Templight, a Template Metaprogram Debugger *

Zoltán Porkoláb, József Mihalicza, Ádám Sipos, Norbert Pataki
gsd@elte.hu, pocok@inf.elte.hu, shp@elte.hu, patakino@elte.hu

Department of Programming Languages and Compilers
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary

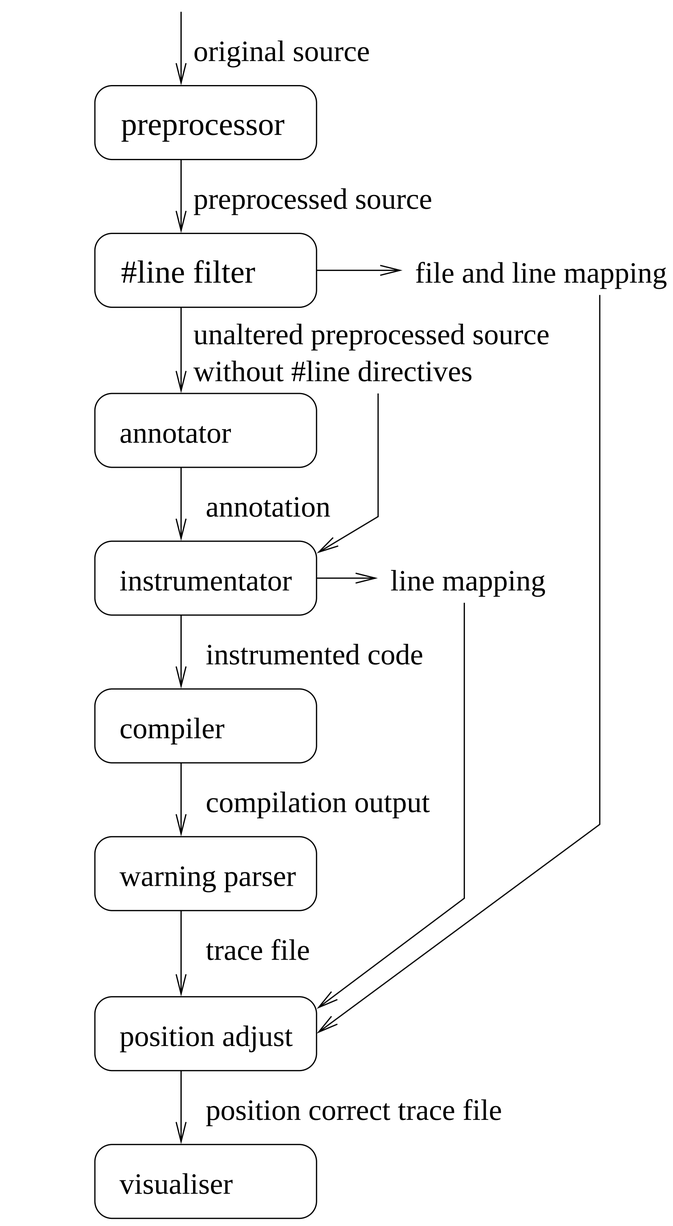
Introduction

Template metaprogramming (TMP) is an emerging new direction in C++ programming for executing algorithms in compilation time. Metaprograms are widely used for the following purposes:

- optimizing runtime programs (e.g. *expression templates*)
- enforcing certain semantic checks.
- emitting error messages
- implementing active libraries

Unfortunately, template metaprogramming is not yet supported with sufficient software tools (e.g. debugger, profiler, etc.). Our poster introduces *Templight*, a debugging framework that reveals the steps executed by the compiler during the compilation of C++ programs with templates. Templight's features include following the instantiation chain, setting breakpoints, and inspecting metaprogram information. This framework aims to take a step forward to help template metaprogramming become more accepted in the software industry.

Architecture of debugging framework



Annotation

Templight generates an XML file to describe the structure of the source file. This file contains annotation entries in a hierarchical structure following the scope:

```
<?xml version="1.0" standalone="yes"?>
<!-- [annotation] generated by Templight -->
<FileAnnotation
  beginpos = "test2.cpp.preprocessed.cpp|1|1"
  endpos = "|1|1">
  <TemplateClassAnnotation
    beginpos = "test2.cpp|1|1"
    endpos = "test2.cpp|5|2"
    afteropenbracepos = "test2.cpp|3|2"
    beforeclosebracepos = "test2.cpp|5|1">
    <ScopeAnnotation
      beginpos = "test2.cpp|4|7"
      endpos = "test2.cpp|4|40"/>
  </TemplateClassAnnotation>
  <TemplateClassAnnotation
    beginpos = "test2.cpp|7|1"
    endpos = "test2.cpp|11|2"
    afteropenbracepos = "test2.cpp|9|2"
    beforeclosebracepos = "test2.cpp|11|1">
    <ScopeAnnotation
       beginpos = "test2.cpp|10|7"
       endpos = "test2.cpp|10|20"/>
  </TemplateClassAnnotation>
  <ScopeAnnotation
    beginpos = "test2.cpp|14|1"
    endpos = "test2.cpp|16|2"/>
</FileAnnotation>
```

Metaprograms

Erwin Unruh created the first C++ template metaprogram in 1994. This program cannot run, but generates error messages when being compiled. Unruh's program produced the prime numbers in the following way:

```
conversion from enum to D<2> requested in Prime
conversion from enum to D<3> requested in Prime
conversion from enum to D<5> requested in Prime
conversion from enum to D<7> requested in Prime
conversion from enum to D<11> requested in Prime
conversion from enum to D<13> requested in Prime
conversion from enum to D<17> requested in Prime
conversion from enum to D<19> requested in Prime
```

- all metaprograms rely on the C++ template mechanism
- with templates we can implement recursions and conditional statements
- this also means that TMP is Turing-complete.

