Thesis book

Extension of strongly typed object-oriented systems using metaprograms

István Zólyomi

Supervisor: Dr. Zoltán Porkoláb

Eötvös Loránd University
Faculty of Informatics
Department of Programming Languages and Compilers

Doctoral School on Informatics
The Basics and Methodology of Informatics

Head of school: Dr. János Demetrovics

Budapest
2009

1 Introduction

Without doubt, the object-oriented paradigm is the most widespread programming paradigm of our days. Besides its well-known advantages, many shortcomings and limitations appeared during several decades of its application. These drawbacks inspired fundamentally new methodologies, and also a lot of extensions to the paradigm, such as the aspect-oriented. feature-oriented, intentional or generic programming. Certainly, they also have their own limitations besides the advantages. Carefully selecting the appropriate paradigms for a specific task, multiparadigm programming aims to eliminate these limitations and unite the advantages of the different paradigms.

However, as a natural abstraction of these extensions and methodologies, we think that metaprogramming seems to be the most promising paradigm. Metaprogramming is general, not strictly based on the object-oriented or any single programming paradigm. To execute transformations on other programs, it uses metaprograms, which are clearly separated from conventional programs that they transform. Thus, all the forementioned paradigms can be considered as a special case of metaprogramming.

Metaprograms need to access information on the program that it processes and transforms, e.g. type information and expression trees. On the other hand, the metaprogram has to be able to execute transformations on the existing program, e.g. generate new code or transform its expression tree. These requirements are still not supported well in practice, metaprogramming is still cumbersome and complex. For this reason, it is mostly applied when no other methodology remains to provide a reasonable solution.

Maybe the first and most important area of application is the optimization of resources (e.g. time or memory) for programs, where metaprograms provided significant gains. Metaprogramming is also

often used for implementing embedded and multistage languages in order to fit a domain-specific programming language into a general-purpose one. Another area of application is generating code based on type information so as to store or convert data automatically, or generate code based on function signatures for communication that follows a certain protocol. Most modern integrated development environments are have a level of automated refactoring capabilities, such as transforming a code block into a function, rearrangement of function parameters or elimination of unreachable code.

2 Goals

In this thesis work I examine the boundaries and limits of application of strongly typed object-oriented programming. My goal is to extend the capabilities of widespread object-oriented systems, for which metaprogramming proves to be an ideal instrument.

I implemented my metaprograms in the C++ programming language, mostly using the template feature of the language. Templates are designed to provide a customizable facility for generating code. Additionally, it was discovered that templates give a separate, Turing-complete functional language inside C++. In many years, template metaprogramming evolved and became an accepted methodology that is used in practice in our days, thus it was natural to choose it as my metaprogramming platform.

3 Results

3.1 Type introspection

Metaprogramming is a relatively new and upcoming paradigm, thus it does not have solid theoretical background and development tools yet. Accessing information on the processed program is an essential necessity. Still, only a few programming languages or development environments support this for metaprograms executed in compile time. I introduced a general system for type introspection. It enables execution of atomic checks based on the predicates of first order logic, and from them, expression of complex conditions using logic operators. Using standard C++ template metaprograms, I gave an implementation for the introspection of types in the C++ language. I published the results in [1, 9].

Thesis 1 I defined a universal, non-intrusive type introspection system based on first order logic. I implemented type introspection for the C++ language, using only C++ template metaprograms in the implementation. This introspection library conforms to the ISO/IEC 14882:1998 standard, thus it is compiler-independent and portable.

3.2 Structural subtyping

An important role of metaprogramming is the extension and improvement of type systems of programming languages. In this thesis work, I showed the weakness in the type system of object-oriented languages expressing subtype relations for stepwise refinement. Solving this weakness requires creating an exponential number of composite classes or interfaces. However, this problem that we named chevron-shape inheritance, can be addressed by using structural subtyping, but no widespread language support this kind of subtyping rules. Hence, I implemented automated conversions for the C++ language that extend the built-in rules of the language and are similar to structural subtyping. The solution is based on code generation automated by template-metaprograms and uses only standard C++ features. This lazy, on-demand code generation reduces the number of generated classes to a linear amount. The results can be found in [3, 5, 7].

