C++ Templates
Metaprogramming

'Essentials of Programming Languages'
Workshop 20504

Gilad Shamir
Supervised by Ehud Lamm

December 2004
Department of Mathematics and Computer Science
Open University of Israel
# Table of contents

1. Overview | 2
2. Introduction | 3
   2.1. Classifications of metaprogramming | 3
   2.2. Use of metaprogramming | 5
   2.3. Alternative | 6
3. Partial evaluation | 8
   3.1. Different kinds of partial evaluation | 10
      3.1.1. Online partial evaluation | 10
      3.1.2. Offline partial evaluation | 11
4. Template metaprogramming | 12
   4.1. The evolve of template metaprogramming | 12
   4.2. C++ Template basics | 12
   4.3. C++ as a two-level language | 13
   4.4. Class Templates | 14
   4.5. Expression templates | 15
5. Examples of compile time optimization | 18
   5.1. First language | 18
   5.2. Second language | 24
   5.3. Third language | 29
6. Drawbacks | 36
   6.1. Technical issues | 36
   6.2. Maintenance of static programs | 36
7. A Comparison between Template Haskell and the C++ metaprogramming
   techniques regarding DSLs | 37
   7.1. Terminology | 37
   7.2. Template Haskell DSL | 37
   7.3. C++ DSL | 38
   7.4. Comparison | 39
8. Conclusion | 42
9. Bibliography | 43
1 Overview

When templates were introduced to the C++ programming language, it was as a means to express parameterized types [6]. Today templates are used in ways that the inventors of C++ templates had not anticipated. In this paper we aim to give some insight on one of the uses of template programming, which is template metaprogramming, a technique whereby the compiler acts as an interpreter. This makes it possible to write programs in a subset of C++ which are interpreted at compile time.

This paper focuses on the implementation of template metaprograms in the C++ language and combines theory with practice.
2 Introduction

A metaprogram is a program that manipulates an object program [12]. Metaprogramming consists of “programming a program.” In other words, we lay out code that the programming system executes to generate new code that implements the functionality we really want. In a metaprogramming system, metaprograms manipulate object-programs, where an object-program is any sentence in a formal language [7]. Meta programs include things like program generators, program analyzers, compilers and interpreters.

2.1 Classifications of metaprogramming

Metaprograms fall into two categories: program generators and program analyzers [7].

A program analyzer - observes the structure and environment of an object-program and computes some value as a result. A result can be another object-program. Examples of these kinds of meta-systems are: optimizers and partial evaluation systems.

A program generator is often used to address a whole class of related problems, with a family of similar solutions, for each instance of the class. It does this by constructing another program that solves a particular instance. Usually the generated program is specialized for a particular problem instance.

There are further distinctions for program generators. Lets review some of them.

- Static versus run-time

A static program generator generates code which is then written to disk and processed by normal compilers. An example of such a program is Yacc. Runtime code generators are programs that write or construct other programs and then immediately execute the programs that they have generated. Letting the object program of a run-time program generator be also a runtime code generator, gives us a "multi-stage program".
Multi-stage programming

A multi-stage programming language provides a small set of constructs for the construction, combination and execution of program fragments. The key novelty in multi-stage languages is that they can have static type systems that guarantee a priori that all programs generated using these constructs will be well typed [2].

MetaOCaml is a multi-stage extension of the OCaml programming language [2].

The basics of programming in MetaOCaml can be illustrated with the following declarations:

```ocaml
let rec power n x = (* int -> .<int>. -> .<int>. *)
  if n=0 then .<1>. else .<x * .(power (n-1) x)>.

let power3 = (* int -> int *)
  .! .<fun x -> .(power 3 .<x>.)>.
```

Ignoring the code type constructor .<t>. and the three staging annotations brackets .<e>., escapes .e and run .!, the above code is a standard definition of a function that computes xn, which is then used to define the specialized function x3. Without staging, the last step just produces a function that invokes the power function every time it gets a value for x. The effect of staging is best understood by starting at the end of the example. Whereas a term fun x -> e x is a value, an annotated term .<fun x -> .(e .<x>.)> is not. Brackets indicate that we are constructing a future stage computation, and an escape indicates that we must perform an immediate computation while building the bracketed computation. The application e .<x>. has to be performed even though x is still an uninstantiated symbol. In the power example, power 3 .<x> is performed immediately, once and for all, and not repeated every time we have a new value for x. In the body of the definition of the power function, the recursive application of power is also escaped to make sure that they are performed immediately. The run .! on the last line invokes the compiler on the generated code fragment, and incorporates the result of compilation into the runtime system.
Homogeneous vs. heterogeneous

Homogeneous systems are systems where the meta-language and the object language are the same, and heterogeneous are systems where the meta language is different than the object language. The big difference is that only homogeneous systems can be n-level, where an n-level program can itself be a program that manipulates n-1 level of object programs. Only in a homogeneous meta-system can a single type system be used to type both the meta-language and the object language.

2.2 Use of metaprogramming

Why would metaprogramming be desirable? As with most other programming techniques, the goal is to achieve more functionality with less effort, where effort can be measured as code size, maintenance cost, and so forth. Here is a short overview of the different uses of metaprogramming:

- Performance - Rather than write a general purpose but inefficient program, one writes a program generator that generates an efficient solution from a specification. The interpretive style eases both maintenance and construction, since a single program solves many problems. The use of the parser generator Yacc is an illustrative example. Rather than using a general purpose parsing program we generate an efficient parser from a specification, i.e. a language grammar.

- Partial evaluation – Partial evaluation is a metaprogramming technique used to improve performance. Partial evaluation optimizes a program using a-priori information about some of the programs' inputs [1]. Further explanations regarding this topic will be presented in a later chapter.

- Translation - Perhaps the most common use of metaprogramming is translation of one program to another object program. The two languages can be homogeneous or heterogeneous. Example of a translator is a very famous program – the compiler.
Mobile code - Metaprogramming has been used as means of program transportation. Instead of using networks to bring the data to the program, networks are used to bring the program to the data. Because of security reasons, representations of programs are transported across the networks. These representations can be analyzed for security and safety purposes to ensure that they do not compromise the integrity of the host machines they run on. The transported programs are object programs and the analyzers are metaprogams.

Industrial use – Libraries such as Blitz++\(^1\) and POOMA\(^2\) use C++ metapograms to provide fast routines of numeric linear algebra.

### 2.3 Alternative
A common way for implementing a 'program creating a program' in many languages is macros. A macro is a part of a language which enables transforming source code to object code before passing it to the compiler [14].

The advantages of macros compared to dynamic code are:

- **Run time efficiency** - Macro calls are more efficient than calls to procedures because the source code is inlined in to the program thus avoiding the overhead of procedure calls.
- **Extending syntax structures** – The macro can sometimes enable defining new syntax structures in terms of existing syntax structures.
- **Polymorphism** – Macros enable the programmer to write generic code, its actual meaning is determined only after the code transformation. Such programming enables a certain degree of polymorphism.

