Native Language Integrated Queries with CppLINQ in C++

V Vassilev

CERN, PH-SFT, Geneva, Switzerland

E-mail: `vvasilev@cern.ch`

Abstract. Programming language evolution brought to us the domain-specific languages (DSL). They proved to be very useful for expressing specific concepts, turning into a vital ingredient even for general-purpose frameworks. Supporting declarative DSLs (such as SQL) in imperative languages (such as C++) can happen in the manner of language integrated query (LINQ).

We investigate approaches to integrate LINQ programming language, native to C++. We review its usability in the context of high energy physics. We present examples using CppLINQ for a few types data analysis workflows done by the end-users doing data analysis. We discuss evidences how this DSL technology can simplify massively parallel grid system such as PROOF.

1. Introduction

Informally, imperative programming languages (such as C++) describe the necessary steps to get desired results, i.e the 'how' to perform the computations. On the other hand declarative programming languages (such as SQL) describe the desired result, i.e the 'what' the result should look like. Over the years, interoperability between both worlds has become vitally important to almost every general-purpose programming language. The interoperability in imperative languages is usually implemented through disabled type safety at compile time. Inter-process communication was done via compiler and library support mainly employing the type variants [1] [2] or type erasure. Another well-known approach to integrate domain-specific declarative programming language support is using strings, passed to bridging layers. For example, SQL is embeddable in a C++ application through string literals. They are passed to a driver, querying the database behind, producing results and finally wrapping them into C++ objects.

Comprehension syntax is very close to the syntax of a number of practical database query languages. A language of comprehensions can naturally and uniformly express operations on various collection types [3]. C# [4] introduced an incarnation of the comprehensions with its Language Integrated Queries (LINQ). LINQ's key concept is to provide a type-safe comprehension (SQL-like) syntax, operating on collections. The C# language provided rich language constructs, allowing LINQ implementation on top of the existing features. Moreover, LINQ introduced the new programming paradigm integrated into the type system of the underlying imperative language.

The newer C++ language specification standards introduced the necessary concepts to build type-safe, platform-independent comprehension syntax in C++. Since then a few open-source implementations of LINQ for C++ emerged. All of them are in early development stages, but definitely worth reviewing their power and usability in the C++ context. In particular we shall:

- describe the advantages of the computational abstraction;
- show the expressiveness of the comprehension syntax via concrete code snippets.

This paper is divided into sections as follows: Section 2, CppLINQ Overview, discusses in brief some of the advantages of the computation abstraction and how it can come naturally by introducing the comprehension syntax in C++. Section 3, Usability & Applications, shows the capabilities of CppLINQ and its possible usage scenarios in the field of the high-energy physics. Section 4, contains Conclusion & Future work.

2. CppLINQ Overview

The declarative programming paradigm helps to abstract out details such as computation control flow and tends to reduce side effects [5]. The paradigm puts less emphasis on the 'control' part of Kowalski's equation *algorithm = logic + control* [6]. Thus it leaves a lot of room for interpretation on how the logic semantics could be realized. The same logic can be expressed in many different ways, depending on the standpoints or the implicit requirements. Thus the system can delegate pinning down the control to a specialized layer, which performs further planning, optimizations and scheduling.

Going declarative can be too harsh for various reasons: some concepts are difficult to express; different algorithms exist only in an imperative language or are more efficient in an imperative language; the difficulty of educating developers; etc. These drawbacks lead to a hybrid approach – mixing a declarative language in an imperative one. The hybrid approach gives the flexibility to the developers of choosing the computational abstraction only where it makes most sense.

