Metagene:
a C++ meta-program generation tool

Francis Maes

EPITA Research and Development Laboratory,
14-16 rue Voltaire, F-94276 Le Kremlin-Bicêtre cedex, France,
francis.maes@lrde.epita.fr,
WWW home page: http://lrde.epita.fr/

Abstract. The C++ language offers a two layer evaluation model. Thus, it is possible to evaluate a program in two steps: the so-called *static and* dynamic evaluations. Static evaluation is used for reducing the amount of work done at execution-time. Programs executed statically (called meta-programs) are written in C++ through an intensive use of template classes.
Due to the complexity of these structures, writing, debugging and maintaining C++ meta-programs is a difficult task. Metagene is a program transformation tool which simplifies the development of such programs. Due to the similarities between C++ meta-programming and functional programming, the input language of Metagene is an ML language. Given a functional input program, Metagene outputs the corresponding C++ meta-program expressed using template classes.

1 Introduction

Many languages have of a two-layers evaluation model: some calculations are done at compile-time, some others at execution-time. A typical compile-time operation is constant-folding (where `2 * 3` is replaced by `6` at compilation). More generally, the field of partial evaluation research is focused on performing computations as early as possible in the compilation and evaluation chain.

The C++ language features advanced compile-time evaluation mechanisms. Recursive algorithms can be written using template classes. These *C++ template meta-programs* (Veldhuizen (1995b, 2002)) are entirely executed at compile-time. The classical example of a compile-time *factorial* function is described in figure 1. The template instantiation process allows many common programming structures to be emulated (*e.g.* `if-then-else`, `for-each`, ...). It is possible to implement a Turing machine using C++ template instantiation (Veldhuizen (2002)).

Template classes were created first to fulfill a need of genericity. This is illustrated in the STL (Standard Template Library), where abstractions such as containers and iterators are generic according to the type of the element. However, those abstractions were not designed for writing compile time functions. As a consequence, developing, debugging or maintaining C++ meta-programs is a difficult task. Type checking is quasi inexistent, error messages are generally

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

template<>
struct fact<0>
{
  enum {res = 1};
};

