Domain-specific Language Intergration with Compile-time Parser Generator Library

Zoltán Porkoláb and Ábel Sinkovics

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

Abstract. Smoothless integration of domain-specific languages into a general purpose programming language requires to absorb domain code written in arbitrary syntax. The integration should cause minimal syntactical and semantical overhead and ideally introduces a minimal dependency on external tools. In this paper we introduce a DSL integration technique for C++ programming language. The solution is based on compile-time parsing of the DSL code. The parser generator is implemented as a C++ template metaprogram library and the full parsing phase is executed when the host program is compiled. Therefore the host language can make compile-time adaptions depending the parsed DSL code. The library uses only standard C++ language features, thus our solution is highly portable.

1 Introduction

Modern general purpose programming languages have the ability to express regular programming idioms in a fairly convenient way: functions, types, classes and class hierarchies, etc. are used to express the programmer's intention. Mostly these tools are applied when the programmer transfers a solution from a specific problem domain. Such transformations require not only good programmer skills in the means of the classical programming language terms but a profound understanding of the specific problem domain.

As an opposite, *Domain-specific languages* (DSLs) are created to express problems in particular domains only. Using DSLs in specific problem areas have many advantages. DSLs are regularly more expressive in the intended problem domain. As an example, the `SQL` language is perfect to express relational database related problems while features of general purpose languages lack this clarity. The special syntax of a DLS is able to catch errors specific to the problem domain. DSLs often invent new constructs to describe domain poroblems or they even apply diffrent programming paradigm. SQL as an example follows declarative paradigm. Thus the syntax of a DSL may reflect the usual notations of the domain, make its usage accessible for the domain experts.

Although DSLs are indispensible in their domain, vast majority of the programs will execute most of their actions out of this domain. SQL might be a perfect solution for describing operations related relational databases, database servers will create threads, open network connections, communicate with the operating system in the means of a general purpose programming language. The usual solution is that the desired Domain-specific language or languages are used together a general purpose programming language. Most cases the integration of these languages happens embedding the DSL(s) into the general purpose language with or without some syntactical quotation.

However, this integration should cause minimal syntactical and semantical overhead on the project. There are many strategies exist to provide the smoothless integration of domain languages and the host language. Some of them applies external framework for integration, others are build on language extensions. A few solution is based on standard programming language features like macros or generative language elements.

Not all of these solutions can be applied in industrial environment. Extrenal tools may introduce unwanted dependency on 3rd party software developers. Language extensions require translators, precompilers or the modification of the compiler. These are fragile solutions when new language or compiler versions appeair. The most portable, manageable solution is based purely standard language features.

In this paper we introduce a DSL integration technique for C++ programming language. The solution is based on compile-time parsing of the DSL code. The parser generator is implemented as a C++ template metaprogram library, and the full parsing phase is executed when the host program is compiled. Therefore the host language can make compile-time adaptions depending the parsed DSL code. The library uses only standard C++ language features, thus our solution is highly portable.

The rest of the paper is organized as follows. In Section 2 we overview C++ template metaprogramming. Current DSL embedding technologies are discussed in Section 3 with their advantages and shortages. In Section 4 we argue for a compile-time parser presenting a number of motivating examples. We explain our template metaprogram based parser in Section 5 with sufficient implementational details. In Section 6 we evaluate our solution with the help of examples. Our paper concludes in Section 7.

2 C++ Template Metaprogramming

Templates are key language elements for the C++ programming language [3]. They are essential for capturing commonalities of abstractions without performance penalties at runtime. The most notable example is the Standard Template Library [8] which is now an unavoidable part of professional C++ programs. In 1994 Erwin Unruh wrote a heavily templated program [17] in C++ which didn't compile, however, the error messages emitted by the compiler during the compilation process displayed a list of prime numbers. Unruh used C++ templates

and the template instantiation rules to write a program that is "executed" as a side effect of compilation. It turned out that a cleverly designed C++ code is able to utilize the type-system of the language and force the compiler to execute a desired algorithm [20]. These compile-time programs are called C++ *Template Metaprograms* and later has been proved to be form a Turing-complete sub language of C++ [5].

