C++ Metastring Library

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 metaprograms are special programs interpreted by the compiler. While regular programs recline upon usual language constructs, these metaprograms are implemented in functional approach and are created with the generic construct of C++, called template.
Metaprograms are widely used for the following purposes: executing algorithms at compile time, optimizing runtime programs, implementing active libraries, and emitting compilation errors and warnings to enforce certain semantic checks.
Developing these metaprograms requires a high level of programming competence and great amount of time, because of the imperfection of language support. Nevertheless, there are metaprogram libraries that can assist C++ metaprogramming, such as Loki and Boost::MPL. Nonetheless, they only provide a weak implementation of metastrings, though these would highly simplify the programmers' work, just as regular strings do in regular programming.
In this research, we analyzed the possibilities of handling string objects at compile time with a metastring library in order to overcome this insufficiency. We created a prototype implementation of metastring based on boost::mpl::string that provides the common string operations (substring, concatenation, replace, etc.). In this paper, we present how to improve the performance or code quality of some commonly used algorithms by applying metastrings.

1 Introduction

The classical compilation model of software development designs and implements subprograms, than compiles them and runs the program. During the first step the programmer makes decisions about implementation details: choosing algorithms, setting parameters and constants. Programming by generative programming paradigm, some of these decisions can be delayed and applied only a compile time. Active libraries [18] take effect at compile time, making decisions based on programming contexts, thus, they can optimize performance. In contrast of 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.

The C++ programming language [17] supports generative programming paradigm with using template facility. The templates are designed to shift the classes and algorithms to a higher abstraction level without loosing efficiency. This way the classes and algorithms can be more general and flexible, making the source code shorter, and easier to read and maintain, which improves the quality of the code.

A C++ code with templates – designed in a special way – is able to utilize the type-system of the language and force the compiler to execute a desired algorithm. This programming style is called template metaprogramming [20]. In this way, the program itself runs during compilation time. The output of this process is also checked by the compiler and can run as an ordinary program. Template metaprogramming is proved to be a Turing-complete sub-language of C++ [4].

Template metaprogramming is widely used for several purposes, like executing algorithms at compile time to optimize or make safer run-time algorithms and data structures. Expression templates [19] allow C++ expressions to be lazy-evaluated. Template metaprograms can perform compile time optimizations on a Finite State Machine [8]. Static interface checking increases the safety of the code, allowing compile time checking of template parameters whether they meet the given requirements [14].

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

There are cases, when some of the input arguments of string manipulation algorithms are known already during compile time. In these cases a template metaprogram can customize the string algorithm to the corresponding input, making it safer and more efficient. As an example, while using Knuth-Morris-Pratt substring search algorithm [1] we know the exact pattern to find in the text. Thus, we can generate the *next* vector of the algorithm at compile time.

Although the `printf` function of the standard library of the C programming language – that is a widely used function to write strings to the standard output – has a nice and compact syntax, it is not recommended to use in C++, because it parses the formatting string at runtime, thus, it is not typesafe. At the same time, in most cases the formatter string is already known during compilation time. Hence, it is possible to generate a formatter string specific printf with metaprograms, so that the compiler would be able to check the type of the other parameters, making our code safer.

In this paper, we present a metastring library based on `boost::mpl::string` [9, 21] to provide a string type for metaprograms. We introduce the meta algorithms of usual string operations of C++ Standard Template Library [11]. Ad-

ditionally, we provide meta-algorithms to make pattern matching more efficient and present a typesafe printf.

The paper is organized as follows. In section 2 we give a short description of template metaprogramming paradigm. Section 3 presents our Metastring library, while in section 4 we present some applications using these metastrings. Related and future work comes in section 5, and the conclusion stands in section 6.

2 Template Metaprograms

The template facility of C++ allows us to write 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 containing 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, and the compiler will generate the integer, character or user defined class version of the list from it. See the example below:

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

int main() {
  list<int> l;  //instantiation
  l.insert(42);
}
```

The list type has one template argument `T`. This refers to the future type, whose object will be contained by the list. To use this list we need to 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 or 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 efficient type-specific implementation of the `list` template for `bool` type. We may define the following specialization:

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

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

In case of certain expressions a concrete instance of template is needed for the compiler to deduce it. Thus, an implicit instantiation will be performed. Let us see the following example of calculating factorial 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 as long as 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. This algorithm runs while the compiler compiles the code. Hence, this example code is equivalent to `int result = 120`.

These programs, which run at compile time are called template metaprograms [2]. Template metaprograms stand for the collection of templates, their instantiations and specializations, and perform operations at compile time. The basic control structures like iteration and condition appear in them in a functional way [15]. As we can see in the previous example iterations in metaprograms are applied by recursion. Besides, the condition is implemented by a template structures and its specialization. See below:

```
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 `else_` will be bound by the `type`. Otherwise the general version of `if_` will be instantiated and `then_` will be bound by `type`.

