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 capability of C++ was not a primary design goal, the language is not capable of clean 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, existing libraries do not follow these requirements. In this paper we give a short and incomplete introduction to C++ template mechanism and the basics of template metaprogramming. We want to 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, 25]. They enable data structures and algorithms be parameterized by types thus capturing commonalities of abstractions at compilation time without performance penalties at runtime [29]. *Generic programming* [23, 22, 15], is recently a 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 is the Standard Template Library [15] 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. In 1994 Erwin Unruh wrote a program

[28] in C++ which didn't compile, however, the error messages emitted by the compiler during the compilation process displayed a list of prime numbers. 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 [30]. These compile-time programs are called C++ *Template Metaprograms* and later has been proved to be form a Turing-complete sublanguage of C++ [8].

Template metaprogramming is now an emerging new direction in C++ programming for executing algorithms at compilation time. 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 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 often hard to read, and 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 of some other applications in Section 5. Practical metaprogramming is presented in Section 6 on an example. Related works are discussed in Section 7.

2 Informal introduction to templates in C++

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 *list* or a *Matrix* of elements of 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, thus this *generic* language construct aids code reuse, and the introduction of higher abstraction levels. This method of code reuse often called *parametric polymorphism*, to emphasize that here 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 often simplify the complex rules of templates for sake of general understanding of the whole mechanism. Those, who are interested in the detailed rules a fundamental source is [29]. For language lawyers the best source is the C++ standard itself [3]. We will be lazy in other syntactical parts too, often omitting headers like `<iostream>`, and namespace tags. For the full, syntactically correct examples see the sample notes.

Let suppose we have to compute the maximum number 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 overloading for other types..
```

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

Moreover, what can we do the types we haven't implemented *yet*? If somebody creates a new type with a well-defined less-then operator to compare the objects, we have to sit down and 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 would be look like. Strongly typed programming languages allows writing programs using only existing types.

We may yield to the temptation to try out non type safe solutions. For a C/C++ programmer a precompile macro seems to solve the problem:

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

As precompile macro functions are *typeless*, this will work not only the existing types, but on every type too defined later. Unfortunately, precompiler macros are not the answer for writing generic algorithms over types. Apart, that precompiler macros are replaced before the run of the C++ compiler, therefore we will encounter a huge number of side-effects and type-safety problems, the attempt to solve more complex problems with macros is desperate.

To demonstrate this, let suppose to 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 swap via the temporary variable `temp` which has the same type as the parameters.

Now we are in a trouble. Since precompiler macros are replaced before the C++ compiler starts, we cannot use any type inference information from C++ compiler. We are not able to detect 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 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;
}
```

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

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

First, we have to understand that templated `swap` and `max` are not functions in the traditional sense. They are not compiled and not called under execution. Templates are rather skeletons, describing manufacturing process of real functions instantiated by the compiler in an automated way under the compilation process. Thus we called them: *function templates* rather then *template functions*.

This automated instantiation process is the most remarkable flavor 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)
```

In the first step, the compiler have 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: here based on the types of variables `i,j,x,y,ci`. This process is called *template parameter deduction*. In the first and third call of `max` lead to calling 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 is created, and will be called. Let recognized, 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 suppose the template parameter type `T` to be `int`. However, the second argument `x` contradict, suggesting a `double` parameter. Therefore the parameter deduction process will fail and the compiler raise a syntax error.

How can we fix this problem? As you might expect, templates may be defined with two or more type parameters. Here 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 expected. 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`, it will be `int` as well as the type of the first parameter. When the function called in run-time, `a > b` evaluated as false, correctly, and `3.14` is about to return. However, as the return type has been decided in compile time as `int`, this value is converted to integer, and thus we got 3 assign to `z`. It does not help, that this value converted to type `double` as `3.0` when stored into `z`. It is clear, that any attempt to change the role of parameters `T` and `S` could lead the same result.

Can we define 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 in compile time: in run-time we cannot change them anymore. As templates are totally compile time language fenomenon, once the template parameter deduction decides template parameters, these decisions are final. In the same time, whether the first or the second argument of the `max(i,x)` call is greater, is a completely run-time property. Compilation time and run-time are fundamentally separated in strongly typed compiled programming languages.

Even if we understand this, it is a bit asthonishing. Looking at the actual code it seems trivial that a max function called by an integer and a double argument is better to return by double type. Why we cannot achieve this?

The problem is, that when speaking about templates, not only two stages – compile time and run-time – of the full process we have to take into consideration, but the very first one too: the *design time* of the template function. When we had defined templated `max(T,S)` with two distinct type parameters, `T` abd `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 return value. At that point we had no information whether the actual arguments in a call environment would be type of `int`, `double` or anything else. We still had to make final decisions. In the second stage, when the compiler compiles the actual code of `max(i,x)`, it sees the environment of the call, recognises the actual types of `i` and `x`, but cannot overrule the decisions we made in template definition time. Finally, in run-time the program works with a given set of types and rules, able to decide, whether `i` or `x` is greater, but unable to overrule the type rules.

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

Table 1. Programming with templates

The two fundamental problem here is (1) the gap between design time and compilation time: this inhibits to choose the "better" return type from `int` and `double`, and (2) the gap between compilation and run-time: this inhibits to choose the type of the greater argument 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.

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

We may attempt 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;
}
```

