Generating a Fluent API with Syntax Checking from an LR Grammar

TETSURO YAMAZAKI, The University of Tokyo, Japan
TOMOKI NAKAMARU, The University of Tokyo, Japan
KAZUHIRO ICHIKAWA*, The University of Tokyo, Japan
SHIGERU CHIBA, The University of Tokyo, Japan

This paper proposes a fluent API generator for Scala, Haskell, and C++. It receives a grammar definition and generates a code skeleton of the library in the host programming language. The generated library is accessed through a chain of method calls; this style of API is called a fluent API. The library uses the host-language type checker to detect an invalid chain of method calls. Each method call is regarded as a lexical token in the embedded domain specific language implemented by that library. A sequence of the lexical tokens is checked and, if the sequence is not acceptable by the grammar, a type error is reported during compilation time. A contribution of this paper is to present an algorithm for generating the code-skeleton for a fluent API that reports a type error when a chain of method calls to the library does not match the given LR grammar. Our algorithm works in Scala, Haskell, and C++. To encode LR parsing, it uses the method/function overloading available in those languages. It does not need an advanced type system, or exponential compilation time or memory consumption. This paper also presents our implementation of the proposed generator.

CCS Concepts: • **Software and its engineering** → **Domain specific languages**; Source code generation.

Additional Key Words and Phrases: fluent API, library generation, metaprogramming, LR parsing

ACM Reference Format:

Tetsuro Yamazaki, Tomoki Nakamaru, Kazuhiro Ichikawa, and Shigeru Chiba. 2019. Generating a Fluent API with Syntax Checking from an LR Grammar. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 134 (October 2019), 24 pages. <https://doi.org/10.1145/3360560>

1 INTRODUCTION

A Fluent API [Fowler 2005] is a promising design pattern for embedded domain-specific languages (embedded DSLs, or EDSLs) [Hudak 1996]. An embedded DSL is a DSL embedded in a general-purpose language, called the host language. It is usually implemented by a library for the host language and hence it can be considered as a library with a language-like programming interface. A naive technique for implementing such embedded DSLs is the string-embedding style, where the whole DSL code is embedded as a string literal in the host language. It is passed to the library function, parsed, and interpreted. Another technique adopted more often is to construct a language-like interface, or the *syntax* of the DSL, by using programming primitives such as method calls

*Currently working for FIVE Inc.

Authors' addresses: Tetsuro Yamazaki, The University of Tokyo, Graduate School of Information Science and Technology, Japan, yamazaki@csg.ci.i.u-tokyo.ac.jp; Tomoki Nakamaru, The University of Tokyo, Graduate School of Information Science and Technology, Japan, nakamaru@csg.ci.i.u-tokyo.ac.jp; Kazuhiro Ichikawa, The University of Tokyo, Graduate School of Information Science and Technology, Japan, ichikawa@csg.ci.i.u-tokyo.ac.jp; Shigeru Chiba, The University of Tokyo, Graduate School of Information Science and Technology, Japan, chiba@acm.org.

This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2019 Copyright held by the owner/author(s).

2475-1421/2019/10-ART134

<https://doi.org/10.1145/3360560>

in the host language. A series of method or function calls to the library in the host language is regarded as the DSL code.

The fluent style of API is a design pattern for the programming interfaces of EDSLs when the interface consists of method calls to the library. A library with the fluent API is used through a chain of method calls to the library. Suppose we have a library for sending a SQL query to a database. The fluent API would enable the following method-call chain for using the library:

```
Query.select().from(BOOK).where(BOOK.eq(2019))
```

We designed the API to make the client code look like an SQL query. Each method call can be regarded as a lexical token in the EDSL implemented by that library. Hence the call chain above represents the following sequence of lexical tokens:

```
Query select from(BOOK) where(BOOK.eq(2019))
```

It is not easy work to implement an embedded DSL so that it will check the given method chain, which is a sequence of lexical tokens in the DSL, is valid or not. They only report an error at runtime although standalone (or external) DSLs statically check the validity and report a syntax error when they find an invalid sequence of lexical tokens. The lack of static checking is a drawback of embedded DSLs since DSLs can compel somewhat programmers through the DSL syntax to follow semantically correct use of their functionalities. An invalid chain of method calls to fluent API should be considered as semantically wrong usage of the library. Note that detecting wrong usage of libraries is a significant topic and thus a number of static analysis tools have been developed, for example, in the security domain [Krüger et al. 2018].

It is known that the validity of a method-call chain can be checked by using host-language types. In the example of the SQL library above, if the return type of the `from` method is a class providing only the `where` method, `from` is followed only by `where`. No other methods can follow `from`. We can declare a number of classes as the return types and thereby control valid chaining of method calls. To mitigate the costs of declaring a number of classes as return types, code-skeleton generators for fluent API, such as EriLex [Xu 2010], have been already proposed.

This paper presents a code-skeleton generator for fluent API that supports LR grammars in a widely-used programming language. As far as we know, existing generators support only grammars in classes smaller than LR. Although an algorithm has been proposed to support LR grammars within the ability of Java's type system [Gil and Levy 2016], it requires exponential time for compiling a method chain accessing a fluent API library. A difficulty of supporting LR grammars is that LR parsing uses a push-down automaton; it does not use a simple finite-state automaton, which can be used only for parsing regular grammars. Since Java's type system is known as being Turing-complete [Grigore 2017], a push-down automaton can be encoded in principle in Java-like languages by using a type system if time and space overheads are ignored. Our challenge was to discover the kind of push-down automata that can be encoded in a widely-used language, which has a type system with limited capability. Using an advanced type system such as Plaid's [Sunshine et al. 2011] is an easy solution but we do not take this approach because we aim at developing a programming tool for existing popular languages.

Our contributions are summarized as follows:

- We present an algorithm to translate LR automata into the Fluent language, which we designed to express single-state non-realtime deterministic push-down automata. These automata are not jump-stack automata [Courcelle 1977] but they can pop multiple elements at once.

Fig. 1. The graph drawn by the DOT program in Listing 1

- We also present an implementation scheme of the automata described in Fluent. The implementing languages are Scala, Haskell, or C++. Our scheme uses function/method overloading that considers type arguments. It is not available in Java.
- We developed a code-skeleton generator of fluent API based on our approach. It generates code skeletons in Scala, Haskell, or C++. The client code to access the generated fluent API was compiled in linear or quadratic time according to our experiments.

Since single-state deterministic push-down automata cannot perform LR parsing [Goldstine et al. 1981], our automata have the ability to pop multiple elements at once. To give the automata non-realtimeness, in other words, to allow ε -transitions, our implementation scheme needs function/method overloading (or type classes in Haskell). The ε -transitions are encoded by the methods that recursively call themselves. These methods are overloaded on their receiver types and return types with a type argument.

In the rest of this paper, Section 2 mentions the background of this work. Section 3 proposes our technique to encode LR parsing in our Fluent language. It also presents our implementation of Fluent in Scala. Section 4 presents the implementations in Haskell and C++. Section 5 shows our experiments and Section 6 compares our work to related work. Section 7 concludes this paper.

2 SYNTAX CHECKING OF FLUENT STYLE CODE

Since each problem domain has its own natural notation, the use of a domain specific language (DSL) is becoming widely accepted. For example, the DOT language [Gansner and North 2000] is a DSL designed for drawing a graph. The graph in Figure 1 is drawn by the program written in the DOT language shown in Listing 1. DOT provides simple and natural notation for describing a graph. However, since DOT is not general-purpose, we often want to write only part of the program in DOT and the rest of the program in a general-purpose language. In this approach, data exchanges between the two parts of the program tend to be awkward.

For better integration, we should use the DSL embedded in the general-purpose language. If the embedded DSL version of DOT is a library with the fluent API in the general-purpose language, it is straightforward to exchange data between the DSL and the general-purpose language, which is now the *host* language, since they are the same language. The DSL part of the program is constructed by chains of method calls to the library. The appearance of the DSL program is not far different from the original. Suppose the general-purpose language is Java. Listing 2 is the program written in the embedded DSL, which is equivalent to Listing 1. In Listing 2, method names and arguments can be regarded as lexical tokens in the DSL. We still see one-to-one correspondence between Listing 1 and 2 although Listing 2 is more verbose.

