Functional Programming with C++ Template Metaprograms

Zoltán Porkoláb

Eötvös Loránd University, Faculty of Informatics
Dept. of Programming Languages and Compilers
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
gsd@elte.hu

Abstract. Template metaprogramming is an emerging new direction of generative programming: with the clever definitions of templates we can enforce the C++ compiler to execute algorithms at compilation time. Among the application areas of template metaprograms are the expression templates, static interface checking, code optimization with adaption, language embedding and active libraries. However, as this feature of C++ was not an original design goal, the language is not capable of elegant expression of template metaprograms. The complicated syntax leads to the creation of code that is hard to write, understand and maintain. Despite that template metaprogramming has a strong relationship with functional programming paradigm, language syntax and the existing libraries do not follow these requirements. In this paper we give a short and incomplete introduction to C++ templates and the basics of template metaprogramming. We will enlight the role of template metaprograms, some important and widely used idioms and techniques.

1 Introduction

Templates are key elements of the C++ programming language [3, 32]. They enable data structures and algorithms to be parameterized by types, thus capturing commonalities of abstractions at compilation time without performance penalties at runtime [37]. *Generic programming* [28, 27, 17] is a recently popular programming paradigm, which enables the developer to implement reusable codes easily. Reusable components – in most cases data structures and algorithms – are implemented in C++ with the heavy use of templates. The most notable example, the Standard Template Library [17] is now an unavoidable part of professional C++ programs.

In C++, in order to use a template with some specific type, an *instantiation* is required. This process can be initiated either implicitly by the compiler when a template with a new type argument is referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and the generated new code is compiled.

This instantiation mechanism enables us to write smart template codes that execute algorithms at compilation time. To demonstrate the power of C++ templates, in 1994 Erwin Unruh wrote a program [36] which displayed a list of prime

numbers as part of the error messages emitted by the compiler during the compilation process. In fact, Unruh used C++ templates and the template instantiation rules to write a program that is "executed" as a side effect of compilation. It turned out that a cleverly designed C++ code is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [40]. These compile-time programs are called C++ *Template Metaprograms* and later has been proved to be a Turing-complete sublanguage of C++ [8].

The relationship between C++ template metaprograms and functional programming is well-known: most properties of template metaprograms are closely related to the principles of the functional style programming paradigm. On the other hand, C++ has a strong heritage of imperative programming (namely from C and Algol68) influenced by object-orientation (Simula67). Furthermore, the syntax of the C++ templates is especially ugly. As a result, C++ template metaprograms are hard to read, understand and often hopeless to maintain.

The rest of the paper is organized as follows. In Section 2 we give a short informal introduction into C++ template mechanism. In Section 3 C++ template metaprogramming is presented and compared to runtime functional programming. We discuss the fundamental connections between functional programming and C++ template metaprogramming in Section 4. We overview the possible application areas in Section 5 including a complete practical metaprogram example in subsection 5.6. Debugging and profiling are essential for program development. We explain possible techniques in Section 6. In Section 7 we discuss the possibility of pure functional style programming interface for C++ template metaprograms. Related works are presented in Section 8.

2 Informal introduction to C++ templates

Templates are essential part of the C++ language, by enabling data structures and algorithms to be parameterized by types. This abstraction is frequently needed when using general algorithms like finding an element in a data structure, or defining data types like a *matrix* of elements of the same type. The mechanism behind a matrix containing integer or floating point numbers, or even strings is essentially the same, it is only the *type* of the contained objects that differs. With templates we can express this abstraction in one chunk of code, avoiding code duplication, thus *generic* language construct aids code reuse, and the introduction of higher abstraction levels. The method of abstraction over type parameters – often called *parametric polymorphism* – emphasizes that the variability is supported by compile-time template parameter(s).

In the following we give a very informal introduction to templates in the C++ language. We will sometimes simplify the complex rules of templates for the sake of general understanding of the whole mechanism. Those, who are interested in the detailed rules a fundamental source is [37]. For language lawyers the best source is the C++ standard itself [3]. We will be less rigorous in other C++ syntactical rules too, often omitting headers like `<iostream>`, and namespace

tags, like `std::`. For the full, syntactically correct examples see the Appendix 10 notes.

Let us start with a very simple problem: we have to compute the maximum of two parameters – a rather trivial task in most programming languages. However, without some kind of abstraction mechanism over the type of the parameters we soon ended up in a nasty, unmanageable code duplication:

```
// a max function for "int" type
int max( int a, int b)
{
  if ( a > b )
    return a;
  else
    return b;
}

// a  max function for "double" type
double max( double a, double b)
{
  if ( a > b )
   return a;
  else
   return b;
}

// and a lot of other overloadings for other parameter types.
```

While overloading allows us in most modern programming languages to write the correct, type-safe functions, the result is a number of overloaded versions of the `max()` function. Should we modify the algorithm (in a more realistic case), we have to update all of their overloaded instances in a consistent way.

Moreover, we can write overloading functions only for the already defined types. If somebody creates a new type with a well-defined *less-then* operator to compare the objects, we have to write a new overloading version. We cannot implement and compile a `max()` function on type `T` *before* creating `T`, even if we know how that function will look like. Strongly typed programming languages allows writing programs using only existing types.

It is tempting to try out non typesafe solutions. For a C/C++ programmer a precompile macro seems to solve the problem:

```
#define MAX(a,b)    a > b ? a : b
```

As precompiler macro functions are *typeless*, this will work not only for the existing types, but on every type. Unfortunately, precompiler macros are not the answer for writing generic algorithms over types. Precompiler macros are replaced before the run of the C++ compiler, therefore we may encounter a huge

number of side-effects and type-safety problems. Apart from that, the attempt to solve more complex problems with macros is desperate.

To demonstrate this, let us implement a `swap` function, to change the values of two parameters. Here is the trivial solution in C++ for parameters of type `int`:

```
void swap( int& x, int& y)
{
  int temp = x;
  x = y;
  y = temp;
}
```

This is fairly simple. The `&` symbols in the parameter list denotes that the parameter passing should happen by *reference*, therefore `x` and `y` inside the function body yield the original values which we want to swap via the temporary variable `temp`. Variable `temp` should have the same type as the function parameters.

At this point we are in trouble. Since precompiler macros are replaced *before* the C++ compiler itself starts, we cannot use any type inference information from the C++ compiler. We are not able to identify the type of the parameters of the `swap` macro in an automated way [1]. What we need is an intelligent macro-like feature working together with the type system of the C++ language. This language element in C++ is called template.

With templates we are able to write both the `max` and `swap` in a fairly generic way in one code snippet working over different types:

```
template <typename T>
void swap( T& x, T& y)
{
  T temp = x;
  x = y;
  y = temp;
}

template <class T>
T max( T a, T b)
{
  if ( a > b )
    return a;
  else
    return b;
}
```

[1] The new C++ standard, C++0x provides us the `auto` keyword, which will allow us to define a variable of the specific type corresponding to the actual initializer. This is a nice feature, but does not invalidate our message here on the lack of type inference regarding macros.

The `typename` and the `class` keywords are interchangeble in the template definitions and declarations, but we should apply them for all parameters. As an example for the inconsistent syntax, using the `stuct` keyword is invalid here.

It is important to understand that templated `swap` and `max` are not functions in the traditional sense. They are not compiled and they will be not called during the execution of the program. Templates are rather skeletons, describing manufacturing process of real functions instantiated by the compiler in an automated way during the compilation process. Thus we call them: *function templates* rather than *template functions*.

The automated instantiation process is the most remarkable feature of the C++ templates. In the following example we apply this process to the function template `max()`:

```
i = 3, j = 4, k;
double x = 3.14, y = 4.14, z;
const int ci = 6;

k = max(i, j);     // -> max(int, int)
z = max(x, y);     // -> max(double, double)
k = max(i, ci);    // -> max(int, int)
```

The compilation of the above code snippet requires a number of distinct actions from the compiler. In the first step, the compiler has to decide, whether a function template is applicable at the calling sites of `max()`. Then the parameter type(s) should be decided. Parameter types are normally decided on the bases of actual arguments: `i,j,x,y,ci`. This process is called *template parameter deduction*. In our example the first and third call of `max` leads to call an instance of `max(int,int)`, while the second indicates to call `max(double,double)`. These concrete versions of templates are called *specializations*.

When a specialization is not available, the compiler generates it. Thus one `max` function with two `int` parameters, and one with two `double` parameters are created, and will be called. Let us recognize that the first and third call will refer the same specialization. The concrete implementation process may compiler dependent, and later we will see, that we should be extremely careful with such situations.

Which specialization will be called in the following case?

```
z = max(i, x);     // syntax error
```

Under the parameter deduction process, from the type of the argument `i` the compiler supposes the template parameter type `T` to be `int`. However, the second argument `x` contradicts this, suggesting a `double` parameter. Therefore, the parameter deduction process will fail and the compiler raises a syntax error.

How can we fix this problem? As you might expect, templates may be defined with two or more type parameters too. Thus we can provide an other templated `max()`, accepting two different type parameters:

```
template <typename T, typename S>
T max( T a, S b)
{
  if ( a > b )
    return a;
  else
    return b;
}

int     i = 3;
double  x = 3.14;

z = max(i, x);     // -> max(int, double)
std::cout << z << std::endl;
```

At the first sight, everything has been solved. The parameter deduction identifies parameter `T` as `int` and parameter `S` as `double` based on the types of actual arguments `i` and `x`. The instantiation process creates `max(int,double)` specialization, and the right function will be called in run-time.

However, the result printed to the output will be `3` and not `3.14` as we may expect. This is a consequence of the template mechanism we discussed above. When parameters have been decided in the deduction process, also the return value has been determined. Yielded by `T` in the code of `max`, it will be `int` as well as the type of the first parameter. When the function is called in run-time, `a > b` evaluates as false, correctly, and `3.14` is about to return. However, as the return type has been decided in compile time as `int`, this 3.14 value will be converted to integer, and thus we get 3 assigned to `z`. It is clear, that any attempt to change the role of parameters `T` and `S` could lead us to the same problem.

It is also irrelevant, that the return value will be assigned to `z` – a variable of type `double`. Programming languages rarely provide overloading on return types, and never do parameter deduction on them.

Can we construct a better `max()`, a template which returns with the type of the *greater value*? Unfortunately, not in a strongly typed programming language like C++. In such languages, types are fully decided during compile time: in run-time we cannot change them anymore. As templates are totally compile time language features, once the template parameter deduction decides template parameters, these decisions are final. Whether the first or the second argument of the `max(i,x)` call is greater, is completely run-time property. Compilation time and run-time are fundamentally separated in strongly typed, compiled programming languages.

Even if we understand this fenomenon, it may be a bit embarassing. Looking at the actual code it seems natural for the programmer to define `double` as the return type of max function called with an integer and a floating point argument. Programmers understand that `double` is "wider" then `int`. Why we were not able to tell this to the compiler?

The root of the problem is, that when speaking about templates, we have to consider not only two stages – compile time and run-time – of the full process, but also the very first one: the *definition time* of the template function. When we had defined templated `max(T,S)` with two different type parameters `T` and `S`, we had no idea about it's usage environment. We had to decide whether the type `T` or `S` or some other value would be the appropriate return value. At that point, however, we had no information whether the actual arguments in a call environment would be a type of `int`, `double` or something else. We still had to make final decisions.

In the next stage, in compile time, the compiler instantiates the code of `max(i,x)`, with actual `i` and `x` arguments. Now the compiler apprehends the environment of the call, recognises the actual types of `i` and `x`, but cannot overrule the decisions made in template definition time.

Finally, in run-time the program works with the given set of types and rules, and is able only to decide, whether the value of `i` or `x` is the greater, but is unable to overrule the type and the conversion rules regarding the return value.

In the next table we summarized the main stages of programming with templates.

Stage	Template definition	Compilation time	Run-time
Role	Design of algorithms The templated code has been defined	Template instantiation Types used in the program is being decided	Run of the algorithm Program evaluates expressions
Example	Return type of max(T,S) has been decided	Parameter deduction determines T and S	Greater argument value is chosen to return

Table 1. Programming with templates

The two fundamental problems we have: (1) the gap between template definition time and compilation/template instantiation time: this inhibits to choose the "better" return type out of `int` and `double`, and (2) the gap between compilation and run-time: this inhibits to choose the type of the greater value to return. Dynamic and script languages sometimes can help in the second problem. Template metaprograms will give us the power to bridge the first gap in C++.

