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

More and more C++ applications use template metaprograms directly or indirectly by using libraries based on that. Given the complexity of template metaprogramming, developers need supporting libraries. The most widely used one is the Boost template metaprogramming library. It implements commonly used compile time algorithms and meta-data structures 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 directly supporting the functional style. In this paper we evaluate how and in what degree boost::mpl supports functional programming and present new elements it can be improved with.

Keywords: C++, boost::mpl, template metaprogramming, functional programming

1 Introduction

Templates are key elements of 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 [21]. 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 [23]. 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* [24], where runtime computations can be replaced 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. to form constraints on template parameters

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

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

[13,18]; *active libraries* [26], acting dynamically at compile time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages such as the Ararat system [5] for a type-safe SQL interface and `boost:xpressive` [29] for regular expressions.

In the last fifteen years major efforts were put into creating the foundations of template metaprogramming including meta data structures and algorithms. Boost is one of the most important C++ library collections. The libraries are used by 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 [27] 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 reducing the amount of boilerplate code in C++ template metaprograms.

C++ template metaprogramming follows the functional paradigm [14], 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. We have developed a complex template metaprogramming library, that depends heavily on the functional paradigm. We were using the boost metaprogramming library. We have missed a number of tools that are commonly used in functional languages but are not yet available for C++ template metaprogramming. In this paper we evaluate some functional aspects of the boost metaprogramming library and propose new elements providing more direct support of functional programming. We have implemented these new elements and have used in the complex template metaprogramming library we built. [30] [16]

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 and how they could be improved.

A nullary metafunction is a metafunction taking 0 arguments [1], it is the implementation of thunks [8] in C++ template metaprogramming. Unfortunately the value of a nullary metafunction can only be accessed explicitly. When someone accesses the value of a nullary metafunction, he has to access a nested type of the nullary metafunction. This nested type is called `type`. The first time the nested `type` is accessed, the C++ compiler evaluates the metafunction. This value is reused every time the nested `type` is accessed again later, during the same compilation. For example, here is a simple value:

```
struct infinite {};
```

and here is a nullary metafunction:

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

There is no other way to access the value of a nullary metafunction, but by using its nested `type`. An argument of a template metafunction can be a value or a nullary metafunction, but only one of them. Thus, a metafunction can't support lazily and eagerly evaluated arguments at the same time, only one of them.

Classes representing data in template metaprogramming can be designed to be flexible by adding a nested `type` member to them. This nested `type` member evaluates to the class itself. For example:

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

This class can be passed to template metafunctions expecting a nullary metafunction: this class is a data-value and a nullary metafunction at the same time. This class as a nullary metafunction evaluates to itself, thus it represents itself when it is evaluated as a nullary metafunction. Wrappers of static constants, such as `int_`, `bool_` support this: they have a nested type called `type` that is a `typedef` of the wrapper class itself. These classes can be passed to metafunctions expecting eagerly evaluated arguments, because these classes represent data values. These classes can also be passed to metafunctions expecting lazily evaluated arguments, because these classes are nullary metafunctions evaluating to themselves.

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

```
template <class x>
struct identity;
```

is a metafunction taking one argument,

```
identity<infinite>
```

is a nullary metafunction.

Nullary metafunctions can 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. As an example, here are the lazy and eager evaluations of the same function:

```
identity<infinite>        // Lazy evaluation
identity<infinite>::type  // Eager evaluation
```

When we do lazy evaluation, we don't access the metafunction's nested type called `type`. We do it, when we need to force eager evaluation. When we access the nested type called `type`, we force the C++ compiler to evaluate the metafunction.

Let's look at a more complex example.

```
struct infinite {};

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

We use the `infinite` class for representing the infinite value and a `divide` function which divides its two operands. When the second operand is zero, the division is invalid. In this case our function evaluates to `infinite`. This code doesn't work. For example `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` evaluates 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<int_<0>, b>::type,
    identity<infinite>,
    divides<a, b>
  >
{};
```

`infinite` had to be passed to `identity` because `infinite` is a value, not a nullary metafunction. One way of transforming a value into a nullary metafunction is passing it to `identity`. A value can be built in a special way, that it is not only a value, but a nullary metafunction evaluating to itself as well. We could add a nested type called `type` to `infinite`, which is a `typedef` of `infinite` to make it a value and a nullary metafunction:

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

Now both functions expecting a nullary metafunction and functions expecting a value accept it, and it behaves as expected in both situations.

The two ideas can be combined:

```
struct infinite : identity<infinite> {};
```

Now `infinite` is a value and a nullary metafunction evaluating to itself at the same time. We reused the `identity` metafunction in the implementation of `infinite`.