There are different ways to implement macros. In the C/C++ languages exists a preprocessing standardized mechanism [14]. This mechanism enables defining macros which are expanded before the compilation stage. The C/C++ macros system is characterized amongst other things in performing transformations on the

\(^{1}\) The Blitz Project, [http://oonumerics.org/blitz/](http://oonumerics.org/blitz/)
A C++ class library for scientific computing which uses template techniques to achieve high performance.

\(^{2}\) home page of the Pooma project, [http://www.acl.lanl.gov/pooma/](http://www.acl.lanl.gov/pooma/)
text of the source program and on expanding the macro separately from the compilation from the compilation stage. The Lisp macro system presents a different approach. The macro expansion in Lisp is preformed during the compilation stage and the transformation is produced on the syntax tree which is built during compilation time. Because the syntax tree being built is produced as a data type which is known by the language it self, thus when expanding the macro it is possible to perform manipulation on this structure using the tools of the language [14]. Unlike the macro system of Lisp, that of Scheme does not present to the macro user the whole strength of the language. On the other hand exist a special language for defining macros, the pattern language. Using the language, it is possible to define the different patterns that the call to the macro will receive. For each possible pattern exists a form in which the macro call will be expanded by.

Macro systems have several severe drawbacks. Some of the important ones are the easiness of defining new syntax forms by macros, which can threaten the standard syntax of the language. There is a concern that communities of developers will define and use new syntax structures which are not part of the language standard, thus different dialect will evolve. Creating new syntax structures interfere with code readability. A code using such structures will be understood only by some users. Another drawback of macros is that macro systems always add one or more stages between the source code and the object code, thus causing difficulty for development tools which are based on the source code such as debuggers, code analyzers and syntax based editors.

Different languages provide various language components that can substitute the usage of macros. Haskell, for instance, uses a 'call by need' technique to pass parameters to procedures[14], so it is possible to define conditional structures as procedures which is not possible in languages which use 'call by value', so to implement an evaluation be need macros are used.

A designer of a language can tackle the issue of a code creating other code in a broader solution then the macro systems. For instance the MetaML language uses multi staged programming. This process of code creation is supported in the syntax and semantics of the language so macro disadvantages are overcome.
3 Partial evaluation

Partial evaluation is a metaprogramming technique used to improve performance. Partial evaluation optimizes a program using a-priori information about some of the programs' inputs. The goal is to identify and perform as many computations as possible in a program before run-time. Partial evaluation has two steps, binding-time analysis and specialization. Binding-time analysis determines which computations can be performed in an earlier stage given only the static inputs. Specialization uses the values of the static input to produce an improved program.

Partial evaluation is a high level optimization technique that, given a program text and some of its input, generates a specialized program. It is a fully automatic tool, that transforms a program by specializing it with respect to some of its inputs, considered as constants, and for which concrete values are supplied, usually by the user. By using information regarding these invariants, a specializer is able to perform a part of the program's computations at specialization time, thus reducing the amount of computations to be done at run time by the residual program. The result is the production of a faster and more efficient application.

The following is an example using the scheme language.

Message-passing is a common programming style to emulate classes and objects in Scheme [1]. A typical representation for a class is a function that maps the initial values of the instance variables to a tuple of closures, the representation of an object. The closures represent the possible messages and they share the current values of the instance variables among them. These values are not accessible otherwise, they are local to the object. Sending a message to the object is implemented by calling one of the closures with appropriate arguments. Thus, we have an instance of a programming technique that employs higher-order functions with shared local state.

Consider this Scheme code:

```scheme
;;; source program
(define-data object (object set get add))
;;; a record type with constructor "object" and selectors "set", "get", "add"
(define (main)
  (let ((counter-class
```


It defines a class of counter objects using the above encoding. The instance variable slot is always initialized to zero, so the class function is parameterless. A counter object cnt is a triple (or record) (object mset mget madd), where (set cnt) is the function that sets the counter, (get cnt) is the function that reads the counter, and (add cnt) is the function that adds to the counter. All these functions operate on the encapsulated shared state in slot.

Let’s try to specialize this program:

;;; program specialized with Similix (after assignment elimination)
(define (main-0)
  (let ((slot_1 (make-cell 0)))
    (cell-set! slot_1 21)
    (let* ((g_2 (cell-ref slot_1)) (g_3 (cell-ref slot_1)))
      (cell-set! slot_1 (+ g_3 g_2))
      (cell-ref slot_1))))

;;; program specialized with our partial evaluator
(define ($goal-1) 42)

Similix [8] removes the message dispatch, but defers all operations on the local state slot to run time. In contrast, our binding-time analysis instructs the specializer to perform all operations at specialization time. In effect, it reduces the source program to its value 42 as shown in the lower part of the figure. Since Similix 5.0 does not handle set! directly, we performed an assignment elimination transformation that introduces explicit boxing operations make-cell, cell-ref, and cell-set! by hand before submitting to Similix. This transformation is built into our specializer.
3.1 Different kinds of partial evaluation

Many different techniques can be used to achieve partial evaluation of a program, but most program specilaizers can be classified as being either online or offline [7]. This section defines online and offline specialization.

An essential task of a partial evaluator is, for a given program, to decide what computations are static and can be reduced at specialization time, and what computations are dynamic and must be postponed until run time (that is residualized).

The distinction between online and offline specializers relates to this decision phase. Online specializers make their decisions during program specialization, and are therefore free to use static values to improve their analyses. On the other hand, offline specializers traditionally perform program analyzes before specialization, which must be done independently of concrete static values.

3.1.1 Online partial evaluation

With respect to the binding time decisions mentioned above, online partial evaluations is a one step process, where static values are inspected during specialization. As a consequence, the main advantages of online specialization are a more precise beginning time separation, and a better quality of residual programs, because more information is available to the specializer.

The following code gives a simple example of a situation where a specializer can benefit from knowing static values.

```
If e1       // static
Then e2     // static
Else e3     // dynamic
```

With e1 and e2 being static and e3 dynamic, an offline specializer must classify the whole conditional dynamic, whereas an online specializer can easily classify it as static if e1 evaluates to true.
3.1.2 Offline partial evaluation

Offline specializers traditionally separate the specialization step from the binding time analysis step. First, a preprocessing phase is applied to the input program, to perform among other things its binding time analysis. The information needed for this stage is which program inputs are static, and the result is an annotated program used afterwards by the offline specializer, when concrete values of static inputs are supplied.

C++ template instantiation resembles offline partial evaluation. The C++ compiler takes template code, evaluates those portions which involve template parameters which are statically bound values, and produces the residual which is the template instance.
4 Template metaprogramming

A template metaprogram is defined as a program in the C++ language, which during compilation time performs code generation by creating code as it is semantically analyzed.

