Workshop on Generative Programming 2008

Functional Extensions to the Boost Metaprogram Library

Ábel Sinkovics[1]

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

Abstract

The Boost metaprogram library is one of the most important foundations for C++ template metaprogramming. The library implements commonly used compilation-time algorithms and meta-datastructures in an extensible and reusable way. Despite the well-known commonality of template metaprogramming and the 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, functional programming

1 Introduction

Templates are key language elements for the C++ programming language [3]. Apart from their 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 template instantiation rules to write a program that is "executed" as a side effect of compilation [17]. It turned out that cleverly designed C++ code is able to utilise 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 template metaprogramming later have been proved to form a Turing-complete sub-language 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 compiler to check the requirements against template parameters at compile-time, i.e. they form constraints on template parameters

[1] Email: `abel@elte.hu`

[10,13]; *active libraries* [21], acting dynamically during compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages such as the Ararat system [7] for a type-safe SQL interface and `boost:xpressive` [26] for regular expressions.

In the last fifteen years major efforts were put into creating the foundations of template metaprogramming. These include the fundamental meta datastructures 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 [24] providing tools to build template metaprograms in a structured way. The library implements commonly used utilities and algorithms in an extensible and reusable way. It helps reduce the amount of boilerplate code when developing C++ template metaprograms.

C++ template metaprogramming follows the functional paradigm [27], thus experience gained in the field of functional programming can 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 with 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 extensions to the boost metaprogram library following the functional paradigm. 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], it is the implementation of thunks [22] in C++ template metaprogramming. Unfortunately the value of a nullary metafunction can only be accessed explicitly, it can not be used transparently. 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 its value is used for the first time, and it may not be evaluated at all in case its 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:

```
int_<13>
```

and here is an example of a nullary metafunction:

```
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

```
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 are the lazy and eager evaluations of the same function as an example:

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

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 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. `eval_if` can evaluate only the selected branch, 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` because `infinite` is a value, not a nullary metafunction. 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 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, its 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.

Unfortunately this problem can not be solved in a generic way, using a template that makes the arguments of a metafunction lazy. Such a solution could delay the evaluation of the arguments until the metafunction is evaluated, but then all arguments have to be evaluated because the metafunction expects eager evaluation. In non-trivial metafunctions, such as metafunctions implementing selection, the metafunction has to take arguments in a lazy way and make a decision on which one to evaluate, it can not be done externally.

We propose a completely lazy version of `eval_if` which takes a nullary metafunction as its condition. Its 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 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 its `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, its 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 its 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 [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:

```
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>
  > {};
```

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. Its implementation requires variadic templates, but the C++ standard hasn't got variadic template [5] support, but there are workarounds [25] and the upcoming standard, C++0x, will have variadic template support. There are implementations of this feature available as well. A future work is implementing `compose` using this new feature. This metafunction can be implemented by boost lambda expressions or manually as well, its implementation is straight forward. Here is an example implementation for a fixed number of functions to compose:

```
template <class f1, class f2> struct compose2 {
  template <class a> struct apply {
    typedef typename boost::mpl::apply<
        f1, typename boost::mpl::apply<f2, a>
      >::type type;
  };};
```

`compose3`, `compose4`, etc. can be implemented similarly, their implementation can be automatically generated using the Boost preprocessor metaprogramming library [25]. A `compose` function can be written to call the above:

```
struct unused {};

template <class f1 = unused, class f2 = unused,
  class f3 = unused, class f4 = unused>
struct compose;

template <class f1, class f2, unused, unused>
struct compose : compose2<f1, f2> {};

template <class f1, class f2, class f3, unused>
struct compose : compose3<f1, f2, f3> {};

template <class f1, class f2, class f3, class f4>
struct compose : compose4<f1, f2, f3, 4> {};
```

It uses default template arguments and template specialisation to detect the number of arguments and choose the right version of `composen`. By using `compose` 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 of partial function application which is difficult to simulate using lambda expressions in `boost::mpl`. Given the functional nature of C++ template metaprograms [27] 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.

```
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:

```
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.