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>
  >
{};
```

We didn't have to pass `infinite` to `identity`, because it's a nullary metafunction evaluating to itself.

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. When the second argument is zero, the division can not be performed, thus we have to handle that case separately. 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.

One possible solution is moving the inner `eval_if` to another template metafunction, such as

```
template <class a, class b>
struct some_calculation_helper :
  eval_if<
    typename less<typename divides<a, b>::type, int_<10> >::type,
    // ...,
    // ...
  >
{};
```

The implementation of `some_calculation` is:

```
template <class a, class b>
struct some_calculation :
  eval_if<
    typename equal_to<b, int_<0> >::type,
    // ....,
    some_calculation_helper<a, b>
  >
{};
```

`some_calculation_helper` is evaluated lazily. When the condition in `some_calculation` evaluates to `false`, `some_calculation_helper` is not instantiated, thus the invalid division is not evaluated and does not break the compilation.

The problem with this solution is that we have to pollute the namespace with new classes and the code of `some_calculation` is defined in multiple metafunctions. A solution is needed, where the nested conditions don't have to be factored out to other metafunctions.

The problem can be solved by evaluating the condition of `eval_if` lazily. 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`, the last example we've seen 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>,
      // ...,
      // ...
    >
  >
{};
```

The nested lazy_eval_if evaluates its condition only when the lazy_eval_if itself is evaluated. When the condition of the outer eval_if evaluates to true, lazy_eval_if it not evaluated, thus its condition is not evaluated either. It guarantees, that when b is 0, the invalid division in the condition of lazy_eval_if is not evaluated and does not break the compilation.

We have considered implementing this solution in a more generic way using a template that makes the arguments of a metafunction lazy. Unfortunately this can not be done. Such a solution could delay the evaluation of the arguments until the metafunction is evaluated. All arguments of it would have to be evaluated at the same time. It means evaluating the condition and the true and false branches of the selection. This would evaluate the unused branch as well and when the evaluation of that branch leads to an error, it would break the compilation.

Vesa Karvonen wrote a fully lazy version of the standard template library as a proof of concept [11]. In his library every template metafunction evaluates its arguments lazily. When someone has to pass a class without a nested type class pointing to itself as argument to a template metafunction, he has to wrap it with a template adding this capability. Only a proof-of-concept implementation of this library is available.

We've seen that in some cases it is essential for template metafunctions to evaluate their arguments lazily. boost::mpl doesn't do it in all cases. We've implemented an addition that covers cases that are not supported by the library in its current form.

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 when the argument is $\pm\frac{\pi}{2}$. Real world examples can be found in [30]. They are more complicated, thus we use this artificial example for the demonstration of the problem.

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 and the enclosing `eval_if`. 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. For example we can't affect third-party libraries. When building template metaprograms, we should be able to use libraries expecting us evaluating our functions in a eager way.

We could factor the code of the branches out to external classes. In this case only the chosen branch is instantiated, thus only that metafunction is evaluated. The other, invalid branch is not instantiated and does not break the compilation.

```
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,
      typename 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 by 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 at one place. This was a simple example. When we have to deal with template metafunction classes [1] instead of template metafunctions, it has a large syntactical overhead. When `square` and `tan` are template metafunction classes, this solution gets more difficult to write, understand and maintain. This is how the `square_tangent` would look like, if `square` and `tan` were template metafunction classes:

```
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 can use the `apply` metafunction of the boost metaprogramming library only in the outermost call of the `false` branch of `eval_if`, otherwise the `square` and `tan` functions are evaluated eagerly. When we're 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 on 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. The current C++ standard hasn't got variadic template [6] support, but there are workarounds we can use with the current compilers [28] and the upcoming standard, C++0x will have variadic template support. 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
        apply<f1, typename apply<f2, a>::type>::type
      type;
  };
};
```

`compose3`, `compose4`, etc. can be implemented similarly, their implementation can be automatically generated using the boost preprocessor metaprogramming library [28]. A `compose` function can be written to call one of the above:

```
struct unused {};

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

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

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

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

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>
  >
{};
```

Here we used `compose` to build the composition of the two metafunctions we have to apply on `deg`. `compose` is a metafunction class we can apply `deg` on to apply the functions on `deg`. It is evaluated lazily, the functions are applied only when the condition of `eval_if` evaluates to `false`, thus the functions are not applied when it would lead to a compilation error.

Compose is easy to use and reduces the syntactical complexity of template metaprograms. It encourages developers to follow the functional paradigm in template metaprogramming.

4 Currying

Currying is supported by several functional languages. When we have a function taking n arguments, we can apply one argument on it and get a function taking $n-1$ arguments. We can continue doing it until n reaches 1. When we have a function taking only 1 argument and we apply one argument on it, we get the value of the function. This is a special form of partial function application which can be simulated using lambda expressions provided by the metaprogramming library of boost. Lambda expressions are difficult to use and have limitations.

Given the functional nature of C++ template metaprograms [14] solutions to problems available in functional languages could be ported to C++ template metaprograms. When we're porting code written in a functional language, keeping the logic the original code follows helps debugging and later improvement of the code. A number of functional libraries make heavy 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 rectangle. 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. It leads to writing a large amount of boilerplate code. We propose a template metafunction taking a template metafunction as argument 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 on the original template metafunction.

We need a function that collects its arguments in a compile time list. The function takes the function to curry and the argument list collected so far as arguments. It has to take the number of arguments left as an argument as well.

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

This template metafunction takes the function to curry as its first argument, `UnpackedMetafunctionClass`, the number of arguments to collect as its second argument, `ArgumentsLeft` and the list collected so far as its third argument, `ArgumentList`. It's important, that the function to curry expects one argument, all arguments of the function in a compile time list. We have to use a helper metafunction class:

```
template <
  class UnpackedMetafunctionClass,
  class ArgumentsLeft,
  class ArgumentList
>
struct nextCurryingStep {
  typedef nextCurryingStep type;

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

It handles one currying step. It implements the metafunction class taking the next currying argument and stores it in the list.

Using these functions we can implement our `curry` functions. These functions take metafunctions as arguments and build a curried version of them. Template metafunctions are template classes, thus the argument of the `curry` functions will be templates. Unfortunately we have to create different `curry` functions to handle template metafunctions taking different number of arguments. We call the `curry` function handling a template metafunction with `n` arguments `curryn`. For example the curry function handling template metafunctions with `4` arguments is called `curry4`. As an example, here is the implementation of it:

```
template <template <class, class, class> class F>
struct curry4 :
  curryImpl<unpack_args<quote4<T> >, int_<4>, deque<> >::type {};
```

We had to use helper functions from `boost::mpl`. We used `quote4` because `curryImpl` expects template metafunction classes while we had a template metafunction, thus we had to generate a metafunction class from it. We used `unpack_args` because `curryImpl` passes the arguments of the metafunction as a compile time list to the metafunction class we call it with.

The rest of the `curry` functions, such as `curry1`, `curry2` are implemented in a similar way. We can use the boost preprocessor library [28] to generate them automatically. The implementation of `curry0` is special, since a metafunction taking 0 arguments evaluates to its value directly. It doesn't take any arguments, thus currying doesn't change anything in this special case. It is supported to make the interface complete:

```
template <class T>
struct curry0 : T {};
```

Using the `curry` functions, the above example can be generated from the simple `area` metafunction we presented at the beginning of this section:

```
curry4<area>
```

When we need currying, `curry` is a tool we can use to avoid writing a large amount of boilerplate code. It makes 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-trivial C++ template metaprograms [25]. He didn't present the functional aspects of template metaprogramming.

Andrei Alexanderscu uses 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 mention functional aspects.

FC++ [12] is a C++ library providing runtime functional programming support for C++. Template metaprograms are always evaluated at compile 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 of functional programming and C++ template metaprogramming in his talk and on his blog [14]. 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 [19] a tool transforming a simple language based on lambda expressions was presented. Lambda expressions form an NP-complete language [9]. The lambda expression based syntax reduced the syntactical overhead of C++ template metaprograms.

In [20] 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.

Vesa Karvonen built a fully lazy version of the boost template metaprogramming library. [11] It was only a proof of concept implementation, the boost library is still not fully lazy, however it would make it usable in many situations.

6 Summary

C++ template metaprogramming can save development and maintenance effort when used appropriately. Given that it's naturally following the functional paradigm [14], we've evaluated how the most widely used C++ template metaprogramming library, `boost::mpl` supports development in a functional style. We've seen that its support for lazy evaluation is good and proposed an addition for further improvement. Explicit support of function composition is missing, we've proposed an implementation of it. We've also presented a way of externally adding currying support to template metafunctions and metafunction classes. The complexity of template metaprograms can be simplified using the tools presented in this paper by eliminating unnecessary helper metafunctions and moving the entire business logic of a metafunction to one location.

We've found that the boost library helps following the functional paradigm. We've presented ways for improving this support further. We've implemented these additions and used in the construction of complex template metaprograms. [30]

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] 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.

[6] 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

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

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

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

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

[11] Vesa Karvonen, Lazy Metaprogramming Library
`http://lists.boost.org/Archives/boost/2004/10/74984.php`

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

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

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

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

[16] Zoltán Porkoláb, Ábel Sinkovics, *Domain-specific Language Integration with Compile-time Parser Generator Library*, In Proceedings of the Generative Programming and Component Engineering, 9th International Conference, GPCE 2010, Eindhoven, The Netherlands, October 10-13, 2010, Accepted for publication.

[17] Zoltán 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.

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

[19] 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

[20] 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.

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

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

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

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

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

[26] 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

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

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

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

[30] The source code of mpllibs
`http://github.com/sabel83/mpllibs`