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

Abstract ID#70

Gordon Watts
University of Washington
CPPM (Marseille)

ACAT 2014—Prague
1-5 September 2014

There has to be a better way.

Modern HEP analysis of ROOT TTrees has...

. too much hand-art,
. too much boilerplate,
. not reproducible enough,
. isn't developed iteratively,
. isn't faster enough!

A **modern HEP analysis** requires many scripts, running on the GRID, use of experiment-validated ROOT TTree datasets that are very large… Most of us code this up in multiple steps, adding to a Franken monster as we go. Rerunning is **error prone** because many of the steps must be done by hand, in a specific order, often with hand editing between. And adding a new plot can take hours because it requires remaking all the old plots on a large dataset. Surely we can do better than this in the modern era of distributed computing, sophisticated programming languages, build bots, and all the other modern software that has sprouted since we invented the analysis chain!

Personally, I need this. I'm a professor and my duty cycle is low. It is even more likely that I will forget a setup than one of my students or postdocs. What I present here **isn't a complete solution**, but represents experiments I've done in an attempt to solve this problem for myself.

This Solution:

1.Declarative Programming

2.Plot provenance for tracking and caching

3.TeamCity BuildServer

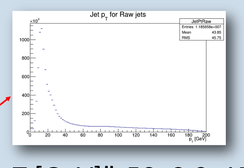
This poster is mostly about the declarative programming aspec, which enables plot provenance and caching, and which is by far the most complex component of this set of analysis tools.

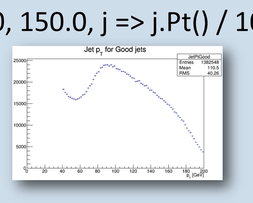
LINQToROOT project
https://linqtoroot.codeplex.com/

PlotLingo project
https://plotlingo.codeplex.com/

Example: make two plots...

```
FileInfo rootFile = new FileInfo(@"..\..\..\output.root");
var rf1 = Queryablebtag.Create(rootFile);

int count = rf1.Count();
Console.WriteLine("The number of events in the ntuple is {0}.", count);

var alljets = from e in rf1
              from j in e.jets
              select j;

var hpt = alljets.Plot("pTAllJets", "pT for all jets; pT [GeV]", 50, 0.0, 150.0, j => j.Pt() / 1000.0);

var goodjets = from j in alljets
               where j.Pt()/1000.0 > 30.0 && TMath::Abs(j.Eta()) < 2.0
               select j;

var hptGood = alljets.Plot("pTGoodJets", "pT for Good jets; pT [GeV]", 50, 0.0, 150.0, j => j.Pt() / 1000.0);

var f = ROOTNET.NTFile.Open("junk.root", "RECREATE");
htp.SetDirectory(f);
hptGood.SetDirectory(f);
f.Write();
f.Close();
```





Open the file, connect to the b-tag TTree

Run through the TTree and count the number of entries in the TTree.

Each entry in the TTree has a collection of jets. Alljets is just a list of every single jet in the output.root file.

Make a plot of the pT of all the jets in units of GeV. htp now points to a TH1F.

Make a new list of all jets in the file, but only those that have pT > 30 GeV and |eta| < 2.0

Write the plots to the junk.root file.

Real World Use:

Most of the lessons and measurements of success and failure come from one analysis performed by the author using this framework. The recently released search for Hidden Valley particles in the calorimeter had a part of the background study done in this framework. The general experience: coding the physics and the selection was much faster than anything done previously. For the analysis see https://atlas.web.cern.ch/Atlas/GROUPS/PHYSICS/CONFNOTES/ATLAS-CONF-2014-041/

Some interesting facts about the code for this analysis:

- Two projects were written. One to calculate the background and one to explore correlations between two jets of various analysis variables. It did not feel natural to put everything in a single analysis program.
- Data and MC ntuples were different, which made using them with common code difficult (C# is strongly typed, so needs to know the layout of the TTree at compile time). A common event data model was built to get around this complication. The expression tree translation *should* have removed all evidence of this in the generated C++ code, though this was not carefully examined.
- The library that handles the main analysis cuts, the EDM translation, MC/reconstruction object matching, etc, is 350 lines of code. In the analysis the function with the most lines of code is just making many plots with few logic decisions

Besides problems mentioned elsewhere in this poster, there were some issues encountered:

- The experiment is moving towards standard tools which depend on data files and libraries, making it much harder to make this a cross platform analysis.
- C# is strongly typed, so a TTree's layout must be well known in advance. The tools to do this scanning are not friction free.

