Static Metaprogramming in C++

Mateus Krepsky Ludwich
mateus@lisha.ufsc.br
http://www.lisha.ufsc.br

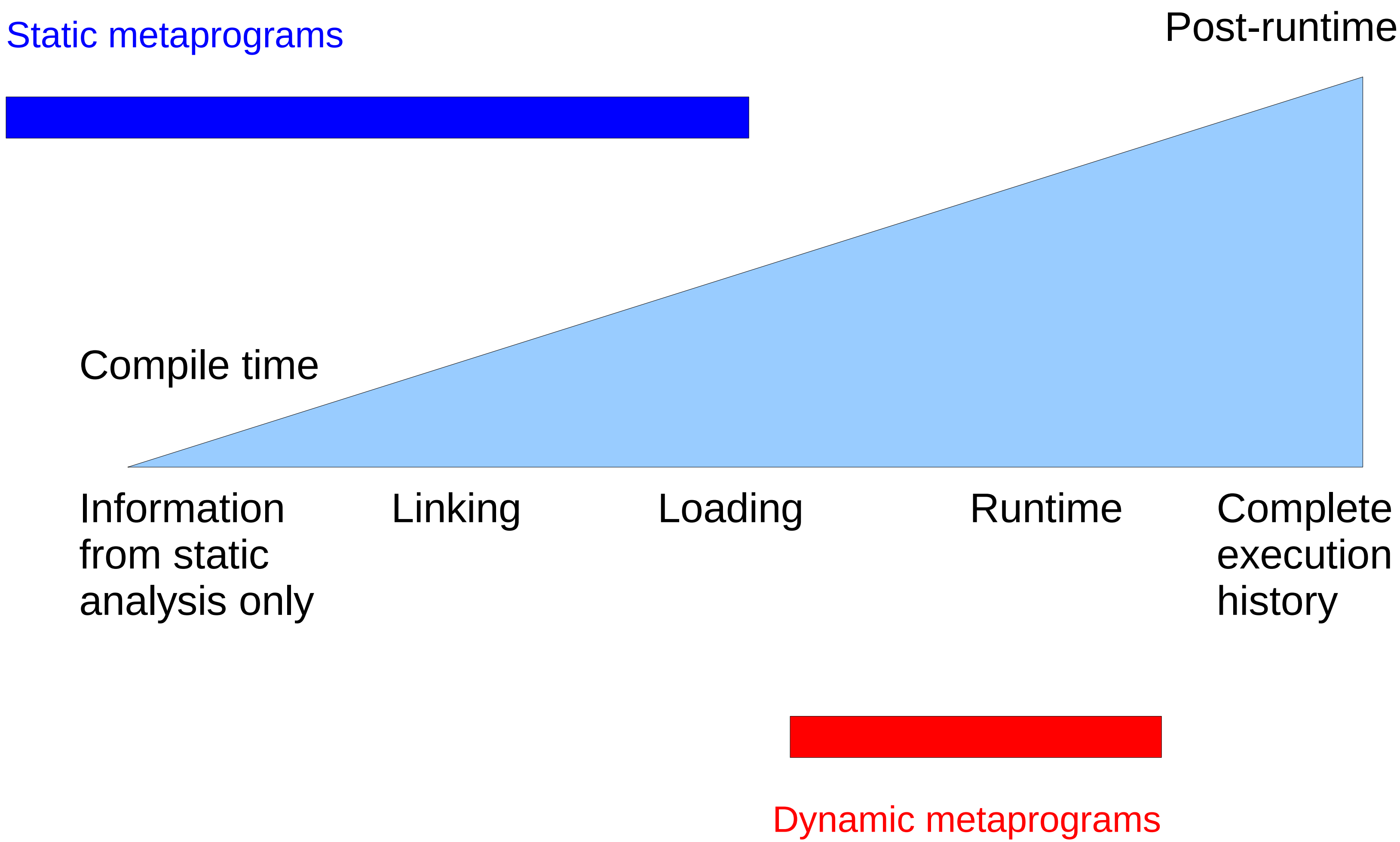
September 27, 2010

Introduction

- Meta: Greek
  - Means: "after" or "beyond"
    - E.g.: Metaphysics, Metapsychology
  - In linguistics: just means "being about" something
    - E.g.: A metalanguage is a language to describe another language