Before we proceed with template metaprograms, we have to learn some more technicalities on templates.

We may be attempted to improve our `max()` template with a third type parameter, which yields the return type:

```
template <class R, class T, class S>
R max( T a, S b)
{
```

```
  if ( a > b )
    return a;
  else
    return b;
}
```

Unfortunately, the parameter deduction will fail, as there is no information about type `R`. There is a number of reasons why template parameters are not deducted based on return values, but to understand the potential problems consider the following example:

```
int     i = 3;
double  x = 3.14, z;

z = max(i, x);       // (1)
cout << max(i, x);   // (2)
```

As deduction (theoretically) may work in case (1), but there is no reasonable way to choose the correct return type in case (2). However, inventive C++ programmers found the way to smuggle the return type into ordinary arguments, to make it deductible:

```
template <class R, class T, class S>
R max( T a, S b, R)
{
  if ( a > b )
    return a;
  else
    return b;
}

double z = max(i, x, 0.0);
```

The extra argument works, but it is ugly and possibly misleading. The C++ standard committee recognized this requirement, and introduced a syntactically more readable notation:

```
template <class R, class T, class S>
R max( T a, S b)
{
  if ( a > b )
    return a;
  else
    return b;
}

double z = max<double>(i, x);
long   l = max<long, int, long>(i, x);
```

This syntax above is called *explicit specialization*. In the first case `max()` will be instantiated with template parameters: `R=double` given explicitly, and `T=int`, and `S=double` deduced from function arguments. In the second case, all the parameters are given explicitly: `R=long`, `S=int`, and `T=long`. Actual parameter `x` will be converted to `long` as well as the return value. The shortage of this solution is that we have to decide the actual type parameters manually.

We can further specialize templates by eliminating all the template parameters.

```
template <> const char *max( const char *s1, const char *s2)
{
  return  strcmp( s1, s2) < 0;
}

char *s1 = "Hello";
char *s2 = "world";

cout << max(s1, s3);
```

It is clear, that the original algorithm of `max()` would work improper way when comparing the pointer values, rather than the contents of the char arrays. We provided *user specialization* for defining an exceptional behavior of the maximum algorithm for character arrays.

Different template definitions may exist with the same name: overloading of templates are possible. Hence, we may define all previously discussed versions of `max` in the same time.

```
template <typename T> T max(T,T);
template <typename R, typename T, typename S> R max(T,S);
template <> const char *max( const char *s1, const char *s2);
```

When instantiating a call of `max`, the compiler will choose the *most specific* version of template definitions applicable for the actual call.

Up to this point we mainly discussed function templates. Class templates play a similarly important role when implementing abstract data structures, like list, generalized array, matrix, etc. In the rest of this section we will discuss class templates in details as they form the bases of template metaprogramming.

The following code snippet defines a matrix class template. The typename of the matrix elements yielded by `T` is the parameter of the class. Apart from the usual set of constructor, copy constructor, destructor and assignment operator we have methods to retrieve size parameters with `rows`, and `cols` parameters, and accessing elements with the pair of `at` methods.

C++ uses value semantics, i.e. when copying a matrix we have to copy each stored element one by one. We implement the copy semantic with the help of the private `copy` method.

```
template <typename T>
```

```
class matrix
{
public:
  matrix(int i, int j);
  matrix(const matrix &other);
  ~matrix();
  matrix& operator=(const matrix &other);

  int rows() const { return x; }
  int cols() const { return y; }

  T& at(int i, int j);
  T  at(int i, int j) const;

  matrix& operator+=(const matrix &other);
private:
  int  x;
  int  y;
  T   *v;
  void copy(const matrix &other);
};
matrix<T>& matrix<T>::operator+=(const matrix &other);
```

Please consider, that each method of a class template is a function template itself. This seems natural for methods explicitly referring the template parameter, like the `at` method, but also holds for other member functions like `rows()` and `cols()`, they are also templated by `T`.

As object constructors' parameters do not hold relevant information on class template parameters, objects of class templates are instantiated explicitly specifying their type parameters. Here we define matrix objects with type parameter `int`, `double`, and `matrix<double>` respectively:

```
matrix<int>              im;
matrix<double>           dm;
matrix<matrix<double> >  dmm;
```

A possible implementation of the matrix allocates `x*y` objects of type `T` dynamically. This is a fair solution unless `T` is (logically) very small. Allocating an `x*y` length array of type `bool` does not neccessary give what ones expect. In some implementions bool type has size of 4 bytes (for compatibility with `int` type). Even if `sizeof(bool)==1`, we can work out a better implementation storing 8 boolean values on every single byte.

Naturally, this economicial solution may require a totally different representation. Additional attributes, methods, different function bodies should be implemented in *class specialization*.

```
template <>
```

```
class matrix<bool>
{
  // a totally different implementation
};
matrix<bool>& matrix<bool>::operator+=(const matrix &other);
```

The specialization and the original template only share their *names*, otherwise they are considered as separate classes. A specialization does not need to provide the same functionality, interface, or implementation as the original one. It is possible, but generally a very bad idea to change the public interface between specializations.

We have to mention, that not only typenames, but constant expressions of certain types (`bool`, `int`, etc..) are also allowed as template arguments:

```
template <typename T, int SIZE>
class array
{
  T  t[SIZE];
  //...
};
```

With a *partial specialization* we can record one or more type of arguments (like the `int` in the full specialization) or their properties (like being pointer types):

```
template<class T, class U>
class A { ... };

template <class U>
class A<int,U> { ... };
```

This partial specialization will be selected by the compiler if `A` is instantiated with its first argument being `int`.

3 C++ Template Metaprograms

In 1994 Erwin Unruh wrote and circulated at a C++ standards committee meeting a very interesting C++ program. The program was not even compiled, but when the compiler printed error messages, part of them the prime numbers appeared in increasing order.

```
// Erwin Unruh, untitled program,
// ANSI X3J16-94-0075/ISO WG21-462, 1994.

template <int i>
struct D
{
```

```
    D(void *);
    operator int();
};
template <int p, int i>
struct is_prime
{
    enum { prim = (p%i) && is_prime<(i>2?p:0), i>::prim };
};
template <int i>
struct Prime_print
{
    Prime_print<i-1>    a;
    enum { prim = is_prime<i,i-1>::prim };
    void f() { D<i> d = prim; }
};
struct is_prime<0,0> { enum { prim = 1 }; };
struct is_prime<0,1> { enum { prim = 1 }; };
struct Prime_print<2>
{
    enum { prim = 1 };
    void f() { D<2> d = prim; }
};
void foo()
{
    Prime_print<10> a;
}
// output:
// unruh.cpp 30: conversion from enum to D<2> requested in Pri..
// unruh.cpp 30: conversion from enum to D<3> requested in Pri..
// unruh.cpp 30: conversion from enum to D<5> requested in Pri..
// unruh.cpp 30: conversion from enum to D<7> requested in Pri..
// unruh.cpp 30: conversion from enum to D<11> requested in Pri..
// unruh.cpp 30: conversion from enum to D<13> requested in Pri..
// unruh.cpp 30: conversion from enum to D<17> requested in Pri..
// unruh.cpp 30: conversion from enum to D<19> requested in Pri..
```

Erwin Unruh's prime number computing template demonstrated that it is possible to use the C++ template system to write compile-time programs. Such programs are called template metaprograms. A metaprogram is a program that manipulates other programs; for example, compilers, partial evaluators, parser generators and so forth are metaprograms. Template metaprograms are special ones in the sense that they are self-containing: the program which manipulates the code is the C++ compiler itself.

The canonical template metaprogram to show the basic behaviour is the compile time evaluation of factorial numbers. Let us compare a run-time solution and the metaprogram version.

The run-time version is straitforward. Basically the similar code could be implemented in various programming languages from FORTRAN to Pascal.

```
// runtime recursion
int Factorial(int N)
{
  if ( 1 == N )  return 1;
  return         N * Factorial(N-1);
};

int main()
{
  int r = Factorial(5);
  cout << r << endl;
  return 0;
}
```

There are other possibilities to implement the algorithm: especially we may use loop instead of recursion.

The template metaprogram solution takes two template definitions: one for the generic solution of `Factorial`, and an other for the specialization of the parameter `value` 1.

```
// compile-time recursion
template <int N>
struct Factorial
{
  enum { value = N * Factorial<N-1>::value };
}

template<>
struct Factorial<1>
{
  enum { value = 1 };
};

int main()
{
  int r = Factorial<5>::value;
  cout << r << endl;
  return 0;
}
```

Let us analyze what happens here. The `main()` function is used to start the instantiation steps. When the assignment expression refers to `Factorial<5>::value`

the compiler is forced to instantiate the `Factorial` template with argument `5`. As we have a correspondent template definition, the compiler starts the instantiation, and reaches the initialisation of enumeration `value` inside `Factorial`. Here we refer to `Factorial<5>::value`. The instantiation of `Factorial<5>` is suspended and the compiler turns to instantiate `Factorial<4>::value`. This way we imitate recursion, wich will descent down to the instantiation request of `Factorial<1>`. Here the compiler can find a full specialization template for `Factorial` with argument value 1, which is "more specialized" than the generic one. Therefore the full specialization is used to generate the requested class, and instantiation of `Factorial<1>` completes.

From this point we are coming back from the instantiation chain. In this process `Factorial<1>::value` is used to finalize `Factorial<2>`, etc... The suspended instantiations are completed in the reverse order. At the end, we result in generating five classes; four of them instantiated from the generic template definition and one from the template specialization.

As the compiler has `Factorial<5>::value` in hand, it simply replaces the right hand side of the assignment in `main()`. In run-time, we will execute only the output statement. Hence, we "executed" the factorial algorithm – a C++ template metaprogram – in compilation time.

Two important template rules have been tacitly used here: (1) Templates wich are not referred wont be instantiated – C++ template mechanism is *lazy*. (2) Constant expressions – which can be evaluated in compilation time, must be evaluated in compilation time. Such a constant expression appears on the left side of the enumeration initialisation of `value` in class `Factorial`.

Lazyness is essential for writing template metaprograms. Let us consider the following example:

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

template <class Then, class Else>
struct IF<false, Then, Else>
{
  typedef Else RET;
};

int main()
{
  IF< sizeof(int)<sizeof(long), long, int>::RET  i;
  cout << sizeof(i) << endl;
  return 0;
}
```

This seems a bit more criptic than the factorial example. First let's draw up an inventory. We have a generic version of a template called `IF` and a *partial specialization* for it. It is partial, since only one, the leftmost argument has been specialized to `false` boolean value. The first type parameter of the class `IF` is a (constant) value, the remaining arguments are type parameters.

When we instantiate the `IF` template, we provide a boolean expression as the first argument. In our example this is `sizeof(int)<sizeof(long)`. The expression is evaluated in compilation time. If this is `true`, then the generic template is instantiated, and hence the `typedef Then RET` is in effect. With the actual arguments this defines `RET` as `long`. However, when the expression is evaluated as false, we have a "better" specialization, and `typedef Else RET` means `RET` is defined as `int`. As a result, based on whether the size of `int` is smaller than the size of `long`, we define `i` as variable of type of the widest type.

The construct is simmetric – it would be an equally working solution to define the generic function typedefing the `Else` branch, and writing a specialization for the `true` value as the first parameter.

The `IF` construct – the generic template and the specialization – works like a branching metaprogram. Having recursion and branching with pattern matching we have a full featured programming language – executing programs in compilation time. In 1966 Bohm and Jacopini proved, that Turing machine implementation is equivalent to the exsistence of conditional and looping control structures. C++ template metaprograms form a Turing complete programming language executed in compilation time [42].

Now we can revisit the `max()` function we discussed about earlier:

```
template <class T, class S>
IF< sizeof(T)<sizeof(S), S, T>::RET max(T x, S y)
{
  if ( x > y )
    return x;
  else
    return y;
}
```

This version of `max()` is able tho choose the "widest" of the argument types and defines it as the return type. In the template definition we did not committed ourselves to the return type. Instead of choosing one of the argument types, we defined a small metaprogram which will be executed in compilation, i.e. template instantiation time. When the template is instantiated the actual types of `T` and `S` are known and the metaprogram is evaluated. Either `T` or `S` will be selected as `typedef` of `IF<...>::RET`, based on the metaprogram's algorithms.

When this template is instantiated with argument types `int` and `double`, the return value will be `double`. Similarly, when the arguments are `short` and `long`, the later will be chosen as the return type.

It is important to understand two facts. First, we cheated a bit. The "widest" type – which has the greater `sizeof` value – is not always the best return type.

Sometimes the size of a class is unrelated to the arithmetical representation – this is true especially for classes allocating extra space in the heap. But conceptually this is not a problem for us: anyway, we are in a Turing complete language, so we are able to define as complex algorithms as we wish.

Second, we were still not able to choose the type of the greater value, we have chosen the type which seemed better under compilation. It is still possible that `double` has been chosen as return type, but the `int` run-time value is greater. In such a situation the return type value will be converted to `double`.

In other words, we are not breaking the rules of strongly typed programming languages. Types are not selected in run-time. What we added to the earlier version of `max` is the possibility of selecting the return type not in template definitin/design time, but later, in compilation time, when the template is instantiated. We delegated an algorithm written in design time, executed in compilation time which – based on the actual types of the template arguments – was able to select the better return type. This has happened in an automated way by the execution of a small and simple template metaprogram.

Stage	Template definition	Compilation time	Run-time
Role	Design of algorithms The templated code has been defined	Template instantiation Parameter deduction happen Metaprograms are executed	Run of the algorithm Program evaluates expressions
Example	Return type of max(T,S) defined with metaprogram	Parameter deduction determines T and S. `IF<T,S>::RET` is selected	Greater argument value is chosen to return

Table 2. Programming with template metaprograms

4 Connection between functional programming and C++ template metaprograms

In our context the notion *template metaprogram* stands for the collection of templates, their instantiations, and specializations, whose purpose is to carry out operations in compile-time. Their expected behavior might be either emitting messages or generating special constructs for the runtime execution. Henceforth we will call a *runtime program* any kind of runnable code, including those which are the results of template metaprograms. Executing programs in either way means executing pre-defined actions on certain entities. It is useful to compare those actions and entities between runtime programs and metaprograms.

C++ template metaprogram actions are defined in the form of template definitions and they are "executed" when the compiler instantiates them. Templates can refer to other templates, therefore their instantiation can instruct the compiler to execute other instantiations. This way we get an instantiation chain

very similar to a call stack of a runtime program. Recursive instantiations are not only possible, but regular in template metaprograms to model loops.

In metaprograms we use `static const` and enumeration values to store quantitative information. Results of computations during the execution of a metaprogram are stored either in other constants or enumerations. Furthermore, the execution of a metaprogram may trigger the creation of new types by the compiler. These types may hold information that influences the further execution of the metaprogram [44].

However, there is a fundamental difference between usual runtime programs and C++ template metaprograms: once a certain entity (constant, enumeration value, type) has been evaluated or constructed, it will be immutable. There is no way to change its value or meaning. When we initialized a constant or enumeration we are not able to change its value. When a type has been constructed, it is not possible to redefine it. Therefore metaprogram assignment does not exist. In this sense metaprograms are similar to pure functional programming languages, where *referential transparency* is obtained. That is the reason why we use recursion and specialization to implement loops: we are not able to change the value of any loop variable. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur.

Based on this observations we can say that C++ template metaprogramming is part of the functional programming paradigm. In the following table we summarized the main similarities, tools, and language features.

	Runtime functional program	C++ template metaprogram
values	run-time data (constant, literal)	static const and enum class members
variables	variables	symbolic names (typenames, typedefs)
initialization	constants generators	static const initialisation enum definition
assignment	no	no
i/o helpers	monads	warnings, error messages no interactive input
branching	pattern matching function specialization	pattern matching template specialization
looping	recursive functions	recursive templates
subprogram	function	(template) class
data types	abstract data structures	typelists, boost::vector
types	type class (Haskell)	concepts

Table 3. Comparison of functional programs and template metaprograms

Abrahams and Gurtovoy [1] defined the term template metafunction as a special template class: the arguments of the metafunction are the template parameters

of the class, the value of the function is a nested type of the template called `type`.

Metafunctions – as we can expect in a functional programming language – are first class citizens in C++ template metaprogramming. In the following example we show a metaprogram `Accumulate` which summarizes the value of a function given as a parameter at points in the intervall 0..N. The function will be a metaprogram itself, and it can be specified as an argument of `Accumulate`.

```
// Accumulate(n,f) := f(0) + f(1) + ... + f(n)

template <int n, template<int> class F>
struct Accumulate
{
  enum { RET = Accumulate<n-1,F>::RET + F<n>::RET };
};

template <template<int> class F>
struct Accumulate<0,F>
{
  enum { RET = F<0>::RET };
};

template <int n>
struct Square
{
  enum { RET = n*n };
};

int main()
{
  cout << Accumulate<3,Square>::RET << endl;
  return 0;
}
```

Previous examples show that there are sophisticated ways to build up, pass as parameter, and execute functions in compilation time. We have similar professional tools to express lists, vectors, etc. as compile time data structures.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favourite implementation forms of expression templates [41]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [50].

We define a typelist with the following recursive template:

```
class NullType {};
struct EmptyType {};        // could be instantiated
```

```
typedef Typelist< char, Typelist<signed char,
            Typelist<unsigned char, NullType> > > Charlist;
```

In the example we store the three character types in a typelist. We can use helper macro definitions to make the syntax more readable.

```
#define TYPELIST_1(x)           Typelist< x, NullType>
#define TYPELIST_2(x, y)        Typelist< x, TYPELIST_1(y)>
#define TYPELIST_3(x, y, z)     Typelist< x, TYPELIST_2(y,z)>
#define TYPELIST_4(x, y, z, w)  Typelist< x, TYPELIST_3(y,z,w)>
// ...
typedef TYPELIST_3(char, signed char, unsigned char)  Charlist;
```

Essential helper functions – like `Length` to compute the size of a list in compile time – have been defined in the Alexandrescu's Loki library[2] in pure functional programming style. Let us consider the typical template metaprogram components. We began with the declaration of the one type-parmeter `Length` template. This is followed by the specific version of `Length` applicable for the empty list as a specialization for `NullType`. This template will be instantiated only at the end of a typelist. Finally, we define the generic case on template parameter `Typelist<T,U>` with further recursion on `U`.

```
/
//  Length
//
template <class TList> struct Length;
template <>
struct Length<NullType>
{
    enum { value = 0 };
};
template <class T, class U>
struct Length <Typelist<T,U> >
{
    enum { value = 1 + Length<U>::value };
};
```

Length reads the size of the list. The `IndexOf` metafunction takes a type parameter and returns the position of that parameter in the list. If the actual argument is not found in the list it returns the value -1.

```
template <class TList, class T> struct IndexOf;

template <class T>
struct IndexOf< NullType, T>
{
```

```
    enum { value = -1 };
};
template <class T, class Tail>
struct IndexOf< Typelist<Head, Tail>, T>
{
private:
    enum { temp = IndexOf<Tail, T>::value };
public:
    enum { value = (temp == -1) ? -1 : 1+temp };
};
```

Similar data structures and algorithms can be found in `boost::mpl`.

5 Applications of template metaprogramming

In this section we will overview the most important application fields of C++ template metaprogramming. Since 1994 template metaprograms are used in various environments from essential components in high speed mathematical libraries – like Blitz++ [46] – to the automatic configuration of `boost::tr1` mathematical functions [49].

5.1 Expression templates

The earliest applications of template metaprogramming aimed to eliminate the overhead of object-oriented programming in numerical computations. To understand the root of the problem, consider the following scenario.

We want to implement numerical computations with the help of the well-designed class `Array`, which encapsulates a vector of floating point numbers, and basic operations like addition and multiplication. With the help of the operator overloading we can write the following code:

```
class Array;
Array a,b,c,d;

a = b + c + d;
```

Unfortunately, when we execute the above operation, certain ineffective events will happen. The operation `b + c` will produce a temporary `Array` as the result, and this temporary will be added to `d`, which produces an other temporary. Temporary `Array` objects will allocate space in the heap – a relatively slow operation, and will copy a huge number of bytes. Not to forget the destruction, we end up with something similar to the following pseudocode:

```
double* _t1 = new double[N]; // b+c
for ( int i=0; i<N; ++i)
  _t1[i] = b[i] + c[i];
```

```
  double* _t2 = new double[N]; // _t1+d
  for ( int i=0; i<N; ++i)
    _t2[i] = _t1[i] + d[i];

  for ( int i=0; i<N; ++i)   // a = _t2
    a[i] = _t2[i];

  delete [] _t2;
  delete [] _t1;
```

Veldhuizen measured 50 – 500 percentage of performance loss due to extra heap operations, memory access, etc. [41]. Meanwhile, a FORTRAN-style code could keep the high performance when implementing the following algorithm:

```
  for( int i=0; i<N; ++i)
    a[i] = b[i] + c [i] + d[i];
```

It seems we have to choose between manageable object-oriented style code and efficient FORTRAN style. *Expression templates* invented independently by Todd Veldhuizen [41] and David Vandevoorde [37] are possible *modus vivendi* for this problem.

Expression templates are recursive templates which are used to represent (typically arithmetic) expressions. Building up the parser tree and delaying computations instead of immediately evaluating the expression gives us more chances to eliminate temporaries and to optimize the execution process. The real evaluation happens later when we refer to a certain element of the result. This technique is often called *lazy evaluation*. In the following we discuss expression templates using an example originated to Veldhuizen.

In the core of the expression template we find such a template class:

```
// the node in the parse tree.
template <typename Left, typename Op, typename Right>
struct Node
{
    Left    left;
    Right   right;

    Node( Left t1, Right t2) : left(t1), right(t2)  { }

    double operator[](int i)
    {
        return Op::apply( left[i], right[i] );
    }
};
struct Array
{
```

```
    // constructor
    Array( double *data, int N) : data_(data), N_(N) { }

    // assign an expression to the array
    template <typename Left, typename Op, typename Right>
    void operator=( X<Left,Op,Right> expr)
    {
        for ( int i = 0; i < N_; ++i)
            data_[i] = expr[i];
    }
    double operator[](int i)
    {
        return data_[i];
    }
    double *data_;
    int N_;
};
```

Class `Node` will represent a node in the expression tree. The `Left` and `Right` parameters refer to the left-hand side and to the right-hand side nodes below the current one. These template parameters can be instantiated as another instances of class `Node` or – in case of a leaf in the tree – as the user's class (`Array` in our example). The constructor just builds up the expression, while `operator[]` will evaluate the expression itself executing `Op`'s `apply` function.

The middle parameter `Op` represents the operation we execute in this node of the expression. Most cases we do not need to store objects of class `Op`, as the operation is executed via static methods:

```
// this class encapsulates the "+" operation.
struct plus
{
    static double apply( double a, double b)
    {
        return a+b;
    }
};
```

We can build up the expression tree manually or we can use a generator template for this purpose:

```
template <typename Left>
X<Left, plus, Array> operator+( Left a, Array b)
{
    return X<Left, plus, Array>(a,b);
}
```

We write the application code as we would do it in pure object-oriented style. However, the generator function `operator+` will expand the expression `A+B` and starts to build up the expression tree.

```
Array A(...); // fill A
Array B(...); // fill B
Array C(...); // fill C
Array D(...); // allocate D

D = A + B + C;
```

After a few steps we get the following code:

```
void D.operator=(X<X<Array,plus,Array>,plus,Array>(
                      X<Array,plus,Array>(A,B),C) expr)
{
    for ( int i = 0; i < N_; ++i)
        data_[i] = expr[i];
}
```

Inlining `X::expr[i]` with its body leads to

```
for ( int i = 0; i < N_; ++i)
    D.data_[i] = A.data_[i] + B.data_[i] + C.data_[i];
```

which is equivalent to the FORTRAN style solution. We wrote the code in object-oriented style, but the expression templates transformed it into the most efficient implementation.

5.2 Language extension

In C++ integer values can be expressed by constants in form of decimal, octal or hexadecimal literals. The syntax is defined by the C++ language standard and is not extendable. However, in certain cases we would like to write integer values in binary form. As in most programming languages, in C++ writing a function which converts it's string argument forming a binary number is trivial. Unfortunately, this function will be executed in run-time, which has a number of shortages:

- It consumes run-time resources, even if the string argument is known in compile time.
- It is repeatedly called when evaluating the same literal.
- The return value is not known in compile time. Apart from optimization questions that fact denies declaring arrays with the size of the return value.
- When non-binary characters appear in the literal, the error is reported in run-time instead of compilation time.