Thesis 2 I showed the weakness of widespread object-oriented type systems in the case of stepwise refinement. I solved the problem for the C++ language by simulating structural subtyping. I extended the type system by custom conversion rules which I implemented using standard template metaprograms. Using this method, the combinatorial explosion of generated classes is avoided.

3.3 Serialization

Another important application of metaprogramming is the serialization of data. If reflection information is available in every type, a good, general and format-independent solution can be given for serialization based on reflection. Such reflection support is included in most modern languages, especially when based on virtual machines (e.g. Java or C#) or interpreters. Consequently, such languages have a built in, standard serialization library. These libraries define the schema of serialized data based on reflected type information.

Serialization is a harder task in languages without any kind of reflection. In this thesis work, I show a solution that chooses the opposite direction, and generates types from the schema descriptions of data. The solution is language and format-independent, but requires a mapping of schema types to the types of programming languages. I explained my solution for the serialization of XML documents and implemented it for the C++ language that has no support for type reflection. Since the solution has low few resource usage (especially for memory), it can be an ideal solution for communication on embedded systems such as phones and handheld computers and other hardware with low resources. The details of the solution are available in [2, 4].

Thesis 3 I gave a general method independent of language and format for the serialization of schema-described data. I mapped the type system of XML Schema Descriptions to the C++ programming language and implemented a code generator in XSLT to perform this mapping. Based on the mapping, I implemented a general metaprogram to execute the serialization. The result is published as an official development tool for web services for the S60 platform by Nokia.

4 Future work

Naturally, every thesis of this work provides the possibility of future work by further improvements. In spite of this, I think that this is not the most important direction. During my research, we have continuously struggled with difficulties that arose not from reaching the possible limits, but recognizing that metaprogramming is still in a very unstable, immature state. Hence I think that the examination, improvement and stabilization of metaprogramming can be the most important future work.

Metaprogramming is still done using mostly ad-hoc solutions and is lack of well-designed methodologies. I think this is implied by the absence of a solid theoretical background and adequate development tools. Consequently it would be important to find the appropriate theories and paradigms that would help to design and write sound, safe and maintainable metaprograms. On the other hand, many development tools that are common to conventional programming are completely missing for metaprogramming. We have no metrics to measure complexity, no test environments, no tracing and profiling tools or integrated development environments. In my opinion it is also important to make progress in creating such development tools.

References

[1] István Zólyomi, Zoltán Porkoláb. Towards a General Template Introspection Library. Generative Programming and Component Engineering LNCS Vol. 3286 (2004) pp. 266-282.

[2] Szabolcs Payrits, Péter Dornbach, István Zólyomi. Metadata-Based XML Serialization for Embedded C++. Proceedings of ICWS 2006, pp. 347-356

[3] István Zólyomi, Zoltán Porkoláb, Tamás Kozsik. An Extension to the Subtype Relationship in C++ Implemented with Template Metaprogramming. Generative Programming and Component Engineering LNCS Vol. 2830 (2003) pp. 209-227.

[4] Szabolcs Payrits, Péter Dornbach, István Zólyomi. Metadata-Based XML Serialization for Embedded C++. International Journal of Web Services Research (IJWSR), megjelenés alatt.

[5] István Zólyomi, Zoltán Porkoláb. A Feature Composition Problem and a Solution Based on C++ Template Metaprogramming. Generative and Transformational Techniques in Software Engineering LNCS Vol. 4143 (2006) pp. 459-470.

[6] István Zólyomi, Zoltán Porkoláb. A generative approach for family polymorphism in C++. Proceedings of ICAI 2004, Eger, Hungary, pp. 445-454.

[7] Zoltán Porkoláb, István Zólyomi. An anomaly of subtype relations at component refinement, and a generative solution in C++. MPOOL Workshop, ECOOP 2004, Oslo, pp. 39-44.

[8] Ádám Sipos, István Zólyomi, Zoltán Porkoláb. On the Correctness of Template Metaprograms. Proceedings of ICAI 2007, Eger, Hungary, pp. 301-308.

[9] István Zólyomi, Zoltán Porkoláb. Improving concept checking in Boost. Boost Workshop, OOPSLA 2004, Vancouver.