Towards more reliable C++ template metaprograms*

Ábel Sinkovics
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
abel@elte.hu

Endre Sajó
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
baja@elte.hu

Zoltán Porkoláb
Eötvös Loránd University
Faculty of Informatics
Budapest, Hungary
gsd@elte.hu

Abstract

Writing reliable software is challenging in the domain of C++ template metaprograms. Among other factors, the relatively complex syntax, the lack of good error handling techniques and the shortcomings of existing test methods can be named as sources of unreliability in metaprograms. In this paper we suggest a unit testing framework for C++ template metaprograms that guarantees the execution of all test cases and provides proper summary-report with enhanced and portable error reporting. Neither of these elements serve as a silver bullet for implementing compile-time algorithms, but together they may form an important step towards writing more reliable C++ template metaprograms.

1 Introduction

Writing reliable software presumes a number of components from various areas of the software infrastructure. In the emerging area of C++ template metaprogramming, toolsets for these elements are still in an experimental phase [7]. Reusable libraries [9] and experimental debuggers and profilers [11] are available. Among other aspects, proper error handling and reporting possibilities, and easy to use, versatile test tools play an important role.

Boost.MPL has its own unit tests and testing solution based on compile-time assertions, thus the developer gets non-customizable compiler dependent error messages for failed test cases. These results are difficult to understand for the developers and are hard to process in an automated way. When the code is compiled with a new compiler or a newer version of the same compiler, the format and the content of the error messages may change.

In this paper we suggest a new way to implement a unit testing framework for C++ template metaprograms. The test tool guarantees the execution of all test cases and collects test results – success or failure with reasons – in a run-time data structure. This container is available at runtime and enables customized summary-reporting or integration into other unit testing frameworks. Proper reporting requires compiler-independent pretty printing of types – first class citizens in template metaprograms. These features together provide an orthogonal way for testing metaprograms, i.e. the developer can use his own favorite unit testing tool (like Boost.Test, GoogleTest or CppUnit) accessing the metaprogram test results. We have re-written many of the Boost.MPL test cases as a proof of concept [5].

The rest of the paper is organized as follows. In Section 2 we owerview of the existing unit testing capabilities, especially in the Boost metaprogram library. Using an example we explain the difficulties of the current methods in Section 3. Our novel approach is is introduced in Section 4. We evaluate our method in Section 5. In Section 6 we overview pretty printing of type information and show how to embed the framework into third-party unit testing tools. In Section 6 we discuss error reporting. In Section 7 we evaluate our solution. In Section 8 we overview related works. Our paper concludes in Section 9.

*The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

2 Existing unit testing facility in Boost

In this paper we use the `min` template metafunction [8, 6] as a running example. It takes two arguments and returns the smaller one. `min` compares the arguments using the `less` metafunction. `min` can be implemented the following way:

```
template <class A, class B>
struct min : if_<typename less<A, B>::type, A, B> {};
```

This solution uses `if_` from Boost.MPL and another metafunction, `less`, to compare the values. We expect `min` and `less` to be polymorphic: they should work with any compile-time data that can be compared. When someone develops a new compile-time data type in the future, they should be able to extend these metafunctions to work with his type, as well, without changing existing code.

In Boost.MPL every compile-time value has a member type called `tag` that is used to determine whether that value is a wrapped integral, a list, a vector, etc. There may be more than one implementations of a compile-time data-type, such as wrapped integrals. As long as they provide the same interface and have the same `tag`, all metafunctions operating on the wrapped integrals can deal with them. The `tag` is used to identify that they are wrapped integrals.

`tag` works like the type information of compile-time data-structures. Polymorphic metafunctions in Boost.MPL get the `tag` information of the arguments, instantiate a helper metafunction class [6] with the tags as template parameters and pass the original arguments to that metafunction class. Different specializations of the metafunction class can be created for different tags and tag combinations. Each specialization implements one overload of the metafunction. Since a new specialization can be implemented without changing existing code, new overloads for newly created classes can be implemented later.

