Yet Another Generating Method of Fluent Interfaces Supporting Flat- and Sub-chaining Styles

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 Arts and Sciences
Japan
nakamaru@graco.c.u-tokyo.ac.jp

Shigeru Chiba
The University of Tokyo
Graduate School of Information Science and Technology
Japan
chiba@acm.org

Abstract

Researchers discovered methods to generate fluent interfaces equipped with static checking to verify their calling conventions. This static checking is done by carefully designing classes and method signatures to make type checking to perform a calculation equivalent to syntax checking. In this paper, we propose a method to generate a fluent interface with syntax checking, which accepts both styles of method chaining; flat-chaining style and sub-chaining style. Supporting both styles is worthwhile because it allows programmers to wrap out parts of their method chaining for readability. Our method is based on grammar rewriting so that we could inspect the acceptable grammar. In conclusion, our method succeeds generation when the input grammar is LL(1) and there is no non-terminal symbol that generates either only an empty string or nothing.

CCS Concepts: • **Software and its engineering** → **API languages**; • **Theory of computation** → Grammars and context-free languages.

Keywords: fluent interface generation, LL(1) parsing, grammar rewriting

ACM Reference Format:
Tetsuro Yamazaki, Tomoki Nakamaru, and Shigeru Chiba. 2022. Yet Another Generating Method of Fluent Interfaces Supporting Flat- and Sub-chaining Styles. In *Proceedings of the 15th ACM SIGPLAN International Conference on Software Language Engineering (SLE '22), December 06–07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3567512.3567533

SLE '22, December 06–07, 2022, Auckland, New Zealand

ACM ISBN 978-1-4503-9919-7/22/12…$15.00
https://doi.org/10.1145/3567512.3567533

1 Introduction

A fluent interface is a library interface designed with method chaining. It has been getting popular [11] since expressions written with it are easy to read from left to right. Several researchers have been *hacking* existing type checkers to make fluent interfaces safe. The resulting *safe* fluent interfaces exploit type checkers so that they can statically determine whether a given chain of method calls is valid for the interface specification [2–4, 8–10, 12, 14, 15]. Since this validity checking can be regarded as parsing, a fluent interface can specify the grammar that defines its valid chains of method calls. Several design techniques for safe fluent interfaces have been studied to support a wider class of grammars and check the validity in a shorter time.

Another stream of research is to enhance the style of fluent interfaces. Although typical fluent interfaces allow a single long chain of method calls, some interfaces allow programmers to construct a long chain by combining several short chains. The former style is called *flat-chaining* while the latter is called *sub-chaining*. In the sub-chaining style, a short chain is separately constructed and attached to its *base* chain as an argument to a chained method. Silverchain [10] is a fluent-interface generator that supports both flat-chaining and sub-chaining. From a given grammar, it generates an interface where a program can mix both styles in the same chain.

Although Silverchain generates a safe fluent interface, the authors of Silverchain do not present which class of grammars Silverchain can generate an interface for. Silverchain translates every grammar rule into a finite state automaton, transforms it into a push-down automaton, and finally combines all the automata together. We argue that this rather ad-hoc technique complicates the interface generation and hence makes it difficult to understand the grammar class that Silverchain supports.

This paper proposes another method for generating a safe fluent interface. It supports both the flat-chaining style and the sub-chaining style. Our method is based on grammar rewriting. It first transforms a given grammar into another grammar directly supporting both styles. It then generates a safe fluent interface from the grammar. The latter step is straightforward; we can use an existing generator, such as

EriLex [14], from the LL(1) grammar to a safe fluent interface supporting the flat-chaining style only. The adoption of grammar rewriting enables us to analyze our method by existing techniques used when analyzing formal grammars. This paper also discusses the grammar class that our method can generate a safe fluent interface for.

Our contribution is twofold.

- We propose another method to generate a safe fluent interface supporting both flat-chaining and sub-chaining styles. The validity of a given method chain is statically checked by exploiting an existing type checker.
- We present the grammar class that our method can generate a safe fluent interface for. We also show the formal proof of this fact.

The rest of this paper is organized as following. We first briefly outline safe fluent interfaces and two styles of method chaining in Section 2. Then we propose our method in Section 3 and formally discuss its grammar class in Section 4. We show two exceptional cases where our method cannot generate a fluent interface in Section 5. Related work is presented in Section 6. Section 7 concludes this paper.

