C++ Metastring Library and its Applications*

Zalán Szűgyi, Ábel Sinkovics, Norbert Pataki, and Zoltán Porkoláb

Department of Programming Languages and Compilers, Eötvös Loránd University
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
{lupin, abel, patakino, gsd}@elte.hu

Abstract. C++ template metaprogramming is an emerging direction of generative programming: with proper template definitions we can enforce the C++ compiler to execute algorithms at compilation time. Template metaprograms have become essential part of today's C++ programs of industrial size; they provide code adoptions, various optimizations, DSL embedding, etc. Besides the compilation time algorithms, template metaprogram data-structures are particularly important. From simple typelists to more advanced STL-like data types there are a variety of such constructs. Interesting enough, until recently string, as one of the most widely used data type of programming, has not been supported. Although, `boost::mpl::string` is an advance in this area, it still lacks the most fundamental string operations. In this paper, we analysed the possibilities of handling string objects at compilation time with a metastring library. We created a C++ template metaprogram library that provides the common string operations, like creating sub-strings, concatenation, replace, and similar. To provide real-life use-cases we implemented two applications on top of our Metastring library. One use case utilizes compilation time inspection of input in the domain of pattern matching algorithms, thus we are able to select the more optimal search method at compilation time. The other use-case implements `safePrint`, a type-safe version of `printf` – a widely investigated problem. We present both the performance improvements and extended functionality we have achieved in the applications of our Metastring library.

1 Introduction

Generative programming is an emerging programming paradigm. The C++ programming language [22] supports the generative programming paradigm with using the *template* facility. Templates are designed to shift the classes and algorithms to a higher abstraction level without losing efficiency. They enable data structures and algorithms to be parametrised by types, thus, the classes and algorithms can be more general and flexible. It makes the source code shorter and easier to read and maintain which improves the quality of the code. Templates and template-based libraries – most notably the Standard Template Library (STL) – is now an unavoidable part of professional C++ programs.

* Supported by TÁMOP-4.2.1/B-09/1/KMR-2010-0003

C++ templates – as opposed to the Java and C# solution – work using the *instantiation* mechanism. Instantiation happens when a template is referred to with some concrete arguments. During instantiation the template parameters are substituted with the concrete arguments and the generated code is compiled.

The instantiation mechanism has an – originally unintentional – side effect. By defining clever template constructs we can enforce the C++ compiler to execute algorithms at compilation time. To demonstrate this in 1994 Erwin Unruh wrote a program which printed a list of prime numbers as part of error messages [23]. Unruh used template definitions and template instantiation rules to compute the primes at compilation time. This programming style is called C++ template metaprogramming [26]. The template metaprogram itself "runs" at compilation time. The output of this process is the generated C++ code – in most cases not the pure source code, but its internal representation – which is also checked by the compiler. The generated program can run as an ordinary "run-time" program. Template metaprogramming has been proven to be a Turing-complete sub-language of C++ [5].

Template metaprogramming is widely used today for several purposes, like executing algorithms at compilation time to optimize or make safer run-time algorithms and data structures. *Expression templates* were the first applications [25] allowing C++ expressions to be evaluated lazily and eliminating the overhead of object-oriented programming mainly in numerical computations.

Static interface checking increases the safety of the code, allowing checking at compilation time whether template parameters meet the given requirements [19]. As the C++ programming language has no language support to describe explicit requirements for certain template properties, only the template metaprogram based library solutions [29] remain.

The classical compilation model of software development designs and implements sub-programs, then compiles them and runs the program. During the first step the programmer makes decisions about implementation details: choosing algorithms, setting parameters and constants. Using template metaprograms some of these decisions can be delayed. *Active libraries* [24, 13] take effect at compilation time, making decisions based on programming contexts. In contrast to traditional libraries they are not passive collections of routines or objects, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves.

Domain specific languages (DSLs) are dedicated to special problems. They are often incorporated into some general purpose host language – many times into C++. The *Ararat* system [9], *boost::xpressive* and *boost::proto* [38, 32] libraries are good examples to libraries for embedding DSLs.

In the last fifteen years lots of research activities focused on improving the process of metaprogramming. Essential compilation time algorithms have been identified and used to develop basic metaprogram libraries [1, 2]. Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as lists or tree structures. The canonical

examples for sequential data structures are `typelist` [1] and the elements of the `boost::mpl` library [30].

Strings are one of the most commonly used data types in programming. Some programming languages provide strings as built-in data types, while others support strings and their operations by their standard library. A number of applications are based on string manipulation, like lexical analysers, pattern matchers and serialization tools. These applications are widely used in most areas of computer science. Numerous research activities and studies managed to improve the efficiency of these algorithms, however these improvements focused only on run time algorithm optimizations.

Sometimes, part of the input arguments of string manipulation algorithms are known at compilation time. In these cases a template metaprogram is able to customize the string algorithm to the corresponding input, making it safer and more efficient. While using the *Knuth-Morris-Pratt* sub-string search algorithm [4] we know the exact pattern we are searching in the text. Thus, we can generate the *next* vector of the algorithm at compilation time. As an other example the *regular expression* library `boost:xpressive` is able to check the syntax of the matching pattern at compilation time to detect erroneous regular expressions.