C++ template metaprogram actions are defined in the form of template definitions and are "executed" when the compiler instantiates these templates. Their instantiations can instruct the compiler to execute other instantiations, since templates can refer to other templates. This way we get an instantiation chain very similar to a call stack of a runtime program. Recursive instantiations are not only possible but regular in template metaprograms to model loops:

```
template <int N>
class Factorial
{
public:
    enum { value = N*Factorial<N-1>::value };
};

template<>
class Factorial<1>
{
public:
    enum { value = 1 };
};

int main ()
{
    const int r = Factorial<5>::value;
}
```

Two important template rules have been tacitly used here: (1) Templates which are not referred must not be instantiated – C++ template mechanism is *lazy*. (2) Constant expressions – which can be evaluated at compilation time – must be evaluated at compilation time. Such constant expression appears on the left side of the enumeration initialization of `value` in class `Factorial`.

Lazyness is essential for writing template metaprograms. Let us consider the following example:

```
template <bool condition, class Then, class Else>
struct IF
{
    typedef Then RET;
};
```

```
template <class Then, class Else>
struct IF<false, Then, Else>
{
    typedef Else RET;
};

int main()
{
    IF< sizeof(int)<sizeof(long), long, int>::RET  i;
    cout << sizeof(i) << endl;
    return 0;
}
```

This seems a bit more cryptic than the factorial example. First let us draw up an inventory. We have a generic version of a template called `IF` and a *partial specialization* for it. It is partial, since only one, the leftmost argument has been specialized to the `false` boolean value. The first type parameter of the class `IF` is a (constant) value, the remaining arguments are type parameters.

When we instantiate the `IF` template, we provide a boolean expression as the first argument. In our example this is `sizeof(int)<sizeof(long)`. The expression is evaluated at compilation time. If this is `true`, then the generic template is instantiated, and hence the `typedef Then RET` is in effect. With the actual arguments this defines `RET` as `long`. However, when the expression is evaluated as false, we have a "better" specialization, and `typedef Else RET` means `RET` is defined as `int`. As a result, based on whether the size of `int` is smaller than the size of `long`, we define `i` as a variable of the widest type.

The construct is symmetric – it would be an equally working solution to define the generic function typedefing the `Else` branch, and writing a specialization for the `true` value as the first parameter.

The `IF` construct – the generic template and the specialization – works like a branching metaprogram. Having recursion and branching with pattern matching we have a complete programming language – executing programs at compilation time. In 1966 Bohm and Jacopini proved, that Turing machine implementation is equivalent to the existence of conditional and looping control structures in a programming language. C++ template metaprograming forms a Turing complete programming language executed at compilation time [5].

Templates be overloaded and the compiler has to choose the narrowest applicable template to instantiate. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [1].

Conditional statements, stopping recursions, and compile-time decisions are implemented with template specializations. Even with a relatively simple template and its specializations we are able to write useful metaprograms. The following metaprogram determines whether its two type arguments are equal.

```
template <class T1, class T2>
struct IsSameType
{
    static const bool value = false;
};

template <class T>
struct IsSameType<T,T>
{
    static const bool value = true;
};

bool b1 = IsSameType<long, int>::value;
bool b2 = IsSameType<int, int>::value;
```

In the general case, when `IsSameType` is called with two distinct types, the first, more general template is instantiated. The `IsSameType<long,int>::value` expression's value equals `false`. On the other hand, when the arguments refer to the same type, the compiler deduces that the partial specialization of the `IsSameType` template is required to instantiate. Thus the metaprogram's "return value" is `true`.

Similarly template metaprogram constructs for decisions on the class inheritance hierarchy could be implemented [2].

```
bool b = IsSuperClass<Bank,InternetBank>::value;
```

The compile-time decisions can directly affect the compilation itself. A *static assert* is capable of halting the compilation of a program at the point of the error's detection, thus we can avoid an incorrect program to come into being. At the same time, we aspire to create a static assert that contains some sensible error message, thus it is easier for the programmer to find the bug. The simplest way to execute this checks is by using a macro defined in [7], whose simplified vesrion is as follows:

```
template <bool> struct STATIC_ASSERT_FAILURE;
template<> struct STATIC_ASSERT_FAILURE<true>{};
template<int x> struct static_assert_test{};

#define STATIC_ASSERT(B, error) \
typedef static_assert_test< \
 sizeof(STATIC_ASSERT_FAILURE<(bool)(B), error>) \
> static_assert_typedef_;
```

If the expression `B` is true, the existing specialization of `STATIC_ASSERTION_FAILURE` is used as the `sizeof`'s argument. Otherwise the missing specialization for `false` causes a compile-time error. In the `error` argument a typename has to be provided that passes messages for the programmer:

```
struct CALLER_IS_NOT_DERIVED_FROM_BANK {};
STATIC_ASSERT(IsSuperClass<Bank,Caller>::value,
              CALLER_IS_NOT_DERIVED_FROM_BANK)
```

Static asserts are widely used for type checkings in C++ programs using templates [2]. Integration of domain-specific languages requires these techniques to detect invalid states in the domain space and to raise custom errors.

