Metaparse

Compile-time parsing with template metaprogramming

Ábel Sinkovics
Eötvös Loránd University
Dept. of Programming Languages and Compilers
Budapest, Hungary
abel@elte.hu

Zoltán Porkoláb
Ericsson Hungary
Eötvös Loránd University
Budapest, Hungary
zoltan.porkolab@ericsson.com
gsd@elte.hu

ABSTRACT

Metaparse is a C++ template metaprogramming library for generating parsers, which are template metaprograms themselves parsing strings at C++ compile-time. Parsers built with Metaparse take free-formed strings as input and parse them at compile-time, thus it is possible to build a parser and apply it in the same session of compilation. The C++11 standard provides constexpr, a construct for executing algorithms at compile-time. We present the connection between metaprogramming and constexpr and utilise it to minimise the syntactical overhead of the input processed by the parsers. Thus Metaparse is capable to solve a wide variety of tasks – from applying syntactical sugars for existing metaprograms to DSL integrations. An accurate error reporting helps Metaparse users to detect parsing errors. All solutions presented in this paper are implemented as an open source library and are available for the reader.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classification – C++

General Terms

Languages

Keywords

C++ template metaprogramming, Parsers

1. INTRODUCTION

Since Erwin Unruh demonstrated his famous prime number generator program [23], the research of C++ Template Metaprogramming has always been a hot topic among both academic researchers and software developers [25, 26, 3]. Today programmers write metaprograms for various reasons, like implementing *expression templates* [24], where runtime computations can be fine-tuned with compile-time activities to enhance runtime performance; *static interface checking*, which increases the ability of the compiler to check the requirements against template parameters by specifying constraints on them [10, 17]; *active libraries* [27], acting dynamically at compile-time, making decisions and optimisations based on programming contexts. Other applications involve embedded domain specific languages as the AraRarat system [5] for type-safe SQL interface, Boost.Xpressive [12] for regular expressions. Foundation techniques for data structures of template metaprograms have been developed based on *typelists*, and implemented in various ways [1, 7].

In this paper the authors present *Metaparse*, a C++ template metaprogram library for generating parsers with the help of parser combinators. Boost already has two parser generator libraries: Boost.Spirit and Boost.Proto. The main difference between Metaparse and Boost.Spirit is that parsers generated by Boost.Spirit run at runtime while parsers generated by Metaparse run at compile-time. Boost.Proto parsers work at compile-time processing valid C++ expressions, while parsers built with Metaparse take free-formed strings as input.

Being able to parse arbitrary text at compile-time can be useful in many situations. An example is parsing the format string of `printf` at compile-time and type-checking the arguments. We present use-cases from relatively simple to more complex ones. One can build a wrapper around Boost.Xpressive enabling the compile-time verification of regular expressions following the common syntax. Another use of compile-time parsers is transforming embedded DSL scripts into native C++ functions at compile-time and executing them at runtime. Our most complex example is demonstrating how one can implement an embedded DSL for defining template metafunctions. Metaparse is capable of generating parsers transforming such DSLs into metafunctions.

The new C++ standard provides constexpr, a construct for executing algorithms at compile-time. We present the connection between metaprogramming and constexpr and utilise it to minimise the syntactical overhead of the input processed by the parsers. We explain the internals of Metaparse, how it works and can be extended. We also present the accurate error reporting capabilities of the library.

All solutions presented in this paper are implemented as an open source library and are available for the reader. Meta-

parse and other libraries it is based on are available at: `http://abel.web.elte.hu/mpllibs` and `https://github.com/sabel83/mpllibs` [18].

The rest of the paper is organised as follows. In Section 2 we overview of the different approaches to generate parsers with the help of template metaprograms. Here we also discuss the new possibilities coming from the generalised constant expressions of C++11 and how they can be utilised to improve C++ template metaprogramming. In Section 3 we explain our approach to implement compile-time parsers. As accurate diagnostic is essential for parsers, in Section 4 we discuss the error reporting capabilities of Metaparse. In Section 5 we evaluate Metaparse by a number of usage examples. Our paper concludes in Section 6.

2. APPROACHES

Parsing embedded languages in C++ has been addressed in several ways. In this section we overview the different approaches available in Boost and Mpllibs.

2.1 Boost.Proto

Boost.Proto is a framework mainly for building Embedded Domain-Specific Languages in C++ [13]. It provides tools for constructing, type-checking, transforming and executing expression templates. Data structures for representing the expressions and a mechanism for giving additional behaviours and members to them are given.

The Proto parsers make use of the C++ parser of the compiler, thus the input of a Proto parser should be in valid C++ syntax. This could be a serious restriction as – among others – no C++ keywords or new operator symbols can be used in the defined DSL. On the other hand no quotations are required to identify the DSL code.

2.2 Boost.Spirit

Boost.Spirit is a set of C++ libraries for parsing and output generation that parse at runtime [4]. Spirit uses expression templates and template metaprogramming techniques to embed DSL scripts in the C++ source code as string literals. The embedding process doesn't introduce syntactical restrictions on the domain specific language, the C++ compiler passes the string literal to the parser as it is, without trying to interpret it.