2 Background

2.1 Safe fluent interface

A fluent interface is a library interface designed to be used by chaining method calls. The following shows an example of such a chain of method calls in Java:

```
SQL.select("*").from("sample.db")
    .where("id = 1006").run();
```

A fluent interface is a promising design for embedded domain-specific languages (DSL). It enables programmers to mimic a DSL sentence while it requires only method call syntax. Many real-world libraries provide fluent interfaces; for example, Stream API[13], jMock[6], jOOQ[7].

Consider another example with the SQL library:

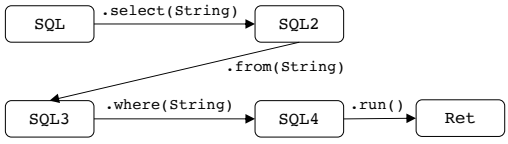
```
SQL.select("*").where("id = 1")
    .from("sample1.db").from("sample2.db").run()
```

This chain contains two consecutive `from` calls and mimics an invalid SQL query. Therefore, a runtime error will be thrown when we execute this chain.

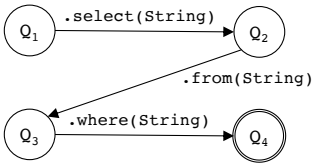
An invalid chain can be detected at compile time by appropriately setting the return type of each interface method. Listing 1 shows such a *safe* design of our fluent interface. Each method returns a type accepting only method calls that can be chained right after that method. For instance, the method `select` returns a type that offers only `from` because one can chain only a `from` call right after a `select` call. When a fluent interface is designed in this *safe* manner, the type checker rejects an invalid chain because it cannot resolve a method call in that chain.

```
class SQL {
    static SQL2 select(String columnSpec) {
        SQL2 result = new SQL2();
        ...
        return result;
    }
}
class SQL2 {
    SQL3 from(String tableSource) {
        SQL3 result = new SQL3();
        ...
        return result;
    }
}
class SQL3 {
    SQL4 where(String searchCondition) {
        SQL4 result = new SQL4();
        ...
        return result;
    }
}
class SQL4 {
    Result run() {
        /* construct and invoke an SQL query */
    }
}
```

Listing 1. Safe fluent interface of the SQL library



(a) Receiver type transition by method calls.



(b) A finite-state automaton.

Figure 1. Correspondence between receiver types and automaton states.

The design shown in Listing 1 can be viewed as a deterministic finite automaton (DFA) on a type system. Figure 1 shows the correspondence between DFA states and type definitions. A state of the DFA corresponds to a type definition in Listing 1, and a transition corresponds to a method definition.

Given that a DFA is a machine for checking a regular grammar, the correspondence in Figure 1 indicates the following: The validation of a method chain can be regarded as parsing with a type system. In general, parsing is a technique to

analyze the structure of a symbol sequence. A symbol does not have to be alphabetical. A symbol is a method call in the context of fluent interfaces (or in the context of validating a method chain).

Researchers have been applying more sophisticated parsing techniques to make fluent interfaces safer: Xu presented a technique to check an LL(1) grammar of method chains[14]; Gil and Levy presented a technique to check LR grammars with the Java type checker[2]; Yamazaki et al. proposed a collection of techniques to check LR grammars using the type systems of C++, Scala, and Haskell[15]. Grigore showed that the Java type checker can recognize any recursive language[4]. Xu's technique for LL(1) is practical since it only consumes a small amount of time and space at compile time. However, the techniques for LR grammars are less practical due to a large amount of time and space required at compile-time.

2.2 Flat-chaining and sub-chaining

To bring the techniques [2–4, 8, 10, 14, 15] into practice, code generators are needed since a number of complicated type definitions are required for safe fluent interfaces. In fact, those studies offer code generators as their research artifacts.

A fluent interface generated by those generators allows programmers to compose a DSL sentence *only as a single flat chain of method calls*. However, in practice, programmers often want to wrap out parts of chains and combine them later. The following shows a typical case where a programmer often creates a sub-chain:

```
Where where;
if (negate) {
    where = Where.column("id").notEquals(123).end();
} else {
    where = Where.column("id").equals(123).end();
}
select("*").from("sample.db").where(where).run();
```

The where clause of an SQL query is wrapped out as a sub-chain. The sub-chain is later combined with the *base* chain, depending on the given condition.

