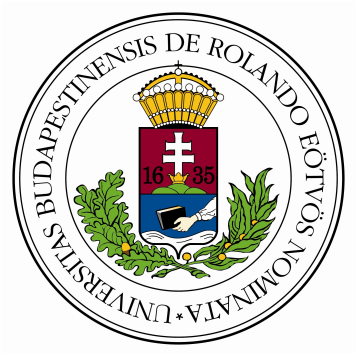
Analysis and Methods for Supporting Generative Metaprogramming in Large Scale C++ Projects

Theses of the doctoral dissertation
2014

József Mihalicza
jmihalicza@gmail.com



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

ELTE IK Doctoral School
Doctoral program: Foundations and Methodologies of Informatics
Head of the doctoral school: Prof. Dr. András Benczúr
Head of the doctoral program: Prof. Dr. János Demetrovics

Contents

1 Introduction

Generative program systems Generative programming is an approach in which carefully generalised software modules are created and then concrete variations of these generalised modules are combined to fulfil the specific needs in different situations. A program system is generative if it employs generative programming. The transition from the generalised version to the concrete variation can either be run time, load time, link time, compile time, or some mixture of these. This thesis discussion is restricted to the compile time version, where parts of the program code are generated by the production toolchain. Among the frequent usages of generic programming are applying the DRY principle and generic constructs.

The DRY principle is short for "do not repeat yourself". The point here is consistency. Whenever the same information is present multiple times, it is an implicit task to keep them consistent, otherwise bugs occur. DRY not only saves you from having to apply a change consequently. In many cases, in particular what generative programming is mostly good for, a long, tedious, or even unfeasable process can be automated.

Generics are the application of the DRY principle in a very direct form. They enable the programmer to write the same algorithm or data structure or big program components once in a general form, which later can be reused in concrete situations, substituting the free parameters with actual values. These free parameters are usually types or constants. Some languages also allow generic parametrisation with function references or other generic constructs.

Programs that manipulate other programs as data are called metaprograms. We speak about generative metaprogramming if generative programming is implemented by metaprograms.

Large-scale program systems There is no objective definition for what large-scale means. The definition I gave is the following: a program system is large-scale, if it is not feasable for a single programmer to know each subtle detail of the implementation. The `zlib` compression library counts 13k LOC with 2 main authors. Linux kernel 3.2 has 15M LOC with 1316 developers involved. Daniel Chapman in [16] says: "The kernel project has long since grown to a size where no single developer could possibly inspect and select every patch unassisted." Interestingly, complexity tends to grow together with size regardless of programming language, paradigm or methodology.

An important aspect is build time. It is not uncommon for a large-scale system to have build times of hours. Build time is an example of non-functional requirements. Another category of non-functional require-

ments is run time performance: speed, responsiveness, CPU and/or memory load, usage of exclusive resources etc. Performance problems are typically sneaky. Performance indicators rarely get worse dramatically. It is rather common that they have a slow tendency of becoming worse and worse.

Comprehension also gets more and more important as size grows. This need is present at each abstraction level: good naming convention for the source code elements, design review, communicating major architectural decisions. If communication is insufficient, diverse solutions appear to similar problems. This not only leads to an inconsistent design, but further extends the code base to maintain, with each its long term consequences listed in this section.

2 Goals

We have seen that large-scale systems have numerous additional issues to handle compared to smaller ones. This is, of course, true for large-scale generative systems as well. My theses focus on specific areas. Instead of giving general answers to general questions, my goal was providing readily applicable methods, approaches to some selected every day industrial problems.

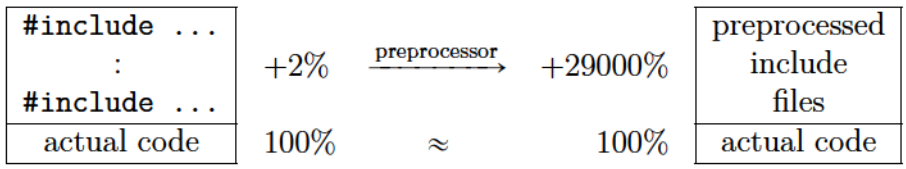
Full build time can be substantially reduced with unity build, which is a very unconventional compilation approach, where originally separately compiled units are compiled together. My aim was evaluating the applicability of unity build on large C++ projects and establishing the foundations of the technique by detailed analysis. Is unity build effective enough in reducing build times to compensate for the unconventional, and consequently error-prone use of the language? What are the possible side effects of applying unity build? How can we avoid the unwanted side effects of unity build?