C# LINQ provides a declarative way to easily extract and process data from arrays, iterable classes, XML documents and relational databases. After its success in the .NET world it was ported to many unmanaged environments such as JavaScript and C++. CppLINQ is a common name of LINQ technology, implemented in C++. The implementation techniques vary, as a result there are a few CppLINQ dialects. Depending on the dialect, the same program logic could be expressed in different ways. Listing 1 introduces a typical CppLINQ program. The snippet illustrates invariant statements for every CppLINQ program. Writing code in CppLINQ is straightforward: include the CppLINQ-specific headers (line 1); add a using CppLINQ clause (line 7); use the 'from' statement (line 8) to enter the CppLINQ world.

```
 1 #include "linq.h" // Include the necessary header files.
 2 // Returns true if the parameter is a prime number.
 3 bool is_prime(int x);
 4
 5 long sum_primes() {
 6   auto xs = int_range(0, 100); // same as boost::counting_iterator.
 7   using namespace cpplinq; // Add the using CppLINQ clauses.
 8   return from(xs) // Switches to the CppLinq world.
 9          .where(is_prime)
10          .sum();
11 }
```

Listing 1: A typical LINQ-style program in C++

2.1. Implementation Approaches

Listing 1 demonstrate how to sum all prime numbers in the range of (0, 100) using CppLINQ. Some implementations rely on the 'operator.' and others on other operators, which are allowed to be overloaded. Usual criticism of the 'operator.' approach is that 'operator.' is not overloadable in C++. This makes user extensions very hard to implement. Alternatively, LINQ provides other hooks to handle third-party extensions. Another common implementation approach is using an 'overloadable' operator such as 'operator<<' or 'operator>>', at the cost of code readability.

One of the most mature and tested implementation of CppLINQ is by Microsoft Reactive Components research group [7]. It uses only standardized C++, making the library platform-independent. All references to CppLINQ in the paper relate to this particular implementation. Without limiting the generality of CppLINQ we will show a few examples in the context of high-energy physics.

3. Usability & Applications

The ROOT Framework [8] has become a standard for the field of high-energy physics. The tool is widely adopted for data analysis. A huge number of physicists are using ROOT on daily basis to perform data analysis on large scale data sets. A usual data analysis program starts with selection of the interesting pieces of data (interesting physics events). Physicists (data analysts) generally have a theory and expectations. They usually think in terms 'what' and not 'how', i.e they think about the logic and not the control of the Kowalski equation. For example, in order to check a theory or an expectation, one needs to 'select' all events, 'where' their properties match some criteria and 'order' them or 'group' them by certain properties. Translation of the process is straightforward to be done in terms of a comprehension syntax. Currently, the ROOT framework can be used only by an imperative manner (recommended is C++, but Python, Ruby and .NET bindings are available).

Lets consider a naive (and common) data layout, presented on Listing 2. A physics event data consists of a number of physics events, each one consists of number of particles.

```
//...
enum ParticleType { Photon = 0, Electron, Kaon, Pion };

struct Particle {
  ROOT::Math::XYZVector fPosition; // vertex position
  ROOT::Math::PtEtaPhiMVector fVector; // particle vector
  int fCharge; // particle charge
  ParticleType fType; // particle type
};

struct EventData {
  std::vector<Particle> fParticles; // particles of the event
};
std::vector<EventData> Events;
```

Listing 2: Physics event data sample

Our example data model is serialized on disk in the form of a ROOT tree in a ROOT file. Reading the data from disk in ROOT5 is tedious and cumbersome for various reasons. A major obstacle is ROOT5's non-conforming C++ interpreter and its lack of template support. This results in an awkward, non-type-safe implementation of data access. Moreover, iteration over the data entries with iterators is not possible.

Since the new C++ interpreter cling [9] came into service in ROOT6, ROOT can interact with very complex C++ constructs (for example, none of STL/Boost has to be hidden anymore, meaning ROOT can be exposed to heavily templated code). This allows implementing a strongly-typed ROOT file reader (aka TreeReader). The type safety comes through the

TTreeReaderValue (Listing3 line 4), where the user specifies the expected types and ROOT6 can report if there was a type mismatch.

```
1 void ReadEventData(std::vector<std::vector<Particle>>& v) {
2   std::unique_ptr<TFile> myFile = std::make_unique<TFile>(TFile::Open("eventdata_s99.root"));
3   TTreeReader tree("tree", myFile);
4   TTreeReaderValue<std::vector<Particle>> particles_value(tree, "fParticles"); //
5   while (tree.Next()) {
6      v.push_back(*particles_value); // particles_value is dereferenced as a 'normal' iterator.
7   }
8 }
```