The DSL scripts are parsed at runtime, even though they are available at compile-time. This has several disadvantages.

- The same DSL script has to be parsed every time the host program is executed. Depending on how the host program is organised, the same DSL script may be parsed more than once during the execution of the host program.
- Errors in the DSL script are not detected until runtime. The parsing of the scripts may not happen every time the host program is executed, only in cases when it is needed. As a result of this optimisation some DSL scripts may never be compiled during the testing period of the software and errors in them are detected by the end users.

2.3 Generalised constant expressions

C++11 provides new a feature called generalised constant expressions to extend the earlier practice of constant expressions to simple functions and objects of user-defined types with simple constructors [16]. Constexpr makes it possible to execute parts of the C++ code at compile-time. Functions whose body consists of one return statement only in which they return a constant expression can be marked as `constexpr` functions. When such a function is called in an expression and all of the arguments passed to it are calculated using constant expressions, the compiler evaluates the `constexpr` function at compile-time.

This feature can be used to parse a DSL script at compile-time. The DSL script can be passed to a `constexpr` function as a string literal. The function can do the parsing and return some result value based on this. Given that the function is a `constexpr` function and the argument that is passed to it – the string literal containing the DSL script – is a constant expression, parsing happens at compile-time. Invalid DSL scripts that can not be parsed are detected at compile-time and the cost of parsing is paid during the compilation of the host program. It doesn't introduce runtime costs, however, it may introduce high compile-time costs which can lead to issues during development.

The approach to construct optimised code or types as the result of parsing a DSL script when parsing happens using `constexpr` parsers is an open question.

2.4 Template metaprogramming

Template metaprograms can construct constant values, types or code that the compiler can optimise. Given that template metaprogramming is a Turing-complete sub-language of C++ [26, 3], one can implement parsers as template metaprograms. Similarly to parsers implemented as `constexpr` functions, the ones implemented as template metaprograms can process a DSL script at compile-time. The result of parsing a DSL script can be anything a template metaprogram can return, thus using this approach one can construct types or optimised code. Given that higher order template metafunctions – template metafunction classes – are represented by types, these parsers can construct template metaprograms as the result of parsing. It makes the implementation of a better syntax for C++ template metaprograms possible.

2.5 Combining constexpr and template metaprogramming

Template metaprogramming and generalised constant expression based parsing can be combined together.

- When a constant expression returns an integral value, it can be a template argument, thus partial results produced by `constexpr`-based parsers can be passed to template metaprogramming ones.
- When a template metaprogram returns a boxed `constexpr` value, it can be part of generalised constant expressions, thus partial results produced by template metaprogramming-based parsers can be passed to `constexpr`-based ones.

An example demonstrating the above approaches can be found in [18]. A combined approach can take advantage of the easier syntax of `constexpr` functions and the expressiveness of template metaprograms at the same time.

3. PARSERS

Metaparse [18] uses template metaprogramming to implement parsers. The input of the parsers are strings, thus we need to be able to represent them as template metaprogramming values to be able to do the parsing. Boost.MPL provides a string implementation we can use. However, the syntax of embedding strings in the source code is the following:

```
mpl::string<'Hell','o Wo','rld!'>
```

It is based on multi-character constants, which allow grouping four characters together. However, the embedded string has to be split into four character chunks which is not convenient and is difficult to work with. The following syntax would be easier to use:

```
_S("Hello World!")
```

It can be provided by combining generalised constant expressions and preprocessor metaprograms. `"Hello World!"` is a string literal, which becomes a character array. Accessing a character of it, such as `"Hello World!"[2]` is a constant expression. A constant expression can be an argument of a template, thus we can do the following:

```
mpl::push_back<
  mpl::push_back<
    // ...
    mpl::push_back<
      mpl::string<>,
      mpl::char_<"Hello World!"[0]>
    >::type,
    // ...
    mpl::char_<"Hello World!"[10]>
  >::type,
  mpl::char_<"Hello World!"[11]>
>::type
```

The above code appends each character of the string literal `"Hello World!"` to an empty template metaprogramming string one by one. The above code can be hidden by a macro. Using preprocessor metaprograms, such as `BOOST_PP_REPEAT` from Boost.Preprocessor [9] we can generate the above code from `_S("Hello World!")`. The problem is that we need to know the length of the string in advance, thus our macro can work with a predefined string length only. Generating different code based on the length of the string literal in a preprocessor metaprogram is an open question, we need to specify a fixed number of steps we generate and use that in all cases. During the generated iterations we need to detect when we reach the end of a string. We can define a special metafunction that optionally appends a character to a string:

```
template <class S, char C, bool EndOfString>
struct append_string :
  mpl::push_back<S, mpl::char_<C>>
{};

template <class S, char C>
struct append_string<S, C, true> : S {};
```

This metafunction takes an extra argument, which is a boolean value indicating if the character really needs to be appended. To avoid over indexing the string literal, we need a `constexpr` function accessing a character of the string:

```
template <int Len>
constexpr char str_at(const char (&s)[Len], int n)
{
  return n >= Len ? 0 : s[n];
}
```

The above function takes a char array and an index as arguments. It receives the length of the array as a template argument, thus it can check if the array has been over indexed. When it is over indexed, it returns the `'\0'` character. Since it is a `constexpr` function, its result can be used as a template argument.