It was two decades ago when the first C++ template metaprogram was published. It turned out that template metaprogramming (TMP) in C++ is actually a Turing-complete embedded functional language. While diagnostic functionalities are natural parts of most programming environments, this was not true for C++ TMP. Still, after many years and wide usage, there was no convenient toolset that supports TMP. My goal was finding methods for debugging and profiling template metaprograms as naturally as with run time programs. Is there a reliable way to get information about the steps of a running template metaprogram without modifying the compiler? What methods could provide the same debugging functionalities for TMP that we are used to for run time programs (stepwise execution, breakpoints, call stack)? What methods could provide memory usage and execution time profiling for TMP?

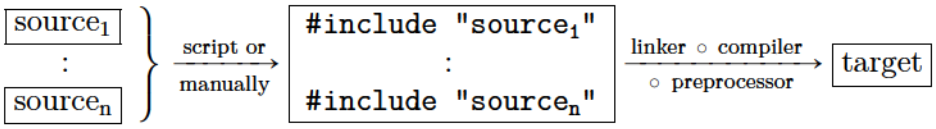
For effective memory behaviour analysis, a type-aware heap profiler is a must. Contrary to other OO languages, C++ does not have direct support for tracking types of allocated objects. I aimed at providing a complete solution for type-aware heap profiling of large scale C++ programs. Is there a reliable way to preserve type information of allocations to use in a C++ heap profiler without modifying the compiler? Provided that we have detailed information about the allocations of our application, including object types, how can we effectively use that information for detailed analysis of heap behaviour? What algorithms can ensure smoothness and responsiveness of an analyser application if the analysed heap behaviour data is collected from long program execution and is therefore huge?

3 Analysis of unity build

According to my measurements with open source libraries the amount of processed source code is significantly larger than the amount of effective code:



Unity build eliminates multiple processing of the same header files in different source units by compiling several, originally standalone units as single compilation unit:



My measurements showed that this unification is able to significantly reduce build time even if the project already used precompiled headers for faster compilation. Therefore the method has relevance in large systems.

I enumerated the possible side effects that are caused by `#include`-ing originally standalone compilation units into one. Originally local (`static`) symbols suddenly become visible in unrelated units, or locally introduced preprocessor macros may have side effects in foreign units, for example. I provided best practices for handling problematic situations, such as conflicting symbols or ODR violation.

I formulated a set of equivalence criteria for determining if the unity built code has any silent side effects compared to the one by one compiled version:

$$EQ := EQ_u \wedge EQ_t \wedge EQ_a$$

EQ_u checks unit level consistency ensuring that the same units are compiled. EQ_t tests if the token sequence of the corresponding units are exactly the same in the two compilation variants. EQ_a, the strongest check, compares substructures of abstract binding trees to avoid side effects caused by dangling `using` declarations or any other hardly detectable semantic difference.

Thesis 1 (Analysis of unity build)**.** *I analysed the unity build technique and measured its effectiveness to reduce build times focusing primarily on full build times. Based on case studies with three open source libraries I determined the pros and cons of applying unity build on existing projects and derived recommended approaches for its implementation. I defined a set of criteria that allow automatic detection of unwanted silent semantic changes caused by the unification. I also derived an algorithm for the implementation of this equivalence check.*

4 Template metaprogram diagnostics

To support TMP diagnostics we should somehow get information out of the compiler about the template instantiations during compilation. Unruh's prime number generator is a template metaprogram that prints prime numbers in the form of warnings at compile time. This means that we have a method for outputting information from within a template metaprogram without stopping its execution. I elaborated this approach and created a complete workflow for retrieving a position correct trace file describing TMP related events of the compilation. Figure 1 shows the structure of the framework, that I named Templight.

The process begins with the automatic identification of template constructs in the source code (annotation). Then warning-emitting code fragments are instrumented at the beginning and end of each template definition. When the compiler instantiates a template, these code fragments produce a warning message on the compilation output signalling that a template related event occurred. These template events are reconstructed from the compilation output, and are written to a trace file.