How does it work?

```
var alljets = from e in rf1
              from j in e.jets
              select j;

var goodjets = from j in alljets
               where j.Pt()/1000.0 > 30.0 && Math::Abs(j.Eta()) < 2.0
               select j;
```

The compiler translates the syntactic sugar of LINQ into this equivalent functional form.

```
var goodjets = rf1
   .SelectMany(e => e.jets)
   .Where(j => j.Pt().1000.0 > 30.0 && TMath::Abs(j.Eta()) < 2.0))
```

C# Lambda Expression

1.The analysis code has implied loops over the events, over jets, etc., as shown above. The variable "goodjets" will represent a sequence of all jets in the input file rf1 that have a pT > 10 GeV and an eta less than 2.0.

2.The compiler translates the syntactic sugar into a functional call shown at right. Often the analyzer will prefer to write directly in the functional form.

3.Each argument to a function, like Where, is actually an expression tree, not a compiled Lambda expression (as it appears to be). This means the underlying library can manipulate those expression trees.

4.Backend code can translate the expression sequence into C++ code, and wrap it with the standard ROOT C++ TSelector boilerplate.

The function **Where** has a special (**magic**) signature:

```
IQueriable<Jet> Where<Jet> (
    IQueriable<Jet> sourceSequence,
    Expression<Func<Jet,bool>> test);
```

The Expression<> type tells the compiler not to pass the value of the expression, but an expression tree that represents the expression. Thus the **Where** function gets passed an expression tree—a data-structure that can be examined by the low level library… and be converted into C++ code (or anything else)!!

Extras:

Because the C++ translator gets a completely specified expression tree it can be manipulated directly by that code. This allows for a surprising amount of flexibility.

- Most ROOT TTree's in ATLAS are flat, not objects, but arrays of floating point numbers. For example, the information for a jet might be spread across several arrays (eta, phi, pT, etc.). Though possible to use with this tool, it isn't very convenient. A dummy jet object can be created, with members eta, phi, and pT. During the expression tree translation phase this dummy jet object can be translated into direct array accesses.
- Sometimes the only way is real C++ code. This happens most often because the ROOT API isn't functional. The most common example seen in code is the TLorentzVector. It is possible to embed C++ code in the C# file and have the translator slip it directly into the C++ file.
- Functions like Math::Abs can be translated directly to C++'s fabs using a simple configuration lookup (which the user can add to if they wish).
- The backend is pluggable. There are currently two working: PROOF server (on Linux) and run locally on your Windows computer. The author wouldn't claim writing a new backend was trivial, but it is far simpler than the C++ emitter or the expression tree transformation code.

Limitations:

- The main limitation is the inability to include functions in the expressions. For example, it would be nice to create a goodJet function, and then use it all over the code. Unfortunately, the most obvious way of doing this means that the goodJet function is not an expression tree, but compiled code. This can't be translated by the C++ emitter code. By specially declaring goodJet it is possible to get around this, and the expression tree translator will work around this, but this is an advanced use of the framework and isn't totally simple, unfortunately. Here is an example that creates an expression to cut on the electromagnetic fraction (EMF) and the nmber of tracks (ntrack). The line after the function shows how it is used in the code:

```
private static Expression<Func<double, int, bool>> GenerateCellWindowExpression(CellRangeResults cell)
{
    Expression<Func<double, int, bool>> cut = (emf, ntrack) => emf >= cell._emf._low && emf < cell._emf._high
        && ntrack >= cell._t._low && ntrack < cell._t._high;
    return cut;
}

Usage:

    var emfRangeJets = matchedTruthJets
        .Where(j => cut.Invoke(j.Jet.EMF, j.Jet.nTracks));
```

Initial Goals:

- Remove as much boiler plate as possible. — Much Improved
- Put everything into a single programming language. — 50/50 Success
- Encode the multiple steps in an analysis in a single program. — Success
- Enable iterative development of the analysis — Much Improved

Originally motivated by the demands on my time as a professor: how can I quickly generate analysis-grade plots but run over LHC size datasets?

Requirements:

- Must use ROOT files as input (I'm a member of ATLAS!)
- Use C++/native code for the processing loop (speed!)
- Use PROOF (Linux) or run locally with "ease"
- Results should be normal ROOT objects and plots

Stretch Goal:

- Can a plot be self describing? — In Progress

Put everything into a single programming language:

The original goal of this project was to do everything from running on multiple initial datasets to final plot manipulations (including adding plot titles, text, etc.) all in one single program.