4.1 The evolve of template metaprogramming

The earliest documented example of a metaprogram written in the C++ language was by Erwin Unruh, on the C++ standardization committee in 1994\(^3\). He wrote a program that produced prime numbers. The intriguing part of this exercise, however, was that the production of the prime numbers was performed by the compiler during the compilation process and not at run time. This sort of compile-time computation that occurs through template instantiation is commonly called template metaprogramming.

4.2 C++ Template basics

The C++ template mechanism was first introduced in the language to provide support for parametric polymorphism, mainly as a way of writing families of functions and classes and as a replacement of C macros [1]. For example, we can define a generic vector as a class template with element type and size as parameters.

```
template <class T, int size> class Vector
{
    public:
        T data[size];
};
```

One way of instantiating a template is is with a typedef statement as follows:

```
typedef Vector<float,10> myVector;
```

\(^3\) Compile-Time Computation of Prime Numbers, Erwin Unruh
http://www.erwin-unruh.de/Prim.html
Now we can use myVector as a regular C++ class:

```cpp
myVector v;
```

A template can be specialized by providing a definition for a specialized implementation as follows:

```cpp
template<int size>
class Vector<bool, size>
{
    // use just enough memory for size bits
    char data[(sizeof(char)+size-1)/sizeof(char)];
};
```

Now whenever we instantiate Vector with bool as its first parameter, the latter specialized implementation will be used.

### 4.3 C++ as a two-level language

Templates are the core component of a static subset of C++, that is to say, a subset of C++ which is executed at compile time. Hence, C++ has the property to be a two-level language, or, in other words, a language that provides both static constructs (used to control code production) and dynamic constructs (used to actually produce code).

Moreover, the compiler time subset of C++ is Turing complete\(^4\) [Czarnecki and Eisencker, 2000].

We will examine some of the binding time notations regarding the C++ language. Binding time of code is inferred from binding time of data, therefore if an expression is assigned to a statically bound variable then the expression must be static (i.e. bound at compile time).

```cpp
//x is statically bound ( const ); therefore y and 3*y must be static too.
const int x = 3 * y;
```

Template parameters have static binding while function arguments have dynamic binding.

---

\(^4\) A language is said to be Turing complete when it provides a conditional construct and a looping construct.
template <int N>  // N: static data
Float pow(float x);  // x: dynamic data

The compiler must evaluate static constructs. Therefore regarding templates instantiation, the compiler takes template code, evaluates those portions which involve template parameters (statically bound values) and produces the residual which is a template instance.

4.4 Class Templates

Template Metaprogramming adds two rather unusual views of class templates: class templates as data constructors of a datatype and class templates as functions [2]. To introduce these views, we will compare them with the corresponding Haskell definitions.

Here is an example of C++ template program defining a list, followed by the corresponding Haskell code.

<table>
<thead>
<tr>
<th>C++:</th>
<th>Haskell:</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct Nil {};</td>
<td>data List = Nil</td>
</tr>
<tr>
<td>template &lt;int H, class T&gt;</td>
<td></td>
</tr>
<tr>
<td>struct Cons {};</td>
<td></td>
</tr>
<tr>
<td>typedef</td>
<td>list = Cons 1 (Cons 2 Nil)</td>
</tr>
<tr>
<td>Cons&lt;1,Cons&lt;2, Nil&gt; &gt; list;</td>
<td></td>
</tr>
</tbody>
</table>

The C++ definitions for Nil and Cons have different dynamic and static semantics. Dynamically, Nil is the name of a class, and Cons is the name of a class template. Statically, both Nil and Cons can be viewed as data constructors corresponding to the Haskell declaration on the right. All such constructors can be viewed as belonging to a single “extensible datatype”, which such definitions extend.

The C++ template instantiation mechanism provides the semantics for compile-time computations. Instantiating a class template corresponds to applying a function that computes a class. In contrast, class template specialization allows us to provide different template implementations for different argument values, which serves as a vehicle to support pattern matching, as common in functional programming languages. Next, we compare how functions are implemented at the C++ compile time and in Haskell:
The template specializations Len<Nil> and Len<Cons<H,T>> handle the empty and non-empty list cases, respectively. In contrast to Haskell or ML, the relative order of specializations in the program text is insignificant. The compiler selects the “best match” (which is precisely defined in the C++ standard[4]).

### 4.5 Expression Templates

Expression templates are a C++ technique for passing expressions as function arguments. The expression can be inlined into the function body, which results in faster and more convenient code than C-style callback functions.

Passing an expression to a function is a common occurrence. In C expressions are usually passed using a pointer to a callback function containing the expression. For example, the standard C library routines qsort() and bsearch() accept an argument `int (*cmp)(void*, void*)` which points to a user-defined function to compare two elements. Another common example is passing mathematical expressions to functions. There are several problems with callback functions. One of them is that repeated calls generate a lot of overhead, especially if the expression which the function evaluates is simple. Another problem can occur with variables declared outside the callback function but are in its scope, by the time the callback function uses them they can be obsolete.

So how does it work? The trick is that the expression is parsed at compile time, and stored as nested template arguments of an "expression type".

The basic notion of expression templates is that parse trees of expressions can be represented as C++ types by using recursive templates.
To build parse trees of expressions using recursive templates, operators such as +, -, *, and / can be overloaded so that, rather than performing any useful computation, they simply return a type representing a parse tree of the operation. No calculation is performed until the expression template (representing a parse tree of the expression) is assigned to a vector or matrix. At this point, the expression can be evaluated in a single pass using no temporaries.

Following is a C++ example demonstrating an implementation of a compile-time version of a vector dot product [6]. The dot product of two vectors is the sum of the products of its corresponding elements. Example: The dot product of the two 3-dimensional vectors (1,2,3) and (4,5,6) would be 1*4 + 2*5 + 3*6, which is 32. The goal is to set up expression templates for the calculation of the dot product of vectors of arbitrary dimensions, like in the following example:

```cpp
int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
cout << dot<4>(a,b);
```

The implementation and the usage of the implementation are shown here:

```
A Compile-Time Implementation of the Dot Product
```

```
template <size_t N, class T>
class DotProduct {  
public:  
    static T eval(T* a, T* b)  
    {  
        return DotProduct<1,T>::eval(a,b)  
            + DotProduct<N-,T>::eval(a+1,b+1);  
    } 
};

template <class T>
class DotProduct<1,T> {  
public:  
    static T eval(T* a, T* b)  
    {  
        return (*a)*(*b);  
    } 
};
```

```
Using the Dot Product Implementation
```

```
template <size_t N, class T>
inline T dot(T* a, T* b)  
{  
    return DotProduct<N,T>::eval(a,b);  
}
```

```
int a[4] = {1,100,0,-1};
int b[4] = {2,2,2,2};
cout << dot<4>(a,b);
```

dot<4>(a,b) on the other hand evaluates to DotProduct<4,size_t>::eval(a,b) which recursively triggers instantiation of further class templates and gradually unfolds as follows:

DotProduct<4,size_t>::eval(a,b) evaluates to
DotProduct<1,size_t>::eval(a,b) + DotProduct<3,size_t>::eval(a+1,b+1)
which evaluates to
(*a)*(*b) + DotProduct<1,size_t>::eval(a+1,b+1) +
DotProduct<2,size_t>::eval(a+2,b+2)
which evaluates to
(*a)*(*b) + (*a+1)*(*b+1) + DotProduct<2,size_t>::eval(a+2,b+2) +
DotProduct<1,size_t>::eval(a+3,b+3)
which evaluates to
(*a)*(*b) + (*a+1)*(*b+1) + (*a+2)*(*b+2) + (*a+3)*(*b+3)
Visible in the executable is only the resulting expression (*a)*(*b) +
(*a+1)*(*b+1) + (*a+2)*(*b+2) + (*a+3)*(*b+3) ; the recursive template instantiation itself will have been performed at compile-time already.
5 Examples of compile time optimization

As described in earlier sections the key idea of template metaprogramming is to optimize algorithms by specializing them with respect to a given compilation context usually supplied by the user as a template argument. We will define a class template that takes a non-type template argument, namely N, and returns the result in form of a nested value. The end of the recursion is specified by means of a template specialization that does not require any further template instantiations.

In this part we will define three languages using grammar notations and regular expressions. For each language we will demonstrate a metaprogram which implements an algorithm checking if a given word is valid for the language. The program parses the word according to the production rules of the grammar thus validating each letter or set of letters at a time. We will start with a very simple algorithm, validating words of a certain language and gradually advance to more complicated algorithms for other languages. These metaprograms rely on the length of the word, which is checked in the language, to be known at compile time and to be passed as a template argument.

5.1 First language

A regular expression of the language: $\sum = a^+$

The grammar defining the language:
1. $S \rightarrow A$
2. $A \rightarrow aA$
3. $A \rightarrow a$

For example some valid words are "a", "aa", "aaaa", "aaaaaaaaaa"
Invalid words are "ab", "c", "aaaaaaaaaaaaaaab"

This algorithm validating a word in the language is very simple thus making it a good example to illustrate how template metaprograms can be used to generate specialized algorithms.
Before plunging into the details of the metaprogram, here is how an algorithm to solve this problem will look like written in dynamic code:

```c
int isValid(char *data, int N)
{
    if (N<1) return 0;
    for (int valid=1, i=0; i<N; ++i)
    {
        if (data[i] != 'a')
            valid = 0;
    }
    return valid;
}
```

Clearly, the algorithm could have been optimized by using a `while` loop that ends as soon as the first occurrence of a letter which is not 'a'. In order to keep the code as simple as possible such an optimization was avoided.

The function takes as input two parameters:

1. "data" - a pointer to a char thus enabling the user of this function to send an array of chars.
2. "N" the length of the array

The output is the integer value, so if the word argument is a valid word of the language the value 1 is returned and if the word is not a valid word in the language the value 0 is returned. All examples in this section will have the same parameters and return values.

Calling this function is done in the following manner for the argument "aaa":

```c
int result = isValid("aaa",3);
```

Thus, the following function calls:

```c
cout << "isValid(aaa) = " << isValid("aaa",3) << endl;
cout << "isValid(aab) = " << isValid("aab",3) << endl;
```

will result with the following output:

```
isValid(aaa) = 1
isValid(aab) = 0
```

A specialized version of isValid for N=3 might look like this:

```c
int isValidNoLoop(char *data, int N=3)
{
    int valid=1, i=0;
    if (data[i++] != 'a') valid = 0;
    return valid;
}
```
if (data[i++] != 'a') valid = 0;
if (data[i++] != 'a') valid = 0;
return valid;
}

Thus, the following function calls:
```
cout << "isValidNoLoop (aaa) = " << isValidNoLoop("aaa",3) << endl;
cout << "isValidNoLoop (aab) = " << isValidNoLoop("aab",3) << endl;
```

will result with the following output:
```
isValidNoLoop (aaa) = 1
isValidNoLoop (aab) = 0
```

Here, again, a basic optimization that ends the function could have been used to return 0 immediately after the first occurrence of a non-'a' letter. It was avoided to comply with the previous example. The technique used here is to avoid loops by unrolling them. This optimization technique is usually done by the optimizer of the compiler in order to avoid the overhead of a loop.

In order to generate a function such as the above for any value of \( N \) it seems that we'll have to unroll the loop by using a recursive version:

```cpp
int isValidRecursive(char *data, int N)
{
    // end recursion condition for a word of length 0
    if (N == 0)
        return 0;

    // end recursion condition for a word of length 1
    if (N == 1)
        {
            if (*data != 'a')
                return 0;
            else
                return 1;
        }

    int valid;
    if (*data != 'a')
        valid = 0;
    else
        valid = isValidRecursive(data++, --N); // the recursive call
    return valid;
}
```

Thus, the following function calls:
cout << " isValidRecursive (aaa) = " << isValidRecursive ("aaa",3) << endl;
cout << " isValidRecursive (aab) = " << isValidRecursive ("aab",3) << endl;

will result with the following output:
isValidRecursive (aaa) = 1
isValidRecursive (aab) = 0

Now the algorithm consists of a recursive call to itself. This structure is simple to implement using a template class:

#elif IS_VALID2_H
#define IS_VALID2_H
// The first template implements the general recursive rule:
// production rule #2
template<int N> //N = number of chars left in array
class isValidClass
{
    public:
        static int isValidFunc(char *data)
        {
            return (data[0] != 'a') ? 0 : isValidClass<N-1>::isValidFunc(++data);
        }
};

// The second template is a specialization that ends the recursion. It establishes the
// result for a word suffix of length 1:
// production rule #3
template<> // specialization
class isValidClass<1>
{
    public:
        static int isValidFunc(char *data)
        {
            return (data[0] != 'a') ? 0 : 1;
        }
};

// The third template is a specialization that ends the recursion. It establishes the
// result for a word suffix of length 0:
// production rule #0
template<> // specialization
class isValidClass<0>
{
    public:
        static int isValidFunc(char *data)
        {
            return 0;
        }
};
#endif // IS_VALID2_H
How does this work? When `isValidClass<N>` is instantiated, the compiler needs `isValidClass<N-1>` in order to call the function `isValidFunc`. So it instantiates `isValidClass<N-1>`, which in turn requires `isValidClass<N-2>`, requiring `isValidClass<N-3>`, and so on until `isValidClass<1>` is reached where template specialization is used to end the recursion.

