Visualization of C++ Template Metaprograms[1]

Zoltán Borók-Nagy, Viktor Májer, József Mihalicza,
Norbert Pataki, Zoltán Porkoláb

Dept. Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary

SCAM 2010

[1] TÁMOP-4.2.1/B-09/1/KMR-2010-0003

Contents

1. Introduction
2. Templight
3. Debugger
4. Visualizer
5. Conclusion

C++ Templates

- Different from Java / C# generics
  - Java / C#: type erasure
  - C++: instantitation
- Mainly used for libraries: STL, etc.
- Templates are skeletons, code generated on demand
- Possibility for specialisation
- Recursive templates are ok

C++ Template Metaprogram - example

```
template <int N>
struct Factorial
{
  enum { Value = N * Factorial<N-1>::Value};
};
template <>
struct Factorial<0>
{
  enum { Value = 1 };
};

int main()
{
  std::cout << Factorial<5>::Value;
}
```

C++ Template Metaprogram features

- Executed at compilation-time
- Functional paradigm
- Why we used them:
  - optimalizations of runtime programs, expression templates
  - static interface checking, concept checking
  - compile-time code adoption, active libraries
  - embedding DSLs
- Turing complete

Motivation

- Metaprogramming is side effect of template construct
- Template syntax is not helpful
- Compiler interprets metaprograms at compilation-time
- No user input, trivial printouts, etc.
- Maintenance is hopeless

Motivation

- Metaprogramming is side effect of template construct
- Template syntax is not helpful
- Compiler interprets metaprograms at compilation-time
- No user input, trivial printouts, etc.
- Maintenance is hopeless

C++ template metaprogram code comprehension tools are essential

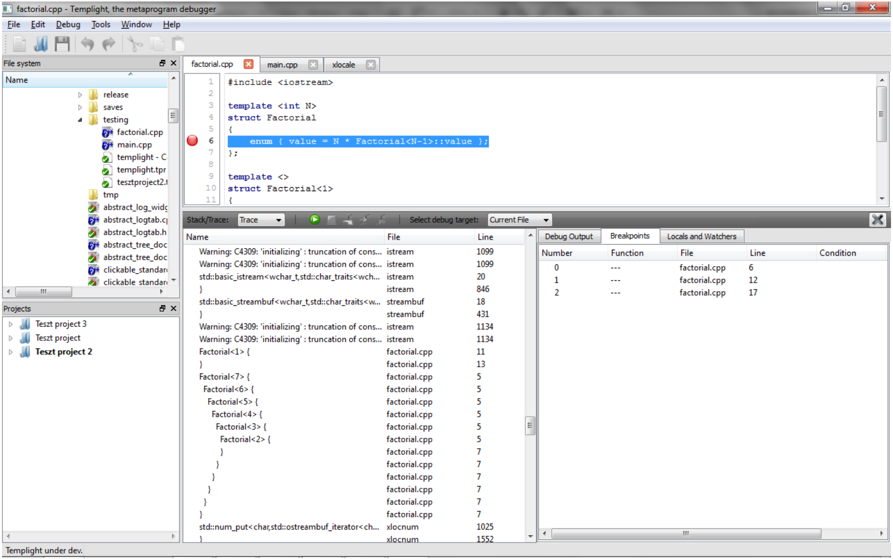
Templight

- Lightweight parser using boost wave and spirit
- Instruments template classes/functions injecting begin/end markers
- Markers emit compilation warnings on instantiation
- Collects warnings generating a "stack-trace"
- Post-mortem way
- Take advantage of compiler dependent implementation details (e.g. *memoization*)

Debugger

- Based on Templight
- GUI is based on QT
- Implements "usual" debugger features:
  - Breakpoints, continue
  - Step in/out/over
  - Locals, watch
- Backward execution

Screenshot



```
#include <iostream>

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

template <>
struct Factorial<1>
{
```