If a fluent interface only supports the flat-chaining style (i.e., allows only composing a single flat chain of method calls), a programmer needs to write as follows, to change a part of the chain dynamically:

```
SQL3 sql3 = select("*").from("sample.db");
SQL4 sql4;
if (negate) {
    sql4 = sql3.where().column("id").notEquals(123);
} else {
    sql4 = sql3.where().column("id").equals(123);
}
s4.run();
```

When sub-chaining is not supported by a fluent interface, a programmer needs to introduce an intermediate variable that groups a less meaningful part of a chain as shown above. Note that `sql3` and `sql4` are typed differently since a safe fluent interface switches the return type of each method based on its state.

From the viewpoint of practical use, a fluent interface should support both the flat-chaining style and the sub-chaining style. As we discussed above, the sub-chaining style works better when changing a part of a chain dynamically. However, the flat-chaining style would be preferred when composing a simple DSL sentence because the flat-chaining style allows a programmer to compose a simple sentence without nesting:

```
// flat-chaining style
select("*").from("sample.db")
    .where().column("id").equals(123)
    .run()
// sub-chaining style
select("*").from("sample.db")
    .where(Where.column("id").equals(123).end())
    .run()
```

Nakamaru et al. [10] discussed this chaining-style problem of existing generators and proposed a technique to generate a safe fluent interface with the sub-chaining support. In their paper, the flat style described above is called the non-subchaining style. In this paper, we refer to the flat style as the flat-chaining style.

2.3 Motivation

Nakamaru et al. [10] present a technique to generate a safe fluent interface with the sub-chaining support, but it only describes a complicated procedure. Their method consists of four steps. In the first step, they translate each right side of production rules into deterministic finite-state automata and generate class definitions that accept sub-chaining style chains. Then, in the second step, they try to extend each automaton by inlining transitions corresponding to non-terminal symbols, but they may not inline transitions to avoid infinite loops caused by their own recursive expansion. In the third step, they collect transitions not inlined in the previous step and generate RDPDA (realtime deterministic pushdown automata without ϵ-transitions) from each corresponding automaton. This translation from finite-state automata to RDPDAs is also done by a similar method to inlining. In the fourth step, they generate the entire fluent interface from obtained RDPDAs.

Due to this complexity, Nakamaru et al. [10] have not stated the limitation of their method clearly. The unclear limitation may bother the users of a fluent interface generator. To see whether it generates a fluent interface or not, the users need to actually give a grammar to the generator. Furthermore, they cannot know whether the generation succeeds by rewriting their grammar appropriately.

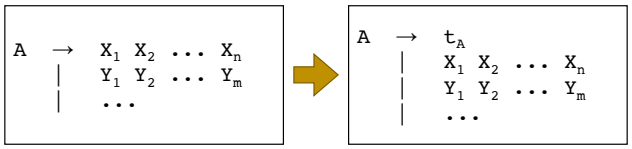


Figure 2. Grammar rewriting by f.

Our motivation is to develop a formal technique to generate a safe fluent interface with the sub-chaining support. If a technique is formalized, we can analyze the limitation of the technique such as the grammar class supported by the generator. Investigation of the grammar class supported by a fluent interface generator is an actively studied topic [2–4, 8, 12, 15], and this paper shares the same motivation with those studies. The development of a formal technique is also beneficial when applying the language theory and other parsing techniques to the generation of practical safe fluent interfaces.

3 Proposal: Sub-Chain Embedding

We propose a new method to generate a safe fluent interface. It is based on grammar rewriting. Like other fluent interface generators, our method takes a grammar definition as its input and generates class definitions for a fluent interface as its output.

The generated fluent interface supports not only the flat-chaining style but also the sub-chaining style. A programmer can separately construct a part of a method-call chain as a sub-chain if it corresponds to a non-terminal symbol in the given grammar. Then the programmer can attach that sub-chain to its *base* chain by giving it as an argument to the method call in that base chain.

Our method generates a safe fluent interface in two steps:

1. It first rewrites the input grammar G by applying a rewriting function f, and then
2. generates a fluent interface from the resulting grammar. The generation can be performed by any existing generator of the *safe* flat-chaining style from the LL(1) grammar. The resulting interface allows a sub-chain corresponding to any non-terminal symbol in G.

The rewriting function f plays a key role in our method. Figure 2 illustrates how the function f rewrites a given grammar. The function f adds a terminal symbol $\mathtt{t}_A$ to the right-hand side of the production rule for every non-terminal symbol A. $\mathtt{t}_A$ represents a method signature that a sub-chain derived from A can be given to as an argument. This simple rewriting *embeds* sub-chaining style method chaining into a grammar for flat-chaining style method chaining so that we can simply generate a fluent interface supporting both styles by applying an existing generation method that only supports flat-chaining style.

