

Metagene, a C++ meta-program generation tool

Francis MAES <francis.maes@lrde.epita.fr>

LRDE seminar, May 21, 2003



Table of Contents

Introducing Metagene	2
Metagene generation paradigm	11
Additional constructions	23
Conclusion	31
References	38
Questions	41

Introducing Metagene

- What is meta-programming?
- What is C++ meta-programming?
- Why is C++ meta-programming bad?
- Why is C++ meta-programming close to functional programming?
- Why Metagene?

What is meta-programming?

- A meta-program is a program that manipulates or generates programs.
- Some well-known meta-programs:
 - Lex and Yacc.
 - GPerf.
- Multi layered evaluation [Tim Sheard and Pasalic \(2000\)](#).
- Two layer evaluation: static and dynamic.

What is C++ meta-programming?

- First, template classes were only created for genericity needs.
- In 1994, Erwin Unruh created the first C++ meta-program.
- Thanks to template classes and template specializations, it is possible to write C++ meta-programs. [Velhuizen \(2002\)](#); [Czarnecki and Eisenecker \(1999\)](#); [Haney and Crotinger \(1999\)](#)
- C++ meta-programs are executed during the C++ compilation (statically).

A very simple C++ meta-program

```
template<unsigned N>
struct fact
{
    enum { res = N * fact < N - 1 >::res };
};
template<>
struct fact<0>
{
    enum { res = 1 };
};

int fact_5 = fact<5>::res;
```

Why is C++ meta-programming bad?

- Heavy syntax.
- Painful error messages.
- Very few typing.
- Grammar inconsistencies (*typename* keyword, partial specialization in nested classes, ...).

Template classes were not designed for meta-programming.

Generating template classes

- We could generate automatically C++ meta-programs. [Crotinger et al. \(2000\)](#)
- Therefore we need to define an input language...
- ...and a program transformation process.

Why is C++ meta-programming close to functional programming?

- No side-effects (or almost not).
- Pattern matching.
- Nested scopes.
- Strict evaluation.

Metagene

```
let rec fact = function
  n -> n * fact (n - 1)
| 0 -> 1
```

is much more readable than

```
template<unsigned N>
struct fact {
  enum { res = N * fact < N - 1 >::res };
};
template<>
struct fact <0> {
  enum { res = 1 };
};
```

Metagene prototype

- The Metagene language is based on Objective Caml. [Weis and al \(1996a\)](#)
- The Metagene transformation uses:
 - OCamlLex for preprocessing.
 - OCamlP4 for parsing. [Weis and al \(1996b\)](#)
 - OCaml code for transformations.
- Metagene is available at <http://www.lrde.epita.fr/>
- All examples presented here are in `tests/seminar`.

Metagene generation paradigm

- Naive transformation.
- Nested structs.
- The box concept.
- Final paradigm.

[1/2] Naive transformation

Let us try to transform the following Caml code:

```
let plus a b = a + b
and incr = plus 1
```

The naive way:

```
template<int a, int b>
struct plus {
  enum { res = a + b };
};
template<int a>
struct incr : public plus<1, a> {
};
```

[2/2] Naive transformation

This way of generating has a major problem: with a different `plus...`

```
let plus a b c = a + b + c
and incr = plus 1
```

...the same `incr = plus 1` is now generated in a different way:

```
template<int a, int b>
struct incr : public plus<1, a, b> {
};
```

This way of generating is highly context-dependent.

[1/2] Nested structs, lambda calculus

In Caml, `let plus a b = a + b` is equivalent to:

```
let plus = function a -> (function b -> a + b)
```

A n -ary function is in reality n nested unary functions. Let us try to apply this model to our generation paradigm:

```
template <int a>  
struct plus {  
  template <int b>  
  struct res {  
    enum { res = a + b };  
  };  
};
```

[2/2] Nested structs, lambda calculus

We can now generate `incr`:

```
template<int a>
struct incr : public plus<1>::res<a> {
};
```

- `incr` generation does not depend anymore on the nature of `plus`.
- This seems to be the right paradigm, but for the moment it only supports integers.

