TODO

Ábel Sinkovics[1] Zoltán Porkoláb[2]

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary

Abstract

Keywords: C++, Boost, Template metaprogramming

1 Introduction

Boost has a template metaprogramming library [1] providing tools to build template metaprograms in a structured way. The library implements commonly used utilities and algorithms in an extendible and reusable way. It helps reducing the amount of boilerplate code when developing C++ template metaprograms.

C++ template metaprogramming follows the functional paradigm (TODO citek), thus all the experience gained in the field of functional programming could be reused in C++ template metaprogramming. When developers intentionally follow the functional paradigm they can easily apply the techniques developed over the years. To follow the functional paradigm directly the tools have to be developed in functional programming in mind. In this paper we evaluate some functional aspects of the boost metaprogramming library and propose new tools for more direct support of functional programming.

2 Laziness

A value in template metaprogramming and a nullary function are two different things: a value is an arbitrary class or compile-time data, such as an int or bool constant, wrapped by a wrapper class. A nullary metafunction is a template metafunction with 0 arguments. It's represented by a class with a nested class called `type`, which is the value of the metafunction. Here is an example of a value:

[1] Email: `abel@elte.hu`
[2] Email: `gsd@inf.elte.hu`

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

```
int_<13>
```

and here is an example of a nullary metafunction:

```
struct thirteen
{
  typedef int_<13> type;
};
```

Both of these things represent a value in template metaprogramming, but there is a difference: a nullary metafunction is a metafunction which is not evaluated until it's value is needed for the first time. It's code may contain errors, unless it's value is used somewhere it will not break the compilation. A nullary metafunction can be built from any template metafunction by applying it to arguments but not accessing the nested `::type`. For example

```
plus<int_<1>, int_<2> >
```

is a nullary metafunction.

Nullary metafunctions can be used to implement lazy evaluation in C++ template metaprogramming because they are not evaluated until their nested `::type` class is used. Should we need it we can also enforce eager evaluation by directly accessing the nested `::type` class. Here is the lazy and eager evaluation of the same function as an example:

```
// Lazy evaluation
plus<int_<1>, int_<2> >

// Eager evaluation
plus<int_<1>, int_<2> >::type
```

In the code

```
struct infinite {};

template <class a, class b>
struct divide :
  if_<
    typename equal_to<b, int_<0> >::type,
    infinite,
    typename divides<a, b>::type
  >
{};
```

we create a new `infinite` class for representing the infinite value and a new `divide` function which divides it's two operands. In case the second operand is zero, it returns `infinite`. This code doesn't work. `divide<int_<3>, int_<0> >::type` doesn't evaluate to `infinite`, it breaks the compilation. The reason why the compiler generates an error is that the second case of `if_` is evaluated eagerly. `if_` takes values as arguments, it expects eager evaluation of both cases.

boost::mpl tackles this problem with `eval_if` which takes nullary metafunctions as arguments for the true and false cases. Doing this, `eval_if` can evaluate the selected one only, avoiding instantiation of invalid templates. Here is the correct version of the above example using `eval_if`:

```
struct infinite {};

template <class a, class b>
struct divide :
  eval_if<
    typename equal_to<b, int_<0> >::type,
    identity<infinite>,
    divides<a, b>
  >
{};
```

As you can see `infinite` had to be passed to `identity`. A value can be transformed into a nullary metafunction by passing it to `identity`.

A class we'd like to use as a value in a template metaprogram can be designed in a smart way: you can add itself to it as a nested type called `type`:

```
struct infinite
{
  typedef infinite type;
};
```

By doing it both functions expecting a nullary metafunction and functions expecting a value will accept it, and it will behave as expected in both situations. For example the advanced `infinite` simplifies the definition of `divide`:

```
template <class a, class b>
struct divide :
  eval_if<
    typename equal_to<b, int_<0> >::type,
    infinite,
    divides<a, b>
  >
{};
```

Integral wrappers in boost use this: they are nullary metafunctions and evaluate to themselves.

