Table of Contents

Physics Cut and Expression Parser

Complete:

Purpose

The Physics Cut and Expression Parser is an utility that allows to define cuts and expressions defining mathematical combination of object variables with a human-readable string. The string is parsed and interpreted in terms of Reflex member functions for a specific object type, and can be used to apply cuts at run time in configurable common modules, like with Generic Selectors.

Where it is currently used

It is used in common selector and combiner modules. Examples are:

- Generic Candidate Selector
- Generic Candidate Combiner
- Histogramming utilities

Examples of string cut configurations

Modules using string cuts as parameters will have configurations similar to the following:

```
goodMuons = cms.EDFilter("CandViewSelector",
  src = cms.InputTag("allMuons"),
  cut = cms.string("pt > 5.0")
)
```

where the cut `"pt > 5.0"` will be applied to select the "good" muon candidates.

Suppported operators and functions

The following variables and operators are supported. Parentheses can be used to disambiguate operator precedence.

Variables mapped to object methods or variables

Any method can be used as variable name to which apply cuts for a given object type provided that:

- returns a value convertible to `double`
- is not a modifier (i.e.: is declared as `const`)
- requires either no arguments or constant arguments: integers, floating point numbers, strings (using either single or double quotes), and enums (entered as strings, with quotes)
- just like in C++, you can omit arguments that have a default value
- if the method takes no arguments, you can omit the braces (e.g. `pt` instead of `pt()`)

In addition public data members of classes can be accessed (e.g. the field `emEt` of a `reco::CMS.MuonIsolation` object)

Examples of valid variables are:

- `pt`: calls `object.pt()`

- `covariance(0, 0)`: calls `object.covariance(0, 0)`
- `nCarrying(0.4)` calls the `'nCarrying'` method of reco::Jet specifying the fraction of carried energy.
- `bDiscriminator("trackCountingBJetTags")` on a `pat::Jet` calls the method `bDiscriminator` passing a string argument `"trackCountingBJetTags"`
- `isGood('AllGlobalMuons')` calls the `reco::Muon::isGood(reco::Muon::AllGlobalMuons)` metod. the 'AllGlobalMuons' string is converted automatically into the enum value.
- `userIso()` or `userIso` will call `userIso(0)` on a `pat::Photon` , as the method is defined as `userIso(int index=0)`

Reflex dictionary will be used to match variable names to methods.

Name resolution

The string expression parser can operate in two ways when trying to resolve names into methods and datamembers:

- **default** (static): the types of the objects are those declared at compile time; this means that if you create an expression parser for a base class type (e.g. `reco::Candidate`) it will be able to access only members defined in the base class. When operating in this mode, the parser will throw an exception at construction time if it finds a name that can't be resolved in a function or method.
- **lazy** (dynamic): the parser checks the runtime type of the specific object it's being evaluated on, and therefore can resolve all methods and datamembers of the type; this means that if you create an expression parser for a base class (e.g. `reco::Candidate`) and you evaluate it on an object from a derived class (e.g. `reco::Muon`) you have access to all the methods of the derived class. When operating in lazy mode, the parser will throw an exception at execution time if it finds a name that can't be resolved as a function or method.

Both methods work, and the performance is essentially the same (the dynamic method lookup is performed only when a new object type is encountered).

Nested object methods

Methods that return objects by value, by reference or by pointer, or reference to objects (`edm::Ref`, `edm::Ptr`, `edm::RefToBase`) can be used to call subsequent object methods, provided that the subsequent method satisfies the conditions listed in the above section.

Examples of such calls are:

- `track.pt > 10`:
  calls the C++ equivalent of `object.track().pt() > 10`, where `track()` returns an `edm::Ref<reco::TrackCollection>`
- `min(daughter(0).pt,daughter(1).pt)`:
  calls the C++ equivalent of `std::min(object.daughter(0)->pt(), object.daughter(1)->pt())`, where `daughter(i)` returns a pointer to a `reco::Candidate`.

Mathematical operators

The following operators are supported with the usual precedence:

- `+`: addition
- `-`: subtraction or unary minus sign
- `*`: multiplication
- `/`: division
- `^`: power raising

Mathematical functions

The following functions are supported:

- `abs(x)`: absolute value
- `acos(x)`: arc cosine
- `asin(x)`: arc sine
- `atan(x)`: arc tangent
- `atan2(y,x)`: arc tangent of ratio `y/x` using arguments sign
- `chi2prob(c2,ndf)`: the integrated probability of having a chi-square greater than `c2` with `ndf` degrees of freedom (`ROOT::Math::chisquared_prob(c2,ndf)`).
- `cos(x)`: cosine
- `cosh(x)`: hyperbolic cosine
- `deltaPhi(x,y)`: signed Δ function, as defined in DataFormats/Math/interface/deltaPhi.h
- `deltaR( 1, 1, 2, 2)`: ΔR function, as defined in DataFormats/Math/interface/deltaR.h
- `exp(x)`: exponential
- `hypot(x,y)`: sqrt(x*x + y*y), as in the math standard library
- `log(x)`: natural logarithm
- `log10(x)`: base-10 logarithm
- `min(x, y)`: smallest element
- `max(x, y)`: largest element
- `pow(x,y)`: power raising (identical to `x^y`)
- `sin(x)`: sine
- `sinh(x)`: hyperbolic sine
- `sqrt(x)`: square root
- `tan(x)`: tangent
- `tanh(x)`: hyperbolic tangent
- `test_bit(number, index)`: tests if bit of index `index` is 1 in the number `number`, that is what you would get in C++ doing `(int(number) >> int(index)) & 1`. Note that bit numbers start from 0, not 1.