The parameter deduction here will fail, as there is no information about type `R`. There is a number of reasons why template parameters are not deducted from return values, but to understand risks just consider this 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), there is no 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);
```

This works, but ugly and possibly misleading. The C++ standard committee recognized this requirement, and allowed an official, syntactically more readable notation:

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

This syntax 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 set the parameters by hand.

We can specialize templates even 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()` will work improper way comparing the pointer values, rather than the contents of the char arrays. Here we provide *user specialization* for defining an exceptional behavior of the maximum algorithms for character arrays.

Different template definitions may exist with the same name: overloading of templates are possible. Hence, we may define the one parameter

```
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);
```

in the same time. When instantiating, the compiler will choose the *most specific* version of template definitions applicable for the actual call.

The following code snippet defines a class template:

```
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, but member functions like `rows()` and `cols()` are also templated.

As object constructors' parameters are say nothing about class template parameters, objects of class templates are instantiated explicity 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` is not neccessary gives what you expect. In some implementions bools have size of 4 bytes (for compatibility with `int` type). Even if `sizeof(bools)==1`, we can work out a better implementation storing 8 bools on every single byte.

Naturally, this economic 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. It is possible, but generally a very bad idea to change the public interface between specializations.

With a *partial specialization* we can record one or more arguments types (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 correctly compiled, but prime numbers were printed at compile-time as error messages.

```
// 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 useful distinction here is that between programs and 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 in the sense that they are self-containing.

The canonical template metaprogram to show basic behaviour is the compile time evaluation of factorial numbers. Let 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 specialization for 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 understand 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 this full specialization is used be used to generate the requested class, and instantiation of `Factorial<1>` completes.

From this point we are coming back from recursion, `Factorial<1>::value` is used to finish `Factorial<2>`, etc... The suspended instantiations continue in the reverse order. At the end, we result in with 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 silently used here: (1) Templates wich are not referred wont be instantiated – C++ template mechanism is *lazy.* (2) Constant expressions – those can be evaluated in compilation time, like the initialisation of enumeration `value` must be evaluated in compilation time.

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;
  retrun 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. It is also interesting, that the first parameter is a (constant) value, the rest 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 specializing 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 the exsistence of conditional and looping control structures. C++ template metaprograms form a Turing complete programming language executed in compilation time [32].

Now we can revisit the `max()` function:

```
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. When this template is instantiated with arguments `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.

Before going forward, it is important to understand two things: First, we cheated a bit. The "widest" type – which has the gretaer `sizeof` value – is not neccessary the best return type. Sometimes the size of a class is unrelated 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 still do not able to choose the type of the greater value, we choose the type which seems better under compilation. It is still possible that `double` has been chosen as return type, but the run-time value `int` argument is greater. 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 that we decided the return type not in design time, but in compilation time, when the template has been instantiated. We delegated an algorithm written is design time and executed in compilation time which – based on the actual types of the template arguments – were able to select the better return type. This has happened in an automated way by the execution of a small and simple template metaprogram.

Design time	Compilation time	Run-time
Design of algorithm	Template instantiation	Run of the algoritm
Template metaprogram IF is written	IF is "executed" and defines return type of max()	Greater argument value is chosen to return

Table 2. Programming with template metaprograms

4 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 and metaprograms. The following table describes the metaprogram, and runtime program entities in parallel.

C++ template metaprogram actions are defined in the form of template definitions and 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 new 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.

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`. Data and even data structures can be expressed in template metaprograms with constructs like *typelist* [2].

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 favorite implementation forms of expression templates [31]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [13].

There is a clear relationship between a number of entities. C++ template metaprogram actions are defined in the form of template definitions and 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.

