GENERATING TYPE-SAFE SCRIPT LANGUAGES FROM FUNCTIONAL APIS

Gábor Horváth, Gábor Kozár, Zalán Szűgyi
Department of Programming Languages and Compilers, Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/C., 1117 Budapest, Hungary, e-mail: xazax.hun, kozargabor, szugyi.zalan@gmail.com

ABSTRACT

It is often useful to expose an Application Programming Interface (API) as a scripting language when developing complex applications, especially when many teams are working on the same product. This allows for a solid separation of concerns and enables rapid development using the scripting language. However, to expose an API might involve huge amount of effort and a common problem with these scripting languages is their type-unsafe nature which can easily result in issues that are hard to debug.

Our solution is to generate an interpreter from a functional API utilizing C++ meta-programming techniques. The generated script language is type safe. Using our libraries it takes minimal effort to generate an interpreter from such an API. We also present a case study with a widely used EDSL. This method also turned out to be a more general solution to several other problems, including providing type-safe auto-completion support for the interpreted language or implementing a plug-in system that enables fast prototyping while remaining type-safe.

Keywords: *EDSL, generative programming, metaprogramming, C++, wrapper, functional*

1. INTRODUCTION

Compiled languages provide us with several advantages. These advantages include superior performance and static guarantees. However compilation can take a significant amount of time which degrades the productivity of the developers. Moreover, once the compilation is finished one needs to stop the old instance of the software and start a new one to utilize the new version. Sometimes initializing a software can be even more time consuming than the compilation. One possible solution would be to use dynamic libraries, and reload them on demand. However, this solution is platform-dependent and compilation still takes time.

Interpreted languages are usually not type-checked, and are therefore more error prone. On the other hand, a fast and iterative trial-and-error method of development works fairly well in an interpreted environment. Due to the lack of in-depth static analysis of the code, the vast majority of the errors are only detected during runtime. These languages are not capable of systems programming, in part because of their poor performance compared to compiled languages. However, since most of the platform-dependent details are usually abstracted away by the interpreter, such scripting languages tend to be portable.

It is not unusual to implement the performance critical part of a software in a compiled language and expose an Application Programming Interface (API) to an interpreted language. Unfortunately, this involves a lot of work, and the scripts require huge amounts of testing, because code on unexplored code paths is not verified by the interpreter. This paper presents a solution to automate the generation of an interpreter for a type-safe scripting language using the type information of the hosting compiled language. This method can save significant amount of time and effort in programming as well as debugging (due to type-checking).

First we present the problem and discuss the general methods that we are using to cope with it. Then we propose a possible implementation to solve the problem. After that there is a case study about our current solution, followed by discussing related works. At the end we summarize the paper.

2. GENERAL METHOD

An API or an Embedded Domain Specific Language (EDSL) in a statically typed language provides the user with some restrictions how it can be used thanks to the type system. These restrictions prevent some misuses of the API. When an API is exposed to a scripting language that is dynamically typed, the restrictions enforced by the type system are no longer protecting the user from such potential errors. One possible solution would be to modify the grammar of the scripting language to have similar restrictions as to what the type system provides. However, most of the script languages today do not give us the flexibility to modify their grammar. Another solution would be to use a statically typed script language and expose all of the type information. This interpreter of this language can be generated from the source code and the language tends to be domain specific.

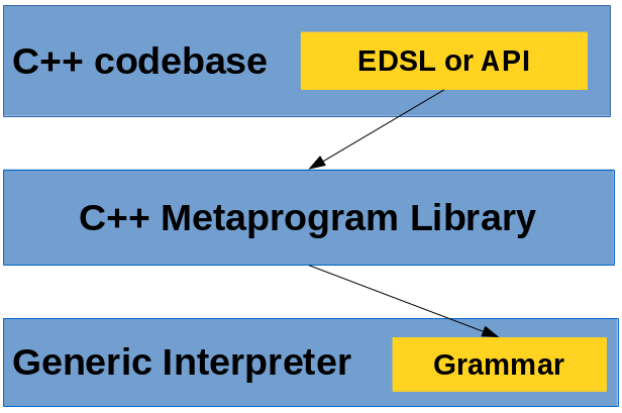

Fig. 1 General method

Both solutions require redundant efforts from the developers. Once the type information is available in the