Listing 3: Reading event data using TTreeReader class in ROOT6

The nested data structures can be flattened by using the TTreeReaderArray (line 4 on Listing 4) concept, which iterates type-safely over iterable containers.

```
1 void ReadEventDataFlat(std::vector<Particle>& v) {
2   std::unique_ptr<TFile> myFile = std::make_unique<TFile>(TFile::Open("eventdata_s99.root"));
3   TTreeReader tree("tree", myFile);
4   TTreeReaderArray<Particle> particles(tree, "fParticles"); //
5   while (tree.Next()) {
6     for (auto& p: particles)
7       v.push_back(p);
8   }
9 }
```

Listing 4: Reading event data elements using TTreeReaderValue class in ROOT6

ROOT6's TTreeReader helps the user express iterations over ROOT trees by implementing the std::iterator pattern. The inclusion of the TTreeReader in ROOT6 allows CppLINQ to operate on data coming directly from ROOT files in the form of tuples or trees. We will show three examples of event selection using the comprehension syntax. The new data reader is flexible and it is effortless to integrate it in new environments such as CppLINQ. All examples can run as ROOT6 macros or at the ROOT6 prompt [10].

Listing 5 shows how to open a ROOT file and prepares the TreeReader. Then using CppLINQ the number of events is calculated (line 7). Line 9 computes the count of all particles in all events in our data. Line 11 displays the particle count in the first 10 events.

```
 1 root [0] #include "EventData.h"
 2 root [1] TFile* f = TFile::Open("eventdata_s99.root");
 3 root [2] TTreeReader t("tree", f);
 4 root [3] TTreeReaderArray<Particle> particles(t, "fParticles");
 5 root [4] #include "linq.hpp"
 6 root [5] using namespace cpplinq;
 7 root [6] from(t).count() //
 8 (typename std::iterator_traits<iterator>::difference_type) 200
 9 root [8] from(t).select([&](Long64_t){ return from(particles).count();}).sum() //
10 (size_t) 9729
11 root [9] from(t).take(10).select([&](Long64_t){ return from(particles).count();}).sum() //
12 (size_t) 492
```

Listing 5: CppLINQ at ROOT6's prompt

All code snippets use 'standard' LINQ operators [11]. Most of the operators take a closure as a parameter in the form of C++11 lambda function. The lambda functions are used to express the concrete rules to be held for every event selection.

Listing 6 shows how to select all events from a given ROOT tree, which have at least four particles with more than 15 GeV energy.

```
void Example1(const TTreeReader& tree, float Emin = 15.0) {
  TTreeReaderArray<Particle> particles(tree, "fParticles");
  typedef decltype(from(particles)) ParticleItr_t;
  auto goodParticles
    = from(tree)
      .select([&](Long64_t) { return from(particles); })
      .where([&](const ParticleItr_t &plist) {
         return plist.where([&](const Particle &p) { return p.fVector.E() > Emin; }).count() >= 4;});
  //...
```

Listing 6: Selection of events having at least 4 particles with energy more than N GeV

Listing 7 demonstrates mixing imperative concepts into CppLINQ. Alternatively, in the particular case one could use the default_if_empty operator. If there are cases where one needs to rely on the host programming language, it is possible to switch back to imperative mode.

```
void Example2(const TTreeReader& tree) {
  TTreeReaderArray<Particle> particles(tree, "fParticles");
  auto result = from(from(tree)
                     .where([&particles](Long64_t) {
                        bool hasAtLeast2 = from(particles)
                                           .where([](const Particle& p) { return p.fType == Pion; })
                                           .count() > 2;
                        auto electrons = from(from(particles)
                                              .where([](const Particle& p) { return p.fType == Electron; }))
                                          .select([](const Particle& p){ return p.fVector.Pt(); });
                        if (hasAtLeast2 && !electrons.empty())
                          return electrons.max();
                        return 0.;
                     }))
                .where([](double pt){ return pt != 0; }).count();
   //...
```

