Generating Fluent Embedded Domain-Specific Languages with Subchaining

Tomoki Nakamaru[a,*], Kazuhiro Ichikawa[a], Tetsuro Yamazaki[a], Shigeru Chiba[a]

[a] *The University of Tokyo, Japan*

Abstract

This paper presents Silverchain, a tool that generates a fluent embedded domain-specific language (EDSL) from a BNF-style grammar. A fluent EDSL is a class library that allows its users to embed domain-specific sentences into a program written in a general-purpose language by method chaining. A generated EDSL is designed so that its users can find syntactic errors from type errors and make effective use of method chaining in various situations. The first feature is realized by setting the return type of each method based on which methods may be invoked next, and the second feature is achieved by providing subchaining APIs besides regular chaining APIs. The contribution of this paper is the development of a translation method from a grammar to such an EDSL. Our translation method is modeled as the construction of a set of single-state real-time deterministic pushdown automata (RPAs). A fluent EDSL is obtained by encoding those RPAs into class definitions. In the construction of RPAs, Silverchain does not add or remove any non-terminal from the given grammar. This constraint is required to generate subchaining APIs as specified in the grammar.

Keywords: Embedded Domain-Specific Language, Fluent API

1. Introduction

Method chaining has been drawing attention as a technique to embed domain-specific languages (DSLs) into a general-purpose language (GPL). By using method chaining, a DSL sentence can be embedded into a GPL program without losing the appearance of the original sentence. For example, an SQL query is embedded into a Java program by chaining method calls as follows:

```
// SELECT * FROM user WHERE id = 1;
select("*").from("user").where().col("id").eq().val(1);
```

An API is often called a fluent API [1] when it is designed to emulate sentences of a language as shown in the example above. In this paper, we refer to a class library with its fluent API as a fluent embedded DSL (fluent EDSL).

*Corresponding author
Email address: nakamaru@csg.ci.i.u-tokyo.ac.jp (Tomoki Nakamaru)

Several techniques have been studied to detect syntactic errors of embedded sentences as type errors of the host language [1, 2, 3]. Here, a syntactic error indicates an arrangement of method calls that is incorrect from the viewpoint of the grammar of a DSL, for example as follows:

```
select("*").where().col("year").eq().val(2018); // Missing FROM clause
```

The detection is implemented by setting the return type of each method based on which methods the users can invoke next. The syntactic error above is detected by setting the return type of `select(String)` to a type that does not accept `where()`.

Silverchain is a tool that translates a BNF-style grammar into a fluent EDSL with the syntax checking feature. Silverchain differs from existing translators in that a generated EDSL supports subchaining APIs in addition to regular chaining (or non-subchaining) APIs. A subchaining API is a chaining API for building a domain-specific sentence from fragments as follows:

```
Condition c1 = col("year").eq().val(2018);
Condition c2 = col("month").eq().val(10);
select("*").from("book").where().condition(bool? c1 : c2);
```

Using subchaining APIs, the users can provide a part of the parse tree structure by naming nonterminals. This API provides good user experiences that can not be achieved through regular chaining APIs, especially when composing a part of a sentence dynamically as shown above.

Our translation is modeled as the construction of deterministic pushdown automata without ϵ-transitions called single-state real-time deterministic pushdown automata (RPAs). The class definitions of a fluent EDSL are generated by encoding those RPAs. Our RPA construction method is different from the literature [4, 5] in that it does not add or remove non-terminals from the given grammar. This property is essential to generate subchaining APIs as specified in the grammar. Silverchain can generate a fluent EDSL from any context-free grammar, but the EDSLs generated from some grammars require subchaining APIs to compose certain parts of sentences. This limitation is due to our RPA construction method.

The rest of this paper is organized as follows: Section 2 describes the motivation for creating a generator of a fluent EDSL with both syntax-checking and multi-style chaining APIs. Section 3 describes our translation method from a BNF-style grammar to a such an EDSL and discusses the limitation of our method. In Section 4, we present use cases to illustrate the ability and the limitation of Silverchain. We discuss related work in Section 5 and conclude with a summary in Section 6.

2. Motivation

With a fluent EDSL, sentences of a DSL can be emulated nicely inside programs written in a GPL. As an example, consider a DSL for writing itemized documents whose syntax is defined by the grammar in Figure 1. Parentheses

```
<list> -> "begin" (<text> <list>?)+ "end";
<text> :: String; // <text> is a double-quoted string.
```

Figure 1: Grammar of example DSL

```
begin
  "Item 1"
    begin
      "Item 1.1"
      "Item 1.2"
    end
  "Item 2"
    begin
      "Item 2.1"
      "Item 2.2"
    end
end
```

(a) Usage example

```
begin()
  .text("Item 1")
    .begin()
      .text("Item 1.1")
      .text("Item 1.2")
    .end()
  .text("Item 2")
    .begin()
      .text("Item 2.1")
      .text("Item 2.2")
    .end()
.end();
```

(b) Fluent EDSL

```
List list = new List();
list.addText("Item 1");
List subls1 = new List();
subls1.addText("Item 1.1");
subls1.addText("Item 1.2");
list.addList(subls1);

list.addText("Item 2");
List sublis2 = new List();
subls2.addText("Item 2.1");
subls2.addText("Item 2.2");
list.addList(subls2);
```

(c) Non-fluent EDSL

```
"begin
    \"Item 1\"
      begin
        \"Item 1.1\"
        \"Item 1.2\"
      end
    \"Item 2\"
      begin
        \"Item 2.1\"
        \"Item 2.2\"
      end
end"
```

(d) String embedding

Figure 2: Usage example and three embedded forms into Java

are used to group elements, and a plus sign represents one or more occurrences of the preceding element. Figure 2a shows a usage example of this DSL, and Figures 2b, 2c, and 2d show three embedded forms of that example in Java. In Figure 2b, that usage example is embedded using method chaining without changing the basic syntax of the original sentence. Since the syntax of a DSL is designed to simplify expressions in the domain, keeping the original syntax helps programmers make effective use of a DSL. Figure 2c shows a non-fluent EDSL version of the same example. This form does not resemble the original sentence, and the code tends to contain more intermediate variables. In Figure 2d, our example is written using the form called string embedding. The program is written as a string literal and passed to a library function for execution. Although this form keeps the original syntax, embedded sentences become hard

```
List listA = begin().text("Item A").end();
List listB = begin().text("Item B").end();
begin().text("...").list(useA? listA : listB).end();
```

(a) With subchaining

```
IncompleteChain1 c1 = begin().text("...");
IncompleteChain2 c2;
if (useA) { c2 = c1.begin().text("Item A").end(); }
else      { c2 = c1.begin().text("Item B").end(); }
List list = c2.end();
```

(b) Without subchaining

Figure 3: Composing a part of a chain dynamically

```
// Without subchaining
begin().text("Item 1").begin().text("Item 1.1").end().end();

// A chain becomes longer with subchaining
begin().text("Item 1").list(begin().text("Item 1.1").end()).end();
```

Figure 4: Simple itemization

to read when they contain escape sequences, and no syntax checking is applied to embedded sentences at compile-time.

The syntax of the embedded sentences by method chaining can be statically checked by setting the return type of each method based on which methods are allowed to invoke next. For example, the following syntactic error can be detected by making `begin()` return a type that does not accept `end()`:

```
// Empty list is not allowed according to the grammar.
begin().end(); // Cause "cannot find symbol method 'end()'"
```

This EDSL improves user experiences in two ways. First, users can find misuses at compile-time because incorrectly chained calls cause a type error. Second, a method-completion system of modern editors can provide only methods that do not cause syntax errors. Users can easily compose a syntactically correct sentence by following provided hints and do not have to pay attention to syntactic errors.

To take advantage of method chaining in various situations, a fluent EDSL should provide subchaining APIs besides regular chaining APIs. A subchaining API allows the users to group semantically related calls into a chain as follows:

```
List sublist = begin().text("Item 1.1").end();
begin().text("Item 1").list(sublist).end();
```

This API helps its users to keep code easy to understand, especially when composing a part of a sentence dynamically as shown in Figure 3a. Without subchaining, intermediate variables tend to be meaningless, and users cannot make good use of method chaining as shown in Figure 3b.