The box concept

Consider the identity function: `function a -> a.`

- Identity works for any type.
- We want to manipulate integers, Booleans, chars, floats, strings, and first order functions in a uniform fashion.

```
template<typename a>  
struct identity {  
    typedef a res;  
};
```

Variable encapsulation in *boxes* that are C++ types.

Integer box

Here is the *box* that encapsulates integers:

```
template <int i>
struct Int {
    enum { value = i };
};
```

Now we can generate our `plus` function:

```
template <typename a>
struct plus {
    template <typename b>
    struct res {
        typedef Int < a::value + b::value > res;
    };
};
```

Function box

Everything can be put in a box including a function: [A. Gurtovoy \(2002\)](#)

```
template <template <typename> typename f >
struct Fun {
    template <class x>
    struct value : public f<x> {
    };
};
template <typename t>
struct incr_implem {
    typedef Int<t::value + 1> res;
};
typedef Fun<incr_implem>  incr;
```

`incr` is now a concrete type encapsulating a template class.

[1/4] Final paradigm

The previous example can be simplified into:

```
struct incr {  
    template<typename t>  
    struct value {  
        typedef Int<t::value + 1> res;  
    };  
};
```

- It works exactly equally except that this new code is shorter.
- The variable names (e.g. `incr`) always correspond to the box.
- A box value is always called `value`; a function result is always called `res`.

[2/4] Final paradigm

An example:

```
let compose a b x = b (a x)
and afunc = compose (( * ) 2) (( + ) 1)
in afunc 25
```

This example is converted into...

[3/4] Final paradigm

```
struct compose {
  template<typename a>
  struct value {
    struct res {
      template<typename b>
      struct value {
        struct res {
          template<typename x>
          struct value {
            typedef a::value < b::value < x >::res >::res    res;
          };
        };
      };
    };
  };
};

typedef compose::value < mtg::plus::value < mtg::Int <1> >::res >::
  res::value < mtg::times::value < mtg::Int <2> >::res >::res    afunc;
typedef afunc::value < mtg::Int <25> >::res    res;
```

[4/4] Final paradigm

Using this *box* paradigm, Metagene currently supports the following Caml elements:

- **Types:** `bool`, `char`, `string`, `int`, variant types (such as `list`), and tuples.
- **Constructs:** function application, `let`, `let-in`, `if-then-else`, `match-with ...`
- **Pervasives:** Almost everything except exception handling.
- **Stdlib:** Parts of the `String` module, most of the `List` module

Additional constructions

- A meta-program is a program that manipulates code.
- This is possible in Metagene using a quotation syntax.
- Compared to OCaml, Metagene programs disposes of two additional builtin types:
 - *cpptype* : A c++ type seen as a value.
 - *cppprim* : A c++ function seen as a value.

C++ types

- *cpptype* values can be created with `<@ a_cpp_type @>`.
- *cpptype* can be matched with the same syntax.
- The *cpptype* builtin type allows to write functions from type to type (such as traits).

C++ primitives

- A *cppprim* value is a C++ method.
- *cppprim* values can be created with
`<@@ parameters @ return_type @ body @>`.
- The `parameters` section is optional.
- *cppprim* is the basis for doing real meta-programming: creating and manipulating code.
- Inside a primitive, the `$ anti-quotation $` syntax allows to refer to Metagene values.

[1/2] Interaction with C++

Two forms of Metagene programs exist:

- Metagene library: only metagene code, generate a C++ header.
- C++ with Metagene:
 - The \$\$ separator allows to switch from one language to the other.
 - The \$ separator allows to access a Metagene value inside C++.
 - The *cpptype* and *cppprim* types allow to express C++ inside Metagene.

