Expressing C++ Template Metaprograms as Lambda expressions

Ábel Sinkovics, Zoltán Porkoláb

June 3, 2009

Template metaprogramming

- Motivation for templates: generic libraries

Template metaprogramming

- Motivation for templates: generic libraries
- Prime numbers: Erwin Unruh, 1994.

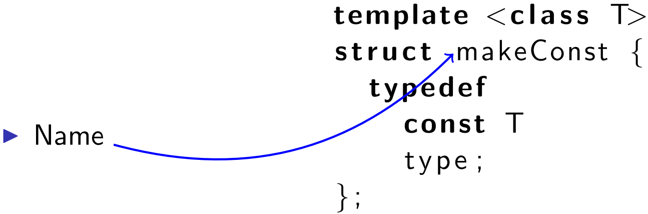
Template metaprogramming

- Motivation for templates: generic libraries
- Prime numbers: Erwin Unruh, 1994.
- Calculation happens at compile time

Template metafunction

```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```

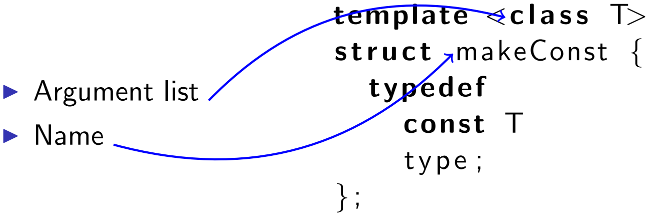
Template metafunction



- Name

```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```

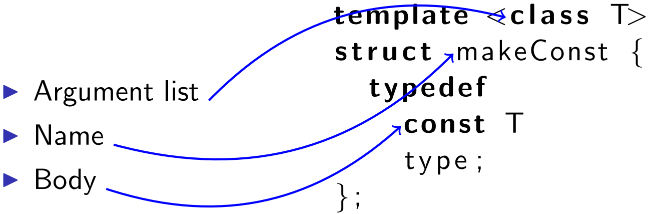
Template metafunction



- Argument list
- Name

```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```

Template metafunction



- Argument list
- Name
- Body

```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```

Template metafunction

- Argument list
- Name
- Body

```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```

Usage:

```
makeConst<int>::type
```

Template metafunction

- Argument list
- Name
- Body

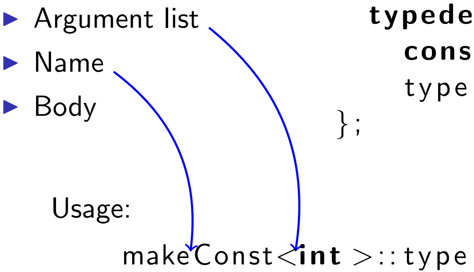
```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```

Usage:

```
makeConst<int>::type
```

Template metafunction

```
template <class T>
struct makeConst {
  typedef
    const T
    type;
};
```



- Argument list
- Name
- Body

Usage:

```
makeConst<int>::type
```

Template metafunction class

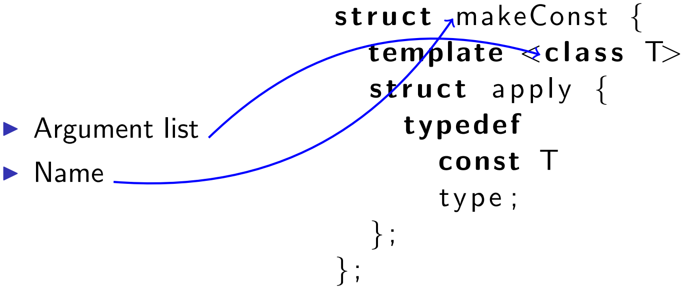
```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```

Template metafunction class

- Name

```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```

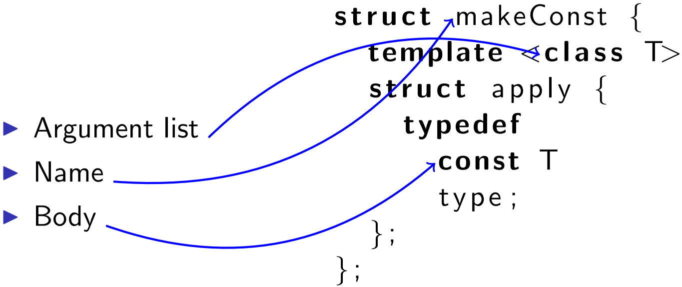
Template metafunction class



- Argument list
- Name

```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```

Template metafunction class



- Argument list
- Name
- Body

```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```

Template metafunction class

- Argument list
- Name
- Body

```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```

Usage:

