The connection between C++ template metaprogramming and functional programming

Theses of the doctoral dissertation
2013

Ábel Sinkovics
abel@elte.hu

Thesis advisor: Zoltán Porkoláb, PhD
Eötvös Loránd University, Faculty of Informatics,
1117 Budapest, Pázmány Péter sétány 1/C

ELTE IK Doctoral School
Doctoral program: Az informatika alapjai és módszertana
Head of the doctoral school: Dr. András Benczúr, academicer
Head of the doctoral program: Dr. János Demetrovics, academicer
The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).
1 Introduction

This dissertation introduces advanced techniques for C++ template metaprogramming supporting the developers and maintainers of applications and libraries implemented in C++. It assumes, that the reader is already familiar with the C++ programming language.

In 1994, Erwin Unruh demonstrated [34] that it is possible to execute algorithms using a C++ compiler as a side-effect of template instantiations. Programs based on this technique are called C++ template metaprograms and they form a Turing-complete sub-language of C++ [37]. Most developers don’t work on template metaprograms directly, but use libraries that are based on template metaprograms. Since template metaprograms are executed at compile-time, a number of extensions to the C++ language can be implemented using them without having to change the compiler itself.

- For C++ libraries supporting a specific domain (database access, regular expressions, etc.) it is useful if domain-specific errors (database field type mismatches when reading or writing code, invalid regular expressions, etc.) can be caught at compile-time, instead of leaving them in the code and break the program execution at runtime. C++ template metaprograms make it possible to implement such verifications [7].

- Domain Specific Languages (DSL) [6] are more and more popular. They are small languages targeting one specific domain. For that domain, they are better suited than other languages, but they are not useful in other domains. There are widely used and well known domain specific languages, such as SQL or the syntax of regular expressions. DSLs are used together with one (or more) general purpose language in a way, that the parts of the program describing the domain specific logic are implemented in the DSL, while the rest of the code is implemented in the general purpose language. When the code snippets written in the DSL are embedded into a source code written in another (in most cases general purpose) language, they are called Embedded DSLs, or EDSLs. Template metaprograms can be used to embed such languages into C++ [29].

- Developers have to write repetitive code from time to time, where there are minor differences between the repetitions. In most cases, such code snippets are implemented by copying and updating another one. Template metaprograms provide solutions for making the C++ compiler generate such code snippets [1].
C++ compilers optimise the code to make it run faster or consume less memory. However, the options for the compiler to optimise are limited by the fact that the optimised code has to work the same way as the original one. In many cases, further optimisations could be done based on extra knowledge about the domain of the application. The compiler is not aware of the domain and can not implement these optimisations, however, they can be implemented using template metaprogramming techniques [36, 38, 35].

When new features are added to the C++ language or when someone would like to try a new idea for a language feature out, the compiler needs to be updated, which is not always easy or even possible. However, many such features can be simulated using template metaprogramming techniques [39].

C++ template metaprograms can be implemented following the standard, thus all of the standard compliant compilers can understand and execute them.

2 Goals

During the development of C++ it was not a design goal to support template metaprogramming. As a result the syntax became complex. It is difficult to read, write and maintain template metaprograms.

The connection between the functional programming paradigm and C++ template metaprogramming is well known [14, 3, 15, 5, 18, 32, 8, 9]. There are a number of similarities between the logic of C++ template metaprogramming and functional languages, like Haskell. For C++ code executed at runtime there are libraries supporting functional programming [13, 4] but in template metaprogramming current approaches [1, 10, 2] try to simulate imperative languages and libraries, such as the Standard Template Library of C++ [12, 16] and most of them does not take advantage of the functional paradigm.

This dissertation evaluates how the readability of template metaprograms can be improved by taking advantages of the functional nature of it. Two different approaches are discussed.

One of them extends the tools and techniques currently used in template metaprogramming with elements commonly used in functional programming languages. Among others it introduces algebraic data types, let expressions, pattern matching, currying and typeclasses in template metaprogramming.
It also presents an implementation of monads which can be used to provide list comprehension and to simulate exception handling.

The other one is based on providing a Haskell-like DSL for template metaprogramming. Metaprograms written in that DSL are embedded into C++ code in string literals and transformed into template metaprograms during the compilation of the source code they are embedded into. The resulting metaprograms are executed immediately. In order to achieve this latter goal, this dissertation discusses how string literals can be processed by template metaprograms. Being able to embed DSL code snippets into C++ source code in string literals and process them at compile-time makes it possible to provide smooth integration of DSLs into C++ without external tools. The dissertation presents example applications of this technique.

3 Results

In spite of the known similarities of template metaprogramming and the functional paradigm the current practice of template metaprogramming is still not based on the functional paradigm. This dissertation presents two approaches to develop template metaprograms following the functional paradigm. All of the techniques discussed are based on the C++11 standard, they can be used with any standard compliant compiler. None of them requires external tools.

The logical structure of the dissertation is presented in figure 1. The cloud represents the initial idea of the dissertation. The rounded rectangles represent the theses. The section numbers in which the topics are discussed are added to the diagram in the small rectangles. The rectangles with thick grey borders represent the benefits of following the topics discussed in this dissertation.

The first approach simulates basic functional language elements in template metaprogramming and builds higher-level abstractions on top of them. The new elements and techniques are built on top of the widely used elements of the Boost.MPL library, thus they can be easily adopted in programs already using that library.