host language, why should we spend time to duplicate it by describing those restrictions to the script language's interpreter? The main source of the problem is not the initial implementation, but the constant overhead of that every time the API changes, we have to update the bindings to the scripting language as well. This is a tedious and also error-prone process.

We created a metaprogram library which is able to use the limited capabilities of C++'s compile-time type introspection to derive grammar rules from the type information of the functions and functors making up the API 1. If we use this library to expose the API to a special scripting language, the cost of maintenance for the bindings will be significantly reduced. The library needs the enumeration of the functions and functors that are the part of the API. When a new function is introduced, or an old one is deleted, only this list has to be maintained. However, when the type signature of a function changes, no changes are required. Also, if a new conversion operator is introduced between two types, the library will still automatically generate the correct grammar.

It is also better for the user of the API to use such an interpreter, because the generated grammar makes some classes of errors impossible. For this reason, the user does not need to write tests for those kind of errors, saving considerable amount of time and effort.

3. IMPLEMENTATION

The introduced technique is implemented in C++ [7] and utilizes a large amount of template metaprogramming. Templates in C++ provide us with a Turing-complete language [9] that is evaluated at compile time. It is also possible to generate whole software components - such as interpreters - solely using metaprograms. This is called generative programming [8]. However, the proposed solution is not unique to C++; it is possible to implement it in any language that has similar support for metaprogramming, such as D [17]. An alternative approach to template metaprogramming is dynamic introspection (reflection), which allows a similar technique to be used in languages such as Java.

The basic idea is that the type information of an API is available to the compiler. Pattern matching with template specializations and relying on the Substitution Failure Is Not An Error (SFINAE) [10] principle of C++ allow us to encode type convertibility information in a matrix of boolean values that are known compile-time constants. These values will then determine how the API is allowed to be used, as the semantic analyzer of the scripting language is generated by a metaprogram based on those values. The parser of this language is not generated by a metaprogram in this solution, however it is possible to generate parsers in compile time [6] [15].

Our solution is able to cope with both functions and functors (objects behaving as functions), however creating instances of callable objects can be non-trivial. However, functors are advantageous for describing exposable APIs because they can be part of a class hierarchy, allowing us to take advantage of runtime polymorphism. A frequently used method to generate EDSLs in C++ is to use the generalized command pattern [1] with functors. Such EDSLs can be easily exposed to an interpreter using our library. Some APIs require relatively small amount of additional work. Potential source of problems are functions with several overloads. Those functions have to be disambiguated by static casts.

3.1. Strings in metaprograms

The first challenge is one that may not be immediately obvious: string handling. This is very much non-trivial in metaprograms, primarily due to the fact that (for technical reasons) a string literal is not (and cannot be) a valid template parameter. The source of the problem is that the semantic analyzer will work on function names - which are strings. The method for effectively work with compile-time strings was developed by Ábel Sinkovics [14]. We have slightly altered his approach to make it easier to create runtime strings from compile-time strings. Sinkovics' main idea was to create a macro, a metaprogram and a constexpr function to reduce a character string literal to a list of characters. His method generates some trailing zeros which can also be easily removed during compile-time.

```
#define DO(z, n, s) at(s, n),

#define _S(s) \
  BOOST_PP_REPEAT(STRING_MAX_LENGTH, \
                  DO, s)

template <int N>
constexpr char
  at(char const(&s)[N], int i)
{
  return i >= N ? '\0' : s[i];
}
```