[2/2] Interaction with C++

```
#include <mtg.hh>           // include Metagene code
#include <mtg/list.hh> // include standard library
// $$ symbol delimits some Metagene code
$$
let mtg_value = 51
$$
// back in C++, let us convert a
// Metagene value into a C++ variable.
// $ symbol delimits a Metagene value
int i = $mtg_value$;
$$
(* again some Metagene code *)
(* $ and $$ also work inside primitives *)
let mtg_prim = <@@ void @ return $mtg_value$; @>
$$
```

Type promotion example

```
let promote a b =
  if (a = b)
  then a
  else match (a, b) with
    (<@ int @>, <@ char @>) -> <@ int @>
  | (<@ char @>, <@ int @>) -> <@ int @>
  | (<@ int @>, <@ float @>) -> <@ float @>
  | (<@ float @>, <@ int @>) -> <@ float @>
  | (<@ float @>, <@ double @>) -> <@ double @>
  | (<@ double @>, <@ float @>) -> <@ double @>
  | (<@ ntg::cplx<rect, float > @>, <@ float @>) ->
    <@ ntg::cplx<rect, float > @>
  ...
```

C++ primitives example

```
#include <mtg.hh>
$$
let rec numbers = function 0 -> []
  | n -> (numbers (n-1)) @ [n]
let base i = <@@ const float a[], const float b[] @ float @
  return a[$i$] * b[$i$]; @>
let zero = <@@ const float a[], const float b[] @ float @
  return 0; @>
let plus u v = <@@ const float a[], const float b[] @ float @
  return $u$(a, b) + $v$(a, b); @>
let generic_dot n = List.fold_left plus zero (List.map base (
  numbers n))
let dot3 = generic_dot 3
$$
float a[3], b[3];
float d = $dot3$(a, b);
```

Possible Extensions

- *cpptype*:
 - Info functions (is_void, is_array, is_reference...) [Maddock and al \(2001\)](#).
 - Standard types manipulation library (essentially with traits).
- *cppprim*:
 - Addition info (return type, parameter list...)
 - Standard primitive manipulation library (with common primitives, common operators between primitives...)
- Additional builtin type: *cppclass*.

Conclusion

- Performances
- Applicability
- Limitations
- Conclusion

Performances: convolution

Convolution of a 200x200 image with a 3x3 kernel statically known, 50 times. Compiled with Comeau compiler with option -O3.

- **Compilation time**: less than one minute.
- **Fully dynamic** : 2148 ms
- **Metagene** : 580 ms
- **Hand written** : 382 ms

Performances: beternary

The beternary example takes a static list of strings, and generate a perfect matching primitive for these strings (as GPerf [Schmidt \(1989\)](#)).

- Beternary with 3 strings, 1 million calls of the primitive with random strings of length 10:
 - **Fully dynamic** : 399 ms.
 - **Metagene** : 3 ms.
 - **Compilation time** : less than one minute.
 - **GPerf** : 70 ms.
- Beternary with 20 strings.
 - **Compilation time** estimated: more than a week.
 - **Fully dynamic, Metagene, GPerf** : not tested.