Using all the above, we can construct template metaprogramming strings from string literals:

```
#define s "Hello World!"

append_string<
  append_string<
    // ...
    append_string<
      mpl::string<>,
      str_at(s, 0),
      (0 >= sizeof(s) - 1)
    >::type,
    // ...
    s[10],
    (10 >= sizeof(s) - 1)
  >::type,
  s[11],
  (11 >= sizeof(s) - 1)
>::type
```

The above code uses `str_at` to access the characters, thus it is protected against over indexing the array. It uses `sizeof` to check the length of the string and `append_string` to append the characters. To each `append_string` call it passes a Boolean value indicating if that character is a character of the string or just a 0 character coming from `str_at`.

The number of appends we generate from our macro sets an upper limit on the length of the string. However, we can use the same technique Boost.MPL uses to provide variadic metafunctions [6]. The maximum value can come from a macro, such as `MPLLIBS_LIMIT_STRING_SIZE` which can be defined by the user.

We have provided a way to embed DSL code snippets into C++ source code. We can now deal with how these snippets can be parsed. Metaparse is based on parser combinators [8]. A parser is a template metafunction taking two arguments:

- A suffix of the input text. This is the text to parse.
- The line and column number of the first character of this suffix in the original text. This information is useful for error reporting.

As the result of parsing a parser returns one of the following:

- Some arbitrary value representing the parsing result, the unprocessed suffix of the input and the line and column number of the unprocessed suffix. This return value indicates success.
- A line and column number and a string describing a parsing error. This return value indicates that the parsing has failed.

To be able to differentiate between the two cases, Metaparse provides the `is_error` metafunction that can tell about a result if it is an error or not. This decision is based on tag dispatching. The value returned by a metafunction is either tagged by `accept_tag` or `fail_tag`. A parser is not expected to consume the entire input. It returns the unconsumed part.

As an example we present a simple parser that rejects every input. We call it `fail`. In general to make passing parsers to parser combinators easier, we implement them as template metafunction classes instead of template metafunctions.

```
struct fail {
  // This metafunction is the parser
  template <class S, class Pos> struct apply {
    // Make it work with lazy metafunctions
    typedef apply type;
    // Make it work with is_error
    typedef fail_tag tag;
    // Make the location of the error available for
    // error diagnostic
    typedef Pos source_position;
    // Some error message
    typedef _S("Parsing failed") message;
  };
};
```

To be able to give a more meaningful error message than just *Parsing failed*, we make `fail` a template class taking the error message as a template argument:

```
template <class Msg> struct fail {
  template <class S, class Pos> struct apply {
    // ...
    typedef Msg message;
    // ...
  };
};
```

As another example we implement a parser accepting everything without consuming any input. We call it `return_` – the name is important when we do monadic parsing. We refer to [21] on monadic parsing using Metaparse. `return_` needs a value that will be the result of parsing. We make `return_` a metafunction class and make this result a template argument of it.

```
template <class Result> struct return_ {
  // This apply metafunction is the parser
  template <class S, class Pos> struct apply {
    // Make it work with lazy metafunctions
    typedef apply type;
    // Make it work with is_error
    typedef accept_tag;

    // The result of parsing
    typedef Result result;
    // The unconsumed suffix
    typedef S remaining;
    // The position of the suffix's first character
    typedef Pos source_position;
  };
};
```

Parser combinators are higher order functions. They take one or more parsers – which are functions – as arguments and return a new parser. Using parser combinators one can have a small number of simple parsers and build more complex ones out of them.

We build a parser consuming the first character of the input string. The result of parsing is this character. This parser fails for empty input. As we'll see, this is the only parser consuming input – the rest of the parsers use this one to consume input. We call this parser `one_char`. We use `return_` and `fail` to report success or error – we don't need to re-implement that logic.

```
struct one_char {
  template <class S, class Pos> struct apply :
    mpl::eval_if<
      typename mpl::empty<S>::type,
      mpl::apply_wrap2<
        fail<_S("Unexpected end of input")>,
        S, Pos
      >
      mpl::apply_wrap2<
        return_<typename mpl::front<S>::type>,
        mpl::front<S>,
        // Not covered: calculate the
        // next source position
        Pos
      > > {};
};
```

This code checks if the input is empty by using `mpl::eval_if` and `mpl::empty`. If it is empty, it reports the error using `fail`, otherwise it returns success using `return_`. How the next source position can be calculated is out of scope here. We refer to the implementation of Metaparse [18].