Thesis 1: I have evaluated the connection between C++ template metaprogramming and functional programming languages. Based on the results I have developed methods for supporting template metaprogrammers using the functional paradigm explicitly. (chapter III)

Thesis 1.1: I have shown the importance of laziness in template metaprogramming and developed an automated adaption method to use non-lazy metafunctions in a lazy way. (section III.1)
Figure 1: Structure of the dissertation
Thesis 1.2: I developed a method for effective implementation of currying in C++ template metaprogramming. (section III.2)

Thesis 1.3: I have developed a method for representing Haskell-like algebraic data-types in C++ template metaprogramming. (section III.3)

Thesis 1.4: I have developed a method for representing Haskell type-classes in C++ template metaprogramming. (section III.4)

Thesis 1.5: I have developed a method to handle template metaprogramming expressions as first class citizens, ie. they can be stored, passed as parameters or returned by functions. This method enables the implementation of let expressions and provides a more convenient way of implementing lambda expressions than what Boost.MPL’s lambda expression implementation, a widely used solution offers. (section III.5)

Thesis 1.6: I have implemented an alternative method for pattern matching in C++ template metaprogramming. This enables the implementation of case expressions. (section III.6)

Following the first approach, a common abstraction in functional languages, monads and a useful syntactic sugar, the do notation Haskell provides for monads are implemented in C++ template metaprogramming. A number of useful techniques can be built based on this. List comprehension can be provided for template metaprogramming based on the List monad. It makes list transformations easier to write, read, understand and maintain. Monads simplify error propagation in template metaprograms and this dissertation shows how to simulate exception handling in template metaprogramming based on them.

Thesis 2: I have developed a method for implementing monads and a Haskell-like do syntax in C++ template metaprogramming and evaluated how a number of different monad variations available in Haskell can be implemented using this method. Based on this I have developed a method for simulating exception handling in C++ template metaprograms. (chapter IV)

Thesis 2.1: I have developed a method for implementing monads in C++ template metaprogramming. (section IV.1)

Thesis 2.2: I have evaluated how a number of monads available in Haskell can be implemented using the approach presented in Thesis 2.1. (section IV.2)

Thesis 2.3: I have developed a method for implementing a Haskell-like do syntax in template metaprogramming. (section IV.3)

Thesis 2.4: I have developed a method for simulating exception handling in C++ template metaprogramming based on monads. (section IV.4)
This dissertation evaluates using two different approaches how the readability of template metaprograms can be improved by taking advantages of the functional nature of it. The first two theses extend the tools and techniques currently used in template metaprogramming with elements commonly used in functional programming languages.

The third thesis presents the second approach, which parses code snippets in string literals at compile-time and builds an interpreter for template metaprograms. Parsing a DSL snippet written in a string literal and its evaluation happens in the same compilation step, which makes it possible to provide a Haskell-like syntax for template metaprograms.

**Thesis 3**: I have developed a method for implementing a parser generator library in C++ template metaprogramming. I have evaluated how it can be used for embedding domain specific languages into C++ and providing a more readable syntax for C++ template metaprogramming. None of these methods require external preprocessors. (chapter V)

**Thesis 3.1**: I have developed a method for turning string literals into character containers for C++ template metaprograms. Utilising this I have developed a method for implementing a parser generator library in C++. (section V.1)

**Thesis 3.2**: I have evaluated how parsers based on Thesis 3.1 can be used to embed domain specific languages into C++ without external preprocessors. (section V.2)

**Thesis 3.3**: I have developed a method based on Thesis 3.1 for providing a Haskell-like syntax for C++ template metaprograms without external preprocessors. (section V.3)

## 4 Summary and future work

The dissertation presents two methods for improving the readability and the maintainability of C++ template metaprograms based on the connection between template metaprogramming and the functional paradigm. One approach is a natural extension of the tools used in practice while the other one provides a more compact syntax.

All the techniques presented in this dissertation have been implemented in an open-source library collection [23]. People can download it and take advantage of the results presented in this dissertation. Table 2 generated using the Cloc [17] utility shows the number of lines of code of the libraries, their tests, the examples and the documentation. The results and the library have been presented to the C++/Boost community. The lecture won the Best Presentation award on the C++Now conference, 2012, Aspen.
Table 1: Related publications

<table>
<thead>
<tr>
<th></th>
<th>[19]</th>
<th>[20]</th>
<th>[21]</th>
<th>[22]</th>
<th>[23]</th>
<th>[24]</th>
<th>[25]</th>
<th>[26]</th>
<th>[27]</th>
<th>[28]</th>
<th>[29]</th>
<th>[30]</th>
<th>[31]</th>
<th>[33]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>×</td>
</tr>
<tr>
<td>2.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Lines of code in Mpllibs

<table>
<thead>
<tr>
<th>Language</th>
<th>files</th>
<th>blank</th>
<th>comment</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTML</td>
<td>186</td>
<td>408</td>
<td>0</td>
<td>12383</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>232</td>
<td>2316</td>
<td>1159</td>
<td>10323</td>
</tr>
<tr>
<td>C++</td>
<td>182</td>
<td>2734</td>
<td>1446</td>
<td>8448</td>
</tr>
<tr>
<td>CMake</td>
<td>32</td>
<td>110</td>
<td>132</td>
<td>160</td>
</tr>
<tr>
<td>CSS</td>
<td>1</td>
<td>19</td>
<td>7</td>
<td>64</td>
</tr>
<tr>
<td>YAML</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>19</td>
</tr>
</tbody>
</table>

References


[27] **Sinkovics, Á., and Porkoláb, Z.** Expressing c++ template metaprograms as lambda expressions. In Horváth et al. [11], pp. 97–111.