As more and more complex applications of template metaprogramming have appeared, it is surprising that for a long time strings were not supported for compilation time programming. The first attempt, `boost::mpl::string` [34] has been created recently, and still lacks a number of essential features like compare, search and replace sub-strings. Therefore we extended `boost::mpl::string` to create our own metastring library to provide a better support for string manipulation in metaprograms. In this paper, we present the meta-algorithms of usual string operations.

To illustrate the importance of the Metastring library, we demonstrate its usage by detailed use cases. One of the examples is from the searching algorithm domain. Knowing the pattern to search at compilation time, we are able to choose various optimizations to improve our algorithm.

The other example is the implementation of the `printf` C function in a type-safe way. Although the `printf` function of the C standard library has a compact and practical syntax – that is why `printf` is so widely used even today – it is not recommended to use it in C++, because it parses the formatting string at run time, thus, it is not type-safe. At the same time, in most cases the formatter string is already known at compilation time. Hence, it is possible to generate a formatter string specific `printf` using metaprograms, so that the compiler is able to check the type of the parameters, making the code safer. We show that our type-safe `printf` performs better than the type-safe stream operations from the standard C++ library.

The paper is organized as follows. In Section 2 we give a short description of the template metaprogramming techniques. Section 3 introduces our Metastring library. In Section 4 we discuss our pattern matching improvements as applications of the metastring construct. In Section 5 we present the type-safe `printf`.

We give an overview on related and future works in Section 6, and we summarize our results in Section 7.

2 Template Metaprograms

The template facility of C++ allows writing algorithms and data structures parametrized by types. This abstraction is useful for designing general algorithms like finding an element in a list. The operations of lists of integers, characters or even user defined classes are essentially the same. The only difference between them is the stored type. With templates we can parametrize these list operations by type, thus, we need to write the abstract algorithm only once. The compiler will generate the integer, double, character or user defined class version of the list from it. See the example below:

```
template<typename T>
struct list
{
  void insert(const T& t);
  // ...
};

int main()
{
  list<int> l;     //instantiation for int
  list<double> d;  //instantiation for double
  l.insert(42); d.insert(3.14);  // usage
}
```

The list type has one template argument `T`. This refers to the parameter type, whose objects will be contained in the list. To use this list we need to generate an instance and assign a specific type to it. That method is called *instantiation*. During this process the compiler replaces the abstract type `T` with a specific type and compiles this newly generated code. The instantiation can be invoked either explicitly by the programmer but in most cases it is done implicitly by the compiler when the new list is first referred to.

The template mechanism of C++ enables the definition of partial and full *specializations*. Let us suppose that we would like to create a more space efficient type-specific implementation of the `list` template for `bool` type. We may define the following specialization:

```
template<>
struct list<bool>
{
  //type-specific implementation
};
```

Nevertheless, the implementation of the specialized version can be totally different from the original one. Only the names of these template types are the same. If during the instantiation the concrete type argument is `bool`, the specific version of `list<bool>` is chosen, otherwise the general one is selected.

Template specialization is essential practice for template metaprogramming too. In template metaprograms templates usually refer to other templates, sometimes from the same class with different type argument. In this situation an implicit instantiation will be performed. Such chains of recursive instantiations can be terminated by a template specialization. See the following example of calculating the factorial value of 5:

```
template<int N>
struct factorial
{
  enum { value = N * factorial<N-1>::value };
};

template<>
struct factorial<0>
{
  enum { value = 1 };
};

int main()
{
  int result = factorial<5>::value;
}
```

To initialize the variable `result` here, the expression `factorial<5>::value` has to be evaluated. As the template argument is not zero, the compiler instantiates the general version of the `factorial` template with 5. The definition of `value` is `N * factorial<N-1>::value`, hence the compiler has to instantiate the `factorial` again with 4. This chain continues until the concrete value becomes 0. Then, the compiler choses the special version of `factorial` where the `value` is 1. Thus, the instantiation chain is stopped and the factorial of 5 is calculated and used as initial value of the `result` variable in `main`. This metaprogram "runs" while the compiler compiles the code.

Template metaprograms therefore stand for the collection of templates, their instantiations and specializations, and perform operations at compilation time. The basic control structures like iteration and condition appear in them in a functional way [20]. As we can see in the previous example, iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structure and its specialization.

```
template<bool cond_, typename then_, typename else_>
struct if_
{
  typedef then_ type;
};

template<typename then_, typename else_>
struct if_<false, then_, else_>
{
  typedef else_ type;
};
```

The `if_` structure has three template arguments: a boolean and two abstract types. If the `cond_` is false, then the partly-specialized version of `if_` will be instantiated, thus the `type` will be bound by the `else_`. Otherwise the general version of `if_` will be instantiated and `type` will be bound by `then_`.

Complex data structures are also available for metaprograms. Recursive templates store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite forms of implementation of expression templates [25]. The canonical examples for sequential data structures are `typelist` [1] and the elements of the `boost::mpl` library [30].