It can be implemented using a function that collects it's arguments into a compile-time list:

```
template <class UnpackedMetafunctionClass,
  class ArgumentsLeft, class ArgumentList>
struct curryImpl : boost::mpl::eval_if<
    typename boost::mpl::equal_to<
      ArgumentsLeft, boost::mpl::int_<0>
    >::type,
    boost::mpl::apply<
      UnpackedMetafunctionClass, ArgumentList
    >,
    nextCurryingStep<
      UnpackedMetafunctionClass,
      ArgumentsLeft, ArgumentList
    > > {};
```

It takes the function to curry as it's first argument, `UnpackedMetafunctionClass`, the number of arguments to collect as it's second argument, `ArgumentsLeft`, and the list collected so far as it's third argument, `ArgumentList`. It's important for the function to curry to expect one argument, which is a compile-time list containing the arguments of the function. We have to use a helper metafunction class for it:

```
template <class UnpackedMetafunctionClass,
  class ArgumentsLeft, class ArgumentList>
struct nextCurryingStep {
  struct type {
    template <class T> struct apply :
      curryImpl<
        UnpackedMetafunctionClass,
        typename boost::mpl::minus<
          ArgumentsLeft, boost::mpl::int_<1>
        >::type,
        typename boost::mpl::push_back<ArgumentList, T>::type
      > {}; }; };
```

Using these functions we can implement our `curry` function:

```
template <class MetafunctionClass, class ArgumentNumber>
struct curry : curryImpl<
  boost::mpl::unpack_args<MetafunctionClass>,
  ArgumentNumber, boost::mpl::deque<> > {};
```

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. When we need currying, `curry` is a tool we can provide to 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

Todd Veldhuizen demonstrated how to implement non-tirival C++ template metaprograms [23]. He didn't present the functional aspects of template metaprogramming.

Andrei Alexanderscu 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 its 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 [27]. 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 [27] we have evaluated how the most widely used C++ template metaprogramming library, `boost::mpl` supports following the functional programming paradigm. We've seen that its 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: 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. The complexity of existing template metaprograms using can be simplified using the tools proposed in this paper by eliminating unnecessary helper metafunctions and moving the whole business logic of a metafunction into one location. As a summary we've found that the tools available help follow the functional programming paradigm, and we've proposed ways for improving this support.

References

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

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

[3] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[4] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[5] D. Gregor, J. 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

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

[7] Y. Gil, K. Lenz, *Simple and Safe SQL queries with C++ templates*, In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.

[8] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.

[9] D. R. Musser, A. A. Stepanov, *Algorithm-oriented Generic Libraries*, Software-practice and experience 27(7), 1994, pp.623-642.

[10] B. McNamara, Y. Smaragdakis, *Static interfaces in C++*, In First Workshop on C++ Template Metaprogramming, October 2000

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

[12] Z. Porkoláb, J. Mihalicza, Á. Sipos, *Debugging C++ template metaprograms*, In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM 2006, ISBN 1-59593-237-2, pp. 255-264.

[13] J. Siek, A. Lumsdaine, *Concept checking: Binding parametric polymorphism in C++*, In First Workshop on C++ Template Metaprogramming, October 2000

[14] Á. Sinkovics, Z. 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

[15] Á. Sipos, Z. Porkoláb, V. 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.

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

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

[18] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.

[19] T. Veldhuizen, *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[20] T. Veldhuizen, *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[21] T. Veldhuizen, D. Gannon, *Active libraries: Rethinking the roles of compilers and libraries*, In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23

[22] P.Z. Ingerman, *Thunks - A Way of Compling Procedure Statements with Some Comments on Procedure Declarations*, IFIP ALGOL Bulletin & CACM Vol 1 No1. pp. 55.58. Retrieved May 21, 2009.

[23] Todd Veldhuizen, *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

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

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

[26] The boost xpressive regular library.
`http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html`

[27] Bartosz Milewski, *Haskell and C++ template metaprogramming*
`http://bartoszmilewski.wordpress.com/2009/10/26/haskellc-video-and-slides`