- It is possible to run on multiple datasets. It is even, with something like a PROOF server, possible to run on multiple datasets simultaneously with some minor modifications to the framework. This was a huge success. The author found that it was a feature that was used regularly, but only if the multi-dataset running was fairly uninvolved.
- The more complex interaction of two datasets leads to tricky and non-obvious code dependencies due to the nature of the *Future* construct. Indeed, any manipulation of the plots or results are a bit messy. There are programming languages that handle this (e.g. functional ones that have extensible monad's built in). This could even be handled in C# with some sacrifice, but it isn't clear that it should be. This issue is subtle and as yet unresolved.
- The final plot manipulations are painful. Frequently one must do 100's of iterations, moving text to the left or right by 0.1" or something silly like that. This only works if the generation of those plots is *fast*. Perhaps less than a second or two. However, a typical analysis has 1000's of generated plots and numbers, and it takes real CPU time to determine that no plots needed to be re-run and it was only the end manipulations that had to be repeated. This framework did not succeed in this aspect.

Encode the multiple steps in an analysis in a single program:

A typical sequence is to make the same histogram from a MC file and a data file, divide the results, and then use that ratio as a reweighting to run on the MC file. In a traditional analysis this is a three step process: scripts to submit a job on the data and MC files, then a script to divide the histograms, and then a script to re-run on the data file. This code can easily be expressed in one file, and in a concise way. For example:

```
var ptRangePlot = rangeJets
                  .Select(j => j.Jet)
                  .FuturePlot(EDMPlot.SpPtPlot, "Restricted" + namemodifier)
                  .Save(outputFolder);
var ptw = ptRangePlot.Value;
var weightedJetCount = (from j in matchedTruthJets
                        let iBin = ptw.FindBin(j.Jet.c.Pt())
                        let wt = ptw.GetBinContent(iBin)
                        select wt).FutureAggregate(0.0, (acc, val) => acc + val);
```

Goal: Iterative Development of an Analysis

The main requirement for iterative analysis is being able to make an adjustment to a plot, hit the *run* key and then be looking at the new plot quickly. In general 30 seconds seems to be about when the author starts to be distracted by Facebook. This is most important for running over smaller ROOT files (gigabytes). At the same time this can't get in the way of reasonably efficient running on the large datasets (which is done much less frequently). A few features have gone into making sure this works well.

Success:

- If the plot has been made on a previous run and nothing has changed, then a cached version of the plot is used. This is perhaps the largest enabler of a quick turn around, and is almost built into the way the problem is solved in this framework. Each plot comes attached with an expression tree that details everything from start to finish that is done to the input files to generate the plot. As long as no parameters are altered, and the input files haven't been altered, then the resulting plot must be the same. Input parameters can be a bit tricky to check. For example, consider a TH1F that is used to reweight a variable before it is plotted. If that TH1F is changed, then the plot must be remade. So the caching must be done carefully. But it isn't hard. Things are cached on the local system in a local directory. The result is if you have 500 plots in your program, and you modify a single one, only that plot is actually run. This can make the difference between hours and minutes for a run.
- Code generated for queries are merged where possible. The key here is if you have 500 plots you want ROOT to loop over the data only once, not 500 times. This gets around one of the major limitations of TTree::Draw (one of the inspirations for this work). This is enabled by ***futures***. Instead of asking for a Plot, the user asks for a FuturePlot. This is a promise of a plot at some point in the future. After all the promises have been accumulated, they can all be combined and run at once.

Failures:

- While the speed up by caching the plots is huge, when one gets up to 1000 plots or so just the time it takes to generate the cache lookups starts to become prohibitive—longer than 30 seconds in total. Some of the queries can be quite complex—leading to very long expression trees that must be compared. Some work has gone into optimizing this step. This is another area of active development—it is not yet well understood what is so expensive about this operation.
- If plots need to be manipulated—say divided to get a ratio or similar, the place this makes the most sense is right after the plots are generated. However, with Futures in the mix this isn't trivial any longer. When one wants to do the division there aren't actual plots to divide yet! There are ways around this, but they significantly obscure the code, significantly damaging one of the goals of this approach.

