Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 238 (2009) 47–58

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

EClean - An Embedded Functional Language

Ádám Sipos[1] and Viktória Zsók[2]

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

Abstract

C++ template metaprogramming is often regarded as a functional language, however, nowadays metaprogram libraries are not implemented in functional programming style. In this paper we discuss a compile-time graph-rewriting engine based on the properties of the functional language Clean. The most important property imported from the functional paradigm is the lazy evaluation strategy. With the help of the engine it is possible to embed lazy functional-style code into C++ programs, and transform it into template metaprograms. We present the implemented lazy evaluation strategy by examples including also infinite lists.

Keywords: C++, template metaprogramming, language embedding, functional languages

1 Introduction

Template metaprogramming is an emerging new direction in C++ programming for executing algorithms in compilation time. The relationship between C++ template metaprograms and functional programming is well-known: most properties of template metaprograms are closely related to the principles of the functional programming paradigm. On the other hand, C++ has a strong heritage of imperative programming (namely from C and Algol68) influenced by object-orientation (Simula67). Furthermore the syntax of the C++ templates is especially ugly. As a result, C++ template metaprograms are often hard to read, and hopeless to maintain.

Ideally, the programming language interface has to match the paradigm the program is written in. *Meta<Fun>* is a running project at the Department of Programming Languages and Compilers at the Eötvös Loránd University, Budapest. The long-term goal of the project is to define and implement a clear and maintanable, purely functional-style interface for C++ template metaprograms. For this purpose, template metaprograms are written in a functional language and embedded

[1] Email: shp@inf.elte.hu
[2] Email: zsv@inf.elte.hu

doi:10.1016/j.entcs.2009.05.006

in C++ programs. This code is translated into classical template metaprograms by a translator. The result is a native C++ program complies with the ANSI standard [3].

Clean is purely functional lazy programming language [11]. In our approach we explore Clean's main features including uniqueness types, higher order functions, and the powerful constructor-based syntax for generating data structures. Clean also supports infinite data structures via delayed evaluation. We defined *EClean* as a subset of the Clean language. EClean is used as an embedded language for writing template metaprograms. The Clean code will be executed by a graph-rewriting engine. The parser recognizes only a subset of the Clean language, as our aim was to create an embedded language aiding programmers in writing metaprograms, and not the implementation of a fully capable Clean compiler. This subset includes:

- recursion, conditional expression
- function definitions
- basic built-in types (int, bool, etc) and their lists
- arithmetic operations and comparisons

In this article we overview the most important properties of the functional paradigm, and evaluate their possible translation techniques into C++ metaprograms. The graph-rewriting system of Clean has been implemented as a C++ template metaprogram library. With the help of the engine and the corresponding parser, EClean programs can be translated into C++ template metaprograms – as clients of this library – and can be evaluated in a semantically equivalent way. Delayed evaluation of infinite data structures are also implemented and presented by examples.

The paper is organized as follows: in section 2 we discuss C++ template metaprogramming, and its relationship with functional programming. Lazy data structures, evaluation, and the template metaprogram implementation of the graph rewriting system of the Clean functional language is described in section 3. In section 4 the transformation process of the EClean system is discussed in detail, section 5 discusses future work, and related work is presented in section 6.

2 C++ Template Metaprogramming

Templates are key elements of C++ programming language [23]. They enable the implementation of data structures and algorithms that can be parameterized by types, thus capturing commonalities of abstractions in compile time without loosing performance in runtime [27]. *Template metaprogramming* is a generative programming style [6] utilizing the C++ template facilities for executing algorithms in compile-time. Recursions are created by forcing the compiler into executing a chain of instantiations of the same template. On the other hand, compile-time conditional statements can also be created with the help of template specializations. With these two constructs available, TMP is *Turing-complete* [30], in theory its expressive power is equivalent to that of a Turing machine (and of most modern programming languages). The most important applications of metaprograms are the implementation of *concept checking* [33] (testing for certain type-properties of in compile-time), the implementation of data structures containing types in compile-time (e.g. *typelist* [2]), the construction of *active libraries* [7], and many others.

