Making the TTreeReader interface more accessible

Ákos Hajdu
Summer Student
CERN PH-SFT

Bertrand Bellenot
Supervisor
CERN PH-SFT

Axel Naumann
Supervisor
CERN PH-SFT

August 14, 2015

Abstract

The ROOT framework is the main data analysis tool for High Energy Physics. One of its main features is *TTree*, a collection type enabling efficient analysis on petabytes of data. ROOT has a new interface called *TTreeReader* to access data in a type-safe and efficient way. This Summer Student project aims at implementing a utility interface to generate source file skeletons as a starting point for the users. This will make the *TTreeReader* much more accessible for the users of ROOT.

1 Introduction

This section introduces the background of the project, namely the ROOT framework (Section 1.1), its main data structure, the *TTree* (Section 1.2) and the way of accessing data inside such trees (Section 1.3).

1.1 The ROOT framework

The ROOT [1] system provides a set of object-oriented frameworks for handling and analyzing large amounts of data in an efficient way. The data is defined as a set of objects and a special data structure, the *TTree* allows direct access to separate attributes of selected data, without having to touch the bulk of the data. The ROOT framework provides histogramming, curve fitting, function evaluation, minimization, graphics and visualization, which allows an easy setup of a system that can query and process the data interactively or in batch mode. Parallelization is also supported by a general framework called PROOF. The command language, the macros and the programming language of ROOT are all C++ thanks to the built in interpreter. ROOT is mainly used in High Energy Physics (e.g., by experiments at CERN) and it is also an open-source project available on its website [1] and mirrored on GitHub [2].

1.2 What is a TTree?

TTree[1] is the main data structure of ROOT, which is capable of handling the enormous amount of data produced by experiments at CERN. Instead of storing the data in a flat, table-like structure, data is organized in a tree-like structure. The data is stored column wise, which allows each branch of the tree to be accessed separately. This yields better compression of the data and also a better performance if only a subset of the attributes of an object is accessed. An example can be seen in Figure 1. The tree contains *EventData* objects that are divided into the list of *Particle*s and the size of the event. *Particle*s are further divided into their properties, e.g., coordinates and momentum. All these members can be accessed without touching other parts of the data.

1.3 Accessing a TTree

Before ROOT version 6, the data in the *TTree* could be accessed by defining data variables and branch addresses and linking them together (Figure 2a). However, in ROOT 6 a new interface, called *TTreeReader* [3] was introduced (Figure 2b), which gives an easy and type-safe access to the data in the tree. Further

[1] In the ROOT framework each class begins with the letter "T", e.g., *TTree*, *TBranch*, *TSelector*.

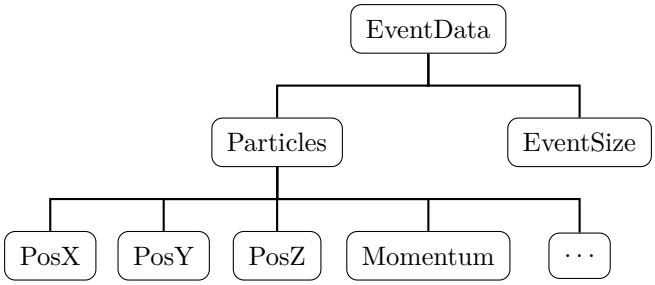

Figure 1: Example TTree.

advantages of *TTreeReader* are loading the data on-demand and using the cache and memory management of the tree.

```
double fParticles_fPosX[100];
int fEventSize;

TBranch *b_fParticles_fPosX;
TBranch *b_fEventSize;

fChain->SetBranchAddress("fParticles.fPosX",
   fParticles_fPosX, &b_fParticles_fPosX);
fChain->SetBranchAddress("fEventSize",
   &fEventSize, &b_fEventSize);
```

(a) Before ROOT 6

```
TTreeReaderArray<double>
   fParticles_fPosX(fReader, " fParticles .fPosX");
TTreeReaderValue<int>
   fEventSize(fReader, "fEventSize");
```

(b) Since ROOT 6

Figure 2: Accessing data in a *TTree*.