Subchaining API, however, introduces redundant method calls and makes a chain longer. This property of subchaining API prevents programmers from composing short or simple sentences briefly. For example, the two chains shown in Figure 4 represent the same document, but the chain using subchaining API contains the extra method call `list(...)`. Such an extra method call does not appear and a chain becomes shorter when using non-subchaining API. As demonstrated, the best style of chaining API depends on the case. Fluent EDSLs supporting only subchaining APIs (or only regular chaining APIs) do not allow the users to choose the best style. Their users cannot fully exploit the benefits of method chaining.

Constructing a well-designed fluent EDSL (a fluent EDSL with syntax-checking and multi-style chaining APIs) is a burdensome task for EDSL developers. For syntax-checking, developers need to define a number of classes and choose the return type of each method carefully. Although the number of required classes varies depending on the grammar, it exceeds a hundred for the grammar of most practical DSLs. Furthermore, each class definition gets larger when making fluent EDSLs support both regular chaining APIs and subchaining APIs. To ease that burden, the developers of widely-used EDSLs relax the rules to follow and thereby reduce the number of classes to define. This workaround, however, allows the EDSL users to compose syntactically incorrect sentences. For instance, the following sentences written with popular fluent EDSLs are accepted by Java type checker, but those should be rejected from the viewpoint of the grammar of the DSLs:

```
// "SELECT * ORDER BY book.id" in jOOQ
select().orderBy(BOOK.id).getSQL(); // Missing ‘FROM’ clause
// "<div src=‘./image.png’></div>" in j2html
div().withSrc("./image.png").render(); // Incorrect attribute ‘src’
```

Such incomplete checking is not only unsafe but also makes method completion show methods that possibly cause a syntactic error. The checking should be complete so that the users can use EDSLs at ease even when they do not know much about the original DSLs.

3. Silverchain

To address the problem mentioned in the previous section, we propose Silverchain, a tool that translates a grammar into a fluent EDSL. Figure 5 illustrates the overview of our translation. Silverchain first constructs a set of RPAs, each of which corresponds to a non-terminal of a given grammar. It then encodes those RPAs into class definitions to obtain a fluent EDSL. For clarity, in the following, we denote an RPA for a non-terminal n by R_n.

R_n accepts the symbol sequences derived from a non-terminal n. The following shows two examples of sequences that are accepted by R_{list} (the RPA for `<list>` in Figure 1):

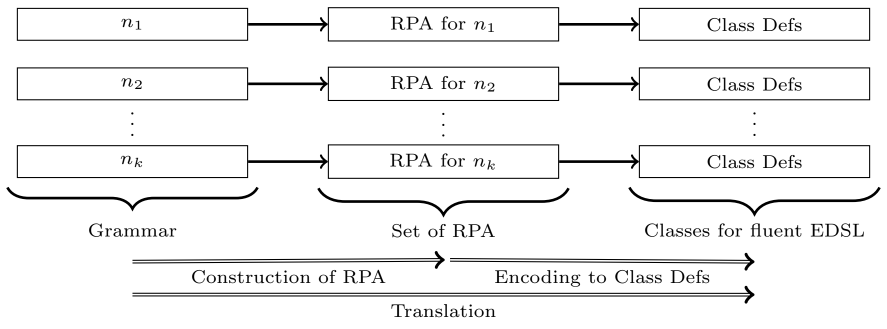


Figure 5: Overview of our translation method (n_i is a non-terminal).

```
begin <text> <list> end
begin <text> begin <text> end end
```

The first sequence is derived by applying the production rule of `<list>` once. The second sequence is derived by applying the rule recursively. More specifically, it is derived by inlining `<list>` in the first sequence with the sequence `begin <text> end`, which is also derived from `<list>`. Note that an accepted sequence contains not only terminals but also non-terminals.

The RPAs are encoded into class definitions in a way that an accepted sequence is emulated by chaining method calls of the generated classes. For example, the symbol sequences above are emulated by the following chains:

```
begin().text("...").list(...).end();
begin().text("...").begin().text("...").end().end();
```

Each symbol in a sequence is encoded into a method. The argument of a method depends on the kind of a symbol. If a symbol is a terminal such as `begin`, a symbol is encoded into a method with no argument. Otherwise, a symbol is encoded into a method that takes one argument. If a symbol is a typed terminal such as `<text>`, the argument is an instance of the type specified on the right-hand side of the rule. If a symbol is a non-terminal such as `<list>`, the argument is a subchain that emulates a production of that non-terminal:

```
begin().text("...").list(begin().text("...").end()).end();
```

In the following, we first provide the formal definition and the table representation of an RPA to illustrate the goal of our construction. We then describe the central part of our RPAs construction method in Section 3.2. Section 3.3 describes the preprocessing part, which rewrites a given grammar into a form that the central part can process. In Section 3.4, we describe the encoding of RPAs and discuss how to implement the semantics of a generated EDSL. Section 3.5 discusses the limitation of our construction method and describes Silverchain's approach to address that limitation.

	list				
	q_0	q_1	q_2	q_3	q_4
`begin`	q_1q_4	-	q_1q_3	-	-
`<text>`	-	q_2	q_2	q_2	-
`<list>`	-	-	q_3	-	-
`end`	-	-	ϵ	ϵ	-
$\$_{list}$	-	-	-	-	ϵ

Table 1: Table representation of R_{list}

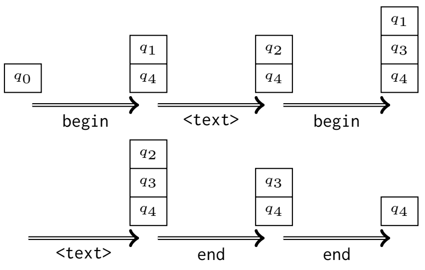


Figure 6: Example stack transition

3.1. RPA and Its Table Representation

An RPA is a 4-tuple $(\Sigma, \Gamma, z_0, \delta)$, where Σ is a finite set of input symbols, Γ is a finite set of stack elements, z_0 is the initial stack element, $\delta : (\Sigma \times \Gamma) \mapsto \Gamma^*$ is a transition function, and Γ^* is the set of all finite sequences of the elements in Γ. In the beginning, the stack is filled with z_0. At every step, an RPA consumes one input symbol, pops the top of its stack, and pushes zero or more elements into its stack. An RPA stops when it consumes all input symbols or when it becomes unable to consume an input symbol anymore with its defined transitions. An input sequence is accepted if the stack becomes empty by consuming the last symbol of a given sequence. A sequence is, on the other hand, rejected if an RPA stops before consuming the last symbol.

An RPA can be described by a table since the transition function of an RPA is a binary function. Table 1 shows the table representation of R_{list}. The table has all input symbols (including the end-of-input symbol $\$_n$) in its rows and all stack elements in its columns. A transition $(s, q) \to Q$ is represented by the cell value Q in row s of column q. When pushing elements, the rightmost element of Q is pushed into the stack first. The symbol - in a cell indicates that no transition is defined, and the symbol ϵ indicates that no element is pushed into the stack on that transition. The initial stack element is indicated by adding the non-terminal name on the column top of that element.

R_n has restrictions on the number of elements to push into its stack. When R_n consumes the first symbol of a nested construct in a sequence, it pushes two elements. When R_n consumes the last symbol of a nested construct, it pushes no element. R_n pushes one element otherwise. Figure 6 shows how the stack content of R_{list} changes at each step when the input sequence is as follows:

```
begin <text> begin <text> end end
```

R_{list} pushes two elements when consuming `begin`, pushes no element when consuming `end`, and pushes one element when consuming other symbols such as `<text>`. When consuming the first symbol of a nested construct, R_{list} firstly pushes an element that will appear on the stack top after consuming the last symbol of that nested construct. For instance, when R_{list} consumes the first `begin` in Figure 6, it first pushes q_4, which is the element on the top after consuming the last `end`.