As the original standalone-DSL version of DOT applies static checking, for example, syntax checking to the source program, the embedded-DSL version of DOT can apply syntax checking at the compilation time by the host-language processor. Here, syntax checking is to check whether

Listing 1. A DOT program

```

digraph small_graph {
  A [shape = rectangle];
  B;
  C [shape = doublecircle];
  A -> B [style = dotted];
  {A B} -> C;
}

```

Listing 2. A fluent-style Java program

```

Graph fluentApiExample = beginDOT()
  .digraph("small_graph")
  .node("A").shape("rectangle")
  .node("B")
  .node("C").shape("doublecircle")
  .edge("A").to("B").style("dotted")
  .edge("A").and("B").to("C")
  .endDOT();

```

a chain of method calls is a valid sequence or not. Suppose that we have made a mistake when writing line 6 in Listing 2. Instead of this:

```
.edge("A").to("B").style("dotted")
```

we have wrongly written line 6 as follows:

```
.edge("A").style("dotted")
```

This lacks a call to the `to` method. The library could throw an exception at runtime when `style` is called without the call to `to`, but this is not convenient from the programmers' viewpoint. Statically detecting this error is more convenient. Another example is the confusion between `style` and `shape`. Line 6 might have been written as follows:

```
.edge("A").to("B").shape("dotted")
```

In the DOT language, the decoration of a node is specified by *shape* but that of an edge is by *style*. The library could also throw a runtime exception unless we change our *upright* notation to make *shape* and *style* interchangeable, but again, throwing a runtime exception is not convenient.

These invalid method chains can be detected by using the host-language type system. An advanced type system such as session types [Honda 1993] obviously can deal with this detection but much simpler type systems such as Java's can also do so to a certain degree. For example, a fluent API generator based on this idea has been proposed for Java [Nakamaru et al. 2017]. It generates a skeleton of library methods from a grammar definition written in the Backus-Naur form (BNF). The library methods report a type error when a chain of method calls is not valid. It regards each method call as a lexical token and reports an error when a sequence of the lexical tokens does not satisfy the given grammar. For example,

```

edges      ::= "edge" "to" | "edge" more_edges
more_edges ::= "and" "to" | "and" more_edges

```

this grammar definition in BNF specifies that `edge` is followed by either `to` or `and`. So, the following method chains are valid:

Listing 3. The return type of the edge method

```

public class AfterEdge {
    private Graph acc;
    private String srcNode;
    public AfterEdge(Graph a, String s){acc=a; srcNode=s;}
    public AfterTo to(String dstNode) {
        return new AfterTo(acc, srcNode, dstNode);
    }
    public AfterAnd and(String andNode) {
        List<String> srcNodes;
        srcNodes.add(srcNode);
        srcNodes.add(andNode);
        return new AfterAnd(acc, srcNodes);
    }
}

```

```

edge("A").to("B")
edge("A").and("B").to("C")

```

Note that the return type of the edge method is the receiver type of the following call to method such as to and and. Therefore, if the return type of edge accepts only to and and, the compiler can report a type error when the method call following edge("A") is neither a call to to or and. For example, when a call chain is edge("A").style("dotted"), the compiler will print an error message “cannot call style”.

The definition of the return type of edge would be as shown in Listing 3. Here, the return type is the AfterEdge class and it has only two methods to and and. It also has two fields acc and srcNode, which holds the accumulated results of the preceding method calls in a call chain. These values are passed to the next receiver object returned by the methods in AfterEdge.

Defining such classes as AfterEdge is tedious and error-prone without a skeleton-code generator. Writing them all by hand is awkward since a large number of classes are necessary when the grammar gets large. Note that a chained method such as edge may return an instance of a different class when edge’s receiver class is different. Therefore, most practical fluent-API libraries do not perform static syntax checking based on the idea above. In the case of our example, if the library were written by hand, all the methods edge, to, style, and and would return an instance of the Edge class or the Graph class. Because the instance would accept all those methods, an invalid chain of method calls would not cause a type error.

Supporting a larger set of grammars without an advanced type system is a challenge for the developers of fluent API generators. As far as we know, no fluent API generators have been proposed to support LR grammars for widely-used programming languages. Although Gil and Levy proposed an algorithm to encode LR parsing by Java’s types [Gil and Levy 2016], an API generator based on that algorithm has not been developed as far as we know. Their paper [Gil and Levy 2016] also reported that the compilation of a method chain based on their algorithm was extremely slow. Exploiting an advanced type system, such as Plaid’s [Kouzapas et al. 2018; Sunshine et al. 2011] and context-free session types [Thiemann and Vasconcelos 2016], might be another option. We did not choose the approach of extending these type systems either since we aimed at developing a programming tool for existing popular languages. Modifying the languages or waiting until the language supports our new type system was not acceptable.

Listing 4. an invalid if-else statement

```

1  message = begin()
2    .if_(n % 3 == 0)
3      .then_().if_(n % 5 == 0)
4        .then_().return_("fizzbuzz")
5        .else_().return_("fizz")
6      .else_().return_("buzz")
7    .else_().return_("oops!")    // no matching if_
8  .end()

```

LR grammars are typical grammars for programming languages. For example, it is known that a grammar is not LL(1) if it includes an if statement where an else clause is optional and the if statement may be nested in another statement. Therefore, the existing fluent-API generators, which do not support a parsing algorithm such as LALR(1), cannot generate a library skeleton for such a typical grammar. Otherwise, they generate a library skeleton that does not cause a compilation error even when the library is used as in Listing 4. Note that the last `else_` in line 7 does not correspond to any `if_`. Hence the method-call chain in Listing 4 is not syntactically valid. To detect this error, the library needs to track a stack in order to balance `if_`, `then_`, and `else_`.

3 OUR FLUENT API GENERATOR

This section proposes an algorithm that translates an LR automaton to the skeleton of a fluent-style library in Scala. An LR automaton is an automaton expressing LR parsing. The Scala compiler reports a type error when a chain of method calls to the generated library is not accepted by the given LR automaton.

Our algorithm encodes an LR automaton by method overloading on the receiver type. The encoding is fairly straightforward except the return types. Our trick is to leave the return type of the overloaded method unspecified and let the Scala compiler infer it depending on the type argument given to the receiver type. For formally describing our algorithm, we first translate the given LR automaton to a program written in our pseudo language named *Fluent*. In *Fluent*, the return type of a method is not explicitly specified; it is expected to be inferred. Then we translate the *Fluent* program into a Scala program. Since the return type cannot be omitted in Scala, we present our technique for expressing an unspecified return type in Scala. We also show the translation from *Fluent* to other languages C++ and Haskell later in Section 4. This paper does not show how to construct an LR automaton from an LR grammar. We assume that the LR automaton has been already constructed in the parsing algorithm mentioned in the literature such as [Heilbrunner 1981].

3.1 LR Automaton

We use following symbols to express an LR automaton.

- Σ , to denote a set of input tokens used in the grammar.
- N , to denote a set of non-terminal symbols used in the grammar.
- R , to denote a grammar represented as a set of derivation rules.
- Q , to denote a set of stack elements.
- $q_{init} \in Q$, to denote an initial stack element.
- δ_{action} , to denote an action table represented as a partial mapping: $Q \times (\Sigma \cup \{\$, \}) \mapsto \{\text{shift, accept}\} \cup (\{\text{reduce}\} \times R)$.