```
int main()
{
  const int di = 12;
  const int oi = 014;
  const int hi = 0xc;
  const int bi0 = binary_value("1100");
  const int bi1 = binary<1100>::value;
}
```

A clever C++ template metaprogram can solve all of these problems. Let us consider the following template definition:

```
template <unsigned long N>
struct binary
{
    static unsigned const value = 2*binary<N/10>::value + N%10;
};
```

When we specify `binary<1100>::value` in a C++ source file a compiler instantiates the generic version of the `binary<N>` template with `1100` as an integer argument. The instantiation will refer to `binary<110>::value` and then `binary<11>::value` and so on until we reach `binary<0>::value`. Then a specialization is required to stop the recursion:

```
template <>
struct binary<0>
{
    static unsigned const value = 0;
};
```

After creating five classes, we finish to construct the topmost instance and `binary<1100>::value` will be 12 as a compile time constant. We can suppose that `binary<1100>` appears in other parts of the source. In that case the compiler need not repeat the whole instantiation process. C++ templates use *memoisation*, i.e. ones instantiated templates will be kept during the compilation process, and will be re-used rather than re-instantiated. (Memoisation is not always a good feature. It also means that we have to keep the unused immediate classes too during compilation. In most cases this is just wasting critical resources during the whole compilation process.)

5.3 Concept checking

Modern programming languages have tools to express parametric polymorphism, i.e. functions or data structures parameterized by types. Generics in Ada, Eiffel, Java or C#, templates in C++ are such constructs. However, it is clear that generic constructs can not always accept arbitrary type parameters. For example, abstract priority queues should contain types which are *comparable*, accumulator

functions *additive* types. These assumptions are restrictions against genericity. In some programming languages the constraints could be expressed explicitly by the language. Java's *wildcards* [35], the inheritance hierarchy in Eiffel and the `with` keyword in Ada serves this purpose. If we break the constraints, we get clear and straightforward error messages from the compiler.

C++ has no language-level support to describe explicit requirements for certain template properties, i.e. C++ templates are not constrained. When we pass a type without proper comparison methods to an abstract priority queue, we do not experience an immediate syntax error. In contrast, the instantiation process starts and it will fail only when the lacking method is explicitly referred, in most cases somewhere deeply in the chain of instantiations.

The canonical example is the standard template library (STL), where algorithms require certain types of iterators. E.g. the `sort` algorithm requires parameters in form of *random access iterators*. When `sort` is called with parameters only satisfying the criteria of *forward iterators* we end up with a few pages of error messages and neither of them will explicitly tell us the root of the problem.

Due to lack of compiler support, the problem had to be remedied on library level. Complex language constructs have been created to inspct the characteristics of types. Existence of certain attributes or methods, usage of polymorphism, inheritance relationships, etc. can be determined in compilation time using template metaprograms [45]. Based on the inspections, in case of breaking rules, the designer of the program may decide to abort compilation. This area of research is called *static interface checking* or *concept checking* [21, 26].

A compilation of such language constructs, the *Boost Concept Checking Library (BCCL)*[47] uses template mechanisms to provide a wide variety of compile-time checks, and produce human-readable error messages when a criterion is not met by a type:

```
// Library function with constraints to T
template <class T>
void generic_library_function(T x)
{
  function_requires< EqualityComparableConcept<T> >();
  // ...
}
// user code
class foo
{
  // ...
};

int main()
{
  foo f;
  generic_library_function(f);
```

```
  return 0;
}
```

When class `foo` does not fulfill the requirement of `EqualityComparableConcept`, the call of `generic_library_function(f)` will cause a compilation error, with a hopefully human-understandable error message about the missing requirement.

Concept checking algorithms are often complex and compiler dependent. In the last ten years lots of effort has been spent to develop high quality concept libraries. During the years it turned out that library-based solutions have significant shortages compared to language-based concepts. Therefore the ANSI C++ committee started to work on a proposal to extend C++ with language-based concepts. With the help of *concepts* [25] programmers could specify the requirements agains template parameters of classes and functions in a clear syntax, and could separate concept checking from the instantiation process.

Unfortunately, this enhancement requires enourmous amount of work – especially reimplementing existing libraries by the enrichment of concepts. In the summer of 2009 the C++ standardization committee excluded concepts from the already late C++0X standard. Concepts are not forgotten but it is hard to predict when they will be part of the official C++ standard. Until then, we may utilize library based concepts implemented mostly by means of template metaprograms.

5.4 Active libraries

With the development of programming languages, user libraries also became more complex. *FORTRAN* programs already relied heavily on programming libraries implementing solutions for re-occuring tasks. With the emerging of object-oriented programming languages the libraries also transformed: the sets of functions were replaced by collections of classes and inheritance hierarchies. However, these libraries are still *passive*: the writer of the library has to make substantial decisions about the types and algorithms at the time of the library's creation. In some cases this constraint is a serious disadvantage. Contrarily, an *active library* [38] acts dynamically, makes decisions in compile-time based on the calling context, chooses algorithms, and optimizes code. These libraries are not passive collections of functions or objects, as are traditional libraries, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves. In C++ active libraries are implemented with the help of template metaprogramming techniques.

In the following we present an example for active libraries based on our own research. We will implement a *Final State Machine* (FSM) with the help of a *State Transition Table* (STT). As soon as the STT is defined, in compilation time, algorithms and transformations can be executed on it, and also optimizations and sanity checking of the whole state transition table can be done. Therefore we decided using template metaprograms to provide automatic operations at compile-time on the FSM. Our goal is to develop an inital study that:

– carries out compound examinations and transformation on the state transition table,
– and shows the relationship between Finite State Machines and Active Libraries over a template metaprogram implementation of the Moore reduction procedure.

The library is based on a simplified version of *Boost::Statechart*'s State Transition Table. In our model the State Transition Table defines a directed graph. We implemented the Moore reduction procedure, used the *Breadth-First Search* (BFS) algorithm to isolate the graph's main strongly connected component and with the help of a special "Error" state we made it complete.

Much like the Boost::Statechart's STT, in our implementation states and events are represented by classes, structs or any built-in types. The STT's implementation based on the Boost::MPL::List compile-time container is described in Figure 1:

```
template< typename T, typename From, typename Event, typename To,
     bool (T::* transition_func)(Event const&)>
struct transition
{
    typedef T       fsm_t;
    typedef From    from_state_t;
    typedef Event   event_t;
    typedef To      to_state_t;

    typedef typename Event::base_t base_event_t;
    static bool do_transition(T& x, base_event_t const& e)
    {
        return (x.*transition_func)(static_cast<event_t const &>(e));
    }
};

typedef mpl::list<
//    Current state   Event    Next state         Action
//    +-----------+----------+-------------+----------------------+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >
//    +-----------+----------+-------------+----------------------+
>::type sample_transition_table; // end of transition table
```

Fig. 1. Implementation of our State Transition Table.

A transition table built at compile-time behaves similarly to a counterpart built in runtime. The field `transition_func` pointer to member function represents the tasks to be carried out when a state transition happens. The member

function `do_transition()` is responsible for the iteration over the table. The state appearing in the first row is considered the starting state.

In the following we present a simple use case. Let us imagine that we want to implement a simple CD player, and the behavior is implemented by a state machine. The state transition table skeleton can be seen in Figure 2. To demonstrate the compile time actions, we intentionally put duplicated functionalities and unreachable states to our STT.

```
typedef mpl::list<
//    Current state   Event    Next state         Action
//    +-----------+----------+-------------+----------------------+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >,
trans < Playing   ,   pause  ,   Paused    ,   &p::pause          >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume         >,
//                   ... duplicated functionality ...
trans < Stopped   ,   play   ,   Running   ,   &p::start_running  >,
trans < Running   ,   stop   ,   Stopped   ,   &p::stop_running   >,
trans < Running   ,   pause  ,   Paused    ,   &p::pause          >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume         >,
//                    ... unreachable states ...
trans < Recording ,   pause  ,   Pause_rec ,   &p::pause_recording>,
trans < Paused_rec,   resume ,   Recording ,   &p::resume_rec     >
//    +-----------+----------+-------------+----------------------+
>::type sample_trans_table; // end of transition table
```

Fig. 2. Sample State Transition Table

The programmer first starts to implement the Stopped, Playing, and Paused states' related transitions. After implementing a huge amount of other transitions, eventually he forgets that a Playing state has already been added, so he adds it again under the name *Running*. This is an unnecessary redundancy, and in generaly could indicate an error or sign of a bad design. A few weeks later it turns out, that a recording functionality needs to be added, so the programmer adds the related transitions. Unfortunately, the programmer forgot to add a few transitions, so the Recording and Paused state cannot be reached. In general that also could indicate an error. On the other hand if the state transition table contains many unreacheable states, these appear in the program's memory footprint and can cause runtime overhead.

Our library can address these cases by emitting warnings, errors messages, or by eliminating unwanted redundancy and unreacheable states. The result table of the reduction algorithm can be seen here:

```
template struct fsm_algs::reduction< sample_trans_table >;
```

After this forced template instantiation, the `enhanced_table` typedef within this `struct` holds an optimized transition table is described in Figure 3:

```
typedef mpl::list<
//    Current state   Event    Next state         Action
//    +-----------+----------+-------------+----------------------+
trans < Stopped   ,   play   ,   Playing   ,   &p::start_playback >,
trans < Playing   ,   stop   ,   Stopped   ,   &p::stop_playback  >,
trans < Playing   ,   pause  ,   Paused    ,   &p::pause          >,
trans < Paused    ,   resume ,   Playing   ,   &p::resume         >,
         ... duplicated functionality has been removed ...
          ... unreachable states have been removed too ...
//    +-----------+----------+-------------+----------------------+
>::type sample_trans_table; // end of transition table
```

Fig. 3. Reduced Transition Table

In the following we present the minimization algorithm implemented in our active library.

Locating strongly connected components The first algorithm executed before the Moore reduction procedure is the localization of the strongly connected component of the STT's graph from i a given vertex. We use Breadth-First Search to determine the strongly connected components. After we have located the main strongly connected component from a given state, we can emit a warning / error message if there is more than one component (unreachable states exist) or we can simply delete them. The latter technique can be seen in Figure 4 (several lines of code have been removed):

Making the STT's graph complete The Moore reduction algorithm requires a complete STT graph, so the second algorithm that will be executed before the Moore reduction procedure is making the graph complete. We introduce a special "Error" state, which will be the destination for every undefined state-event pair. We test every state and event and if we find an undefined event for a state, we add a new row to the State Transition Table. (Figure 5.)

The destination state is the "Error" state. We can also define an error-handler function object[17]. After this step, if the graph was not complete, we've introduced a lot of extra transitions. If they are not needed by the user of the state machine, these can be removed after the reduction. The result after the previously executed two steps is a strongly connected, complete graph. Now we are able to introduce the Moore reduction procedure.

```
// Breadth-First Search
template < typename Tlist, typename Tstate, typename Treached,
//                  STT ^    Start state ^   Reached states ^
           typename Tresult = typename mpl::clear<Tlist>::type,
//                  ^ Result list is initialized with empty list
           bool is_empty =  mpl::empty<Treached>::value  >
struct bfs
{
   // Processing the first element of the reached list
   typedef typename mpl::front<Treached>::type process_trans;
   typedef typename process_transition::to_state_t next_state;

   // (...) Removing first element
   typedef typename mpl::pop_front<Treached>::type
         tmp_reached_list;

   // (...) Adding recently processed state table rows
   // to the already processed (reachead) list
   typedef typename merge2lists<tmp_result_list, tmp_reached_list>
         ::result_list tmp_check_list;

   // (...)  Recursively instantiates the bfs class template
   typedef typename bfs< Tlist, next_state, reached_list,
         tmp_result_list, mpl::empty<reached_list>::value>
 ::result_list result_list;
};
```

Fig. 4. Implementation of Breadth-First Search

The Moore reduction procedure Most of the algorithms and methods used by the reduction procedure have already been implemented in the previous two steps.

First we suppose that all states may be equivalent i.e. may be combined into every other state. Next we group non-equivalent states into different groups called equivalence partitions. When no equivalence partitions have states with different properties, states in the same group can be combined. We refer to equivalent partitions as sets of states having the same properties.

We have simulated partitions and groups with Boost::MPL's compile time type lists. Every partition's groups are represented by lists in lists. The outer list represents the current partition, the inner lists represent the groups. Within two steps we mark group elements that need to be reallocated. These elements will be reallocated before the next step into a new group (currently list).

After the previous three steps the result is a reduced, complete FSM whose STT has only one strongly connected component. All of these algorithms are executed at compile time, so in run-time we are working with a minimized state machine.