Metagene Applicability

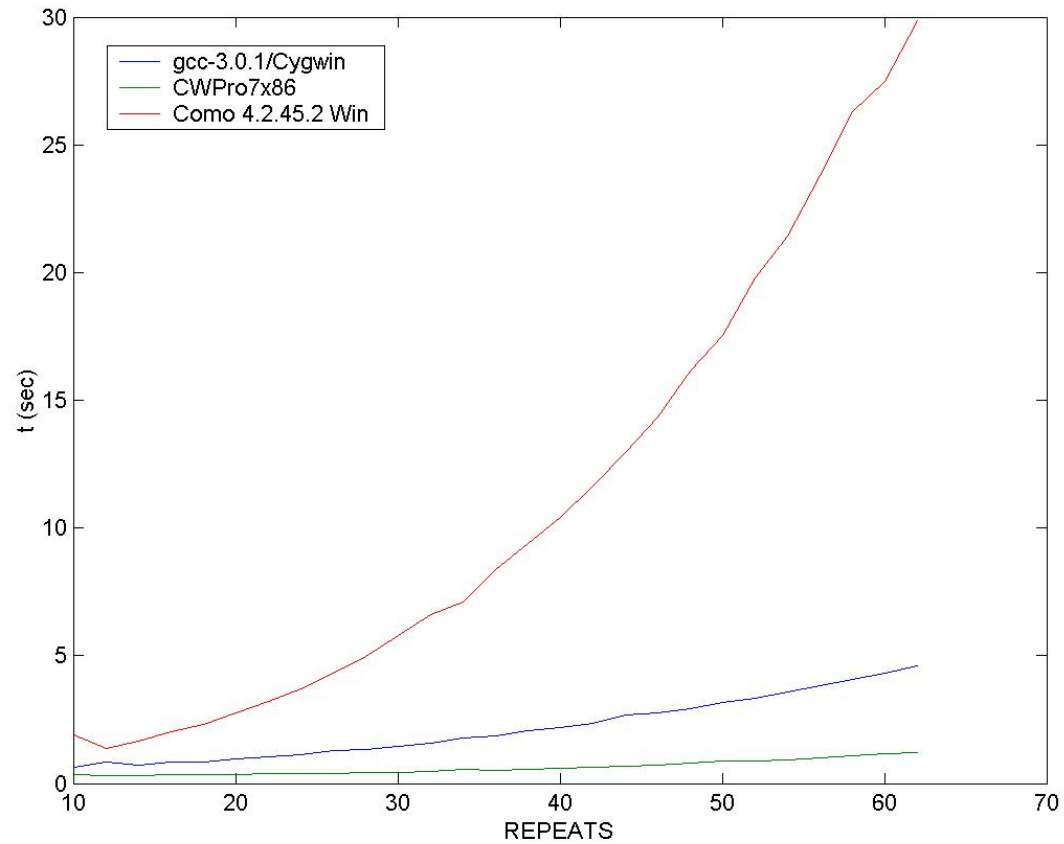
- Traits (functions from type to type) [Velduizen \(2002\)](#).
- Loop unrolling.
- Static data-types processing.
- Static manipulation of ASTs: can be used for C++ functional libraries or for Tiger program compilation [de Guzman and al \(1998\)](#); [McNamara and Smaragdakis \(2001\)](#); [Striegnitz and Smith \(2000\)](#)...

Limitations

- Very slow execution (C++ compilation).
- Huge memory needs.
- Too clean: lots of C++ meta-programs rely on hacks that are impossible to express with Metagene (Think about Expression Templates [Velduizen \(1995\)](#))).
- Complexity increase.

Template classes were not designed for meta-programming.

Limitations: complexity



Conclusion

- △ Much simpler than classe templates uses : anybody can do C++ meta-programming.
- △ Good interaction with C++: can be used for writing libraries, where the user does not need Metagene.
- △ Allows a better understanding of what C++ meta-programming is.
- ▽ Metagene highly suffers of template problems:
 - ▽ Very slow execution.
 - ▽ Huge memory needs.
 - ▽ Non-uniform support by different C++ compilers.
 - ▽ .. (the list is long)

References

A. Gurtovoy, D. A. (2002). The boost c++ metaprogramming library.

Crotinger, J., Cummings, J., Haney, S., Humphrey, W., Karmesin, S., Reynders, J., Smith, S., and Williams, T. (2000). Generic programming in POOMA and PETE. In *Generic Programming, Proceedings of the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 218–. Springer-Verlag.

Czarnecki, K. and Eisenecker, U. (1999). *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley.

de Guzman, J. and al (1998). Spirit parser framework.

References

- Haney, S. and Crotinger, J. (1999). How templates enable high-performance scientific computing in C++. *IEEE Computing in Science and Engineering*, 1(4).
- Maddock, J. and al (2001). The boost type traits library.
- Maes, F. (2003). Metagene, a c++ meta-program generation tool. Technical report, Epita Research And Development Laboratory, Paris, France.
- McNamara, B. and Smaragdakis, Y. (2001). Functional programming in c++ using the fc++ library. *SIGPLAN Notices*.
- Schmidt, D. C. (1989). Gperf, gnu perfect hash function generator.
- Striegnitz, J. and Smith, S. A. (2000). An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*.

References

Tim Sheard, Walid Taha, Z. B. and Pasalic, E. (2000). Metaml.

Velduizen, T. (1995). Expression templates. *C++ Report*, 7(5):26–31.

Velduizen, T. (2002). Techniques for scientific C++. Technical report, Computer Science Department, Indiana University, Bloomington, USA.

Weis, P. and al (1996a). Objective caml.

Weis, P. and al (1996b). Ocamlp4: Pre processor and pretty printer for ocaml.

Questions