Usually not a single element of a branch is picked, but rather all elements are looped over. This is often done in the following three steps. At first some initialization and setup takes place, then each element is processed (e.g., calculations are done with the data) and in the end some finalization is done (e.g., presenting histograms). This pattern is similar for each tree, the only difference is the name and type of the branches. Therefore, so-called *skeleton analysis files* (also called selectors) can be generated automatically from a *TTree* by the utility function *MakeSelector*. These skeletons contain all the necessary data member and branch declarations to access data and they provide empty functions like *Begin*, *Process* and *Terminate*, which must be filled by the user. *Begin* and *Terminate* is called at the beginning and the end of processing respectively, while *Process* is called for each element. This way the repetitive part of the code can be automatically generated and the user can focus on the specific analysis part.

However, there are some issues with the existing code generator (*MakeSelector*). On one hand it cannot deal with some complex trees and also the code it generates uses the old (before ROOT 6) way of accessing the tree. Thus, the main task of the project was to implement a new code generator that uses the new (*TTreeReader*) interface for accessing the tree. This way, *TTreeReader* will become more accessible and widespread among the users.

2 The main task

This section presents the main task of the project, namely writing the code generator (Section 2.1) and testing its functionality (Section 2.2). Furthermore, some of the challenges throughout the project are also presented (Section 2.3).

2.1 Writing the code generator

Code generation consists of the following three main steps.

1. Parse the options provided by the user.
2. Analyze the structure of the tree.
3. Write code in the output files.

Parsing options. By default, the generated selector will include a reader for each leaf of the tree (e.g., *EventSize*, *PosX*, *PosY*, *PosZ*, *Momentum* for the tree in Figure 1). However, this behavior can easily be overridden by the user. With an option string, the user can

- specify that only the top-level branches should be included,
- pick individual leaves,
- pick individual inner branches,
- pick leaves of individual inner branches.

These options are given as an option string, which needs parsing and validity checking.

Analyzing the tree. The next step is to traverse and analyze the structure of the tree. In order to generate proper code, several properties of each branch are collected, including the name of the branch, the type of the data stored, information about collections and the relation (parent-child) between branches. The collected data is filtered based on the options provided in the previous step, i.e., branches that do not match the criteria are discarded.

Writing output code. Writing the output code is quite straightforward using information from the analysis. The following two files are generated:

- A header file containing the declaration of the selector with the branches that are collected in the previous step. Also some already implemented functions are provided.
- A source file containing empty skeletons of functions (e.g., *Begin*, *Process*, *Terminate*) that the user has to fill.

2.2 Testing

An other important subtask of the project was to test whether the new code generator works well. This consisted of the following steps.

1. Create various types of test trees and fill them with data.
2. Open the trees and generate analysis skeletons.
3. Fill the skeletons with some simple code and run them to check if data is accessible.

Creating trees. *TTree*s can store basically any type of data in various ways, so several test trees were created from quite simple ones to rather complex instances. The list below gives some examples.

- Trees with basic types, leaflists, simple classes.
- Trees with nested classes.
- Trees with various collections, storing basic types and classes.
- Trees with nested collections.

Generating analysis skeletons. After opening the trees and calling the new code generator, it can be manually checked whether the generated output files are correct. In this case, correctness means that the file contains all data members (specified by the options) and all (empty) functions. Also the names and the types of the data members must match the branches of the tree.

Accessing the data. Having correct output files indicates that the new generator works well for the given examples. However, it was also important to check whether the generated code can actually access data in the tree. Therefore, the generated skeletons were filled with some simple code that reads and prints data in order to make sure that it is accessible.