Consider a more complicated, but still simple example:

```
template <class a, class b>
struct some_calculation :
  eval_if<
    typename equal_to<b, int_<0> >::type,
    // ....,
    eval_if<
      typename less<
        typename divides<a, b>::type,
        int_<10>
      >::type,
```

```
      // ...,
      // ...
    >
  >
{};
```

In this metafunction taking two arguments we need to make a decision based on the quotient of the two arguments but we have to handle the case when the second argument is zero, this is what the outer `eval_if` is for. The code above doesn't work when the second argument, `b`, is zero because even though the branches of `eval_if` are evaluated lazily, it's condition isn't. Thus the condition of the nested `eval_if` is instantiated when `some_calculation` is instantiated, regardless of the value of the outer `eval_if`'s condition. When the value of `b` is zero, instantiation of the nested `eval_if`'s condition generates an error.

We suggest a completely lazy version of `eval_if` which takes a nullary metafunction as it's condition. It's implementation is straight forward:

```
template <
  class condition,
  class true_case,
  class false_case
>
struct lazy_eval_if :
  eval_if<
    typename condition::type,
    true_case,
    false_case
  >
{};
```

Using `lazy_eval_if` our more complicated example can be solved as well:

```
template <class a, class b>
struct some_calculation :
  eval_if<
    typename equal_to<b, int_<0> >::type,
    // ....,
    lazy_eval_if<
      apply<less<divides<a, _1>, int_<10> >, b>,
      // ...,
      // ...
    >
  >
{};
```

3 Function composition

Suppose we have to write a metafunction taking a number in the range $[-\pi, \pi]$ as it's argument and returning the square of the tangent of that number or a special class called not_a_number in case the argument is $\pm\frac{\pi}{2}$.

Assume we have template metafunctions to calculate the absolute value (abs) and the tangent (tan) of a number. tan breaks the compilation when evaluated with a number the tangent of which is not defined. The following solution doesn't work

```
template <class deg>
struct square_tangent :
  eval_if<
    typename equal_to<
      typename abs<deg>::type,
      divides<pi, int_<2> >::type
    >::type,
    not_a_number,
    square<typename tan<deg>::type>
  >
{};
```

when the argument is $\pm\frac{\pi}{2}$ because the C++ compiler tries instantiating both cases of eval_if and the instantiation of the second case generates an error. eval_if takes nullary metafunctions as second and third arguments, thus they are evaluated lazily, but those nullary metafunctions may not take nullary metafunctions as arguments. In case the function we use in the true or false case of an eval_if doesn't take nullary metafunctions as arguments, it's arguments need to be evaluated prior to the evaluation of the function itself. In our example the false case of the eval_if is the evaluation of square with the value of tan<deg> as it's argument. square doesn't accept nullary metafunctions as arguments, we have to evaluate tan<deg> before evaluating square. We embedded square in an eval_if expression, thus we have to evaluate tan<deg> before evaluating eval_if. It means that we have to calculate the tangent of a value before we could check if it's a valid operation or not.

If every template metafunction took nullary metafunctions as arguments we wouldn't have this problem. Requiring all metafunctions to take nullary metafunctions as arguments would solve the problem, but we can't ensure that and we can't affect third-party libraries developed by someone else.

Another solution is factoring the code of the branches out to external classes and only the chosen one is instantiated:

```
template <class deg>
struct square_tangent_impl :
  square<typename tan<deg>::type>
{};

template <class deg>
struct square_tangent :
  eval_if<
    typename equal_to<
      typename abs<deg>::type,
```

```
      divides<pi, int_<2> >::type
    >::type,
    not_a_number,
    square_tangent_impl<deg>
  >
{};
```

This solution works, but in this case the business logic of the function is scattered in multiple metafunctions which makes it difficult to understand. The more selection points a function has the more splits it requires.