We define a typelist with the following recursive template:

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

template <typename H, typename T>
struct Typelist
{
  typedef H head;
  typedef T tail;
};
typedef Typelist< char, Typelist<signed char,
             Typelist<unsigned char, NullType> > > Charlist;
```

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

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

Essential helper functions – like `Length`, which computes the size of a list at compilation time – have been defined in Alexandrescu's Loki library[1] in pure functional programming style. Similar data structures and algorithms can be found in the `boost::mpl` metaprogramming library [30].

3 Metastring Library

In this chapter we introduce our *metastring* library. In the examples we write the type- and function names of boost without the scope (`string`, instead of `boost::mpl::string`) to save space. If we write the names of functions or objects in STL we put the scope before them.

Metastring library is based on `boost::mpl::string` [34]. The Boost Metaprogram Library provides us a variety of meta containers, meta algorithms and meta iterators. The design of that library is based on STL, the standard library of C++. However, while the STL acts at run time, the `boost::mpl` works at compilation time. The meta version of regular containers in STL, like list, vector, deque, set and map are provided by `boost::mpl`. Also, there are meta versions of most algorithms and iterators. The string metatype was added to boost in the release 1.40. Contrary to other meta containers, the metastring has limited features. Almost all regular string operations, like concatenation, equality comparison, substring selection, etc. are missing. Only the `c_str` meta function, which converts the metastring type to constant character array, is provided by boost.

In our Metastring library we extended the `boost::mpl::string` with the most common string operations. The `boost::mpl::string` is a *variadic template* type [10] like the `boost::mpl::vector` [36], but only accepts characters as template arguments. The instantiated metastring type can perform as a concrete string at compilation time. Since an instantiated metastring is a type, one can assign a shorter name to it by the `typedef` keyword.

Certain programming languages define the string datatype as a sequence of characters; i.e. array or list of characters. The metastring itself is a template type. The template arguments contain the value of the string. Because C++ does not support passing string literals to template arguments [18], we need to pass string arguments character by character.

```
typedef string<'H','e','l','l','o'> str;
```

Setting metastrings char-by-char is very inconvenient, therefore, the boost library offers an improvement. In most architectures, the `int` contains at least four bytes and since the size of a character is one byte, it can store four characters. As a template argument can be any integral type, hence it is possible to pass four characters as integer, and later on a metaprogram transforms it back to characters. This provides a more readable notation:

```
typedef string<'Hell','o'> str;
```

Nevertheless, this is still not the simplest way of setting a metastring. The next C++ standard will provide a better and standard solution: with the combination of *variadic templates* [10] and *user defined literals* [28] we can pass a string in the form of `"Hello"s` to variadic templates with character arguments.

Since the literature usually uses the parameter passing by four character convention, in the rest of the paper we will follow that.

In the Metastring library we provide the meta-algorithms of the most common string operations, like `concat`, `find`, `substr`, `equal`, etc. These algorithms are also template types, which accept metastring types as template arguments. The `concat` and `substr` defines a type called `type` which is the result of the operation. `equals` provides a static boolean constant called `value`, which is initialized as true if the two strings are equal, otherwise as false. `find` defines a static `std::size_t` constant called `value`, which is initialized as the first index of matching, if the pattern appears in the text and as std::string::npos otherwise. See the example about concatenation of strings below:

```
typedef string<'Hell','o'> str1;
typedef string<' Wor','ld!'> str2;

typedef concat<str1, str2>::type res;

std::cout << c_str<res>::value
```

The type defined by `concat<str1, str2>` is a new metastring type, which represents the concatenation of `str1` and `str2` metastrings.

In the next chapter we present applications which can take either efficiency or safety advantages of metastrings. The first example is pattern matching. If the text or a pattern is known at compilation time, we can improve the matching algorithms. (If both the text and the pattern are known, we can perform the whole pattern matching algorithm at compilation time.) The second application is a type-safe `printf`. If the formatter string is known at compilation time, we can generate a specialized kind of `printf` algorithm to it, which can perform type checking.

4 Pattern matching applications with metastrings

Most of the pattern matching algorithms start with an initialization step. This step depends only on the pattern. If the pattern is known at compilation time, we can shift this initialization subroutine from run time to compilation time. This means that while the compiler compiles the code it will wire the result of the initialization subroutine into the code. Thus, the algorithm does not need to run the initialization step, because it is already initialized. The more often the pattern matching algorithm is invoked, the more speed-up we achieve. The example below shows how to use these algorithms:

```
typedef string<'patt','ern'> pattern;
std::string text;
// reading data to text

std::size_t res1 = kmp<pattern>(text);
std::size_t res2 = bm<pattern>(text);
```

The kmp and bm function templates implement the Knuth-Morris-Pratt [15] and the Boyer-Moore [3] pattern matching algorithms. The return values are similar to the std::string's find memberfunction in STL and are either the first index of the match or std::string::npos. The implementation of these functions are the following:

```
template<typename pattern>
std::size_t kmp(const std::string& text)
{
  const char* p = c_str<pattern>::value;
  const char* next = c_str<initnext<pattern>::type>::value;
  //implementation of Knuth-Morris-Pratt
}

template<typename pattern>
std::size_t bm(const std::string& text)
{
  const char* pattern = c_str<pattern>::value;
  const char* skip = c_str<initskip<pattern>::type>::value;
  //implementation of Boyer-Moore
}
```

The pattern template argument must be a metastring type for both of the functions. The initnext and the initskip meta algorithms create the next and the skip vectors for the algorithms at compilation time. The rest of the algorithms are the same as the normal run time version.

We compared the full run time version of algorithms with our solution where the initialization is performed at compilation time. Fig. 1 shows the results related to Knuth-Morris-Pratt and fig. 2 to Boyer-Moore. We tested these algorithms with several inputs. The input was a common English text and the pattern contained a couple of words. The pattern did not appear in the text, thus the algorithms had to read all the input. We measured the running cost with one, two, five and ten kilobyte long inputs. In both charts, the first columns show the running cost of the original algorithms and the second ones show the performance of the algorithms optimized at compilation time. The X-axis shows the inputs and the Y-axis shows the instructions consumed during the algorithm. The larger improvement can be achieved applying pattern match repeatedly.

It is quite rare but still interesting case when the text is known in compile time. In this case a meta program can analyze the characteristic of the text and choses the best pattern matching algorithm. For example if the alphabet of the input text is large, the Boyer-Moore is more efficient, but if the alphabet is small, choosing the Knuth-Morris-Pratt algorithm is more beneficial. The figure 3 shows the differences of search speed, when a search is performed by only Knuth-Morris-Pratt (1st group) or Boyer-Moore(2nd group) algorithm or the optimized version (3rd group). InputA denotes an ordinary English text and the pattern is a single word. InputB denotes the text containing a few characters,

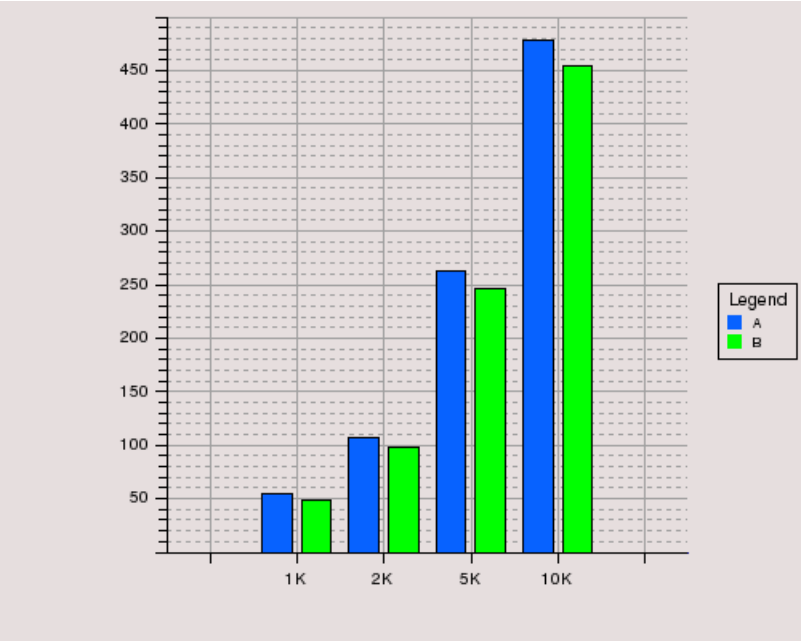

Fig. 1. Comparison of Knuth-Morris-Pratt

and the pattern has several repetitions. The Y-axis illustrates the consumed instructions.

5 Type-safe printf

The `printf` function of the standard C library is easy to use and efficient but has a major drawback: it is not type-safe. Due to the lack of type-safety, mistakes of the programmer may cause undefined behavior at runtime, because the compiler does not verify the validity of the arguments passed to `printf`. There are workarounds, for example gcc type checks `printf` calls and emits warnings when they are incorrect, but it is specific to gcc. C++ introduced *iostreams* as a replacement of `printf`. Iostreams are type-safe, but they have runtime and syntactical overhead. The syntax of `printf` is more compact than the syntax of streams, the structure of the displayed message is defined at one place, in the *format string*, when we use `printf` but it is scattered across the whole expression when we use streams. Here is an example for using `printf` and streams to display the same thing:

```
printf("Name: %s, Age: %d\n", name, age);
std::cout << "Name: " << name << ", Age: " << age << std::endl;
```

In this section we implement a type-safe version of `printf` using compilation time strings assuming that the format string is available at compilation time,

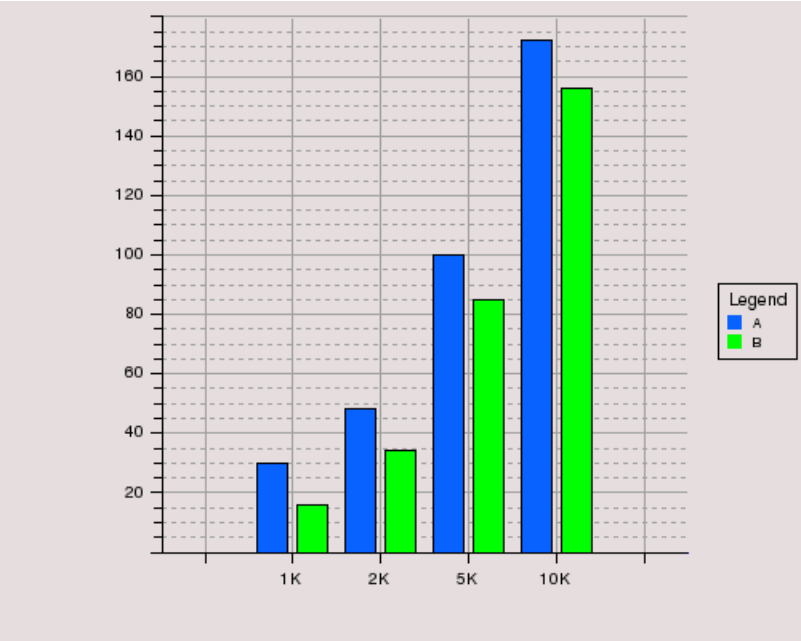

Fig. 2. Comparison of Boyer-Moore

which is true in most cases. We write a C++ wrapper for `printf` which validates the number and type of its arguments at compilation time and calls the original `printf` without any runtime overhead.

We call the type-safe replacement of `printf` `safePrintf`. It is a template function taking one class as a template argument: the format string as a compilation time string. The arguments of the function are the arguments passed to `printf`. As the example usage