Today programmers write metaprograms for various reasons, like implementing *expression templates* [21], where we can replace runtime computations with compile-time activities to enhance runtime performance; *static interface checking*, which increases the ability of the compile-time to check the requirements against template parameters, i.e. they form constraints on template parameters [9, 11]; *active libraries* [19], acting dynamically during compile-time, making decisions and optimizations based on programming contexts. Other applications involve embedded domain specific languages as the AraRarat system [6] for typed safe SQL interface and `boost:xpressive` [29] for regular expressions.

3 DSL integration techniques

In this section we overview common patterns in technologies currently used for integration domain-specific languages.

3.1 External frameworks

In the following we discuss a few notable solution for language integration using external frameworks. The common feature of these approaches that they intent to use some language independent solution. Most cases the source code written in a specific syntax is transformed into a language-neutral internal representation. Transformation steps take place in this format. The result of the integration could be accessed by re-generation of the program in the desired syntax.

Stratego/XT The *Stratego/XT* developed in TU Delft is one of the most promising program transformation system using external toolset to integrate DSLs. The Stratego/XT metaprogram system [31, 22] is containing the Stratego language describing the program trasformations and the XT toolset, which executes the transformations and provides a framework for constructing stand-alone program transformation systems.

Sorce code written in arbitrary syntax can be parsed into *Annotated Term Format* (ATF), an internal representation form to bridge the differences between syntactical diversity. Step of transformations are executed on ATF before a pretty printer generates the output source code on the required language. Parsing and pretty printing is language dependent based on an external description, therefore the set of available language syntaxes are extensible. Some languages (like C++ and Java) are already supported.

The Stratego language is based on *strategic term rewriting*. Transformation definitions have two parts: rewriting rules and strategies. *Rewriting rules* describe basic transformation steps. Application of 0these rules are controlled using *strategies*. Rewriting rules can de defined in a language independent way in the form of the internal representation. This form, however, is often lenghty therefore a subsistem called *Metaborg* exists to describe the rewriting rules in the source language.

Intentional programming Current software development often uses high level, domain-specific notations in the design phase, but is almost always ends up implementing the program in some programming language. This last step is not only costly and error-prone, but causes recoding the software when some domain-specific content changes. The idea behind *intentional programming* [14, 30] is to separate the domain contents of the software from its implementations in a specific programming language and automatically regenerate the software as its domain contents change.

Intentional programming makes allow to express a porgram in a heterogeneus syntax, i.e. the code could be appear in the syntax a general purpose programming language while some of its part can be expressed in a domain notation when that is more expressive. Lazy evaluation strategies avoid unnecessary parsing-unparsing steps to improve efficiency.

Domain contents can be extended behind classical programming idioms. Comments, version control informations or even the full documentation could be integrated into the program and can be visualize on request.

3.2 Language extensions

Language extensions are attractive solutions for embedding domain-specific languages. They keep the most of host language syntax and therefore have zero impact on those code parts where DSL is not used. Keywords or even variables from the domain-specific language could be used without any quotation or syntactical marker.

However, there are several problems have to be solved when more then one domain language is used in the host language: keywords may collide, domain syntax can be ambigous, etc. Special parsing and context-aware scanning algorithms irequired in which the scanner uses contextual information to disambiguate lexical syntax [23]. Van Wyk and others shown the applicability of the extension mechanism.

Language extensions are fragile in many ways. They require either the modification of the compiler or an extensive set of translators or precompilers. Althoug for some languages like Java there exists a set of techniques and frameworks to make language extension less painfull, other languages – especially C++ – are very hard to extend when conformance to the existing language, stability, and efficiency of the generated code are all targeted.

3.3 Generativ approach

Expression Templates are an advanced technique that C++ library developers use to define embedded mini-languages that target specific problem domains. The technique has been used to create efficient and easy-to-use libraries for linear algebra as well as to define C++ parser generators with a readable syntax. But developing such a library involves writing an inordinate amount of unreadable and unmaintainable template code.

In the following we overview three application examples of expression templates to implementing domain-specific language integration.

AraRat The *AraRat* system targets one of the most important domain; it demonstrate the integration of relational algebra language into C++ [6]. Use of the system makes it possible to generate typesafe SQL queries and generating effective POD types for storing query results.

The system works in a two step way. In the first step a little external tool is used to discover the database schema and to generate a set of C++ types and operator overloads to reflect the schema information. In the host language, relational expressions are represented as C++ expressions using the overloaded operators. Template metaprogram techniques are used to check consistency of relational operations and generating result sets in effective way.