He created a Boost MPL [11] vector from the characters, but we stored the characters in a template parameter pack instead. This made it possible to create a runtime string easily by utilizing the C++11 uniform initialization syntax [7]. The `MetaStringImpl` class is responsible for removing the trailing zeros, but its implementation is not relevant to this paper. The `GetRuntimeStr` method of the `Accumulator` class shows how easy it is to create runtime strings from this representation.

```
template <char... cs>
struct Accumulator {
  static std::string GetRuntimeStr() {
    return {cs...};
  }
};

template <typename T, char...>
struct MetaStringImpl;

template <char... cs>
struct MetaString {
  typedef typename
    MetaStringImpl<
```

```
      Accumulator<>,
      cs...>::result str;

  static std::string GetRuntimeStr() {
    return str::GetRuntimeString();
  }
};
```

3.2. Storing type information

The next obstacle is how to provide all the necessary information for the automata (the semantic analyzer) in a convenient way. We used preprocessor macros to make it easy for the user to register functions - this is unavoidable, given the lack of introspection in C++. The information involves the type signature of the callable object, the name of the function to identify it, and the type of an object to be instantiated. We use tuples to store all the necessary type information.

```
template<typename T, T* ptr>
struct FunctionPointer;

template<typename Ret,
         typename... Args,
         Ret (*f)(Args...)>
struct FunctionPointer<Ret(Args...),
                       f> {
  Ret operator()(Args... args) {
    return f(args...);
  }
};

#define FUNCTION(x)                          \
  std::tuple<decltype(x),                    \
    FunctionPointer<decltype(x), &x>,\
    MetaString<_S( #x )>>

#define FUNCTOR(x)                           \
  std::tuple<                                \
    decltype(&x::operator()), x, \
    MetaString<_S( #x )>>
```

Template metaprograms primarily work with types. In case of functors, it is easy to supply all of the information - however, for functions, it is important to box the function pointer into a functor type. The `FunctionPointer` class does exactly that, making it easy to work with function pointers in our metaprograms. The automata is generated from a list of those tuples. In fact, if we want to instantiate the boxed function pointers, we will face another issue: the C++ functions can not be overloaded on return types. To solve this issue we added a common base class to all of the function pointer boxes. The interface of the instantiation function will return a pointer to the common base, which points to the allocated derived box on the heap.

3.3. Generating the analyzer

What the automata actually does is generating several matrices: one for each possible argument. Currently, the number of matrices is a pre-configured constant and it is equal to the maximum arity (number of parameters) supported by the automata, but it is possible to deduce this number from the input and it is a target for future development.

$$M_i = \begin{array}{c} \\ A \\ B \\ \vdots \\ \vdots \end{array} \begin{array}{c} \begin{array}{cccc} A & B & \cdots & \cdots \end{array} \\ \begin{pmatrix} true & false & \cdots & true \\ false & false & \cdots & false \\ \vdots & \vdots & \ddots & \vdots \\ true & true & \cdots & true \end{pmatrix} \end{array}$$

Each matrix is indexed with function names. The nodes of the syntax tree which we want to analyze are representing function compositions such as $A(...,B(...),...)$, where the result of B is applied to the ith parameter of function A. Let's call the ith matrix M_i. The node of the syntax tree representing the call above is valid if and only if the value of $M_i[A,B]$ is *true*. This implies that $M_i[A,B]$ holds the information of whether the return type of B is implicitly convertible to the type of the ith formal parameter of A. To make it work, the metaprogram have to generate all the matrices at compile time, and generate the code to lookup the values in those matrices. Right now the lookup in the matrix involves an $O(N^2)$ complex function call chain where N is the number of functions in the API, which can result in stack overflow in moderately sized APIs - however, it is possible to reduce it to $O(1)$ long call chain if we use the information to build a hash maps instead of generating a recursive function chain.