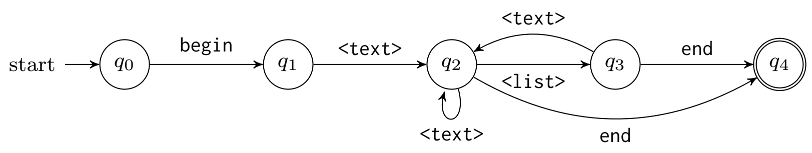


Figure 7: State diagram of D_{list}

Table 2: Tables appearing in RPA construction

(a) Table of R'_n

	list				
	q_0	q_1	q_2	q_3	q_4
begin	q_1	-	-	-	-
<text>	-	q_2	q_2	q_2	-
<list>	-	-	q_3	-	-
end	-	-	q_4	q_4	-
$\$_{list}$	-	-	-	-	ϵ

(b) Intermediate table

	list				
	q_0	q_1	q_2	q_3	q_4
begin	$q_1\boldsymbol{q_4}$	-	-	-	-
<text>	-	q_2	q_2	q_2	-
<list>	-	-	q_3	-	-
end	-	-	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}$	-
$\$_{list}$	-	-	-	-	ϵ

(c) Table of R_n

	list				
	q_0	q_1	q_2	q_3	q_4
begin	q_1q_4	-	$\boldsymbol{q_1q_3}$	-	-
<text>	-	q_2	q_2	q_2	-
<list>	-	-	q_3	-	-
end	-	-	ϵ	ϵ	-
$\$_{list}$	-	-	-	-	ϵ

3.2. Construction of RPAs

The central part of our construction consists of two steps. For each nonterminal n, our method first constructs the RPA that accepts only direct productions of n. It then extends that RPA into R_n. Here, a direct production of n refers to a sequence that matches the regular expression on the right-hand side of n's production rule. Note that the right-hand side can be considered as a regular expression on symbol sequences. For clarity, we denote the RPA accepting only direct productions of n by R'_n.

The first step constructs R'_n from a deterministic finite automaton (DFA) that accepts n's direct productions. Such a DFA (hereinafter denoted by D_n) can be obtained by using well-known algorithms [6, 7] since the expression on the right-hand side is a regular expression that matches the direct productions. Figure 7 shows the state diagram of D_{list}. R'_n is constructed by converting D_n to an equivalent RPA. The following describes the conversion from D_n to R'_n:

(1.1) Enumerate all input symbols (including $\$_n$) and all states of D_n across the rows and the columns, respectively. A state of D_n is converted to a stack element of R'_n.

(1.2) Put q_i in row s of column q_j if D_n can move from q_j to q_i by consuming s. A transition of D_n from q_j to q_i is converted to an action of R'_n that pops q_j and pushes q_i into the stack.

(1.3) Put n on the top of the column whose header is the initial state of D_n. The initial state of D_n is converted to the initial stack element of R'_n.

(1.4) Put ϵ in row $\$_n$ of a column whose header is an accepting state of D_n. An accepting state of D_n is converted to an element that allows R'_n to make its stack empty by consuming $\$_n$

Table 2a is the table representation of R'_{list}, which is obtained from D_{list}. Since a DFA can be regarded as an RPA whose stack depth is limited to one, the table of R'_n can always be constructed from D_n.

The second step extends R'_n to R_n to fill the following two gaps. While R'_n always pushes one element for every consumption, R_n pushes two elements when consuming the first symbol of a nested construct and pushes no element when consuming the last symbol. R'_n differs from R_n also in that, while R'_n accepts only direct productions, R_n accepts not only direct productions but also indirect productions. An indirect production is a sequence derived by replacing a non-terminal occurrence in the direct production with a production of that non-terminal.

The second step assumes that every application of a production rule introduces a nested construct into a sequence. Under this assumption, the first and last symbol of a nested construct is the first and last symbol on the right-hand side of the production rule. Although most grammars are not in this form, the preprocessing part of our algorithm, which is described later, rewrites a given grammar into this form.

To fix the number of pushed elements, the second step changes cell values in the rows of the first and last symbols of nested constructs. In our example case, our method changes cell values in row `begin` and row `end`. Let e_n be an element whose column contains ϵ in row $\$_n$. e_n is the element pushed when R'_n consumes the last symbol of the nested construct. The number of pushed elements is fixed by the following process:

(2.1) Replace all occurrences of e_n with ϵ. By this modification, R'_n pushes no element when consuming the last symbol.

(2.2) Append e_n to cell values in the column of the initial stack element for n. By this modification, R'_n pushes two elements when consuming the first symbol of a nested construct.

In the example case, e_n is q_4 and Step (2.1) replaces all occurrences of q_4 with ϵ. Step (2.2) appends q_4 to the value in row `begin` of column q_0. Table 2b shows the intermediate table obtained by applying the above process to Table 2a.

To accept indirect productions, the second step puts new values in the table. Since a non-terminal m introduces a nested construct under our assumption, R_n pushes two elements when it consumes the first symbol of an inlined sequence derived from m. The first pushed element is the element that appears on the stack top after consuming the last symbol of the inlined sequence, that is, the element pushed when consuming m. The second pushed element is the next stack top, that is, the element pushed by consuming the first symbol of m's production rule. Let C_m be the collection of actions that consume m and let P_m be the collection of actions that pop the initial stack element for m. The following describes this process to accept indirect productions:

(2.3) Put $q_k q_j$ in row s of column q_i for each $((x, q_i) \to q_j, (s, y) \to q_k z) \in C_m \times P_m$. Here, x, y, and z are placeholders that are not related to the newly put value.

Table 3: Tables appearing in RPAs construction

(a) R'_{doc} and R'_{list}

	doc			list				
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
`beginDoc`	q_1	-	-	-	-	-	-	-
`beginLst`	-	-	-	q_4	-	-	-	-
`<text>`	-	-	-	-	q_5	q_5	q_5	-
`<list>`	-	q_1	-	-	-	q_6	-	-
`endLst`	-	-	-	-	-	q_7	q_7	-
`endDoc`	-	q_2	-	-	-	-	-	-
$\$_{doc}$	-	-	ϵ	-	-	-	-	-
$\$_{list}$	-	-	-	-	-	-	-	ϵ

(b) R_{doc} and R_{list}

	doc			list				
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
`beginDoc`	$q_1\boldsymbol{q_2}$	-	-	-	-	-	-	-
`beginLst`	-	$\boldsymbol{q_4 q_1}$	-	$q_4\boldsymbol{q_7}$	-	$\boldsymbol{q_4 q_6}$	-	-
`<text>`	-	-	-	-	q_5	q_5	q_5	-
`<list>`	-	q_1	-	-	-	q_6	-	-
`endLst`	-	-	-	-	-	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}$	-
`endDoc`	-	$\boldsymbol{\epsilon}$	-	-	-	-	-	-
$\$_{doc}$	-	-	ϵ	-	-	-	-	-
$\$_{list}$	-	-	-	-	-	-	-	ϵ

When m is `<list>`, C_m is $\{(\texttt{<list>}, q_2) \to q_3\}$ and P_m is $\{(\texttt{begin}, q_0) \to q_1\}$. Step (2.3) adds $(\texttt{begin}, q_2) \to q_1 q_3$. Table 2c shows the table obtained by applying the above process to Table 2b.

To show how our method works when an input grammar contains multiple non-terminals, consider a new example grammar:

```
<doc>  -> "beginDoc" <list>* "endDoc";
<list> -> "beginLst" (<text> <list>?)+ "endLst";
<text> :: String;
```

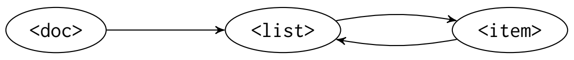
From this grammar, our method first constructs Table 3a. In Table 3a, R_{doc} and R_{list} are shown as one table, but the representation is the same as the case where the table describes only one RPA. Our method then defines e_m, C_m, and P_m as follows:

$$\begin{aligned}
&e_{doc} = q_2,\ e_{list} = q_7,\\
&C_{doc} = \emptyset,\ C_{list} = \{(\texttt{<list>}, q_1) \to q_1, (\texttt{<list>}, q_5) \to q_6\},\\
&P_{doc} = \{(\texttt{beginDoc}, q_0) \to q_1 q_2\},\ P_{list} = \{(\texttt{beginLst}, q_3) \to q_4 q_7\},
\end{aligned}$$

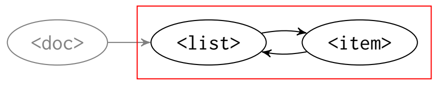
where $\emptyset$ denotes an empty set. In Step (2.1), our method first replaces q_2 and q_7 with ϵ. In Step (2.2), our method then appends q_2 and q_7 to the cell values in column q_0 and q_3, respectively. In Step (2.3), our method finally adds two actions $(\texttt{beginLst}, q_1) \to q_4 q_1$ and $(\texttt{beginLst}, q_5) \to q_4 q_6$, which are derived from $C_{list} \times P_{list}$. (No new actions are derived from $C_{doc} \times P_{doc}$ since $C_{doc} \times P_{doc}$ is $\emptyset$.) Table 3b is the table representation of R_{doc} and R_{list}, which is obtained by modifying Table 3a.