```
makeConst::apply<int>::type
```

Template metafunction class

- Argument list
- Name
- Body

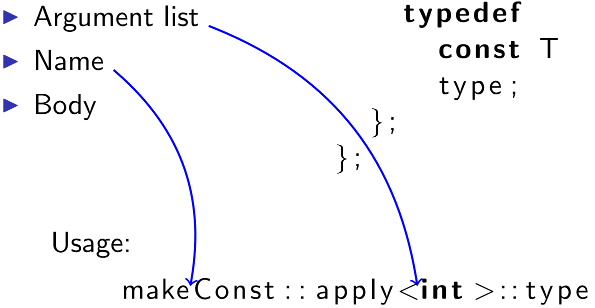
```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```

Usage:

```
makeConst::apply<int >::type
```

Template metafunction class

```
struct makeConst {
  template <class T>
  struct apply {
    typedef
      const T
      type;
  };
};
```



- Argument list
- Name
- Body

Usage:

```
makeConst::apply<int>::type
```

Arithmetic calculation

```
template <int a, int b>
struct plus {
  static const int value =
    a + b;
};
```

Arithmetic calculation

```
template <int a, int b>
struct plus {
  static const int value =
    a + b;
};
```

Usage:

```
plus<5, 8>::value
```

Arithmetic calculation

```
template <int a, int b>
struct plus {
  static const int value =
    a + b;
};
```

Usage:

```
plus<5, 8>::value

plus< plus<2, 3>::value, 8 >::value
```

Connection with functional programming

Capabilities of template metaprogramming:

- Higher order functions

Connection with functional programming

Capabilities of template metaprogramming:

- Higher order functions
- Lazy evaluation

Connection with functional programming

Capabilities of template metaprogramming:

- Higher order functions
- Lazy evaluation
- Pattern matching

Connection with functional programming

Capabilities of template metaprogramming:

- Higher order functions
- Lazy evaluation
- Pattern matching
- Recursion

Fibonacci

```
template <int n> struct fib {
  static const int value =
    fib<n-1>::value + fib<n-2>::value;
};

template <> struct fib<0> {
  static const int value = 1;
};

template <> struct fib<1> {
  static const int value = 1;
};
```

Example

```
namespace { int helper_begin(char*); }
template <int n> struct Print { enum { helper_begin_ = sizeof(helper_begin("")) }; };

template <bool condition, class True, class False> struct If : True {};

template <class True, class False> struct If<false, True, False> : False {};

template <bool b> struct Bool { static const bool value = b; };

template <class a, class b> struct And : Bool<a::value && b::value> {};

template <int from, int to, int n>
struct IsPrimeImpl : If< from <= to, And< Bool<n % qfrom != 0>,
  IsPrimeImpl<from+1, to, n> >, Bool< true > > {};

template <int n> struct IsPrime { static const bool value =
  IsPrimeImpl<2, n/2, n>::value; };

struct Nop {};

template <int n> struct PrintIfPrime : If< IsPrime<n>::value, Print<n>, Nop > {};

template <class A, class B> struct Sequence { A a; B b; };

template <int from, int to> struct PrintPrimes :
  If< from <= to, Sequence< PrintIfPrime<from>, PrintPrimes<from+1, to> >, Nop> {};
```

Maintenance problems

Problems with template metaprogramming:

- Syntax of templates

Maintenance problems

Problems with template metaprogramming:

- Syntax of templates
- No tools supporting template metaprogramming

Using functional syntax

A potential solution for the problems of template metaprograms:

- A functional syntax should be used

Using functional syntax

A potential solution for the problems of template metaprograms:

- A functional syntax should be used
- A tool could transform the functional program to a template metaprogram

Using functional syntax

A potential solution for the problems of template metaprograms:

- A functional syntax should be used
- A tool could transform the functional program to a template metaprogram
- Our solution: use lambda expressions

Our syntax

Syntax of lambda expressions:

```
<expression> ::=
  <constant> |
  <variable> |
  <expression> <expression> |
  \ <variable> . <expression> |
  ( <expression> )
```

Our syntax

Syntax of lambda expressions:

```
<expression> ::=
  <constant> |
  <variable> |
  <expression> <expression> |
  \ <variable> . <expression> |
  ( <expression> )
```

Supported operators:

+, −, ∗, /, %, <, >, <=, >=, <>, =, $

Embedding lambda expressions into C++ code

```
__lambda fact =
  \n. (= n 0) 1 (* (fact (- n 1)) n);

__lambda myLambaExpression =
  + (fact 3) (fact 5);

int main() {
  cout
    << Reduce<myLambdaExpression>::type::value
    << endl;
}
```

Lazy and eager evaluation

- Every (sub)expression is evaluated only when it's value is needed
- Eager evaluation is not supported by embedded lambda expressions