However its idea is impressive, the AraRat system has serious constraints. Its domain is restricted to relational algebra domain, moreover mainly for (typesafe) selections. The domain language has to follow valid C++ expression syntax.

Boost::Xpressive The *boost::xpressive* library is an advanced, object-oriented regular expression template library for C++ [29]. Regular expressions can be written as strings that are parsed at run-time, or as expression templates that are parsed at compile-time. Regular expressions can refer to each other and to themselves recursively, allowing you to build arbitrarily complicated grammars out of them.

Regular expressions are a paragon of domain-specific languages. They are used for a very special purpose – text manipulation – and have a specific (usually implementation-independent) syntax. Regular experssions are used mostly in some host language environment implemented as a library. Classical regular expression libraries (like `boost::regex`) are powerful and flexible; patterns are represented as strings which can be specified at runtime. However, that means that syntax errors are likewise not detected until runtime. Also, regular expressions are ill-suited to advanced text processing tasks such as matching balanced, nested tags.

`boost::xpressive` brings these two approaches seamlessly together and occupies a unique niche in the world of C++ text processing. With `xpressive`, user can represent regular expressions as strings, or can use it as C++ expression templates. In this case writing regular expressions are statically bound – hard-coded and syntax-checked by the compiler – and others are dynamically

bound and specified at runtime. These regular expressions can refer to each other recursively, matching patterns in strings that ordinary regular expressions cannot.

While `boost::xpressive` behaves similarly to our solution integrating a domain-specific language in compile time and performing syntax checks on it, its purpose is limited to a pre-defined domain: text manipulation.

Boost::Proto The *boost::proto* library advances one step forward from `xpressive` to provide a framework for building Domain Specific Embedded Languages in C++ [28]. It provides tools for constructing, type-checking, transforming and executing domain-specific languages expressible as expression templates. Proto provides data structure for representing the expression and a mechanism for giving additional behaviors and members to them.

Expression trees are built from an expression of the domain-specific language using operator overloads. Utilities for defining the grammar to which an expression must conform and an extensible set of mechanism for immediately executing and for tree transformations are also provided. The use of `boost::proto` to define the primitives of a domain-specific language radically simplifies the task of integrating a DSL.

The `boost::proto` library is one of the most general existing solution for embedding a domain-specific language into C++. Unfortunatelly, proto has its own restrictions. As the expression tree is built up with the help of operator overloads, the domain-specific language has to follow valid C++ expression syntax, i.e. keywords or variables have to be connected with overloaded C++ operators. This is a serious restriction when speaking on general purpose domain languages. In return no quotations should be applied to identify domain language code.

4 Type-safe printf: a motivating example

4.1 The problem

Though the `printf` function of the standard C library is efficient and easy to use, it's not type-safe, hence mistakes of the programmer may cause undefined behaviour 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 type-safe, but they have runtime and syntactical overhead.

In most cases the pattern of `printf` is a static string constant, its value is available at compile-time, thus the compiler could do type-checking and it could spot misuses of the function. `boost::mpl` (TODO cite) supports compile-time strings which could be used to represent the format string. A safe `printf` could be implemented as a template function taking the format string as a template argument and the values to be inserted into the format string as runtime arguments. This function could evaluate a template-metafunction at compile time, which could try to verify the number and type of the arguments and in

case this verification fails, it could emit a compilation error. On the other hand, if the verification succeeds it could call `printf` with the same arguments that the safe `printf` was called with. The template metafunction verifying the arguments cannot have a runtime overhead, only a compile time overhead. The body of the safe `printf` consists of a call to `printf`, which is likely to be in-lined, thus, using this safe `printf` has no runtime overhead compared to `printf` and has the same run-time performance.

Stroustrup wrote a type-safe `printf` using variadic template functions (TODO cite), which are part of the upcoming standard C++0x (TODO cite). 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 optimised 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 its `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 could be used by the type-safe implementation, is almost four times faster than the example at (TODO cite Stroustr) and more than two times faster than the optimised 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

The grammar of the format strings is complex and the validator metafunction has to parse them, thus the implementation of a type-safe `printf` requires a compile-time parser.