Metaprogram	Runtime program
(template) class	subprogram (function, procedure)
static const and enum class members	data (constant, literal)
symbolic names (typenames, typedefs)	variable
recursive templates, typelist	abstract data structures
static const initialisation enum definition type inference	initialisation *(but no assignment!)*

Table 3. Comparison of runtime and metaprograms

Data is expressed in runtime programs as constant values or literals. In metaprograms we use static const and enumeration values to store quantitative information. Results of computations under the execution of a metaprogram are stored either in new constants or enumerations. Furthermore, execution of metaprograms can cause new types be created. Types hold information that can influence the further run of the metaprogram.

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 form of expression templates [31]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [38].

In Loki, we define *typelist* as the following recursive template:

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

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

We can use helper macro definitions to make the syntax a bit better 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;
```

Being a compile time functional language and using a list data structure it is really easy to define the essential helper algorithms:

```
/
//  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` takes a parameter and returns the position of that parameter in the list. If the actual argument is not found it returns -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`.

However, there is a fundamental difference between runtime programs and C++ template metaprograms: once a certain entity (constant, enumeration value, type) has been defined, it will be immutable. There is no way to change its value or meaning. Therefore no such thing as a metaprogram assignment exists. 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.

Metafunctions – as we can expect in a functional programming language – are first class citisens in C++ template metaprogramming. In the following example we define a metaprogram `Accumulate` which summarize the value of a function at points in the intervall 0..N. The function will be a metaprogram itself and 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;
  }
```

5 Applications of template metaprogramming

5.1 Expression templates

The eraliest 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 foillowing scenario.

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

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

a = b + c + d;
```

Unfortunatelly, when we execute the operation above, some uneffective events 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 Arrays will allocate space in the heap – a relatively slow operation, and will copy a not small number of bytes. Not to forget the destruction, we end up something similar to the following:

```
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 operations, etc. Meanwhile, a FORTRAN-like code could keep the high performance with impolementing the following:

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

C++ *Expression templates* [31] are templates that represent an arbitrary expression, so that they can easily be passed to functions. These expression arguments can be transformed into *inline* i within the function body, resulting in faster, more efficient code compared to the classic C-style *callback* functions. In the C language expressions are usually passed as function pointers referring to the desired callback function. In the standard C library the `qsort()`, `lsarch()` functions expect a `int (*cmp)(void*, void*)` function pointer be passed referring to a comparison function. Expression templates are a more enhanced substitution for callback functions.

5.2 Concept checking

C++ has no language-level support for explicitly requiring certain properties, *concepts* (e.g. a comparison function for the type, `operator<`) from template arguments when instantiation of a template is being done. This deliberate language design decision that any type can be a template's type argument regardless of its properties made C++ templates more flexible and applicable, as opposed to more strict languages, like Ada. However, if a type fails to meet the implicitly declared requirements of a template, the compilation will fail, and results in complex and long error messages in the case of a heavily templated code, like the STL [18, 21]. Because of the lack of compiler support, the problem had to be remedied on the language level. Complex language constructs were created to determine in compile-time the characteristics of a type used for instantiation. This area of research is called *concept checking.*

A compilation of such language constructs, the *Boost Concept Checking Library (BCCL)*[36] 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;
}
```

In the last ten years lots of effort spent to develop high quality concept libraries. Unfortunately, it turned out that library-based solutions are significantly weaker than language-based concepts. Therefore the ANSI C++ committee accepted a proposal to extend C++ with language-based concepts. With the help of concepts [20] we can define the requirements agains template parameters of classes and functions.

5.3 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 often incorporate with 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 the use of template metaprograms.

AraRat system [12] is an example to implement 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 [37] is used for compile time checking of regular expressions. In most regex library, the patterns are represented as string literals or variables and the syntax of the regular expressions (i.e. every parantheses has a closing symbol, etc.) are checked only in run-time. The `boost:xpressive` library allows the creation and compile time checking certain regular expressions. This way we can detect soem syntactically bogus patterns in compilation time.

6 The matrix example

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 assignmnet operator to ensure the meaningful copy of `matrix` objects. The textbook example for such assignment operators look 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];
}
```

There will be a serious error to replace the loop in `copy()` function with a bitwise copy, like the one `memcpy()` implements. Since type argument `T` could be any (copyable) type, in the generic solution we have to call the assignment operator of type `T` to copy the content of the matrix. Exactly this happens in `v[i] = other.v[i]`.

However, with the loop we created a *safe* copy, in most of the cases we will store `double` elements in the matrix, which is completely safe to copy with *memcpy()*. 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: the specialization. Let 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 optimal way.

To modularize type-specific codes we can use *trait*s.