In other words to check if a word is valid we invoke `isValidClass<N>::isValidFunc(char* data)` which makes a recursive call to itself.

A base case is provided to end the recursive calls and is implemented with a template specialization for N=1. A template specialization for N=0 is also provided to prevent from the empty word to be considered as valid.

Let's study the details of what happens when we use this template to find the validity of the word "aaa". We invoke it like this:

`isValidClass<3>::isValidFunc("aaa");`

We can manually expand this to see the effect:

`isValidClass<3>::isValidFunc("aaa");`  
`isValidClass<2>::isValidFunc("aa");`  
`isValidClass<1>::isValidFunc("a");`

The static function `isValidFunc` uses the conditional expression operator (?:) to decide on a course of action. Using this operator gives the following effect of a manual expansion:

Step 1:  
Invoking the program  
`int returnValue = isValidClass<3>::isValidFunc("aaa");`

Step 2:  
The compiler instantiates `isValidClass<3>`. Its result is:  
`int returnValue = ('a' != 'a')`  
? 0  
: `isValidClass<2>::isValidFunc("aa");`

Step 3:  
The compiler instantiates `isValidClass<2>`. Its result is:  
`int returnValue = ('a' != 'a')`  
? 0  
: ('a' != 'a')

22
Step 4:
The compiler instantiates isValidClass<1>. Its result is:
```c++
int returnValue = ('a' != 'a')
    ? 0
    : ('a' != 'a')
    ? 0
    : ('a' != 'a')
    ? 0
    : 1;
```

The main difference between the two flavors (static and dynamic), demonstrated above, of this algorithm is their execution time. The dynamic form is a regular program, compiled and executed when the output program is invoked. The static form is executed at compile time: none of its code is compiled, and the calls to isValidFunc are instead replaced by the value they compute. Although the input word is also static the compiler relies in its static form on the 'N' which is the length of the word to be validated. The reason is that the semantics of template instantiations relies on the template parameter but not on a regular function parameter.
5.2 Second language

Let's look at a slightly more complicated example.

A regular expression of the language: $\Sigma = a^* b^*$

The grammar defining the language:

1. $S \rightarrow AB$
2. $A \rightarrow aA$
3. $A \rightarrow a$
4. $B \rightarrow bB$
5. $B \rightarrow b$

For example some valid words are "ab", "aabb", "aaaab", "aaaabbbbbbbb"

Invalid words are: "aba", "aaaabbc", "a", "b", "bbbbbaaaaa"

Following is the metaprogram which finds out if a given word is valid for the language described above. Like in the first language, the basic idea is to use recursive instantiations that end with the instantiation of specializations. This algorithm is more complex then the one of the previous example because it is not enough to look at current letter checked, it is also necessary to check the 'look ahead letter' which is the next letter of the word, following the current letter being checked. Based on the look ahead letter the algorithm decides on its course of action.

```cpp
#ifndef IS_VALID3_H
#define IS_VALID3_H

// production rule #1
template<int N> //N = number of chars left to check
class isValidClass3
{
    public:
        static int isValidFunc(char *data)
        {
            return (isValidClass3Rule3<N>::isValidFunc(data));
        }
};
```

```cpp
#endif
```
// production rule #3 and #2, check for a prefix of 'a's.  
// Relate to the look ahead letter to determine the next function call, e.g. at end of  
// 'a's activate rule 5  
template<int N> //N = number of chars left in array  
class isValidClass3Rule3  
{  
    public:  
        static int isValidFunc(char *data)  
        {  
            return (data[0] != 'a')  
                ? 0 // current char is not 'a' – word is invalid  
                : (data[1] == 'a')  
                    ? isValidClass3Rule3<N-1>::isValidFunc(++data) // next is 'a' continue rule 3  
                    : isValidClass3Rule5<N-1>::isValidFunc(++data) // next isn't 'a' check rule 5  
                ;  
        }  
};  

// production rule #5  
template<int N> //N = number of chars left in array  
class isValidClass3Rule5  
{  
    public:  
        static int isValidFunc(char *data)  
        {  
            return (data[0] != 'b')  
                ? 0  
                : isValidClass3Rule5<N-1>::isValidFunc(++data);  
        }  
};  

// production rule #4  
template<> // specialization  
class isValidClass3Rule5<1>  
{  
    public:  
        static int isValidFunc(char *data)  
        {  
            return (data[0] != 'b') ? 0 : 1;  
        }  
};  

// production rule #2 and #3  
template<> // specialization  
class isValidClass3Rule3<1>  
{  
    public:  
        static int isValidFunc(char *data)  
        {  
            return (data[0] != 'a') ? 0 : 1;  
        }  
};
// production rule #1
template<> // specialization
class isValidClass3<0>
{
    public:
        static int isValidFunc(char *data)
        {
            return 0;
        }
};
#endif // IS_VALID3_H

To check if a word is valid we invoke isValidClass<N>::isValidFunc(char* data).

The program has several classes, each has a static function which checks if the
next letter of the input array is valid according to the production rule that the
function represents.

isValidClass3 - checks rule 1
isValidClass3Rule3 - checks rules 3 and 2
isValidClass3Rule5 - checks rules 5 and 4

Each function has two tasks:

1. The first task is to check the first letter of the input word. If a certain letter
   violates a rule of the language then 0 is returned.

2. The second task after the prefix letter is validated is to decide according to
   the next letter (a.k.a look ahead letter) on a course of action. The case
   differentiation is done using the conditional expression operator (?:). If the
   next letter is the same as the current one then a self recursive call is done
   with a pointer to the next letter and a subtraction of 1 from the word length
   parameter. Otherwise, call the other function.

The template specialization 'class isValidClass3<0>}' for N=0 is intended for an
empty word (of length 0), which is not a part of the language. That's why its
function, isValidFunc, returns 0.

The template specialization 'class isValidClass3Rule5<1>'
for N=1 is provided as the base case to end the recursive calls. Since the compiler
turns to this specialization if all the word up to the last one is valid, its function
validates the last letter. Thus, returning 1 if it is a 'b' and 0 otherwise.

