Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (3) (2011) 27–40

www.elsevier.com/locate/entcs

Nested Lambda Expressions with Let Expressions in C++ Template Metaprograms

Ábel Sinkovics

Department of Programming Languages and Compilers
Eötvös Loránd University
Budapest, Hungary
e-mail: abel@elte.hu

Abstract

More and more C++ applications use template metaprograms directly or indirectly by using libraries based on them. Since C++ template metaprograms follow the functional paradigm, the well known and widely used tools of functional programming should be available for developers of C++ template metaprograms as well. Many functional languages support let expressions to bind expressions to names locally. It simplifies the source code, reduces duplications and avoids the pollution of the namespace or namespaces. In this paper we present how let expressions can be introduced in C++ template metaprograms. We also show how let expressions can be used to implement lambda expressions. The Boost Metaprogramming Library provides lambda expressions for template metaprograms, we show their limitations with nested lambda expressions and how our implementation can handle those cases as well.

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

1 Introduction

Let expressions are common tools in functional programming languages. Their purpose is giving names to expressions in the scope of another expression. For example in Haskell [19] let expressions look like the following:

```
let { d1 ; ... ; dn } in e
```

where `d1`, ..., `dn` are declarations in the scope of the `e` expression.

Templates were designed to capture commonalities of abstractions without performance penalties at runtime, however in 1994 Erwin Unruh showed how they can force any standard C++ compiler to execute specific algorithms as a side effect of the compilation process. This application of templates is called C++ template metaprogramming, which turned out to form a Turing-complete sub-language of C++. [3]

Template metaprogramming has many application areas today, like implementing *expression templates* [15], *static interface checking* [6,9], *active libraries* [16], or

1571-0661
doi:10.1016/j.entcs.2011.11.036

domain specific language embedding [4,17,8,18].

C++ template metaprograms are functional programs [7,11]. Unfortunately they have a complex syntax leading to programs that are difficult to write and understand. Template metaprograms consist of *template metafunctions* [1]. There may be sub expressions that are used multiple times in the body of a metafunction, which has to be copied leading to maintenance issues or moved to a new metafunction leading to namespace pollution. The ability to bind expressions to names locally in metafunctions could simplify both the development and maintenance of C++ template metaprograms.

Many programming languages provide tools to create no-name function objects inside an expression. These tools are called lambda expressions and by using them programmers don't have to create small utility functions when they need function objects with simple implementations. There is no lambda expression support in the current C++ standard, however there are workarounds implemented as a library. The Phoenix and Lambda libraries of Boost provide tools to build lambda expressions [17]. The upcoming standard, C++0x [13] has language support for lambda expressions. Without lambda expressions, the business logic is scattered across lots of small utility functions making the code difficult to understand and change.

The Boost Metaprogramming Library [17] provides tools to build lambda expressions for algorithms executed at compilation time. Arguments of the lambda expressions are called `_1`, `_2`, etc. This causes issues when programmers have to create nested lambda expressions inside other lambda expressions. The solution we present for let expressions can be used to implement lambda expressions in C++ template metaprograms that can express nested lambda expressions correctly. A library implemented based on the ideas presented here is available at [18].

The rest of the paper is organised as the following. In section 2 we detail the concept of let expressions. In section 3 we present our approach to add let expressions to a functional language, that has no built-in support for it. In section 4 we present how our approach can be used to implement nested lambda expressions. In section 5 we extend our approach to support recursive let expressions. We present future works in section 6 and we summarise our results in section 7.

2 Let expressions

Let expressions in Haskell bind declarations to names. A declaration can use pattern matching to bind values to names, for example:

```
let
  a = f 11
  (b, c) = returnTuple 13
in
  a + b + c
```

The above example binds the expression `f 11` to `a`. It evaluates `returnTuple 13` and tries matching the result of it to the pattern `(b, c)`. When it doesn't match,

it generates an error.