int val = fact<4>::res;
```

The evaluation of `fact 4` is illustrated in appendix A.

Fig. 1. Compile-time factorial in C++

not adapted, some grammar incoherences occur, and so on. Because of these drawbacks, very few C++ developers use meta-programming techniques.

Metagene is a program transformation tool which allows one to write C++ template meta-programs in a friendly functional style. It consists in a translator which transforms functional algorithms into C++ template meta-programs. Using a functional style language is relevant for several reasons. Firstly, the C++ templates instantiation process is very close to the evaluation process of a functional program (pattern matching, nested scopes, no side-effects...). For this reason, most existing template meta-programs are written using a functional paradigm (use of recursive functions, no side-effects, first order functions...). Secondly, many open source implementations of functional languages exist, with more or less reflexivity. Our prototype is built on top of a complete existing architecture: Objective CAML and its powerful preprocessor CAMLp4. Finally, using a functional paradigm allows us to include a real type checking model in our language. This is an elegant solution for solving the lack of type checking in C++ meta-programming.

This paper begins with an overview of related work. Next we introduce our Metagene prototype. Metagene is made of two distinct parts: the language and the program transformation process. This is detailed in section 3 and 4. At that point we describe and study some concrete examples of use. Finally we discuss possible applications and extensions of Metagene.

2 Related Work

The Expression Templates technique (Veldhuizen (1995a)) has a central place in C++ meta-programming. This technique allows to build compile-time abstract syntax trees, what is useful in many C++ meta-programs. This technique

is generally used to transform C++ statements into equivalent (but more efficient) statements. Its implementation in C++ is essentially based on operators overloading and template classes techniques.

Unfortunately, this technique suffers from its perceived complexity. To solve this problem, the PETE project Crotinger et al. (2000) generates all the necessary C++ code for using a particular case of Expression Templates. PETE is highly specialized in this technique, and offers only a few general C++ meta-programming tools. Expression Templates and tools such as PETE can be used to build input data to Metagene programs.

MPL (A. Gurtovoy (2002)) is a C++ template meta-programming framework of compile-time algorithms, part of the Boost package (Boost (2003)). This library provides common compile-time operations such as tests and sequences manipulation, and some simple compile-time types. The code generated by Metagene is very close to the code written when using MPL. Boost also includes two other libraries dedicated to template meta-programming: *static_assert* (compile-time assertions) and *type_traits* (types properties Maddock and al. (2001)).These libraries could be usefull when programming with Metagene.

C++ template meta-programming is now at the heart of highly efficient (Haney and Crotinger (1999)) scientific container libraries like *POOMA* (Crotinger et al. (2000)) - Parallel Object-Oriented Methods and Applications - and *Blitz++* (Veldhuizen and al. (2002)). C++ meta-programming is also much used in functional programming libraries. The *Fact* library (Striegnitz and Smith (2000)), built on top of *PETE*, provides typical functional features such as currying, lambda expressions and lazy evaluation in C++. The Boost package also includes a library specialized in lambda expressions: the Boost Lambda Library (J. Jarvi (2002)). FC++ (McNamara and Smaragdakis (2001)) is a similar library inspired by the Haskell language. Some other libraries are based on C++ meta-programming such as Spirit (The Spirit group (2002)), which uses Expression Templates for building parsers. Most of these quoted libraries could have been written using Metagene.

C++ meta-programming can be used for manipulation of compile-time abstract trees. Using such techniques, we showed in a previous work (Maes (2003)) how to transform fully featured programs written in Tiger (Appel (1997)) into effective C++ code. This work was an extension of usual Expression Templates, and required a large quantity of hard-to-maintain C++ template meta-code.

3 The Metagene language

In the remainder of this paper, Metagene designate two elements: a language, and a program transformation process. Our implementation is based on the Objective CAML language (Weis and al. (1996a)). The program transformation process uses a CAMLLex preprocessor and the CAMLp4 parser (Weis and al. (1996b)). The whole translator is written in Objective CAML.

Some constructs cannot be translated into template meta-programs. The major restriction is that side-effects cannot be expressed with template meta-

programs. Therefore, Metagene cannot handle mutable values, neither records. For the same reasons,object oriented features are also impossible in Metagene. The grammar of Metagene is a subset of the Objective CAML one, that corresponds to a pure functional language: Metagene has no imperative features. Another restriction concerns exceptions, which are not implemented currently in Metagene. However, they could be implemented using compile-time assertions mechanisms.

Metagene includes the most important functional features: functions as value (in parameters or in results), partial application, pattern matching, recursive functions, nested scopes, builtin types (integer, Boolean, string) and variant types (such as lists and trees). The Metagene typing mechanism is the same as the Objective CAML one. Thanks to this proximity, some of the Objective CAML standard libraries are available in Metagene, such as the *list* module, or parts of the *string* module.

Simple *toy* C++ compile-time programs can be written using this simple language core. An example of the factorial function in Metagene is given in figure 2.

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

Fig. 2. Factorial in Metagene

Metagene translates this factorial function into a C++ meta-program equivalent to the one presented in 1.

A meta-program is a program that manipulates or generates programs. Since Metagene is a typed language manipulating C++ pieces of code and C++ types, these structures are seen as particular Metagene values. Thus, compared to Objective CAML, we added two builtin types: *cxxtype* (C++ type seen as a value) and *cxxprim* (C++ piece of code seen as a value). Values of those two types are created with a quotation syntax (`<@ ... @>`).

A *cxxtype* value represents a C++ type. Figure 3 shows some examples of use of this type.

```
let a = <@ int @>
let b = <@ std::string @>
let f = function <@ int @> -> 0 | <@ std::string @> -> 1
```

Fig. 3. Use of C++ types

cxxtype values can be matched, which makes it very easy to write functions from types to types (the so-called traits).

A *cxxprim* value represents a piece of C++ code. Such a piece of code is called a *primitive* in Metagene. It corresponds to a function with its parameters, its return type, and a body. In a primitive, a Metagene value can be referred to by using the '$' symbol. The figure 4 illustrates primitives.