The second step generates a fluent interface from the grammar $f(G)$, where $\mathtt{t}_A$ is added for every non-terminal symbol A. To support both flat- and sub-chaining styles, our method generates a base interface and sub-interfaces for base chains and sub-chains, respectively. Let $codegen(g, n)$ be a function that generates class definitions for a safe fluent interface in the flat-chaining style. g denotes a grammar and n denotes a non-terminal symbol in g. We generate a top-most fluent interface by $codegen(f(G), A_{init})$ and sub-chain interfaces by $codegen(f(G), A)$ for each non-terminal symbol $A \in N$. Programmers use a sub-chain interface generated by $codegen(f(G), A)$ when they construct a sub-chain of method calls corresponding to a sequence derived from A. For example, if the production rule for A is $A \to \mathtt{t}_A \mid \mathtt{t}_0\mathtt{t}_1$ (where $\mathtt{t}_A$, $\mathtt{t}_0$,$\mathtt{t}_1$ are terminal symbols), the generated interface will accept a method-call chain like `A.t0().t1()`.[1] This sub-chain will be given to its base chain, for example, as `B.t2().tA(A.t0().t1()).t3()`. Here, we assume that `t0` and `t2` are static methods in class `A` and `B`, respectively, and the interface for the base chain is generated by $codegen(f(G), B)$ such that $B \to \mathtt{t}_B \mid \mathtt{t}_2\ A\ \mathtt{t}_3$.

Developers who implement our method can select any safe fluent interface generators for *codegen* if it accepts any LL(1) grammar as input, for example, EriLex [14]. In Section 3.3, we will briefly describe an example of *codegen* implemented with the algorithm for EriLex.

Note that developers implementing our method may also select a safe fluent interface generator that takes a grammar class other than LL(1) (*e.g.* LR and regular). However, it is out of the scope of this paper to discuss the grammar class supported by our method in that case.

Although there are several possible ways to give semantics to the generated fluent interfaces, we here assume that a method-call chain finally returns an object containing a parse tree representing the chain (or executes it at the last method call). To do that, a sub-chain for a non-terminal symbol A constructs a parse tree of that sub-chain for A and returns it. The method corresponding to $\mathtt{t}_A$ appends the given parse tree for A under the tree node representing the terminal symbol $\mathtt{t}_A$, which is a part of the parse tree of the chain including that method.

3.1 Grammar to denote a set of method chaining

A language is a set of symbol sequences. For example, the C language is a set of ASCII code sequences where ASCII code is aligned in the manner defined in the language specification. As we described earlier, a fluent interface can be safe by exploiting parsing techniques. In this context, the alphabets of a language are method signatures, and the sentences are chains of method calls. Note that the alphabets are *not* a

[1] The generated interface will also accept a chian like `A.tA(A.t0().t1())`. This is also a sub-chain for A.

set of method calls since a method call is identified by its signature when type checking.

Since a language is usually an infinite set with complex internal structures, it is often difficult to describe a language directly. A *grammar* is a tool to describe a language instead of directly describing them. It is a set of formal rules that produce a language. The Backus–Naur form (BNF) is a well-known meta language to write such rules.

Since only context-free grammars will appear in this paper, we will denote a grammar by 4-tuple $\langle \Sigma, N, \delta, A_{init} \rangle$, where

- Σ is a finite set of terminal symbols,
- N is a finite set of non-terminal symbols,
- $\delta : N \mapsto 2^{(\Sigma \cup N)*}$ is a production function, and
- $A_{init} \in N$ is a start symbol.

Terminal symbols are alphabets constituting sentences. A sentence $\overline{t}$ is a sequence of terminal symbols. We will put an overline on a variable if it represents a sequence. Non-terminal symbols are intermediate states appearing in derivation process. $(\Sigma \cup N)*$ represents a set of symbol sequences, and $2^{(\Sigma \cup N)*}$ is the power set of that sequence set.

