STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

META<FUN> – TOWARDS A FUNCTIONAL-STYLE INTERFACE FOR C++ TEMPLATE METAPROGRAMS

ÁDÁM SIPOS, ZOLTÁN PORKOLÁB, AND VIKTÓRIA ZSÓK

Abstract. Template metaprogramming is an emerging new direction in C++ programming for executing algorithms at compilation time. Despite that template metaprogramming has a strong relationship with functional programming, existing template metaprogram libraries do not follow the requirements of the functional paradigm. In this paper we discuss the possibility to enhance the syntactical expressivity of template metaprograms using an embedded functional language. For this purpose we define *EClean*, a subset of Clean, a purely functional lazy programming language. A parser, and a graph-rewriting engine for EClean have been implemented. The engine itself is a compile-time template metaprogram library using standard C++ language features. To demonstrate the feasibility of the approach lazy evaluation of infinite data structures is implemented.

1. Introduction

Template metaprogramming is an emerging new direction in C++ programming for executing algorithms at 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. The subject of the *Meta<Fun>* project is writing template metaprograms in a functional language and embedding them into

Received by the editors: September 15, 2008.

1998 *CR Categories and Descriptors.* D.3 [**Programming Languages**]: D.3.2 Applicative (functional) languages; D.3 [**Programming Languages**]: D.3.2 Language Classification, C++.

Key words and phrases. C++ Template Metaprogramming, Clean.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

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

Clean is a general purpose, purely functional, lazy language [8]. 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 expressing template metaprogramming.

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, 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 rest of the paper is organized as follows: In section 2 we give a technical overview of C++ template metaprograms (TMP), and discuss the relationship between TMP and functional programming. Lazy data structures, evaluation, and the template metaprogram implementation of the graph rewriting system of Clean is described in section 3. Section 4 discusses future work, and related work is presented in section 5. The paper is concluded in section 6.

2. Metaprogramming and Functional Programming

Templates are key elements of C++ programming language [13]. They enable data structures and algorithms be parameterized by types thus capturing commonalities of abstractions at compilation time without performance penalties at runtime [17]. *Generic programming* [12, 11, 9], is a recently emerged programming paradigm, which enables the developer to implement reusable codes easily. Reusable components – in most cases data structures and algorithms – are implemented in C++ with the heavy use of templates.

In C++, in order to use a template with some specific type, an *instantiation* is required. This process can be initiated either implicitly by the compiler when a template with a new type argument is referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and the generated new code is compiled. The instantiation mechanism enables us to write smart template codes that execute algorithms at compilation time [16, 18]. This paradigm, *Template Metaprogramming* (TMP) is used for numerous purposes. These include transferring calculations to compile-time, thus speeding up the execution of the program; implementing *concept checking* [22, 14, 21] (testing for certain properties of types at compilation); implementing *active libraries* [5], and others.

Conditional statements, like the stopping of recursions, are implemented with the help of specializations. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [6]. Data and computation results are expressed at runtime programs as constant values or literals. In metaprograms we use `static const` and enumeration values to store quantitative information.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [19]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [22, 7, 1].

By enabling the compile-time code adaptation, C++ template metaprograms (TMP) is a style within the *generative programming* paradigm [6]. Template metaprogramming is *Turing-complete* [20], in theory its expressive power is equivalent to that of a Turing machine (and thus most programming languages).

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.

Template metaprogramming is many times regarded as a pure functional programming style. The common properties of metaprogramming and functional languages include referential transparency and the lack of variables, loops, and assignments. One of the most important functional properties of TMP is that if a certain entity (the aforementioned constants, enumeration values, types) has been defined, it will be immutable. A metaprogram does not contain assignments. That is the reason why we use recursion and specialization to implement loops: we are not able to change the value of any loop variable. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur.

In our opinion, the similarities require a more thorough examination, as the metaprogramming realm could benefit from the introduction and library implementation of more functional techniques.

Two methods are possible for integrating a functional interface into C++: modifying the compiler to extend the language itself, or creating a library-level solution and 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 to re-implement the graph-rewriting engine of the Clean language as a compile-time metaprogram library using only ANSI standard compliant C++ language elements. Thus our solution has numerous advantages. As the user written embedded code is separated from the graph-rewriting engine, the embedded Clean code fragments can be translated into C++ template metaprograms independently. Since the engine follows the graph-rewriting rules of the Clean language as it is defined in [4], the semantics of the translated code is as close to the intentions of the programmer as possible. As our solution uses only standard C++ elements, the library is highly portable.

3. Lazy Evaluation and Implementation of the Graph-rewriting Engine

As lazy evaluation is one of the most characteristic features of the Clean language [10], our research centers around lazy evaluation and its application in C++ template metaprograms. A *lazy* evaluation strategy means that *"a redex is only evaluated when it is needed to compute the final result"*. This enables us to specify lists that contain an infinite number of elements, e.g. the list of natural numbers: `[1..]`. Our running example for the usage of lazy lists is the *Eratosthenes sieve* algorithm producing the first arbitrarily many 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..]))
```

The Clean graph rewriting engine carries out the following evaluation.