Listing 5. The grammar of *oops*

```
oops ::= os "ps"
os   ::= "o" os | ε
```

Table 1. The action table for *oops*

Stack top	Input token		
	"o"	"ps"	\$
q_1	<i>shift</i>	$\langle \text{reduce, os} \rightarrow \text{eps} \rangle$	—
q_2	—	—	<i>accept</i>
q_3	—	<i>shift</i>	—
q_4	—	—	$\langle \text{reduce, oops} \rightarrow \text{os "ps"} \rangle$
q_5	<i>shift</i>	$\langle \text{reduce, os} \rightarrow \text{eps} \rangle$	—
q_6	—	$\langle \text{reduce, os} \rightarrow \text{"o" os} \rangle$	—

- δ_{goto} , to denote a goto table, also represented as a partial mapping: $Q \times (\Sigma \cup \{\$ \} \cup N) \mapsto Q$

This automaton is a single-state deterministic push-down automaton. Each derivation rule in R has the form of " $nt \rightarrow s_1 s_2 s_3 \dots s_n$ " where $nt \in N$ and $s_i \in \Sigma \cup N$ ($1 \leq i \leq n$). For example, `dot -> "digraph" stmt-list` is a derivation rule. The right-hand side must be a sequence of either input token or non-terminal symbol. Alternatives are denoted as multiple rules sharing the left-hand side. For example, the pair of edge `-> "edge"` to and edge `-> "edge" more_edges` is equivalent to edge `:= "edge" to | "edge" more_edges` in Backus-Naur form. We require a goto table to return different stack elements when the second argument s is different. We can assume this without loss of generality since it is preserved for the LR automata constructed by a practical parsing algorithm such as SLR, LALR(1), and LR(1).

With this notation, we can write the semantics of an LR automaton by the relation $\vdash^*$, which is the reflective transitive closure of the following relation $\vdash$ over $(Q * \cup \{accept\}) \times \Sigma^*$:

$$\langle q \overline{qs}, t \overline{ts} \rangle \vdash \begin{cases} \langle \delta_{goto}(q, t) q \overline{qs}, \overline{ts} \rangle & (\delta_{action}(q, t) = \text{shift}) \\ \langle \text{accept}, \overline{ts} \rangle & (\delta_{action}(q, t) = \text{accept}) \\ \langle \text{REDUCE}(q \overline{qs}, nt, s \overline{ss}), t \overline{ts} \rangle & (\delta_{action}(q, t) = \langle \text{reduce}, nt \rightarrow s \overline{ss} \rangle) \end{cases}$$

We use an overline to denote a sequence and write, for example, $\overline{xs}$ as short hand for $x_1, x_2, \dots, x_n$. We use ϵ to denote an empty sequence. *REDUCE* is the following function.

$$\begin{aligned} \text{REDUCE}(q \overline{qs}, nt, s \overline{ss}) &= \text{REDUCE}(\overline{qs}, nt, \overline{ss}) \\ \text{REDUCE}(q \overline{qs}, nt, \epsilon) &= \delta_{goto}(q, nt) q \overline{qs} \end{aligned}$$

An LR automaton M_{LR} accepts a sequence of input tokens $\overline{ts}$ iff $\langle q_{init}, \overline{ts} \$ \rangle \vdash^* \langle \text{accept}, \epsilon \rangle$. $\$$ is the end-of-input symbol.

Table 2. The goto table for *oops*

Stack top	Shift			Reduce	
	"o"	"ps"	\$	oops	os
q_1	q_5	—	—	q_2	q_3
q_2	—	—	—	—	—
q_3	—	q_4	—	—	—
q_4	—	—	—	—	—
q_5	q_5	—	—	—	q_6
q_6	—	—	—	—	—

For example, the *oops* language (its grammar is shown in Listing 5) is converted to an LR automaton $M_{oops} = \langle \Sigma, N, R, Q, q_{init}, \delta_{action}, \delta_{goto} \rangle$ such that:

Σ	=	{ "o", "ps" },
N	=	{ <i>oops</i> , <i>os</i> },
R	=	{ <i>oops</i> $\rightarrow$ <i>os</i> "ps", <i>os</i> $\rightarrow$ "o" <i>os</i> , <i>os</i> $\rightarrow$ ϵ },
Q	=	{ $q_1, q_2, q_3, q_4, q_5, q_6$ },
q_{init}	=	q_1 ,
δ_{action}	=	The action table shown in Table 1,
δ_{goto}	=	The goto table shown in Table 2.

3.2 The Fluent Language

We designed a simple pseudo language *Fluent* to describe the skeleton of a fluent API with type checking. In *Fluent*, we can define a method. Each method can make a new instance and, if needed, call a method on it. The syntax of *Fluent* is given as follows:

M	$\rightarrow$	def <X> T.m() = e	method declarations
T	$\rightarrow$	C C[T] X	types
e	$\rightarrow$	new T() new T().m()	expressions

C, X, m are metavariables; C ranges over class names; X ranges over type-parameter names; and m ranges over method names. M is a method declaration, consisting of a type-parameter, its receiver type, its method name, and a method body. C[T] is the type C with the type argument T. For example, the following code declares a `toString` method in the `Array` class with any type argument given by T.

```
def <T> Array[T].toString() = new String()
```

This method returns a new instance of `String`. In *Fluent*, a receiver class such as `Array` is implicitly declared.

Note that the return type of a method is not explicitly specified; it is automatically inferred from the method body. The inference is not always easy. For example,

```
def <T> Z[X[T]].m() = new X[T]()
def <T> Z[Y[T]].m() = new Y[T]()
def <T> Z[Z[T]].m() = new Z[T]().m()
```

Here, X, Y, and Z are concrete types and the method m is overloaded by the three declarations. Since the last declaration of m causes a recursive call, the return type of the last declaration `Z[Z[T]].m()` depends on the type parameter T. If T is `X[Unit]` or `Y[Unit]`, the return type is `X[Unit]` or `Y[Unit]`, respectively. If T is `Z[U]`, the return type depends on the form of the type argument U. If U

is $Z[V]$, the return type further depends on the form of V . This causes infinite regression. Therefore, explicitly specifying the return type for each declaration is not feasible for this example.

This difficulty in the return-type inference appears in our encoding scheme of an LR automaton. The aim of *Fluent* is to deal with this difficulty in a separate phase of the translation. We will revisit this later in 3.4.

3.3 Translate LR Automata to Fluent

We first translate an LR automaton into *Fluent* by generating one or more method definitions for each pair of $q \in Q$ and $\sigma \in \Sigma \cup \{\$\}$ if $\delta_{action}(q, \sigma)$ exists. We encode each $\sigma \in \Sigma \cup \{\$\}$ to a method name and each $\overline{qs} \in Q^*$ to a type. We express a sequence of stack elements by a nested type-argument. For example, $q_1[q_2[\dots[q_n[\text{Bottom}]]\dots]]$ corresponds to $q_1 q_2 \dots q_n$.

The action table δ_{action} and the goto table δ_{goto} are directly encoded into *Fluent* methods. Let q is the top element of the current stack and σ is an input token. When $\delta_{action}(q, \sigma) = \text{shift}$, the LR automaton pushes a new stack element $q' = \delta_{goto}(q, \sigma)$ onto the stack and consumes the input token σ . We encode this behavior to a method definition such that:

```
def <T> q[T].σ() = new q'[q[T]]()
```

This code declares the σ method in $q[T]$. Calling this method returns an instance of $q'[q[T]]$. $q[T]$ corresponds to the stack containing q as its top element. $q'[q[T]]$ corresponds to the stack after q' is pushed onto $q[T]$. Thus, we can call the σ method anywhere the previous method call in the chain returns a stack containing q as its top element. The method pushes q' and returns the resulting stack.

Next, when $\delta_{action}(q, \sigma) = \text{accept}$, the LR automaton changes its state to *accept*. We encode this into the following method declaration:

```
def <T> q[T].σ() = new Accept()
```

This code also declares the σ method in $q[T]$ but the declared method returns an instance of *Accept*. *Accept* is the type denoting *accept*. No methods are declared in *Accept*. Hence the call to this method has to be at the end of the chain. Otherwise, the chain causes a type error.