4.2 Embedded SQL

Any language can be embedded into C++ source code by using compile-time parsers. The embedded source code can be a compile-time string parsed by a metaprogram as part of the compilation process. For example SQL queries can be validated and the corresponding C++ classes can be built from them. For example

```
SELECT name, age FROM people WHERE department = "%s"
```

can be automatically transformed into

```
std::string exampleSqlQuery(const std::string& a1)
{
  std::ostringstream s;
  s
    << "SELECT name, age FROM people WHERE department = \""
    << sql_escape(a1)
    << "\"";
  return s.str();s
}
```

where the string returned by `exampleSqlQuery` is guaranteed to be a valid SQL query and it can provide safety against SQL injection as well.

The translators and validators presented in this chapter can be implemented as C++ template metafunctions, these extensions use the C++ standard and don't require any translator, thus they are easily portable.

5 Our solution

Our solution is based on the parser described in (TODO cite). The paper describes a Haskell parser generator library in detail. We implemented the same library in C++ template metaprogramming and the result is a compile-time parser generator library for C++. In this section we present the details of the translation.

5.1 Syntax for embedding source codes

The input of the parser is the text to parse represented as a string. In Haskell it's a string, which is a list of characters (TODO cite). In C++ template metaprogramming it's a list of characters as well (TODO cite). For example the string `Hello World!` in Haskell is

```
"Hello World!"
```

in a C++ template metaprogram it's

```
list_c<char, 'H','e','l','l','o',' ','W','o','r','l','d','!'>
```

`boost::mpl` has a tool for string definition which simplifies the declaration of compile time strings:

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

By using an external translator it can be simplified to

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

Support for user-defined literals has been proposed to be included in the upcoming C++ standard, C++0x. This proposal contains solution for the conversion of a string literal to the instantiation of a variadic template (TODO cite) function with the characters of the string as template arguments. With the combination of this, `decltype` (TODO cite) and the C++ pre-compiler the external translator could be simulated: we could get the same behaviour without using any external tool, thus we'd remain portable.

We present how we implemented those features of Haskell which are used by the library. Because of the size of the library we don't describe every part of the translation, we focus only on the key elements.

5.2 Algebraic types

Algebraic data types in Haskell have the following form:

```
data <name> [<type arguments>] =
  <constructor name> <constructor arguments> |
  <constructor name> <constructor arguments> |
  ...
```

We implement each constructor with a C++ template. The constructor arguments are the template arguments. For example the constructor `Div Expr Expr` is implemented as

```
template <class Expr1, class Expr2>
struct Div {};
```

We couldn't express Haskell types in C++ template metaprograms, the type of the arguments is always `class`. Algebraic data types and their arguments have no direct representation in C++ template metaprogramming, only the constructors are implemented.

In Haskell the constructors of algebraic data types act as functions to construct objects. We need to turn their C++ template metaprogramming implementations into functions as well. We can do it by turning them into nullary template metafunctions evaluating to themselves. For example the `Div` function could the enhanced the following way:

```
template <class Expr1, class Expr2>
struct Div
{
  typedef Div<Expr1, Expr2> type;
};
```

This template works with functions expecting a data-type and it works with functions expecting a nullary template metafunction as well. It behaves as expected in both situations.

As an example for translating algebraic data types we present our translation of Haskell's `Maybe`. In Haskell it's

```
Maybe a = Nothing | Just a
```

In C++ template metaprogramming it's

```
struct Nothing
{
  typedef Nothing type;
};

template <class a>
struct Just
{
  typedef Just<a> type;
};
```

5.3 Functions

Haskell builds on currying to represent functions, a function takes exactly one argument. Functions taking multiple arguments are implemented as functions taking 1 argument and returning other functions. For example a function taking 3 arguments is implemented as a function taking 1 argument and returning a function taking another argument and returning a function taking a third argument returning the value of the 3 argument function.

In our C++ template metaprogramming representation of the Haskell functions we didn't represent currying: we implemented Haskell functions as functions taking multiple arguments. Haskell functions have the form of

```
f :: <arg 1> -> <arg 2> -> <arg 3> -> ... -> <arg n> -> <result type>
```

which we implemented in C++ template metaprogramming with template metafunctions or template metafunction classes depending on how we wanted to use them:

```
template <class arg1, class arg2, ..., class argn>
struct f
```

```
  // ...
{};

// or

struct f
{
   template <class arg1, class arg2, ..., class argn>
   struct apply
     // ...
   {};
};
```

The result of the function is the value of the template metafunction or metafunction class. Functions are first-class citizens in Haskell, they can be passed around as data values. In C++ template metaprogramming we can do the same with template metafunction classes. Thus functions in the library that were arguments or values of other functions we implemented as template metafunction classes, not as simple template metafunctions. `boost::mpl` provides tools which can transform template metafunctions into template metafunction classes in cases we need to turn a template metafunction into a first-class citizen.