```
(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..])))]
```

. . .

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: the *parser* – responsible for transforming EClean code into metaprograms–, and the *engine* – doing the actual execution of the functional code.
The compilation of a C++ program using EClean code parts is done in the following steps:

- 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.
- The source code is divided into C++ parts and EClean parts.
- The EClean parts are transformed by the parser of EClean into C++ metaprogram code snippets.
- This transformed source code is handed to the C++ compiler.
- The C++ compiler invokes the instantiation chain at 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.
- When no further rewriting can be done, the finished expression's value is calculated, if necessary.

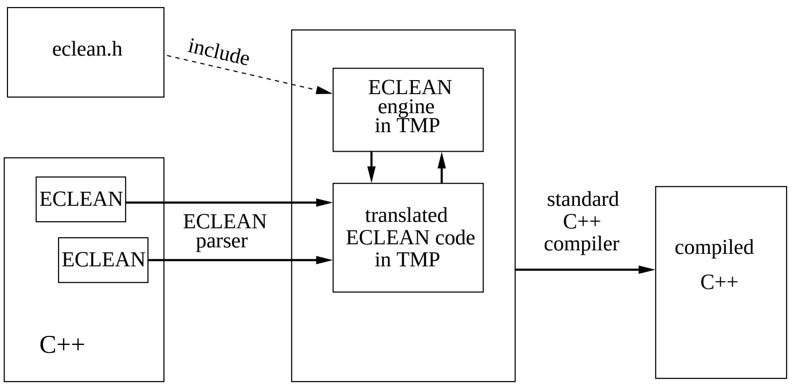

FIGURE 1. EClean transformation and compilation process

3.1. **The sieve program.** In Section 2 we have described the various language constructs available in metaprogramming. We now use `typedefs`, and types created from templates to represent the EClean expressions. In this approach our example Start expression 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` rule 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 at compilation 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 at compilation 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 in Clean: if a rewriting is needed with `EnumFrom`, a new list is created consisting of the list's head number, and an `EnumFrom` and the next number.

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

All other functions can also be translated into templates using analogies with the previous examples.

In the following we present the parser recognizing EClean expressions, and transforming them to the previous form.

3.2. **The parser.** The parser was written in Java, using the ANTLR LL(k) parser generator. The parser recognizes 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.

The parser's workings are as follows. The first stage in transforming an embedded clean code into a template metaprogram is parsing the EClean code. The notation for distinguishing between regular C++ code and EClean code is the two apostrophes: `''`

3.2.1. *Function transformation.* Each function's signature is recorded when the function's declaration is parsed. At the same time, the declaration is transformed into a general template definition with the `NoMatch` tag, supporting the non-matching cases of the graph rewriting.
Let us consider the following example:

```
take :: Int [Int] -> [Int]
```

This function declaration is transformed into the following template:

```
template <class,class>
struct take {
    typedef NoMatch right;
};
```

The two function alternatives of `take` are transformed as follows:

```
template <int n, class x, class xs>
struct take<mpl::int_<n>, Cons<x,xs> > {
    typedef Cons<x,take<mpl::int_<n - 1>,xs> > right;
};

template <class xs>
struct take<mpl::int_<0>, xs> {
    typedef NullType right;
};
```

The first alternative accepts three parameters, an `int n` representing the first `Int` parameter (how many elements we want to take), and two arbitrary types `x` and `xs` representing the head and tail of a list. On the other hand it is guaranteed that when this function is invoked, `x` will always be a `mpl::int_`, and `xs` will either be a list of `mpl::int_` types, or the `NullType` (`Nil`). The working mechanism of the parser's code transformation is the guarantee for this.

3.3. **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 at compilation 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 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:

```
 1 template <class T1, template <class> class Expr>
 2 struct Eval<Expr<T1> >
 3 {
 4     typedef typename
 5         if_c<is_same<typename Expr<T1>::right,
 6                      NoMatch>::value,
 7         typename
 8             if_c<!Eval<T1>::second,
 9                 Expr<T1>,
10                 Expr<typename Eval<T1>::result>
11             >::type,
12             typename Expr<T1>::right
13         >::type result;
14
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 `ints`, or a further subexpression. This way `Eval` handles other templates. 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 3.1. The compile-time `if_c` in line 5 is used to determine if this is the case, and the `Expr<T1>::right` is equal to `NoMatch`.

- If this is the case, 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 course 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 readibility's sake.)

We have demonstrated the evaluation engine's implementation, and its working mechanism.

4. 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, 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++.

In the future we will include support for more scalar types (`bool`, `long`, etc) besides the implemented `Int`, and the list construct. 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.

5. Related Work

Functional language-like behavior in C++ has already been studied. *Functional C++* (FC++) [15] 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 at compilation 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.

6. 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 is highly portable.

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] T. H. Brus, 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, Springer-Verlag, 1987, pp.364-384.

[5] K. Czarnecki, U. W. Eisenecker, R. Glck, D. Vandevoorde, T. L. Veldhuizen, *Generative Programming and Active Libraries*, Springer-Verlag, 2000.

[6] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.
[7] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.
[8] P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers, *Functional programming in Clean*, 2002
[9] D. R. Musser, A. A. Stepanov, *Algorithm-oriented Generic Libraries*, Software-practice and experience 27(7), 1994, pp.623-642.
[10] R. Plasmeijer, M. van Eeekelen, *Clean Language Report*, 2001.
[11] J. Siek, A. Lumsdaine, *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.
[12] J. Siek, *A Language for Generic Programming*, PhD thesis, Indiana University, 2005.
[13] B. Stroustrup, *The C++ Programming Language Special Edition*, Addison-Wesley, 2000.
[14] G. Dos Reis, B. Stroustrup, *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.
[15] B. McNamara, Y. Smaragdakis, *Functional programming in C++*, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.
[16] E. Unruh, *Prime number computation*, ANSI X3J16-94-0075/ISO WG21-462.
[17] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.
[18] T. Veldhuizen, *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36-43.
[19] T. Veldhuizen, *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26-31.
[20] T. Veldhuizen, *C++ Templates are Turing Complete*
[21] I. Zólyomi, Z. Porkoláb, *Towards a template introspection library*, LNCS Vol.3286 (2004), pp.266-282.
[22] Boost Libraries.
`http://www.boost.org/`

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary

E-mail address: `shp@inf.elte.hu, gsd@elte.hu, zsv@inf.elte.hu`