Note that the `min` and `less` metafunctions are already implemented in Boost.MPL. In this paper we present how they are currently implemented, what features of them can and can not be tested and how they can be implemented in a way to improve their testability and error reporting.

The basic idea behind the existing compile time unit testing facility of Boost.MPL is to generate syntactically erroneous code on failed assertions and make an effort to decorate the compiler's error messages so relevant information stands out and can be relatively easily identified. To achieve this, the library provides a handful of macros for different assertions and test suite organization. The following assertion macros are available:

- `BOOST_MPL_ASSERT((predicate))` with `predicate` a nullary metafunction: Generate a compilation error when `predicate:: type::value != true`, has no effect otherwise.
- `BOOST_MPL_ASSERT_NOT((predicate))` with `predicate` a nullary metafunction: Generate a compilation error when `predicate:: type::value == true`, has no effect otherwise.
- `BOOST_MPL_ASSERT_MSG(expression, message, (type1,type2,...))` with `expression` an arbitrary integral constant expression, `message` a C++ identifier and `typeN` arbitrary types: Generate a compilation error if `expression != true`, including the `message` itself and the listed types in the error message (effectively exploiting the compiler's own pretty printing capabilities).
- `BOOST_MPL_ASSERT_RELATION(x, relation, y)` with `(x relation y)` a legal C++ expression: Generate a compilation error if `(x relation y) == false`, has no effect otherwise.

For test suite organization, Boost.MPL provides the special macro `MPL_TEST_CASE`, which expands to a unique function name that has the line number of the current expansion as a suffix. Since the function

name in which a particular syntax error occurs is normally output along with the description of the error itself, this can help determine where bodies of failed assertions are located in the original source code.

3 A simple example

Consider the following example in which we try to test `min`. We refer to the `boost::mpl` namespace as `mpl`.

```
MPL_TEST_CASE()
{
  MPL_ASSERT((
    boost::is_same<
          min< mpl::int_<5>, mpl::int_<7> >::type,
          mpl::int_<5>
    >
  ));

  MPL_ASSERT((
    boost::is_same<
          min< mpl::int_<7>, mpl::int_<5> >::type,
          mpl::int_<5>
    >
  ));
}
```

this code compiles without warnings which indicates the snippet has passed our tests, and an executable is created with the sole reason to print on the screen what we already know – “No errors detected.”. For the next snippet we change the expected results to violate the assertions, essentially simulating library bugs.

```
MPL_TEST_CASE()
{
  MPL_ASSERT((
    boost::is_same<
          min< mpl::int_<5>, mpl::int_<7> >::type,
          mpl::int_<6>
    >
  ));

  MPL_ASSERT((
    boost::is_same<
          min< mpl::int_<7>, mpl::int_<5> >::type,
          mpl::int_<6>
    >
  ));
}
```

From this code `GCC` generates the following compiler errors:

```
mintest.cpp: In function 'void test9()':
mintest.cpp:11:3: error: no matching function for call to
 'assertion_failed(mpl_::failed************
 boost::is_same<mpl_::int_<5>, mpl_::int_<6> >::************)'
mintest.cpp:17:3: error: no matching function for call to
 'assertion_failed(mpl_::failed************
 boost::is_same<mpl_::int_<5>, mpl_::int_<6> >::************)'
```

This output can be thought of as a harsh unit test report: the function name `test9` identifies line 9 as the location of the problematic test case declaration, and the "no matching function..." messages pinpoint and describe failed assertions.

Clearly, while arguably usable and helpful, this output is difficult to read even in this trivial example. In fact, it gets worse. When compiled with `Clang`, a compiler with considerably more sophisticated diagnostic capabilities than `GCC`, there is an approximately tenfold growth in output size. The real problem is that while the compiler goes out of its way to produce the most detailed error report, its very efforts are turned into a hindrance as the information artificially injected in the error messages becomes more and more lost in the noise. Another major issue is the non-standard format of the output which means automatic processing would be difficult and certainly not portable.