```
safePrintf< string<'Hell', 'o %s', '!'> >("John");
```

shows there is only a slight difference between the usage of `printf` and our type-safe `safePrintf`. On the other hand, there is a significant difference between their safety: `safePrintf` guarantees that the `printf` function called at runtime has the right number of arguments and they have the right type.

Under the hood `safePrintf` evaluates a template-metafunction at compilation time which verifies the number and type of the arguments. `safePrintf` emits a compilation error [14] when at least one of the arguments is not correct. If the evaluation succeeds `safePrintf` calls `printf` with the same arguments `safePrintf` was called with. The template metafunction verifying the arguments has only compilation time overhead, it has zero runtime overhead, the body of `safePrintf` consists of a call to `printf` which is likely to be inlined. At the end of the day using `safePrintf` has zero runtime overhead compared to `printf`. Here is a sample implementation of our `safePrintf`:

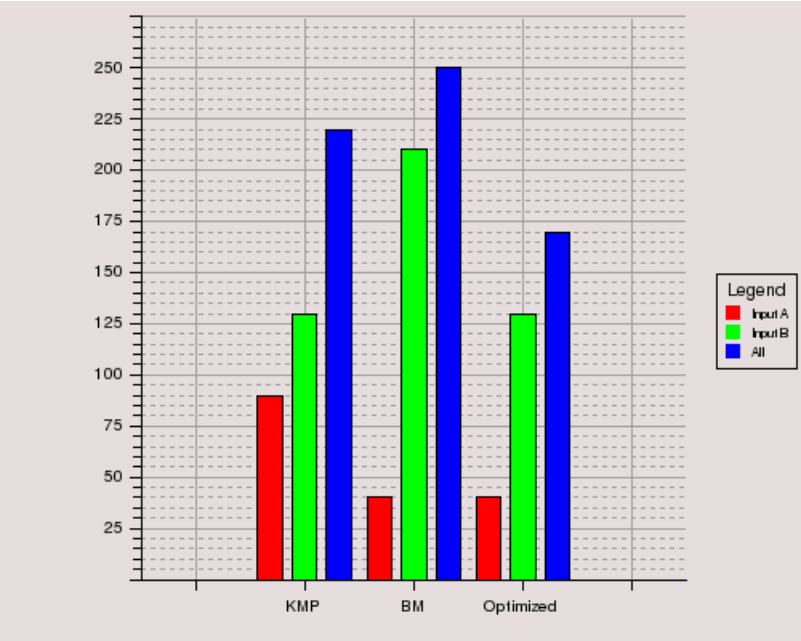

Fig. 3. Comparison of algorithms

```
template <typename FormatString, typename A1, typename A2>
int safePrintf(A1 a1, A2 a2)
{
  BOOST_STATIC_ASSERT((
    CheckArguments<
      FormatString,
      boost::mpl::list<A1, A2>
    >::type::value
  ));
  return
    printf(boost::mpl::c_str<FormatString>::type::value, a1, a2);
}
```

This example works only when exactly two arguments are passed to `safePrintf`. We will generalise it later. We evaluate a metafunction called `CheckArguments` which takes the format string, which is a compilation time string, and a type list containing the types of the arguments passed to `printf`. `CheckArguments` evaluates to a `bool` value: it is `true` when the argument types are valid and it is `false` when they are not. `CheckArguments` parses the format string character by character and verifies that the arguments conform to the format string. After verifying the validity of the arguments `safePrintf` generates code calling the original `printf` function of the C library. The format string passed to

`printf` is automatically generated from the compilation time string argument of `safePrintf`. For example

```
safePrintf< string<'Hell', 'o %s', '!'> >("John");
```

calls `printf` with the following arguments:

```
printf("Hello %s!", "John");
```

Under the hood `CheckArguments` uses a finite state machine [13] to parse the format string. The states of the machine are represented by template metafunctions, the state transitions are done by the C++ compiler during template metafunction evaluation. Template metafunctions are evaluated lazily, thus the C++ compiler instantiates only valid state transitions of the finite state machine. When an argument of `safePrintf` has the wrong type according to the format string, `CheckArguments` stops immediately, skipping further state transitions of the finite state machine. Thus the C++ compiler has a chance to emit the error immediately and continue compilation of the source code. We use a helper function, `CheckArgumentsNonemptyFormatString`, to implement `CheckArguments`:

```
template <typename FormatString, typename Ts>
struct CheckArgumentsNonemptyFormatString :
  boost::mpl::eval_if<
    typename boost::mpl::equal_to<
      typename boost::mpl::front<FormatString>::type,
      boost::mpl::char_<'%'>
    >::type,
    ParseSpecifier<
      typename boost::mpl::pop_front<FormatString>::type,
      Ts
    >,
    CheckArguments<
      typename boost::mpl::pop_front<FormatString>::type,
      Ts
    > > {};

template <typename FormatString, typename Ts>
struct CheckArguments :
  boost::mpl::eval_if<
    typename boost::mpl::empty<FormatString>::type,
    boost::mpl::empty<Ts>,
    CheckArgumentsNonemptyFormatString<FormatString, Ts>
  > {};
```

As one can see the template metafunction `CheckArguments` is just a wrapper for `CheckArgumentsNonemptyFormatString` to handle empty format strings, the

real parsing is done by `CheckArgumentsNonemptyFormatString`. The combination of these metafunctions represent one state of the finite state machine. Every character except `%` transitions back to this state, those characters are not important for us. The `%` character transitions to another state, represented by the `ParseSpecifier` metafunction:

```
template <typename FormatString, typename Ts>
struct ParseSpecifier : boost::mpl::and_<
    IsArgumentValid<
      typename boost::mpl::front<FormatString>::type::value,
      typename boost::mpl::front<Ts>::type>,
    CheckArguments<
      typename boost::mpl::pop_front<FormatString>::type,
      typename boost::mpl::pop_front<Ts>::type > > {};
```

This metafunction verifies the argument specified by the currently parsed placeholder in the format string using `IsArgumentValid`. When it is ok it continues the verification, otherwise it emits an error immediately. The implementation of `IsArgumentValid` is straightforward, it takes a character constant and a type as its arguments and evaluates to a `bool` value. It can be implemented in a declarative way:

```
template <char specifier, typename Ts>
struct IsArgumentValid : boost::mpl::false_ {};

template <>
struct IsArgumentValid<'c', char> : boost::mpl::true_ {};

template <>
struct IsArgumentValid<'d', int> : boost::mpl::true_ {};
// ...
```

Note that only the implementation of a simplified version of `safePrintf` was presented here to demonstrate how our solution works, the implementation of a verification function supporting the whole syntax of `printf` is too long to discuss here.

We have only shown the implementation of a `safePrintf` taking exactly 2 arguments. Other versions can be implemented in a similar way:

```
template <typename FormatString>
int safePrintf();

template <typename FormatString, typename A1>
int safePrintf(A1 a1);

template <typename FormatString, typename A1, typename A2>
int safePrintf(A1 a1, A2 a2);
```

```
template <typename FormatString,
          typename A1,
          typename A2,
          typename A3>
int safePrintf(A1 a1, A2 a2, A3 a3);
// ...
```

These functions can be automatically generated using the Boost precompiler library [31]. As it is the case with other Boost libraries, the number of `printf` functions generated can be specified by a macro evalutaing to an integer value. Thus, users of the library can increase it according to their needs. We do not present here how we generate these functions, it can be done using `BOOST_PP_REPEAT` provided by the Boost precompiler library.

This solution combines the simple usage and small run-time overhead of `printf` with the type-safety of C++ using compilation time strings. Stroustrup wrote a type-safe `printf` using variadic template functions [10, 28] which are part of the upcoming standard, C++0x [21]. His implementation uses runtime format strings and transforms `printf` calls to writing to C++ streams at runtime. For example the code

```
printf("Hello %s!", "John");
```

using his type-safe `printf` does

```
std::cout << 'H' << 'e' << 'l' << 'l' << 'o'
          << ' ' << "John" << '!';
```

at runtime. This solution prints the format string character by character which makes it extremely slow. The author's intention was to demonstrate the use of variadic templates, but it can be further optimized in the following way:

```
std::cout << "Hello " << "John" << "!";
```

We have measured the speed of normal `printf`, used by our implementation, and both of the above. We measured the speed of the following call:

```
printf("Test %d stuff\n", i);
```

and its `std::cout` equivalents. We printed the text 100 000 times and measured the speed using the `time` command on a Linux console. The average time it took can be seen in Table 1. `printf`, which is used by our type-safe implementation, is almost four times faster than the example on [28] and more than two times faster than the optimized version of that example.

We measured the performance of the C style `printf` function and the C++ style `std::cout` stream with several kinds of input from the simple ones to more compound samples. To do this measurement we used the profiler module of Valgrind [35] dynamic analysis tool called Callgrind. Table 2 shows the results.

Method used	Time
std::cout for each character	0,573 s
normal std::cout	0,321 s
printf	0,152 s

Table 1. Elapsed time

Pattern	printf	cout
”hello”	326	363
”hello%s”, ”world”	603	722
”hello%s%d”, ”world”, 1	942	1217
”hello%s%d%c”, ”world”, 1, ’a’	1149	1500
”hello%s%d%c\n”, ”world”, 1, ’a’	1395	2148

Table 2. Instructions fetched

In the first column we present the printed pattern. The second column shows the instructions needed to print it using `printf`, and the third one shows the same using `cout`.
As we can see from the table, `cout` is slower than `printf`. When the printed text is simple, the difference is slight, but it is growing as the text becomes more and more complex.

Another difference between Stroustrup's type-safe `printf` and ours is the way they validate the types of the arguments. Stroustrup's solution ignores the type specified in the format string, it displays every argument supporting the streaming operator regardless of its type. For example, it accepts the following incorrect usage of `printf`