We can manually expand this program for N=4, data="aabb" to see the effect:
isValidClass3<4>::isValidFunc("aabb")
isValidClass3Rule3<4>::isValidFunc("aabb")
isValidClass3Rule3<3>::isValidFunc("abb")
isValidClass3Rule5<2>::isValidFunc("bb")
isValidClass3Rule5<1>::isValidFunc("b")

For the expression is isValidClass3<4>::isValidFunc("aabb") lets look at a
different way to track the manual expansion.:

Step 1: Invoking the program
   isValidClass3<4>::isValidFunc("aabb")

Step 2: The compiler instantiates isValidClass3<4>. Its result is:
   return isValidClass3Rule3<4>::isValidFunc("aabb")

Step 3: The compiler instantiates isValidClass3Rule3<4>::isValidFunc("aabb").
   Its result is:
   return (a != 'a') // first letter
     ? 0
     : ('a' == 'a') // second letter
       ? isValidClass3Rule3<3>::isValidFunc("abb")
       : isValidClass3Rule5<3>::isValidFunc("abb")

Step 4: The compiler instantiates isValidClass3Rule3<3>::isValidFunc("abb")
   and isValidClass3Rule5<3>::isValidFunc("abb")
   Its result is:
   return (a != 'a') // first letter
     ? 0
     : ('a' == 'a') // second letter
       ? (a != 'a') // second letter
       ? 0
       : ('a' == 'a') // third letter
         ? isValidClass3Rule3<2>::isValidFunc("bb")
         : isValidClass3Rule5<2>::isValidFunc("bb")
   :[instantiation of isValidClass3Rule5<3>::isValidFunc("abb")]

Step 5: The compiler instantiates isValidClass3Rule3<2>::isValidFunc("bb")
   and isValidClass3Rule5<2>::isValidFunc("bb")

27
Its result is:

```cpp
return
(a != 'a') // first letter
? 0
: ('a' == 'a') // second letter
  ? ('a' != 'a') // second letter
    ? 0
      : ('a' == 'a') // third letter
        ? [instantiation of isValidClass3Rule3<2>::isValidFunc("bb")]
          : ('b' != 'b') // third letter
            ? 0
              : isValidClass3Rule5<1>::isValidFunc("b");
: [instantiation of isValidClass3Rule5<3>::isValidFunc("abb")]
```

Step 6: The compiler instantiates isValidClass3Rule5<1>::isValidFunc("b")
Its result is:

```cpp
return
(a != 'a') // first letter
? 0
: ('a' == 'a') // second letter
  ? ('a' != 'a') // second letter
    ? 0
      : ('a' == 'a') // third letter
        ? [instantiation of isValidClass3Rule3<2>::isValidFunc("bb")]
          : ('b' != 'b') // third letter
            ? 0
              : (b' != 'b') // fourth letter
                ? 0
                  : 1
: [instantiation of isValidClass3Rule5<3>::isValidFunc("abb")]
```

The final stage ends the recursive process because
isValidClass3Rule5<1>::isValidFunc("b") matches the explicit specialization that catches the last letter of the word. The final result is therefore: 1.

What do the square brackets [ ] in this example mean? Semantically, I put them around class templates which I chose not to track their instantiation because they had no relevance for the final result. This is not so compiler-wise because the compiler does instantiate them and continues recursively instantiating all the templates they use.

This will be elaborated in the next example which is an example of a third language.
5.3 Third language

Let's examine our final and most complicated algorithm validating a word of a language.

A regular expression of the language: \[ \Sigma = (hij)^+ b^+ (yy)^+ b^+ | (hij)^+ b^+ \]

The grammar defining the language:
1: S \rightarrow M BYB | MB
2: M \rightarrow hij
3: M \rightarrow hijM
4: B \rightarrow b
5: B \rightarrow bB
6: Y \rightarrow yy
7: Y \rightarrow yyY

For example valid words are hijbyyb, hijhijbyyb, hijbbbyyb, hijbyyyyyb, hijbbyyyyyb, hijhijhijbbyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
template<int N> // N = number of chars left in array
class isValidClass4Rule3
{
  public:
    static int isValidFunc(char *data)
    {
      return (data[0] == 'h')
        ? (data[1] == 'i')
          ? (data[2] == 'j')
            ? (data[3] == 'h') // look ahead to choose next rule
              ? isValidClass4Rule3<N-3>::isValidFunc(++(++(++data)))
                : isValidClass4Rule5<N-3>::isValidFunc(++(++(++data)))
              : 0
            : 0
          : 0
        : 0;
    }
};

template<int N> // N = number of chars left in array
class isValidClass4Rule5
{
  public:
    static int isValidFunc(char *data)
    {
      return (data[0] == 'b')
        ? (data[1] == 'b') // look ahead to choose next rule
          ? isValidClass4Rule5<N-1>::isValidFunc(++data)
            : isValidClass4Rule7<N-1>::isValidFunc(++data)
          : 0;
    }
};

template<int N> // N = number of chars left in array
class isValidClass4Rule7
{
  public:
    static int isValidFunc(char *data)
    {
      return (data[0] == 'y')
        ? (data[1] == 'y')
          ? (data[2] == 'y') // look ahead to choose next rule
            ? isValidClass4Rule7<N-2>::isValidFunc(++(++data))
              : 0
          : 0
        : 0;
    }
};
isValidClass4Rule5<1>::isValidFunc(++(++data))
: 0
: 0
;
}

// from this point of the program all the rest is specializations
//---------------------
// specialization
//---------------------

// production rule #5
template<> // specialization
class isValidClass4Rule5<1>
{
public:
    static int isValidFunc(char *data)
    {
        return (data[0] != 'b') ? 0 : 1;
    }
};

// production rule #3
template<> // specialization
class isValidClass4Rule3<0>
{
public:
    static int isValidFunc(char *data)
    {
        return 0;
    }
};

// production rule #3
template<> // specialization
class isValidClass4Rule3<1>
{
public:
    static int isValidFunc(char *data)
    {
        return 0;
    }
};

// production rule #3
template<> // specialization
class isValidClass4Rule3<2>
{
public:
    static int isValidFunc(char *data)
    {
        return 0;
    }
}
// production rule #1
// specialization - case empty word
class isValidClass4<0>
{
    public:
        static int isValidFunc(char *data)
        {
            return 0;
        }
};

// production rule #1
// specialization - case 1 letter
class isValidClass4<1>
{
    public:
        static int isValidFunc(char *data)
        {
            return 0;
        }
};

// production rule #7
// specialization - case 0 letter
class isValidClass4Rule7<0>
{
    public:
        static int isValidFunc(char *data)
        {
            return 0;
        }
};

// production rule #7
// specialization - case 1 letter
class isValidClass4Rule7<1>
{
    public:
    static int isValidFunc(char *data)
    {
        return 0;
    }
};

// production rule #7
template<> // specialization - case 2 letter
class isValidClass4Rule7<2>
{
    public:
    static int isValidFunc(char *data)
    {
        return 0;
    }
};
#endif // IS_VALID4_H

To invoke the program we use isValidClass4<1>::isValidFunc("char* data") .

Like in the second example of the language \( \Sigma = a^+ b^+ \) I have described each class represents a production rule which function determines if the prefix of the word it got does not contradict the rule it represents. Also, the functions check according to the 'look ahead letter', how to progress the validation process by choosing the next function call. This is done using the operator (?:).