Comparison operators

The following comparison operators are supported:

- `=`: equal to
- `!=`: not equal to
- `>`: greater than
- `>=`: greater or equal than
- `<`: less than
- `<=`: less or equal than
- `==`: equal than

Boolean operators

The following boolean operators are supported with the usual precedence:

- `&` or `&&`:: logical AND
- `|` or `||`: logical OR
- `!`: logical NOT

Conditional evaluation

Sometimes you want to evaluate an expression only if a condition is true. In C++, you do this with the ternary operator ?: writing `condition ? expression_if_true : expression_if_false` which will evaluate only one of the two expressions according to the value of the condition. This feature is implemented also in the string parser for recent releases, but the syntax requires an extra ? at the beginning* of the expression. That is, you have to write **`? c ? x : y`** instead of `c ? x : y`

For example, to take the number of silicon hits on track for a `reco::Muon` object, assigning a value of 0 for muons that don't have a silicon track at all, you would have to write **`? track.isNonnull ? track.numerOfValidHits : 0`**

Element access in vector-like containers

The `[index]` notation can be used to retrieve elements from a vector-like container. E.g. you can do `overlaps("jets")[0].pt` to get the pt of the first element of the CandidatePtrVector returned by the method `overlaps(label)` of a pat object. This was implemented only recently, so it might not yet be available in the release you're using.

A few extra comments:

- the array access is supported only for real C++ classes that define a `const operator[]`, not for C arrays
- you can't use the `operator[]` directly on the top level object (e.g. it does not work for `StringCutObjectSelector<vector<X>>`, only on some `StringCutObjectSelector<T>` where `T` has a method that returns `vector<X>`).
- any form of `operator[]` is supported, not just those that take as argument a single integer.

Using the string-configurable generic object selector

For a type `T`, a generic selection functor is defined with the type:

- `StringCutObjectSelector<T>`

It returns `true` or `false` if an object passes or fails a selection specified as a string a constructor parameter.

It can be used as follows:

```
StringCutObjectSelector<reco::Track> select( "pt > 15.0 & abs(eta) < 2");
reco::Track trk = ...; // get a track
bool pass = select( trk );
if( pass ) { /* ... */ }
```

If you want to enable the dynamic name resolution, you have to pass an extra parameter in the constructor

```
StringCutObjectSelector<reco::Track> select( "pt > 15.0 & abs(eta) <2", true);
```

or you can pass an extra argument in the template

```
StringCutObjectSelector<reco::Track, true > select( "pt > 15.0 & abs(eta) <2");
```

Using the string-configurable generic object function

For a type `T`, a generic selection functor is defined with the type:

- `StringObjectFunction<T>`

It returns for any object passed, the value of an expression specified as a string a constructor parameter.

It can be used as follows:

```
StringObjectFunction<reco::Track> f( "px^2+py^2");
reco::Track trk = ...; // get a track
double ptSquare = f( trk );
```

Just like for the functor, you can enable dynamic name resolution by specifying an extra argument in the constructor or in the template.

Caveat with EDM provenance tracking system

The string-based cuts and expressions can be used with framework modules, and the specified cuts or expressions will be saved in the event files within the provenance tracking system.

Anyway, users should be aware that the cuts will be saved as string, so deriving the numeric values of the specified cuts will be a non-trivial operation because it will require a parsing of the cut string.

Performance issues

Parsing the expression string is done only once, usually in the constructor of selector and function objects, so it does not add significant overhead, provided that the same selector or function object is kept for more events.

What may cause a slight performance loss is the actual evaluation of expression. The parses, in fact, creates a composite structure in memory that invokes a virtual function call for every syntactical element in the expression.

Moreover, the passed object (track, in the example) has to be "converted" into a Reflex object in order to be evaluated.

Those two elements may be cause some performance overhead for critical applications, like HLT. For those application, compiled C++ expressions are expected to have better CPU performances.

For those analysis applications where configuration flexibility is an advantage w.r.t. extreme speed, those issues may be not important.

Review Status

Editor/Reviewer and date	Comments
LucaLista - 08 Jan 2008	page author
BenediktHegner - 30 Jul 2010	removed outdated information
GiovanniPetrucciani - 06-Oct-2010	document new features and explain a bit the name resolution

Responsible: LucaLista
Last reviewed by:

This topic: CMSPublic > SWGuidePhysicsCutParser
Topic revision: r23 - 2013-12-12 - GiovanniPetrucciani