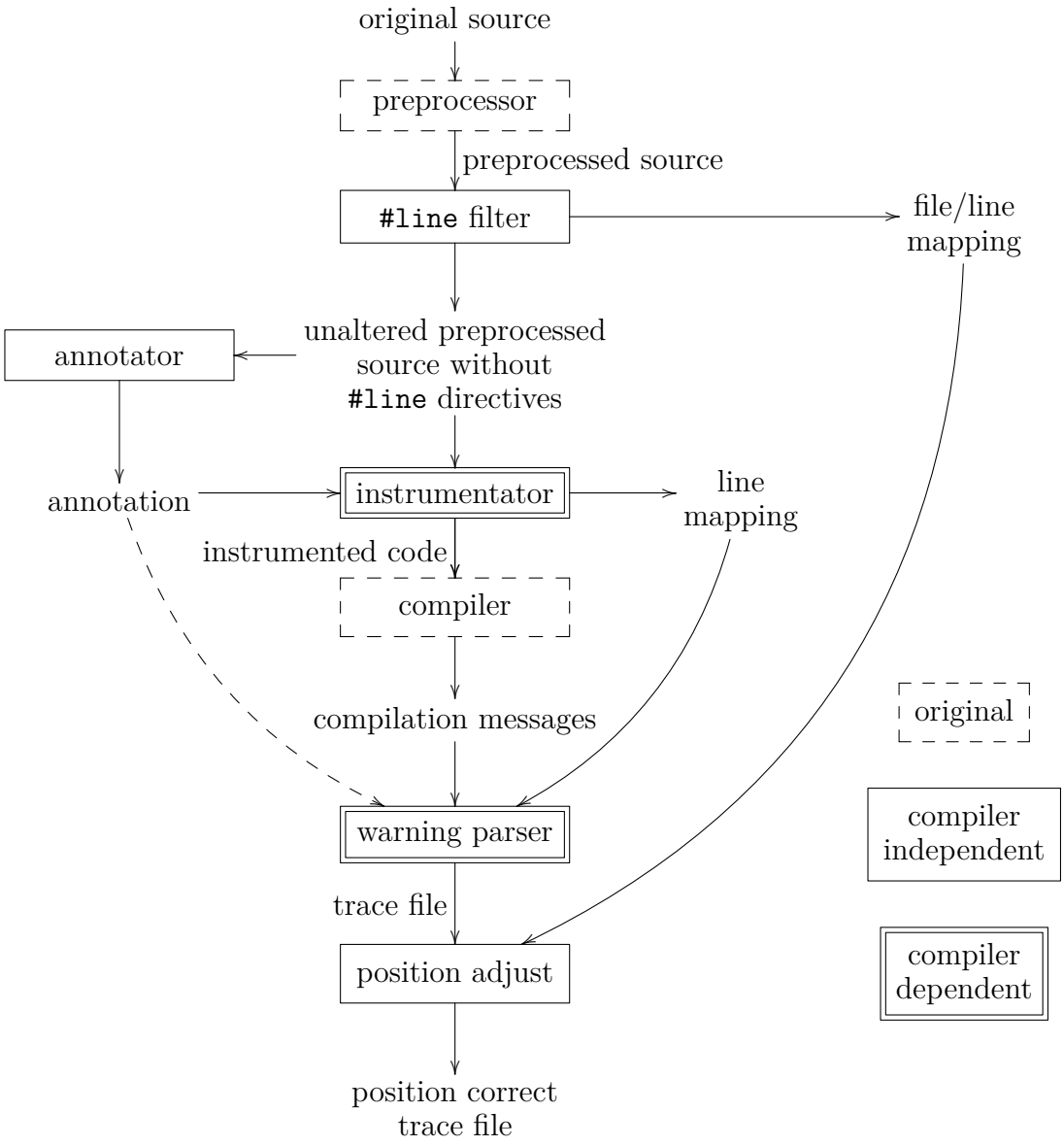


Figure 1: Templight control flow. Dashed line denotes optional connection.

Thesis 2 (Template metaprogram instrumentation)**.** *I developed a method for instrumenting C++ template metaprograms. The basic concept of the method is the pattern based recognition of template constructs and the insertion of warning generator code fragments. Based on the method I implemented a prototype framework that outputs a position-correct trace file containing all necessary information to reconstruct the full TMP execution. The framework can be adapted to new compilation environments easily.*

When thinking about an ideal TMP debugger, first I enumerated the frequently used debugging operations of run time programs. I found an analogy between TMP execution and run time execution, and mapped these debugging operations to TMP. According to this analogy I specified the operations of a TMP debugger. Based on the properties of TMP (referential transparency, for example, that comes from TMP being a functional language, as already stated in Section 2) I derived that reverse debugging and direct time jump operations can be implemented in a TMP debugger.