Here is a manual walk through of this program for N=7, data="hijbyyb" to see the effect.

isValidClass4Rule3<7>::isValidFunc("hijbyyb")
isValidClass4Rule5<4>::isValidFunc("byyb")
isValidClass4Rule7<3>::isValidFunc("yyb")
isValidClass4Rule5<1>::isValidFunc("b")

Please note that in the above example instantiation of templates that actually had an influence on the output were tracked, so only one side of the (?:) expression was followed. Yet this is not the case for the compiler, it does not relate to which input word it gets. This brings me to the next question.
There are 10 specializations, why are so many needed?

Well, one reason is the 'blindness' of the compiler for the input word. Failing to supply the specializations might lead to a compilation error. The compiler disregards the input word as far as the instantiations are concerned. It only relates to the template parameter, \( N \) which is the length of the word. Based on it the compiler decides on its course of action, therefore instantiates both sides of the expression (\(?:)\. Had the program failed to supply specializations for calls of both sides, the recursion would have continued on and on, failing to stop, until some limit of the compiler would have reached causing the compilation to fail.

This is the case because the functions which validate the production rules relate to the first one, two or three letters of the input word they get. This causes the next function called to begin its check one, two, or three letters after the previous function call so the length of the word gets subtracted by more then one. These 'jumps' in size may cause the value of \( N \) to never get the values 1 or 0 thus causing the problem of failing to meet a base class for ending the recursion.

Let's examine the program had it not had the following:

```
// production rule #7
template<> // specialization - case 1 letter
class isValidClass4Rule7<1>
{
    public:
        static int isValidFunc(char *data)
        {
            return 0;
        }
};
```

For the input "hijbyyb", as seen before, the above specialization does not influence the outcome of the program. Nevertheless, we will follow a certain instantiation rout of the compiler and see the effect:

Here is a manual walk through of this program for \( N=7 \), data="hijbyyb" to see the effect:

- `isValidClass4Rule3<7>::isValidFunc("hijbyyb")`
- `isValidClass4Rule5<4>::isValidFunc("byyb")`
- `isValidClass4Rule7<3>::isValidFunc("yyb")`
- `isValidClass4Rule5<1>::isValidFunc("b")`
Up to here we tracked the same instantiations as the previous example. Here is where we diverge.

Here is another class template that gets instantiated:

isValidClass4Rule7<1>::isValidFunc("b")

Since we removed from the program the specialization of rule 7 for N=1 the compiler continues its instantiations.

isValidClass4Rule7<-1>::isValidFunc(some unknown data)
isValidClass4Rule7<-3>::isValidFunc(some unknown data)
isValidClass4Rule7<-5>::isValidFunc(some unknown data)
and so on.

Running this program with some specializations missing on a PC using microsofts Visual C++ Version 6.0, resulted in a "compiler limit" error when trying to instantiate a template with N= -1107. Here is the what it looked like:

c:\program files\microsoft visual studio\myprojects\metaprogram\isvalidclass4.h(34) : fatal error C1076: compiler limit : internal heap limit reached; use /Zm to specify a higher limit
c:\program files\microsoft visual studio\myprojects\metaprogram\isvalidclass4.h(42) : see reference to class template instantiation 'isValidClass4Rule3<-1107>' being compiled

As we've seen, immense amount of code is instantiated needlessly.
There is a way to prevent from both sides of the operator (?:) to be instantiated. Its basic idea is to create a template class that does an emulation of the 'if else' statement. It looks like this:

Template <bool C>
Class _name {};

class _name<true>
{
  public:
    static void f()
    { statement1; }
};
class _name<false>
{
  public:
    static void f()
The template metaprogram version generates either statement1 or statement2, depending on whether the condition is true. Since the condition is used as a template parameter, it must be known at compile time.

6 Drawbacks

The use of C++ as a two-level language has proven an invaluable tool, for program specialization and performance issues, such as in the Blitz++ project mentioned earlier. However, static programming in this language is possible only at the cost of several drawbacks.

6.1 Technical issues

- With many compilers, the depth of the template instantiation can be specified to a certain degree, but still is a limit for some computations.
- Even when a compiler can handle the programs it is given, the extensive use of template programming introduces a large increase of compile time.

6.2 Maintenance of static programs

- In practice, the use of static programming in C++ is made difficult by the syntax and the restrictions of the compile time subset of the language.
- Writing programs at the static level is difficult in itself. Because the static sublanguage has a very reduced syntax, and is very different from regular C++, programs become hard to understand.
- Static programs are difficult to debug due to lack of debugging support or IO during template instantiation.
- Long and incomprehensible compilation error reporting.
7 A Comparison between Template Haskell and the C++ metaprogramming techniques regarding DSLs

Domain specific languages (DSL) are languages which are meant to supply a solution to a specific problem domain. One approach for implementing a DSL is embedding it in another language. This means that the new language is both syntactically and semantically a subset of an existing host language.

This section compares the metaprogramming support offered by Template Haskell and C++ regarding the techniques they make available the implementer of an embedded domain specific language.

7.1 Terminology

Section 8 has an emphasis on concepts of programming languages therefore this section begins with presenting definitions for some of the terms used later on.

- Type inference – A strategy in which the type analyzer deduces the type of each variable on the basis of its use in the program.
- Splicing – a process in which the language processing system evaluates a compile time term and inserts the result into a runtime expression.
- Quotations – A template Haskell notation which means that the code is to be computed only in the next stage.
- Reification - a process in which the language processing system takes a runtime term, and makes its abstract syntax tree available to a compile time term.
- Monad – In Haskell, computations are implemented using an abstract type constructor called a monad [2].

7.2 Template Haskell DSL

Haskell is a pure functional language. A variety of experimental extensions to Haskell have been implemented, many of which require additional syntax, semantics, or compile-time program transformations [3].

Template Haskell is such an extension to Haskell, whose purpose is to enable any programmer to define new language features by writing algorithms that generate new Haskell code to be executed. This avoids the need to modify the compilers.
each time a new feature is added to the language, and it directly supports the use of Haskell for defining domain specific languages.

Type safety in Template Haskell is ensured by staged typing [3]. All metacode is type checked before it is executed; any code which it generates is then type checked before it is spliced into the final program. The final object code is guaranteed not to fail due to type errors. Template Haskell is quite flexible: for example, a metaprogram can read data from a disk file and use that while generating code.

The main components of Template Haskell are an algebraic data type for representing Haskell programs; a monad for constructing code while maintaining an evolving state mechanism for gaining access to the concrete syntax tree for code and splicing concrete syntax back into the program. Template Haskell is a homogeneous two-level language [3], which means that terms in a Template Haskell program appear in one of two levels: runtime terms define computations to be performed when the final program is executed, while compile time terms are executed during compilation, and are typically used to generate runtime terms. Template Haskell provides syntactic constructs for controlling which level a term belongs to. Splicing evaluates a compile time term and inserts the result into a runtime expression. Reification takes a runtime term, and makes its abstract syntax tree available to a compile time term. However, there is still a distinct compile time, during which all meta-computations occur and code is generated. The compiler produces object code, which contains no further metacomputation.