A third solution is building anonymous template metafunctions in place, so we don't have to move parts of the business logic to external classes. We can do it using `boost::mpl`'s lambda expressions. The lambda expression is then evaluated lazily by `eval_if`. The lambda-based implementation of our example metafunction

```
template <class deg>
struct square_tangent :
  eval_if<
    typename equal_to<
      typename abs<deg>::type,
      divides<pi, int_<2> >::type
    >::type,
    not_a_number,
    apply<square<tan<_1> > >, deg>
  >
{};
```

solves the problem and keeps the business logic in one place. But when we have to deal with template metafunction classes [6] instead of template metafunctions, or template metafunction class arguments it has a large syntactical overhead. If `square` and `tan` are template metafunction classes, this solution is still difficult to write, understand and maintain:

```
template <class deg>
struct square_tangent :
  eval_if<
    typename equal_to<
      typename abs<deg>::type,
      divides<pi, int_<2> >::type
    >::type,
    not_a_number,
    apply<square::apply<tan::apply<_1> > >, deg>
  >
{};
```

We had to use complex tools to solve a rather simple problem which is applying a chain of functions to an argument. It is so common that functional languages often have a special operator for it in the language or the standard library. Due to the functional nature of C++ template metaprograms introducing it in template metaprogramming could reduce the complexity of the code of metaprograms. We propose a `compose` metafunction for function composition. It takes any number of metafunction classes as arguments and evaluates to an anonymous metafunction class implementing the chain of the arguments. The C++ standard hasn't got variadic template [8,?] support, but there are workarounds [2]. This metafunction can be implemented by boost lambda expressions or manually as well, it's implementation is straight forward. Using it we get a cleaner implementation of our sample function:

```
template <class deg>
struct square_tangent :
  eval_if<
    typename equal_to<
      typename abs<deg>::type,
      divides<pi, int_<2> >::type
    >::type,
    not_a_number,
    apply<compose<square, tan>, deg>
  >
{};
```

4 Currying

Currying is supported by several functional languages. When we have a function taking n arguments we can apply one argument to it and get a function taking n-1 arguments, and so on. When we have a function taking only 1 argument and we apply that one argument we get the value of the function. This is a special form a partial function application which is difficult to simulate using lambda expressions in boost::mpl.

We're going to use the following example to demonstrate what Currying means in C++ template metaprogramming. Consider a function that calculates the area of a rectangle.

```
template <class x1, class y1, class x2, class y2>
struct area :
  multiplies<minus<x2, x1>, minus<y2, y1> >
{};
```

This function takes 4 numbers as arguments: two opposite points of the rectangle. It takes 4 arguments in one step and calculates the result immediately. If this function was using currying, it would be a function accepting one number. The value of this function would be an anonymous function taking 1 number as arguments. The value of that function would be another anonymous function taking 1 argument. The value of that function would be the area of the function. It would be something like the following template metaprogram:

```
template <class x1>
struct area
{
```

```
    struct type
    {
      template <class y1>
      struct apply
      {
        struct type
        {
          template <class x2>
          struct apply
          {
            struct type
            {
              template <class y2>
              struct apply :
                multiplies<minus<x2, x1>, minus<y2, y1> >
              {};
            };
          };
        };
      };
    };
  };
```

As you can see adding currying to a function by hand has a large syntactical overhead. A large amount of boilerplate could be the result of using it. We propose a template metafunction taking a template metafunction class and the number of arguments as arguments and building the curried version automatically. The generated metafunction maintains a compile-time list internally and every time a new argument is passed to it, it simply stores the argument in the list. When all of the arguments are available it applies the full argument list to the lambda expression. There is no need for preprocessor based workarounds in this solution, it can be completely implemented using C++ template metaprogramming techniques. Using this metafunction the above example can be generated from the simple `area` metafunction we presented for the first time:

```
curry<quote4<area>, int_<4> >
```

Note that we had to use `quote4` from `boost::mpl` because `curry` expects template metafunction classes while we had a template metafunction, thus we had to generate a metafunction class from it.

