Program Templates: Expression Templates Applied to Program Evaluation

Francis Maes

EPITA Research and Development Laboratory,
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France,
francis.maes@lrde.epita.fr

Abstract

The C++ language features a two-layers execution model: computations can be done at compile-time and at execution-time. This corresponds to the static execution of metaprograms, and the dynamic execution of resulting programs. Thanks to recent evolutions in C++ template metaprogramming ([Veldhuizen(2002)], [Czarnecki and Eisenecker(2000)]), more and more tasks can be done at compilation time.

In C++, a technique called Expression Templates described by [Veldhuizen(1995)] allows the exploitation of this two-layers execution model. This technique relies on transformations of simple arithmetic expressions at compile-time to increase the performances of the executable code. Moreover some evaluation can be done entirely statically with mechanisms such as constant propagation. This way, some computation usually done at execution-time is processed at compile-time.

A program written in any language can also be expressed as an abstract syntax tree. Therefore, it is natural to wonder whether it is possible to extend the Expression Templates technique to a whole programming language. Expressing a full program with a C++ type reflecting its abstract syntax tree (AST) could thus be made possible. In this paper, this type is called the TAT (Tree As Type). A TAT is a representation of an AST using a C++ type formalism.

Expressing a program with a TAT would allow us to adapt the Expression Templates evaluation method to a whole program and therefore to take advantage of the two-layers execution model of C++ (see [Haney and Crotinger(1999)]). The entire process of compiling and executing a program expressed as a TAT corresponds to its evaluation.

To experiment this idea, we have to choose a programming language which is simple and which have few constructions. Nevertheless, this language must at least include types, functions, records, arrays and flow control constructions. Tiger, a language defined by [Appel(1997)], suits our needs: with only 40 rules in its EBNF grammar, it respects all our conditions.

We use a front-end program which parses Tiger, and does the semantic analysis: type checking, scopes and bindings. The output of this front-end is a C++ program which declares a TAT. Our front-end is based on techniques explained by [Appel(1997)]. This front-end associated with the C++ static processor constitute a compilation chain. Indeed, the input of this chain is a textual Tiger program, and its output is an executable program.

Our Tiger compiler is originally inspired by the Expression Templates technique. However, the evaluated constructions are not restricted to basic ones, such as unary or binary operators, but includes the common flow control constructions, structured types, variables, and nested functions. Moreover, thanks to the use of a static environment, such advanced operations can be evaluated by jumping from one point of the program to another. This happens for example each time a function is called. That characteristic is a noticeable difference with the Expression Templates which are evaluated in a simple bottom-up fashion.

This work is essentially a proof of concept. No-one before has mapped an entire language to a C++ meta-program. Those that consider C++ (expression) templates for prototype implementations should be interested in this project.

The C++ meta-language has here been introduced as an intermediate language. This view diverges from the current trend which is to support meta-programming by designing meta-languages as extensions of existing programming languages. This work is of relevance for both the current meta-programming discussions and the on-going discussions around expression templates.

References

[Appel(1997)] A. Appel. *Modern Compiler Implementation in C / Java / ML.* Cambridge University Press, 1997.

[Czarnecki and Eisenecker(2000)] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications.* Addison-Wesley, 2000.

[Haney and Crotinger(1999)] S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *Computing in Science and Engineering*, 1 (4), 1999. URL `http://www.acl.lanl.gov/pooma/papers.html`.

[Veldhuizen(1995)] T. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[Veldhuizen(2002)] T. Veldhuizen. Techniques for scientific C++. Technical report, Computer Science Department, Indiana University, Bloomington, USA, 2002. URL `http://osl.iu.edi/ tveldhui/papers/techniques/`.