Proceedings of the 7th International Conference on Applied Informatics
Eger, Hungary, January 28–31, 2007. Vol. 2. pp. 265–270.

Properties of C++ template metaprograms*

Norbert Pataki, Tamás Kozsik, Zoltán Porkoláb

Dept. of Programming Languages and Compilers
Fac. of Informatics, Eötvös Loránd University, Budapest
e-mail: {patakino, kto, gsd}@elte.hu

Abstract

Verifying properties of programs is a common way to ensure the proper behaviour of those programs. *Invariants, pre-* and *postconditions* are program properties often used when proving correctness of programs.

C++ template metaprograms (TMPs) are special progams interpreted by the compiler. Metaprograms are widely used for the following purposes: executing algorithms in compile-time, optimizing runtime programs and emitting compilation errors and warnings to enforce certain semantic checks.

In this paper we step to "meta-meta-level": we present a technique to make safer C++ TMPs with static asserts. We describe how to check invariants, pre- and postconditions of TMPs and enforce the compiler to refuse metaprograms if any of the specified program properties is dissatisfied. We present some examples where semantic errors in TMPs are revealed by our method.

Keywords: template metaprogramming, program correctness

MSC: 68N19 Other programming techniques

1. Introduction

Correctness of a given program is the most important property. Every programmer wants to avoid bugs and creates error-free code. Testing cannot discover all bugs in a program, so we need something more precise method.

Verifying properties of programs is a common way to ensure the proper behaviour of those programs. *Invariants, pre-* and *postconditions* are program properties often used when proving correctness of programs.

C++ template metaprograming is a recently emerged programming paradigm. We use this paradigm execute algorithms in compilation time, optimize our runtime

*Supported by GVOP-3.2.2.-2004-07-0005/3.0.

programs, create active libraries, etc. Unfortunately, we do not have a framework to check C++ template metaprograms and the common debuggers cannot help generally. C++ template metaprogramming will be more widely-used if the programmers are more aided (e.g. with a correctness framework).

2. C++ Template metaprogramming

C++ template metaprogram actions are defined in the form of template definitions and are "executed" when the compiler instantiates them. Templates can refer to other templates, therefore their instantiation can instruct the compiler to execute other instantiations. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops.

Conditional statements (and stopping recursion) are solved via specializations. Templates can be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [3].

Data is expressed in runtime programs as constant values or literals. In template metaprograms we use `static const` and enumeration values to store quantitative information. Results of computations during the execution of a metaprogram are stored either in new constants or enumerations. Furthermore, the execution of a metaprogram may trigger the creation of new types by the compiler. These types may hold information that influences the further execution of the metaprogram.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [10]. The canonical examples for sequential data structures are `typelist` [1] and the elements of the `boost::mpl` library [4].

However, there is a fundamental difference between runtime programs and C++ template metaprograms: once a certain entity (constant, enumeration value, type) has been defined, it will be immutable. There is no way to change its value or meaning. A metaprogram does not contain assignments. In this sense metaprogramming is similar to pure functional programming languages, where *referential transparency* is obtained. That is the reason why we use recursion and specialization to implement loops: we are not able to change the value of any loop variable. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur.

Template metaprograming is proved to be a Turing-complete sublanguage of C++ [3]. We write metaprograms for various reasons, here we list some of them:

- *Expression templates* [10] replace runtime computations with compile-time activities to enhance runtime performance.

- *Static interface checking* increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [5, 6]. In this paper we want to use this feature to implement safer metaprograms.

- *Active libraries* [11]. Active libraries act dynamically during compile-time, making decisions based on programming contexts and making optimizations.

In our context the notion *template metaprogram* stands for the collection of templates, their instantiations, and specializations, whose purpose is to carry out operations in compile-time. Their expected behavior might be either emitting messages or generating special constructs for the runtime execution.

3. Motivating example

Let us consider the following C++ template programs that calculates the factorial of N in compilation time:

```
template <int N>
struct Factorial
{
  enum {Value = N * Factorial<N-1>::Value};
};

template <>
struct Factorial<0>
{
  enum{Value = 1};
};
```

Unfortunately, this "metafunction" can be called with negative number or we can miss the full specialization (i.e. Factorial<0>).