Instrumentation

The instrumentator takes the annotation file and for each element that marks a debug point it inserts a corresponding code fragment that generates a compilation warning containing the desired information (indicated with red color in the following example).

```
namespace Templight {
  template<class C, int C::*>
  struct ReportTemplateBegin {
    static const unsigned Value = -1.0;
  };

  template<class C, int C::*>
  struct ReportTemplateEnd {
    static const unsigned Value = -1.0;
  };

  template<class C, int C::*, class Type>
  struct ReportTypedef {
    typedef Type Result;
    static const unsigned Value = -1.0;
  };
}

template<int i>
struct Factorial
{

  struct _TEMPLIGHT_0s { int a; };
  enum { _TEMPLIGHT_0 =
    Templight::ReportTemplateBegin<
      _TEMPLIGHT_0s, &_TEMPLIGHT_0s::a
    >::Value
  };

  enum { value = i * Factorial<i-1>::value };

  struct _TEMPLIGHT_1s { int a; };
  enum { _TEMPLIGHT_1 =
    Templight::ReportTemplateEnd<
      _TEMPLIGHT_1s, &_TEMPLIGHT_1s::a
    >::Value
  };

};
```

Compilation

The C++ compiler is executed on the instrumented source le to get the talkative error messages. Here we can see the artificially generated warning message that comes from our `Templight::ReportTemplateBegin` template class showing that it is not a warning of the original compilation, but an instrumented one.

```
test2.cpp.patched.cpp(1) : warning C4244:
  'initializing' : conversion from 'double' to
  'const unsigned int', possible loss of data
    test2.cpp.patched.cpp(9) : see reference to
    class template instantiation
    'Templight::ReportTemplateBegin<C,__formal>'
    being compiled
    with
    [
      C=Factorial<4>::_TEMPLIGHT_0s,
      __formal=pointer-to-member(0x0)
    ]
    test2.cpp.patched.cpp(10) : see reference to
    class template instantiation
    'Factorial<i>' being compiled
    with
    [
      i=4
    ]
```

In our following example a compile-time recursion is implemented with a template and its full specialization:

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

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

Metaprogramming is used extensively in a number of popular C++ libraries, like `boost`, `Loki`, `Blitz++`, and others.

Parsing warnings

The warning parser takes the compilation output and looks for our special warning messages and collects the encoded information as well as the position and instantiation history of the event.

```
<TemplateBegin>
  <Position position=
    "test2.cpp.patched.cpp|9|1"/>
  <Context context="Factorial<4>"/>
  <History>
    <TemplateContext instance="Templight::
       ReportTemplateBegin<C,__formal>">
    <Parameter name="C" value=
       "Factorial<4>::_TEMPLIGHT_0s"/>
    <Parameter name="__formal"
       value="pointer-to-member(0x0)"/>
    </TemplateContext>
    <TemplateContext
      instance="Factorial<i>">
      <Parameter name="i" value="4"/>
    </TemplateContext>
  </History>
</TemplateBegin>
```

Profiling template metaprograms

Code efficiency is an all-important aspect of software design. In order to improve the efficiency of a program, a programmer must identify the critical parts. As static analysis methods in many cases fail to explore the dynamical behavior of the program, execution profiling is a key element to finding bottlenecks in the code.

Profilers are software tools carrying out performance analysis by measuring the runtime behavior of programs. *Instrumentation* is a widely used technique in profilers. Instrumentation profilers use *code instrumentation*, they modifiy the analyzed program, inserting profiling code fragments.

Templight is capable of adding timestamps to template instantiations, and measuring their times. So this profiler uses *code instrumentation*, Templight as a profiler does have overhead. Templight's overhead is examined in [2].

Future work

Future work includes the implementation of new IDE extensions providing better functionality for the debugger frontend. This includes the placing and removing of breakpoints, following the compilation process, and examination of the instantiation stack.

Another direction is to create a real, interactive debugger, that can suspend the compilation process at breakpoints, collect information, and is even able to modify the compilation process. As far as we know this can be done only by the modification of the compiler. However, the most adequate modifications and the protocol describing how the compiler and the outer tools interact are currently open.

References

[1] Zoltán Porkoláb, József Mihalicza, Ádám Sipos: Debugging C++ Template Metaprograms, Generative Programming and Component Engineering, *The ACM Digital Library*, pp. 255-264.

[2] Zoltán Porkoláb, József Mihalicza, Norbert Pataki, Ádám Sipos: Towards Profiling C++ Template Metaprograms, The 10th Symposium on Programming Languages and Software Tools (to appear), *Lecture Notes in Computer Science*

[3] David Vandevoorde, Nicolai M. Josuttis: *C++ Templates: The Complete Guide.* Addison-Wesley (2003)

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