3.3. Preprocessing

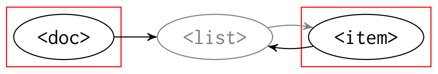
The preprocessing part of our method rewrites a given grammar into a form where every non-terminal has a direct recursion. To obtain that form, the preprocessing part expands all the occurrences of non-terminals in the right-hand side of a production rule. They are replaced with the right-hand side of their production rule. For example, consider the following grammar that has an indirect recursion:



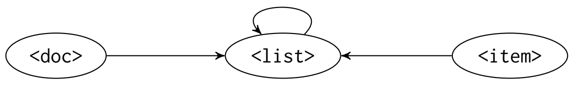
(a) Initial reference relation graph



(b) Examining (`<doc>`,`<list>`). This edge is not selected since the SCC containing `<list>` consists of two nodes.



(c) Examining (`<list>`,`<item>`). This edge is selected since the SCC containing `<item>` consists only of one node without a self-loop.



(d) Updated graph after expanding `<item>` in the rule for `<list>`

Figure 8: Selecting an edge for inline expansion. (A red rectangle indicates an SCC.)

```
<doc>  -> "beginDoc" <list>* "endDoc";
<list> -> "beginLst" <item>+ "endLst";
<item> -> <text> <list>?;
<text> :: String;
```

The preprocessing part expands `<item>` in the second line since it makes mutual recursion:

```
<doc>  -> "beginDoc" <list>* "endDoc";
<list> -> "beginLst" (<item> | <text> <list>?)+ "endLst";
<item> -> <text> <list>?;
<text> :: String;
```

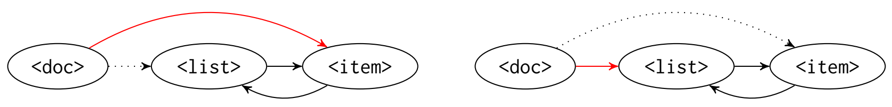
The replaced non-terminals are ignored when building C_m in the central part since those occurrences have already been inlined by the preprocessing. In the case shown above, the occurrence of `<item>` on the right-hand side of the rule of `<list>` is ignored.

The preprocessing part does not expand all the occurrences of non-terminals since infinite regression will occur when the non-terminals are defined recursively. It selects non-terminals to be expanded by examining refer-to relations among non-terminals. A refer-to relation from a non-terminal n_s to a non-terminal n_d indicates that the rule for n_s contains one or more un-expanded n_d on its right-hand side. For example, the example grammar contains three references from `<doc>` to `<list>`, from `<list>` to `<item>`, and from `<item>` to `<list>`. Our preprocessing is performed as follows:

(I) Create a graph G. A node in G represents a non-terminal in the given grammar. An edge (n_s, n_d) in G represents a relation from n_s to n_d.

(II) Find an edge (n_s, n_d) such that the following sub-process returns true:

  (i) Create a subgraph G' of G by removing n_s and the edges with n_s on their source or destination.



(a) Expanding (`<doc>`, `<list>`) (b) Expanding (`<doc>`, `<item>`)

Figure 9: Without decomposition into SCCs

(ii) Decompose G' into strongly connected components (SCCs) and let the component containing n_d be C [8].

(iii) Return true if C consists of a single node without a self-loop, and false otherwise.

(III) Apply inline expansion to the production rule for n_s. All the occurrences of n_d in the rule are expanded.

(IV) Update G to reflect the inline expansion at (iii). The selected edge (n_s, n_d) is removed from G and new edges are added to G to represent refer-to relations in the expanded production rule.

In Step (I), the preprocessing part constructs the graph shown in Figure 8a from the example grammar. In Step (II), it then selects (`<list>`, `<item>`) through the sub-processes shown in Figure 8b and 8c. In Step (III), `<list>` will be expanded into the following as we have seen before:

```
"beginLst" (<item> | <text> <list>?)+ "endLst";
```

By Step (IV), G is updated to the graph in Figure 8d. Our preprocessing repeats the above process until no edges found in Step (II). An edge further from the node for the start symbol is examined earlier.

Decomposition into SCCs in Step (II) is required to avoid an infinite regression of inline expansion caused by mutual recursion in the given grammar. Suppose that the edge (`<doc>`, `<list>`) in Figure 8a is selected for expansion. The dotted edge is removed, and the red edge is added as shown in Figure 9a. Then the edge (`<doc>`, `<item>`) in Figure 9a is selected for expansion and the graph is updated to Figure 9b, where the dotted edge is removed and the red edge is added. Figure 9b is equivalent to Figure 8a. Further inline expansions will cause infinite regression. By the decomposition, Silverchain selects an edge composing a mutual-recursion cycle earlier than other edges. Such an edge is expanded, and the cycle is transformed into non-cyclic edges and self-loops. Which edge is first selected among cyclic edges does not matter to resolve infinite regression.

3.4. Encoding into Class Definitions

Our method encodes a snapshot of an RPA into nested generics. Since an RPA has only one state, a snapshot of the RPA is the stack content. For example, when the stack contains q_i and q_j from its top, the stack is encoded into the type `Qi<Qj<Bottom>>`. A transition $(s, q) \to Q$ is encoded into a method whose name is s and whose owner is the class for q. The return type of that

```
 1  // Classes each of which
 2  //   corresponds to a row
 3  class Q0 {
 4    static Q1<Q4<Bottom>> begin() { ... }
 5  }
 6  class Q1<T> {
 7    Q2<T> text(String text) { ... }
 8  }
 9  class Q2<T> {
10    Q1<Q3<T>> begin() { ... }
11    Q2<T> text(String text,
12                String... textArray) { ... }
13    Q3<T> list(List list) { ... }
14    T end() { ... }
15  }
16  class Q3<T> {
17    Q2<T> text(String text) { ... }
18    T end() { ... }
19  }
20  class Q4<T> extends List {}
21
22  // Auxiliary classes:
23  // Class corresponding to <list>
24  class List   {}
25  // Class for the stack bottom
26  class Bottom {}
```

	list				
	q_0	q_1	q_2	q_3	q_4
begin	q_1q_4	-	q_1q_3	-	-
<text>	-	q_2	q_2	q_2	-
<list>	-	-	q_3	-	-
end	-	-	ϵ	ϵ	-
$\$_{list}$	-	-	-	-	ϵ

Figure 10: Class definitions generated from Table 1

method is a type that represents Q. In the following, we describe our encoding scheme as the encoding from a table into class definitions.

The code in Figure 10 shows the classes generated from Table 1. (For better readability, we put Table 1 on the right of Figure 10.) The generated classes are categorized into two kinds. One is a set of classes each of which corresponds to a column of the table. The classes named `Qn` in the figure belong to this category. A class in this category has one type parameter except the class corresponding to the initial stack element. The type parameter of `Qn` is used to represent the stack content below the stack element corresponding to `Qn`. The other category consists of two auxiliary classes: the class corresponding to a non-terminal and the class representing the stack bottom. In Figure 10, `List` and `Bottom` belong to this category.

Each method in the generated class corresponds to a table cell. A method is defined in the class corresponding to column q when the method corresponds to a cell in column q. For example, the method of `Q1` on Line 7 corresponds to the cell in row `<text>` of column q_1. A method is modified with `static` if the owner class corresponds to the initial stack element. By adding `static`, the users of the EDSL can invoke the first method of a chain directly without writing the receiver class using a static import statement. Instead of encoding a cell in row $\$_n$ into a method, the cell is encoded into an `extend` clause. In the case of the example, Q4 extends List as shown on Line 20 since column q_4 has a value in row $\$_{list}$. By this special encoding, a chain representing a syntactically correct sequence can be assigned to a variable whose type is the class corresponding to the source non-terminal.

The return type of a method is determined by the corresponding cell value. A method returns a nested generics if the cell value is not ϵ. Classes are nested in a way that the leftmost element of the value is the outermost class of the nested generic. The innermost class is Bottom if the method is modified with static. Otherwise, the innermost class is the type parameter of the owner class. For example, q_1q_4 in column q_0 is encoded into Q1<Q4<Bottom>> as shown on Line 4. A method returns just the type parameter if the cell is filled with ϵ such as the method on Line 14. The argument of a method depends on the kind of the input symbol on the column of the corresponding cell. A method takes no argument if the symbol on the column is a terminal, and takes one argument otherwise. As we mentioned earlier, the type of that argument depends on the kind of symbol. If the symbol has a typed terminal, the argument type is as specified on the right of the :: operator. Otherwise, the argument type is the class corresponding to the symbol.