When $\delta_{action}(q, \sigma) = \langle \text{reduce}, nt \rightarrow s_1 s_2 \dots s_n \rangle$, the LR automaton performs a reduce action; it first pops n elements from the stack and then pushes a new element $q' = \delta_{goto}(q_{n+1}, nt)$ onto the stack. Here, n denotes the length of the derivation rule $nt \rightarrow s_1 s_2 \dots s_n$ and q_{n+1} denotes the $(n+1)$ -th element from the top of the stack, which will become the stack top element after popping the n elements. Note that the reduce action does not consume the input token σ . The next action is thus selected by the combination of q' and σ . To encode this behavior, we declare several methods in the following form:

```
def <T> q_1[q_2[\dots[q_n[q_{n+1}[T]]]\dots]].σ() = new q'[q_{n+1}[T]]().σ()
```

This code declares the σ method in $q_1[q_2[\dots[q_n[q_{n+1}[T]]]\dots]]$. Note that q_1 is the stack top element and it is an alias of q . Its body makes a new instance of $q'[q_{n+1}[T]]$ and recursively calls the σ method on this new instance. This recursive call to σ corresponds to the fact that the reduce action does not consume the input token. Like the *ungetc* function in the C language, the method pushes σ back to the input stream.

We declare multiple methods in the form above for every possible sequence of $q_1, q_2, \dots, q_n, q_{n+1}$. We can enumerate all the possible sequences in finite time since the number of the sequences is at most $|Q|^{n+1}$. We can make the enumeration more efficient by attending to the fact that $\forall i (1 \leq i \leq n), \delta_{goto}(q_{i+1}, s_{n-i+1}) = q_i$. The number of valid sequences can be exponential but we could not observe such a case in our experiment. Furthermore, Chomsky normal form limits the upper bound to polynomial since the exponent (n) is the length of the derivation rule.

3.4 Translate Fluent to Scala

We finally translate the generated *Fluent* program to a fluent-style Scala library. For brevity, we below omit the semantic actions of each method, for example, constructing an abstract syntax tree. The semantic action is implemented by a callback function attached to the `invoke` method for the implicit value that we show below for performing a reduce action.

An issue here is how to infer the return type of each method. Recall that the return type is not explicitly specified in *Fluent*. We also mentioned that the return-type inference may cause infinite regression. To address this problem in Scala, we use type classes. An instance of a type class, which is in Scala the concrete implementation of a method for particular type arguments, is generated on demand only when it is necessary for compiling the given source code. We translate the program so that the compiler will generate only the type-class instances that are necessary for compiling the given method-call chain. Only the finite number of instances will be generated.

First, we declare the following trait in Scala, which is used as a type class:

```
trait MethodExists[Recv, Sigma, Ret] {
  def invoke(receiver: Recv): Ret
}
```

This Scala code declares the `MethodExists` trait. `MethodExists` takes three type parameters `Recv`, `Sigma`, and `Ret`. They denote a receiver type, a method name, and a return type, respectively. We will declare instances of the type class represented by `MethodExists` later if and only if there exists a *Fluent* method that the receiver type is `Recv` and the method name is `Sigma`. To treat a method name as a type, the type corresponding to each σ in Σ is assumed to be declared.

Then we declare the following receiver class for each q in Q :

```
class q[T] {
  def  $\sigma$ [To](implicit t: MethodExists[q[T],  $\sigma$ , To]): To =
    t.invoke(this)
  /* ... define the  $\sigma$  method for every  $\sigma$  in Fluent ... */
  def $[To](implicit t: MethodExists[q[T], $, To]): To =
    t.invoke(this)
}
```

This Scala code declares class q . Each q takes just one type parameter T , which denotes the rest of the stack elements below q . The class q contains the methods corresponding to all the σ methods in q in *Fluent*. It also contains the method for the end-of-input token $\$$. These methods take one implicit parameter of the type `MethodExists[q[T],  $\sigma$ , To]`. In Scala, when a method is called but its implicit parameter is not given, the compiler finds the implicit value of that parameter type and passes that value to the method. A type error is reported when the compiler cannot find the value or it finds multiple values. Our trick is to define an implicit value for `MethodExists[q[T],  $\sigma$ , To]` only when the program in *Fluent* contains a method σ in $q[T]$. To is the return type of σ . The σ method calls `invoke` on the implicit parameter with the receiver object `this`.

We finally declare implicit values for each *Fluent* method declaration generated. We show two forms of declaration.

When a method in *Fluent* is the form of `def <T>  $T_1$ . $\sigma$ () = new  $T_2$ ()`, where T is a type parameter referred to in T_1 and T_2 , it expresses a shift action. We translate the method in *Fluent* into the following declaration in Scala:

```
implicit def  $\alpha$ [T]: MethodExists[ $T_1$ ,  $\sigma$ ,  $T_2$ ] =
  new MethodExists[ $T_1$ ,  $\sigma$ ,  $T_2$ ] {
    def invoke(receiver:  $T_1$ ):  $T_2$  = new  $T_2$ ()
  }
```

```
}

```

This Scala code declares an implicit value α of type `MethodExists[ $T_1, \sigma, T_2$ ]`. α is a uniquely generated name for each implicit value declaration. The return type is obvious in this case; thus the translation is straightforward.

When a method in *Fluent* is the form of `def <T>  $T_1$ . $\sigma_1$ () = new  $T_2$ () .  $\sigma_2$ ()`, it expresses a reduce action. We translate the method in *Fluent* into the following declaration in Scala:

```
implicit def  $\alpha$ [T, Ret](implicit t: MethodExists[ $T_2, \sigma_2, Ret$ ])
  : MethodExists[ $T_1, \sigma_1, Ret$ ] =
  new MethodExists[ $T_1, \sigma_1, Ret$ ] {
    def invoke(receiver:  $T_1$ ): Ret = t.invoke(new  $T_2$ ())
  }
```

This Scala code also declares an implicit value α of type `MethodExists[ $T_1, \sigma_1, Ret$ ]`. This α takes another implicit parameter t of type `MethodExists[ $T_2, \sigma_2, Ret$ ]`. It requires that there exists a *Fluent* method σ_2 in T_2 that returns an instance of `Ret`. Since the implicit parameter ensures that the return type of the nested method call on t is `Ret`, we know the return type of `invoke` taking T_1 is also `Ret`. Hence the type of α is `MethodExists[ $T_1, \sigma_1, Ret$ ]`.

We also need to declare the `begin` function, which starts a method chain. The return type of `begin` is `qinit[Unit]`, which corresponds to the initial stack. Here, `Unit` expresses an empty stack. We declare `begin` as follows:

```
def begin(): qinit[Unit] = new qinit[Unit]()
```

3.5 Literals

Since every lexical token is encoded into a method name, our approach does not directly support number literals or string literals. We express such literal values by passing a runtime value as an argument to a method in a method-call chain. The examples are found in Listing 4. The arguments passed to `if_` and `return_` are regarded as literal values. For brevity, we do not present details of how we extend our translation to support literal values. As we later show in Listing 8, the input grammar to the API generator would specify some methods take a parameter. For example, `digraph` method takes a `String` object as its parameter. The translation algorithm is not largely extended. The σ method in *Fluent* is extended to take a parameter according to the input grammar and the corresponding Scala methods are also extended.

4 TRANSLATION TO HASKELL AND C++

In the previous section, we presented how we can translate LR automata to fluent-style Scala libraries. We next present the translation to Haskell and C++. As in Scala, we first translate LR automata to *Fluent* and then translate it to the library code in those languages. We below present the translation from *Fluent* to Haskell and C++. For brevity, we below omit the semantic actions of each library method as we did in Scala.

4.1 Haskell

In Scala, we used the programming idiom for type classes. Since Haskell natively supports type classes, we take the same approach. Since we need multi-parameter type classes, we use four GHC extensions: *MultiParamTypeClasses*, *FunctionalDependencies*, *FlexibleInstances*, and *UndecidableInstances*.

Since Haskell is not an object-oriented language, we cannot directly write a method chain in Haskell. We design a similar fluent API library by using a pipe operator `|>`. The pipe operator is

popular in several functional programming languages including F#, Ocaml, and Elixir. The definition of the pipe operator `|>` is as follows:

```
infixl 1 |>
(|>) :: a -> (a -> b) -> b
x |> f = f x
```

The pipe operator is an infix operator that takes a value for the left operand and a function for the right operand. It applies the function to the value and it is left-associative. This operator allows us to write the following chain for the fluent style DSL for SQL queries in Section 1.

```
begin |> query |> select |> from BOOK |> where_ (BOOK `eq` 2019) |> end
```

This is somewhat verbose but we can say it is sufficiently fluent. Here, `begin`, `query`, `select`, `from`, `where_`, and `end` are functions.