To show how easy it is to work with this metaprogram, let's study a trivial example. Suppose we have the following tiny API with a dummy implementation:

```
struct A {};
struct B : A {};
struct C {};
struct E : B {};

A* func1(A*) { return 0; }
B* func2(A*, B*) { return 0; }
C* func3(A*) { return 0; }

struct D
{
  E* operator() (A*) { return 0; }
};
```

To create the automata we only need to enumerate the functions of our API with the corresponding macros:

```
typedef typename Automata<
    FUNCTION(func1),
    FUNCTION(func2),
    FUNCTION(func3),
    FUNCTOR(D)
  >::result GA;

std::set<std::string>
  expected {"func1", "func2", "D"};

auto tmp =
```

```
  GA::GetComposables("func1", 0);

std::set<std::string>
  result(tmp.begin(), tmp.end());

EXPECT_EQ(expected, result);
```

Now, if we want to, for example, know which functions can appear as the first argument of `func1`, we can use the `GetComposables` function of the generated automata. Here, because casting a pointer to a derived class to a base class pointer is a valid implicit conversion, the functions returning pointer to `A` or to a descendant of `A` are all valid functions. Those are `func1`, `func2` and the functor `D` in this example.

The template parameters of `Automata` are in fact tuples of three components describing a function: the type signature of the function, the object to instantiate, and the name of the function. `Automata` checks all the possible compositions and stores whether it is valid for all of the possible parameters and generates the lookup code. Unfortunately due to the verbosity and the noisiness of meta-programming the implementation of the lookup code generation is more than 200 lines of code. The lookup code operates solely on runtime strings. The current implementation involves a relatively time consuming lookup, but in the future it is possible to generate code that initializes a hash table for lookup instead of generating the lookup code directly. This future improvement will improve the performance of the interpreted scripting languages significantly.

4. CASE STUDY

We developed a tool, that executes queries on a C++ codebase. The tool is based on an EDSL that is already available in Clang.

Clang [4] is a modern C++ compiler, consisting of a set of a libraries each responsible for a separate task of the compiler. It was designed for tools to build on top of its libraries. One of these libraries provides an embedded domain-specific language [2] [3] [5] called AST matchers for expressing patterns in the abstract syntax tree of the subject code. It basically exposes a number of predicate functions (and functors) that can be composed together to form a pattern.

C++ is one of the most complex languages in existence. An AST built from C++ source code naturally has to reflect this complexity. This poses a problem for tool developers who want to use the matcher library: forming correct patterns is very much non-trivial. This often means that developer has to adapt a trial-and-error work flow, as he is trying to piece together a correct AST matcher expression. This is due to the fact that he has no easy way of testing the matcher, short of re-compiling the tool, re-parsing the subject code, and running the matchers against the built AST. This overall makes for an extremely inefficient work-flow.

To solve this, we are building a Read Eval Print Loop (REPL) interface that provides instant feedback [12]. The underlying engine is responsible for parsing the provided matcher expression into a primitive AST, and translating it into a matcher object. It is of course absolutely critical to ensure semantic correctness. It is also imperative that adding support for new AST matchers is very easy, as Clang is rapidly developing.

The tool we developed relied on an interpreted query language to run queries against C++ code bases - more specifically, their Abstract Syntax Tree (AST). The general design of the tool is shown on Figure 2. The input of the tool is an AST matcher expression as query string, that is, an expression describing a pattern to be searched for in the target AST, much like how regular expressions describe patterns over strings (text). The front-end is responsible for parsing the query, resulting in a syntax tree of the query expression itself. This task is fairly trivial, as the matcher expressions consist of only function calls and function composition, besides constant literals. Any lexical or syntactical errors are emitted during parsing. The next stage is semantic analysis: type-checking the function compositions. The verifier is generated by a metaprogram, based on the type information available to the compiler. This will detect type mismatches while building the compiled matcher expression from the query's syntax tree. The created matcher expression will be executed by the execution module, matching the pattern defined by the expression against the AST of the target C++ code base.

The module responsible for semantic analysis can be also used to provide auto-completion, in order to help the users to write such matcher expressions by listing all the possible type-safe subexpressions while editing a query. The topic of the present paper is the design and implementation of the metaprogram which generates this module.