As a manually developed fluent API often provides several convenience methods, Silverchain also generates such convenience methods. The method on Line 11 is such a convenience method and takes multiple String objects as its arguments. The following lines show the example usage of that convenience method:

```
begin().text(  // begin()
    "Item 1",  //     .text("Item 1")
    "Item 2",  //     .text("Item 2")
    "Item 3"   //     .text("Item 3")
).end();       // .end();
```

Silverchain generates such a method with variable length arguments when it finds two actions $(s, q_i) \to q_j$ and $(s, q_j) \to q_j$.

Figure 11 shows the generated code for Q2 including the method bodies. Each method appends the object representing an invoked method to a list shared among state instances. Each method first appends a Method instance, which is an instance that holds the name and arguments of the method invocation, to the list. Line 9 in Figure 11 is the line that appends such an instance. A method appends multiple Method instances at once when it takes a subchain as its argument as shown on Line 16. The method then creates and return an instance that is used as the receiver of the next method invocation. The list of invoked methods is shared by passing it as the argument to the constructor as shown on Line 11. We need a little trick for generated methods to return a nested parametric type as shown from Line 26 to Line 31. Instead of executing new T(), Silverchain calls newInstance on a class object representing T. Since a type parameter is not a first-class entity in Java, the generated EDSL uses the reflection API and explicitly passes a type parameter as a class object.

We showed the method bodies only in Q2, but the method bodies in the other classes are very similar to the ones in Q2. For example, the body of

```java
class Q2<T> {
    ArrayList<Method> methodList;
    Stack<Class<?>> classStack;
    Q2(ArrayList<Method> list, Stack<Class<?>> stack) {
        methodList = list;
        classStack = stack;
    }
    Q1<Q3<T>> begin() {
        methodList.add(new Method("begin", null));
        classStack.push(Q3.class);
        return new Q3<>(methodList, classStack);
    }
    Q2<T> text(String text, String... textArray) {
        methodList.add(new Method("text", text));
        for (String t: textArray) {
            methodList.add(new Method("text", t));
        }
        return new Q2<>(methodList, classStack);
    }
    Q3<T> list(List list) {
        methodList.addAll(list.methodList);
        return new Q3<>(methodList, classStack);
    }
    T end() {
        methodList.add(new Method("end", null));
        try {
            return (T) classStack.pop()
                .getDeclaredConstructor(ArrayList.class, Stack.class)
                .newInstance(methodList, classStack);
        } catch (Exception e) {}
    }
}
```

Figure 11: Definition of `Q2`

`text(String)` in `Q1` is the same as the one in `Q2`. There is only one difference in the body of `Q0.begin()`. Since the method is modified with `static` and invoked at the beginning of a chain, it does not have a context to be passed. Therefore, it has to create a new list to record the invoked methods and a stack to store class objects as follows:

```java
static Q1<Q4<Bottom>> begin() {
    ArrayList<Method> methodList = new ArrayList<>();
    Stack<Class<?>> classStack = new Stack<>();
    methodList.add(new Method("begin", null));
    classStack.push(Q4.class)
    return new Q1<>(methodList, classStack);
}
```

EDSL developers (i.e., the Silverchain users) can add semantics to the generated EDSL by implementing an evaluation method, a method that interprets a written chain. Figure 12 shows an example implementation that constructs an itemized document in TEX from a written chain. In this implementation, each

```
String toTeX() {
    String tex = "";
    for (Method m: this.methodList()) {
        if (m.name == "begin")     { tex += "\begin{itemize}";     }
        else if (m.name == "end")  { tex += "\end{itemize}";       }
        else if (m.name == "text") { tex += "\item " + m.argument; }
    }
    return tex;
}
```

Figure 12: Example implementation of EDSL semantics

method in `methodList` is converted into a token of TEX. The Silverchain users can put all semantic actions into one method rather than editing method bodies scattered over the generated code. This approach to implement the semantics reduces the EDSL developers' task when updating and re-generate a modified EDSL. They do not have to edit a number of parts of the generated code.

The flattened list of invoked methods (e.g. `Q2.methodList`) helps EDSL developers when the EDSL is just an embedded interface to an external DSL such as SQL and TEX. However, such a list does not help the developers when building actions for the DSL that does not have an external execution system, or when the developers apply optimization to generated DSL code. In those cases, the developers may prefer tree-structured data over a flattened list of invoked methods. Although the developers can obtain tree-structured data from the flattened list by implementing a parser, for convenience, Silverchain generates method bodies that construct tree-structured data from a written chain. The generation of better method bodies is a future work that is required to use Silverchain in practice.

3.5. Limitation

Silverchain can generate regular (flat) chaining APIs for all parts when every nested construct in a given grammar begins and ends explicitly. If-else syntax with dangling-else is a common syntax component that does not have such explicit symbols:

```
<if-else> -> "if" <cond> <stmts> ("else" <stmts>)? "fi";
<stmts>   -> (<if-else> | <stmt>)*;
```

Since `else` corresponds to `if`, those two symbols introduce a nested construct into a sequence. However, that nesting may end implicitly since `else` is optional. Try-catch-finally syntax with dangling-finally is also an example of such a syntax component:

```
<TCF>   -> "try" <stmts> "catch" <err> ("finally" <stmts>)?;
<stmts> -> (<TCF> | <stmt>)*;
```

In this case, `try` and `finally` introduce a nested construct, but that nesting may end implicitly without `finally`.

Table 4: Table violating Condition (a) and its modification

(a) Table of R_{nest}

	nest			
	q_0	q_1	q_2	q_3
begin	q_1	-	-	-
<nest>	-	q_2		-
end	-	-	q_3	-
<text>	q_3	-	-	-
$\$_{nest}$	-	-	ϵ	ϵ

(b) $e_n = q_2$

	nest			
	q_0	q_1	q_2	q_3
begin	q_1**q_2**	-	-	-
<nest>	-	ϵ		-
end	-	-	q_3	-
<text>	q_3	-	-	-
$\$_{nest}$	-	-	ϵ	ϵ

(c) $e_n = q_3$

	nest			
	q_0	q_1	q_2	q_3
begin	q_1**q_3**	-	-	-
<nest>	-	q_2		-
end	-	-	ϵ	-
<text>	ϵ	-	-	-
$\$_{nest}$	-	-	ϵ	ϵ

Silverchain fails to generate flat chaining APIs since R_n pushes the element that will appear on the stack top after a nested construct ends. If a given grammar contains a recursive structure without explicit ending symbols, our method cannot determine which element will appear on the stack top after a nested construct ends. This limitation can be stated as follows in terms of the table representation of R'_n:

(a) Only one column contains ϵ in row $\$_n$.

(b) No value exists in row s of column q for all $((*, q) \to *, (s, *) \to *) \in C_m \times P_m$, where $*$ is a placeholder that is not related to this condition.

When the table of R'_n violates Condition (a), our method cannot perform Step (2.2) successfully. Table 4a is an example table that does not satisfy Condition (a), which is constructed from the following grammar:

```
<nest> -> "begin" <nest> "end"? | <text>;
<text> :: String;
```

Table 4a has two columns that contain ϵ in row $\$_{nest}$. To fix the number of pushed elements in Step (2.2), our method needs to find e_n, the element to append to the value in row `begin`. When the table contains multiple columns that contain ϵ in row $\$_n$, our method needs to choose either of those elements since a cell can contain only one value. However, the RPA obtained by appending one those elements accepts only a part of sequences that should be accepted. If our method chooses q_2 as e_n and perform Step (2.1) and Step (2.2) as shown in Table 4b, the RPA does not accept the following:

```
<text>
```

On the other hand, if our method chooses q_3 as e_n and perform Step (2.1) and Step (2.2) as shown in Table 4c, the RPA does not accept the following:

```
begin <nest>
```

When the table of R'_n violates Condition (b), our method cannot construct R_n correctly. Table 5a is an example of such tables, which is constructed from the following grammar:

```
<nest> -> "begin" (<nest> | "begin" <text>) "end";
<text> :: String;
```

Table 5: Table violating Condition (b) and its modification

(a) Table of R_{nest}

	nest				
	q_0	q_1	q_2	q_3	q_4
begin	q_1	q_3	-	-	-
<nest>	-	q_2	-	-	-
end	-	-	q_4	-	-
<text>	-	-	-	q_4	-
$\$_{nest}$	-	-	-	-	ϵ