As in Scala, a stack is expressed by a nested type-parameter. For example, $q_1(q_2(q_3()))$ expresses the stack containing three elements q_1 , q_2 , and q_3 . The unit type expresses an empty stack. Hence, we declare the following data types for each q in Q to express stacks:

```
data q t = q t
```

Here, t is a type parameter to q . A value of q is constructed by a constructor named q , which takes a value of type t as an argument. The parameter expresses the stack excluding the stack-top element.

We then translate each method in *Fluent* to a function in Haskell. The receiver object and its type in *Fluent* is translated into the (first) function parameter and its type in Haskell. So, the functions in Haskell are overloaded on the parameter type. In Haskell, function overloading is implemented by type classes. For each method name σ in *Fluent*, we define the following σ method in the type class `MethodExists_ $\sigma$` in Haskell:

```
class MethodExists_ $\sigma$  recv ret | recv -> ret where
   $\sigma$  :: recv -> ret
```

σ ranges over the method names defined in *Fluent*. The type class `MethodExists_ $\sigma$` takes two type parameters `recv` and `ret`. The σ method takes a value of type `recv` as an argument and returns a value of type `ret`. The functional dependency `recv -> ret` after `|` specifies that `ret` is uniquely determined from `recv`.

Each implementation of the σ method in *Fluent* is translated into an instance of `MethodExists_ $\sigma$` . It is only available for a particular pair of `recv` and `ret`.

When a method in *Fluent* is the form of `def <t>  $T_1$ . $\sigma$ () = new  $T_2$ ()`, we define the following instance of type class:

```
instance MethodExists_ $\sigma$  Hs( $T_1$ ) Hs( $T_2$ ) where
   $\sigma$  Hs( $T_1$ ) = Hs( $T_2$ )
```

This instance of `MethodExists_ $\sigma$` supplies the implementation of σ effective only when the type of the parameter to σ is $Hs(T_1)$. It returns a value of type $Hs(T_2)$. Here, Hs is a metafunction that converts a type in *Fluent* to the corresponding type in Haskell. For example, $Hs(q_1[q_2[T]]) = (q_1(q_2\ t))$ although the outermost parenthesis may be redundant. Hs replaces T with t since capitalized T is not considered as a type parameter in Haskell. We also use $Hs(T_2)$ as a value in Haskell since it has the same notation as its type.

When a method in *Fluent* is the form of `def <T>  $T_1$ . $\sigma_1$ () = new  $T_2$ () .  $\sigma_2$ ()`, we define the following instance of type class:

```
instance (MethodExists_ $\sigma_2$  Hs( $T_2$ ) ret) =>
  MethodExists_ $\sigma_1$  Hs( $T_1$ ) ret where
```

$$\sigma_1 \text{ Hs}(T_1) = \sigma_2 \text{ Hs}(T_2)$$

This implementation of σ_1 constructs a value of type $\text{Hs}(T_2)$ from the given argument and then calls σ_2 with that value. The left-hand side of $\Rightarrow$ is a context. It requires that the instance $\text{MethodExists}_{\sigma_2} \text{ Hs}(T_2)$ ret exists and thus we can call the σ_2 function taking an argument of type $\text{Hs}(T_2)$.

Finally, we also define the begin function:

```
begin :: qinit ()
begin = qinit ()
```

The begin function returns a value of type $q_{\text{init}}()$, which corresponds to an initial stack.

4.2 C++

Although C++ has not supported type classes yet, we can take a similar approach by using function templates. We below omit the code for memory management although the implementation used for the experiment in Section 5 exploits reference-counting garbage collection.

First, we declare the q class for each q in Q , which expresses a stack containing q as the top element.

```
template<typename T>
class q {
public:
    auto  $\sigma$ () { return invoke_ $\sigma$ (this); }
    // ... declare a function for every  $\sigma$  method in  $q$  in Fluent
};
```

As in Scala, the template parameter T expresses the stack excluding the top element. The class q contains the member functions corresponding to every σ method in q in *Fluent*. They call the global function invoke_{σ} . Their return types are automatically inferred by `auto`. We assume that the prototypes of the invoke_{σ} functions have been already declared.

As we did in Scala, we overload the invoke_{σ} functions; each implementation of these functions corresponds to a method declaration in *Fluent*. We use function templates for overloading the functions. When a method in *Fluent* is the form of `def <T>  $T_1$ . $\sigma$ () = new  $T_2$ ()`, we declare the following invoke_{σ} function:

```
template<typename T>
auto invoke_ $\sigma$ ( $T_1$ * stack) {
    return new  $T_2$ ();
}
```

This invoke_{σ} function receives a value of T_1 and returns a value of T_2 . Note that invoke_{σ} is not a member function but a global function. Hence it can be overloaded on the template parameter T_1 .

When a method in *Fluent* is the form of `def <T>  $T_1$ . $\sigma_1$ () = new  $T_2$ () .  $\sigma_2$ ()`, we declare:

```
template<typename T>
auto invoke_ $\sigma_1$ ( $T_1$ * stack) {
    return invoke_ $\sigma_2$ (new  $T_2$ ());
}
```

This invoke_{σ_1} function constructs a value of type T_2 from the given argument of type T_1 and then calls invoke_{σ_2} function with that value.

Finally, we show the global function `begin`, which starts a method chain:

Listing 6. The grammar of the if/else language

```

syntax ifElse (Stmt) {
  Return      : Stmt -> "return_(String)"
  Throw       : Stmt -> "throw_(String)"
  IfThen      : Stmt -> "if_(Boolean)" Then
  IfThenElse  : Stmt -> "if_(Boolean)" Then Else
  ThenClause  : Then -> "then_" Stmt
  ElseClause  : Else -> "else_" Stmt
  TryCatch    : Stmt -> "try_" Stmt "catch_(String => Stmt)"
}

```

```

qinit<int>* begin() {
  return new qinit<int>();
}

```

The `begin` function returns an instance of `qinit<int>` expressing to an initial stack, which contains only `qinit`. Note that we here express an empty stack by `int`.

5 EXPERIMENT

We have developed a fluent API generator based on our approach. This section presents this generator and the compilation time of the generated code.

5.1 Fluent API Generator

We have developed a fluent API generator named `TypelevelLR`.¹ Its design is based on our approach described above. It reads the definition of an LR grammar and generates the skeleton of a library with a fluent API with type checking.

`TypelevelLR` can generate a library in Scala, Haskell, and C++. The program in Haskell uses the pipe operator. The generated library provides a fluent API for the deep embedding style [Gibbons and Wu 2014]. Thus, a chain of method calls to the library constructs a parse tree representing the chain. Each method call in the chain incrementally constructs the parse tree. Giving the semantics to the method-call chain is the responsibility of the library developer, or in other words, the user of `TypelevelLR`. The developer could implement an interpreter that executes the parse tree constructed by the generated library.

For example, `TypelevelLR` can generate a fluent-API library from the grammar shown in Listing 6. `TypelevelLR` reads a source file containing Listing 6 and generates a library that constructs a parse tree from a method-call chain if the chain is valid in the grammar. The identifiers before the colons, such as `Return` and `Throw`, are used as the type names of the parse-tree nodes constructed by the library.

In the grammar definition, a terminal symbol is denoted by an identifier enclosed in double-quotations. It can take an argument list enclosed in parentheses. For example, `return_` takes an argument of type `String`. The argument value is stored in the parse-tree node corresponding to that terminal symbol. `TypelevelLR` does not check the existence of the argument type. If that type does not exist, the fact is reported as a compilation error when the generated library is compiled by the host-language compiler.