5. RELATED WORK

There are several alternatives such as Simplified Wrapper and Interface Generator (SWIG) [16] which is an external tool that can generate such glue code for many existing scripting and compiled languages. However, the supported scripting languages did not meet our requirements. One of the problems is that it does not support statically typed scripting languages. The other issue is that it does not handle EDSLs written in C++ well.

Comprehending DSLs is not always an easy task [12]. However if a REPL is available for the developers to experiment with, it makes it much easier for new developers to explore and understand the DSL.

Using solely C++ for generating bindings for an interpreter is not a unique idea. The Boost.Python [18] library is taking this approach. However it is only limited to Python which was not suitable for our purposes.

6. SUMMARY

Generic programming is a very popular paradigm for solving various tasks. There are also some automated tools to generate bindings to a language. However, as it turned out, the C++ programming language is capable of generating type-safe bindings to a language without any external tools. This shows off the power of compile-time computations and the richness of information provided by static typing. We created a library that makes it straightforward

for the users to expose an EDSL or an API to a scripting language and significantly reduce the maintenance cost of the created bindings. The upcoming C++17 standard is expected to introduce compile-time reflection to the language. This will help us to write significantly more efficient code and greatly reduce the size of our code base. Moreover, it will open up new possibilities, such as automatically exposing whole classes to an object oriented type-safe scripting language. There are still several tasks waiting for us to achieve a completely robust solution, but the results so far look promising.

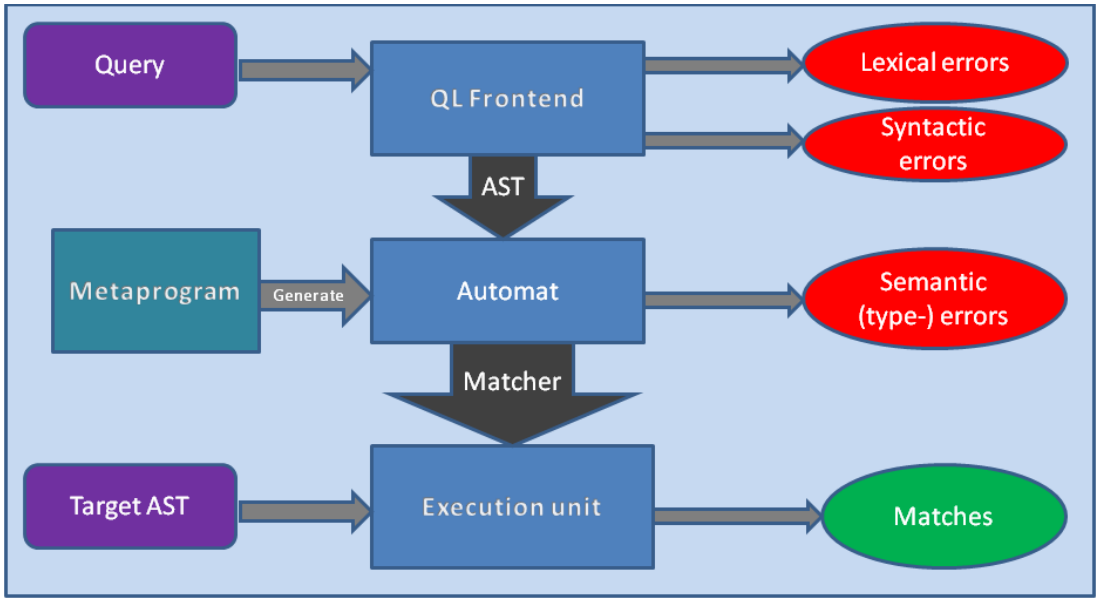

Fig. 2 Design of the Query Program

REFERENCES

[1] ALEXANDRESCU, A.: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison Wesley (2001).

[2] FOWLER, M.: *Domain-Specific Languages*, Addison-Wesley, 2010.