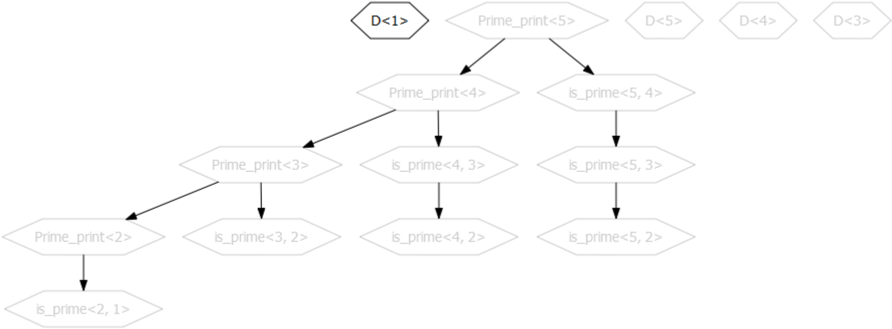
Name	File	Line
Warning: C4309: 'initializing' : truncation of cons...	istream	1099
Warning: C4309: 'initializing' : truncation of cons...	istream	1099
std::basic_istream<wchar_t,std::char_traits<wch...	istream	20
}	istream	846
std::basic_streambuf<wchar_t,std::char_traits<w...	streambuf	18
}	streambuf	431
Warning: C4309: 'initializing' : truncation of cons...	istream	1134
Warning: C4309: 'initializing' : truncation of cons...	istream	1134
Factorial<1> {	factorial.cpp	11
}	factorial.cpp	13
Factorial<7> {	factorial.cpp	5
Factorial<6> {	factorial.cpp	5
Factorial<5> {	factorial.cpp	5
Factorial<4> {	factorial.cpp	5
Factorial<3> {	factorial.cpp	5
Factorial<2> {	factorial.cpp	5
}	factorial.cpp	7
}	factorial.cpp	7
}	factorial.cpp	7
}	factorial.cpp	7
}	factorial.cpp	7
}	factorial.cpp	7
std::num_put<char,std::ostreambuf_iterator<ch...	xlocnum	1025
}	xlocnum	1552

Number	Function	File	Line	Condition
0	---	factorial.cpp	6	
1	---	factorial.cpp	12	
2	---	factorial.cpp	17	

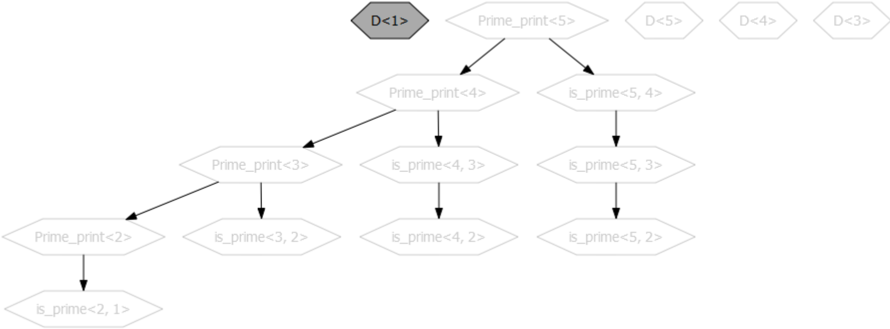
Visualizer

- Based on Templight
- Transform the instantiation chain into a directed graph:
  - nodes: types generated from templates
  - edges show the instantiation requests
- Show corresponding code
- Filter out irrevelant nodes
- Export to png, jpg etc,