As a simple example for parser combinators we can build one that takes a parser, applies it on the input and checks the result of it. In this last checking step it may accept or reject the result. It can be implemented the following way:

```
template <class P, class Pred, class Msg>
struct accept_when {
  template <class R, class S, class Pos>
  struct impl : mpl::eval_if<
    typename mpl::apply_wrap1<
      Pred,
      typename R::type
    >::type,
    R,
    mpl::apply_wrap2<fail<Msg>, S, Pos>
  > {};

  template <class S, class Pos>
  struct apply : mpl::eval_if<
    typename is_error<
      mpl::apply_wrap2<P, S, Pos>
    >::type,
    mpl::apply_wrap2<P, S, Pos>,
    impl<mpl::apply_wrap2<P, S, Pos>, S, Pos>
  > {};
};
```

This code defines `accept_when`, which takes 3 template arguments: the parser to transform, a predicate to check the result of the parser with and an error message that can be returned when the predicate rejects the result. It applies the original parser on the input. When it fails, the error message is escalated. When it succeeds the result is checked and either this result or an error with the predefined message is returned. Because of difficulties with Boost.MPL and lazy evaluation [19] a helper metafunction, `impl` is defined.

We don't present Metaparse and parse combinators in further detail here. We refer to earlier publications on the topic [21, 22, 15].

4. ERROR REPORTS

Parsers often have to deal with invalid input. In such cases, they are the ones that detect the errors and have to report it to the developer. In many cases the developer has to find and fix the problem based on these error reports. Since high quality error reports can help the developer finding and fixing the problem, they can improve productivity. During the presentation of Metaparse and its approach, we've already shown where the error reports are coming from. This chapter is about how it can be displayed to the developer.

Parsers built with template metaprogramming are executed as part of the C++ compilation process. When they detect errors, they have to report it to the developer in a way that is easy to understand. The following options are available for reporting errors from the C++ compilation process.

- Breaking the C++ compilation.
- Generating code that displays the error at runtime [22].

The first approach fits more naturally to the development process of C++ applications. The developer writes the code containing the embedded snippets to parse at compile time, compiles it and gets the errors. However, the error report coming from the compiler is about the templates and template instantiations metaprograms are implemented with, not about the problem with the embedded script.

The second approach, which generates code displaying the error message produces easy to read and informative error messages. Its drawback is that it doesn't fit that well to the development process most programmers are used to. The developer writes the code containing the embedded DSL snippet. He parses the code and gets some – probably meaningless – error report from the compiler pointing to a DSL snippet. The developer has to copy that snippet to another program, compile and run it. As a result of running it, he can get the error message. Metaparse provides a tool, `debug_parsing_error` to support this approach. It is a template taking the parser and the invalid input as arguments. The constructor of the template instance displays the debug information on the standard output, thus it can be used the following way:

```
typedef /* ... */ some_parser;

debug_parsing_error<some_parser, _S("Bad input")>
  show_result();
```

We create an object of type `debug_parsing_error<...>` to instantiate and run the constructor that displays the debug information.

Even though it requires some extra effort from the developer, the error message it can display is about the DSL and the embedded snippet. It points to developer to the problematic location of his DSL code snippet and tells what the problem is – without any syntactic noise coming from the C++ compiler.

5. APPLICATIONS

Being able to parse at compile time makes it possible to embed code snippets implemented in domain-specific languages. These snippets can implement the following types of things:

- *Generate types.* New types can be constructed as the result of parsing a DSL script and can be passed to metaprograms or can be instantiated.
- *Generate optimised executable code.* Template metaprograms and parsers implemented using them can generate executable code by combining small inline functions. This gives the compiler the opportunity to optimise the code.
- *Generate runtime objects.* Metaprograms can generate code that initialises these objects during static initialisation.
- *Generate constant values.* The result of metaprograms and parsers implemented as metaprograms can be a constant expression producing constant values.

- *Do compile-time assertions.* Metaprograms can optionally break the compilation process based on some conditions. These conditions can be implemented in a DSL and parsed at compile-time.
- *Generate template metaprograms.* Since types can be generated as the result of parsing a DSL script, template metaprograms can be generated as well. This makes it possible to implement easy to read languages for template metaprogramming, that work like scripts interpreted by the C++ compiler.

We highlight three areas where using compile-time parsers can make a significant difference.

5.1 Interface of libraries

Libraries have their own specific domain with their own common notations. The more the interface of the library follows that notation, the easier the experts of that domain can use it. Many C++ libraries [5, 13] try to follow the syntax of special problem domains by overloading C++ operators. Boost provides the Proto library making the development of such approaches easier. However, this approach has its constraints: all DSL expressions have to be valid C++ expressions as well.

Being able to parse DSL snippets at compile-time makes it possible to provide an interface that follows the notation of the domain without being constrained by the syntax of C++ expressions.