Despite all of its advantages TMP is not yet widely used in the software industry due to the lack of coding standards, and software tools. A common problem with TMP is the tedious syntax, and long code. Libraries like `boost::mpl` help the programmers by hiding implementation details of certain algorithms and containers, but still a big part of coding is left to the user. Due to the lack of a standardized interface for TMP, naming and coding conventions vary from programmer to programmer, which causes comprehensibility problems.

Template metaprograms are many times regarded as a pure functional language programs. The common properties include referential transparency (metaprograms have no side-effects) and the lack of variables, loops, and assignments. One of the most important functional properties of TMP is the immutability of defined entities, i.e. in case we define constants, enumeration values, types their value or meaning can not be changed. Metaprograms does not contain assignments. Instead of them recursions and specializations are used in order to implement loops since the value of any loop variable can not be changed. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur in metaprograms.

The similarities between the two programming paradigms require a more thorough examination, as the metaprogramming realm could benefit from the introduction and library implementation of more functional techniques. In our view two methods are possible for integrating a functional interface into C++: either modifying the compiler to extend the language itself, or creating a library-level solution by using a preprocessor or macros. The first approach is probably quicker, easier, and more flexible, but at the same time a language extension is undesirable in the case of a standardized, widely used language like C++. Our approach is the aforementioned second one. We re-implement the graph-rewriting engine of the Clean language as a compile-time metaprogram library using only ANSI standard compliant C++ language elements. The goal is to implement in TMP the laziness property of functional languages. Our approach has many advantages, like:

- independent translation of the Clean code fragments into C++ template metaprograms by separating the user written embedded code from the graph-rewriting engine.
- easy usage, since the engine follows the graph-rewriting rules of the Clean language as it is defined in [5], the semantic of the translated code is as close to the programmers intension as possible.
- high portability of the introduced library, as our solution uses only standard C++ elements.

The above advantages will be presented the following by detailed examples. First we describe the implemented laziness.

3 Lazy Evaluation Strategy

As lazy evaluation is one of the most characteristic features of the functional language Clean, our research focuses on lazy evaluation and its application in C++ template metaprograms. In order to present the analogies between the lazy evaluation strategies and the behaviour of template metaprograms, we modeled the purely functional lazy language Clean in TMP.

Clean programs are represented by an expression graph in the compiler. This graph is *rewritten* automatically in several phases in runtime. The rewriting process is starting when the main function expression on the right side of the *Start* symbol is evaluated.

One of the basic data structures in Clean is the list. A list in Clean is defined as a linked list [11]. In the following we will describe lists as a head element and the "tail", as they are regarded in functional programs by the pattern matching mechanisms. The constructor handling these two parts of the list is called `Cons`. For example the list `[2,3,4]` is written as `Cons 2 Cons 3 Cons 4 Nil`, with `Nil` representing the end of the list.

The *lazy* evaluation strategy means *"a redex is only evaluated when it is needed to compute the final result"* [17]. This laziness enables us to specify lists that contain an infinite number of elements, e.g. the list of natural numbers: `[1..]`. A classic example for the usage of lazy lists is the *Eratosthenes sieve* algorithm producing the first arbitrarily many primes.

Our running example uses `EnumFrom`. We defined the `EnumFrom` constructor to create an infinite list starting at a certain number. The list `[2..]` can thus be written as `EnumFrom 2` (or `[2..]`). Of course to acquire the head element of a list we need to rewrite this expression to `Cons 2 EnumFrom 3` (or `[2,EnumFrom 3]`), i.e. this is a list containing 2 and `[3..]`.