```
//    Current state   Event    Next state         Action
//    +-----------+---------+-------------+----------------------+
trans <     Stop  ,   pause  ,    Error    ,   &p::handle_error   >
```

Fig. 5. Adding new transition.

For more implementatational details refer to [13].

5.5 DSL-based language extentions

Domain specific languages are dedicated to some special problems, like database related tasks, or expressing regular expressions in an effective way. They are often incorporated into some general purpose host language. The main problem is to provide type safety and consistency between the host language and the embedded language. One way to implement this in C++ is to use template metaprograms.

AraRat system [12] is an example which implements a domain specific language using C++ template metaprograms. AraRat provides a type safe SQL interface for queries. It uses operator overloading over types generated based on the actual database schema. When expressions violate schema rules or used inconsistent way communicating with host C++ environment a compile-time error is generated.

The `boost:xpressive` library [48] is used for compile time checking of regular expressions. In most regex libraries, the patterns are represented as string literals or variables and the syntax of the regular expressions (i.e. each parantheses has a closing symbol, etc.) are checked only in run-time. The `boost:xpressive` library allows creating and compile time checking certain regular expressions. This way we can detect some syntactically bogus patterns in compilation time.

5.6 Traits, policy classes

In Section 3 we shortly discussed a `matrix` template. As this template uses a buffer allocated in the heap to store the elements, we have to provide copy constructor and assignment operator to ensure the meaningful copy of `matrix` elements. The textbook example for such copy operators looks similar to this:

```
template <class T>
matrix<T> matrix<T>::operator=( const matrix &other)
{
  if ( this != &other )
  {
    delete [] v;
    copy( other);
  }
  return *this;
```

```
}
template <class T>
void matrix<T>::copy( const matrix &other)
{
  x = other.x;
  y = other.y;
  v = new T[x*y];
  for ( int i = 0; i < x*y; ++i )
    v[i] = other.v[i];
}
```

The most expensive part of the function is the loop for copying the elements. However, there will be a serious mistake to replace the loop with an otherwise much faster bitwise copy function, like `memcpy()`. Since the type argument `T` could be any copyable type, in the generic solution we have to call the assignment operator of type `T` to ensure the correct copy behavior for the content of the matrix. Exactly that happens in the line `v[i] = other.v[i]`.

With the loop we created a *safe* copy. However, it is possible that in most of the cases we will store elements of type `int` or `double` the matrix, i.e. elements which are completely safe to copy with functions, like `memcpy()`. Such types are called *Plain Old Data* (POD) types in C++. Can we somehow accomodate safety with efficiency? Can we use `memcpy()` when copying POD types, and apply the loop on other cases?

We can start with the most essential template tool we have: specialization. Let us specialize `copy` for some POD types (like `long` and `double`) using `memcpy()` and leave the generic solution (with loop) for the rest of the types:

```
template <class T>
void matrix<T>::copy( const matrix &other) // generic version
{
  x = other.x;
  y = other.y;
  v = new T[x*y];
  for ( int i = 0; i < x*y; ++i )
    v[i] = other.v[i];
}
template <>
void matrix<long>::copy( const matrix &other) // specialization
{
  x = other.x;
  y = other.y;
  v = new long[x*y];
  memcpy( v, other.v, sizeof(long)*x*y);
}
// similar copy() for double, ...
```

This works, but quickly leads to unmanagable code. Type specific template specializations are scattered across the code and we have to repeat this procedure for all new types we want to copy in the optimal way.

To modularize type-specific codes we can use *trait* classses. Traits in C++ provide a convenient way to associate related types, values, and functions with a template parameter type without requiring that they be defined as members of the type [2, 19]. This is extremely useful when we do not want to or not able to add new members to an existing class.

One well-known application field of traits is the extension of non-class types, such built-in types or pointers. For example, when an *iterator* is implemented in the means of a C++ class, like `vector::iterator` we can define associated types, like `difference_type` or `value_type` as members. However, this is impossible for non-class types, like pointers which otherwise should behave similarly for iterators. The solution is the `iterator_traits<>` template, where we can specify essential information for an *iterator* class:

```
template <class Iterator> struct iterator_traits;

template <class Iterator>
struct iterator_traits
{
  typedef typename Iterator::value_type value_type;
  typedef typename Iterator::reference  reference;
  typedef typename Iterator::pointer    pointer;
  typedef typename Iterator::difference_type   difference_type;
  typedef typename Iterator::iterator_category iterator_category;
  // ...
};

template <>
struct iterator_traits<T*>
{
  typedef T  value_type;
  typedef T& reference;
  typedef T* pointer;
  typedef ptrdiff_t difference_type;
  typedef random_access_iterator_tag iterator_category;
  // ...
};
```

Applications can access iterator's features via the `iterator_traits` class. Thus we eliminate the difference between class types and built-in types.

```
template <class ForwardIter1, class ForwardIter2>
void iter_swap( ForwardIter1 it1, ForwardIter2 it2)
{
```

```
  typename terator_traits<ForwardIter1>::value_type tmp = *it1;
  *it1  = *it2;
  *it2  = tmp;
}
```

A similarly important application of traits is the `char_traits<>` template, where we can collect essential extra information on the actual character type. The `char_traits` as a generic template class itself is defined, but never used. There are several specializations for specific character types, like `char_traits<char>`, where fundamental features of the template argument type (here `char`) is described. When generic templates, like `basic_string<>` are about to use a character type, the appropriate traits class is the source of the implementational details.

Traits provide an upward-compatible technique to allow greater flexibility, even at runtime, at no cost in convenience or efficiency. We will use traits to describe and to gather together copy-related informations in the `matrix` example.

We will create a generic trait class to handle the general case: copying with loop applying the assignment operator of the template argument type. For individual POD types we create copy trait specializations where the `copy()` function is defined by the means of `memcpy()`.

```
template <typename T>
struct copy_trait
{
  static void copy( T* to, const T* from, int n)
  {
    for( int i = 0; i < n; ++i ) // generic trait
      to[i] = from[i];
  }
};
template <>
struct copy_trait<long>
{
  static void copy( long* to, const long* from, int n)
  {
    memcpy( to, from, n*sizeof(long));  // specialization
  }
};
template <>
struct copy_trait<double>
{
  static void copy( double* to, const double* from, int n)
  {
    memcpy( to, from, n*sizeof(double));  // specialization
  }
};
```

```
template <class T, class Cpy = copy_trait<T> >
class matrix
{
  //...
};
// ...
template <class T, class Cpy>
void matrix<T,Cpy>::copy( const matrix &other)
{
  x = other.x;
  y = other.y;
  v = new T[x*y];
  Cpy::copy( v, other.v, x*y);
}
```

We added an extra argument to class `matrix` as the trait class to describe the expected behaviour in case of copying the object. When the `matrix` template is about to be instantiated with a certain argument type `X`, the second argument will be `copy_trait<X>`. For those `X` parameters which has no trait specializations the generic version of `copy_trait<>` will be used. This will execute the loop-based copy. For those `X` parameters we specialized copy traits we will execute `copy_trait<X>::copy()`.

All type-specific functionalities can be concentrated into the appropriate trait class. In the same time, class `matrix` does not include any type-specific code anymore. Specifying new types to use `memcpy()` requires creating a new trait class, i.e. we still have to repeat specializations, but at least we can modulize them.

We can consider, that the copy we execute on `long` and `double` has essentially the same code. In other words, our decision does not depend on the exact type argument anymore. We have to know only that they are both POD types. Can we generalize our solution based on this fact?

While traits contain basically collected information on a specific type argument, *policy classes* manifest strategies typically applicable for several types [2].

We will define `is_pod` class to hold information whether its type argument is a POD type or not. Teh default value in the generic template will be `false`, as we may not suppose PODness for unknown types. Declaring POD types is expressed by specializations of `is_pod`:

```
template <typename T>
struct is_pod
{
  enum { value = false };
};
template <>
struct is_pod<long>
```

```
{
  enum { value = true };
};
template <>
struct is_pod<double>
{
  enum { value = true };
};
// other POD types...
```

Basically, that is the all type-specific part of the policy-based solution. The rest of the code does not contain variations on different types and does not need to modify when adding new POD types to the system.

Class `copy_policy` retrieves information on the POD status of its template argument and the `matrix` class will instantiate the appropriate copy policy automatically deduced from its first template argument.

```
template <typename T, bool B>
struct copy_policy
{
  static void copy( T* to, const T* from, int n)
  {
    for( int i = 0; i < n; ++i )
      to[i] = from[i];
  }
};
template <typename T>
struct copy_policy<T, true>
{
  static void copy( T* to, const T* from, int n)
  {
    memcpy( to, from, n*sizeof(T));
  }
};
template <class T, class Cpy = copy_policy<T,is_pod<T>::value> >
class matrix
{
  // ...
};
```

Here we impoved the solution separating two policies: copying POD types, and non-POD types. To define a type as POD type we simply create a specialization of `is_pod<>` for that type. Thus, adding new POD types is done in a very declarative way.

But we can still improve the solution using typelists. In the following solution we simply declare the required POD types in a typelist and everything else is

done automatically. Compile time alogorithms, like Loki's `IndexOf`, discussed in section 4 is able to detect whether a type is member of a typelist.

```
typedef TYPELIST_4(char, int, long, double)  Pod_types;

template <typename T>
struct is_pod
{
  enum { value = ::Loki::TL::IndexOf<Pod_types,T>::value != -1 };
};
struct copy_trait
{
  static void copy( T* to, const T* from, int n)
  {
    for( int i = 0; i < n; ++i )
      to[i] = from[i];
  }
};
template <typename T>
struct copy_trait<T, true>
{
  static void copy( T* to, const T* from, int n)
  {
    memcpy( to, from, n*sizeof(T));
  }
};
template <class T, class Cpy = copy_trait<T,is_pod<T>::value> >
class matrix
{
  //...
};
```

Further automatization is compiler dependent. Using `boost::type_traits` library, we can apply the `boost::type_traits::is_pod<>` template, which returns true for POD types only. This functionality, however, uses highly sophisticated template tricks and often depends on non-standard compiler intrinsics.

6 Debugging template metaprograms

Programming is a human activity to understand a problem, make design decisions, and express our intentions for the computer. In most cases the last step is writing code in a certain programming language. The compiler then tries to interpret the source code through lexical, syntactic, and semantic analysis. In case the source code is syntactically correct, the compiler takes further steps to generate runnable code.

However, in numerous cases the code accepted by the compiler will not work as we had expected, and intended. The causes vary from simple typos – that (unfortunately) do not affect the syntax – to serious design problems. There are various methods to decrease the possibility of writing software diverging from its specification, nevertheless in many cases we have made some error, and we have to fix it. For this we have to recognise that the bug exists, isolate its nature, and find the place of the error to apply the fix. This procedure is called debugging.

Debuggers are software tools to help the debugging process. The main objective of a debugger is to help us understand the hidden sequence of events that led to the error. In most cases this means following the program's control flow, retrieving information on memory locations, and showing the execution context. Debuggers also offer advanced functionality to improve efficiency of the debugging process. These include stopping the execution on a certain *breakpoint*, continuing the running step by step, step into, step out, or step over functions, etc. Still, debugging can be one of the most difficult and frustrating tasks for a programmer.

In this section we describe possible debugging stategies for C++ template metaprograms. First we discuss the ontology of template metaprogram errors, than we overview possible implementation strategies for debugging template metaprograms.

6.1 Ontology of template metaprogram errors

As we have seen in section 3, Unruh's first template metaprogram emitted error messages to print prime numbers. The program is errorneous in the traditional sense, as it would not compile and therefore is unable to run. Was this program correct or errorneous as a template metaprogram? As the goal of the program – printing prime numbers – has been achieved, we should consider Unruh's code as a correct metaprogram. This example points out the difference of the notions *correct* and *erroneous* behaviour between traditional runtime programs and template metaprograms.

Let us examine the `Factorial` metaprogram described in Section 3, and let us suppose that the template specialization `Factorial<1>` has a syntactic error: a semicolon is missing at the end of the class definition.

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

```
} // ; missing
```

This is an *ill-formed* template metaprogram, with a *diagnostic message*. The metaprogram has not been run: no template instantiation happened. Another *ill-formed* template metaprogram with *diagnostic message* is shown in the next example. However, it starts to "run", i.e. the compiler starts to instantiate the `Factorial` classes, but the metaprogram *aborts* (in compilation time).