- Metaprograms
  - Programs that represent and manipulate other programs or themselves

Metaprograms execution



Dynamic Metaprogramming

- Reflection
  - *..."The ability of a program to manipulate as data something representing the state of a program during its own execution." [Gabriel, B. W., 1993]*
- Reification
  - Encoding state as data
- Introspection
  - Observe / reason about its own state
- Intercession
  - Modify its own execution state or its own interpretation or meaning

Examples of levels of reflection

- Smalltalk
  - Meta-objects: provide language concepts (methods, classes, execution stacks, the processor) in the form of libraries
  - High level of reflection
- Java
  - Reflection API: used to discover methods and attributes at runtime
  - No direct modification of classes or methods
- C++
  - RTTI: Runtime type information
  - Dynamic cast

Static Metaprogramming

- “Run“ before load time of the code they manipulate – usually compile time
- Most common examples
  - Compilers
    - AST → Assembly
  - Preprocessors
    - Ling1 → Ling2

Types of SMP

- Open Compilers
  - May provide access to its parts (parser, code generator)
    - E.g.: OpenC++, MPC++, Magik, Xroma
  - Transformation systems
    - Provides an interface to write transformations on AST
- Two-level languages
  - Static code: “runs” at compile time
  - Dynamic code: runs at runtimme
  - E.g.: Templates C++

C++ as Two-Level Language

- Static code
  - Templates + other C++ features*
    - * e.g.: conditional operator: “?”
- Dynamic code
  - “ordinary” C++ (all the others constructions and features)

- Static code (subset of C++) is Turing-complete
    - Conditional construction → Template specialization
    - Loop construction → Template recursion

Factorial example

- Dynamic factorial

```
int factorial(int n)
{ return (n==0) ? 1 : n*factorial(n-1); }

void main()
{ cout << "factorial(7)= " << factorial(7) << endl; }
```

Factorial example

- Static factorial

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

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

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}

/* Same effect as:
  cout << "factorial(7)= " << 5040 << endl;
*/
```

“Functional flavor of the static level”

Class templates as functions

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

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

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}

/* Same effect as:
  cout << "factorial(7)= " << 5040 << endl;
*/
```

“Functional flavor of the static level”

Class templates as functions

Integer and types as data

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

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

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}

/* Same effect as:
  cout << "factorial(7)= " << 5040 << endl;
*/
```

“Functional flavor of the static level”

Class templates as functions

Integer and types as data

Template recursion instead of loops

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

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

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}

/* Same effect as:
  cout << "factorial(7)= " << 5040 << endl;
*/
```

“Functional flavor of the static level”

Class templates as functions

Integer and types as data

Template recursion instead of loops

Constant initialization instead of assignment

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

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

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}

/* Same effect as:
  cout << "factorial(7)= " << 5040 << endl;
*/
```

Template Metaprogramming

Metainformation

Metafunction
- Computing numbers
- Computing types
- Generating code

Expression Templates

Template Metaprogramming

- Metainformation
- Metafunction
  - Computing numbers
  - Computing types
  - Generating code
- Expression Templates

Template Metaprogramming

Metainformation

Metafunction

- Computing numbers factorial<>
- Computing types
- Generating code

Expression Templates

Template Metaprogramming

Metainformation

Metafunction

- Computing numbers
- Computing types
- Generating code

Expression Templates

Template Metaprogramming

Metainformation

Metafunction →	Computing numbers
	Computing types IF<>
	Generating code

Expression Templates

Metaprogrammed IF<>