A grammar generates a language by repeatedly applying the production function δ to the start symbol A_{init} until there remains no non-terminal symbol. Formally, the language $\mathrm{L}(G)$ generated from a grammar G can be described as $\mathrm{L}(G) = \Sigma^* \cap \mathit{derivation}_{\mathrm{G}}(A_{init})$ by using *derivation*, which is defined as follows:

$$\mathit{derivation}_{\mathrm{G}}(\overline{X}) = \{\overline{X}\} \cup \bigcup_{\overline{Y} \in \mathit{apply\text{-}rule}_{\mathrm{G}}(\overline{X})} \mathit{derivation}_{\mathrm{G}}(\overline{Y}),$$

where

$$\mathit{apply\text{-}rule}_{\mathrm{G}}(\overline{X}) = \bigcup_{\overline{Y_1}\, A\, \overline{Y_2} = \overline{X}, A \in N} \; \bigcup_{\overline{Z} \in \delta(A)} \overline{Y_1}\, \overline{Z}\, \overline{Y_2}.$$

A derivation of a symbol sequence $\overline{X} \in (\Sigma \cup N)*$ is a reflexive transitive closure of the function *apply-rule*. *apply-rule* first chooses a non-terminal symbol A from a given symbol sequence $\overline{X}$. It then chooses $\overline{Z}$ from the set of $\mathit{delta}(A)$. Finally, it substitutes $\overline{Z}$ for the symbol A chosen at the beginning. A symbol sequence $\overline{X} \in \mathit{derivation}_{\mathrm{G}}(A_{init})$ is a sentence of the language $\mathrm{L}(G)$ if it contains no non-terminal symbol. We write $\mathrm{L}_{\mathrm{G}}(A)$ to denote the set of sentences derived from a non-terminal symbol A of a grammar G.

Figure 3 shows an example of grammar *SQL*, which describes the grammar of a fluent interface for a subset of SQL's select queries. $\mathrm{L}(SQL)$ is a finite set of the following three elements:

- `.select(String selectList).from(String tableSource)`
- `.select(String selectList).from(String tableSource)` `.where(String searchCondition)`
- `.select(String selectList).from(String tableSource)` `.where().column(String columnSpec).equals()` `.value(String value)`

$SQL = \langle \Sigma_{SQL}, N_{SQL}, \delta_{SQL}, \mathit{SelectQuery} \rangle$
where
$\Sigma_{SQL} = \{$
`.select(String selectList),`
`.from(String tableSource),`
`.where(String searchCondition),`
`.where(),`
`.column(String columnSpec),`
`.equals(),`
`.value(String value)`
$\}$
$N_{SQL} = \{\mathit{SelectQuery}, \mathit{WhereClause}\}$
$\delta_{SQL}(\mathit{SelectQuery}) = \{$
`.select(String selectList)`
`.from(String tableSource)` *WhereClause*
$\}$
$\delta_{SQL}(\mathit{WhereClause}) = \{$
ϵ,
`.where(String searchCondition),`
`.where().column(String columnSpec)`
`.equals().value(String value)`
$\}$

Figure 3. A grammar for select queries, *SQL*.

For instance, the following method chain is valid since the sequence of method signatures is derivable from the grammar *SQL*:

```
SQL.select("*").from("sample.db")
    .where("sample_id = 1001").end()
```

3.2 Grammar rewriting function f

We below formally define f, the rewriting function that adds a terminal symbol to the right-hand side of each rule:

$$f(G) = \langle \Sigma', N, \delta', A_{init} \rangle,$$

where

$$\Sigma' = \Sigma \cup \{\mathit{subchain}(A) \mid A \in N\} \text{ and}$$
$$\delta'(A) = \{\mathit{subchain}(A)\} \cup \delta(A).$$

$\mathit{subchain}(A)$ is a special terminal symbol corresponding to a non-terminal symbol A. It denotes the method signature of a method for receiving a sub-chain as an argument. The code generating function *codegen* generates a method named A for $\mathit{subchain}(A)$.

$f(SQL) = \langle \Sigma'_{SQL}, N_{SQL}, \delta'_{SQL}, \mathit{SelectQuery} \rangle$
where
$\Sigma'_{SQL} = \{$
`.select(String selectList),`
`.from(String tableSource),`
`.where(String searchCondition),`
`.where(),`
`.column(String columnSpec),`
`.equals(),`
`.value(String value),`
`.selectQuery(SelectQuery subChain),`
`.whereClause(WhereClause subChain)`
$\}$
$N_{SQL} = \{$`SelectQuery, WhereClause`$\}$
$\delta'_{SQL}(\mathit{SelectQuery}) = \{$
`.selectQuery(SelectQuery subChain),`
`.select(String selectList)`
`.from(String tableSource)` *WhereClause*
$\}$
$\delta'_{SQL}(\mathit{WhereClause}) = \{$
`.whereClause(WhereClause subChain),`
ϵ,
`.where(String searchCondition),`
`.where() .column(String columnSpec)`
`.equals() .value(String value)`
$\}$

Figure 4. The grammar after rewriting, $f(SQL)$.

