Chapter 2
Domain-Specific Language Integration with C++
Template Metaprogramming

Ábel Sinkovics
Eötvös Loránd University, Hungary

Zoltán Porkoláb
Eötvös Loránd University, Hungary

ABSTRACT

Domain specific language integration has to provide the right balance between the expressive power of the DSL and the implementation and maintenance cost of the applied integration techniques. External solutions may perform poorly as they depend on third party tools which should be implemented, tested and then maintained during the whole lifetime of the project. Ideally a self-contained solution can minimize third-party dependencies. The authors propose the use of C++ template metaprograms to develop a domain specific language integration library based on only the standard C++ language features. The code in the domain specific language is given as part of the C++ source code wrapped into C++ templates. When the C++ source is compiled, the C++ template metaprogram library implementing a full-featured parser infrastructure is executed. As the authors’ approach does not require other tool than a standard C++ compiler, it is highly portable. To demonstrate their solution, the chapter implements a type-safe printf as a domain specific language. The library is fully implemented and downloadable as an open source project.

DOI: 10.4018/978-1-4666-2092-6.ch002

INTRODUCTION

Although domain specific languages are indispensable in their domain, the vast majority of the programs execute most of their actions out of that domain. As an example, SQL might be a perfect solution for describing operations related to relational databases, but database servers will create threads, open network connections, communicate with the operating system in the means of a general purpose programming language. The usual solution is that the desired domain-specific
Domain-Specific Language Integration with C++ Template Metaprogramming

language or languages are used together with a general purpose programming language. In most cases the integration of these languages happens by embedding the DSL(s) into the general purpose language with or without some syntactical quotation. However, this integration should add minimal syntactical, semantic and maintenance overhead to the project.

Many strategies exist to provide smooth integration of the domain languages and the host language. Some of them apply external frameworks for integration; others are built on language extensions. Only a few solutions are based on standard programming language features like macros or generative language elements. External tools may introduce unwanted dependencies on third party software. There are only a few languages where language extensions do not require translators, precompilers or the modification of the compiler. The most manageable solution is based purely on standard language features.

In this chapter we would like to describe a DSL integration technique for the C++ programming language. The solution is based on compile-time parsing of the DSL code and a further development of the authors’ earlier work (Porkolab, 2010). To eliminate the dependency on external tools we implemented a parser generator as a C++ template metaprogram library. Thus, the full parsing phase of the DSL is executed when the host program is compiled. The end result is an optimized, highly effective executable produced by the C++ compiler. The library uses standard C++ language elements only, thus our solution is extremely portable.

Defining new DSLs is a relatively simple task using our solution. Taking advantage of the declarative nature of C++ template metaprogramming the formal syntax of the DSL can be directly expressed in the source code. Thus, the grammatical rules of the DSL are presented explicitly with a minimal syntactical overhead. The result of this method is a highly self-documenting and maintainable C++ program. Our solution can be applied for a number of practical problems:

Most of the current C++ template metaprogramming code base suffers from unmaintainable syntax. The syntax of Boost.Xpressive and Boost.Proto as examples, are seriously restricted. With the help of our library it is possible to provide a clear syntax using the usual notions of the library domain. When this code is compiled our parser generator library takes effect first, and translates the new syntax to the old one without any user interaction or external tool other than the standard C++ compiler.

A further application area is to raise C++ template metaprogramming itself to a higher abstraction level using functional programming style script languages. Using our library it is possible to implement a Haskell-like syntax for template metaprogramming. Parsers based on our library are able to translate this syntax to template metaprograms as we know them now.

The structure of the chapter is the following. First we give a brief technical overview of C++ template metaprogramming. We explain how templates work and how they can be used to control the compilation process to execute algorithms. We define the basic building blocks of template metapograms: template metafunctions and metafunction classes. Next we discuss some C++ specific examples of DSL embedding, especially those, that are connected to generative techniques. We mention the AraRat system implementing SQL embedding into C++ as the host language using expression templates, Boost.Xpressive, a regular expression library utilizing template metaprogramming for embedding. We also discuss Boost.Proto, a library which is designed for domain-specific language embedding. Then we introduce our solution, a parser generator library working with parser combinators implemented using template metaprogramming. As a real world example, we implement a type-safe printf library using DSL-based approach. This library type checks the printf arguments and reports type mis-