Infinite list

```
__lambda true = \x.\y. x;
__lambda false = \x.\y. y;

__lambda pair = \x.\y.\z. z x y;
__lambda first = \x. x true;
__lambda second = \x. x false;

__lambda cons = \x.\y. pair false (pair x y);
__lambda nil = pair true true;
__lambda head = \x. first (second x);
__lambda tail = \x. second (second x);
```

Currying

The following expressions are equivalent:

```
+ 5
\y . + 5 y
```

Interoperability with native C++ metafunctions

```
struct HandwrittenMetafunction
{
  template <class Argument>
  struct apply
  {
    // implementation
  };
};

__lambda f = \x. \y.
  HandwrittenMetafunction (+ x y);
```

Fixpoint operator and recursion

- Y H → H (Y H)
- Y could be implemented as an embedded lambda expression

```
__lambda Y = \h. \x. h (x x)) (\x.h (x x));
```

- Our solution has a built-in operator: $
- Recursion could be implemented using the fixpoint operator
- Named lambda expressions can use themselves

Constants

- Classes are themselves
- Integral values have a wrapper class

```
template <int n>
struct Integer
{
  static const int value = n;
};
```

Constants

- Classes are themselves
- Integral values have a wrapper class

```
template <int n>
struct Integer
{
  static const int value = n;
};
```

- They can be easily used

```
Integer<13>
```

Constants

- Classes are themselves
- Integral values have a wrapper class

```
template <int n>
struct Integer
{
  static const int value = n;
};
```

- They can be easily used

```
Integer<13>
```

- The boost::mpl library provides wrappers

Lambda abstraction

- Implemented using metafunction classes

```
__lambda I = \x. y;
```

Lambda abstraction

- Implemented using metafunction classes

```
__lambda l = \x. y;
```

- is implemented as

```
struct l
{
  template <class x>
  struct apply
  {
    typedef y type;
  };
};
```

Application evaluation

Lambda abstractions are metafunction classes, application can be implemented as evaluation of these metafunctions. The following application

$$I \quad x$$

Application evaluation

Lambda abstractions are metafunction classes, application can be implemented as evaluation of these metafunctions. The following application

```
    l x
```

is implemented as

```
    l::apply<x>::type
```

Application evaluation

Lambda abstractions are metafunction classes, application can be implemented as evaluation of these metafunctions. The following application

```
l x
```

is implemented as

```
l::apply<x>::type
```

Note that this is **eager application.**

Lazily evaluated application

Our solution uses a helper template:

```
Application<left, right>
```

Lazily evaluated application

Our solution uses a helper template:

```
Application<left, right>
```

It could be used to store an expression tree:

```
Application<
  Application<x, y>,
  z
>
```

Lazily evaluated application

Our solution uses a helper template:

```
Application<left, right>
```

It could be used to store an expression tree:

```
Application<
  Application<x, y>,
  z
>
```

A template metafunction can evaluate it:

```
Reduce< Application<x, y> >::type
```

Currying

- Each lambda abstraction takes one argument
- The body of a lambda abstraction can be another lambda abstraction

Evaluation method

- Implemented the same algorithm directly and using lambda expressions
- The task:
  - find the primes in an interval
  - produce a warning for each
  - prime test: for n test values [2..n/2]

Code size

- Effective Lines of Code
- Direct implementeation: 100% (34 lines)
- Using lambda: 41% (14 lines)

Instantiation depth

- Checking how long the input ([1..k]) has to be to exceed default maximum instantiation depth limit
- Direct implementation: 100% ([1..109])
- Using lambda: 52% ([1..57])

Compilation time

- x axis: the input interval [1..x]
- y axis: compilation time in seconds

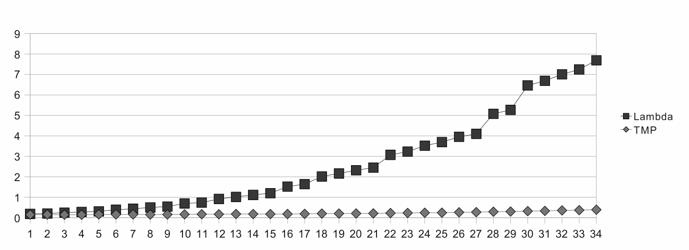


Figure: Compilation time

Debugging

- Error messages emitted by the C++ compiler are about C++ implementation of lambda expressions
- Future work: map the error messages to the original lambda expressions.

Related work

- FC++
- Boost metaprogramming library
- Boost lambda library
- EClean

Summary

- C++ template metaprogramming is a functional sublanguage of C++
- Development and maintainance is difficult
- No supporting tools
- Embdedding lambda expressions to C++ code
- Our tool transforms them to C++ template metafunctions

QA

Questions and answers