4 Our approach

The basic principle of our approach is to compile the source, evaluate test cases at compile time and generate an executable that outputs a detailed and customizable report on the results. The exact procedure is as follows. Test cases are written as nullary metafunctions returning wrapped boolean values. These metafunctions are evaluated and, depending on the fail/success statuses, report items are generated utilizing the framework's own type pretty printing facility. Pretty printing is needed to display detailed error messages. For example when a test case compares the result of a metafunction evaluation with the expected value and they differ, the test framework should display the result and the expected value in the report. Our solution achieves this by generating a string at compile-time containing all these details. To generate these, we need to be able to convert classes – data, metaprograms operate on – into strings. A number of similar approaches have addressed this issue so far [10, 3]. Our approach takes Boost.MPL and `tag`s into account. We have implemented our solution as a special metafunction, `to_stream`, which, as a class, provides a static `run` method taking an output stream as its parameter and pretty printing the type to the stream. To use it with a custom type, one has three options.

- `to_stream` can be specialized directly.
- Classes built with `to_stream` in mind can contain an inner type, `to_stream`, with a static `run` method implementing pretty printing.
- For classes with a `tag`, tag-dispatched metafunction class `to_stream_impl` is provided and can be specialized.

The above options are checked in the order they are presented, thus for example the `tag` of a class with a `to_stream` inner class is never checked. The above options for specialization are demonstrated in the following example.

```
struct UDT; // UDT stands for User Defined Type

struct UDT_tag;

struct tagged_UDT { typedef UDT_tag tag; };

struct UDT_with_to_stream
{
  struct to_stream
  {
    static std::ostream& run(std::ostream& os)
    {
      return os << "UDT_with_to_stream";
    }
  };
};

template<>
struct to_stream<UDT>
{
  typedef to_stream type;

  static std::ostream& run(std::ostream& os)
  {
    return os << "UDT";
  }
};

template<>
struct to_stream_impl<UDT_tag>
{
  template <typename Type>
  struct apply
  {
    typedef apply type;

    static std::ostream& run(std::ostream& os)
    {
      return os << "tagged_UDT";
    }
};
```

Test results are collected in STL containers in a test suite hierarchy, the root of which is located in a global test driver object. This hierarchy is accessible through an iterator interface. Test result objects provide their own methods via which fail/success statuses, test names and detailed descriptions can be queried.

Our implementation defines a class for test results with the following public methods:

```
class test_result
{
public:
  // ...
  test_result(bool success, const std::string& reason);

  bool success() const;
  const std::string& get_reason() const;
  // ...
};
```

These methods can be used to query the the fact wether the test case succeeded or not and the pretty-printed reason at runtime. These objects can be constructed by a template function taking the test case as a template argument:

```
template <class F>
test_result run_test()
{
  std::ostringstream s;
  to_stream<F>::run(s);
  return test_result(F::type::value, s.str());
}
```

The function evaluates the test case by accessing its nested type called `type`. The result is expected to be a boxed logical value. To unbox it, the function uses the nested `value` and passes it to the constructor of `test_result`. The function uses `to_stream` to pretty-print the test function and passes the result to the `test_result` constructor.

The test cases are collected into *test suites*. A test suite can contain test cases and other test suites, thus the tests are collected in a tree structure of nested suites. This hierarchy is built at runtime, by the constructors of global objects. We do not present the details of these constructors and the types forming the tree here, they can be found in [5]. This tree can be processed by regular C++ code, the root can be accessed as a singleton object. The tree provides STL-like iterators to process the children of each node. This data-structure can be used to either generate a report directly or to integrate compile-time tests into runtime testing frameworks. In our implementation we provide a number of report generators and a tool that adds compile-time tests to the Boost unit testing framework's test suite hierarchy.