```
let rec power n = function
    n -> let p = power (n - 1) in
                 <@ float x @ float @ return x * $p$(x) @>
  | 0 ->         <@ float x @ float @ return 1 @>
```

This function evaluates a piece of code which computes the power of a float value. For example, using the C++ inlining mechanisms, `power 3` will be evaluated into a primitive equivalent to `float f(float x) {return x * x * x}`. Metagene primitives are the basis for evaluating and creating C++ code at compile-time (Czarnecki and Eisenecker (2000)).

Fig. 4. Use of primitives

Most of the time, Metagene programs are used in a C++ context. Therefore, switching from one language to the other (from one evaluation stage to the other) should be easy. This is done in Metagene with the '$$' separator that allows to switch from C++ to Metagene, and from Metagene to C++. An example that illustrates the interaction between C++ and Metagene is proposed in figure 5.

```
// include Metagene core
#include <mtg.hh>

$$
(* some Metagene code *)
let str = "Hello world!"
let h = <@ void @ void @ std::cout << $str$ << std::endl; @>
$$
// back to C++
int main()
{
  $h$();
  return 0;
}
```

Fig. 5. Hello world!

This interaction with C++ allows two typical Metagene usages:

– Using Metagene code mixed with C++ normal code in any project.
– Writing active libraries, which does not need Metagene to be used.

4 The transformation process

The Metagene translator produces C++ template meta-programs. In this section, we present the key generation rules and conventions that we used in order to make this possible.

Consider the identity function, `function a -> a`. The usual translation of this function into a C++ meta-program is given in figure 6. This implementation of `identity` corresponds to the one proposed in introduction to the Boost Meta-programming Library.

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

Fig. 6. Typical implementation of identity as a C++ meta-program

A function is translated to a template class whose return value is a type, arbitrarily called `res`. A function argument is translated to a template parameter. All Metagene values are stored as C++ types. These are usual conventions when doing C++ meta-programming.

Metagene is a *typed* functionnal language; the identity function can work on any type. Therefore we need to treat integers, Booleans, strings, or functions in an uniform way. The remainder of this section presents our main contribution which solves this problem: functionnal values encapsulation in C++ types. These technique is in the core of Metagene generated code.

Values of type `int` or `char` are directly encapsulated into C++ template classes. The figure 7 shows the example of integers. Characters are treated in a very similar way.

Metagene includes many variant types, such as Boolean and list. This kind of types are handled via the generation of their constructors as empty (template or non-template) classes. This is enough for the construction and the matching of variant type values. This common C++ meta-programming technique is presented in figure 8.

In functional programming, a function is a particular kind of value. Therefore, functions must also be encapsulated into types. The code given in figure 6 cannot be used for this reason: identity must be a *type*. The code really produced by Metagene when translating this function is given in figure 9.

The function identifier is mapped to an encapsulation of the implementation into a C++ class. This allows to see `identity` as a particular case of value.

```
// builtin support for Metagene integers
template<int i>
struct Int
{
  enum {value = i};
};

// integer encapsulation into a C++ type
typedef Int<1> one_t;

// access to the content of an encapsulated integer
int one = one_t::value;
```

Fig. 7. Simple values encapsulation

```
// type 'a list = EmptyList | List of 'a * 'a list
struct EmptyList          {};
template<typename fst, typename snd>
struct List               {};


// let list12 = List(1, List(2, EmptyList))
typedef List<Int<1>, List<Int<2>, EmptyList>> list12;

// let operation = function
//     EmptyList          -> ...
//   | List(head, tail)   -> ...
template<typename T>
struct operation          {};
template<>
struct operation<EmptyList>
{
  // code for the EmptyList case
};
template<typename head, typename tail>
struct operation<List<head, tail>>
{
  // code for the List(head, tail) case
};
```

Fig. 8. Variant values encapsulation