7.3 C++ DSL

C++ supports implementing two kinds of compile-time embedded DSLs: type-based DSLs and expression-based DSLs [3]. Both variants are implemented by means of template metaprogramming and allow for domain-specific code generation, optimization, and error checking. As discussed in an earlier section C++ resembles a two-level language (just like Template Haskell): a C++ program may contain both level-1 code, which is evaluated at compile time, and level-0 code, which is compiled and later executed at runtime. However, unlike Template
Haskell, C++ is a heterogeneous two-level language because the languages at each level are different: level-(−1) is functional and level-0 is object-oriented.

Template metaprogramming uses templates in three ways: Level-0 code uses templates as generic components. Level−1 code adds two rather unusual views of class templates: class templates as data constructors of an algebraic data type and class templates as functions. The C++ template instantiation mechanism provides the execution semantics for level−1 code.

7.4 Comparison
The following is a comparison on a number issues between the two languages:

- **Style of DSL implementation supported** - Template Haskell and C++ both support implementing compile-time embedded DSLs.

- **Metaprogramming model** - Template Haskell is a homogeneous two-level language with Haskell as its meta and object language. C++ is a heterogeneous two-level language: functional at the metalevel and imperative object-oriented at the object level. Metaprograms are encoded in the type system and use templates both as algebraic data constructors and functions. Homogeneity has the advantage of enabling reuse of code and programming skills across levels.

- **When is code generated?** - In Template Haskell and C++ all code generation is done at compile time.

- **How is code generated?** - In C++ code generation involves composing functions, classes, and templates. Functions are selected using a metaprogram and composed by function call or inlining (the latter provides a way to generate code in one piece). Templates are composed by template inlining. In C++, the smallest fragment to be used in the generated target has to be a valid function, class or template.

- **What can be generated?** - C++ can generate classes (and class hierarchies), functions, expressions (using expression templates and templates (as members)). Functions can be generated using function templates. Template Haskell can generate expressions and declarations of types, functions, values, classes, and instances. Only expressions and declarations can be spliced in. No new (user-defined) identifiers can be generated in C++, but this is possible in Template Haskell.
Type checking of the generator code - In the context of program generation, we have three opportunities for performing static type checking:

1. Type checking the code executed at generation time before generation
2. Type checking the quoted code (or functions and templates representing the code to be composed in C++) before generation
3. Type checking of the generated code before running it Template Haskell performs (1) and (2), compiles and runs the metacode, splices the results, and performs (3). This is called staged type inference. C++ will only do (3). The metalevel in C++ has just two kinds of values: types and integers, and it is dynamically typed, i.e., typing problems are discovered only when the metacode is run. With regard to (2), functions in C++ get type checked before generation, but templates won’t. Template Haskell strikes a balance between type-safety and the expressiveness necessary for a compile-time only meta-system.

- Separate compilation - Haskell supports separate compilation of generator code. C++ does not. In Template Haskell, the generator will not fail due to a typing error at the site of use, but it may generate ill-typed code. In C++, both kinds of typing error will surface at the site of generator use.
- Syntax for invoking the metalevel - Template Haskell uses splice as an explicit notation to invoke computations at the metalevel. In C++, the metalevel is invoked by instantiating a template.
- How are transformations performed? - Template Haskell transforms parse trees represented by algebraic data-structures and it has standard data-structures for representing Haskell programs. In C++, transformations are performed by manipulating algebraic data-structures representing the structure of an expression in expression DSLs or parameters in type-based DSLs.
- What is the scope of transformations? - In C++, the scope of transformations in the expression-based DSLs is limited to a single C++ expression, i.e., no global optimizations are possible. The scope of a transformation in Template Haskell is limited to a splice. If desired, a whole program could be quoted and passed to an optimization function in a splice.
- Reification - Compile-time reification allows discovering properties of code written by the user (including quoted code). Template Haskell has extensive reification capabilities. First, quoted code can be inspected as a parse tree. Furthermore, there are built-in operators to access the parse tree of a user-
defined declaration or the current source position (useful for error reporting). It is possible, in effect, to gain access to an entire module by enclosing its entire contents inside quotation brackets in importing the resulting code into another module. Compile-time reification in C++ is quite limited. A few properties of types can be discovered automatically, e.g., whether a type is a pointer type or if type A is derived from B, but the vast majority (e.g., names of variables, functions and classes; number of parameters of a function; member functions of a class; etc.), if needed, must be provided manually. Expression templates can also be viewed as an elegant reification technique: normally, we do not have access to the structure of an expression at compile-time. But expression templates reify the structure of an expression as a parse tree at compile time. This reification is programmed manually for different type and function combinations, but it can be included in a library and the user can benefit automatically and does not need to write any reification code. Unfortunately, expression templates only reify the algebraic structure of an expression, e.g., we cannot query the name of a variable (unless we encode it in the type), which could be useful to optimization. This allows the metaprogram to discover everything defined within the module.

- Analysis and error reporting - Analysis and error reporting need special attention in compile-time embedded DSLs. Both Template Haskell and C++ allow writing code that will check the structure a DSL program for some domain specific properties, e.g., whether different units of some measurements system are properly combined and used. The latter usually requires embedding some domain-specific type inference and can be done using static metaprogramming. C++ does not allow any I/O at the metalevel. Template Haskell supports I/O at the metalevel. Besides error reporting, I/O also has other applications such as inspecting the code being generated for debugging purposes (which is not possible in C++), reading additional specification from some file, or generating code in other languages.
8 Conclusion

A C++ program may contain both static code, which is evaluated at compile time, and dynamic code, which is compiled and later executed at runtime. Static computations are encoded in the C++ type system, and writing static programs is usually referred to as template metaprogramming.

The introduction of templates to C++ added a facility whereby the compiler can act as an interpreter. This makes it possible to write programs in a subset of C++ which are interpreted at compile time.

In this hybrid approach, source code contains two programs: the normal C++ runtime program, and a template meta-program which runs at compile time. The interpretive nature of the compiler is exploited to generate specialized algorithms. In practice, algorithms are decomposed into a set of code-generation rules encoded as C++ templates; these templates are the template metaprogram used for generating the algorithm. When an algorithm is needed, the C++ compiler uses the metaprogram to generate the necessary code.

Template meta-programs can generate useful code when interpreted by the compiler, such as a massively inlined algorithm - that is, an implementation of an algorithm which works for a specific input size, and has its loops unrolled. This results in large speed increases for many applications.
9 Bibliography


scheme