Test cases are registered in the driver by constructors of global objects. Our implementation provides a convenient macro for creating these global objects. It takes two parameters – an object representing the location of the test case in the test suite hierarchy and the name of the metafunction. In-line test case declarations, like the ones which can be defined with the current `BOOST_MPL_ASSERT` macro, are not supported. This approach has a positive side-effect: when the test case contains compilation errors, the diagnostic messages will point to the definition of the test case as opposed to the Boost solution in which they point to the macro call. This difference becomes significant when the test case exceeds a single line, as our approach enables the compiler to locate the failure in the source code more accurately.

The tree structure the test results are collected in is available and can be traversed at runtime. Code based on an unit testingt framework dealing with runtime code can traverse this tree and register the tests results in the runtime unit testing framework. Adapters for different runtime unit-testing frameworks can be developed. Our implementation provides an adapter for Boost.Test.

5 Unit tests for `min` revisited

With the above applied, we get the following test suite for `min`.

```
const suite_path suite = suite_path("sample")("suite");

typedef boost::is_same<
      min< mpl::int_<5>, mpl::int_<7> >::type,
      mpl::int_<5>
  > test1;
ADD_TEST(suite, test1)

typedef boost::is_same<
      min< mpl::int_<7>, mpl::int_<5> >::type,
      mpl::int_<5>
  > test2;
ADD_TEST(suite, test2)
```

By compiling it, the unit tests are executed. By linking it together with an object file containing a `main` function that calls a report generator, an executable is generated that displays the test summary. For example by using the plain text report generator we provide with our implementation, the following output is generated:

```
The following tests have been executed:
  suite::test1: OK
  suite::test2: OK
=======================
Number of tests: 2
Number of failures: 0
```

Now, to simulate library bugs, we modify the suite.

```
typedef boost::is_same<
      min< mpl::int_<5>, mpl::int_<7> >::type,
      mpl::int_<6>
  > test1;
ADD_TEST(suite, test1)

typedef boost::is_same<
      min< mpl::int_<7>, mpl::int_<5> >::type,
      mpl::int_<6>
  > test2;
ADD_TEST(suite, test2)
```

When run the test cases, we expect two failures. This yields the following summary.

```
The following tests have been executed:
  suite::test1: FAIL (mintest.cpp:16)
        is_same<int_<5>, int_<6>>
  suite::test2: FAIL (mintest.cpp:22)
        is_same<int_<5>, int_<6>>
========================
Number of tests: 2
Number of failures: 2
```

Besides being concise, this output no longer depends on the compiler either. The format is standard, but also customizable.

6 Reporting errors

To test a metafunction properly, one has to implement tests verifying the behaviour of the functions for valid and for invalid input as well. When a metafunction breaks the compilation process (which is similar to a unit test causing a core-dump in run time tests), the unit testing framework can not generate a test report. To make metafunctions testable for invalid input, they must not break the compilation process.

`less` is a metafunction taking 2 arguments: the two values to compare. Following the solution Boost.MPL uses to make template metafunctions polymorphic, we define `less` the following way:

```
template <class T1, class T2>
struct less_impl;

template <class a, class b>
struct less : mpl::apply<
  less_impl<
    typename mpl::tag<a>::type,
    typename mpl::tag<b>::type
  >, a, b
> {};
```

We can implement `less` for integrals by specializing `less_impl` for `integral_c_tag`:

```
template <>
struct less_impl<integral_c_tag, integral_c_tag>
{
  template <class A, class B>
  struct apply
  {
    typedef mpl::bool<(A::value < B::value)> type;
  };
};
```

Assume that later we need to develop the following compile-time data-structure to represent rational numbers:

```
struct rational_tag;

template <class Numerator, class Denominator>
struct rational
{
  typedef rational_tag tag;
  //...
};
```

To make rational numbers comparable using `less`, we create a new specialization of `less_impl`:

```
template <>
struct less_impl<rational_tag, rational_tag>
{
  template <class A, class B> struct apply /*...*/;
};
```