(b) Fixed table of R_{nest}

	nest				
	q_0	q_1	q_2	q_3	q_4
begin	$q_1\boldsymbol{q_4}$	q_3	-	-	-
<nest>	-	q_2	-	-	-
end	-	-	$\boldsymbol{\epsilon}$	-	-
<text>	-	-	-	$\boldsymbol{\epsilon}$	-
$\$_{nest}$	-	-	-	-	ϵ

(c) Incorrect update

	nest				
	q_0	q_1	q_2	q_3	q_4
begin	q_1q_4	$\boldsymbol{q_1q_2}$	-	-	-
<nest>	-	q_2	-	-	-
end	-	-	ϵ	-	-
<text>	-	-	-	ϵ	-
$\$_{nest}$	-	-	-	-	ϵ

```
class Q0 {
    static Q1<Q4<Bottom>> begin() { ... }
}
class Q1<T> {
    Q3<T> begin() { ... }
    Q2<T> nest(Nest nest)  { ... }
}
class Q2<T> {
    T end() { ... }
}
class Q3<T> {
    T text(String text) { ... }
}
class Q4<T> extends Nest {
}
class Nest {}
class Bottom {}
```

Figure 13: Class definitions

Our method fixes the number of pushed elements in Step (2.1) and Step (2.2) as shown Table 5b. It then constructs C_{nest} and P_{nest} as follows:

$$C_{nest} = \{(\texttt{<nest>}, q_1) \to q_2\}\,, P_{nest} = \{(\texttt{begin}, q_0) \to q_1q_4\}\,.$$

In Step (2.3), our method put q_1q_2 in row `begin` of column q_1 as shown in Table 5c. However, the RPA described by Table 5c does not accept the following:

```
begin begin <text> end
```

If our method skips Step (2.3) to avoid overwriting cell values, the constructed RPA for a non-terminal n does not accept indirect productions of n.

Silverchain skips the second step in our RPA construction method when it finds the violation of Condition (a) or (b). Since the second step is the process to make RPAs accept indirect productions, RPAs remain to accept only direct productions. The result is that the generated fluent EDSL only allows its users to use subchaining APIs for certain parts of a chain. For example, consider encoding Table 5b into class definitions. Figure 13 shows the classes generated from the table and the EDSL users can write the following chains:

Table 6: Number of classes and methods

	#Symbols	#Non-terminals	#Classes (Generated)	#Methods (Generated)	#Classes (Hand-written)	#Methods (Hand-written)
LINQ	32	17	132	323	16	44
DOT	40	15	389	1687	25	337

```
begin().begin().text("...").end();
begin().nest(begin().begin().text("...").end()).end();
```

The users cannot write the following chain that uses regular chaining APIs:

```
begin().begin().begin().text("...").end().end();
```

This limitation is problematic from the viewpoint of DSL emulation since sub-chaining APIs introduce redundant parts into a chain.

4. Use Cases

In this section, we compare Silverchain-generated fluent EDSLs and popular hand-written EDSLs, using LINQ[1] and DOT[2] as examples. LINQ is a DSL for operating collection data, and DOT is a DSL for describing graphs. We chose coollection[3] as a popular EDSL for LINQ and graphviz-java[4] as one for DOT. Besides, we experimentally investigate how the length of a composed chain affects the compilation time since a Silverchain-generated EDSL heavily uses generics.

Table 6 summarizes the numbers of classes and methods generated from the grammars of those DSLs. The values in the first and second columns are the numbers of unique symbols and non-terminals in the grammar, respectively. The values in the third column to the sixth column are the numbers of classes and methods in the generated EDSLs and the hand-written EDSL. When counting classes and methods, we picked up only public ones since our primary concern is the API but not non-public implementations. As seen from the table, the numbers of generated definitions are too large to handle by hand although the grammars of those languages are relatively simple. Those numbers are significantly small in hand-written EDSLs. This smaller size is mainly due to the hand-written EDSLs allowing their users to compose syntactically incorrect chains as follows:

```
// Multiple ORDER BY clauses
from(users).orderBy("age",Order.DESC).orderBy("age",Order.ASC);
```

[1]http://programminglinq.info/tag/bnf/
[2]http://www.graphviz.org/content/dot-language
[3]https://github.com/19WAS85/coollection
[4]https://github.com/nidi3/graphviz-java

```
from u in users
    where u.age > 2
    orderby u.age descending
    select u;
```

(a) Example sentence

```
from(users)
    .where("age", gt(2))
    .orderBy("age", Order.DESC)
    .all();
```

(b) With coollection

```
from().collection(users)
    .where().field("age").gt().value(2)
    .orderBy("age").descending().select();
```

(c) With generated EDSL

Figure 14: Comparison in LINQ

No support of non-subchaining APIs also reduces the numbers of classes and methods of the hand-written EDSLs.

Figure 14 and 15 show example sentences and their embedded versions of LINQ and DOT, respectively. As seen from these examples, the code written with the generated EDSLs are similar to the one written with hand-written EDSLs in most parts. Figure 14c is similar to Figure 14b, and Figure 15c and 15d are similar to Figure 15b.

The first drawback is that code written with the generated EDSLs tends to contain redundant method calls. For instance, `from().collection(...)` in Figure 14c is expressed in a shorter way by `from(...)` in Figure 14b. Similarly, `beginGraph()` in Figure 15c is omitted in Figure 15b. These redundant methods are generated since our translation method encodes every token of a DSL into a method. Those methods can be omitted by using information obtained from the table representation of RPAs. A method can be omitted if it is the only transition that exists between states and no ambiguity arises after omitting that transition. However, if such automatic omission is applied naively, a method chain with the generated EDSLs might be unreadable for programmers. Some redundant symbols in a DSL are necessary for programmers to read and understand written sentences. To avoid this problem, Silverchain does not apply that automatic omission.

Another drawback is that programmers should not edit those classes to make APIs better since generated EDSLs use mechanically named classes such as `Q1` and `Q2`. The APIs of coollection and graphviz-java are designed using domain-specific knowledge that is not represented in the grammar. For instance, coollection uses `enum` to specify the order as shown in Figure 14b and graphviz-java uses `enum` to specify attributes of a node as shown in Figure 15e. However, to add such methods reflecting domain-specific knowledge, the developers of an EDSL need to fix scattered parts of generated classes. The generation of such methods is difficult since our translation method naively encodes every token of a sentence into a method. Finding a smarter way of encoding such features is

```
digraph G {
    A -> B
    B -> C
    C -> A
}
```

(a) Example sentence

```
graph("G").directed().with(
    node("A").link(node("B")),
    node("B").link(node("C")),
    node("C").link(node("A"))
);
```

(b) Using graphviz-java

```
digraph().id("G").beginGraph().edge(
    node("A").arrow().node("B"),
    node("B").arrow().node("C"),
    node("C").arrow().node("A")
).endGraph();
```

(c) Usage with subchaining

```
digraph().id("G").beginGraph()
    .node("A").arrow().node("B")
    .node("B").arrow().node("C")
    .node("C").arrow().node("A")
.endGraph();
```

(d) Usage without subchaining

```
// With graphviz-java
node("X").with(
    Shape.RECTANGLE,
    Style.FILLED
);
```

```
// With generated EDSL
node("X").beginAttr()
    .shape().eq().rectangle()
    .style().eq().filled()
.endAttr();
```

(e) Difference between generated EDSL and graphviz-java

Figure 15: Comparison in DOT

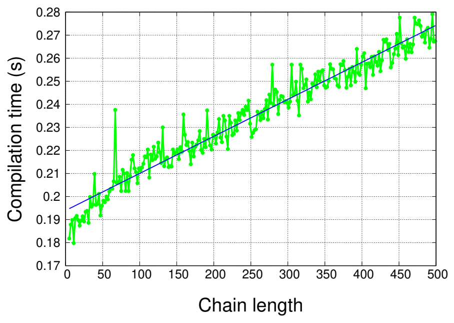
important future work to put Silverchain into practical use.