The declarations can be recursive, thus we can define recursive functions in a `let` block. For example:

```
let
  fact = \n -> if n == 0 then 1 else (n * (fact (n - 1)))
in
  fact 3
```

The above example defines a factorial function in the scope of the expression `fact 3`. The definition of `fact` refers to itself. There are languages not allowing recursion in `let` expressions. Those languages provide a `letrec` construct for recursion.

Common Lisp [12] supports let expressions as well. They look like the following:

```
(let ((<name 1> <body 1>) ... (<name n> <body n>))
  <body>)
```

It creates local variables called `<name 1>`, ..., `<name n>` with the values `<body 1>`, ..., `<body n>` in `<body>`.

Our implementation follows the semantics of the following construct:

```
let <a> = <e1> in <e2>
```

We bind `<e1>` to `<a>` in the scope of `<e2>`. We don't support pattern matching and we expect `<e1>` not to reference `<a>`. We provide another construct for recursive let expressions:

```
letrec <a> = <e1> in <e2>
```

`letrec` supports recursive name bindings, thus `<e1>` is bound to `<a>` in the scope of `<e1>` itself as well.

We evaluate `e1`, the expression we bind to the name lazily. It is evaluated the first time it is used. When it is not used, it isn't evaluated, thus when its evaluation leads to an error but it's not used, it doesn't cause errors.

3 Implementation of let

Most functional languages supporting let expressions have built-in support for them in the language. These constructs are part of the language and the compiler has to deal with them. Template metaprogramming was not a design goal of C++ and has no compiler support for constructs like let expressions. In this section we present how let expressions can be added using only standard C++ features.

The let expressions we have defined can be implemented as lambda expressions [5]. `let a = e1 in e2` is equivalent to `(\a -> e2) e1`. The Boost Metaprogramming Library provides tools for building lambda expressions, but the names of the arguments are fixed, they have to be `_1`, `_2`, etc. Thus by using lambda expressions we could only bind to these names. Let expressions have to be able to bind to arbitrary names, thus we can not use lambda expressions provided by the Boost library.

A *nullary metafunction* [1] is a metafunction that can be evaluated without additional arguments. We can look at a metafunction as a piece of executable code. A nullary metafunction can be created from any metafunction by passing it the required number of arguments, but not evaluating it. For example:

```
// Metafunction taking two arguments
template <class a, class b>
struct f : boost::mpl::plus_<a, b> {};

// Nullary metafunction
typedef f<boost::mpl::int_<0>, boost::mpl::int_<1> >
  nullary_metafunction;

// Evaluated nullary metafunction
typedef
  nullary_metafunction::type evaluated_nullary_metafunction;
```

The above code demonstrates that passing arguments to a metafunction and evaluating it are two separate steps.

Using let expressions in template metaprograms, we should be able to express something like this:

```
template <class n>
struct my_metafunction:
  LET
    x = boost::mpl::plus<n, boost::mpl::int_<13> >
  IN
    boost::mpl::times<x, x>
{};
```

In the above example we bind `boost::mpl::plus<n,boost::mpl::int_<13>>` to the name `x` in the scope of `boost::mpl::times<x, x>`.

Both the expression we bind and the expression we do the binding in the scope of are C++ classes, thus they can be arguments of template metafunctions. We can declare an empty class, `x` to represent the name `x`, so it can be an argument of a metafunction as well:

```
struct x;
```

Now all elements in our example are C++ classes, we can create a template metafunction implementing the semantics of the let expression. This metafunction could be used the following way:

```
template <class n>
struct my_metafunction :
  let<
    x, boost::mpl::plus<n, boost::mpl::int_<13> >,
    boost::mpl::times<x, x>
  >::type {};
```

The signature of `let` is the following:

```
template <class a, class e1, class e2>
struct let;
```

This metafunction takes

- `a`, the name to bind to
- `e1`, the expression to bind
- `e2`, the expression to bind in the scope of

as arguments. It binds `e1` to `a` in the scope of `e2`. Since we can't override the binding rules of C++, `let` has to change the code of `e2` and replace every occurrence of `a` with `e1`. The result of this is a new nullary metafunction, which behaves as if it was the original piece of code with the new binding rule.