As an example, one can look at Boost.Xpressive. It provides an interface for building regular expressions. Assuming that the regular expression is available at compile-time, the user of the library can choose one of the following options:

- Embed the regular expression in the C++ code as a string literal. This is parsed at runtime – parsing has its own runtime cost and when the regular expression is invalid, it causes a runtime error, thus, the problem is not detected until runtime.
- Embed the regular expression in the C++ code as a C++ expression. Xpressive provides an interface for that and it follows the logic of regular expressions. However, the commonly used syntax had to be altered to make it a valid C++ expression, which makes it difficult for people not familiar with the syntax of Xpressive to read and understand it.

A third option, offered by compile-time parsers is embedding the regular expression as a string literal and parsing it at compile time to build the same structure one could build with the C++ expression-based interface.

5.2 Template metaprogramming

Since the result of parsing is a type, we can construct template metaprograms as the result of parsing an embedded DSL code snippet. Using this technique we can build an easy to read language for C++ template metaprograms. Given the similarities between Haskell and C++ template metaprogramming [14, 11, 2] the language we are building will be similar to Haskell. We start with the following simple language and extend it later.

```
single_exp ::= int_token | name_token
```

Using this language we can write simple expression. An expression is either an integer value or the name of a variable or function. We parse it into an abstract syntax tree (AST). In the AST we use the following templates to represent integer values and variable or function references:

```
template <class Val> struct ast_value;
template <class Name> struct ast_ref;
```

`ast_value` represents a value, `ast_ref` represents a reference to something. We can construct a parser for the above grammar.

```
typedef transform<int_token,
  mpl::lambda<ast_value<_1>>::type> int_exp;
typedef transform<name_token,
  mpl::lambda<ast_ref<_1>>::type> name_exp;
typedef one_of<int_exp, name_exp> single_exp;
```

We created a parser parsing integers (`int_exp`) and one parsing names (`name_exp`). We get `int_token` from Metaparse, but we have to implement `name_token` ourselves. We don't present the details of this here. The `_token` parsers return the parsed value which we turn into an AST by using the `transform` combinator. Finally, we combine the two parsers by using the `one_of` combinator to accept either an integer or a name.

We extend the above language with function application. The syntax is the following: `<function> <arg1> <arg2> ...` where `<function>` is an expression evaluating to a function. For example it can be the name of a function, or an expression evaluating to a function (we present such expressions late). We use the following grammar for function application:

```
application ::= single_exp+
```

We represent both the function and its arguments as expressions. We expect the first expression to evaluate to a function that accepts at least as many arguments, as we apply on it. Accepting more will not be a problem, because our language will support currying [19, 20]. Not meeting this requirement will cause an error during the execution of the compiled metaprogram. We don't implement a type system that could detect these errors ahead of execution. We represent applications in the AST by instances of the following template:

```
template <class F, class Arg>
struct ast_application;
```

Following the logic of currying, we apply arguments one by one. Thus by parsing the expression `f 1 2 3` we get the following AST:

```
ast_application<
  ast_application<
    ast_application<
      ast_ref<mpl::string<'f'>>,
      ast_value<int_<1>>>,
    ast_value<int_<2>>>,
  ast_value<int_<3>>>
```

The above AST applies the arguments on the function `f` one by one. The parser for `application` can be constructed from `single_exp` by using the `any1` and `transform` parser combinators provided by Metaparse. It can be implemented in a more efficient way by using the `foldl` parser combinator (also provided by Metaparse). We don't present the details here. An implementation can be found in [18].

After building the AST, we need to bind the references in it to metafunctions and values (which are classes in template metaprogramming). To be able to do this, we need an associative container describing the binding rules. We can use `mpl::map` for that. The keys are the names, the values are the values (or metafunctions classes) to bind to. We can implement a metafunction, `bind` taking an AST and an `mpl::map` as arguments and doing the binding. The result of this binding is a nullary metafunction. The evaluation of the expression happens by evaluating that metafunction (and not when the binding happens). We construct these nullary metafunction by combining the following templates:

```
template <class V>
struct lazy_value { typedef V type; }

template <class F, class Arg>
struct lazy_application :
  F::type::template apply<Arg>::type {};
```

We have a template representing values and one representing function applications. There is no template representing references, since we construct the nullary metafunctions after binding. We expect all references to be resolved during binding. Invalid references are interpreted as errors.

Since we assume that values are evaluated lazily, we need to enforce the evaluation of the function we apply the argument on in the implementation of `lazy_application`.

`bind` can be implemented based on the above. We don't present the details here. An implementation of it can be found in [18]. We need to be able to construct the `mpl::map`. We introduce the following syntax for this:

```
template <class T> struct f;
```

```
typedef meta_hs
  ::import<_S("some_value"), mpl::int_<13>>::type
  ::import1<_S("f"), f>::type
  ::import2<_S("plus"), mpl::plus>::type
sample_map;
```

`meta_hs` is a class we implement. It has nested template classes called `import`, `import1`, `import2`, etc. They take a name as a template metaprogramming string and the referenced entity. `import` adds values, while `import1`, `import2`, ... add metafunctions to the `mpl::map`. The latter take regular metafunctions as their second argument, add currying to them [19] and add them to the `mpl::map`. The number after `import` is the number of the arguments. We don't present their implementation here. They can be found in [18].