In the following we present a simple Clean program calculating the first 10 primes. (The symbols `R1`..`R6` are line numberings)

```
(R1)   take 0 xs = []
(R2)   take n [x,xs] = [x, take n-1 xs]
(R3)   sieve [prime:rest] = [prime : sieve (filter prime rest)]
(R4)   filter p [h:tl] | h rem p == 0  = filter p tl
                                       = [h : filter p tl]
(R5)   filter p [] = []
(R6)   Start = take 10 (sieve ([2..]))
```

Clean follows the *left-right outermost* rewriting strategy. The first examined expression is the `Start` expression. Clean first tries to apply one of the rewriting rules to the examined expression by substituting the current parameters with the rule's parameters. If it does not succeed, the arguments of every expression (subexpression) are examined recursively starting from left to right. If any of the subexpressions can be rewritten, the evaluation returns to the outermost expression. The evaluation process terminates, when none of the rules is applicable to any of the subexpressions.

In our example the first examined expression is (F1). Since none of the rules' left sides takes the form of (F1), Clean examines the first argument in (F1). It is `10`, for which we have no rewriting rule, and this expression has no subexpressions either. The other argument `sieve (EnumFrom 2)` cannot be rewritten either. On the other hand the argument `EnumFrom 2` is equivalent to `[2, EnumFrom 3]`, and this is the substitution rule that takes place in this case.

As a rewriting rule has been applied, we return to the outermost expression which is now (F2). Again, neither this whole expression, nor its first argument can be rewritten, thus the second argument `sieve [2, EnumFrom 3]` is examined. The (R3) rule can be applied here with `prime=2` and `rest=EnumFrom 3` respectively.

The outermost expression now takes the form of (F4), which is the expression we return to. Now (R2) can be directly applied with `n=10`, `x=2`, and `xs=sieve (filter 2 EnumFrom 3)`.

```
(F1)   take 10 (sieve [2..])
(F2)   take 10 (sieve [2, [3..]])
(F3)   take 10 ([2, sieve (filter 2 [3..])])
(F4)   [2, take 9 (sieve (filter 2 [3..]))]
(F5)   [2, take 9 (sieve [3, filter 2 [4..])]
(F6)   [2, take 9 [3, sieve (filter 3 (filter 2 [4..]))]]
(F7)   [2, 3, take 8 (sieve (filter 3 (filter 2 [4..])))]
...
```

The above example presents very well the applied lazy evaluation strategy. In order to simulate the inner workings of the lazy functional programming language Clean, the rewriting algorithm was implemented using TMP. The rewriting is effectuated by our engine, described in the next section.

4 The Implementation of the Graph-rewriting Engine

In the following we present via examples the transformation method of an EClean program into C++ templates. Our EClean system consists of two main parts: a *parser* – responsible for transforming EClean code into metaprograms–, and the *engine* – executing of the functional code. In this paper we discuss the implementation of the latter.

The evaluation of the EClean code parts when a mixed C++ program is compiled is done in the following steps (see Figure 1):

- The C++ preprocessor is invoked in the execution of the necessary header file inclusions and macro substitutions. The EClean library containing the engine and supporting metaprograms is also imported at this point, at the beginning.
- The original source code is divided into C++ parts and EClean parts.
- The EClean parts are transformed by the parser of the EClean into C++ metaprogram code snippets.
- This transformed source code is passed to the C++ compiler.
- The C++ compiler invokes the instantiation chain at the code parts where the

`Start` expression is used, thus activating the EClean engine.

- The engine emulates Clean's graph rewriting, and thus executes the EClean program snippets.
- When no further rewriting can be done, the finished expression's value is calculated, by demand.