```
template <int N>
class Factorial
{
public:
  enum { value = N*Factorial<N-1>::value };
};
template<>
class Factorial<1>
{
// public: missing
  enum { value = 1 };
};
int main ()
{
  const int f = Fibonacci<4>::value;
  const int r = Factorial<5>::value;
}
```

As the full specialization for `Factorial<1>` is written in form of a `class`, the default visibility rule for a class is `private`. Thus `enum { value=1 }` is a private member, so we receive a compile-time error when the compiler tries to acquire the value of `Factorial<1>::value`, when `Factorial<2>` is being instantiated. The main difference from the earlier *ill-formed* example is that here instantiations are started. For example, the `Fibonacci<4>::value` is computed.

In our next example we remove the full specialization `Factorial<1>`:

```
template <int N>
struct Factorial
{
  enum { value = N*Factorial<N-1>::value };
};
// specialization for N==1 is missing
int main ()
{
  const int r = Factorial<5>::value;
}
```

As the `Factorial` template has no explicit specialization, the `Factorial<N-1>` expression will trigger the instantiations of `Factorial<1>` followed by the instantiation of `Factorial<0>`, `Factorial<-1>` etc. We have written a compile-time

infinite recursion. This is an *ill-formed* template metaprogram with *no diagnostic message*, equivalent to infinite loops of run-time programs.

The C++ standard requires a minimum of 17 level of recursive template instantiations. Therefore portable metaprograms must not exceed this limit. However, different compilers have rather diverse behavior.

Compiler `g++ 3.4` halts the compilation process after the 17 levels of implicit instantiations is reached, as defined by the C++ standard. This limit can be modified by compiler flags. The `MSVC 6` compiler runs until its resources are exhausted (reached `Factorial<-1308>` in our test). `MSVC 7.1` halted the compilation reaching a certain recursion depth. The error message received was *fatal error C1202: recursive type or function dependency context too complex*.

However, some compilers, like `g++` can be parameterised to accept deeper instantiation levels. In this case the compiler continues the instantiation risking that the resources will be exhausted. In that unfortunate situation the compiler will crash.

6.2 Debugging techniques

Tools we use in run-time programming for debugging are not available in the well–known way, when dealing with metaprograms. We have no command for printing to the screen (in fact we have practically no commands at all), we have no framework to manage running code. On the other hand, we still have some options. Having a set of good debugging tools in the runtime world and a strong analogue between the runtime and compile-time realm we can attempt to implement a template metaprogram debugging framework. In the following we explain the structure of *templight*, a template metaprogram debugger framework.

A common property of debugging tools is that they analyse a specific execution of the program. In the case of debugging C++ template metaprograms our goal is to retrieve the chain of template instantiations with as much additional information (template parameters, etc.) as we can.

In the most favourable case the execution does not depend on the usage of the debugging tool. In such cases it does not matter whether we are using the tool on the running program itself or we analyse a previously generated trace of its runtime steps. Most compilers generate additional information for debuggers and profilers. Obviously the simplest way for providing trace information on instantiations would be the implementation of another compiler feature. However, an immediate and more portable solution is to use external tools cooperating with standard C++ language elements. The appropriate compiler support could be an ideal long-term solution.

Without the modification of the compiler the only way of obtaining any information during compilation is generating informative warning messages that contain the details we are looking for [1]. Therefore the task is the *instrumentation* of the source, i.e. its transformation into a functionally equivalent modified form that triggers the compiler to emit talkative warning messages. The concept of such instrumentation is a usual idea in the field of debuggers, profilers and program slicers. Everytime the compiler starts to instantiate a template, defines

an inner type etc. the inserted code fragments generate detailed information on the actual template-related event. Similar warnings should be emitted when we reach the end of the template. Embedded start and end markers unambigously identifies the chain of template instantiations – similarly to the stack frames of run-time programs. We have to gather the desired information from the corresponding warning messages in the compilation output and form a trace file. A front-end tool may use this information to implement various debugging features and visualization of the instantiations.

The input of the process is a C++ source file and the output is a trace file, a list of events like *instantiation of template X began*, *instantiation of template X ended*, *typedef definition found* etc. The procedure begins with the execution of the preprocessor with exactly the same options as if we were to compile the program. As a result we acquire a single file, containing all #included template definitions and the original code fragment we are debugging. The preprocessor decorates its output with #line directives to mark where the different sections of the file come from. This information is essential for a precise jump to the original source file positions as we step through the compilation while debugging in an IDE for example. To simplify the process we handle the mapping of the locations in the single processed file to the original source files in a separate thread. Simple filter scripts move the location information from #line directives into a separate mapping file and delete #line directives.

At this point we have a single preprocessed C++ source file, that we transform into a C++ token sequence. To make our framework as portable and self-containing as possible we apply the `boost::wave` C++ parser. Note that even though `boost::wave` supports preprocessing, we still use the original preprocessor tool of the compilation environment to eliminate the chance of bugs occurring due to different tools being used. Our aim is to insert warning-generating code fragments at the instrumentation points. As `wave` does no semantic analysis we can only recognise these places by searching for specific token patterns. We go through the token sequence and look for patterns like *template keyword + arbitrary tokens + class or struct keyword + arbitrary tokens +* { to identify template definitions. The end of a template class or function is only a } token that can appear in quite many contexts, so we should track all { and } tokens in order to correctly determine where the template contexts actually end. This pattern matching step is called annotating, its output is an XML file containing annotation entries in a hierarchical structure following the scope.

The instrumentation takes this annotation and the single source and inserts the warning-generating code fragments for each annotation at its corresponding location in the source thus producing a source that emits warnings at each annotation point during its compilation. The next step is the execution of the compiler to have these warning messages generated. The inserted code fragments are intentionally designed to generate warnings that contain enough information about the context and details of the actual event. Since the compiler may produce output independently of our instrumentation, it is important for debugger warnings to have a distinct format that differentiates them. This is the step

where we ask the compiler for valuable information from its internals. Here the result is simply the build output as a text file. The warning translator takes the build output, looks for the warnings with the aforementioned special format and generates an event sequence with all the details. The result is an XML file that lists the events that occurred during the compilation in chronological order. The annotations and the events can be paired. Each event signals that the compiler went through the corresponding annotation point. We can say events are actual occurrences of the annotation points in the compilation process. More technical details on templight can be found in [24].

6.3 Profiling

Unfortunately, implementations of template metaprograms are typically far from optimal [1]. One reason is that compilers are optimized to generate efficient runtime code and not designed to maximize efficiency of the compilation process itself. Another reason is that programmers are not familiar with all the background costs of the metaprogram constructs. This may result in a very long compilation time and huge memory usage. With a profiling tool we should be able to identify these "noisy" code segments, which hold up the compilation process. Since traditional profiler tools are unapplicable to metaprograms running in compile-time, the development of metaprogram-specific profiling tools is crucial. Unfortunately, today there are no C++ template metaprogram profiling tools available. In this subsection we describe methods for template metaprogram profiling, which could serve as foundations of an optimization process.

Measuring compilation units The most available method to measure compile time performance is measuring full compilation of units. Compilation of full source files does not require code modification, thus this is a non-intrusive method, and do not add overhead or significant distorsion. Although filtering out all perturbations is not easy, most of the operating systems provide us fair tools to measure the experienced real-time, user and system times on the run of a compilation session.

In most cases locating, loading, and parsing header files is a non-trivial effort. To filter out this effect we can run the precompiler in a separate session and measure only further compilation stages. Figure 6 shows that separating precompiler tasks changes the compilation times significantly.

Compiling full programs or compilation units can reveal significant behavioural patterns of programs or template constructs. Abrahams and Gurtovoy measured template metaprogram constructs in [1] with this method and could point to fundametal differences in strategy and tactics of different compilers. They have shown the effect of certain techniques, like memoisation and have measured structural complexity of metaprograms.

However, measuring full compilation time has shortages. It is not always trivial to write wrapper programs around specific template constructs without seriously distort measurement results. The full session of compilation includes activities we are not interested in: initializations, outputting, solving non-template

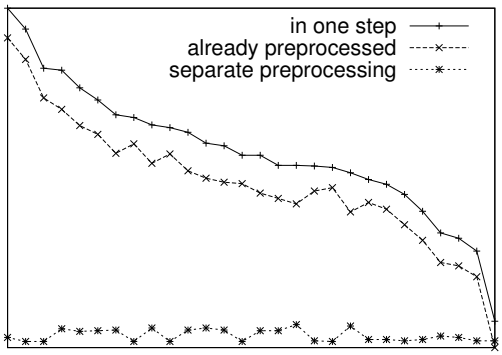

Fig. 6. Compilation time with separate precompilation

related tasks. Code generation, optimization steps produce significant overload too. When we analyse the results we have the compilation times, but no implications on how this gross time splits among different code components. Measuring full compilation is great to prove concepts but hard to use for analysis.

Measuring with instrumenting Most compilers generate additional information for profilers. An appropriate compiler support for measuring template metaprogram profiles would be the ideal solution. However, as this support is unavailable as of now, an immediate and portable method is to use external tools cooperating with standard C++ language elements.

A natural choice is using the templight tool, described in subsection 6.2, to instrument the source code providing profiling information. With the templight framework we have to execute only one compilation that emits warnings for each instantiation, and a post processing pipelined tool memorizes the timestamps whenever a warning occures. This way we have timestamps for each template-related event, and the processing time of a certain template instance can be easily computed by subtracting the timestamps stored at the corresponding template-begin and template-end events (warning messages).

A factor of distortion is the way we add timestamps to the emitted warning messages. Compilers do not decorate warnings with timestamp info. In the simplest solution an external program reads compiler output and records the actual time whenever it sees some of our special warning messages being produced by the instrumented fragments. In this case the delay between the warning is generated and timestamped can be sigificant. Better way, if timestamp is generated inside the compiler when constructing the warning message, this delay can be eliminated. But this requires the modification of the compiler.

Using templight there is a special attention have to be paid to inheritance relationship. As warnings emitted by the injected code appear at the begin and end of templated code, the end marker of the base class will be emitted *before* the begin marker of the derived class. This could be solved by the extra decoration of the (first) base class.

```
template <typename T>
class Derived : public ReportInherit<Derived<T>, Base<T> >::Base
{
  /* skipping this instrumentation point */
  // ..
  /* remaining instrumentation point */
};
```

Modification of the compiler The most accurate way of evaluating compilation times is by acquiring timing information from the compiler itself. As our metaprogram is executed on a meta-level from the viewpoint of C++, a meta-level profiler is needed, i.e. one measuring the compiler's action times. The naïve approach – to use a profiler tool (like `gprof`) and measure the compiler's runtime – does not work, since we cannot identify which metaprogram elements of the subject code are under compilation at a certain moment. Even though we would be able to measure some kind of compiler method's running time in general, we could not disambiguate certain instantiations. In other words, we could acquire the sum of all instaniation times, but would not be able to measure each instantiation separately.

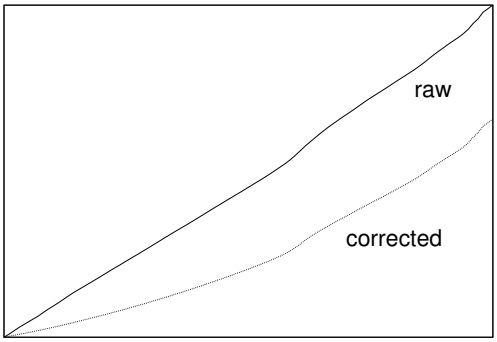

Fig. 7. Compile time with the time spent for warnings (raw) and without it (corrected) in a test of increasing template depth

To gain the required detailed data on particular instantiations we have to modify the compiler fur the purpose. We instrument the code with templight, but generate warnings decorated with timestamps via the modified compiler. In Figure 7 we show the compile time after we instrumented GNU `g++` compiler (version 3.4.3) The modification consists of generating timestamps when entering and exiting these functions and adding it to the emitted message. We used this approach to eliminate the distortion of generating the warning itself. Experiments showed, that in many cases the time we spent in these functions is significant.

The measured code contains a recursive template instantiated with a large parameter. The `raw` data shows the observed times, i.e. the compilation time the compiler spent on instantiating up to 2500 instances of the measured class *plus* the time of the warning generation due to code instrumentation. The `corrected` data has been constructed by subtracting the time the compiler spent with warning generation from the observed time.

7 Functional interface for template metaprograms

Writing programs today is largely supported by various automated tools, like code generators mapping UML notations to source code, model driven architectures, cross-compilers, RAD tools, etc. Coding, however, is still considerably influenced by personal experiences, conventions, traditions, and customs. The syntax and the semantics of the programming language is a major factor as it seriously drives the programmer's attitude. It is possible, but not easy to program in a style which is not directly supported by the actual programming language. Even worse if the required programming approach is not a supported paradigm. Similarly, as the spoken language has impact on human perception, the programming language may drive programmer's style. In an ideal situation the applied programming language supports the paradigm the task have to be solved in.

C++ templates has been designed to express genericity on data structures and algorithms – i.e. parametric polymorphism. Template metaprogramming has been discovered almost as a side effect, and template syntax that time has already been formulated. That syntax is far to be expressive regarding template metaprograms.

Let's examine the following C++ template metaprogram which decides in compile time whether it's parameter is a prime number.