We can bind names to *regular* metafunctions in the expressions. We can extend `meta_hs` to be able to implement functions in the language we build for template metaprogramming and bind them to names in the `mpl::map`. The syntax is be the following:

```
typedef meta_hs
  ::define<_S("f x y = ...")>::type
sample_map;
```

This definition defines a function using the DSL we're building and binds it to a name. We store the AST of the expression in the `mpl::map` we build and postpone binding it. The binding happens when the value is looked up, thus it can reference named that are defined after the current definition. Since we store ASTs instead of values in the `mpl::map`, the values we have stored in it has to be turned into ASTs. We don't want to do any binding on them, thus we need to introduce a special AST element, `ast_bound` that is skipped by the binding process. We use that for wrapping everything the `import` metafunctions add to the `mpl::map`.

When the `mpl::map` mapped the references to values, binding was a simple process: the reference had to be replaced by the referenced value. Now that we store an AST, we need to resolve the referenced AST before the substitution. `bind` can be extended to support it, we don't present the implementation details here.

Now that we can add functions defined in the new DSL to the `mpl::map`, we need to extend the DSL to make it possible to implement such functions in it. So far we can write expressions. We extend it in a way that we'll be able to write function definitions as well:

```
definition ::= name_token+ define_token application
```

The definition of a function begins with the name of the function followed by the names of the arguments. These are the `name_token` part of the above rule. It is followed by `define_token`, which expects an `=` character. An expression (`application`) describes the body of the function. We extend the AST to be able to describe function definitions. We introduce lambda abstractions:

```
template <class F, class ArgName>
struct ast_lambda;
```

It describes one lambda argument. A function expecting multiple arguments can be represented by nested lambda abstractions. The `bind` metafunction binding names to values has to be prepared for lambda abstractions as well. A lambda abstraction has an argument, `ArgName`. When a value is applied to this lambda abstraction, it binds that value to the name `ArgName` in the body of the lambda abstraction [19, 20]. To implement it, `bind` has to create template metafunctions class from the `lambda` elements of the AST. The following metafunction (wrapped into a class) can be returned by this binding:

```
template <class ArgValue>
struct apply : bind<
  F, typename mpl::insert<
    Env, mpl::pair<ArgName, ArgValue>
  >::type
>::type {};
```

`Env` is the `mpl::map` describing the mapping of the values to names. This metafunction maps `ArgName`, the lambda argument to the value that the lambda abstraction was applied on. This mapping happens by calling `mpl::insert`. Then this metafunction does the binding of the body of the lambda abstraction using this new mapping.

When the binding of an AST happens, a lambda abstraction stores the mapping and keeps the body of the abstraction as an AST. The binding process continues when the resulting code is evaluated and the lambda argument is known.

Since we do the binding of the lambda abstractions at the execution time of the compiled metaprograms, we assume the argument to already be bound and we have to avoid binding it again. We can use `ast_bound` for this. The `bind` function unwraps it and leaves the content unchanged. Using this new element, we can protect the argument of a lambda abstraction to be bound multiple times by calculating the new environment the following way:

```
typename mpl::insert<
  Env, mpl::pair<ArgName, ast_bound<ArgValue>>
>::type
```

The above code is almost the same as the one we had before with the only difference that we map `ArgName` to `ast_bound< ArgValue>` instead of `ArgValue`. Mapping to ASTs instead of values introduces another problem as well. Consider the following example:

```
meta_hs
  ::define<_S("x = s")>::type
  ::define<_S("s = 13")>::type
  ::define<_S("f s = x")>::type
```

When we apply something – eg. 11 – on `f`, the lambda abstraction does the binding of the body of `f`, that is the expression `x`. `x` refers to the first definition, its body is the expression `s`. Since we store ASTs in the `mpl::map`, before substituting the reference `x` in the body of `f` we do the binding of `x` as well. But in the `mpl::map` we're using in the body of the lambda expression in `f` the `s` reference has been overridden. It refers the value `11` instead `13` now, thus the result of evaluating `f 11` will be `11`, while it should be `13`. The problem is that the updated `mpl::map` we use inside the lambda abstraction belongs to the body of the lambda abstraction only. The original `mpl::map` has to be used to do the binding of the elements referenced by the body of the lambda abstraction. To be able to implement this, the `bind` metafunction takes two environments as arguments: the original and the current `mpl::map`. It uses the current one to do the mapping, but when it does the mapping of a referenced AST, it uses the original one. We don't present the implementation of it here. It can be found in `sinkovics:mpllibs`. We can parse the following function definitions now:

```
definition ::= name_token+ define_token application
```

The first `name_token` is the name of the function. The rest of them are the argument names. The `application` is the body of the function. Bind builds the AST of the body by parsing `application` and construct a lambda abstraction from it using the rightmost argument as the lambda argument. The lambda abstraction is an AST itself as well, thus we can build another lambda abstraction from it by using the argument before the last one. We can continue until we have used all of the arguments. The order is important to make it work with currying correctly. When there are no arguments, we don't build any lambda abstractions but leave the body as it is. This way we can define constant values using this language.