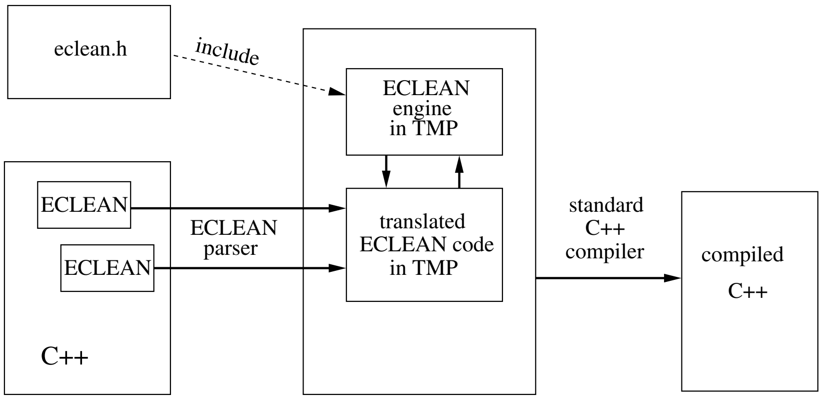

Fig. 1. EClean transformation and compilation process

4.1 The sieve program

In the following we describe the above transformation procedure carried out by the parser to create metaprograms from the original EClean functional code written in section 3.

The EClean expressions are represented by types and `typedefs`. In this approach the Start expression of our example has the form `take<mpl::int_<10>,sieve<EnumFrom<mpl::int_<2> > > >`. Here `take`, `sieve`, and `EnumFrom` are all `struct templates` having the corresponding signatures.

The graph rewriting process can be emulated with the C++ compiler's instantiation process. When a template with certain arguments has to be instantiated, the C++ compiler chooses the narrowest matching template of that name from the specializations. Therefore the rules can be implemented with template partial specializations. Each partial specialization has an inner `typedef` called `right` which represents the right side of a pattern matching rule. At the same time the template's name and parameter list represent the left side of a pattern matching rule, and the compiler will choose the most suitable of the specializations of the same name. Let us consider the following example, which describes the `sieve` rule (`sieve [prime:rest] = [prime :  sieve (filter prime rest)]`).

```
template <class prime, class ys>
struct sieve<Cons<prime,ys> >
{
   typedef Cons<prime,sieve<filter<prime,ys> > > right;
};
```

The `sieve` template has two parameters, `prime` and `ys`. This template describes the workings of (R3) in our Clean example. In case a subexpression has the form `sieve<Cons<N,T> >` where `N` and `T` are arbitrary types, the previously defined `sieve` specialization will be chosen by the compiler as a substitute for the subexpression. Note that even though `N` and `T` are general types, the `sieve` template expects `N` to be a `mpl::int_`, and `T` a list of `mpl::int_` types.

However, in order to be able to apply this rewriting rule, an exact match is needed during the rewriting process. For example in (F1) during the evaluation process the previous `sieve` partial specialization will be considered as applicable when rewriting the subexpression `sieve [2..]`.

The problem is that the argument `[2..]` (`EnumFrom 2`) does not match the `sieve` partial specialization parameter list, which is expecting an expression in the form `Cons<N,T>` with types `N` and `T`. During the compilation the C++ compiler will instantiate the type `sieve<EnumFrom<mpl::int_<2> > >`. However this is a pattern matching failure which has to be detected. Therefore each function must implement a partial specialization for the general case, when none of the rules with the same name can be applied. The symbol `NoMatch` is introduced, which signs that even though this template has been instantiated with some parameter `xs`, there is no applicable rule for this argument. `NoMatch` is a simple empty `class`.

```
template <class xs>
struct sieve
{
    typedef NoMatch right;
};
```

The previously introduced `filter` function's case distinction is used to determine in compile-time whether `x` is divisible by `p`, and depending on that decision either of the two alternatives can be chosen as the substitution. The C++ transformation of `filter` utilizes `mpl::if_` for making a compile-time decision:

```
template <int p, class x, class xs >
struct filter<boost::mpl::int_<p>, Cons<x,xs> >
{
    typedef typename boost::mpl::if_
    <
        typename equal_to
        <
            typename modulus<x,p>::type,
            boost::mpl::int_<0>
        >::type,
        filter<p,xs>,
        Cons<x,filter<p,xs> >
    >::type right;
};
```

The `mpl::if_` construct makes a decision in compile-time. The first type parameter is the `if` condition, which in our case is an `equal_to` template, whose inner `type` typedef is a `mpl::bool_`. Depending on this `bool_`'s `value`, either the first, or the second parameter is chosen.