```
// let identity = function a -> a
struct identity
{
  template<typename a>
  struct value
  {
    typedef a res;
  };
};
```

Fig. 9. Function as value

In Metagene just as in Objective CAML, `let plus a b = a + b` is syntactic sugar for `let plus = function a -> (function b -> a + b)`. In these languages, functions with multiples arguments are treated as nested unary functions. Such functions returning functions are generated exactly as any other function. See appendix C, for the translation of the `plus` function.

Here is a short list of some other generation rules:

- Function application is done by accessing the implementation of functions. For example `let g = f a` is translated to `typedef f::value<a>::res g;`.
- A pattern matching is translated to a list of template class specializations (see figure 8).
- C++ types are not changed during their translation: `<@ std::string @>` becomes `std::string`.
- C++ primitives are a particular case of Metagene values. Therefore, they are encapsulated into C++ types, as any other Metagene value. This encapsulation, illustrated in figure 10, corresponds to the object-function technique.

```
//let primitive = <@ int i @ bool @ return i > 0; @>
struct primitive
{
  bool value(int i) { return i > 0; }
};
```

Fig. 10. A translated cxxprim

- Strings are implemented as lists of chars.
- Tuples are implemented as lists of their members.
- Mutual recursion is not supported. The C++ syntax does not allow to predeclare nested template classes, which is needed in the case of mutual recursions. This problem, which is illustrated in figure 11, could be handled by generating a complex particular code in these case.