As a result of parsing the above, we get a function name and an AST describing the body. By doing the above parsing in a `define` element, we can bind the resulting function to the resulting name in the `mpl::map` `meta_hs` builds. This function can be used by other functions we build with the same `meta_hs`.

This way, we can build simple functions using our language for template metaprogramming and we can construct more complicated ones from them. We get a `meta_hs` mapping as the result. We can use *regular* metafunctions, defined outside of the `meta_hs` block by importing them using techniques described above. To be able to construct metafunctions that can be used outside of the `meta_hs` block, we need a way to export them. We introduce the following syntax for exporting metafunctions from `meta_hs` blocks:

```
typedef meta_hs
  ::define<_S("id x = x")>::type
  ::get<_S("id")>::type
id;
```

`get<_S("...")>` exports a metafunction. Its result is a metafunction class that behaves the same way as any other metafunction class:

```
typedef id::apply<int>::type also_int;
```

`define` elements bind abstract syntax trees to names. Metafunctions are constructed when the binding happens. Thus, `get` has to do the binding of the AST the name is mapped to. The mapping `get` uses is the entire mapping `meta_hs` has constructed, thus functions can be referenced before they are defined – names are resolved when the exporting happens. For example, the following works:

```
typedef meta_hs
  ::define<_S("f x = g x")>::type
  ::define<_S("g x = x")>::type
  ::get<_S("f")>::type
f;
```

In the above code `f` uses `g`, which is defined later. But it works, since `f` is exported after `g` has been defined. The above language can be extended to use operators and brackets. ASTs calling functions with special names can be constructed from operator usage. For example `11 + 2` can be parsed into:

```
ast_application<
  ast_name<_S(".+.")>,
  ast_value<mpl::int_<11> >,
  ast_value<mpl::int_<2> >
>
```

The above code turns operator `+` usage into `.+.` function calls.

Half-constructed `meta_hs` blocks are types, thus one can create type aliases for them.

```
typedef meta_hs
  ::define<_S("f x = g x")>::type
  ::define<_S("g x = x")>::type
my_lib;
```

The above code defined two functions, `f` and `g` and creates a type alias for the result. As its name – `my_lib` – suggests, one can create template metaprogramming libraries this way. `my_lib` can be used the same way as `meta_hs` for defining further metafunctions:

```
typedef my_lib
  ::define<_S("h x = f f x")>::type
  ::get<_S("h")>::type
h;
```

The above code defines a metafunction called `h` that uses `my_lib` and the `f` function provided by it. When operator calls, such as operator `+` are parsed into special function calls, such as `.+.`, this technique can be used to map those function names to imported metafunctions implementing the operator evaluations. For example:

```
template <class A, class B>
struct lazy_plus :
  mpl::plus<typename A::type, typename B::type> {};

typedef meta_hs_base // some base class
  ::import2<_S(".+."), lazy_plus>::type
meta_hs;
```

The above code defines a function for `.+.` in `meta_hs`. To be able to do that, it has to use a different name for the base we've called `meta_hs` so far. This different name is `meta_hs_base`. Due to the lack of lazy evaluation in Boost.MPL [7], we have to define a lazy version of the `plus` metafunction. Using `import` can't make metafunctions automatically lazy, because it wouldn't work with metafunctions that are already lazy.

What we have presented is just a core language. It can be extended to support further features of the Haskell language, such as pattern matching, control structures, etc. We leave it as a future work.

5.3 Interface for itself

Since one can build better syntax for template metaprograms using Metaparse and Metaparse itself is a template metaprogram, one can build a better syntax for Metaparse using Metaparse. The logic is similar to the one we presented above for template metaprograms in general. We can provide a syntax for constructing an `mpl::map`. This maps names of non-terminals to parsers. A grammar definition looks like the following:

```
typedef grammar<_S("S")>
  ::import<_S("int"), int_>::type

  ::rule<_S("S ::= '-'* int")>::type
integers;
```

The above code defines a grammar using `S` as the start symbol. It defines a rule for `S` expecting any number of `'-'` characters and an integer. The grammar doesn't define the rule for parsing integers itself, it imports the `int_` parser provided by Metaparse.

The implementation of this is similar to the implementation of the Haskell-like language we have presented. Metaparse has this `grammar` element, which can be used to implement parsers. We refer to the implementation of this for further details.

6. CONCLUSION

In this paper we presented Metaparse, a C++ template metaprogramming library for generating parsers. Metaparse functionality exceeds the existing parsers, Boost.Spirit and Boost.Proto in various ways. The main difference between Metaparse and Boost.Spirit is that parsers generated by Boost.Spirit run at runtime while parsers generated by Metaparse run at compile-time – possibly in the same compilation session as the parsers were generated. Boost.Proto parsers work at compile-time but their input is restricted to syntactically valid C++ expressions, while parsers built with Metaparse take free-formed strings as input.