```
template <typename T>
struct copy_trait
{
  static void copy( T* to, const T* from, int n)
  {
    for( int i = 0; i < n; ++i ) // generic
      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));  // specific
  }
};
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 `matrix` to describe the expected behaviour in case of copying the object. When the template is about to instantiated with a certain argument `X`, the second argument will be `copy_trait<X>`. As the generic version of `copy_trait<>` using loop-based copy, this will be the default. However, we may specialize the copy for type `X`. We still have to repeat specializations, but at least we can modulize.

We can consider, that the copy we execute on `long` and `double` is the same code. We do not really have to distinguish them. We only have to know that they are POD types.

```
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 types...
template <typename T, bool B>
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
{
  // ...
};
```

Here we impoved the solution separating two policies: copying POD types, and non-POD types. The `is_pod<>` template declares whether a type is POD or not. When a type is declared in `is_pod<>` (in either way) the compiler automatically decides which `copy_trait<>` have to be used.

But we can still improve the solution using typelists. In the following solution we simply declare the POD types and everything else is made automatically.

```
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
{
  //...
};
```

Moreover, using `boost::type_traits` library, we can apply the `is_pod<>` template, therefore even the typelist could be omitted.

7 Related work

7.1 FC++

FC++ is a C++ library providing runtime support for functional programming [17]. 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.

7.2 Boost metaprogramming library

Boost has a template metaprogramming library called `boost::mpl` which implements several data types and algorithms following the logic of STL [13]. 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 λx.λy.+xy can't be expressed (and neither can be the `Y` fixpoint operator).

7.3 Boost lambda library

Boost has a library for implementing lambda abstractions in C++ [39]. 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.

7.4 Haskell type classes

Zalewski et al. defined a mapping from generic Haskell specifications to C++ with concepts [33]. 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 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.

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. Björn Karlsson: Beyond the C++ Standard Library, A Introduction to Boost. Addison-Wesley, 2005.
14. P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers, Functional programming in Clean, 2002
15. David R. Musser and Alexander A. Stepanov: Algorithm-oriented Generic Libraries. Software-practice and experience, 27(7) July 1994, pp. 623-642.
16. David R. Musser and Alexander A. Stepanov: The Ada Generic Library: Linear List Processing Packages. Springer Verlag, New York, 1989.
17. B. McNamara, Y. Smaragdakis: Functional programming in C++, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.
18. Brian McNamara, Yannis Smaragdakis: Static interfaces in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
19. 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.

20. 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.
21. Jeremy Siek and Andrew Lumsdaine: Concept checking: Binding parametric polymorphism in C++. In First C++ Template Programming Workshop, October 2000, Erfurt.
22. 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.
23. Jeremy Siek: A Language for Generic Programming. PhD thesis, Indiana University, August 2005.
24. 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.
25. Bjarne Stroustrup: The C++ Programming Language Special Edition. Addison-Wesley (2000)
26. Bjarne Stroustrup: The Design and Evolution of C++. Addison-Wesley (1994)
27. 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.
28. Erwin Unruh: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.
29. David Vandevoorde, Nicolai M. Josuttis: C++ Templates: The Complete Guide. Addison-Wesley (2003)
30. Todd Veldhuizen: Using C++ Template Metaprograms. C++ Report vol. 7, no. 4, 1995, pp. 36-43.
31. Todd Veldhuizen: Expression Templates. C++ Report vol. 7, no. 5, 1995, pp. 26-31.
32. T. Veldhuizen, *C++ Templates are Turing Complete*
33. 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.
34. István Zólyomi, Zoltán Porkoláb, Tamás Kozsik: An extension to the subtype relationship in C++. GPCE 2003, LNCS 2830 (2003), pp. 209 - 227.
35. István Zólyomi, Zoltán Porkoláb: Towards a template introspection library. LNCS Vol.3286 pp.266-282 2004.
36. Boost Concept checking.
http://www.boost.org/libs/
concept_check/concept_check.htm
37. The boost xpressive regular library.
http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html.
38. Boost Metaprogramming library.
http://www.boost.org/libs/mpl/doc/index.html
39. The boost lambda library.
http://www.boost.org/doc/libs/1_39_0/doc/html/lambda.html
40. Boost Preprocessor library.
http://www.boost.org/libs/
preprocessor/doc/index.html
41. Boost Static assertion.
http://www.boost.org/regression-logs/
cs-win32_metacomm/doc/html/boost_staticassert.html