Listing 7: Select all events having at least 2 pions and show the momentum of the leading electron

```
void Example3(const TTreeReaderArray<Particle>& particles) {
  auto PosPtSum = from(tree).select([&](Long64_t) {
                             return from(particles)
                                    .where([](const Particle& p) { return p.fCharge > 0; })
                                    .select([](const Particle& p) { return p.fVector.Pt(); });
                                    .sum();
                           });
 }
 auto h = new TH1F("ptSum", "Sum p_T of events; p_T [GeV]", 200, 0, 500);
 from(PosPtSum).all([&h](double pt){h->Fill(pt); return true;});
 //...
```

Listing 8: Calculate the event momentum distribution for all positively charged particles

Listing 8 shows how CppLINQ works with the TTreeReaderArray by flattening the event data structure and iterating over all particles from in events in order to sum the momenta of the individual particles. More examples and demos can be found online [10].

4. Conclusion & Future work

CppLINQ is a platform-independent, open-source library, which uses well-standardized features of C++. It provides concepts to abstract out details of the algorithms. Combined with ROOT6. they can be helpful in the analysis of physics events. The library relies on iterator design pattern [12]. It works smoothly with STL and Boost.

Another complementary and fundamental concept of the technology is the delayed, on request, computation. They introduce parallelism through *expression trees* and will be a subject of a future paper. The technology is already adopted in cloud systems. They prove that the delayed computation and computational abstraction can be beneficial in many cases. Introducing the delayed computation concept from CppLINQ could enhance the current communication model between users and batch systems or between ROOT and PROOF, for example.

Abstracting out the control of the program opens a lot of opportunities. For example, it would be much easier to translate a data flow visual programming language for physics data analysis into C++. The data flow visual language constructs could be compiled into their CppLINQ representations, and vice-verse – CppLINQ programs could be visualized by a data flow visual representation, making them more understandable.

Certainly, the domain is rather new and opens a broad research and development horizon. While CppLINQ needs some more work to become more robust, mixing comprehensions with C++ is beneficial in many cases in the field of high-energy physics. It provides the necessary computational abstraction layer, which can be a mediator between programmers and tools. Such tools could perform powerful analyses, transformations, optimizations and parallelizations dynamically and interactively.

References

[1] MSDN VARIANT structure URL `http://msdn.microsoft.com/en-us/library/windows/desktop/ms221627%28v=vs.85%29.aspx`

[2] Cantu M 2008 *Essential Pascal* (CreateSpace Independent Publishing Platform) chap 10

[3] Buneman P, Libkin L, Suciu D, Tannen V and Wong L 1994 Comprehension syntax *SIGMOD Rec.* **23** 87–96 ISSN 0163-5808 URL `http://doi.acm.org/10.1145/181550.181564`

[4] Hejlsberg A, Wiltamuth S and Golde P 2003 *C# Language Specification* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.) ISBN 0321154916

[5] Lloyd J W 1994 Practical advantages of declarative programming *Joint Conference on Declarative Programming, GULP-PRODE* vol 94 p 94

[6] Kowalski R 1979 Algorithm = logic + control *Communications of the ACM* **22** 424–436 ISSN 0001-0782 URL `http://doi.acm.org/10.1145/359131.359136`

[7] Microsoft reactive components team reactive extensions URL `http://msdn.microsoft.com/en-us/data/gg577609.aspx`

[8] Brun R and Rademakers F 1997 ROOT – an object oriented data analysis framework *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* **389** 81–86 ISSN 0168-9002 new Computing Techniques in Physics Research V URL `http://www.sciencedirect.com/science/article/pii/S016890029700048X`

[9] Vasilev V, Canal P, Naumann A and Russo P 2012 Cling – the new interactive interpreter for ROOT 6 *Journal of Physics: Conference Series* **396** 052071 URL `http://stacks.iop.org/1742-6596/396/i=5/a=052071`

[10] Vassilev V ROOT 6 examples with cpplinq URL `https://github.com/vgvassilev/RxCpp/tree/master/Ix/CPP/samples`

[11] MSDN The .NET standard query operators URL `http://msdn.microsoft.com/en-us/library/bb394939.aspx`

[12] Gamma E, Helm R, Johnson R and Vlissides J 1995 *Design Patterns: Elements of Reusable Object-oriented Software* (Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc.) ISBN 0-201-63361-2