5.4 Parsers

Parsers are functions with the following signature:

```
type Parser a = String -> Maybe (a, String)
```

A parser takes the input string as its argument and returns a parsed object and the remaining part of the input when it accepts a prefix of the input string and returns `Nothing` when it rejects the input string. Note that the second element of the tuple is always a postfix of the input string.

A tuple with two elements can be implemented with a pair of classes. `boost::mpl` has a pair data structure which we can use. A parser is a function in the Haskell library, so it's a template metafunction in C++ template metaprogramming. Here is the definition of one of the basic parsers in Haskell:

```
char :: Parser Char
char (c:cs) = Just (c, cs)
char [] = Nothing
```

and in C++ template metaprogramming:

```
struct one_char
{
  template <class s>
  apply :
```

```
    eval_if<
      typename empty<s>::type,
      Nothing,
      Just<build_pair<front<s>, pop_front<s> > >
    >
  {};
};
```

Note that in C++ we had to call it `one_char` because `char` is a reserved word. `build_pair` is a helper metafunction taking nullary metafunctions as arguments and building a pair structure from them. We had to use `eval_if` instead of pattern matching. Even though C++ templates have excellent pattern matching support (TODO cite) when we're constructing code from the building blocks `boost::mpl` provides we can't use it. To be able to pass `one_char` to parser combinators, which are template metafunctions, we had to implement it as a template metafunction class.

Some parsers have arguments. The Haskell library builds on currying in Haskell: parsers taking arguments are functions with multiple arguments and the input string is always the last argument. By applying all arguments except the input string to these functions we get a parser: a function taking an input string as an argument and parsing it. For example `return` is a parser with an argument:

```
return :: a -> Parser a
return a cs = Just(a, cs)
```

Its C++ template metaprogramming implementation has to be a metafunction returning a parser, which is a metafunction:

```
template <class a>
struct return_
{
  struct type
  {
    template <class cs>
    struct apply : Just<pair<a, cs> > {};
  };
};
```

5.5 Parser combinators

Complex parsers are built by combining basic parsers. The Haskell library uses parser combinators which are parsers taking other parsers as arguments. For example the Haskell library defines an ? operator which is an infix operator: it's left argument is a parser, it's right argument is a predicate providing a boolean value for each result of the parser. We implemented it with a metafunction taking two metafunction classes (a parser and a predicate) as arguments and returning a parser:

```
template <class m, class p>
struct accept_when
{
  // This metafunction class is the value
  // of the accept_when metafunction
  struct type
  {
    template <class cs>
    struct apply :
      lazy_eval_if<
        equal_to<
          typename apply<m, cs>::type,
          Nothing
        >,
        nothing,
        lazy_eval_if<
          apply<p, just_value<apply<m, cs> > >,
          apply<m, cs>,
          nothing
        >
      >
    {};
  };
};
```

Note that the application of an argument to a function in Haskell, which is writing the function and the operand after each other, can be implemented using the `apply` metafunction in template metaprogramming.

This function can be used the same way it's used in the Haskell library. For example we can implement the `digit` function with it:

```
template <class cs>
struct digit : accept_when<one_char, isDigit>::type {};
```

`isDigit`'s C++ template metaprogramming implementation is straight forward but lengthy, we're not going to present it here.

5.6 Recursive functions

Recursive functions can be translated as well, template metafunctions can call themselves. We present our implementation of `iter` here as an example, other recursive functions can be translated similarly. The Haskell implementation of it is

```
iter :: Parser a -> Parser [a]
iter m = m # iter m >-> cons ! return []
```

while our translated implementation is

```
struct iter
{
  template <class m>
  struct apply :
    parser::one_of< // !
      parser::transform< // >->
        parser::sequence< // #
          m,
          boost::mpl::apply<parser::iter, m>
        >,
        parser::cons
      >,
      parser::return_<boost::mpl::list<> >
    >
  {};
};
```

Note that we combined the C++ template metaprogramming implementations of the operators the Haskell implementation uses the same way the Haskell code does it. In the example above we added the original names of the operators as comments to the functions.

The whole Haskell library can be translated to C++ template metaprograms following this approach, we don't present every step here. As a result we get the same functionality at compile time in C++ the Haskell library provides.

6 Evaluation

Embedded languages can be compiled as part of the C++ compilation process using template metaprograms. We have built a library for constructing these compile-time parsers. We present two grammars and compile-time parsers for them built using our library.