For trace file generation besides the more portable instrumentation method of Thesis 2 I also discussed the more powerful compiler modification variant. I showed three different TMP debugger applications: an IDE extension [8], a standalone TMP IDE [3], and a standalone TMP debugger [14].

Thesis 3 (Methods for template metaprogram debugging)**.** *I developed two methods for implementing debuggers for C++ template metaprograms. Both method is based on a trace file describing the template events. Besides the non-intrusive data extraction method that utilises the instrumentation technique of Thesis 2, I presented one that is based on compiler modification, and has some advantages over the non-intrusive version. I showed three applications that demonstrate how my methods can be used to implement template debugger functionalities. An implementation based on my research is becoming de facto standard for template metaprogram debugging [14, 17, 18, 19, 20].*

For TMP profiling I identified one existing and three new methods:

FC measure full compilation time
WO instrument warning generating code fragments and measure times out-of-process
WI instrument warning generating code fragments and measure times in-process
BI built-in support for profiling

In case of FC I analysed the effects of separate preprocessing as a possible enhancement. I described the details of the BI method with important implementation considerations about achieving insignificant distortion. Then I presented measurements performed by using the BI method, to analyse different aspects of the WI and WO methods and TMP compilation times in debug and release modes. I discussed the pros and cons of each method, compared their implementation difficulties and precision based on the measurements.

Thesis 4 (Methods for template metaprogram profiling)**.** *I identified one existing and three new methods for implementing profilers for C++ template metaprograms with different strengths. Supported by detailed measurements I pointed out important aspects of applying these methods in practice. An implementation based on my research is becoming de facto standard for template metaprogram profiling [14, 18, 19, 20].*

5 Type-preserving heap profiler for C++

Memory profilers are key tools for understanding the run time behaviour of modern object-oriented programs. Although languages like C++ strongly rely on types and classes, most C++ memory profilers fail to provide sufficient information on actual types of memory allocations.

I showed the system structure and data strucures of a type preserving heap profiler for C++. My framework provides detailed type information of each heap related operation in addition to usual profiling features. I discussed important implementation details as well as possible pitfalls and their solutions.

To support program comprehension a filter graph approach allows the user to construct arbitrary queries in a fairly convenient way. Various visualisation possibilities are available to examine profiling result. My solution is highly platform independent and has moderate memory and speed footprint.

Timeline view displays total memory occupation of the application as a function of time. This view is very useful in practice, but it needs efficient algorithms to cope with the huge amount of aggregated entries. I used real usage statistics to determine key optimisation targets and presented asymptotically optimal algorithms for timeline view visualisation. My algorithm for displaying cumulative values is capable of handling millions of allocation entries and displayed entries well above 10^{10}.

As a case study I used the framework to find and fix a performance issue in a code base that was new and unknown to me.

Thesis 5 (Type-preserving heap profiler)**.** *I developed a method for preserving type information in C++ heap profilers. The method is platform independent, and has low memory footprint. I provided optimal algorithms for the performance critical timeline view visualisation. I demonstrated the applicability of the approach in real projects by pinpointing and fixing a performance bug in an open source text editor.*

thesis name	relevant publications											
	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]
Analysis of unity build	•	•										
Template metaprogram instrumentation			•	•	•	•	•		•	•	•	
Methods for template metaprogram debugging			•	•	•	•	•	•				
Methods for template metaprogram profiling				•	•	•	•		•	•	•	
Type-preserving heap profiler												•

6 Impact

The relevance of my research is demonstrated by citations both in academic papers and by developer communities. I found 14 independent citations to my publications. 18 additional citations are from my fellow researchers to 8 of my publications in 10 papers. Altogether 32 citations refer to 9 of my publications, covering all five theses.

The Templar TMP debugger and profiler [14] that is based on my TMP diagnostic methods is getting more and more popular. In [18] a gdb-like command line interface is available for TMP debugging. This debugger uses Template Instantiation Observer, an extension of clang for monitoring template instantiation events, which is a generalisation of the former Templight implementation. The corresponding modifications in clang (Templight trace generation [19] and the observer [20]) are expected to become official features of the compiler making the method de facto standard.

For other compilers and IDEs I found feature requests for TMP debugging and profiling referencing my corresponding publications.

My heap profiler was chosen to be the opening meetup topic [15] of the Hungarian C++ Community. Authors of a recent work [21] cite my publication as the only profiler they are aware of that provides fine-grained type information for C.