Since the style of metaprograms is unusual and difficult, it requires high programming skills to write. Besides, it is sorely difficult to find errors in it. Porkoláb et al. provided a metaprogram debugger tool [12] in order to help finding bugs.

3 Metastring Library

Our metastring library is based on boost::mpl::string [21]. 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 runtime, the Boost::Mpl works at compile 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 in STL. The string metatype was added to Boost in the last 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 metastring itself is a template type. The template arguments contain the value of the string. While C++ does not support passing string literals to template arguments, we need to pass it character by character. See below:

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

The boost::mpl::string is a variadic template type like boost::mpl::vector [22], but only accepts characters as template arguments. The instantiated metastring type can perform as a concrete string at compile time. Since an instantiated metastring is a type, one can assign a shorter name to it by `typedef` keyword.

In the examples of the paper 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.

Setting metastrings char-by-char is very inconvenient, therefore, the Boost offers an improvement. 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. See example below:

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

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. We are exploiting this feature now.

Nevertheless, this is still not the simplest way of setting a metastring, but the C++ standard [13] only supports this one. We provide a non-standard extension to enable setting metastrings in a regular way. This method requires a code transformation tool, which transforms the source to a standard C++ code before compilation. Since our solution does not follow the C++ standard, in the following examples we will use the four character per template argument style.

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 string 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.

4 Applications with Metastrings

In this chapter we present some applications which can take either efficiency or safety advantages of metastrings. The first examples are pattern matching applications. If the text or a pattern is known at compile time, we can improve the matching algorithms. (If both of the text and the pattern are known, we can perform the whole pattern matching algorithm at compile time.) The third application is a typesafe printf. If the formatter string is known at compile time, we can generate a specialized kind of printf algorithm to it, which can perform type checking.

4.1 Pattern Matching with Known Pattern

Most of the pattern matching algorithms start with an initialization step. This step depends only on the pattern. If the pattern is known at compile time,

we can shift this initialization subroutine from runtime to compile 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 the pattern matching algorithm is invoked, the more advantage we get.

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 [10] and the Boyer-Moore [3] pattern matching algorithms. The return values are similar to the `find` function 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 Knuth-Morris-Pratt
}
```

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 compile time. The rest of the algorithms are the same as the normal runtime version.

We compared the full runtime version of algorithms with our solution where the initialization is performed at compile 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 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 blue columns show the running cost of the original algorithms and the green ones show the performance of the

algorithms optimized at compile time. The X-axis shows the inputs and the Y-axis shows the instructions consumed during the algorithm.

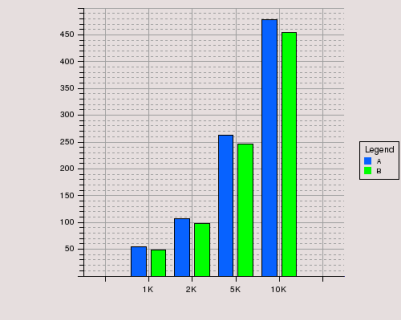

Fig. 1. Comparison of Knuth-Morris-Pratt

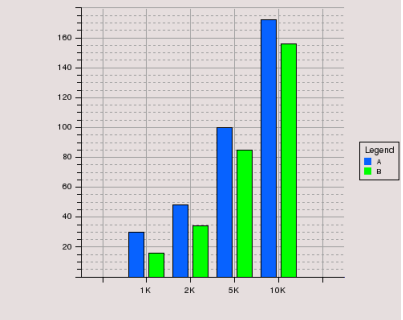

Fig. 2. Comparison of Boyer-Moore

4.2 Pattern Matching with Known Text

Several algorithms exist for fast pattern matching. Some of these are general algorithms, the worst-case time complexity of which are good in every kind of texts, like the Knuth-Morris-Pratt's algorithm. Others like the Boyer-Moore's algorithm [1] can be very fast in some special kinds of texts but very slow in other cases. If the text, where we want to find a pattern is known at compile time, our meta-algorithm can analyze it and choses the best fitting pattern matching algorithm. For example, if the alphabet of the input text is large, the Boyer-Moore algorithm is more efficient, but if the alphabet is small, choosing the Knuth-Morris-Pratt algorithm is more beneficial.

Since the text is known at compile time, the analysis can be done by the programmer. However, doing it manually with large texts is difficult, boring and it is easy to make mistakes, thus the result is often not the most efficient algorithm.

Another advantage of using our meta-algorithm is that it provides a unified interface to perform pattern matching, and the compile time optimizations are hidden from the user as well.

The example below shows the usage the of the `find` function:

```
typedef string<'Text,'we w','ant ','to s','earc','h fr','om'> text;

std::size_t result = find<text>("pattern");
```

Just like the `find` function in STL, our function returns with the index of the first match in text, or with `std::string::npos` if there is no match. The `find` function is a function template, where the template argument is a metastring containing the text. The function applies the text analyzer metaprogram and invokes the best fitting pattern matching algorithm. See below the definition of the `find` function.

```
template<typename text>
std::size_t find(const std::string& pattern) {
   return find(c_str<text>::value,
               pattern,
               analyze<text>::type() );
}
```

The template argument `text` must be a metastring type, while the pattern can be any string, not necessarily known at compile time. The `find` function calls a three-argument `find` function, which is hidden from the user. The inner `find` function is overloaded by the third argument `algorithm_tag` type. The implementation is similar to the implementation of the function `advance` in STL [11]. The template type `analyze` counts the different characters in `text` and defines `boyer_moore_algorithm` as `algorithm_tag` if the alphabet is large, otherwise defines `knuth_morris_pratt_algorithm`. See the sample code below:

```
struct boyer_moore_algorithm {};
struct knuth_morris_pratt_algorithm {};

std::size_t find(const std::string& text,
                 const std::string& pattern,
                 boyer_moore_algorithm)
{
  // Implementation of Boyer-Moore algorithm
}

std::size_t find(const std::string& text,
```

```
                const std::string& pattern,
                knuth_morris_pratt_algorithm)
{
  // Implementation of Knuth-Morris-Pratt algorithm
}

template<typename text>
struct analyze :
  eval_if<typename greater<typename different_chars<text>::type,
                           int_<N> >::type,
          boyer_moore_algorithm,
          knuth_morris_pratt_algorithm
        >
{
};
```

The `different_chars` is a meta-algorithm that counts the different characters in metastring `text` and if the number is more than N, then it is the `boyer_moore_algorithm` that is bound to the `type` by the meta `eval_if`, otherwise it is the `knuth_morris_pratt_algorithm`. In general English texts, the Boyer-Moore algorithm runs faster [3], thus we set N to 52, where 52 is the number of the small and big capital letters in the English alphabet.

We tested our solution against the pure runtime versions of Knuth-Morris-Pratt (KMP) and Boyer-Moore (BM) algorithms. In order to do so, we applied two kinds of inputs to these. The first, marked with `A` contained only a few different characters and there were several repetitions in the pattern. The second one, marked with `B` was an ordinary English text and the pattern consisted of a single word. Both texts contained about 2K characters. First we performed the search in both texts with the KMP, second with the BM and third with our optimized algorithm. Fig. 3 illustrates the result: the red rectangle shows the instructions required to perform the pattern matching in text `A`, while the green one shows the same in text `B`. The BM algorithm was very fast in case of input `B` but it was fairly slow in case of input `A`. In contrast, the KMP algorithm was much more balanced. Our solution was the optimal of all, because it was the only one being able to chose the better algorithm depending on the given text. The performance of these three algorithms are shown in the columns, while the consumed instructions are illustrated by the Y-axis.

4.3 Type-safe printf

Though the `printf` function of the standard C library is efficient and easy to use, it is not typesafe, hence mistakes of the programmer may cause undefined behavior at runtime. Some compilers – such as gcc – type check `printf` calls and emit warnings in case they are incorrect, but this method is not widely available. To overcome the problem, C++ introduced streams as a replacement of `printf`, which are typesafe, but they have runtime and syntactical overhead.

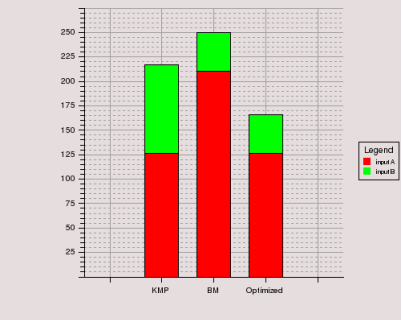

Fig. 3. Comparison of algorithms

In this section, we implement a typesafe version of `printf` using compile time strings. Contrary to the original `printf` function, this solution works in case the format string is available at compile time, which is true for most of the cases. We write a C++ wrapper for `printf` which validates the number and type of its arguments at compile time and calls the original `printf` without runtime overhead.

We call the typesafe replacement of `printf` `safePrintf`. It is a template function taking one class as a template argument: the format string as a compile time string. The arguments of the function are the arguments passed to `printf`. See below:

```
safePrintf<‘Hello %s!‘>("John");
```

As the example shows there is only a slight difference between the usage of `printf` and our typesafe `safePrintf`, while there is a significant difference between its safety: `safePrintf` guarantees that the `printf` function called at runtime will have the right number of arguments that will have the right type.

`safePrintf` evaluates a template-metafunction at compile time, which tries to verify the number and type of the arguments and in case this verification fails, `safePrintf` emits a compilation error [9]. On the other hand, if the verification succeeds `safePrintf` calls `printf` with the same arguments that `safePrintf` was called with. The template metafunction verifying the arguments cannot have a runtime overhead, only a compile time overhead. The body of `safePrintf` consists of a call to `printf`, which is likely to be inlined, thus, using `safePrintf` has no runtime overhead compared to `printf`.

Our `safePrintf` uses a metafunction called `ValidPrintf` to verify the correctness of the arguments. This metafunction takes two arguments: the format string as a compile time string and a compile time list of types. This metafunction parses the format string character by character, and verifies that the types conform the format string.

`ValidPrintf` uses a final state machine [8] to parse the format string. The states of this machine are represented by template metafunctions, and the state

transitions are done by the C++ compiler during template metafunction evaluation. Template metafunctions are lazy evaluated, thus the C++ compiler instantiates only valid state transitions of the final state machine. In case an argument of `safePrintf` has the wrong type according to the format string, `ValidPrintf` stops immediately, and it skips further state transitions of the final state machine. Thus the C++ compiler has a chance to emit the error immediately and continue compilation of the source code.

After verifying the validity of the arguments, `safePrintf` calls the original `printf` function of the C library. The format string passed to `printf` is automatically generated from the compile time string argument of `safePrintf`. For example:

```
safePrintf<‘Hello %s!‘>("John");
```

Here `safePrintf` calls `printf` with the following arguments:

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

This solution combines the simple usage and small run-time overhead of `printf` with the type-safety of C++ using compile time strings.

Stroustrup wrote a typesafe `printf` using variadic template functions [26], which are part of the upcoming standard C++0x [16]. His implementation uses runtime format string and transforms `printf` calls to write C++ streams at runtime. See the example:

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

Stroustrup's method does the following at runtime:

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

This solution was primarily written to demonstrate the power of variadic templates, that is why printing the format string is done character by character, making the process extremely slow. This method can be optimized in the following, more efficient way:

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

We have measured the speed of these operations and of the normal `printf` used by our implementation. We printed the following and it's `std::cout` equivalents:

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

The text was printed 100 000 times and the speed using the `time` command on a Linux console was measured. The average time of the process can be seen in Table 1. The `printf` function, which is used by our typesafe implementation, is almost four times faster than the example at [26] and more than two times faster than the optimized version of the example.

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

Another difference between Stroustrup's typesafe `printf` and ours is the way they validate the type 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`, where our solution would emit an error at compile time:

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

Our solution, however, can only deal with the types that the C `printf` can deal with, while Stroustrup's solution can deal with any type that supports the streaming operator.

Nevertheless, a drawback of Stroustrup's solution is that it does not detect the shifting or wrong order of `printf` arguments, and displays them incorrectly. For example Stroustrup's `printf` accepts and displays the following:

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

Name: 27
Age: John
```

On the other hand, our solution emits a compilation error in this case.

Stroustrup's solution throws an exception at runtime if the number of arguments passed to `printf` is incorrect. This can lead to hidden bugs due to incomplete testing, while our solution emits compilation errors in such cases.

5 Related Work

There are some third party libraries that also apply compile time operations related to strings in some special areas of programming.