6.1 Hello^n world^n grammar

First we present how to build a parser for the following grammar:

```
S ::= hello S world | hello world
```

It accepts inputs such as `hello world`, `hello hello world world`, and so on. The number of `hello` and `world` words have to be equal. Here is a parser for it:

```
class Hello {};
class World {};
```

```
struct Extend
{
  template <class L>
  struct apply :
    boost::mpl::push_front<
      typename boost::mpl::push_back<L, World>::type,
      Hello
    > {};
};

typedef parser::token<parser::keyword<‘hello‘, Hello> > AcceptHello;
typedef parser::token<parser::keyword<‘world‘, World> > AcceptWorld;

struct S :
  parser::one_of<
    parser::always<
      parser::sequence<AcceptHello, AcceptWorld>,
      boost::mpl::deque<Hello, World>
    >,
    parser::transform<
      parser::keep_middle<AcceptHello, S, AcceptWorld>,
      Extend
    >
  >
{};

typedef parser::build_parser<S> HelloParser;
```

It constructs a compile-time sequence of `Hello` and `World` classes as a result of parsing the input string. For example the expression

```
HelloParser::apply<‘hello   hello world    world‘>::type
```

is reduced to the following at compile-time:

```
boost::mpl::deque<Hello, Hello, World, World>
```

But when we try compiling an invalid embedded code, such as

```
HelloParser::apply<‘hello hello world‘>::type
```

it generates an error and breaks the C++ compilation process.

We've measured the compilation speed of this parser. We were using g++ 4.4.1 on a Linux PC. We measured the compilation time for different number of `hello` and `world` words. Figure 1 shows the compilation times. The horizontal axis is the number of `hello` and `world` words in the embedded source code, the vertical axis is the number of seconds spent on compilation.

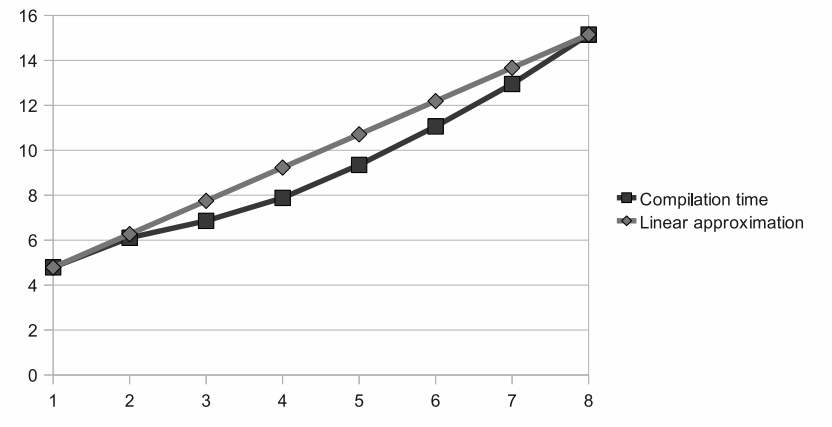

Fig. 1. Compilation time

6.2 Alternation at compile-time

The type constructed as the result of the parsing depends on the embededd code. We can easily construct a parser that takes a number as it's input and returns the `int` or `double` type, depending on which type of variable could store the number. Here is the parser:

```
typedef
  parser::keep_second<
    parser::any1<parser::digit>,
    parser::if_<
      parser::sequence<
        parser::lit_c<'.'>,
        parser::any<parser::digit>
      >,
      double,
      int
    >
  >
  S;

typedef parser::build_parser<S> Num;
```

And here is how it can be used:

```
Num::apply<‘13‘>::type // int
Num::apply<‘11.13‘>::type // double
```

7 Conclusion

Smoothless integration of domain-specific languages into a general purpose programming language is not an easy task. A domain specific language is intended to express the domain knowledge in the best possible way, thus its syntax may radically differ from the ones of the host language. A general case of language integration therefore could be solve only applying a full-featured parser infrastructure. Extrenal tools, and frameworks exists for the problem but they introduce unwanted dependency on third party tools. The best self-containing solution should use only standard language features and should use only a minimal set of external tools other then the compiler of the host language.

Our solution fulfills most of these requirements. We created a C++ template metaprogram library with the meaningful translation of a similar Haskell run-time tool, which implements a full-featured parser infrastructure. Domain-specific language code is presented for the parser as template arguments and evaluated during the compilation of the host code. The result of the parsing process is a set of C++ classes which could be used for further compile-time decisions in template metaprogramming environment. We presented a number of examples to show the usability of our library.

The library uses only standard C++ language features, thus our solution is highly portable. Current presentation has a minimal syntactical overhead which can be eliminated by a trivial transformation on the source code. This transformation later could be avoided as the next C++ standard will introduce user-defined custom literals which supports the straitforward presentation of the embedded domain-specific language syntax.