`let` is a template metafunction, its result is the substituted expression. For example

```
let<x, boost::mpl::int_<13>, boost::mpl::plus<x, x> >
```

is a nullary metafunction, however

```
let<x, boost::mpl::int_<13>, boost::mpl::plus<x, x> >::type
```

is `boost::mpl::plus<boost::mpl::int_<13>, boost::mpl::int_<13> >`.

To implement `let` we need a helper metafunction, `let_impl` with the same signature:

```
template <class a, class e1, class e2>
struct let_impl;
```

A precondition of this metafunction is that `e2` is never the same as `a`, thus `let_impl` is never instantiated with the name to replace as its argument.

`let_impl` can be implemented using pattern matching. Its general version covers the case, when there is nothing to substitute in `e2`:

```
template <class a, class e1, class e2>
struct let_impl : id<e2> {};
```

Note that we use a helper metafunction, `id` here. It implements the identity metafunction:

```
template <class x>
struct id { typedef x type; };
```

Because of using `id`, the substituted expression is `let<....>::type`. This type is a `typedef`, thus only a new name of the substituted expression. By not using `id`, users of `let` wouldn't have to access the nested type, `type`. The implementation of the general case of `let_impl` would be

```
template <class a, class e1, class e2>
struct let_impl : e2 {};
```

The problem with this is that `let<....>` wouldn't be a `typedef` of the substituted

expression. It would be a subclass of it. When a template is specialised for a type, it is not specialised for that type's subclasses. [14] For example:

```
struct B {};
struct D : B {};

template <class T>
struct TemplateClass {};

template <>
struct TemplateClass<B> {};

TemplateClass<B> usesSpecialisation;
TemplateClass<D> usesGeneralCase;
```

When we specialise the class template for `B` as its argument type, the specialisation is used when the template class is instantiated with `B` as its argument. But when it's instantiated with `D`, a subclass of `B`, the general case is used. Since pattern matching in template metaprograms is implemented using specialisations [7], metafunctions should not return subclasses of the result instead of the result itself. This is the reason, why using `id` in the implementation of `let` is important.

We have presented the general case of `let_impl`, which covers almost all cases. The only structures it doesn't cover are instances of template classes. They implement metafunction calls. The arguments the templates are instantiated with are the arguments passed to the metafunctions. They can be covered by using template template arguments [14]. For example instances of template classes taking one argument can be covered by the following specialisation of `let_impl`:

```
template <class a, class e1, template<class> class t, class a1>
struct let_impl<a, e1, t<a1> > :
  id< t<typename let<a, e1, a1>::type> > {};
```

We apply `let` recursively on the template arguments. Recursive processing ensures that name binding is simulated for metafunction arguments as well.

Instances of template classes taking two, three, etc arguments can be covered with similar specialisations. These cases can be automatically generated using the Boost Preprocessor Library [17], we don't present it here due to space issues, but it is available in our implementation [18]. We leave implementing it using *variadic templates* instead of the preprocessor library as a future work.

We're targeting template metaprograms with `let`, thus we can assume that arguments of template classes are always classes. Template classes with integral arguments are expected to implement boxing only, `let` has to deal with the boxed value only, which is covered by the general case of `let_impl`.

Now that we have finished `let_impl`, we can implement `let` as well. We can use pattern matching. The general case is the following:

```
template <class a, class e1, class e2>
```

```
struct let : let_impl<a, e1, e2> {};
```

Note that we forward the call to the helper metafunction we have just implemented. It covers every case, except the one when `e2` is the same as `a`. We can cover it with a specialisation of `let`:

```
template <class a, class e1>
struct let<a, e1, a> : id<e1> {};
```

This case does the replacement of the name the `let` expression binds to with the expression bounded to the name.

Haskell supports shadowing a name defined in a let expression. The value of the following expression is 35.

```
let
  x = 11
in
  x + (let x = 13 in x + 11)
```

The outer let expression binds `11` to `x`, but the internal let expression shadows `x`, and in the body of the internal let expression `x` is bound to `13`.