We assume that $\mathit{subchain}(A)$ is a unique terminal symbol. This is not an unrealistic assumption. The method for $\mathit{subchain}(A)$ does not conflict with the one for $\mathit{subchain}(B)$ since their method name is not the same. The method for $\mathit{subchain}(A)$ does not conflict with the method for a terminal symbol either *in realistic cases* since the names of terminal symbols and non-terminal symbols are often different. Even if a grammar contains both a terminal symbol named A and non-terminal symbol A, the method overloading mechanism helps us avoid this naming conflict in most cases. The exception is a case where a grammar contains a terminal symbol named A that takes a sub-chain as its argument. In this case, the method for $\mathit{subchain}(A)$ *does* conflict with the method for that terminal. However, we do not think that such a case appears in a realistic situation.

$\mathit{Paren} = \langle \Sigma_{\mathit{Paren}}, N_{\mathit{Paren}}, \delta_{\mathit{Paren}}, S \rangle$
where
$\Sigma_{\mathit{Paren}} = \{$`.a(), .b(), .c()`$\}$
$N_{\mathit{Paren}} = \{S\}$
$\delta_{\mathit{Paren}}(S) = \{$
`.a()` S `.c()`,
`.b()`
$\}$

Figure 5. An LL(1) grammar for `.a()`^{n} `.b()` `.c()`^{n}.

Figure 4 shows $f(SQL)$, the grammar obtained after rewriting SQL by f. Both `.selectQuery(SelectQuery subChain)` and `.whereClause(WhereClause subChain)` are the terminal symbols added when rewriting by f, and they correspond to $\mathit{subchain}(\mathit{SelectQuery})$ and $\mathit{subchain}(\mathit{WhereClause})$, respectively. Here, we assume that *subchain* changes a given non-terminal symbol to a lower camel case and use it as a method name. The fluent interface generated by *codegen* from the grammar $f(SQL)$ allows method chaining in the sub-chaining style. For example,

```
SQL.select("*").from("sample.db")
    .whereClause(
        WhereClause.column("sample_id")
            .equals().value("1001")
    ).end();
```

3.3 Overview of safe fluent interface generator

We below outline how we implement a generator for safe fluent interfaces so that it can accept any LL(1) grammar. The generator takes an LL(1) grammar as its input and it first converts the grammar to a single-state realtime deterministic pushdown automaton. Since this automaton is equivalent to an LL(1) parsing table, we can construct the automaton by the same way as the LL(1) parsing algorithm, for example as described in the literature [1, 5]. This automaton recognizes the language derived from the grammar.

Suppose that the grammar shown in Figure 5 is given to the generator as an input. The language derived from this grammar contains sequences of method signatures such that they start with zero or more iterations of `.a()`, followed by `.b()`, and by as many `.c()` as `.a()`. Figure 6 shows a single-state realtime deterministic pushdown automaton that recognizes this language.

Once the automaton is obtained, the generator generates a generic class for every stack symbol. It represents a stack. For example, a class type `S<Rest>` generated for a stack symbol S represents the stack where the stack top symbol is S and the rest of the stack elements is represented by a type parameter `Rest`. A transition rule is represented by a

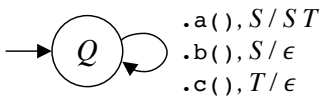


Figure 6. A single-state realtime deterministic pushdown automaton that recognizes `.a()`^{n} `.b()` `.c()`^{n}.

```
class S<Rest> {
    S<T<Rest>> a();
    Rest b();
}
class T<Rest> {
    Rest c();
}
class Bottom { Result end(); }
S<Bottom> begin();  // entry point
```

Listing 2. Generated fluent interface skeleton.

method in that generic class. Since a single-state realtime deterministic pushdown automaton has only a single trivial state and performs no ϵ-transition, a transition is modeled as a function from an input symbol and a stack symbol into stack symbols. Suppose that, when the next input symbol is `.a()` and the stack top symbol is S, the automaton pops the stack top and pushes ST. For this transition, a method `.a()` is declared in the class `S<Rest>` generated for the stack symbol S. The return type of the method `.a()` is `S<T<Rest>>`. Note that `S<Rest>` and `S<T<Rest>>` represent the current and new stacks, respectively.