This is how `less` is implemented in Boost.MPL. We should create unit tests for `less`. First, we need to make sure that `less` works for wrapped integrals:

```
typedef
  less<mpl::int_<int, 11>, mpl::int_<int, 13> >
  test_less_for_integrals;
```

This compares two integrals using `less` and verifies the result of the comparison. Other cases – such as cases when `less` should return `false` – can be tested in a similar way. We should create another unit test checking that `less` does not work for arguments that can not be compared:

```
less<mpl::list_c<int, 1, 2>, mpl::list_c<int, 3> >
```

Boost.MPL's implementation, we have presented here, emits a compilation error in such cases, since the general case of `less_impl` is not implemented. We have no chance to catch that error in a unit testing environment.

We propose an extension of `less` and template metafunctions in general to return a special value indicating error instead of emitting a compilation error. It can be processed by the code calling the metafunction. Only a top-level metafunction returning error should cause a compilation error. A new compile-time data type can be used to describe errors. Error values contain a description of the error, which is an arbitrary compile-time data structure.

```
struct exception_tag;

template <class Data>
struct exception
{
  typedef exception_tag tag;
  typedef exception type;
};

template <class Exception>
struct get_data;
```

We create a new tag, `exception_tag`, for error values and the `exception` template to represent error values. The custom data an error value holds can be accessed using the `get_data` metafunction. Its implementation can be found in [5]. Using `exception` we can implement the general, non-specialized case of `less`:

```
struct not_comparable;

template <class T1, class T2>
struct less_impl
{
  template <class A, class B>
  struct apply
  {
    typedef exception<not_comparable> type;
  };
};
```

We created a helper class, `not_comparable`, representing the error message. Specializations of `less_impl` can implement comparison and return the results. When there is no specialization, the general implementation takes care of returning the `not_comparable` error. Now we can implement the test checking the behavior of `less` when called with arguments that cannot be compared:

```
template <class X>
struct is_error /* check if X is an exception */;

typedef boost::is_error<
  less<mpl::list_c<int,1,2>, mpl::list_c<int, 3> >
> test_less_with_non_comparable;
```

We can use `less` to implement `min`. `min` takes two arguments, compares them using `less` and returns the smaller one. It is implemented the following way in Boost.MPL:

```
template <class A, class B>
struct min : mpl::if_<less<A, B>, A, B> {};
```

When the arguments are comparable, it works fine. However, when the two arguments can not be compared, `less` returns an `exception`, not a wrapped boolean value, which breaks `if_`. We could expect `if_` to return an error when the condition is not a boolean value, but in that case `min` would return an error indicating that `if_`'s condition was not a boolean value. This error message would hide the original reason that the two arguments cannot be compared. We need to check if `a` and `b` are comparable, return an error when they are not and return the smaller value when they are:

```
template <class A, class B>
struct min : mpl::eval_if<
  typename is_error<less<A, B> >::type,
  less<A, B>,
  mpl::if_<less<A, B>, A, B>
> {};
```

This version of `min` can deal with incomparable arguments, as well, and reports meaningful error messages. These error messages don't break the compilation process. They return errors a unit testing framework can process and add to the generated report.

Following the approach presented above, propagation of errors has to be implemented manually by explicitly checking the return value of every metafunction being called. This is a tedious and error prone process, which can be significantly improved by using monads [12]. A number of different monads exist for dealing with error propagation. Given the complexity of the of topic, we don't cover it in this paper. It is important to note that using the approaches described in [12] for error-propagation require the modification of the metafunctions in the entire chain. However, the paper presents a technique that makes these changes syntactically trivial (wrapping the body of the metafunctions with `try_<...>`). The technique impacts compilation time, however, based on the measurements in the paper, it has a constant cost.

Template metaprograms are compiled and evaluated during the compilation of the C++ code they are embedded into. It makes it difficult to differentiate between the two steps, however the difference becomes significant when the code is tested. Syntax errors of the code being tested or the tests themselves can not be caught by the testing framework. Since syntax errors break the compilation process, the tests are not compiled and not executed, thus no report can be generated in those cases. The testing framework can check the result of assertions during the execution of the metaprograms. The fact that both the compilation and the execution of the metaprograms happen during the compilation of the C++ code may be confusing for the developer.