A part of the generated code for the tree in Figure 1 can be seen in Figure 3. Lines 1–14 are from the header file, while lines 16–28 are from the source file. The data members[2] generated for the leaves are between lines 5–9 and the empty functions are declared in lines 11–13. Lines 20–25 are not generated, but show an example on how *Process* can be filled by users to access *EventSize* and *PosX* (other members can be accessed similarly).

```
 1 class EventTree : public TSelector {
 2 public :
 3    TTreeReader fReader;
 4
 5    TTreeReaderArray<double> fParticles_fMomentum = {fReader, "fParticles.fMomentum"};
 6    TTreeReaderArray<double> fParticles_fPosX = {fReader, "fParticles.fPosX"};
 7    TTreeReaderArray<double> fParticles_fPosY = {fReader, "fParticles.fPosY"};
 8    TTreeReaderArray<double> fParticles_fPosZ = {fReader, "fParticles.fPosZ"};
 9    TTreeReaderValue<int> fEventSize = {fReader, "fEventSize"};
10
11    virtual void Begin(TTree *tree);
12    virtual bool Process(Long64_t entry);
13    virtual void Terminate();
14 };
15
16 bool EventTree::Process(Long64_t entry)
17 {
18    fReader.SetEntry(entry);
19
20    printf("EventSize: %d\n", *fEventSize);
21
22    printf("Px:");
23    for (int i = 0; i < fParticles_fPosX.GetSize(); ++i)
24       printf(" %.1lf", fParticles_fPosX[i]);
25    printf("\n");
26
27    return true;
28 }
```

Figure 3: Example generated code.

2.3 Challenges

This section presents some of the most important challenges during the project. These challenges are related to writing the code generator, accessing the tree and also to testing the work.

Writing the code generator. The old *MakeSelector* utility function seemed to be a good starting point for writing the new code generator by adapting it to generate code with *TTreeReader*s. However, it is a quite old code and it can not handle several types of trees, e.g., some STL[3] and ROOT collections. Also it is not flexible, since it only considers the leaves of the tree. On the other hand, there is a more recent and similar utility function (called *MakeProxy*), which also analyzes the tree but it generates a different type of skeleton (proxy). *MakeProxy* can handle more types of trees than *MakeSelector*, so the

[2]ROOT supports C++11, thus members can be initialized in-class.
[3]STL: Standard Template Library of C++.

analysis code from *MakeProxy* was used as a starting point and it was adapted to generate analysis skeletons with *TTreeReader*s. However, *MakeProxy* is a quite complex code (about 4000 lines) so it took a while to understand how it works and adapt it[4].

Accessing the tree. When testing whether the generated code can access the tree, some issues were found with the existing interfaces of ROOT (*TTreeReaderArray*). Some examples are listed below.

- Accessing *vector*s of basic types failed.
- Accessing nested collections failed.
- Accessing non-split ROOT collections (*TClonesArray*) failed.

Together with bug reports from users, a total of 9 issues were found during the project. Since these issues are related to the project (the new code generator relies on these interfaces), some time was spent on identifying and fixing them. At the end of the project, 5 out of the 9 issues were fixed. The rest of them are tracked for future correction.

Automating tests. It was easy to test manually whether the generated code is correct and can access the tree (by checking the code and filling some of the functions). However, this involved a lot of time and manual, repetitive work. Therefore, automatic tests were created. Automation however, was not an easy task, since each time the code is generated, empty functions are created that do not access the data. Therefore, pre-filled functions were written, which can replace the generated empty ones.

3 Conclusion

The ten week project at CERN PH-SFT provided me a lot of experiences. I had the chance to work on a large software, improving my skills in C++ programming and also in teamwork. I gained experience in the advanced usage of software version control systems (git), reporting and keeping track of issues and also experience in discussing my work at team meetings and presentations.

Together with my supervisors we concluded that the project was successful. I examined *TTree*s and the new interfaces (*TTreeReader*) for accessing data of the tree. I implemented a new utility function for generating analysis skeletons using the new interfaces. I tested the new code generator and found several issues in the existing interfaces from which I also fixed some. The new code generator is now merged into the latest (master) version of ROOT, replacing the old utility function *MakeSelector*. Although I created several tests which are working for me, we are sure that when the real users will start using it, we will get useful feedback for the future development.

References

[1] Website of the ROOT framework. `http://root.cern.ch/` [Online, accessed: 14/08/2015].

[2] ROOT on GitHub. `http://github.com/root-mirror/root` [Online, accessed: 14/08/2015].

[3] ROOT 6 release notes. `http://root.cern.ch/root/html600/notes/release-notes.html#ttreereader` [Online, accessed: 14/08/2015].

[4] We would like to express our acknowledgment to Philippe Canal (CERN PH-SFT) for helping us with *MakeProxy*.