To investigate the relation between the compilation time and the length of a chain, we measured the compilation time of a chain of various length, using the generated EDSL for DOT. In our experiments, we compiled chains that contain various numbers of `subgraph().beginSubgraph()` as follows:

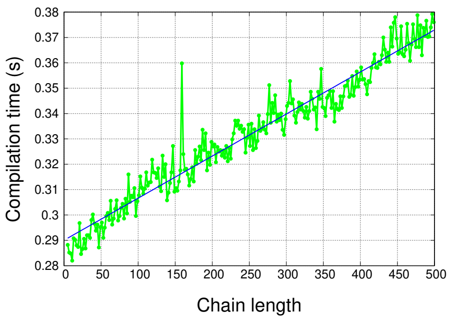
```
digraph().id("G").beginGraph()
  .subgraph().beginSubgraph(); // Length = 5
digraph().id("G").beginGraph()
  .subgraph().beginSubgraph()
  .subgraph().beginSubgraph(); // Length = 7
```

Since a call to `beginSubgraph` starts a new nested construct, the type size at the end of a chain increases as the chain becomes longer. Here, the size of a type is defined by the number of type names in the textual representation of the type. (The size of `G<T,S>` is three for example.) Our experiments are performed on a machine with Intel Core i7 3.3 GHz processor and 16 GB memory, using `javac 1.8.0_114` and `javac 9`. Two versions of Java were used in our experiments since Java 9 has a new type-checking strategy [9]. We used `javac -verbose` to compile chains and extracted total compilation time from its output.

Figure 16 shows the results of this experiment. We measured the compilation time five times for each length of a chain, and the averages are shown in the figures. The blue line in each figure is the linear regression line of data. The



(a) Java 8



(b) Java 9

Figure 16: Result of experiments

Table 7: Fitted parameters in $y = ax + b$

	a	b
Java 8	$1.61 \times 10^{-4} \pm 2.52 \times 10^{-6}$ ($\pm$1.57%)	$1.19 \times 10^{-1} \pm 7.30 \times 10^{-4}$ ($\pm$0.376%)
Java 9	$1.66 \times 10^{-4} \pm 2.67 \times 10^{-6}$ ($\pm$1.61%)	$2.90 \times 10^{-1} \pm 7.72 \times 10^{-4}$ ($\pm$0.266%)

fitted parameters are summarized in Table 7. As seen from these results, with an EDSL generated by Silverchain, the compilation time grows linearly to the length of a chain even in the worst case.

We also measured the compilation time for the chain shown in Figure 17. The chain is obtained by manually rewriting an official example[5] of DOT in our generated EDSL syntax. We compiled the chain 50 times using `javac 1.8.0_114` and `javac 9` on the same machine as the previous experiment. Table 8 summarizes the result of this experiment. From this result and the result in the previous experiment, we concluded that the compilation time is not as long as it causes problems in practical use.

5. Related Work

Levy and Gil proposed an algorithm to translate a BNF-style grammar into a fluent EDSL with a syntax-checking feature, although it does not support subchaining APIs [10, 11]. The algorithm generates such an EDSL by encoding a jump-stack single-state real-time deterministic pushdown automaton (JRPA) into class definitions. Since a JRPA can recognize deterministic context-free languages [12] and this class of languages is larger than the class that RPAs can recognize [4], the algorithm of Levy and Gil can generate non-subchaining APIs for grammars that Silverchain cannot generate them for. However, with an EDSL generated by the algorithm of Levy and Gil, the compilation time of a method chain grows exponentially to the length of the chain in the worst case.

[5]https://graphviz.gitlab.io/_pages/Gallery/directed/unix.html

```
digraph().id("G").beginGraph()
  .node("5th Edition").arrow().node("6th Edition")
  .node("5th Edition").arrow().node("PWB 1.0")
  .node("6th Edition").arrow().node("LSX")
  .node("6th Edition").arrow().node("1 BSD")
  .node("6th Edition").arrow().node("Mini Unix")
  .node("6th Edition").arrow().node("Wollongong")
  .node("6th Edition").arrow().node("Interdata")
  .node("Interdata").arrow().node("Unix/TS 3.0")
  .node("Interdata").arrow().node("PWB 2.0")
  .node("Interdata").arrow().node("7th Edition")
  .node("7th Edition").arrow().node("8th Edition")
  .node("7th Edition").arrow().node("32V")
  .node("7th Edition").arrow().node("V7M")
  .node("7th Edition").arrow().node("Ultrix-11")
  .node("7th Edition").arrow().node("Xenix")
  .node("7th Edition").arrow().node("UniPlus+")
  .node("V7M").arrow().node("Ultrix-11")
  .node("8th Edition").arrow().node("9th Edition")
  .node("1 BSD").arrow().node("2 BSD")
  .node("2 BSD").arrow().node("2.8 BSD")
  .node("2.8 BSD").arrow().node("Ultrix-11")
  .node("2.8 BSD").arrow().node("2.9 BSD")
  .node("32V").arrow().node("3 BSD")
  .node("3 BSD").arrow().node("4 BSD")
  .node("4 BSD").arrow().node("4.1 BSD")
  .node("4.1 BSD").arrow().node("4.2 BSD")
  .node("4.1 BSD").arrow().node("2.8 BSD")
  .node("4.1 BSD").arrow().node("8th Edition")
  .node("4.2 BSD").arrow().node("4.3 BSD")
  .node("4.2 BSD").arrow().node("Ultrix-32")
  .node("PWB 1.0").arrow().node("PWB 1.2")
  .node("PWB 1.0").arrow().node("USG 1.0")
  .node("PWB 1.2").arrow().node("PWB 2.0")
  .node("USG 1.0").arrow().node("CB Unix 1")
  .node("USG 1.0").arrow().node("USG 2.0")
  .node("CB Unix 1").arrow().node("CB Unix 2")
  .node("CB Unix 2").arrow().node("CB Unix 3")
  .node("CB Unix 3").arrow().node("Unix/TS++")
  .node("CB Unix 3").arrow().node("PDP-11 Sys V")
  .node("USG 2.0").arrow().node("USG 3.0")
  .node("USG 3.0").arrow().node("Unix/TS 3.0")
  .node("PWB 2.0").arrow().node("Unix/TS 3.0")
  .node("Unix/TS 1.0").arrow().node("Unix/TS 3.0")
  .node("Unix/TS 3.0").arrow().node("TS 4.0")
  .node("Unix/TS++").arrow().node("TS 4.0")
  .node("CB Unix 3").arrow().node("TS 4.0")
  .node("TS 4.0").arrow().node("System V.0")
  .node("System V.0").arrow().node("System V.2")
  .node("System V.2").arrow().node("System V.3")
.endGraph();
```

Figure 17: UNIX Family tree with our generated EDSL

Table 8: Compilation time for the chain in Figure 17

	Average	Median	Standard Deviation
Java 8	2.24×10^{-1}	2.21×10^{-1}	8.82×10^{-3} (3.94%)
Java 9	3.28×10^{-1}	3.26×10^{-1}	1.21×10^{-2} (3.69%)

This is caused by the exponential growth of the size of the type at the end of a chain, as Levy and Gil showed in their experiment using Java 8.

In principle, an EDSL with only non-subchaining APIs can be built for any grammar (including grammars that are not context-free) if the EDSL is implemented in a language with a Turing-complete type system such as Java [13] and C++ [14]. For instance, if given grammar is context-free, such an EDSL can be built by creating a CYK parser [15, 16, 17] implemented by a Turing machine emulated on a type system [13]. 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 just for compiling a small chain of two or three methods [18].

ScaLALR [19] generates a Scala EDSL only with non-subchaining APIs from a grammar by encoding an LALR parser into class definitions. With an EDSL generated by ScaLALR, the size of the type at the end of a chain does not grow exponentially to the length of the chain even in the worst case. The generated EDSL however uses uncommon features internally such as implicit classes and implicit conversions to avoid the exponential growth. Therefore, the technique used in ScaLALR is not portable to other languages such as Java and C++. On the other hand, the technique used in Silverchain is relatively simple and is portable to languages whose type system has generic classes.

EriLex [20] is a tool to generate an EDSL only with non-subchaining APIs from a given grammar by encoding an RPA into class definitions. The encoding method in EriLex is similar to the one in Silverchain and is portable to other languages. The drawback of EriLex is that input grammar needs to be LL(1) in Greibach normal form [21], which is a form that most of manually written grammars do not follow. Since the users of EriLex need to rewrite a grammar into that form, EriLex cannot generate a subchaining style API for each non-terminal. Furthermore, there is no algorithm to rewrite a grammar into the form required by EriLex as far as we know. Whether a given grammar can be rewritten into that form is undecidable [22]. There are several tools known as fluent API generators such as TS4J [23] and fluflu [24]. These tools, however, generate only subchaining APIs.