```
template<bool condition, class Then, class Else>
struct IF
{ typedef Then RET;
};

//specialization for condition==false
template<class Then, class Else>
struct IF<false, Then, Else>
{ typedef Else RET;
};

void main()
{
  //...
  IF<(1+2>4), short, int>::RET i; //the type of i is int!
}
```

Template Metaprogramming

Metainformation

Metafunction

- Computing numbers
- Computing types
- Generating code

Expression Templates

Template Metaprogramming

Metainformation

Metafunction

- Computing numbers
- Computing types
- Generating code — Recursive code expansion

Expression Templates

Recursive code expansion

```
/* Dynamic Power -- power(m,n) */
inline int power(const int& m, int n)
{ int r = 1;
  for (; n>0; --n) r *= m;
  return r;
}

/* Looping unrolling for n = 3 */
inline int power(const int& m, int n)
{ int r = 1;
  r *= m;
  r *= m;
  r *= m;
  return r;
}
```

Recursive code expansion

```
power(m,n)
power<N>(m)
```

```
template<int n>
inline int power(const int& m)
{ return power<n-1>(m) * m; }

template<>
inline int power<1>(const int& m)
{ return m; }

template<>
inline int power<0>(const int& m)
{ return 1; }

//test
void main()
{ cout << power<3>(2) << endl;
}

/* Will generate:
    cout << m * m * m << endl;
*/
```

Template Metaprogramming

Metainformation

Metafunction

- Computing numbers
- Computing types
- Generating code

Expression Templates

Template Metaprogramming

Metainformation	Lists and Trees as Nested templates

Metafunction	Computing numbers
	Computing types
	Generating code

Expression Templates

Metaprogrammed list

```
(cons 1 (cons 2 (cons 3 (cons 9 nil))))
```
Lisp

creates:

```
[1, 2, 3, 9]
```

```
Cons<1, Cons<2, Cons<3, Cons<9, End> > > >
```
C++

```
// tag marking the end of a list
const int endValue = ~(~0u >> 1); //initialize with the smallest int

struct End
{ enum    { head = endValue };
  typedef End Tail;
};

template<int head_, class Tail_ = End>
struct Cons
{ enum    { head = head_ };
  typedef Tail_ Tail;
};
```

Metaprogrammed list

```
// Length<>

template<class List>
struct Length
{ // make a recursive call to Length and pass Tail of the list as the argument
  enum { RET = Length<typename List::Tail>::RET+1 };
};

// stop the recursion if we've got to End
template<>
struct Length<End>
{ enum { RET = 0 };
};
```

Template Metaprogramming

Metainformation

Metafunction

- Computing numbers
- Computing types
- Generating code

Expression Templates

Expression templates

- Allows for optimized code generation for adding a number of vectors
  - V4 = v1 + v2 + v3
    - Simply overloading the + operator is inefficient (temporary vector for each +)
- Can be used to implement compile-time domain-specific checks
  - E.g. “an expression cannot contain more than five + operators”
- Useful to implement domain-specific languages

Conclusions

- Metaprogramming
  - Dynamic: reflection
  - Static: template metaprogramming
- Static metaprogramming in C++
  - Turing-complete
  - Metainformation
  - Metafunction
  - Expression templates

References

- Czarnecki, K. and Eisenecker, U. W. 2000 Generative Programming: Methods, Tools, and Applications. ACM Press/Addison-Wesley Publishing Co.
  - Chapter 10: Static Metaprogramming in C++
- Stroustrup, B. 2000 The C++ Programming Language. 3rd. Addison-Wesley Longman Publishing Co., Inc.
  - Chapter 13: Templates

Exercises

- Implementing fibonacci metafunction

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

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

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

void main()
{ cout << "factorial(7)= " << Factorial<7>::RET << endl;
}

/* Same effect as:
  cout << "factorial(7)= " << 5040 << endl;
*/
```