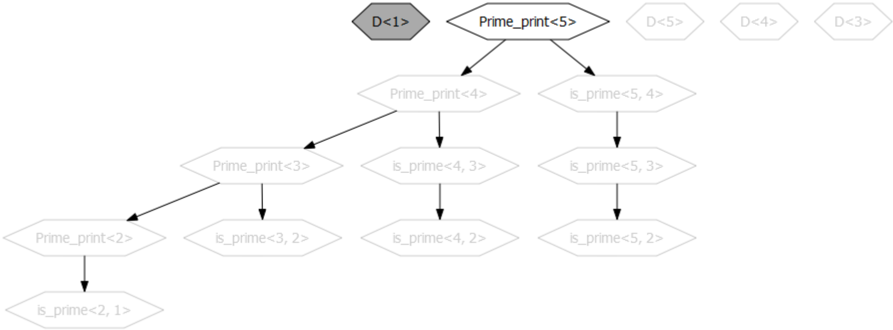
Unruh Example Demonstration



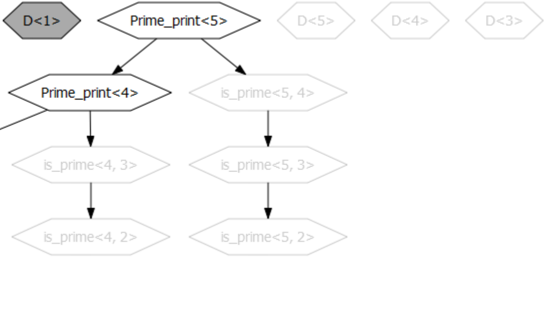
Unruh Example Demonstration



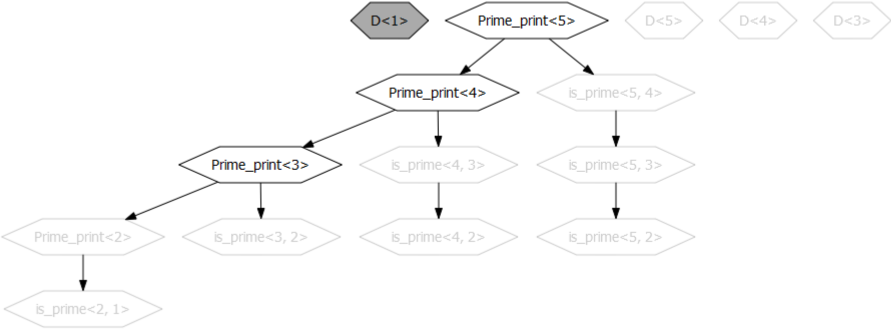
Unruh Example Demonstration



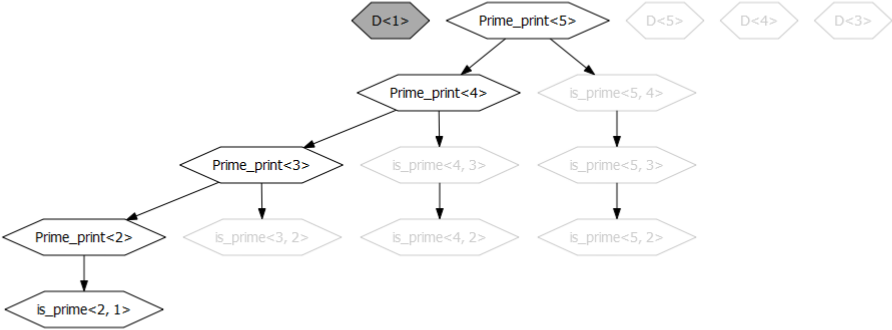
Unruh Example Demonstration



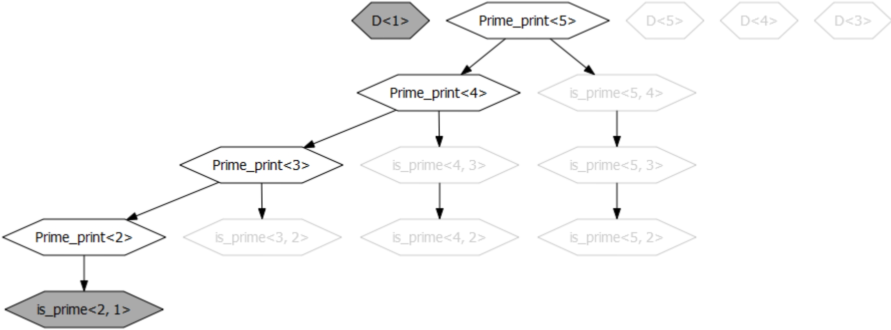
Unruh Example Demonstration



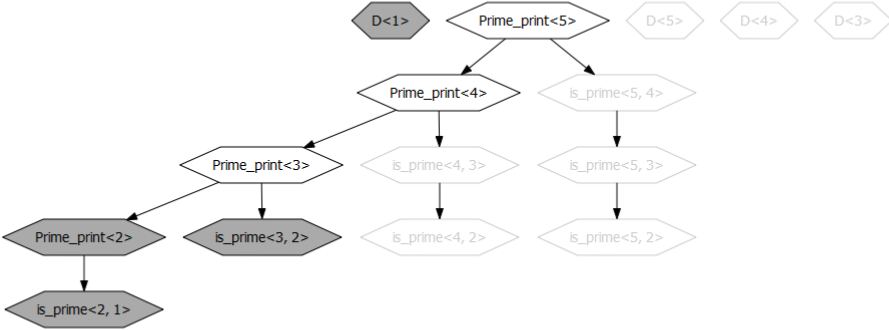
Unruh Example Demonstration



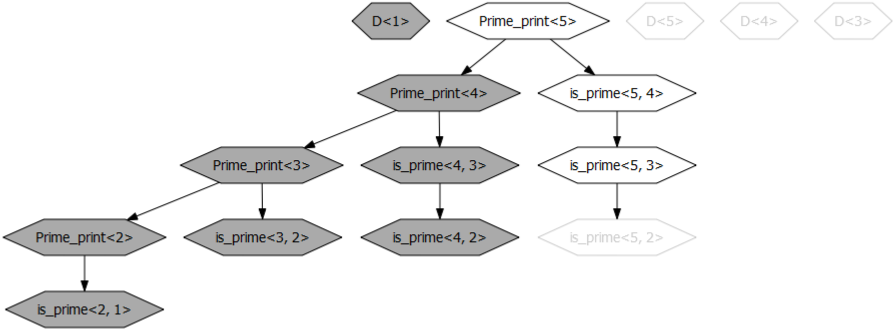
Unruh Example Demonstration



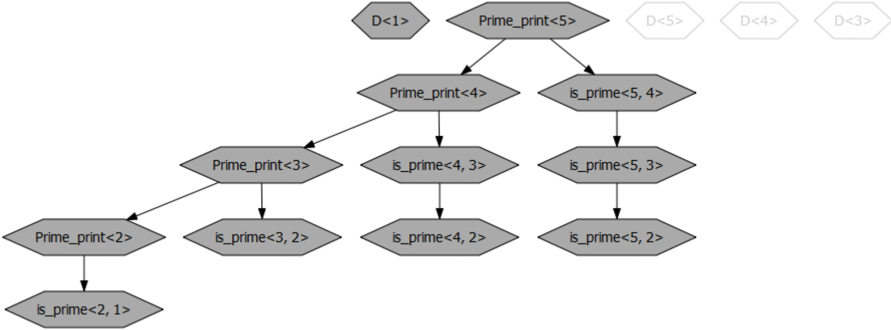
Unruh Example Demonstration



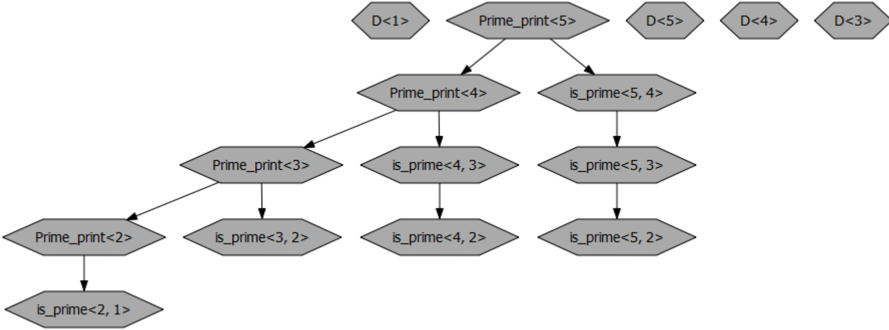
Unruh Example Demonstration



Unruh Example Demonstration



Unruh Example Demonstration



Conclusion

- It is hard to understand and maintain C++ template metaprograms
- Visualization of programs is essential
- We have created a basic framework called *Templight*
- We have developed a graphical user interfaced post-mortem debugger
- We have implemented a tool to visualize the C++ template metaprograms as graphs

Controversial

```
template <int p, int i>
struct is_prime {
  enum {
    prim = (p==2) ||
           (p%i) &&
           is_prime<(i>2?p:0),i-1>::prim
  };
};
template<>
struct is_prime<0,0> {
  enum {prim=1};
};
template<>
struct is_prime<0,1> {
  enum {prim=1};
```

Controversial

C++ source is the assembly of template metaprogram.

We have to use high level functional programming languages, like Haskell, to write metaprograms, and **generate** C++ source.

Controversial

C++ source is the assembly of template metaprogram.

We have to use high level functional programming languages, like Haskell, to write metaprograms, and **generate** C++ source.

Thank you for attention