Functional Extensions to the 
Boost Metaprogram Library

Ábel Sinkovics¹

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

Abstract
Boost metaprogram library is one of the most important foundation for C++ template metaprogramming. The library implements commonly used compilation-time algorithms and meta-datastructures in an extendible and reusable way. Despite the well-known commonality of functional programming paradigm, boost mpl lacks a few important features to directly support functional style. In this paper we propose some new library elements to boost mpl for more explicit support of functional programming.

Keywords: C++, Boost::mpl, Template metaprogramming

1 Introduction

Templates are key language elements for the C++ programming language [3]. Apart its primary role – capturing commonalities of abstractions without performance penalties at runtime – they form the base of template metaprogramming. In 1994 Erwin Unruh used C++ templates and the template instantiation rules to write a program that is “executed” as a side effect of compilation [17]. It turned out that a cleverly designed C++ code is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [19]. These compile-time programs are called C++ Template Metaprograms and later has been proved to be form a Turing-complete sublanguage of C++ [4].

Today programmers write metaprograms for various reasons, like implementing expression templates [20], where we can replace runtime computations with compile-time activities to enhance runtime performance; static interface checking, which increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [10,13]; active libraries [21], acting dynamically during compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded

¹ Email: abel@elte.hu

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

In the last fifteen years major efforts have been made to create the fundations of template metaprogramming. These include the fundamental meta data structures and algorithms. Boost is one of the most important selection of third-party C++ libraries. The libraries are aimed at a wide range of C++ users and application domains. Boost makes extensive use of templates and has been a source of extensive work and research into metaprogramming in C++. Boost has a template metaprogramming library [22] 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 [25], 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.

In this paper we introduce some functional-based extensions to the boost metaprogram library. In Section 2 we discuss lazy evaluation of compile-time selection, in Section 3 we implement (meta)function composition, and Section 4 overviews currying. Related works are found in Section 5 and we summarize our results in Section 6.

2 Laziness

When there is a selection in a metaprogram, such as a boost::if_ or boost::eval_if, one path of execution is selected based on the condition of the selection. Evaluating functions on the other path may lead to an error, in these situations being able to evaluate expressions lazily is critical. We’ll examine how boost::mpl supports lazy evaluation in the selection constructs it provides and how they could be improved.

A nullary metafunction is a metafunction taking 0 arguments. [1] In template metaprogramming functions are always pure: they always have the same value when they are evaluated with the same arguments [1], thus nullary C++ template metafunctions always have the same value. Template metafunctions may take nullary metafunctions as arguments instead of values. A value and a nullary metafunction are almost the same, but a nullary metafunction is evaluated lazily. It’s evaluated when it’s value is used for the first time, and it may not be evaluated at all in case it’s value is not needed. 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 simple value in template metaprogramming:

```cpp
int_<13>
```
and here is an example of a nullary metafunction:

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

A nullary metafunction can be built from any template metafunction by applying it on arguments but not accessing the nested ::type. For example

```cpp
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. We can enforce eager evaluation by directly accessing the nested ::type class. Here is the lazy and eager evaluation of the same function as an example:

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

In the code

```cpp
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 its two operands. When 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`:

```cpp
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, \texttt{infinite} had to be passed to \texttt{identity} because \texttt{infinite} is a value, not a nullary metafunction. A value can be transformed into a nullary metafunction by passing it to \texttt{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 \texttt{type}:

\begin{verbatim}
struct infinite  {
  typedef infinite type;
};
\end{verbatim}

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 \texttt{infinite} simplifies the definition of \texttt{divide}:

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

Integral wrappers in boost use this: they are nullary metafunctions and evaluate to themselves. Consider a more complicated, but still simple example:

\begin{verbatim}
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,
      // ...
    >
  > {};
\end{verbatim}

In this metafunction 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 \texttt{eval_if} is for. The code above doesn’t work when the second argument, \texttt{b}, is zero because even though the branches of \texttt{eval_if} are evaluated lazily, it’s condition isn’t. Thus the condition of the nested \texttt{eval_if} is instantiated when \texttt{some_calculation} is instantiated, regardless of the value of the outer \texttt{eval_if}’s condition. When the value of \texttt{b} is zero, instantiation of the nested \texttt{eval_if}’s condition generates an error.

We propose a completely lazy version of \texttt{eval_if} which takes a nullary metafunction as its condition. It’s implementation is straightforward:
Sinkovics

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 its 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 evaluation of the true and false cases of eval_if happens lazily, but square takes a value as its argument, not a nullary metafunction, thus tan has to be evaluated eagerly by accessing it’s type member, and eager evaluation happens when square_tangent is instantiated. 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 eval-
uation 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 metafunc-
tions 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:

```cpp
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

```cpp
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 [1] 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:

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

When developing higher order metafunctions, and the metafunction classes are arguments of our metafunctions it gets more complicated. 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. It’s implementation requires variadic templates, but the C++ standard hasn’t got variadic template support, but there are workarounds. This metafunction can be implemented by boost lambda expressions or manually as well, it’s implementation is straightforward. Using it we get a cleaner implementation of our sample function:

```cpp
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 of partial function application which is difficult to simulate using lambda expressions
in boost::mpl. Given the functional nature of C++ template metaprograms [25], solutions to problems available in functional languages could be ported to C++ template metaprograms. When porting code written in a functional language keeping the logic the original code follows helps debugging and later improvement of the code. Functional codes make use of currying, it should be supported in C++ template metaprograms as well. We propose a solution for extending metaprograms with currying support without changing existing code.

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.

```cpp
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 argument. 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:

```cpp
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. Using it leads to writing a large amount of boilerplate code. 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 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:

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

8
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. When we need currying, `curry` is a tool we can avoid writing a large amount of boilerplate code with 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 Related work

Andrei Alexandrescu built template metaprogramming tools in his library called Loki [2]. 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++ [16] 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.

Bartosz Milewski pointed out the commonalities between functional programming and C++ template metaprogramming in his talk and on his blog [25]. 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 [14] 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 C++ template metaprograms.

In [15] a 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.

6 Summary

C++ template metaprogramming can save development and maintenance effort when used well. Given that it’s naturally following the functional programming paradigm [25] 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 evaluation 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.
References