```
#include <iostream>

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%from!=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> {};
int main()
{
  std::cout << IsPrime<337>::value << std::endl;
}
```

As we have seen in Section 4, C++ template metaprograms' behaviour and their programming are very close to functional programming paradigm. Although, this relationship is well-known, current C++ template metaprogramming libraries does not support functional programming directly. Metaprogram implementors are forced to use alien techniques and extremely intricate syntax to implement

their own concepts. This often leads to cryptic, unmanageable and fragile code as the sample above.

In this Section we propose a functional programming interface for C++ template metaprograms. Using this idea metaprogram developers write embedded Haskell code to express compile time algorithms and data structures inside C++ host language. These Haskell fragments are automatically translated to native C++ code, which then can be compiled by any standard compliant C++ compiler. Haskell snippets can communicate with the surrounding C++ environment and – via the host language – to each other.

With the help of such a translator we can write the previous prime-decider program in the following way:

```
#include <lambda.h>

__BEGIN(Haskell)
  divides b a = (a `mod` b == 0)
  hasDivider n from to = (from <= to) && ((divides from n) ||
                         (hasDivider n (from + 1) to));
  isPrime 1 = False;
  isPrime n = not (hasDivider n 2 (n `div` 2));

  main = print (isPrime 1);
__END(Haskell)

#include <iostream>

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

To implementing this idea we use a step-wise transformation using intermediate languages. Intermediate languages not only make the implementation more stable but also useful to execute everyday tasks, like debugging. Our experimental transformator uses *Yhc.Core*, the York Haskell Compiler's core language [55] as the first intermediate language, and *Lambda* language, our own language to express lambda expressions [29] as the second intermediate language. Therefore the transformation takes three major steps. First Haskell code is translated to *Yhc.Core* with the Yhc compiler. In the second step the Yhc.Core is adjusted to our *Lambda* language. In the last step, Lambda is used to generate standard compliant C++ source. Then users may compile the final result with any recent C++ compiler.

There are other possible transformation schemas. Instead of Yhc, one can consider using the Glasgow Haskell Compiler [54] to utilize better parsing possibilities.

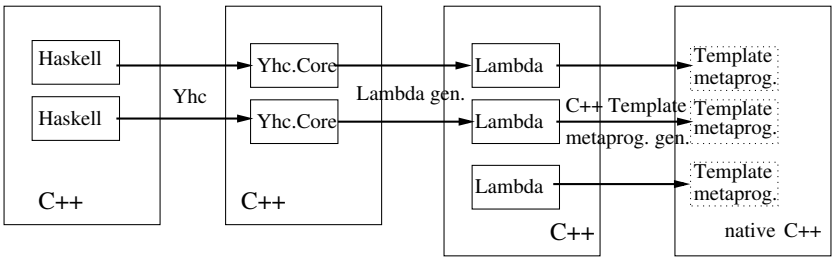

Fig. 8. Transformation schema of embedded Haskell to template metaprograms

Another experimental project uses *EClean* a subset of the *Clean* functional language [15, 23]. A Clean to Template Metaprogram translator has been written, and tested on varoius applications [30].

7.1 Generating Yhc.Core code

Yhc.Core [16, 55] is a core Haskell-like language all Haskell programs can be expressed in. It uses a small amount of structures making it easy to process programs further. Haskell programs can be transformed into Yhc.Core using the York Haskell Compiler using the `--showcore` argument. It generates a human readable code which is easy to use for further processing. The Core language can be treated as a subset of Haskell with restrictions:

- Case statements examine their outermost constructor
- Does not contain type classes
- Does not contain `where` statements
- Has only top level functions
- Fully qualified names
- Constructors and primitives are fully applied

Currently lambda expressions are guaranteed not to appear in the output of the Haskell to Core transformation. The syntax of Yhc Core is found in [16].

7.2 Generating lambda expressions

We defined our Lambda language to express lambda expressions in a handy way. Lambda is a full-featured language. Programmers may embed Lambda code into C++ [29] and generate C++ template metaprograms. However, in this case Lambda is used as an intermediate language.

We use the definition of non-typed enriched lambda expressions from [22]. We express the λ symbol with the \ character. As you can see our solution supports naming lambda expressions. The syntax is the following:

```
<named lambda expression> ::=
  __lambda <name> = <expression>;

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

Decimal numbers and built-in operators are valid constants. Supported operators are: $+$, $-$, $*$, $/$, %, $<$, $>$, $<=$, $>=$, $<>$, $=$, \$. (These operators have the usual meanings, % is modulo and \$ is the fix point operator). We restrict the form of a general lambda abstraction allowing only one variable, i.e. the expression `\xy.E` should be written in form of `\x.\y.E`. This restriction doesn't affect expressiveness.

The code generated by Yhc contains a list of function definitions. Each function definition is converted into a named lambda expression with a corresponding name. Functions taking arguments are converted into lambda abstractions: a new abstraction is introduced for each argument of the function. These lambda abstractions wrap each other in their order of appearance in the argument list. The lambda abstraction generated for the leftmost argument is the outermost. The innermost lambda abstraction encapsulates the body of the function.

Function applications are handled by our lambda expressions. The `let` expressions and the `case` expressions are transformed into lambda expressions supported by our syntax based on the transformation techniques described in [22].

7.3 Generating template metaprograms

We have another tool transforming lambda expressions into C++ template metaprograms which can be compiled by any standard C++ compiler. These metaprograms have access to directly implemented template metaprograms making interoperability between lambda expressions (and Haskell functions) and directly implemented template metaprograms possible.

During the execution of the generated template metaprograms the C++ compiler builds the graph of the expression and reduces it lazily. Our compiler compiles named lambda expressions into C++ classes (metafunction classes [1]) implementing the lambda expression. The names of the classes are the names of the lambda expressions indicating that names have to be valid C++ names. Since these expressions are translated into C++ classes they can be at any part of the code where classes can be defined [3] indicating that Haskell code can be embedded at any part of the C++ code where classes can be defined.

Lazy and eager evaluation Our compiler supports lazy evaluation of lambda expressions: every (sub)expression is evaluated only when it's value is needed.

It makes implementation of infinite data structures (such as infinite lists) possible. Eager evaluation is supported by the classes implementing the lambda expressions in C++ but are not supported directly in the lambda expressions themselves: they are always evaluated lazily indicating that Haskell functions are always evaluated lazily.

Currying Currying is supported: when the number of elements applied to a function symbol is less than the number of elements required by the function symbol the result is a new function symbol. For example: we have an anonymous function requiring two elements to be applied to it: `\x.\y. + x y`. When only one element is applied to this function the result is a new function requiring one element to be applied to it. `(\x.\y. + x y) 5` is equivalent to `\y. + 5 y`.

Haskell function applications are translated into applications of lambda expressions indicating that the template metaprograms generated from Haskell functions support currying and are evaluated using currying.

Constants Constants are implemented by a class. Currently two types of constants are supported: integral constants and types. Types are implemented by themselves, for example the type `int` is implemented by `int`. Integral constants are implemented by a wrapper class, such as the wrappers from `boost::mpl`. Currently Haskell code can't reference types, it has access to integral constants only.

Lambda abstractions Lambda abstractions are implemented by metafunction classes whose embedded `apply` metafunction takes exactly one argument. The name of the argument is the name of the variable the lambda abstraction bounds.

For example here is a lambda expression and it's implementation:

```
// The lambda expression
__lambda I = \x. y;

// It's implementation
struct I {
  template <class x>
  struct apply {
    typedef y type;
  };
};
```

Variables Variables are implemented by their name. A name symbol from the lambda expression becomes a name symbol in C++. Binding of the names in lambda abstractions is done by the C++ compiler. As we could see it in the previous example the lambda expression `y` becomes `typedef y type` in the C++ template metaprogram. The example has a lambda abstraction binding `x`. This

lambda abstraction is represented by a template metafunction taking one argument called `x`. When this metafunction is instantiated the `x` symbols in it's body (if there are any) are replaced by the class the metafunction is instantiated with.

Eagerly evaluated applications Eager application of a lambda expression to a lambda abstraction is implemented by the evaluation of the `apply` metafunction. The C++ compiler does β conversion during the instantiation because the name of the bounded variable is the name of the argument of the nested `apply` metafunction (and the variables are implemented by their names).

The `I` lambda expression defined in the previous code example can be evaluated either in an eager or lazy way. To specify eager evaluation, the user should use the following C++ construct:

```
typedef I::apply<I>::type ApplicationOfIToItself;
```

Currying in built-in functions Built-in in functions (such as the arithmetical or logical operators) have more than one arguments. Their implementation has to support currying. They have to be implemented as lambda abstractions. For example applying an element to the plus operator has to evaluate to another lambda abstraction, applying another element to that has to evaluate to a constant (and the value of it has to be the sum of the arguments). It can be implemented easily using nested types and templates. As an example here is the implementation of the plus operator:

```
struct OperatorPlus {
  template <class a>
  struct apply {
    struct type {
      template <class b>
      struct apply {
        // ... implementation of addition,
        // possibly by boost::mpl
      };
    };
  };
}
```

We assume that every built-in function supports partial evaluation (to a lambda abstraction).

7.4 Lazy application

Applications in lambda expressions (and in Haskell) are evaluated only when their value is needed, they can't be translated into eager applications. We use the following template to implement lazy application:

```
template <class left, class right>
struct Application {};
```

Using this metafunction lazily evaluated template expressions can be built as binary trees of applications: the instances of the `Application` template represent the application nodes of the tree, the `left` and `right` arguments represent the sub trees of the application nodes.

We define a metafunction implementing reduction of expressions to weak head normal form [5]. Standalone lambda abstractions, constants and built-in functions are in weak head normal form. Lazy applications are never in weak head normal form, since we assume that every built-in function supports partial evaluation. These considerations simplify the reduction algorithm:

```
while (the top level element is a lazy application)
  reduce the left side of the top level element to
    weak head normal form
  evaluate the top level application
```

We implemented this in a metafunction called `Reduce`:

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

template <class left, class right>
struct Reduce< Application<left, right> > {
  typedef
    typename Reduce<
      typename
      Reduce<left>::type::template
      apply<right>::type
    >::type  type;
};
```

The general case handles lambda expressions which are already in weak head normal form, there is a specialization of the template for reducing lazy applications in normal order reduction: it reduces the left sub-expression of the application to weak head normal form (`typename Reduce<left>::type`) after which the left side is in weak head normal form, so the next redex is this application:

```
typename Reduce<left>::type::template apply<right>::type
```

Finally the resulting expression is reduced as well.

7.5 Interoperability with directly implemented C++ metafunctions

Lambda expressions are translated to their C++ equivalents. The generated code is valid C++ sources with template definitions. Such templates can be written directly, without implementing their Lambda equivalents. Directly implemented Lambda expressions can be used in generated Lambda expressions as constants. For example:

```
struct DirectLambdaExpression {
  // implementation...
};

__lambda f = \n. DirectLambdaExpression 2 n;
```

It makes extension of the built-in operators possible and parts of the expressions can be implemented using other techniques.