We should deal with these situations in C++ template metaprograms as well. The above example would look like the following in C++ template metaprograms:

```
let<
  x, boost::mpl::int_<11>,
  boost::mpl::plus<
    x,
    let<
      x, boost::mpl::int_<13>,
      boost::mpl::plus<x, boost::mpl::int_<11> >
    >
  >
>
```

The `let` function we have defined so far would substitute `x` with `11` in the entire expression, including the inner `let`. It would evaluate the following:

```
  boost::mpl::plus<
    boost::mpl::int_<11>,
    let<
      boost::mpl::int_<11>, boost::mpl::int_<13>,
      boost::mpl::plus<
        boost::mpl::int_<11>,
        boost::mpl::int_<11>
      >
    >
  >
```

The inner `let` would then replace `11` with `13`, which is not what we would expect

from the original expression. The implementation of `let` has to stop replacing the bound name when it reaches another `let` binding the same name. It can be implemented by adding the following specialisation of `let_impl`:

```
template <class a, class e1, class e1a, class e2>
struct let_impl<a, e1, let<a, e1a, e2> >: id<let<a, e1a, e2> > {};
```

Due to the instantiation rules of template classes [2] when a nested `let` binds to the same name, the compiler instantiates this specialisation and stops replacing the bound name with the value defined by the outer `let`. It leaves the inner `let` unevaluated to support lazy evaluation.

Using `let_impl` and not adding every specialisation to `let` is important. By not using `let_impl` and covering all cases as specialisations of `let` some situations become ambiguous. There are no restrictions on the name to bind to, it can be any class. Thus, it can be an instance of a template class as well. Using instances of templates classes as the name would lead to ambiguous situations: the compiler wouldn't be able to decide whether its the name we bind to or an instance of a template class, that has to be processed recursively. By handling them in two separate metafunctions, these two patterns don't conflict.

The specialisations we have presented cover all cases, the implementation of `let` is now complete. It simulates the name binding by replacing all occurrences of the name to bind with the value to bind to.

4 Nested Lambda expressions

Regardless of the programming language we use, the more we use generic functions in development, the more we need small utility functors implementing custom logic for the generic algorithms. Consider `std::transform`, or its metaprogramming equivalent, `boost::mpl::transform`. Both functions change each element of a sequence. They take a functor changing one element as an argument and apply it on all elements of the sequence. We can use this and many other similar functions in many places, but we need to provide small utility functions implementing the custom bits, such as the transformation of one element of a sequence. These small functions contain bits of the business logic of the application. By implementing them as utility functions we move bits of the business logic to different locations of the source code.

Lambda functions provide a solution to this issue. It is a technique for implementing small utility functors in-place in the middle of a piece of code. These functions have no names unless we store them in variables. This solution makes implementing the functors for generic algorithms in-place possible.

As developers use generic functions in complex situations, they need to construct complex functors as lambda expressions. Complex lambda expressions may contain nested lambda expressions. Consider that we have the following data structure:

```
typedef
  boost::mpl::list<
```

```
    boost::mpl::list_c<int, 1, 2>,
    boost::mpl::list_c<int, 3>
  >
  list_in_list;
```

We'd like to use `boost::mpl::transform` to double every element of the nested lists. We can implement it using `boost::mpl::lambda` the following way:

```
boost::mpl::transform<
  list_in_list,
  boost::mpl::lambda<
    boost::mpl::transform<
      boost::mpl::_1,
      boost::mpl::lambda<
        boost::mpl::times<boost::mpl::_1, boost::mpl::int_<2> >
      >::type
  > >::type
>::type
```

We need to use `boost::mpl::_1` in the outer and in the inner lambda expression as well. Implementing it was possible, but it makes understanding the code more difficult, since the occurrences of `boost::mpl::_1` refer to different things.