References

[1] J. Mihalicza, "Compile C++ systems in quarter time," in *Proceedings of 10th International Scientific Conference on Informatics (Informatics'2009)*, 2009, pp. 136–141.

[2] J. Mihalicza, "How #includes affect build time in large systems," in *Proceedings of the 8th international conference on applied informatics (ICAI 2010)*, vol. 2. Eger: BVB Nyomda és Kiadó Kft., 2012, pp. 343–350.

[3] Z. Borók-Nagy, V. Májer, J. Mihalicza, N. Pataki, and Z. Porkoláb, "Visualization of C++ template metaprograms," in *Proceedings of the 2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, ser. SCAM '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 167–176. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2010.16

[4] Z. Porkoláb, J. Mihalicza, and Á. Sipos, "Debugging C++ template metaprograms," in *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, ser. GPCE '06. New York, NY, USA: ACM, 2006, pp. 255–264. [Online]. Available: http://doi.acm.org/10.1145/1173706.1173746

[5] J. Mihalicza, "C++ template metaprogramok nyomkövetését segítő programcsomag," Master's thesis, Eötvös Loránd Tudományegyetem, 2006.

[6] N. Pataki, J. Mihalicza, Z. Szűgyi, V. Májer, and Z. Porkoláb, "Features of C++ template metaprograms," in *Proceedings of the 8th international conference on applied informatics (ICAI 2010)*, vol. 2. Eger: BVB Nyomda és Kiadó Kft., 2012, pp. 451–451.

[7] Z. Porkoláb, J. Mihalicza, Á. Sipos, and N. Pataki, "Templight, a template metaprogram debugger," 2007, poster at European Conference on Object-Oriented Programming (ECOOP'07).

[8] Z. Porkoláb, J. Mihalicza, Á. Sipos, and N. Pataki, "Templight, a template metaprogram debugger," 2007, demonstration at European Conference on Object-Oriented Programming (ECOOP'07).

[9] Z. Prokoláb, J. Mihalicza, N. Pataki, and Á. Sipos, "Towards profiling C++ template metaprograms," in *Proceedings of the 10th Symposium on Programming Languages and Software Tools (SPLST'07)*. Eötvös University Press, 2007, pp. 96–111.

[10] Z. Porkoláb, J. Mihalicza, N. Pataki, and Á. Sipos, "Analysis of profiling techniques for C++ template metaprograms," *Annales Universitatis Scientiarum Budapestinensis de Rolando Eötvös Nominatae. Sectio Computatorica*, vol. 30, pp. 97–115, 2009.

[11] J. Mihalicza, N. Pataki, and Z. Prokoláb, "Compiler support for profiling C++ template metaprograms," in *Proceedings of the 12th Symposium on Programming Languages and Software Tools (SPLST'11)*, oct 2011, pp. 32–43.

[12] J. Mihalicza, Z. Porkoláb, and Á. Gábor, "Type-preserving heap profiler for C++," in *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ser. ICSM '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 457–466. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2011.6080813

[13] Z. Prokoláb, Z. Borók-Nagy, and J. Mihalicza, "Debugging and profiling C++ template metaprograms," http://github.com/boostcon/cppnow_presentations_2013/blob/master/thu/tmpdebug_cppnow13.pdf?raw=true, May 2013, presentation at C++Now Conference.

[14] Z. Prokoláb, J. Mihalicza, and Z. Borók-Nagy, "Templight: A C++ template metaprogram debugger and profiler," http://plc.inf.elte.hu/templight/, 2014.

[15] J. Mihalicza, "Type-preserving heap profiler for C++," http://www.meetup.com/Hungarian-Cpp-Community/events/205167032/, Budapest, Hungary, September 2014, presentation at Hungarian C++ Community Meetup.

[16] D. Chapman, "How to participate in the linux community," http://www.linuxfoundation.org/content/23-how-patches-get-kernel, May 2011.

[17] M. Schulze, "Templar: Visualization tool for Templight C++ template debugger traces," http://github.com/schulmar/Templar, Feb 2014.

[18] S. M. Persson, "Templight 2.0: Template instantiation profiler and debugger," http://github.com/mikael-s-persson/templight, October 2014.

[19] M. Clow, "Template tracing patch from Zoltan Porkolab," http://reviews.llvm.org/D809, May 2013.

[20] S. M. Persson, "Template Instantiation Observer + a few other templight-related changes," http://reviews.llvm.org/D5767, October 2014.

[21] K. Saur, M. Hicks, and J. S. Foster, "C-Strider: Type-aware heap traversal for C," May 2014.