Lambda expressions can be used by directly implemented C++ template metaprograms as well since lambda expressions are compiled into template metaprograms. After they are compiled into template metaprograms there is no difference between a directly implemented lambda expression and a compiled one: the compiled one can be used as a directly implemented one. Lambda expressions can be used as built-in functions in other lambda expressions, for example:

```
__lambda add = \a.\b. + a b;
__lambda f = \n. * n (add 6 7);
```

Lambda expressions can be used in their own definition simplifying the creation of recursive expressions:

```
__lambda rec = \n. (< n 1) 13 (rec (- n 1));
```

Due to the visibility rules of C++ [3] lambda expressions are visible after their declaration. For example the following code wouldn't compile because `b` is defined after `a`:

```
__lambda a = \n. b n;
__lambda b = \n. + 1 n;
```

Our compiler supports forward declaration of lambda expressions by ensuring that every lambda expression compiled to C++ will be implemented as a `struct`. In the previous example `b` can be declared before `a` is defined:

```
struct b;
__lambda a = \n. b n;
__lambda b = \n. + 1 n;
```

Haskell functions are visible in the whole Haskell block, to support this our Yhc.Core to lambda expression transformation tool adds forward declaration of the named lambda expressions to the beginning of each lambda expression list generated from an embedded Haskell block. Note that this makes functions visible to each other within an embedded Haskell block. Visibility of functions defined in separate Haskell blocks depend on the C++ visibility rules [3] because Haskell functions are transformed into C++ classes.

7.6 Evaluation

Ideally, the syntax of a programming language should match the paradigm the program is written in. Template metaprogramming, a Turing-complete subset of the C++ language, is many times regarded as a pure functional language. Unfortunately, the current way of writing metaprograms is far from the ideal, mainly due to the complicated template syntax and the different original design goals of C++.

In this section we described a method which makes metaprogram developers able to express their intentions directly in functional style using Haskell syntax. Haskell code snippets are embedded into the C++ program and are translated into native C++ code. The translation process uses a stepwise approach; and the last step generates C++ template metaprogramsi which could be compiled by any standard conformant C++ compiler.

We have shown that using embedded Haskell simplifies template metaprograms, make them easier to write and maintain. The developer can focus on the functionality of the metaprogram, reusing a huge number of existing algorithms and data structures implemented as Haskell libraries make them available to the C++ metaprogramming community.

8 Related work

8.1 FC++

FC++ is a C++ library providing runtime support for functional programming [20]. Using the tools the library provides functional programs can be written in C++ from which the expression graph is built and evaluated at runtime. They don't require any external tool (such as a translator) they use standard language features only. The library focuses on runtime execution.

8.2 Boost metaprogramming library

Boost has a template metaprogramming library called `boost::mpl` which implements several data types and algorithms following the logic of STL [14]. Our solution is designed to be compatible with it (the lambda expressions produced by our compiler are designed to be template metafunction classes taking one argument).

`Boost::mpl` has lambda expression support: the library provides tools to create lambda abstractions easily: placeholders (`_1`, `_2`, etc.) are provided and arguments of metafunctions can be replaced by them. The result of evaluating a metafunction with one (or more) placeholder argument is not directly usable, a metafunction called `lambda` generates a metafunction class from them. Using these lambda abstractions partial function applications can be implemented, but since `lambda` bounds every placeholder lambda abstractions with other lambda abstractions as their value can't be defined. For example $\lambda x.\lambda y.+xy$ can't be expressed (and neither can be the `Y` fixpoint operator).

8.3 Boost lambda library

Boost has a library for implementing lambda abstractions in C++ [51]. It's main motivation is simplifying the creation of function objects for generic algorithms (such as STL algorithms). With the library function objects can be built from expressions (using placeholders). The lambda abstractions built using this library can be used at runtime.

8.4 Haskell type classes

Zalewski et al. defined a mapping from generic Haskell specifications to C++ with concepts [43]. Haskell multi-parameter type classes with functional dependencies have been translated to ConceptC++, an experimental implementation of the concept feauture of C++0x. The translation process consists of three major parts: the division of Haskell class variables i nto ConceptC++ concept parameters and associated types, the corresponding division of superclasses in the context of a type class, and the flattening of Haskell AST to the concrete syntax of ConceptC++. The main motivation of the authors was to model software components in Haskell and implemented in C++ automated the translation.

8.5 Debugging and profiling

Template metaprogramming was first investigated in Veldhuizen's articles [40]. Static interface checking was introduced by McNamara [21] and Siek [26]. The compile-time assertion appeared in Alexandrescu's work [2]. Vandevoorde and Josuttis introduced the concept of a *tracer*, which is a specially designed class that emits runtime messages when its operations are called [37]. When this type is passed to a template as an argument, the messages show in what order and how often the operations of that argument class are called. The authors also defined the notion of an *archetype* for a class whose sole purpose is checking that the template does not set up undesired requirements on its parameters. In their book [1] Abrahams and Gurtovoy devoted a whole section to diagnostics, where the authors showed methods for generating textual output in the form of warning messages. They implemented the compile-time equivalent of the aforementioned runtime tracer (`mpl::print`, see [50]).

9 Conclusion

Ideally, the syntax of a programming language should match to the paradigm the program is written in. Template metaprogramming, a Turing-complete subset of the C++ language for implementing compile-time algorithms via cleverly placed templates, is many times regarded as a pure functional language. Unfortunately, the current way of writing metaprograms is far from the ideal, mainly due to the complicated template syntax and the different original design goals of C++.

In this paper we gave a brief and noncomplete introduction to C++ templates and C++ template metaprogramming. We learned the base techniques of writing

metaprograms, and using a motivating example we followed how deeply can we automatize code adoption using metaprograms.

10 Examples

The examples come here.

References

1. David Abrahams, Aleksey Gurtovoy: C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley, Boston, 2004.
2. Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
3. ANSI/ISO C++ Committee. Programming Languages – C++. ISO/IEC 14882:1998(E). American National Standards Institute, 1998.
4. T. H. Brus, C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer, CLEAN: A language for functional graph rewriting, Proc. of a conference on Functional programming languages and computer architecture, Springer-Verlag, 1987, pp.364-384.
5. Zoltán Csörnyei and Gergely Dévai, An introduction to the lambda-calculus, Lecture Notes in Computer Science, Springer-Verlag, LNCS Vol. 5161, pp. 87-111 ISSN 0302-9743, ISBN 3-540-88058-5
6. Olaf Chitil, Zoltán Horváth, Viktória Zsók (Eds.): Implementation and Application of Functional Languages, Springer, 2008, [273], ISBN: 978-3-540-85372-5
7. K. Czarnecki, U. W. Eisenecker, R. Glck, D. Vandevoorde, T. L. Veldhuizen, *Generative Programming and Active Libraries*, Springer-Verlag, 2000.
8. Krzysztof Czarnecki, Ulrich W. Eisenecker: Generative Programming: Methods, Tools and Applications. Addison-Wesley (2000)
9. Ulrich W. Eisenecker, Frank Blinn and Krzysztof Czarnecki: A Solution to the Constructor-Problem of Mixin-Based Programming in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
10. David Flanagan, Yukihiro Matsumoto: The Ruby Programming Language O'Reilly Media, Inc. (January 25, 2008) ISBN-10: 0596516177, ISBN-13: 978-0596516178
11. Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock: A Comparative Study of Language Support for Generic Programming. Proceedings of the 18th ACM SIGPLAN OOPSLA 2003, pp. 115-134.
12. Yossi Gil, Keren Lenz, Simple and Safe SQL queries with C++ templates In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.
13. Zoltn Juhsz, dm Sipos, Zoltn Porkolb, Implementation of a Finite State Machine with Active Libraries in C++. In: Ralf Lammel, Joost Visser, Joao Saraiva (Eds.): Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Lecture Notes in Computer Science 5235 Springer 2008, ISBN 978-3-540-88642-6., pp. 474-488.
14. Björn Karlsson: Beyond the C++ Standard Library, A Introduction to Boost. Addison-Wesley, 2005.

15. P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers, Functional programming in Clean, 2002
16. N. Mitchell, C. Runciman: A Supercompiler for Core Haskell. In Chitil et al. Implementation and Application of Functional Languages: 19th International Workshop, IFL 2007, Freiburg, Germany, September 27-29, 2007. Revised Selected Papers, Springer-Verlag, Berlin, Heidelberg, 2008
17. David R. Musser and Alexander A. Stepanov: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.
18. David R. Musser and Alexander A. Stepanov: The Ada Generic Library: Linear List Processing Packages. Springer Verlag, New York, 1989.
19. Nathan Myers: Traits: a new and useful template technique. C++ Report, June 1995.
20. B. McNamara, Y. Smaragdakis: Functional programming in C++, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.
21. Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
22. Simon L. Peyton Jones: The Implementation of Functional Languages. Prentice Hall, 1987, [445], ISBN: 0-13-453333-9 Pbk
23. R. Plasmeijer, M. van Eeekelen, *Clean Language Report*, 2001.
24. Zoltán Porkoláb, József Mihalicza, Ádám Sipos, Debugging C++ template metaprograms, In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM 2006, ISBN 1-59593-237-2, pp. 255-264.
25. Douglas Gregor, Jaakko Järvi, Jeremy G. Siek, Gabriel Dos Reis, Bjarne Stroustrup, and Andrew Lumsdaine: Concepts: Linguistic Support for Generic Programming in C++. In Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.
26. Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
27. Jeremy Siek and Andrew Lumsdaine: Essential Language Support for Generic Programming. Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, NY, USA, pp 73-84.
28. Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.
29. Ábel Sinkovics, Zoltán Porkoláb: Expressing C++ Template Metaprograms as Lambda expressions. In Tenth symposium on Trends in Functional Programming (TFP '09, Zoltán Horváth, Viktória Zsók, Peter Achten, Pieter Koopman, eds.) Jun 2 - 4, Komarno, Slovakia 2009., pp. 97-111.
30. Ádám Sipos, Zoltán Porkoláb, Viktória Zsók: Meta<fun> – Towards a functional-style interface for C++ template metaprograms, In Frentiu et al ed.: Studia Universitatis Babes-Bolyai Informatica LIII, 2008/2, Cluj-Napoca, 2008, pp. 55-66.
31. Ádám Sipos: Effective development of C++ Template Metaprograms. PhD thesis. Eötvös Loránd University, Budapest, Hungary, 2009.
32. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
33. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)

34. Gabriel Dos Reis, Bjarne Stroustrup: Specifying C++ concepts. Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006: pp. 295-308.
35. M. Torgersen, C. P. Hansen, E. Ernst, P. Ahe, G. Bracha, N. Gafter: Adding Wildcards to the Java Programming Language Proceedings of the 2004 ACM Symposium on Applied Computing (SAC) 2004, pp. 1289-1296.
36. Erwin Unruh: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.
37. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)
38. Todd L. Veldhuizen and Dennis Gannon: Active libraries: Rethinking the roles of compilers and libraries. In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientic and Engineering Computing (OO'98). SIAM Press, 1998 pp. 21–23
39. Todd Veldhuizen: Five compilation models for C++ templates. In First Workshop on C++ Template Metaprogramming, October 2000
40. Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, 1995, pp. 36-43.
41. Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.
42. T. Veldhuizen, C++ Templates are Turing Complete
43. M. Zalewski, A. P. Priesnitz, C. Ionescu, N. Botta, and S. Schupp, Multi-language library development: From Haskell type classes to C++ concepts. In MPOOL 2007 Ecoop workshp, 2007.
44. István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830 (2003), pp. 209 - 227.
45. István Zólyomi, Zoltán Porkoláb: Towards a template introspection library. LNCS Vol.3286 pp.266-282 2004.
46. The Blitz++ library.
www.oonumerics.org/blitz
47. Boost Concept checking.
http://www.boost.org/libs/concept_check/concept_check.htm
48. The boost xpressive regular library.
http://www.boost.org/doc/libs/1_40_0/doc/html/xpressive.html.
49. Boost tr1 mathematical functions.
http://www.boost.org/doc/libs/1_40_0/libs/math/doc/html
50. Boost Metaprogramming library.
http://www.boost.org/libs/mpl/doc/index.html
51. The boost lambda library.
http://www.boost.org/doc/libs/1_40_0/doc/html/lambda.html
52. Boost Preprocessor library.
http://www.boost.org/libs/preprocessor/doc/index.html
53. Boost Static assertion.
http://www.boost.org/regression-logs/cs-win32_metacomm/doc/html/boost_staticassert.html
54. Glasgow Haskell Compiler.
http://www.haskell.org/ghc/
55. York Haskell Compiler.
http://community.haskell.org/~ndm/yhc/