`curry` is a tool we can avoid writing a large amount of boilerplate code when we need currying making heavy use of automatic code generation in C++. In situations where we can't change the implementation of a metafunction because other codes rely on it or because it's coming from a third party library external currying support is the only option and in such cases this tool can do the hard work.

5 Summary

C++ template metaprogramming can save development and maintance effort when used well. Given that it's naturally following the functional programming paradigm (TODO cite milewksi) we have evaluated how the most widely used C++ template metaprogramming library, `boost::mpl` supports following the functional programming paradigm. We've seen that it's support for lazy evaulation is good and we've proposed an addition for further improvement. We've also evaluated the support for an often used task, the function composition and we've proposed an addition for further improvement. We've also proposed a way for automatically adding currying support to existing template metafunctions and metafunction classes. As a summary we've found that the tools available help following the functional programming paradigm, and we've proposed ways for improving this support.

6 Related work

Andrei Alexanderscu built template metaprogramming tools in his library called Loki [7]. He builds compile time lists called Typelists and uses them as a source of code generation. He doesn't talk explicitly about template metaprogramming and he doesn't mention it's functional aspects either.

FC++ [10] is a C++ library providing runtime functional programming support for C++. Template metaprograms are always evaluated at compilation time. The development of template metaprograms is different from runtime programs, thus they need different supporting tools to develop software following the functional paradigm.

Barotsz Milewski pointed out the commonalities between functional programming and C++ template metaprogramming in his talk and on his blog. He demonstrates the capabilities of C++ and C++0x to support the functional paradigm in template metaprograms but he doesn't consider the tools of the boost metaprogramming library and compatibility with those tools.

In [3] a tool transforming a simple language based on lambda expressions was presented. Lambda expressions form an NP-complete functional language [11]. Using lambda expressions strongly simplified the

In [4] transformation tool was presented which transforms code written in a simplified version of Clean, called E-Clean, to C++ template metaprograms. The generated code was more efficient than the hand-written C++ template metaprogram for the same problem.

References

[1] The boost metaprogram libraries.
`http://www.boost.org/doc/libs/1_39_0/libs/mpl/doc/index.html`

[2] The boost preprocessor metaprogramming library.
`http://www.boost.org/doc/libs/1_41_0/libs/preprocessor/doc/index.html`

[3] Ábel Sinkovics, Zoltán Porkoláb: *Expressing C++ Template Metaprograms as Lambda expressions*, In Tenth symposium on Trends in Functional Programming (TFP '09, Zoltn Horvth, Viktria Zsk, Peter Achten, Pieter Koopman, eds.), Jun 2 - 4, Komarno, Slovakia 2009., pp. 97-111

[4] Ádám Sipos, Zoltán Porkoláb, Viktória Zsók: *Meta<fun> – Towards a functional-style interface for C++ template metaprograms* In Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII, 2008/2, Cluj-Napoca, 2008, pp. 55-66.

[5] E. Unruh, *Prime number computation*, ANSI X3J16-94-0075/ISO WG21-462.

[6] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2004.

[7] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[8] Douglas Gregor, Jaakko Järvi, *Variadic templates for C++*, In Proceedings of the 2007 ACM symposium on Applied computing, March 11-15, 2007, Seoul, Korea Pp.1101-1108, 2007, ISBN:1-59593-480-4

[9] D. Gregor, J. Jrvi, G. Powell, *Variadic Templates (Revision 3)*, ISO SC22 WG21 TR N2080==06-0150.

[10] B. McNamara, Y. Smaragdakis, *Functional programming in C++*, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.

[11] Simon L. Peyton Jones: *The Implementation of Functional Languages*, Prentice Hall, 1987, [445], ISBN: 0-13-453333-9 Pbk