¹ The source code of `TypelevelLR` is publicly available from <https://www.github.com/csg-tokyo/typelevelLR> and <https://doi.org/10.5281/zenodo.3374835>. We have also developed another generator for Scala. This generator `ScaLALR` is also based on our approach but it provides more functionalities by exploiting language features unique to Scala. It is available from <https://github.com/csg-tokyo/scalalr>.

Listing 7. A syntactically incorrect program using the if/else language in Listing 6

```

1  object ifElseTest {
2      import ifElse._
3      def main(args: Array[String]) = {
4          for (n <- 1 to 100) {
5              val message: String = begin()
6                  .if_(n % 3 == 0)
7                  .then_().if_(n % 5 == 0)
8                      .then_().return_("fizzbuzz")
9                      .else_().return_("fizz")
10                     .else_().return_("buzz")
11                     .else_().return_("oops!") // unacceptable else_()
12                 .end().run()
13             println(message)
14         }
15     }
16 }

```

Listing 8. The grammar of our DOT-like language

Directed	: Graph	-> "digraph(String)" Stmts
Undirected	: Graph	-> "graph(String)" Stmts
StmtsCons	: Stmts	-> Stmt Stmts
StmtsNull	: Stmts	-> eps
NodeStmt	: Stmt	-> "node(String)" Ands NodeAttrs
AndsCons	: Ands	-> "and(String)" Ands
AndsNull	: Ands	-> eps
EdgeStmt	: Stmt	-> "edge(String)" Ands "to(String)" Ands EdgeAttrs
NodeAttrsCons	: NodeAttrs	-> NodeAttr NodeAttrs
NodeAttrsNull	: NodeAttrs	-> eps
EdgeAttrsCons	: EdgeAttrs	-> EdgeAttr EdgeAttrs
EdgeAttrsNull	: EdgeAttrs	-> eps
NodeAttrColor	: NodeAttr	-> "color(String)"
NodeAttrShape	: NodeAttr	-> "shape(String)"
EdgeAttrColor	: EdgeAttr	-> "color(String)"
EdgeAttrStyle	: EdgeAttr	-> "style(String)"

As we mentioned in Section 2, the grammar in Listing 6 is an LR grammar. The library generated by TypelevelLR from this grammar consists of 903 lines of Scala code with 82 implicit functions. When compiling the user code shown in Listing 7, the Scala compiler reports a compilation error in line 11; the call to `else_` is not acceptable there since that `else_` does not match any `if_`.

5.2 Compilation Time of DOT-Like DSL

We next present the compilation time of the user code of the fluent-API library generated by TypelevelLR. We show that the compilation time is significantly shorter than Gil's approach in 2016, which was reported as being impractically slow [Gil and Levy 2016]. Gil's paper [Gil and Levy 2016] reported that the compilation of a chain of 26 method calls took around 30 seconds in Java. It also mentioned that the compilation time exponentially grows probably due to a design flaw of the Java compiler. We also observed similar results of our own experiment with the newer JVM.

Listing 9. A user program of our DOT library

```

object test {
  import dot.rightRec._
  def main(args: Array[String]) = {
    val graph = begin()
      .digraph("test")
      .node("A")
      .node("B")
      .node("C")
      .node("D")
      .node("E")
      .edge("A").to("B")
      .edge("B").to("C")
      .edge("C").to("D")
      .edge("D").to("E")
      .edge("E").to("A")
    .end()
    println(graph)
  }
}

```

Fig. 2. The graph drawn by Listing 9

For the experiment, we used the grammar similar to the DOT language shown in Listing 8. In this grammar, the iteration is right-recursive. Since the form of recursion may affect the compilation time, we also used the equivalent grammar except that the iteration is left-recursive. The differences between the two grammars are line 3, 6, 10, and 12. The followings are the syntax rules for left-recursion:

```

StmtCons      : Stmts      -> Stmts Stmt
AndCons       : Ands       -> Ands "and(String)"
NodeAttrCons  : NodeAttrs  -> NodeAttrs NodeAttr
EdgeAttrCons  : EdgeAttrs  -> EdgeAttrs EdgeAttr

```

Then we ran TypelevelLR to generate libraries for each grammar and target language: Scala, Haskell, or C++. The programs we compiled were like Listing 9. The graph drawn by Listing 9 is shown in Figure 2. We wrote programs like Listing 9 for the three host languages and for the different numbers of graph nodes from 1 to 100. The number of the graph nodes changes the length of the method-call chain in the programs.

We measured the compilation time of these programs. We used the Scala compiler scalac 2.11.6 with the option `-J-Xss100m`, the Glasgow Haskell compiler ghc 7.10.3 with `-O2 -fcontext-stack=5000`, and the GNU C++ compiler g++ 5.4.0 20160609 with `-O2 -std=c++17`. The Scala compiler was run on the JVM 1.8.0_101. These compilers were run on a machine with Intel® Core™ i7-4770S, 16 GB memory, and Ubuntu-16.04.5 LTS. The compilation time was measured after 5 warm-up runs. We measured the means and standard deviations of 20 runs of compilation.

Figures 3, 4, and 5 present the results. We observed that the compilation time was not exponential; it looked linear or quadratic for the number of the method calls in the chain in any host language. The grammar with left recursion showed shorter compilation time. This would be because the stack does not grow deep under the left-recursive grammar and hence a fewer types were generated. Even in C++, the user program with more than 200 method calls in the chain of the right-recursive grammar was compiled within 30 seconds. We believe that our approach can generate a library for an LR grammar so that the user's code can be compiled in polynomial time. Furthermore, our compilation speed was significantly faster than Gil's approach in 2016 [Gil and Levy 2016]. On the

Fig. 3. Compilation time in Scala

Fig. 4. Compilation time in Haskell

Fig. 5. Compilation time in C++

Table 3. The DSL grammars

DSL	<i>expr</i>	<i>syntax</i>	<i>while-lang</i>	<i>SQL</i>
# of lines	8	10	18	70
# of non-terminal symbols	3	5	3	33
# of terminal symbols	5	7	20	39

other hand, our approach is not applicable to Java since it exploits language features that are not available in Java although Gil’s approach can be used in Java and maybe other languages.

5.3 Compilation Time of Randomly Generated Chain

Since our translation directly encodes an LR automaton into overloaded methods in the target programming language, the compilation time of a chain of method calls to the generated library is ideally linear with respect to the length of the chain as an LR automaton accepts an input sequence in linear time. However, we observed a quadratic curve for the compilation time during the experiment with the DOT-like language shown above. The curve did not look like an exponential curve but we could not assess it as linear. This fact would be due to the various implementation issues of the target compilers.

For further investigation, we did similar experiments for other DSLs. Since we needed a large variety of method-call chains that are valid in the DSL grammars, we implemented a random method-call chain generator based on the literature [Mairson 1994]. The generator takes a context-free grammar and that length of a chain. Then it randomly generates a method-call chain of that length so that it will be valid in that grammar.

The experiments used the following four DSLs²:

- *expr*: a small subset of arithmetic expressions. It consists of only two binary operators, addition and multiplication, and parentheses. Multiplication takes precedence over addition.
- *syntax*: the DSL we designed to express a grammar definition for TypelevelLR. A program in this DSL consists of several derivation rules. Each derivation rule consists of three parts: a name, a non-terminal symbol derived by the rule, and a sequence of terminal or non-terminal symbols. Listing 8 is written in this DSL.
- *while-lang*: a DSL with a similar grammar to the while language described in the literature [Nielson and Nielson 1992]. It consists arithmetic expressions, boolean operations, variable assignments, if statements, and while statements. No operator precedence is given. The operators are right-associative.
- *SQL*: a subset of the SQL language. We designed this DSL by referring to the SQL grammar publicized by Apache Phoenix project [Foundation 2014]. This DSL supports only a subset of the select statement.

The sizes of the grammars of these DSLs are summarized in Table 3.

We measured the compilation time of various lengths of method-call chains in those DSLs embedded in Scala, Haskell, and C++. We used the Scala compiler scalac 2.11.6 with the option -J-Xss100m, the Glasgow Haskell compiler ghc 7.10.3 with -O2 -fcontext-stack=5000, and the GNU C++ compiler g++ 5.4.0 20160609 with -O2 -std=c++17. The Scala compiler was run on the JVM 1.8.0_222. These compilers were run on a machine with dual Intel® Xeon® E5-2637v3 Haswell 3.50GHz, 512 GB memory, and Ubuntu-18.04.3 LTS.

