plot to a webserver, the webserver could extract the plot code and run it nightly on new data. One could paste it into a log book, and come back 6 months later and realize it wasn't quite run on the proper dataset.

The code for these experiments has not been placed in the public repositories yet – now that the proof of principle code has been developed real implementation can start.

7. Conclusions and future work

The LINQtoROOT project has been more successful than the author predicted, though it has been a great deal of work. It, or a derivative of it, will almost certainly be used by the author in the future if ROOT continues to maintain compatibility with Windows.

The ability to do a continuous analysis and the possibility of "easily" recording plot history were pleasant surprises to this method. While there are a number of limitations that will likely prevent this tool from being adopted by the community (e.g. it is currently Windows only), the fact that these sorts of features are possible should drive all of us to find solutions on the platforms that we use regularly.

References

[1] G. Watts, The LINQtoROOT Project, hosted on CodePlex (https://linqtoroot.codeplex.com/)

[2] G. Watts, Using Functional Languages and Declarative Programming to Analyze Large Datasets: LINQtoROOT, Computers in High Energy Physics 2012, Conference Proceeding

[3] Wikipedia, *Futures and Promises in Programming Languages*, http://en.wikipedia.org/wiki/Futures_and_promises.

[4] The ATLAS Collaboration, *Search for pair produced long-lived neutral particles decaying in the ATLAS hadronic calorimeter in pp collisions at sqrt(s) = 8 TeV*, ATLAS-CONF-2014-041.

[5] Wikipedia, *Abstract Syntax Tree*, http://en.wikipedia.org/wiki/Abstract_syntax_tree

[6] G. Watts, The PlotLingo Project, hosted on CodePlex (https://plotlingo.codeplex.com/)

[7] Wikipedia, *Aspect-oriented Programming*, http://en.wikipedia.org/wiki/Aspect-oriented_programming