`Spirit` [23] is an object oriented recursive descent parser framework. It enables to write `EBNF` grammars in C++ syntax and it can be inlined to the C++ source code. While the implementation of `spirit` uses template metaprogramming techniques, the parser from the `EBNF` grammar is generated by the C++ compiler.

The `Wave` C++ preprocessor library [24] uses the `Spirit` parser construction library to implement a C++ lexer with ISO/ANSI Standards conformant preprocessing capabilities. It provides an iterator interface, which returns the

current preprocessed token from the input stream. This preprocessed token is generated on-the-fly while iterating over the preprocessor iterator sequence.

The `xpressive` is a regular expression template library [25] dealing with static regular expressions. It can perform syntax checking and generates optimized code for static regexes.

Similar to our solution the new standard of C++ [16] also provides a typesafe `printf` function. The differences are explained in chapter 4.3.

6 Conclusion and Future Work

We have developed a metastring library based on `boost::mpl::string` and extended its functionality with the usual operations of runtime string libraries. We presented some applications which benefit from these metastrings.

One of these are the pattern matching algorithms that can be improved if either the text or the pattern are known at compile time. In this paper, our main goal was not to provide the best pattern matching solution, but to demonstrate the power of metastrings, thus we only dealt with Boyer-Moore and Knuth-Morris-Pratt pattern matching algorithm. Our work in the future aims to create a more sophisticated method – which counts more pattern matching algorithms – in order to find the optimal pattern matching solution.

We have created a C++ wrapper for `printf` taking the format string as a compile time string and validating the types of the runtime arguments based on that. In this solution, validation happens already at compile time, ensuring type-safety and providing that there is no runtime overhead either. We have compared our typesafe `printf` solution with Stroustrup's and found that our version provides stricter type-safety besides running significantly faster. However, Stroustrup's method is capable of deducing the format of the output by the argument's type, which is still missing from our version. Our future plan is to improve this `printf` function by implementing the `%a` specifier to provide a similar functionality.

We proved the benefit of our metastring library in empirical way on these applications. Moreover several other fields are exist in computer science, which can be improved using metastrings.

References

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C.: Introduction to Algorithms. The MIT Press (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. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools and Applications. Addison-Wesley, Reading (2000)
5. Czarnecki, K., Eisenecker, U. W., Glck, R., Vandevoorde, D., Veldhuizen, T. L.:Generative Programmind and Active Libraries. Springer-Verlag, (2000)

6. 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
7. Gil, J. (Y.), Lenz, K.: Simple and safe SQL queries with c++ templates, in proc. of Simple and safe SQL queries with c++ templates (2007), The ACM Digital Library pp. 13–24, (2007)
8. Juhász, Z., Sipos, Á., Porkoláb, Z.: Implementation of a Finite State Machine with Active Libraries in C++. In: GTTSE 2007. LNCS, vol. 5235, pp. 474–488. Springer, Heidelberg (2008)
9. Karlsson, B.: Beyond the C++ Standard Library, An Introduction to Boost. Addison-Wesley, Reading (2005)
10. 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)
11. Meyers, S.: Effective STL. Addison-Wesley (2001)
12. 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)
13. Programming languages C++, ISO/IEC 14882 (2003)
14. Siek, J., Lumsdaine, A.: Concept checking: Binding parametric polymorphism in C++. In: First Workshop on C++ Template Metaprogramming (October 2000)
15. 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
16. Stroustrup, B.: Evolving a language in and for the real world: C++ 1991-2006. ACM HOPL-III. June 2007
17. Stroustrup, B.: The C++ Programming Language Special Edition. Addison- Wesley, Reading (2000)
18. 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 Inter-operable Scientic and Engineering Computing (OO 1998), pp. 21–23. SIAM Press, Philadelphia (1998)
19. Veldhuizen, T.: Expression Templates. C++ Report 7(5), 26–31 (1995)
20. Veldhuizen, T.: Using C++ Template Metaprograms. C++ Report 7(4), 36–43 (1995)
21. http://www.boost.org/doc/libs/1_40_0/libs/mpl/doc/refmanual/string.html
22. http://www.boost.org/doc/libs/1_40_0/libs/mpl/doc/refmanual/vector-c.html
23. http://spirit.sourceforge.net/
24. http://www.boost.org/doc/libs/1_40_0/libs/wave/index.html
25. http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html
26. http://www.research.att.com/~bs/C++0xFAQ.html