Being able to parse arbitrary text at compile-time can be useful in many situations. One usage area is to write wrappers around existing metaprogram libraries to provide a more user-friendly syntax. An other example is to extend type-checking possibilities of C++, like in the case of implementing a type-safe `printf`. It is also possible to transform embedded DSL scripts into native C++ functions at compile-time and executing them at runtime. We also demonstrated how one can implement an embedded DSL for defining template metafunctions. Metaparse is capable of generating parsers transforming such DSLs into metafunctions.

7. ACKNOWLEDGEMENTS

The Project is supported by the European Union and co-financed by the European Social Fund (grant agreement no. TAMOP 4.2.1./B-09/1/KMR-2010-0003).

8. REFERENCES

[1] A. Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] J.-P. Bernardy, P. Jansson, M. Zalewski, and S. Schupp. Generic programming with c++ concepts and haskell type classes: A comparison. *J. Funct. Program.*, 20:271–302.

[3] K. Czarnecki and U. W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.

[4] J. de Guzman and H. Kaiser. Boost.Spirit, 2011. http://boost-spirit.com/home/.

[5] J. Y. Gil and K. Lenz. Simple and safe sql queries with c++ templates. *Sci. Comput. Program.*, 75:573–595, July 2010.

[6] D. Gregor and J. Järvi. Variadic templates for c++0x. *Journal of Object Technology*, 7(2):31–51, 2008.

[7] A. Gurtovoy and D. Abrahams. Boost.Mpl, 2004. http://www.boost.org/doc/libs/1_47_0/libs/mpl/doc/index.html.

[8] G. Hutton and E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[9] V. Karvonen and P. Mensonides. Boost.Preprocessor, 2010. http://www.boost.org/doc/libs/1_47_0/libs/preprocessor/doc.

[10] B. McNamara and Y. Smaragdakis. Static interfaces in C++. In *C++ Template Programming Workshop*, Oct. 2000.

[11] B. Milewski. What does haskell have to do with c++?, 2009. http://bartoszmilewski.wordpress.com/2009/10/21/what-does-haskell-have-to-do-with-c/.

[12] E. Niebler. Boost.Xpressive, 2007. http://www.boost.org/doc/libs/1_47_0/doc/html/xpressive.html.

[13] E. Niebler. Boost.Proto, 2011. http://www.boost.org/doc/libs/1_49_0/doc/html/proto.html.

[14] B. O'Sullivan, J. Goerzen, and D. Stewart. *Real World Haskell*. O'Reilly Media, Inc., 1st edition, 2008.

[15] Z. Porkoláb and Á. Sinkovics. Domain-specific language integration with compile-time parser generator library. In E. Visser and J. Järvi, editors, *GPCE*, pages 137–146. ACM, 2010.

[16] G. D. Reis, B. Stroustrup, and J. Maurer. Generalized constant expressions (revision 5). Technical Report N2235=07-0095, ISO/IEC JTC 1, Information Technology, Subcommittee 22, Programming Language C++, Apr. 2007.

[17] J. G. Siek and A. Lumsdaine. Concept checking: Binding parametric polymorphism in C++. In *Proceedings of the First Workshop on C++ Template Programming*, Erfurt, Germany, Oct. 2000.

[18] Á. Sinkovics. The source code of mpllibs, 2012. Available as http://github.com/sabel83/mpllibs.

[19] Á. Sinkovics. Functional extensions to the boost metaprogram library. *Electr. Notes Theor. Comput. Sci.*, 264(5):85–101, 2010.

[20] Á. Sinkovics and Z. Porkoláb. Expressing c++ template metaprograms as lamda expressions. In Z. Horváth, R. Plasmeijer, and V. Zsók, editors, *CEFP*, volume 6299 of *Lecture Notes in Computer Science*, pages 97–111. Springer, 2009.

[21] Á. Sinkovics and Z. Porkoláb. Implementing monads for c++ template metaprograms. Technical Report TR-01/2011, Eötvös Loránd University, Sept. 2011. http://plcportal.inf.elte.hu/en/publications/TechnicalReports/monad-tr.pdf.

[22] Á. Sinkovics, E. Sajó, and Z. Porkoláb. Towards more reliable c++ template metaprograms. In J. Penjam, editor, *SPLST'11*, pages 260–271. TUT Press, 2011.

[23] E. Unruh. Prime number computation, 1994.

[24] T. Veldhuizen. Expression templates. *C++ Report*, 7:26–31, 1995.

[25] T. Veldhuizen. *Using C++ template metaprograms*, pages 459–473. SIGS Publications, Inc., New York, NY, USA, 1996.

[26] T. L. Veldhuizen. C++ templates are turing complete. Technical report, 2003.

[27] T. L. Veldhuizen and D. Gannon. Active libraries: Rethinking the roles of compilers and libraries. In *In Proceedings of the SIAM Workshop on Object Oriented Methods for Inter-operable Scientific and Engineering Computing OO'98*. SIAM Press, 1998.