Type safe physics analysis framework

Nobu Katayama

Physics division
High Energy Accelerator Research Organization (KEK)
Oho 1–1, Tsukuba-shi, 305 Japan

A prototype framework for data analysis for B physics was designed and implemented with type safety and extensibility in mind. The analysis of B meson decays can become very complicated as many decay modes must be looked at at the same time. In order to avoid programming mistakes in the analysis it is desirable to make each decay mode a new type (class). Using template and inheritance a new technique was developed to realize such requirements.

Key words: C++;type safety;data analysis

1 Introduction

Typically the physics data analysis is done in stages. First the physics objects are reconstructed from the raw data taken from the detector. The data summary tape (DST) is written and the physicists are ready for the final physics data analysis. A physicist writes programs to analyze the DST based on the hypotheses he/she is trying to prove. For example in the study of the B meson decays, the physicist might want to study the B^o meson decaying into the J/ψ and the K short mesons. As neither J/ψ nor K_s meson is stable in the detector, the physicist tries to observe their decay into a pair of leptons and charged pions respectively. These decays (or decay chains) occur at certain branching fractions. (Some are known and compiled in Particle Data Table. Others are not known yet.) Therefore a typical procedure of the physics data analysis is as follows; using a programming language of his/her choise, the physicist describes the decay chains and find such candidates in the events. Then he/she studies the properties of the candidates using histograms and/or n-tuples and visualizing them using tools such as PAW. The programming language used to describe the physics hypotheses is typically FORTRAN. Some physicists have used other languages such C, Pascal. Several specialized programming language have been invented for physics analysis. The most famous one is KAL developed at DESY. The author had written a package called CABS and have presented at previous CHEP conferences. The package has been used by members of the CLEO experiment successfully. Recently C++ has been getting a lot of attention and is considered suitable for the physics data analysis because the physics objects such as tracks and particles can be abstracted into classes. It is very natural to use classes for the physics objects. However, when analyzing a very complicated decay chain or many decay chains at the same time it would be very desirable if each decay chain can be made into a class. A new technique was developed to realize such a requirement. Using inheritance and template of C++ this package allows the user to create a new decay chain class. Things like looping over all candidates without multiple counting are taken care of by the base classes. With this technique the C++ compiler will catch most of mistakes that the user might make such as misusing one decay daughter as another. If the user uses FORTRAN or even C++ with generic particle classes such errors are not caught by the compiler as daughters of the decay chain are built using the same types.

2 Framework without type safety

Using the C++ programming language, one can write a framework that helps novice users to do the complicated physics analysis. The framework should handle the reconstructed objects such as charged tracks and photons, "composite" particles, the lists and combinatorics of the objects. For example the following program reconstructs the $D^{*+} \to D^o \pi^+, D^o \to K^- \pi^+$ decay candidates using such framework.

```
class Definition : public list<Particle*> {
     ... };

template<class T>class DefinitionTR : public list<Particle*> {
    typedef Particle *(*analysis_functionTR)(const T&);
    DefinitionTR(list<T>&,analysis_functionTR);
    ... };

void analyze() {
  //
  // Create piplus, piminus, Kplus and Kminus particles.
  //
  DefinitionTR<Track> piplus_list(track_list, select_piplus);

  DefinitionTR<Track> Kplus_list(track_list, select_Kplus);
  // Make an alias for Kminus_list as the ch. conj. of Kplus_list.
  list<Particle*> &Kminus_list(Kplus_list.cc());
  //
```

```
  // Make Dzero and Dstar using Definition class
  //
  Definition Dzero_list(Kminus_list, piplus_list, analyze_dzero);

  Definition Dstar_list(Dzero_list, piplus_list, analyze_dstar);
  //
  // Build combinatorics
  //
  Definition_registry::instance()->build_all();
```

where track_list is the list of charged track objects, select_pi(K)plus is a user function containing criteria to select good charged π (K) candidates, analyze_dzero(dstar) to select good $D^o(D^{*+})$ candidates. As you can see, Dstar_list will become a list of successful $D^{*\pm}$ candidates. In this example all candidates are type "Particle" and therefore the user may write by mistake:

```
  Definition Dstar_list(Kplus_list, piplus_list, analyze_dstar);
```

and the compiler will not catch this mistake.

3 Framework without type safety

A framework with much stronger typing can be created using templates.

```
void analyze() {
  //
  // Create piplus, piminus, Kplus and Kminus particles.
  //
  DefinitionTR<Pion, Track> piplus_list(track_list, select_piplus);

  DefinitionTR<Kaon, Track> Kplus_list(track_list, select_Kplus);
  // Make alias for Kminus_list
  list<Kaon*> &Kminus_list(Kplus_list.cc());
  //
  // Make Dzero and Dzerobar using Definition class
  //
  DefinitionTS2<Dzero, Kaon, Pion>
     Dzero_list(Kminus_list, piplus_list, Dzero::analyze);

  DefinitionTS2<Dstar, Dzero, Pion>
    Dstar_list(Dzero_list, piplus_list, Dstar::analyze);

  Definition_registry::instance()->build_all();
```

```
}
```

However, this framework requires the user to define Dzero and Dstar classes like:

```
class Dzero : public Particle {
  Dzero(const Pion&, const Kaon &);
  ...
};
class Dstar : public Particle {
  Dstar(const Dzero&, const Pion &);
  ...
};
```

On the other hand the user can also add member functions like Dzero::pi() returning (a reference to) the daughter Pion object and use it like pi() in Dzero::analyze function instead of daughter(0) which does not have a specific type of the daughter particle (Pion).

4 Extensible framework

The author of the framework may not be the best person to provide the "Particle" class. The experts of different analysis categories may be able to write a better base class for the analysis. It is not simple because even when class D is derived from B, list of D cannot be used as the list of B:

```
class D : public B {};
...
list<D*> dlist;
...
list<B*> & bl(dlist); // compile error
```

This makes almost impossible to write a generic combinatorics engine without a fixed base particle. However, using templates a framework which allows the user to choose his/her own Particle base class can be written.

5 Summary

Using templates extensively, a framework that is statically type safe and flexible to take choices of the base particle class can be created for the analysis of B meson decays with many complicated final states.