Effective Template Metaprogram Development

Thesis Book

Ádám Sipos

Supervisor:
Zoltán Porkoláb

Eötvös Loránd University
Doctoral School of Informatics
Basics and Methodology of Informatics

Budapest, 2009
**Introduction**

C++ Template Metaprogramming (TMP) utilizes the generic construct of the language, the template. Among the application areas of TMP are: executing algorithms at compilation time making compile-time decisions, generating more optimized, faster code with the compiler, compile-time code transformation, reflection, and others.

The expressive power of TMP has opened unforeseen possibilities. Nowadays the most current C++ research topics are related to templates, and the possibilities of metaprogramming are poorly understood. Besides the academic applications, today an increasing number of industry projects are using the paradigm realizing its positive impact on C++ development.

Despite its advantages template metaprogramming is still not a widely used paradigm. Development tools are lacking: widely used coding standards, programming methods, tools, and libraries are missing. The existing solutions are usually ad hoc, bugfixing is done heuristically, therefore development costs are hard to estimate. The complicated syntax of TMP – one of the primary weaknesses of the paradigm – leads to the creation of code that is hard to write and maintain. On the other hand, C++ compilers cannot efficiently execute metaprograms as they are not specifically designed for processing long instantiation chains. The complex TMP syntax leads to programming errors even when taking extreme caution and using the right methodology during development. Once a programming error finds its way into the code, a difficult debugging process must begin without appropriate support from a programming framework.

In the following we present a template metaprogram calculating the factorial of \( N \) at compilation time:
template <int N>
struct Factorial
{
    enum { value = N*Factorial <N-1>::value };}

template<>  
struct Factorial<1>
{
    enum { value = 1 };}

int main()
{
    int r = Factorial<5>::value;
}

Goals of the thesis

In the present thesis work the template metaprogramming paradigm is discussed. I have examined the general rate of usage of the paradigm today, and discussed the syntax and semantics of TMP codes, and their quality parameters. I have proposed possible solutions for enhancing these parameters. The thesis discussed existing programming libraries and solutions. I have suggested new methods and tools for more effective development of metaprograms.
Theses

Thesis 1 – I have generalized the notion of a well-formed C++ program defined by the ISO/IEC 14882 (1998) C++ standard, extending this definition to template metaprograms executed at compilation time. As our intention is giving solutions to specific problems related to template metaprograms – i.e. correcting erroneous metaprograms –, we must clearly define the criteria for correct and erroneous behavior. I have proposed a generalization of the C++ standard notion of a well-formed program for distinguishing correct and erroneous metaprograms. In this thesis new definitions are suggested for well-formed and ill-formed metaprograms, and analogies are described between runtime and metaprogram error types. Besides soundness, I have also examined other aspects to be considered during development: the types of errors caused by the compiler consuming all of its resources, and the errors arising from writing non-portable code. I have examined the consequences of the different template metaprogram errors, and the relationship of these errors to the functional- and interpreter-like properties of metaprogramming. These new scientific results are published in [1, 5].

Thesis 2 – I have defined EClean, an abstract functional language over template metaprograms. I have defined the transcription mechanism of EClean for the ISO/IEC 14882 (1998) C++ language, implemented an experimental translator that carries out the transcription, and I have analyzed the efficiency of the generated code. One of the most frequent source of errors in TMP is its complex syntax. It is difficult to express even the simplest algorithms and ideas due to the long codes and syntactic burden. On the other hand, current libraries and other metaprogram codes do not take into consideration that TMP inherits the properties of a purely functional lazy programming language, therefore it is a natural approach to write functional-style code in TMP. In order to
overcome this problem, I have defined the EClean language, a syntactic and semantic subset of the purely functional, lazy *Clean* language. I have designed a preprocessor transforming an EClean program into special template code that is executed at compilation time as a regular metaprogram. The proposed method provides a clear, functional-style embedded interface, hiding the syntactic details. I have demonstrated on an implementation of the Erathostenes sieve algorithm utilizing lazy evaluation that the efficiency of code generated with EClean is equivalent to that of a non-generated metaprogram implementation of the sieve algorithm.