The working of the transformed `EnumFrom` is similar to the one of Clean: if a rewriting is needed with `EnumFrom`, a new list is created consisting of the list's head number, and an `EnumFrom` withe the next number of the list.

```
template <class r>
struct EnumFrom
{
    typedef Cons<r,EnumFrom<boost::mpl::int_<r::value+1> > > right;
};
```

All other functions of the EClean example of the Section 3 can be translated into templates using analogies with the previous examples.

In the following we present the rewriting engine recognizing EClean expressions, and transforming them into TMP using the previous rules.

4.2 The graph-rewriting engine

Until now we have translated the Clean rewriting rules into C++ templates, by defining their names, parameter lists (the rule's partial specialization), and their right sides. These templates will be used to create types representing expressions thus storing information in compile-time. This is the first abstraction layer. In the following we present the next abstraction level, that uses this stored information. This is done by the library's core, the partial specializations of the `Eval struct template`, which evaluate a given EClean expression.

Since the specialization's parameter is a template in itself (representing an expression), its own parameter list has to be defined too. Because of this constraint, separate implementations are needed for the evaluation of expressions with different arities. In the following we present one version of `Eval` that evaluates expressions with exactly one parameter:

```
template <class T1, template <class> class Expr>
struct Eval<Expr<T1> >
{
    typedef typename
        if_c<is_same<typename Expr<T1>::right,
                     NoMatch>::value,
        typename
            if_c<!Eval<T1>::second,
                Expr<T1>,
                Expr<typename Eval<T1>::result>
            >::type,
            typename Expr<T1>::right
        >::type result;

15      static const bool second =
16          !(is_same<typename Expr<T1>::right,NoMatch>::value &&
17           !Eval<T1>::second);
18 };
```

The working mechanism of `Eval` is as follows: `Eval` takes one argument, an expression `Expr` with one parameter `T1`. The type variable `T1` can be any type, e.g. `int`, a list of integers, or a further subexpression. The return type `result` defined in line 13 contains the newly rewritten subexpression, or the same input expression if no rule can be applied to the expression and its parameters.

When the template `Expr` has no partial specialization for the parameter `T1`, the compiler chooses the general template as described in Section 4.1. The compile-time `if_c` in line 5 is used to determine whether this happened.

- If this is the case, the `Expr<T1>::right` is equal to `NoMatch`. Now another `if_c` is invoked. In line 8 `T1`, the first (and only) argument is evaluated, with a recursive call to `Eval`. The boolean `second` determines whether `T1` or any of its parameters could be rewritten. If no rewriting has been done among these children, `Eval`'s return type will be the original input expression. Otherwise the return type is the input expression with its `T1` argument substituted with `Eval<T1>::result`, which means that either `T1` itself, or one of its parameters has been rewritten. This mechanism is similar to type inference.
- On the other hand, if a match has been found (the `if_c` conditional statement returned with a false value), the whole expression is rewritten, and `Eval` returns with the transformed expression (line 12).

The aforementioned boolean value `second` is defined by each `Eval` specialization (line 15). It is the logical value signaling whether the expression itself, or one of its subexpressions has been rewritten.

The implementation of `Eval` for more parameters is very similar to the previous example, the difference being that these parameters also have to be recursively checked for rewriting.