Figures 6, 7, and 8 show the results of our measurements for *expr*, *syntax*, *while-lang*, and *SQL*, respectively. The compilation time was the means of five runs after two warm-up runs. During the

² Their grammar definitions are available from <https://github.com/csg-tokyo/typelevelLR>. See the examples folder.

Fig. 6. Compilation time of random chains in Scala.

Fig. 7. Compilation time of random chains in Haskell

experiment for each DSL, we randomly generated 10 different instances of method-call chains for every chain-length from 1 to 200. For the *SQL* DSL, the lengths of the generated chains were every 10 from 10 to 200. All the figures are scatter plots. The horizontal axis represents the number of method calls in the chain. The vertical axis represents the compilation time in seconds.

In all the figures, we do not observe the exponential growth of the compilation time. They rather look linear. As in the experiment with the DOT-like language, the C++ compiler was slower

Fig. 8. Compilation time of random chains in C++.

than the other compilers. The compilation speed, however, still looks linear. Although the Scala compiler showed steady performance in these experiments, it slowed down by a factor of ten when it compiled short method-call chains in *SQL* on a different machine with a similar processor but a smaller amount of memory. The slow-down of the other compilers was a factor of two on this machine. This would be because the size of the *SQL* grammar was large and hence a large library was generated from that grammar and it caused frequent garbage collection. The size of the generated library was 1.8 MB (37K lines).

5.4 Error Messages

The readability of the error messages is often raised as a drawback of the type-based encoding that we adopted for TypelevelLR. An error message is printed when an invalid chain of method calls is detected, but the content of the message depends on the implementation of the host-language compiler. As far as we know, most compilers do not enable a source program to customize an error message.

We below show example error messages printed when the library generated by TypelevelLR detects an invalid chain of method calls. We showed the DOT-like language in Section 2 and implemented a Scala library for this DSL in Section 5.2. When the programmer using this library wrongly calls the `shape` method instead of the `style` method as follows:

```
.edge("A").to("B").shape("dotted")
```

then this invalid call to `shape` is detected. the host Scala compiler `scalac` prints this error message:

```
test.scala:10: error: value shape is not a member of dotDSL.Node6[
dotDSL.Node18[dotDSL.Node7[dotDSL.Node16[dotDSL.Node16[dotDSL.Node16[
dotDSL.Node9[dotDSL.Node1[Unit]]]]]]]]
    .edge("A").to("B").shape("dotted")
                        ^
one error found
```

Although the state of the LR automaton is exposed in the message, we can still read the line number and the method name that causes an error. This line number and the method name indicate the correct position of the error. Only the reason of the error is difficult to read. The programmer has to know that this error message reports an invalid chain of method calls. This is a limitation of our approach.

In Haskell, the example shown above is written as follows:

```
|> edge "A" |> to "B" |> shape "dotted"
```

This erroneous code causes the following error messages by the Haskell compiler ghc:

```
Test.hs:5:8:
  No instance for (Show r0) arising from a use of 'print'
  The type variable 'r0' is ambiguous
  Note: there are several potential instances:
    instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
    instance Show Ordering -- Defined in 'GHC.Show'
    instance Show Integer -- Defined in 'GHC.Show'
    ...plus 30 others
  In the expression: print
-- omit 28 lines --
Test.hs:9:34:
  No instance for (ShapeTransition
    (Node5 (Node9 (Node6 (Node10 (Node11 (Node1 ()))))) s0)
    arising from a use of 'shape'
  In the second argument of '(|>)', namely 'shape "dotted"'
  In the first argument of '(|>)', namely
    'begin |> digraph "small_graph" |> node "A" |> shape "rectangle"
      |> node "B"
      |> node "C"
      |> shape "doublecircle"
      |> edge "A"
      |> to "B"
      |> shape "dotted"'
  In the first argument of '(|>)', namely
-- omit 9 lines --
Test.hs:10:12:
  No instance for (EdgeTransition s0 s1) arising from a use of 'edge'
  The type variables 's0', 's1' are ambiguous
-- omit the rest and three other errors --
```

In total, 209 lines are generated. Most lines are not useful to understand the error but only the second message starting with “Test.hs:9:34:” indicates that the error occurs at the call to shape in line 9.

For the same example, the C++ compiler g++ prints the following messages:

```
In file included from test.cpp:2:
In file included from ./dotDSL.hpp:627:
./dotDSL.hpp:impl:48:10: error: function 'shape_transition<dotDSL::Node6,
    dotDSL::Node18, dotDSL::Node7, dotDSL::Node16, dotDSL::Node16,
    dotDSL::Node16, dotDSL::Node9, dotDSL::Node1>' with deduced return type
    cannot be used before it is defined
    return shape_transition( this->lock(), arg1 );
           ^
test.cpp:13:29: note: in instantiation of member function
    'dotDSL::State<dotDSL::Node6, dotDSL::Node18, dotDSL::Node7,
    dotDSL::Node16, dotDSL::Node16, dotDSL::Node16, dotDSL::Node9,
    dotDSL::Node1>::shape' requested here
    ->edge("A")->to("B")->shape("dotted")
                   ^
./dotDSL.hpp:607:6: note: 'shape_transition<dotDSL::Node6, dotDSL::Node18,
    dotDSL::Node7, dotDSL::Node16, dotDSL::Node16, dotDSL::Node16,
```

```

dotDSL::Node9, dotDSL::Node1>' declared here
auto shape_transition( std::shared_ptr< State< Stack... > > const& src, ...
    ^
1 error generated.

```

Although the error messages are verbose, the second message “test.cpp:13:29: note: ...” indicates the correct error position. It also mentions the error happens at the call to `shape`. The other two messages refer to the implementation of the generated DSL library `dotDSL.hpp.impl`. In Haskell and C++, the compilers we used print verbose error messages although they correctly report the error position. Printing the verbose messages is a drawback of our approach.

6 RELATED WORK

The syntax checking in any grammar class (including the grammars that are not context-free) is possible in principle if the type system of the host language is Turing-complete such as Java [Grigore 2017] and C++ [Veldhuizen 2003]. For instance, when a given grammar is context-free, the syntax checking can be achieved by creating a CYK parser [Cocke 1969; Kasami 1965; Younger 1967] using a Turing machine emulated on a type system [Grigore 2017]. However, using a Turing machine in this way is overly complicated for most DSLs and causes practical problems. The technique requires a significant amount of memory and time to check even the syntax of a small chain [Grigore 2017]. Our encoding technique does not cause such practical problems as shown in the experiment in Section 5.

Gil and Levy proposed an algorithm to translate a grammar into a DSL embedded in Java with syntax checking by the type checker [Gil and Levy 2016]. It supports LR grammars. The algorithm generates the type definitions for encoding a jump-stack single-state real-time deterministic push-down automaton (JRPA), a variant of push-down automaton that can recognize deterministic context-free languages [Courcelle 1977]. However, the algorithm by Levy and Gil suffers from the exponential growth of compilation time to the length of the chain in the worst case. That exponential growth of compilation time is caused by the exponential growth of the size of the types appearing in the chain, as Levy and Gil showed in their experiment in [Gil and Levy 2016]. Our technique does not suffer from that problem as we mentioned above.

In parallel to our work, Gil and Roth proposed another algorithm to translate an LR language into a DSL embedded in Java [Gil and Roth 2019]. A main difference between their work and ours is that the approaches are totally different. Their approach uses only Java generics and generates a fluent API from a deterministic pushdown automaton by encoding a tree into a type. The tree is a state of that automaton. On the other hand, ours uses function overloading, which is implicit parameters in Scala, type classes in Haskell, and templates in C++, and generates a fluent API from an LR automaton by encoding a stack into a type. Another difference is that their generator Fling generates an API only from an LL(1) grammar although their algorithm supports the deterministic context free languages, which is known as being equivalent to LR languages. Our generator can generate an API from an LALR(1) grammar. Their paper only claims that the compiler could compile in a few seconds a chain of 30 method calls with signatures including parametric types used in their approach. Their experiment did not use a tool-generated DSL or a hand-written DSL.