Data and Analysis Tracking & Preservation:

The author has used electronic log books for over a decade now (OneNote, primarily). As this project evolved, the author realized the following scenario was possible: take a jpg/png from the logbook, drag it onto a special program, and have the program dump out the names of the input ROOT files and every single thing that happened after that to make the plot.

- As each plot is generated, the complete expression tree is known and in a single place in the program. In fact, it is serialized as a string for a cache key. While there is some loss in fidelity—you could not recreate code from this string—it is more than enough to see what cuts were made. A serialization of the actual expression tree is not too difficult (not attempted yet). This is called the *query string*.
- The input files and datasets are also well known (they are a TChain).

There are a few issues however:

1. The data must be carried along with the plot. Attached to the plot somehow.
   - ROOT's TH1 doesn't really have any method to do that (like a "userdata" store). Several ideas were explored. One was to subclass TH1F, and the new subclass could store its history. However, opening a ROOT data file containing these special histograms no longer worked correctly unless the code had been pre-loaded. This approach was rejected.
   - Another was to store the query string in a TObjString along side the plot. The only trick was that the code, as seen above, wants to use a simple histogram object—a pair of objects would significantly destroy the usability. So this approach was rejected.
   - The current *experimental* approach is a combination of the two. Use the extended histogram to generate all the plots, and when it is written out write out a separate TH1F and a TObjString.
2. Manipulations of the plots in the C# code must be recorded. For example, if you generate an efficiency plot you must divide two plots. So a when a call is made to the histogram TH1F::Divide method, it must be recorded. This requires modifying a every single operation and manipulation to update the query string and could potentially be quite ugly. The author was saved some trouble because of the *Future* problem mentioned elsewhere. Lots of small utility methods already encapsulated histogram manipulation—only those had to be modified to get around this issue. *PlotLingo* was also created, partly, as an effort to solve this.
3. Finally this must be recorded in the PNG or JPEG or PDF plot. This turns out to be fairly easy. All these image formats (including PDF) allow the user to add arbitrary payload to them. Writing code to take a plot and the associated TObjString is not too difficult. The same for the code that dumps the result back when an image is dragged onto the program. This additional data increases the size of the image by about a 50% (from about 40 KB to 60 KB). One could potentially use DOI's to shrink it back down, but that comes with its own set of challenges.

PlotLingo:

A new plotting language, in the extreme alpha stage, invented initially to quickly solve the last mile problem with making plots quickly. However, it also has potential to help solve some of the issues around the tracking and preservation. It is an interpreted language, functional, with lots of hooks for extensibility. The core is very simple. It is not built for speed (e.g. running 1000's of plots), but it is built for being able to quickly create a nicely formatted plot or four.

```
f = teamcity("http://tc-higgs.phys.washington.edu:8080/repository/download/Atlas_HvMcCompareCi/2362:id/CalRatioScan.root");

mc = {
      JZ1 => f.Get("dataJZ1W/EF_J15/ProbeJet/JetPtProbe"),
      JZ2 => f.Get("dataJZ2W/EF_J15/ProbeJet/JetPtProbe"),
      JZ3 => f.Get("dataJZ3W/EF_J15/ProbeJet/JetPtProbe"),
      JZ4 => f.Get("dataJZ4W/EF_J15/ProbeJet/JetPtProbe")
      };

mc
  .sum()
 .plot();
```

Analysis by Build Server:

A build bot is a very simple program. It sits up in the cloud somewhere watching a svn or a git or similar repository. Every time it sees a check in, it extracts the changes and runs a set of commands. It keeps a log file, and archives any output files you desire from the process (e.g. a ROOT file).

Analysis by build bot is simply that. The analysis runs each time you check something into source control. It is always run in a carefully controlled environment. There can be no hand-art in the making of the results. Because re-running the framework is cheap when adding one or two plots, running this in a build bot works quite well. This method was used with the TeamCity build bot for an analysis. Being able to go back to previous versions at any time and collect the plots was golden.

Conclusions:

- Declarative programing is superior to the standard imperative for the analysis tasks done by the author.
- There are lots of small friction points in the current tool set. Some due to Windows (HEP is Linux), some due to the nature of leaky abstractions, some due to ROOT, and some due to declarative programming.
- The ability to track and carry meta-data with the final plots is very interesting, but not yet fully explored.
- **Where next**? ATLAS is going through a major rework of its EDM, will have to see how hard that is to incorporate.
- Great deal isn't covered here (e.g. code optimization!)