References

1. D. Abrahams, A. Gurtovoy, C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond, Addison-Wesley, Boston, 2004.
2. A. Alexandrescu, Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.
3. ANSI/ISO C++ Committee, Programming Languages – C++, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.
4. K. Czarnecki, U. W. Eisenecker, R. Glück, D. Vandevoorde, T. Veldhuizen, Generative Programming and Active Libraries, Springer-Verlag, 2000.
5. K. Czarnecki, U. W. Eisenecker, Generative Programming: Methods, Tools and Applications, Addison-Wesley, 2000.
6. Y. Gil, K. Lenz, Simple and Safe SQL queries with C++ templates, In: Charles Consela and Julia L. Lawall (eds), Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, pp.13-24.
7. B. Karlsson, Beyond the C++ Standard Library, An Introduction to Boost, Addison-Wesley, 2005.
8. D. R. Musser, A. A. Stepanov, Algorithm-oriented Generic Libraries, Software-practice and experience 27(7), 1994, pp.623-642.

9. B. McNamara, Y. Smaragdakis: Static interfaces in C++. In First Workshop on C++ Template Metaprogramming, October 2000
10. Z. Porkoláb, J. Mihalicza, Á. Sipos, Debugging C++ template metaprograms, In: Stan Jarzabek, Douglas C. Schmidt, Todd L. Veldhuizen (Eds.): Generative Programming and Component Engineering, 5th International Conference, GPCE 2006, Portland, Oregon, USA, October 22-26, 2006, Proceedings. ACM 2006, ISBN 1-59593-237-2, pp. 255-264.
11. J. Siek and A. Lumsdaine: Concept checking: Binding parametric polymorphism in C++, In First Workshop on C++ Template Metaprogramming, October 2000
12. J. Siek, A. Lumsdaine, Essential Language Support for Generic Programming, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.
13. D. Gregor, J. Järvi, J.G. Siek, G. Dos Reis, B. Stroustrup, A. Lumsdaine, Concepts: Linguistic Support for Generic Programming in C++, In Proceedings of the 2006 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'06), October 2006.
14. C. Simonyi, M. Christerson, S. Clifford, Intentional software, In Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, October 22-26, 2006, Portland, Oregon, USA, pp. 451–465.
15. B. Stroustrup, The C++ Programming Language Special Edition, Addison-Wesley, 2000.
16. G. Dos Reis, B. Stroustrup, Specifying C++ concepts, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.
17. E. Unruh, Prime number computation, ANSI X3J16-94-0075/ISO WG21-462.
18. D. Vandevoorde, N. M. Josuttis, C++ Templates: The Complete Guide, Addison-Wesley, 2003.
19. T. Veldhuizen, D. Gannon, 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'98). SIAM Press, 1998 pp. 21–23
20. T. Veldhuizen, Using C++ Template Metaprograms, C++ Report vol. 7, no. 4, 1995, pp. 36-43.
21. T. Veldhuizen, Expression Templates, C++ Report vol. 7, no. 5, 1995, pp. 26-31.
22. E. Visser, Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in StrategoXT-0.9. In C. Lengauer et al., editors, Domain-Specific Program Generation, vol. 3016 of Lecture Notes in Computer Science, pp. 216–238. Spinger-Verlag, June 2004.
23. E.R. Van Wyk, A.C. Schwerdfeger, Context-aware scanning for parsing extensible languages, Proceedings of the 6th international conference on Generative programming and component engineering, October 01-03, 2007, Salzburg, Austria, pp. 63-72.
24. M. Zalewski, A. P. Priesnitz, C. Ionescu, N. Botta, and S. Schupp, Multi-language library development: From Haskell type classes to C++ concepts, In MPOOL 2007 Ecoop workshp, 2007.
25. I. Zólyomi, Z. Porkoláb, Towards a template introspection library, LNCS Vol.3286 (2004), pp.266-282.
26. The boost lambda library. http://www.boost.org/doc/libs/1_39_0/doc/html/lambda.html

27. The boost metaprogram libraries.
http://www.boost.org/doc/libs/1_39_0/libs/mpl/doc/index.html
28. The boost proto library. http://www.boost.org/doc/libs/1_37_0/doc/html/proto.html
29. The boost xpressive regular library.
http://www.boost.org/doc/libs/1_38_0/doc/html/xpressive.html.
30. The Intentional Software. http://intentsoft.com/
31. The Stratego Program Transformation Language. http://strategoxt.org/