Consider another example: use `transform` to add to every element the length of the list it is in. The expected result is:

```
typedef boost::mpl::list<
    boost::mpl::list_c<int, 3, 4>, // length is 2
    boost::mpl::list_c<int, 4> // length is 1
  >
  result_of_second_example;
```

In this case, we need to use the argument of the outer lambda expression in the inner one. If we use the lambda expressions provided by the Boost Metaprogramming Library, we can do it by using workarounds only. The inner lambda expression has to take two arguments: the value of the outer expression's argument and the *real* argument of the inner expression. We need to use some *currying* solution [10] to hide the first argument and make it work with the generic algorithm, `transform`. Given the complexity of this solution, developers are likely to create small helper functions instead, which suffers from the issue of having the business logic at different locations of the source code.

The approach we have presented in this paper for let bindings can be used to implement lambda expressions in C++ template metaprogramming. The following metafunction class implements lambda expressions with one argument using `let`:

```
template <class arg, class f>
struct lambda
{
```

```
  template <class t>
  struct apply : let<arg, t, f>::type {};
};
```

It is a metafunction class [1] taking the name of the argument and the body of the lambda expression as arguments. When an argument is applied on the metafunction class, it uses `let` to bind the argument to the name in the body. We leave the extension of it to support more than one argument as a future work. Using `lambda` the above example can be implemented the following way:

```
boost::mpl::transform<
  list_in_list,
  lambda<nested_list,
    boost::mpl::transform<nested_list,
      lambda<i,
        boost::mpl::plus<i,boost::mpl::size<nested_list> > > > > >
>::type
```

This solution makes use of the fact that name binding takes effect in the entire body of the lambda expression, including inner lambda expressions as well. We can use arguments of the external lambda expression in the internal ones, thus solving the second example is a trivial task using our lambda utility.

We can give meaningful names to the lambda arguments and we can use different names at different levels of nesting. The binding mechanism we have presented binds the values to the names inside the nested lambda expressions as well, thus using the argument of the outer lambda expression in the inner one is simple.

5 Recursive let expressions

So far we have presented how non-recursive let expressions and advanced lambda expressions can be implemented in C++ template metaprogramming. In this section we present how recursive let expressions can be supported.

Consider the following example, which bounds a higher order function implementing factorial to the name `fact` in the expression `boost::mpl::apply<fact, boost::mpl::int_<3> >`:

```
let<fact,
  lambda<n,
    boost::mpl::eval_if<
      boost::mpl::equal_to<n, boost::mpl::int_<0> >,
      boost::mpl::int_<1>,
      boost::mpl::times<
        boost::mpl::apply<fact,
          boost::mpl::minus<n, boost::mpl::int_<1> > >,
        n > > >,
  boost::mpl::apply<fact, boost::mpl::int_<3> > >
```

The above code doesn't work, because the factorial function is implemented recursively, but `let` expects the expression to bind not to rely on recursion. `let` doesn't bind the name `fact` in factorial's implementation to the name `fact`, thus recursive calls will not work.

Let expressions in Haskell support recursion. Languages, whose let expressions don't support recursion usually provide a recursive version of let expressions, which are typically called `letrec`. We follow this approach and implement the recursive version separately, using the non-recursive version in its implementation. We call the recursive version of `let` `letrec`. The declaration of `letrec` is the following:

```
template <class a, class e1, class e2>
struct letrec;
```

We need to bind `e1` to `a` in the scope of `e1` and bind the expression we get to `a` in the scope of `e2`. The following code does this:

```
template <class a, class e1, class e2>
struct letrec : let<a, let<a, e1, e1>, e2> {};
```

The problem with this implementation is that the expression `e1` we bind to `a` in the scope of `e1` – in the inner `let` binding – may use `a` as well. Thus, we need to bind `e1` to `a` recursively in the scope of `e1` as well.

```
template <class a, class e1, class e2>
struct letrec : let<a, letrec<a, e1, e1>, e2> {};
```

The above code uses `letrec` recursively to implement recursive binding of the expression `e1` to the name `a` in the expression `e1` itself. Since binding happens lazily, recursion happens only when `e1` uses the name `a`. When `e1` does not use the name `a`, `letrec` is not evaluated again and the recursion stops. We get an expression as the result of this recursive binding. This is the expression we bind to the name `a` in the scope of `e2`. Since we don't bind `e2` to any name, we don't need recursive binding at this step, thus we can use `let`.