EriLex [Xu 2010] and Fajita [Levy 2017] are fluent API generators. These existing tools can take only an LL(1) grammar as their input while our generator supports LR(1) grammars as we described earlier. Silverchain [Nakamaru et al. 2017] is a tool that generates an embedded DSL with sub-chaining API support. Sub-chaining API improves the user experience in that the API allows programmers to compose a chain by combining several sub-chains. The support of subchaining API is our future work to make our generator more practically applicable. However, Silverchain can take only an LL(1) grammar.

Several techniques for semantic checking have been also studied although this paper focuses only on the syntax correctness of DSL code. For example, AraRat [Gil and Lenz 2010] uses C++ template metaprogramming to check the syntax of SQL queries and the type-safety with respect to the database schema. The integration of such semantic checking and our fluent API generator is also future work. Other host language mechanisms such as operator overloading can be used to emulate a DSL sentence in a program written in a general-purpose language [Bock 2016]. Although we focused on fluent APIs in this paper, mixing those mechanisms with our technique would be a possible direction for future research.

Checker Framework [Dietl et al. 2011] is a framework to extend Java's type system. A number of static checking including syntax checking can be implemented by using Checker Framework. Standalone static analyzers such as Android Lint [Inc. 2011] can statically check the syntactic correctness of method chains. However, since those external standalone checkers are separately developed from the library, it might not be easy to maintain the checkers up-to-date to be compatible to the latest version of the library. Our generator would not cause this problem since the syntax checker is included in the library and they are developed together.

If we can freely extend the syntax of host languages, we can implement an embedded DSL that provides a more natural API without using a chain of method calls. ProteaJ [Ichikawa and Chiba 2017] and Wyvern [Nistor et al. 2013] are programming languages that natively support syntax extension. Since their extensibility is achieved by their underlying language mechanism such as their new type systems, this approach is not directly available to widely-used current programming languages without modifying these languages. A fluent API using method-call chains is a technique easily applicable to these languages.

7 CONCLUSION

This paper presented our fluent API generator for Scala, Haskell, and C++. It generates the skeleton of a library with a fluent API, or a library accessed through a method-call chain. The type safety by the host-language type checker ensures that all the chains of method calls to the generated library satisfies the constraints on the order of method calls. In this paper, we called the constraints the syntax of the fluent API since each method call in the chain is regarded as a lexical token and the constraints specify acceptable sequences of the lexical tokens. Our generator supports LR grammars for specifying that syntax. This paper proposed our algorithm for generating type declarations to enable that type safety. It first translates the given LR grammar into an LR automaton, translates it into a program in our *Fluent* language, and then translates it into the host-language program. The algorithm assumes that the host language supports function overloading considering type arguments, or type classes. The compilation time of the method-call chains to the generated library is polynomial.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their fruitful comments. This work was partly supported by JSPS KAKENHI Grant Number JP15H02682 and JST CREST Post-Petascale Software Project.

REFERENCES

- Roland Bock. 2016. rbock/sqlpp11: A type safe SQL template library for C++. <https://github.com/rbock/sqlpp11>.
- John Cocke. 1969. *Programming Languages and Their Compilers: Preliminary Notes*.
- Bruno Courcelle. 1977. On Jump-Deterministic Pushdown Automata. *Mathematical systems theory* (1977).
- Werner Dietl, Stephanie Dietzel, Michael D. Ernst, Kivanç Muşlu, and Todd W. Schiller. 2011. Building and Using Pluggable Type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, 681–690. <https://doi.org/10.1145/1985793.1985889>

- Apache Software Foundation. 2014. Apache Phoenix. <https://phoenix.apache.org/>.
- Martin Fowler. 2005. FluentInterface. <https://www.martinfowler.com/bliki/FluentInterface.html>.
- Emden R. Gansner and Stephen C. North. 2000. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE* 30, 11 (2000), 1203–1233.
- Jeremy Gibbons and Nicolas Wu. 2014. Folding domain-specific languages: Deep and shallow embeddings (functional Pearl). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*.
- Joseph (Yossi) Gil and Keren Lenz. 2010. Simple and safe SQL queries with C++ templates. *Science of Computer Programming* 75, 7 (2010), 573 – 595. Generative Programming and Component Engineering (GPCE 2007).
- Yossi Gil and Tomer Levy. 2016. Formal Language Recognition with the Java Type Checker. In *Proceedings of 30th European Conference on Object-Oriented Programming*.
- Yossi Gil and Ori Roth. 2019. Fling - A Fluent API Generator. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. <https://doi.org/10.4230/LIPIcs.ECOOP.2019.13>
- Jonathan Goldstine, John K. Price, and Detlef Wotschke. 1981. On reducing the number of states in a PDA. *Mathematical systems theory* 15, 1 (01 Dec 1981), 315–321. <https://doi.org/10.1007/BF01786988>
- Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*.
- Stephan Heilbrunner. 1981. A parsing automata approach to LR theory. *Theoretical Computer Science* 15, 2 (1981), 117 – 157. [https://doi.org/10.1016/0304-3975\(81\)90067-0](https://doi.org/10.1016/0304-3975(81)90067-0)
- Kohei Honda. 1993. Types for dyadic interaction. In *CONCUR'93*, Eike Best (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 509–523.
- Paul Hudak. 1996. Building Domain-Specific Embedded Languages. *Comput. Surveys* (1996).
- Kazuhiro Ichikawa and Shigeru Chiba. 2017. User-Defined Operators Including Name Binding for New Language Constructs. *The Art, Science, and Engineering of Programming* 1, 2, Article 15 (2017), 15:1–15:25 pages.
- Google Inc. 2011. Improve your code with lint checks. <https://developer.android.com/studio/write/lint>.
- Tadao Kasami. 1965. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages*. Technical Report.
- Dimitrios Kouzapas, Ornela Dardha, Roly Perera, and Simon J. Gay. 2018. Typechecking protocols with Mungo and StMungo: A session type toolchain for Java. *Science of Computer Programming* 155 (2018), 52–75. <https://doi.org/10.1016/j.scico.2017.10.006>
- Stefan Krüger, Johannes Späth, Karim Ali, Eric Bodden, and Mira Mezini. 2018. CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs. In *European Conference on Object-Oriented Programming (ECOOP)*. 10:1–10:27.
- Tomer Levy. 2017. A Fluent API for Automatic Generation of Fluent APIs in Java.
- Harry G. Mairson. 1994. Generating Words in a Context-free Language Uniformly at Random. *Inf. Process. Lett.* (1994). [https://doi.org/10.1016/0020-0190\(94\)90033-7](https://doi.org/10.1016/0020-0190(94)90033-7)
- Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, and Shigeru Chiba. 2017. Silverchain: a fluent API generator. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*.
- Hanne Riis Nielson and Flemming Nielson. 1992. *Semantics with Applications: A Formal Introduction*.
- Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan Aldrich. 2013. Wyvern: A Simple, Typed, and Pure Object-oriented Language. In *Proceedings of the 5th Workshop on Mechanisms for Specialization, Generalization and Inheritance (MASPEGHI '13)*. ACM, 9–16.
- Joshua Sunshine, Karl Naden, Sven Stork, Jonathan Aldrich, and Éric Tanter. 2011. First-class State Change in Plaid. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, 713–732. <https://doi.org/10.1145/2048066.2048122>
- Peter Thiemann and Vasco T. Vasconcelos. 2016. Context-free Session Types. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP 2016)*. ACM, 462–475. <https://doi.org/10.1145/2951913.2951926>
- Todd Veldhuizen. 2003. *C++ Templates are Turing Complete*. Technical Report. Indiana University Computer Science.
- Hao Xu. 2010. *EriLex: An Embedded Domain Specific Language Generator*.
- Daniel Younger. 1967. Recognition and parsing of context-free languages in time n^3 . *Information and Control* (1967).