```
struct u {
  template<typename T>
  struct value {
    // use of v::value
  }
}
struct v {
  template<typename T>
  struct value {
    // use of u::value
  }
};
// Such code is impossible in C++.
```

Fig. 11. Mutual recursion problem

Currently, the code generated by Metagene compiles with Comeau and Icc compilers. Gcc 3.3 is near to be fully supported: it works fine on simple examples, but fails (due to an unimplemented feature of gcc) when it becomes heavy (More than thirty pages of C++ meta-code). This can be excused because of the complexity of the generated code.

5 Examples and results

This section presents two complete programs using Metagene. The first one addresses a common meta-programming issue: unrolling a short loop. More precisely, the goal is to perform an image convolution whose kernel is known at compile-time. A simple example of loop unrolling is given in Appendix D (unrolling of a dot product). The image convolution implementation corresponds to two nested loops unrolling.

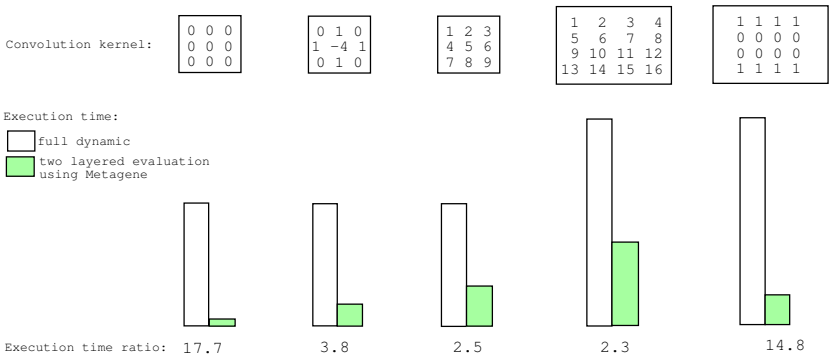

Fig. 12. Image convolution with kernel known at compile-time

Figure 12 compares the execution times of our image convolution for different (more or less complex) convolution kernels. Two techniques are compared: full dynamic evaluation and two-layers evaluation using Metagene. As shown by these results, execution time for an image convolution is significantly improved by using a two-layers evaluation (the simpler the convolution kernel is, the better the resulting primitive is). This can be applied to all applications that know their convolution kernel before execution-time. The convolution primitive evaluation is done in 13 lines of Metagene code. The C++ meta-program generated by Metagene has 160 lines of code. An equivalent C++ meta-program can be written in approximatively 50 (hard to read, hard to debug) lines of code.

Our second example illustrates a typical meta-programming task that was currently almost impossible to implement in C++. The *beternary* program is parameterized by a set of strings, and computes a perfect matching primitive. This corresponds to the action of GPerf (Schmidt (1989)). At execution-time, the generated primitive allows to match exactly the set of predefined strings. Such programs are typically used for lexing a set of keywords.

The beternary source code is given in Appendix E. Writing this program in pure C++ meta-programming (without any tool) is almost impossible for several reasons. Beternary intensively uses the string type, which is not builtin in C++ meta-programming. This is solved by using Metagene string type and its string standard functions. More generally, our program uses a lot of lists which traditionally requires to use heavy static lists machinery. Metagene offers great features for doing that simply (builtin type, standard library, OCaml operators...). Finally the beternary algorithm is much more complex than average C++ meta-programs, which generally do not do more than unrolling a loop.

We have compared the execution times of three techniques: full dynamic evaluation, two-layers evaluation using Metagene and two-layers evaluation using GPerf. Theses techniques have been compared with the following input: `["if"; "then"; "else"; "begin"; "end"]`. For each technique, one thousand random

strings of length 10 were tested. This programs were executed on a K6-350Mhz processor:

- full dynamic: 536 milliseconds.
- using Metagene: 5 milliseconds.
- using GPerf: 101 milliseconds.

On this simple example, the two-layers evaluation using Metagene is much better than the full dynamic evaluation. Thus, thanks to C++ compile-time evaluations, the matching functions was ultra-specialized. On such little examples (very few keywords), Metagene is even better than GPerf.

This primitive ultra-specialization has a cost. However, with more than five strings, the compilation times becomes really important. Figure 13 illustrates the evolution of compile-time with the number of strings (all strings are less than 10 characters long). Two C++ compilers are compared: Comeau 4.3.0.l and Icc 7.0 The compilation-times are exponential: the beternary is realistically limited to seven or eight strings. This is due to the machinery of C++ template classes instantiation. Depending on the implementation of C++ compilers, a simple linear meta-program is often executed in a — more or less — exponential time (Abrahams and Coelho (2001)). In our case, the complexity of the beternary's critical part is quadratic.

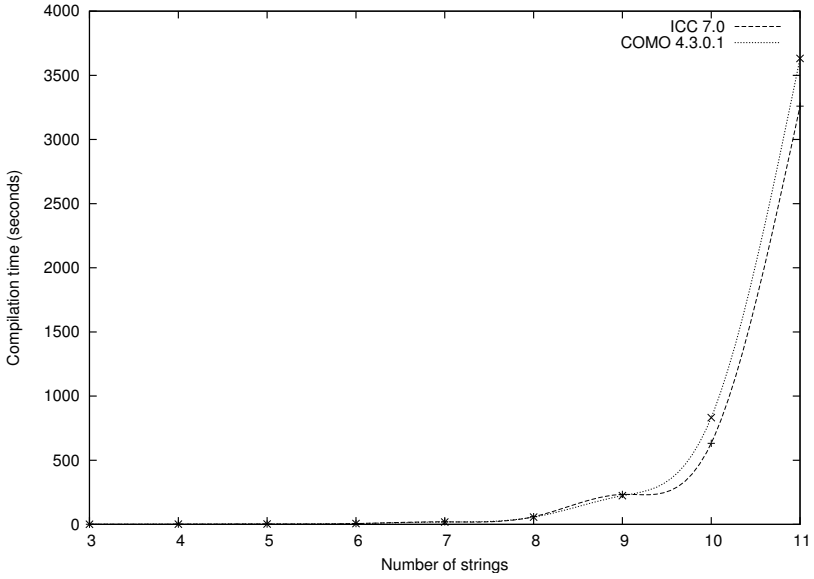

Fig. 13. Beternary compilation times

6 Conclusion

Metagene has the same goal as the Boost Meta-programming Library: simplifying C++ template meta-programs development. The strategy adopted by Metagene is to generate the template classes code. PETE is a tool which generates a particular case of template meta-programs: Expression Templates. Contrary to PETE we want to be able to translate any C++ meta-program.

Some similarities between C++ meta-programming and functional programming led us to build Metagene on top of a functional language. Our implementation is based on Objective CAML and its parser: CAMLp4. In order to do meta-programming with Metagene, we have added two builtin types in the core of the language: cxxtype (C++ type seen as a value) and cxxprim (C++ function seen as a value).