Using `letrec` we can implement the factorial example:

```
letrec<fact,
  lambda<n,
    boost::mpl::eval_if<
      boost::mpl::equal_to<n, boost::mpl::int_<0> >,
      boost::mpl::int_<1>,
      boost::mpl::times<
        boost::mpl::apply<
          fact,
          boost::mpl::minus<n, boost::mpl::int_<1> >
        >, n > > >,
  boost::mpl::apply<fact, boost::mpl::int_<3> > >
```

`letrec` binds our recursive implementation of the factorial function to the name `fact` in the scope of the function as well, thus it works. However, we rely heavily on

lazy evaluation, thus we need to make sure that all functions we use in the body of our factorial function evaluates its arguments lazily. The functions the boost library provides expect eagerly evaluated arguments [10], thus we need to wrap them with code that lazily evaluate the arguments and then call the function coming from the Boost library.

```
template <class f, class x>
struct lazy_apply :
  boost::mpl::apply<typename f::type, x> {};

template <class a, class b>
struct lazy_equal_to :
  boost::mpl::equal_to<typename a::type, typename b::type> {};

template <class a, class b>
struct lazy_times :
  boost::mpl::times<typename a::type, typename b::type> {};

let<
  fact,
  lambda<
    n,
    lazy_eval_if<
      lazy_equal_to<n, boost::mpl::int_<0> >,
      boost::mpl::int_<1>,
      lazy_times<
        lazy_apply<
          fact,
          boost::mpl::minus<n, boost::mpl::int_<1> >
        >,
        n >
    >
  >,
  lazy_apply<fact, boost::mpl::int_<3> >
>
```

This is finally a working version of our factorial example, that uses only lazy metafunctions. We leave the creation of a general wrapper making all functions of the Boost library lazy as a future work.

6 Future works

We've seen that this solution supports binding a value to a name in a template metaprogram. The name so far has been a name literal, which we introduced in the system by creating an empty class with that name. Due to the fact that this solution happens at metaprogramming execution time and uses pattern matching

to implement binding, we're not limited to bind to names only. We can bind to any class or instance of a template class, including nullary template metafunctions. Thus, we can bind new code to other pieces of code and replace or modify bits of existing metafunctions. This technique makes it possible to support aspect oriented programming in C++ template metaprogramming without using external weaving tool. We leave it as a future work.

7 Summary

Let expressions are one of the basic building blocks of many functional programming languages, like Haskell, Lisp, etc. We have shown that support for expressions can be added to C++ template metaprogramming, a functional language without built-in support for this. We have also shown how let expressions can be used to add support for lambda expressions. We have built a library supporting lambda expressions and let expressions for C++ template metaprogramming [18]. Using our `let` solution a fundamental improvement can be achieved for the Boost Metaprogramming Library. We have shown that lambda expressions built following the approach presented in this paper can deal with nested lambda expressions as well.

References

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

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

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

[4] 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.

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

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

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

[8] 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.

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

[10] Ábel Sinkovics, Functional extensions to the Boost Metaprogram Library, In Porkolab, Pataki (Eds) Proceedings of the 2nd Workshop of Generative Technologies, WGT'10, Paphos, Cyprus. pp.56–66 (2010), ISBN: 978-963-284-140-3

[11] 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

[12] Guy L. Steele, *Common Lisp the Language*, 2nd edition Digital Press, 1990. ISBN: 1-55558-041-6 Pbk

[13] B. Stroustrup, Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007

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

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

[16] 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

[17] The Boost programming libraries
www.boost.org

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

[19] Haskell 98 Language and Libraries. The Revised Report.
http://www.haskell.org/onlinereport