As our expressions are stored as types, during the transformation process the expression's changes are represented by the introduction of new types. The process of the transformation is the very same as with the Clean example. The following types are created as `right` typedefs:

```
take<10,sieve<EnumFrom<2> > >
take<10,sieve<Cons<2,EnumFrom<3> > > >
take<10,Cons<2,sieve<filter<2,EnumFrom<3> > > > >
Cons<2,take<9,sieve<filter<2>,EnumFrom<3> > > >
Cons<2,take<9,sieve<3,filter<2,EnumFrom<4> > > > >
Cons<2,take<9,Cons<3,sieve<filter<3,EnumFrom<4> > > > > >
Cons<2,3,take<8,filter<3,filter<2,EnumFrom<4> > > > >
...
```

(Note that in the example all `mpl::int_` prefixes are omitted from the `int` values for the sake of readability.) The above steps demonstrates the implementation and the working mechanisms of the evaluation engine. The last step taken by the engine is given in the next subsection.

4.3 Final rewriting of the expression

In the `factorial` example we have seen that the finished expression after the instantiation process will be: `times<mpl::int_<5>, times<mpl::int_<4>, times<mpl::int_<3>, times<mpl::int_<2>, mpl::int_<1> > > > >`. With a simple iteration it is now easy to multiply the values of list member `mpl::int_` types with each other, thus giving the final answer in the form of an `int` value.

5 Future work

One of the most interesting questions in our hybrid approach is to distinguish between problems that can be dealt with by EClean alone (e.g. factorial computation), and those that do require template metaprogramming and compiler support. The EClean parser could choose function calls that can be run separately and their result computed without the transformation procedure and the invocation of the C++ compiler. On the other hand, references to C++ constants and types could be placed within the EClean code, and used by the EClean function in a callback-style. This would result in much greater flexibility and interactivity between EClean and C++.

Our proof-of-concept demo version of the EClean parser handles only the `int` elementary type and lists of integers. Other built-in types can be analogously used. In the future we will include support for more scalar types (`bool`, `long`, etc) besides the implemented `Int`, and the list construct. User defined composed types may require more complex solutions. Another interesting direction is the introduction of special EClean types like `Type` representing a C++ type, `Func` representing a C++ function or even a function pointer.

The parser will be extended with many error detections and recovery features, which can be easily implemented. The helpful error messages will aid the programmer in using EClean and writing metaprograms.

The priority of EClean rewriting rules cannot be explicitly defined as opposed to Clean. In the EClean code if `factorial 1 = 1` alternative is defined under the general `factorial n = n*factorial (n-1)`, then the specialized alternative will never match. However, in template form, due to the nature of the template instantiation mechanism, the `Factorial<mpl::int_<1> >` alternative will match even if defined below the general `Factorial`. At this moment neither the parser nor the engine are capable of handling multiple matching. However, it is an important and interesting future development direction.

6 Related Work

Functional language-like behavior in C++ has already been studied. *Functional C++* (FC++) [12] is a library introducing functional programming tools to C++, including currying, higher-order functions, and lazy data types. FC++, however, is a runtime library, and our aim was to utilize functional programming techniques in compile-time.

The `boost::mpl` library is a mature library for C++ template metaprogramming. `Boost::mpl` contains a number of compile-time data structures, algorithms, and functional-style features, like *Partial Metafunction Application* and *Higher-order metafunctions.* However, `boost::mpl` were designed mainly to follow the interface of the C++ Standard Template Library. There is no explicit support for lazy infinite data structures either.

7 Conclusion

In this paper we discussed the Meta<Fun> project which enhances the syntactical expressivity of C++ template metaprograms. EClean, a subset of the general-purpose functional programming language Clean is introduced as an embedded language to write metaprogram code in a C++ host environment. The graph-rewriting system of the Clean language has been implemented as a template metaprogram library. Functional code fragments are translated into classical C++ template metaprograms with the help of a parser. The rewritten metaprogram fragments are passed to the rewriting library. Lazy evaluation of infinite data structures is implemented to demonstrate the feasibility of the approach. Since the graph-rewriting library uses only standard C++ language features, our solution requires no language extension and it is highly portable.