7 Evaluation

Boost.MPL has a large number of unit tests. They use static assertions and generate compilation errors in case of failed test cases. We have ported a number of the Boost.MPL tests to our framework [5]. We changed the expected results of the test cases to simulate library bugs and run the tests to see the error reports. We have run our tests with three different compilers: gcc 4.5.2, clang 2.8 and Visual C++ 10.

- With the solution based on static assertions we got compilation errors, where the reason was inside long error reports. The length and verbosity of the error reports was different with different compilers. The format of the error report was different with different compilers.

- With the approach presented in this paper the error reports showed the reason of the failure only. The format of the report was the same with every compiler.

We have presented a new approach for implementing template metafunctions which can be tested for success and for failure as well. Following this approach metafunctions return special values instead of breaking the compilation process. Unit testing frameworks can detect these values and generate reports based on them.

8 Related work

A number of unit testing frameworks are available for C++.

- The Boost library collection has a test library for testing C++ code. It focuses on testing runtime C++ code. It supports organizing test cases into a hierarchical structure of test suites. A test case consists of a number of assertions.

- The Boost template metaprogramming library has its own unit tests and uses the Boost test library. Its testing method is based on compile-time assertions, thus the developer gets compilation errors for failed test cases. These compilation errors are difficult to understand and on some compilers overly verbose. The quality of failure reporting is heavily dependent on the compiler.
- Google has a C++ testing framework [4] for testing runtime code. It supports a wide range of assertions with useful diagnostic information in case of failed assertions. It provides tools for displaying useful diagnostic information for new user defined predicates. It also supports writing code for printing user defined data types. These features provide similar functionality to our `to_stream` solution at runtime.
- CppUnit [2] is the C++ port of JUnit [1], a popular unit testing framework for Java. It supports creating test suites and different test cases and it can generate a report about passed and failed test cases. Testing is based on runtime assertions. It supports testing runtime code.

9 Conclusion

Easy to use test tools, proper error handling and reporting is weakly supported in C++ template metaprogramming. In this paper we have presented solutions improving these areas. We have implemented a testing framework with advanced and portable error reporting capabilities. As the results of compile time tests are collected in a runtime data-structure, it can be easily integrated into various popular unit testing frameworks. We have ported many of the Boost.MPL unit tests to our framework and have achieved more portable error reporting. All solutions have been implemented as an open source library [5].

References

[1] Junit project home page, 2010. http://www.junit.org/.

[2] Cppunit project home page, 2011. http://sourceforge.net/apps/mediawiki/cppunit/index.php?title=Main_Page.

[3] Geordi – c++ eval bot, 2011. http://www.xs4all.nl/ weegen/eelis/geordi.

[4] Google c++ testing framework, 2011. http://code.google.com/p/googletest/.

[5] Ábel Sinkovics. The source code of mpllibs, 2010. Available as http://github.com/sabel83/mpllibs.

[6] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[7] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[8] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[9] Aleksey Gurtovoy and David Abrahams. Boost.mpl, 2004. http://www.boost.org/doc/libs/1_47_0/libs/mpl/doc/index.html.

[10] Scott Meyers. Appearing and disappearing consts in c++, 2011. Available as http://cpp-next.com/archive/2011/04/.

[11] Zoltán Porkoláb, József Mihalicza, and Ádám Sipos. Debugging c++ template metaprograms. In *Proceedings of the 5th international conference on Generative programming and component engineering*, GPCE '06, pages 255–264, New York, NY, USA, 2006. ACM.

[12] Ábel Sinkovics and Zoltán Porkoláb. Implementing monads for c++ template metaprograms. Technical Report TR-01/2011, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages and Compilers, September 2011. Available as http://plcportal.inf.elte.hu/en/publications/TechnicalReports/monad-tr.pdf.