The role of type parameters in Silverchain differs from ones in manually written fluent APIs such as AssertJ [25], jOOQ [26], and j2html [27]. In those manually written fluent APIs, type parameters are used to eliminate boilerplate code in classes for APIs. For example in AssertJ, type parameters are used to store the type of `this` in Java [28] and help the developers to implement methods that have the same signature but a different return type. Conversely, Silverchain uses type parameters to express an infinite number of states that are produced from a CFG. The algorithm of Levy and Gil and EriLex also use type parameters in the same way as the one of Silverchain.

Although we focused only on the syntax correctness of a DSL sentence in this paper, techniques for semantic checking have also been studied. For example, AraRat [29] uses C++ template metaprogramming to allow its users to compose SQL queries that are syntactically correct and type-safe with respect to the database schema. The integration of such semantic checking and our EDSL generation technique is left for future research. We also focused on fluent EDSLs in this paper, but other host language mechanisms such as operator overloading can be used to emulate a DSL sentence in a GPL program [30]. The advantage of method chaining is that it requires only method invocation syntax, which should exist in any object-oriented language.

Extending the host syntax is another technique to embed DSL sentences into a GPL program. SugarJ [31] is a syntax extension mechanism of Java that allows programmers to write DSL sentences in their original syntax. The appearance of a written sentence is much better compared to the emulation by method chaining, but the parsing time gets longer when using SugarJ, since it converts written DSL sentences into plain Java sentences before compiling a program. ProteaJ [32], Wyvern [33], and Honu [34] are programming languages that natively support syntax extension. However, their features are realized by their underlying language mechanism such as type systems, so it is difficult to introduce a similar system to a language that is currently used in practice. A fluent EDSL, on the other hand, is a technique that can be applied to a number of general-purpose languages since it is just a class library.

6. Conclusion

This paper presented Silverchain, a tool that generates fluent EDSLs from a given grammar. Silverchain differs from existing tools in that a generated EDSL have the following two properties. First, it causes a type error when a built chain represents a syntactically incorrect sentence of the original DSL. Second, the generated EDSL supports subchaining APIs in addition to regular chaining APIs. Silverchain can generate fluent EDSLs for existing practical DSLs as we showed in Section 4.

Silverchain fails to generate regular chaining APIs for some grammars as we mentioned in Section 3.4. The first future work is to clarify the range of grammars that our current translation method can handle and to find its relation to the well-known classes of grammars. Finding a better translation method that handles a wider range of grammars but does not cause the exponential growth of compilation time is also future work. Another possible direction for further research is to improve the way of emulating DSLs by using another mechanism in the host language. For example, DSL sentences are currently emulated only by method chaining, but they could be emulated in a better way by using another syntax such as operator overloading. The properties of an emulated DSL other than syntactic rules could also be statically checked if the DSL's type system or name binding system is also mapped to its host language mechanism.

References

[1] M. Fowler, FluentInterface (December 2005).
URL `https://www.martinfowler.com/bliki/FluentInterface.html`

[2] E. Bodden, Efficient Hybrid Typestate Analysis by Determining Continuation-equivalent States, in: Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10, ACM, 2010, pp. 5–14 (2010).

[3] R. Strom, S. Yemini, Typestate: A programming language concept for enhancing software reliability, IEEE Transactions on Software Engineering SE-12 (1) (1986) 157–171 (Jan 1986).

[4] M. Harrison, I. Havel, Real-Time Strict Deterministic Languages, SIAM Journal on Computing 1 (4) (1972) 333–349 (1972).

[5] J. Pittl, A. Yehudai, Constructing a realtime deterministic pushdown automaton from a grammar, Theoretical Computer Science 22 (1) (1983) 57 – 69 (1983).

[6] J. Brzozowski, Canonical Regular Expressions and Minimal State Graphs for Definite Events, Mathematical theory of Automata (1962) 529–561 (1962).

[7] K. Thompson, Programming Techniques: Regular Expression Search Algorithm, Commun. ACM 11 (6) (1968) 419–422 (Jun. 1968).

[8] R. Tarjan, Depth-first search and linear graph algorithms, 12th Annual Symposium on Switching and Automata Theory (swat 1971) (1971) 114–121 (Oct 1971).

[9] V. Romero, M. Cimadamore, JEP 215: Tiered Attribution for javac (July 2016).
URL `http://openjdk.java.net/jeps/215`

[10] Y. Gil, T. Levy, Formal Language Recognition with the Java Type Checker, in: S. Krishnamurthi, B. S. Lerner (Eds.), 30th European Conference on Object-Oriented Programming (ECOOP 2016), Vol. 56, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 10:1–10:27 (2016).

[11] T. Levy, A Fluent API for Automatic Generation of Fluent APIs in Java, Master's thesis, Israel Institute of Technology (2017).

[12] B. Courcelle, On Jump-Deterministic Pushdown Automata, Mathematical systems theory 11 (1) (1977) 87–109 (Dec 1977).

[13] R. Grigore, Java Generics Are Turing Complete, in: Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, ACM, 2017, pp. 73–85 (2017).

[14] T. Veldhuizen, C++ Templates are Turing Complete, Tech. rep., Indiana University Computer Science (2003).

[15] J. Cocke, Programming Languages and Their Compilers: Preliminary Notes, Courant Institute of Mathematical Sciences, New York University, 1969 (1969).

[16] D. Younger, Recognition and parsing of context-free languages in time n3, Information and Control 10 (2) (1967) 189 – 208 (1967).

[17] T. Kasami, An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages, Tech. rep., DTIC Document (1965).

[18] R. Grigore, Java Generics are Turing Complete (August 2016).
URL `http://rgrig.appspot.com/static/papers/javats.html`

[19] K. Ichikawa, phenan/scalalr: ScaLALR : LALR parser generator for embedded DSLs in Scala (December 2016).
URL `https://github.com/phenan/scalalr`

[20] H. Xu, EriLex: An Embedded Domain Specific Language Generator, Springer Berlin Heidelberg, 2010 (2010).

[21] S. Greibach, A New Normal-Form Theorem for Context-Free Phrase Structure Grammars, J. ACM 12 (1) (1965) 42–52 (Jan. 1965).

[22] D. Rosenkrantz, R. Stearns, Properties of Deterministic Top Down Grammars, in: Proceedings of the First Annual ACM Symposium on Theory of Computing, STOC ’69, ACM, 1969, pp. 165–180 (1969).

[23] E. Bodden, TS4J: A Fluent Interface for Defining and Computing Typestate Analyses, in: Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis, SOAP ’14, ACM, 2014, pp. 1–6 (2014).

[24] P. Verhas, verhas/fluflu: Fluent Api Creator (July 2014).
URL `https://github.com/verhas/fluflu`

[25] J. Costigliola, AssertJ / Fluent assertions for java (December 2014).
URL `http://joel-costigliola.github.io/assertj/`

[26] D. G. GmbH, jOOQ: The easiest way to write SQL in Java (September 2017).
URL `https://www.jooq.org/`

[27] j2html, Fast and fluent Java HTML5 builder - Java HTML builder (September 2017).
URL `https://j2html.com/`

[28] A. Konermann, Emulating ”self types” using Java Generics to simplify fluent API implementation (November 2010).
URL `http://web.archive.org/web/20130721224442/http:/passion.forco.de/content/emulating-self-types-using-java-generics-simplify-fluent-api-implementation`

[29] J. Y. Gil, K. Lenz, Simple and safe SQL queries with C++ templates, Science of Computer Programming 75 (7) (2010) 573 – 595, generative Programming and Component Engineering (GPCE 2007) (2010).

[30] R. Bock, rbock/sqlpp11: A type safe SQL template library for C++, `https://github.com/rbock/sqlpp11`.

[31] S. Erdweg, T. Rendel, C. Kästner, K. Ostermann, SugarJ: Library-based Syntactic Language Extensibility, in: Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11, ACM, 2011, pp. 391–406 (2011).

[32] K. Ichikawa, S. Chiba, User-Defined Operators Including Name Binding for New Language Constructs, The Art, Science, and Engineering of Programming 1 (2) (2017) 15:1–15:25 (2017).

[33] L. Nistor, D. Kurilova, S. Balzer, B. Chung, A. Potanin, J. Aldrich, 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, 2013, pp. 9–16 (2013).

[34] J. Rafkind, M. Flatt, Honu: Syntactic Extension for Algebraic Notation Through Enforestation, in: Proceedings of the 11th International Conference on Generative Programming and Component Engineering, GPCE '12, ACM, 2012, pp. 122–131 (2012).