These new scientific results are published in [2, 3, 4].

**Thesis 3** – I have defined the conditions for debugging C++ template metaprograms. For debugging purposes I have modified the open source g++ compiler, and have taken part in the research, implementation, and testing of the Templight template metaprogram debugger.

In TMP it is problematic to prevent possible programming errors, and to find existing ones. These two problems decrease the efficiency of metaprogram development. One of the tools for preventing errors is the *static assert* construct, used for checking conditions at compilation time, and emitting an informative message whenever the check fails. I have proposed a possible modification for the static assert construct, aiming to provide more detailed and flexible error messaging. For simplifying the debugging process I have modified the g++ compiler, which now provides information about template instantiations, therefor it is easier to trace the execution of the metaprogram. Similarly, information on the instantiations, type arguments, inner typedefs, etc. can be redirected into a *trace file* for subsequent analysis. I have proposed the *Templight* template metaprogramming framework designed for the above tasks.

These new scientific results are published in [1].
**Future work**

Currently the Templight debugging framework carries out *post-mortem* debugging, i.e. the instantiation chain of an already executed metaprogram can be traced. The possibility of intervening during metaprogram execution time – similarly to regular runtime debuggers – would speed up the debugging process.

The thesis provides tools for an important research field related to error prevention, contract-based metaprogram development. Similarly, the application of subtype marks on metaprogram correctness would be an interesting research topic.

The application of slicing methods to metaprograms is another possibility for detecting errors in the code. As opposed to runtime codes, metaprograms have no unknown values (e.g. user input), no virtual functions, etc. However, metaprograms can be passed as arguments to other metaprograms, therefore the examination of the calling context would be necessary. On the other hand, since the runtime and compilation time of metaprograms coincide, static slicing is equivalent to dynamic slicing.

It calls for further research to extend the interface of EClean to other functional languages, primarily Haskell. The implementation of a callback mechanism between EClean and the template metaprogram compilation time would be an important result, making possible the manipulation of C++ types and functions in EClean preprocessing time.

It would be useful to define code metrics over template metaprograms, and particularly to measure the psychological complexity of metaprograms. Thus metaprogram project costs could be estimated more precisely.
Bibliography

[1] Z. Porkoláé, J. Mihalicza, Á. Sipos:
   *Debugging C++ Template Metaprograms*,
   In The ACM Digital Library, pp. 255-264, GPCE, Portland, USA, 2006

[2] Á. Sipos, N. Pataki, Z. Porkoláé:
   *Lazy Data Types in C++ Template Metaprograms*,
   5th International Workshop on Multiparadigm Programming with
   Object-Oriented Languages (ECOOP MPOOL), Berlin, 2007

[3] Á. Sipos, Z. Porkoláé, V. Zsók:
   *Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms*,
   In Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII

[4] Á. Sipos, V. Zsók:
   *EClean - An Embedded Functional Language*,
   In Porkoláé et al ed.: Workshop on Generative Technologies Draft
   Proceedings, pp. 1-12, Budapest, 2008

[5] Á. Sipos, Z. Porkoláé, I. Zólyomi:
   *On the Correctness of Template Metaprograms*,
   In Kovcs et al ed.: Proceedings of the 7th International Conference on
   Applied Informatics, pp. 301-308, Eger, 2007

   *Towards Profiling C++ Template Metaprograms*

*Implementation of a Finite State Machine with Active Libraries in C++*


[8] Á. Sipos, N. Pataki, Z. Porkoláb:

*On Multiparadigm Software Complexity Metrics*

In Tallos ed.: Pure Mathematics and Applications Vol. 17, No. 3-4, pp. 469-482, Budapest, 2006