Most important functional features are already implemented. We have shown how our program transformation process was based on value encapsulation in C++ types. The major restriction is that side-effects are impossible in C++ template meta-programming. Therefore, records and mutable fields are not possible in Metagene.

Metagene allows to write C++ meta-programs very easily, without any knowledge of template meta-programming. Metagene syntax is much more adapted to meta-programming than the template classes one. Moreover, Metagene includes some standard libraries such as list and string processing. Using these libraries, complex C++ compile-time operations (*e.g.* sorting a list, manipulating multiple lists or trees, ...) become fully accessible. Thanks to Metagene meta-programming builtin types (*cxxtype* and *cxxprim*), most of common C++ meta-programs, such as type traits or loops unrolling, can be expressed simply in a functional style. Metagene breaks the complexity of C++ meta-programming.

The major restrictions of Metagene are related to the complexity of the template classes instantiation process. This leads essentially to very long C++ compilation times, and a high quantity of memory usage. Simple meta-programs do not raise these problems, but when the complexity becomes higher (quadratic or more), the compilation time becomes a major restriction.

Several improvements could be done on Metagene. Many C++ meta-programs are parametrized by an abstract syntax tree. This can be done with the Expression Templates technique. This feature is not integrated in Metagene yet. To add this feature, we could see how to integrate Metagene with the PETE tool.

Currently, Metagene supports two types designed for C++ meta-programming: *cxxtype* and *cxxprim*. In order to manipulate classes in the same way, we could add a new builtin type *cxxclass*. This way, it would be possible to write functions from class to class, etc... Moreover, Boost meta-programming libraries could be encapsulated into Metagene. For example, their type_traits library provides lots of informative functions about C++ classes, types and functions.

Finally, in order to improve the expression power of Metagene programs, we could generate C++ introspective data when manipulating primitives and classes. This would allow to have a primitive parameters list, or even a list of

C++ classes members or methods. This way, we could write completely generic Metagene programs.

Bibliography

D. A. A. Gurtovoy. The boost C++ metaprogramming library, March 2002. URL http://www.boost.org/libs/mpl/doc/paper/html/index.html/.

D. Abrahams and C. P. Coelho. Effects of metaprogramming style on compilation time, September 2001. URL http://users.rcn.com/abrahams/instantiation_speed/.

A. Appel. *Modern Compiler Implementation in C / Java / ML*. Cambridge University Press, 1997.

Boost. Boost libraries, March 2003. URL http://www.boost.org/.

J. Crotinger, J. Cummings, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, and T. Williams. 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, 2000. URL http://www.acl.lanl.gov/pete/.

K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques and Applications*. Addison-Wesley, 2000.

S. Haney and J. Crotinger. How templates enable high-performance scientific computing in C++. *Computing in Science and Engineering*, 1(4), 1999. URL http://www.acl.lanl.gov/pooma/papers.html.

G. P. J. Jarvi. The boost lambda library, 2002. URL http://www.boost.org/libs/lambda/doc/.

J. Maddock and al. The boost type traits library, 2001. URL http://www.boost.org/libs/type_traits/.

F. Maes. Program templates: Expression templates applied to program evaluation, 2003.

B. McNamara and Y. Smaragdakis. Functional programming in C++ using the FC++ library. *SIGPLAN Notices*, April 2001.

D. C. Schmidt. Gperf, gnu perfect hash function generator, 1989. URL http://www.gnu.org/software/gperf/.

J. Striegnitz and S. A. Smith. An expression template aware lambda function. In *First Workshop on C++ Template Programming, Erfurt, Germany*, October 10 2000. URL http://oonumerics.org/tmpw00/.

The Spirit group. Spirit parser framework, 2002. URL http://spirit.sourceforge.net/.

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

T. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995b. URL http://monet.uwaterloo.ca/ tveldhui/papers/Template-Metaprograms/meta-art.html.

T. Veldhuizen. Techniques for scientific C++. Technical report, Computer Science Department, Indiana University, Bloomington, USA, 2002. URL http://osl.iu.edi/ tveldhui/papers/techniques/.

T. Veldhuizen and al. Blitz++, October 2002. URL http://www.oonumerics.org/blitz/.

P. Weis and al. Objective caml, January 1996a. URL http://caml.inria.fr/ocaml/.
P. Weis and al. Ocamlp4: Pre processor and pretty printer for ocaml, January 1996b. URL http://caml.inria.fr/camlp4/.

A Compile-time factorial in C++

`fact 4` is entirely evaluated during the compilation. The instantiation process is the following:

```
fact < 4 >::res ->
4 * fact < 3 >::res ->
4 * 3 * fact < 2 >::res ->
4 * 3 * 2 * fact < 1 >::res ->
4 * 3 * 2 * 1 * fact < 0 >::res ->
4 * 3 * 2 * 1 * 1 ->
24
```

B Expression templates example

The following arrays statement:

```
A = -B + 2 * C;
```

can be transformed into a single loop of the form:

```
for (i = ... ; ... ; ...)
  A[i] = -B[i] + 2 * C[i];
```

Most of time, this last code is much more efficient.

C Plus function translation

```
// let plus = function a -> (function b -> a + b)
struct plus
{
  template<typename a>
  struct value
  {
    struct res
    {
      template<typename b>
      struct value
      {
         typedef mtg::intplus                    temp1;
```

```
        typedef typename temp1::value<a>::res  temp2;
        typedef typename temp2::value<b>::res  res;
      };
    };
  };
};
```

D A generic dot product

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

float a[10], b[10];
float res = $generic_dot 10$(a, b);
```

E Beternary: perfect matching function

```
(* primitives *)
let prim_const i = <@@ const char* @ int @ return $i$; @>
let prim_false = prim_const (-1)
let prim_char i (c, p) e = <@@ const char* str @ int @
    return str[$i$] == $c$ ? $p$(str) : $e$(str); @>
let prim_switch i = List.fold_right (prim_char i)

(* helper functions *)
let select_strings i c strlist =
  let test (num, str) = ((i < (String.length str)) &&
                         ((String.get str i) = c))
  in List.filter test strlist
let give_indexes strlist =
  let rec give_indexes_ i = function
    [] -> []
  | hd :: tl -> (i, hd) :: (give_indexes_ (i + 1) tl)
```

```
  in give_indexes_ 0 strlist

(* main algorithm *)
let beternary strlist =
  let rec beternary_ i plist =
    let switch_case c =
      (c, beternary_ (i + 1) (select_strings i c plist))
    and next_chars =
      let rec next_chars_ cur = function
          [] -> cur
        | (num, str) :: tl ->
          let c = if
           ((String.length str) <= i) or (List.mem (String.get
             str i) cur)
           then cur
           else ((String.get str i) :: cur)
          in next_chars_ c tl
      in next_chars_ [] plist
    in let recurse () = prim_switch i
        (List.map switch_case next_chars) prim_false
    in match plist with
     [] -> prim_false
   | [ (num, str) ] -> if (i = (String.length str))
        then (prim_const num)
        else (recurse ())
   | _ ->  recurse ()

in beternary_ 0 (give_indexes strlist)

(* use of this C++ meta-program *)
let example = beternary
  ["if"; "then"; "else"; "begin"; "end"]
```

Here is a simple example of use of the beternary meta-program:

```
(* let input = ["a"; "aa"; "ab"; "b"] *)
int output(const char* str)  // after C++ inlining
{
  return str[0] == b
    ? (str[1] == 0 ? 3 : -1)
    : (str[0] == a
      ? (str[1] == b
        ? (str[2] == 0 ? 2 : -1)
        : (str[1] == a
          ? (str[2] == 0 ? 1 : -1)
          : (str[1] == 0 ? 0 : -1)))
      : -1)
}
```