References

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

[2] Alexandrescu, A., "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] Bracha, G., M. Odersky, D. Stoutamire, P. Wadler, *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language*, Proc. of ACM Symposium on Object Oriented Programming: Systems, Languages, and Applications (OOPSLA), pp.183-200, 1998

[5] Brus, T. H., C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer, *CLEAN: A language for functional graph rewriting.*
Proc. of a conference on Functional programming languages and computer architecture, pp.364-384, Springer-Verlag, 1987

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

[7] Czarnecki, K., U. W. Eisenecker, R. Glück, D. Vandevoorde, T. L. Veldhuizen, *Generative Programming and Active Libraries*, Lecture Notes In Computer Science Vol. **1766** (1998), Selected Papers from the International Seminar on Generic Programming, pp.25-39, 1998

[8] Eisenecker, U. W., F. Blinn, K. Czarnecki, *A Solution to the Constructor-Problem of Mixin-Based Programming in C++*, First C++ Template Programming Workshop, 2000

[9] Garcia, R., J. Järvi, A. Lumsdaine, J. Siek, J. Willcock, *A Comparative Study of Language Support for Generic Programming* Proc. of the 18th ACM SIGPLAN OOPSLA, pp. 115-134, 2003

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

[11] Koopman, P., R. Plasmeijer, M. van Eeekelen, S. Smetsers, "Functional programming in Clean", 2002

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

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

[14] Musser, D. R., A. A. Stepanov, "The Ada Generic Library: Linear List Processing Packages", Springer Verlag, 1989

[15] McNamara, B., Y. Smaragdakis, *Static interfaces in C++*, First C++ Template Programming Workshop, 2000

[16] Odersky, M., P. Wadler, *Pizza into Java: Translating theory into practice*, Proc. of Symposium on Principles of Programming Languages, pp.146-159, 1997

[17] Plasmeijer, R., M. van Eeekelen, "Clean Language Report", 2001

[18] Siek, J., "A Language for Generic Programming", PhD thesis, Indiana University, 2005

[19] Siek, J., A. Lumsdaine, *Concept checking: Binding parametric polymorphism in C++*, First C++ Template Programming Workshop, 2000

[20] Siek, J., A. Lumsdaine, *Essential Language Support for Generic Programming*, Proc. of the ACM SIGPLAN 2005 conference on Programming language design and implementation, pp.73-84, 2005

[21] Sipos, Á., "Effective Metaprogramming", M.Sc. Thesis, Eötvös Loránd University, 2006

[22] Sipos, Á., N. Pataki, Z. Porkoláb, V. Zsók, *Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms*, Proc. of Implementation of Functional Languages, pp.489-502, 2007

[23] Stroustrup, B., "The C++ Programming Language Special Edition", Addison-Wesley, 2000

[24] Stroustrup, B., "The Design and Evolution of C++", Addison-Wesley, 1994

[25] Reis, G. D., B. Stroustrup, *Specifying C++ concepts*, Proc. of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), pp.295-308, 2006

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

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

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

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

[30] Veldhuizen, T., *C++ Templates are Turing Complete*

[31] Zólyomi, I., Z. Porkoláb, *Towards a template introspection library*, Lecture Notes on Computer Science **3286** (2004), pp.266-282, 2004

[32] Zólyomi, I., Z. Porkoláb, T. Kozsik, *An extension to the subtype relationship in C++*, GPCE 2003, Lecture Notes on Computer Science **2830** (2003), pp.209-227, 2003

[33] Boost Concept checking
http://www.boost.org/libs/concept_check/concept_check.htm

[34] Boost Metaprogramming library
http://www.boost.org/libs/mpl/doc/index.html

[35] Boost Preprocessor library
http://www.boost.org/libs/preprocessor/doc/index.html

[36] Boost Static assertion
http://www.boost.org/regression-logs/cs-win32_metacomm/doc/html/boost_staticassert.html