[3] HUDAK, P.: *Building domain-specific embedded languages*, ACM Comput. Surv. 28, 4 (Dec), 1996.

[4] LATTNER, C.: *LLVM and Clang: Next Generation Compiler Technology*, The BSD Conference, 2008.

[5] MERNIK,M. – HEERING, J. – SLOAN, A.: *When and how to develop domain-specific languages*, ACM Computing Surveys, 37(4) pp. 316 – 344, 2005.

[6] PORKOLÁB, Z. – SINKOVICS, Á.: *Domain-specific Language Integration with Compile-time Parser Generator Library*, In Proc. 9th international conference on Generative programming and component engineering (GPCE 2010). ACM, October 2010, pp. 137-146.

[7] STROUSTRUP, B.: *The C++ Programming Language*
Addison-Wesley Publishing Company, fourth edition, 2013.

[8] CZARNECKI, K. – EISENECKER, U. W.: *Generative Programming*, Addison-Wesley, 2000.

[9] VELDHUIZEN, T. L.: *C++ templates are Turing complete*, Technical report, Indiana University Computer Science, 2003.

[10] VANDERVOORDE, D. – JOSUTTIS, N. M.: *C++ Templates: The Complete Guide*, Addison-Wesley Professional, 2002.

[11] ABRAHAMS, D. – GURTOVOY, A.: *C++ Template Metaprogramming Concepts, Tools, and Techniques from Boost and Beyons* Pearson Education, Inc., 2005.

[12] VARANDA, M. J. – MERNIK. M. – Da CRUZ, D. – HENRIQUES, P. R.: *Program comprehension for domain-specific languages*, Comput. Sci. Inf. Syst., Dec. 2008, vol. 5, no. 2, pp. 1–17.

[13] Clang AST matcher library
`http://clang.llvm.org/docs/LibASTMatchers.html`
(11. June 2013).

[14] SINKOVICS, Á. – ABRAHAMS, D.: *Using strings in C++ template metaprograms*
`http://cpp-next.com/archive/2012/10/using-strings-in-c-template-metaprograms/`
(02. June 2013).

[15] Boost Spirit, a compile time parser generator
`http://boost-spirit.com/home/about-2/`
(08. January 2014).

[16] BEAZLEY, D. M.: *SWIG: an easy to use tool for integrating scripting languages with C and C++*

In Proc. 4th conference on USENIX Tcl/Tk Workshop, 1996 - Volume 4, pp. 15-15.

[17] ALEXANDRESCU, A.: *The D Programming Language*
Addison-Wesley Publishing Company, 2010.

[18] ABRAHAMS, D. – GROSSE-KUNSTLEVE, R. W.: *Building hybrid systems with Boost.Python*, C/C++ Users J. 21, 2003.

Received November 6, 2013, accepted December 22, 2013

BIOGRAPHY

Gábor Horváth was born on 26. 08. 1991. In 2011 he started the BSc in Computer Science at Eötvös Loránd University in Budapest. In 2012 he worked for Graphisoft on a software for energy performance analysis of buildings. Since 2013 he is a teaching assistant at the department of Programming Languages and Compilers of the Faculty of Informatics and working for Ericsson on a static analysis research project.

Gábor Kozár was born on 20. 02. 1992. In 2011 he started the BSc in Computer Science at Eötvös Loránd University in Budapest. In 2013 he worked for Ericsson on a static analysis research project, while working as a teaching assistant at the department of Programming Languages and Compilers of the Faculty of Informatics. He's spending the 2013/2014 spring term in Stockholm on a professional internship at Ericsson, developing code coverage analysis tools.

Zalán Szűgyi was born on 11. 02. 1980. In 2007 he graduated (MSc) at the department of Programming Languages and Compilers of the Faculty of Informatics at Eötvös Loránd University in Budapest. He is working on his PhD in the field of programming languages, C++. Since 2010 he is working assistant lecturer at department of Programming Languages and Compilers.