```
printf("Incorrect: %d", "this argument should be an integer");
```

while our solution emits an error at compilation time. On the other hand, our solution can only deal with types the C `printf` can handle, while Stroustrup's solution can deal with any type which supports the streaming operator.

A drawback of Stroustrup's solution is that it does not detect when the arguments of `printf` are shifted or are in the wrong order and displays them incorrectly. For example Stroustrup's `printf` accepts

```
printf("Name: %s\nAge: %d\n", "27", "John");
```

and displays

```
Name: 27
Age: John
```

while our solution emits a compilation error.

Stroustrup's solution throws an exception at runtime when the number of arguments passed to `printf` is incorrect, which can lead to hidden bugs due to

incomplete testing. Our solution emits compilation errors in such cases to help detecting these bugs.

6 Related Work

Modern programming languages with object-oriented features and operator overloading are able to create classes with an interface close to high level matematical notations. For instance, arrays, matrices, linear algebraic operations are typical examples. However, as Veldhuizen noted, the code generated by such libraries tends to be inefficient [27]. As an example, he measured array objects using operator overloading in C++ were 3-20 times slower than the corresponding low level implementation. This is not because of poor design on the part of library developers, but because object-oriented languages force inefficient implementation techniques: dynamic memory allocations, high number of object copying, etc. These performance problems are commonly called as *abstraction penalty*.

Attempts to implement smarter optimizers were largely unsuccessful, mainly because of the lack of semantical information. Efforts to describe semantics of a type is still in experimental phase without too much result [11]. On the other hand, a more promising approach is to write an *active library*. Active libraries [24] act dynamically, makes decisions at compilation time based on the calling context, choose algorithms, and optimize code. These libraries are not passive collections of functions or objects, like traditional libraries, but take an active role in generating code. Active libraries provide higher abstractions and can optimize those abstractions themselves. In C++ active libraries are implemented with the help of template metaprogramming techniques [13].

An other possible optimization technique is *partial evaluation* [8, 12]. Partial evaluators regard a program's computation as containing two subsets: static computations which are performed at compile time, and dynamic computations performed at run time. A partial evaluator executes the static optimizations and produces a specialized *residual* program. To determine which portions of a program can be evaluated, a partial evaluator may perform binding time analysis to separate static and dynamic data and language constructs. Sometimes we call such a language as two-level language. C++ template resemble a two-level language, as function templates take both statically bound template parameters and dynamically bound function arguments [27].

There are third party libraries to apply string-related compilation time operations in some special areas of programming. The `boost::spirit` library is an object oriented recursive descent parser framework [33]. EBNF grammars can be written with C++ syntax and these grammars can be inlined in the C++ source code. Since the implementation of `spirit` uses template metaprogramming techniques, the parser of the EBNF grammar is generated by the C++ compiler. The `boost::wave` C++ preprocessor library [37] uses the `spirit` parser construction library to implement a C++ lexer with ISO/ANSI Standards conformant preprocessing capabilities. Wave provides an iterator interface which gives access to the currently preprocessed token of the input stream. These preprocessed tokens

are generated on-the-fly while iterating over the preprocessor iterator sequence. The `boost::xpressive` is a regular expression template library [38] dealing with static regular expressions. This library can perform syntax checking and generate optimized code for static regexes.

Stroustrup demonstrates how a type-safe `printf` can be built using the features of the upcoming C++ standard [21]. The differences between this and our solution are explained in chapter 5.

Since the style of metaprograms is unusual and difficult, it requires high programming skills to write. Maintenance of template metaprograms are much more harder. Besides, it is sorely difficult to find errors in template metaprograms. Porkoláb et al. provided a metaprogram debugger tool[17] in order to help finding bugs.

7 Summary and Future Work

Strings, one of the most commonly used data types in programming, had only weak support for C++ template metaprograms. In this paper we emphasize the importance of string manipulation at compile time. We have developed Metastring library based on `boost::mpl::string` and extended its compile time functionality with the usual operations of run time string libraries. We presented the implementational details of our Metastring library and discussed syntactic simplifications to reduce the syntactical overhead. To illustrate the importance of the metastring, we investigated two application areas in details.