Listing 2 shows a code skeleton of the fluent interface generated from the automaton shown in Figure 6. We omit method bodies. If a call chain to the methods in Listing 2 is typable and hence valid, this sequence of method signatures is also accepted by the automaton. A valid method call is specified by a combination of method signatures and receiver types. The receiver type of the next method call is the return type of the current method call. Similarly, a valid transition for the automaton is specified by a combination of input symbols and stack top symbols. The next stack top is pushed by the current transition. Our fluent-interface generator translates transitions into methods while preserving the validity of the sequences and chains. Note that the automaton stops when the stack is empty. An empty stack is represented by the `Bottom` class in Listing 2. Since this class does not declare any methods except for a special method `.end()`, no further method call is allowed when the receiver type is `Bottom` as the automaton stops with an empty stack. `.end()` is the special method that terminates the method call chaining and return the result.

4 Grammar Class of Our Proposal

This section discusses the grammar class that our proposed method can generate a safe fluent interface for. Since our method assumes that $codegen(g, s)$ can generate a safe fluent interface if the given grammar g is LL(1), our method can also generate a safe fluent interface from the given grammar G if $f(G)$ is an LL(1) grammar. In this section, we show that $f(G)$ is LL(1) if and only if the following two conditions are satisfied:

- G is LL(1).
- G does not include a non-terminal symbol A such that $\mathrm{L_G}(A) \subseteq \{\epsilon\}$.

As we defined in Section 3.1, $\mathrm{L_G}(A)$ is the set of symbol sequence derived from a non-terminal symbol A of a grammar G. One may think that the second condition is always satisfied when the first condition is satisfied, but there are a few exceptional cases. Those cases are discussed in Section 5.

The rest of this section is organized as follows. In Section 4.1, we formalize the LL(1) property of a grammar. Section 4.2 proves that $f(G)$ is an LL(1) grammar when G is an LL(1) grammar except for a few exceptional cases. Section 4.3 proves the reversed proposition: G is an LL(1) grammar when $f(G)$ is an LL(1) grammar.

4.1 Formalization of LL(1)

LL(1) is a class of grammars such that for any combination of a symbol sequence $\overline{x}$ and a non-terminal symbol A, we can uniquely identify the production rule that must be first applied to A to derive the sequence $\overline{x}$ from only the first letter of $\overline{x}$. For a grammar G to be LL(1), it must satisfy the following conditions:

$$\forall A \in N,\ \overline{X}, \overline{Y} \in \delta(A).\quad \overline{X} \neq \overline{Y} \implies \mathit{first}^{\Sigma}_{\mathrm{G}}(\overline{X}) \cap \mathit{first}^{\Sigma}_{\mathrm{G}}(\overline{Y}) = \phi \tag{1}$$

$$\forall A \in N.\quad \mathit{nullable}_{\mathrm{G}}(A) \implies \mathit{first}^{\Sigma}_{\mathrm{G}}(A) \cap \mathit{follow}^{\Sigma}_{\mathrm{G}}(A) = \phi \tag{2}$$

$$\forall A \in N,\ \overline{X}, \overline{Y} \in \delta(A).\quad \overline{X} \neq \overline{Y} \implies \text{not } (\mathit{nullable}_{\mathrm{G}}(\overline{X}) \text{ and } \mathit{nullable}_{\mathrm{G}}(\overline{Y})) \tag{3}$$

where *nullable*, *first*, *follow* are functions such that:

$$\mathit{nullable}_{\mathrm{G}}(\overline{X}) = \epsilon \in \mathit{derivation}_{\mathrm{G}}(\overline{X})$$

$$\mathit{first}^{*}_{\mathrm{G}}(\overline{X}) = \{Y \mid Y\overline{Z} \in \mathit{derivation}_{\mathrm{G}}(\overline{X})\}$$

$$\mathit{first}^{\Sigma}_{\mathrm{G}}(\overline{X}) = \mathit{first}^{*}_{\mathrm{G}}(\overline{X}) \cap \Sigma$$

$$\mathit{first}^{\mathrm{N}}_{\mathrm{G}}(\overline{X}) = \mathit{first}^{*}_{\mathrm{G}}(\overline{X}) \cap N$$

$$\mathit{follow}^{*}_{\mathrm{G}}(A) = \{X \mid \exists B \in N.\overline{Y}\,A\,X\,\overline{Z} \in \mathit{derivation}_{\mathrm{G}}(B)\}$$

$$\mathit{follow}^{\Sigma}_{\mathrm{G}}(A) = \mathit{follow}^{*}_{\mathrm{G}}(\overline{X}) \cap \Sigma$$

$$\mathit{follow}^{\mathrm{N}}_{\mathrm{G}}(A) = \mathit{follow}^{*}_{\mathrm{G}}(\overline{X}) \cap N$$