4. Basic annotations and utilities

First of all, we write a macro to create a static const object from the argument with a unique identifier. The defined object will check the annotation.

```
// Special macro to create a static const object:
#define ANNOTATION_CHECK_REQUIRES(TYPE) static const \
TYPE ANNOTATION_CHECK_FOO___##TYPE
```

We present implementation of some basic annotations with the help of the boost::mpl library:

```
// Is the parameter a positive number?
template <int N>
struct positive {
  BOOST_MPL_ASSERT_RELATION(N, >=, 1);
};
```

An object of the previous type can check if its argument is a positive number. If the argument is not a positive number, the assert raises a compiler error diagnostics. If the argument is positive number it does nothing. Similarly we can implement a class that checks if the argument is a natural number.

This way we can write classes to check if the two arguments are equal or not:

```
// Are the two parameters equal?
template <int I, int J>
struct equal {
  BOOST_MPL_ASSERT_RELATION(I, ==, J);
};

// Are the two parameters not equal?
template <int I, int J>
struct not_equal {
  BOOST_MPL_ASSERT_RELATION(I, !=, J);
};
```

5. Towards safer metaprograms

We saw the implementation of the factorial template metaprogram as a motivating example. We extend this program with special annotations to be safer:

```
template <int N, class Annotation = natural<N> >
struct Fac {
  ANNOTATION_CHECK_REQUIRES(Annotation);
  enum{Value = N * Fac<N-1>::Value};

  typedef positive<Value> Post;
  ANNOTATION_CHECK_REQUIRES(Post);
};
```

We do not have to modify the full specialization. This implementation checks if its argument a natural number and if its result is a positive number. These are not just a simple pre- and postconditions because of the recursion. These annotations are checked in all instantiated classes. Since these annotations are checked in all steps in the algorithm, these checks are worked as invariants.

What can we achieve with this technique? We can get an error diagnostics when the class is instantiated with negative number: for example we call it with a

negative number, or we miss the full specialization. For instance, g++ 3.3.5 gives the following message when we miss the specialization:

```
d.cpp: In instantiation of ‘natural<-1>’:
d.cpp:19:   instantiated from ‘Fac<-1, natural<-1> >’
d.cpp:33:   instantiated from ‘Fac<0, natural<0> >’
d.cpp:33:   instantiated from ‘Fac<1, natural<1> >’
d.cpp:33:   instantiated from ‘Fac<2, natural<2> >’
d.cpp:33:   instantiated from here
d.cpp:19: conversion from ‘
   mpl_::failed************mpl_::assert_relation
   <greater_equal, -1, 0>::************’ to non-scalar
   type ‘mpl_::assert<false>’ requested
d.cpp:19: enumerator value for ‘mpl_assertion_in_line_7’
not integer constant
```

6. Conclusion and future work

In this paper we present a technique to check some annotations in C++ template metaprograms. This technique uses template metaprogramming. The existing programs can be extended to this way easily. With the help of this basic framework we can write safer C++ template metaprograms.

Our aim is to create a more complex library. We will implement more annotations, and support nicer diagnostics. More complex expressions and checks are very important.

References

[1] Alexandrescu, A., Modern C++ Design: Generic Programming and Design Patterns Applied, *Addison-Wesley*, (2001).

[2] Czarnecki, K., Eisenecker, U. W., Glück, R., Vandevoorde, D., Veldhuizen, T. L., Generative Programmind and Active Libraries, *Springer-Verlag* (2000).

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

[4] Karlsson, B., Beyond the C++ Standard Library, An Introduction to Boost, *Addison-Wesley* (2005).

[5] McNamara, B., Smaragdakis, Y., Static interfaces in C++, In *First Workshop on C++ Template Metaprogramming*, (October 2000).

[6] Siek, J., Lumsdaine, A., Concept checking: Binding parametric polymorphism in C++, In *First Workshop on C++ Template Metaprogramming*, (October 2000).

[7] Stroustrup, B., The C++ Programming Language, Special Edition, *Addison-Wesley* (2000).

[8] Vandevoorde, D., Josuttis, N. M., C++ Templates: The Complete Guide, *Addison-Wesley* (2003).

[9] Veldhuizen, T. L., Using C++ Template Metaprograms, *C++ Report*, vol. 7, no. 4, (1995), 36–43.

[10] Veldhuizen, T. L., Expression Templates, *C++ Report*, vol. 7, no. 5, (1995), 26–31.

[11] Veldhuizen, T. L., Gannon, D., Active libraries: Rethinking the roles of compilers and libraries, *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing* (OO'98), SIAM Press, (1998), 21–23.

[12] Boost C++ Libraries, `http://boost.org/`