When either the text or the pattern are known at compilation time, pattern matching algorithms can be significally improved. We dealt with two pattern matching algorithms: Boyer-Moore and Knuth-Morris-Pratt to demonstrate the power of metastrings. Our future work is to create a more sophisticated method – which takes more pattern matching algorithms into account – to find the best pattern matching solution.

As the other motivating application, we have created a C++ wrapper for `printf` function taking the format string as a compilation time string argument and validating the type of the runtime arguments based on that string. Validation happens at compilation time, therefore our solution has zero run time overhead but ensures type-safety. We have compared our type-safe `printf` solution to the one on Stroustrup's website and found that our solution provides stricter type-safety and runs at least two times faster. Our future plan is to introduce the `%a` specifier – which means *any* – to force the compiler to deduce the argument's type. Stroustrup's solution behaves similarly.

References

1. Alexandrescu, A: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley (2001)
2. Abrahams, D., Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Addison-Wesley (2004)

3. Boyer, R. S., Moore, J. S.: A Fast String Searching Algorithm. Communication of the ACM, vol. 20, pp. 762–772. (1977)
4. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms. The MIT Press (2001)
5. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading (2000)
6. Czarnecki, K., Eisenecker, U. W., Glück, R., Vandevoorde, D., Veldhuizen, T. L.: Generative Programming and Active Libraries. Springer-Verlag, (2000)
7. Devadithya, T., Chiu, K., Lu, W.: C++ Reflection for High Performance Problem Solving Environments, in Proceedings of the 2007 spring simulation multiconference - Volume 2, pp. 435–440.
8. Futamura, Y.: Partial Evaluation of Computation Process – An approach to a Compiler-Compiler. In Systems, Computers, Controls 2 (5): 4550. Reprinted in Higher-Order and Symbolic Computation 12 (4): pp. 381391, 1999, with a foreword. Kluwer Academic Publishers, 2000.
9. Gil, J. (Y.), Lenz, K.: Simple and safe SQL queries with c++ templates. In proc. of s (2007),Generative And Component Engineering 2007, The ACM Digital Library pp. 13–24, (2007)
10. Gregor, D., Järvi, J., Powell, G.: Variadic Templates (Revision 3). Number N2080=06-0150, ANSI/ISO C++ Standard Committee, October 2006.
11. Gregor, D., Järvi, J., Kulkarni, M., Lumsdaine, A., Musser, D., Schupp, S.: Generic programming and high-performance libraries. International Journal of Parallel Programing, Vol. 33, 2 (Jun. 2005), pp. 145-164. 2005.
12. Jones, N. D.: An introduction to partial evaluation. ACM Computing Surveys 28, 3 (Sept. 1996), pp. 480–503, 1996.
13. Juhász, Z., Sipos, Á., Porkoláb, Z.: Implementation of a Finite State Machine with Active Libraries in C++. In: Ralf Lammel, Joost Visser, Joao Saraiva (Eds.): Generative and Transformational Techniques in Software Engineering II, International Summer School, GTTSE 2007, Braga, Portugal, July 2-7, 2007. Lecture Notes in Computer Science 5235 Springer 2008, ISBN 978-3-540-88642-6., pp. 474-488.
14. Karlsson, B.: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, Reading (2005)
15. Knuth, D. E., Morris, J. H. Jr., Pratt, V. R.: Fast Pattern Matching in Strings. SIAM J. Comput. vol. 6, issue 2, pp. 323–350 (1977)
16. Meyers, S.: Effective STL. Addison-Wesley (2001)
17. Porkoláb, Z., Mihalicza, J., Sipos, Á.: Debugging C++ Template Metaprograms, in proc. of Generative Programming and Component Engineering (GPCE 2006), The ACM Digital Library pp. 255–264, (2006)
18. ANSI/ISO C++ Committee. Programming Languages – C++. ISO/IEC 14882:1998(E). American National Standards Institute, 1998.
19. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Metaprogramming (October 2000)
20. Sipos, Á., Porkoláb, Z., Pataki, N., Zsók, V.: Meta<Fun> - Towards a Functional-Style Interface for C++ Template Metaprograms, in Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007), pp. 489–502
21. Stroustrup, B.: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007
22. Stroustrup, B.: The C++ Programming Language Special Edition. Addison- Wesley, Reading (2000)

23. Unruh, E.: Prime number computation. ANSI X3J16-94-0075/ISO WG21-462.
24. Veldhuizen, T. L., Gannon, D.: Active libraries: Rethinking the roles of compilers and libraries. In: Proceedings of the SIAM Workshop on Object Oriented Methods for Interoperable Scientic and Engineering Computing (OO 1998), pp. 21–23. SIAM Press, Philadelphia (1998)
25. Veldhuizen, T. L.: Expression Templates. C++ Report 7(5), pp. 26–31. 1995.
26. Veldhuizen, T. L.: Using C++ Template Metaprograms. C++ Report 7(4), pp. 36–43. 1995.
27. Veldhuizen, T. L.: C++ Templates as Partial Evaluation. In Olivier Danvy (Ed.): Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, San Antonio, Texas, January 22-23, 1999. Technical report BRICS-NS-99-1, University of Aarhus. pp. 13–18. 1999.
28. http://www.research.att.com/~bs/C++0xFAQ.html
29. Boost Concept checking. http://www.boost.org/libs/concept_check/concept_check.htm
30. Boost Metaprogramming library. http://www.boost.org/libs/mpl/doc/index.html
31. The boost preprocessor metaprogramming library. http://www.boost.org/doc/libs/1_41_0/libs/preprocessor/doc/index.html
32. The boost proto library, http://www.boost.org/doc/libs/1_37_0/doc/html/proto.html
33. http://spirit.sourceforge.net
34. http://www.boost.org/doc/libs/1_42_0/libs/mpl/doc/refmanual/string.html
35. http://valgrind.org
36. http://www.boost.org/doc/libs/1_40_0/libs/mpl/doc/refmanual/vector-c.html
37. http://www.boost.org/doc/libs/1_40_0/libs/wave/index.html
38. The boost xpressive regular library. http://www.boost.org/doc/libs/1_40_0/doc/html/xpressive.htm