$\mathit{nullable}_{\mathrm{G}}(A)$ is a boolean function that denotes whether the given non-terminal symbol A can derive an empty string

under the grammar G. $first_{\mathrm{G}}(\overline{X})$ denotes the first set; a set of the first symbols Y of the symbol sequences $Y\overline{Z}$ derived from $\overline{X}$ under the grammar G. Since we are interested in not only the first terminal symbols but also the first non-terminal symbols, we add a superscript (either of Σ, N, $*$) to denote what kind of symbols to be included in the first set (terminal-symbols, non-terminal symbols, or both of them, respectively). $follow_{\mathrm{G}}(A)$ denotes the follow set; a set of the symbols following the given symbol A in some symbol sequence of the language $L(G)$. Superscripts denote the same as on *first*.

We can describe $nullable_{\mathrm{f(G)}}$, $first_{\mathrm{f(G)}}$, and $follow_{\mathrm{f(G)}}$ for the grammar $f(G)$ by using the corresponding sets for the grammar G.

$$nullable_{\mathrm{f(G)}}(\overline{X}) = \begin{cases} nullable_{\mathrm{G}}(\overline{X}) & (\overline{X} \in (\Sigma \cup N)*) \\ \bot & (\text{otherwise}) \end{cases} \tag{4}$$

$$first^{*}_{\mathrm{f(G)}}(\overline{X}) = \begin{cases} first^{*}_{\mathrm{G}}(\overline{X}) \\ \quad \cup \{subchain(A) \mid A \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})\} \\ \qquad (\overline{X} \in (\Sigma \cup N)*) \\ first^{*}_{\mathrm{f(G)}}(\overline{Y_1}) \cup \{subchain(A)\} \\ \qquad (\overline{X} = \overline{Y_1}\ subchain(A)\ \overline{Y_2} \\ \qquad\quad \text{and}\ \ nullable_{\mathrm{f(G)}}(\overline{Y_1})) \\ first^{*}_{\mathrm{f(G)}}(\overline{Y_1}) \\ \qquad (\overline{X} = \overline{Y_1}\ subchain(A)\ \overline{Y_2} \\ \qquad\quad \text{and not}\ \ nullable_{\mathrm{f(G)}}(\overline{Y_1})) \end{cases} \tag{5}$$

$$follow^{*}_{\mathrm{f(G)}}(A) = follow^{*}_{\mathrm{G}}(A) \cup \{subchain(B) \mid B \in follow^{\mathrm{N}}_{\mathrm{G}}(A)\} \tag{6}$$

Since $\overline{X}$ may include $subchain(A)$ for some non-terminal symbol A, which is introduced by f and is not a terminal symbol in G, the formulas above describe such a case separately. The nullability of a non-terminal symbol does not change at all before or after applying f. It is always false when $\overline{X}$ includes $subchain(A)$, since $subchain(A)$ is a terminal symbol in $f(G)$. Similarly, the first sets do not change by f except that a terminal symbol $subchain(A)$ is added. The formula for *first* is divided into three cases. In the latter two cases, the parameter $\overline{X}$ includes a terminal symbol $subchain(A)$. In the first case, $\overline{X}$ does not include it. We list the three cases here to provide the complete definition of *first*, but we use only the first case in our proof in Section 4.2.1 and Section 4.2.2. The first case describes that $subchain(A)$ is added iff the non-terminal symbol A is included in the first set $first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})$ under the grammar G before rewriting. The follow sets change in the same way. There is no case separation for the follow sets since the parameter to *follow* is a non-terminal symbol A and it may not be $subchain(B)$.

4.2 Is $f(G)$ LL(1) when G is LL(1)?

We first discuss whether $f(G)$ is LL(1) or not under an assumption that G is LL(1). Again, the proposition *"a grammar G is LL(1)"* means that all the three conditions (1), (2), and (3) are satisfied. We found that if a grammar G satisfies the three conditions, the rewriting function f keeps all these conditions in most cases. We below will look at each case in turn.

4.2.1 Derivation of (1).

Assume $A \in N, \overline{X}, \overline{Y} \in \delta(A)$ such that $\overline{X} \neq \overline{Y}$. We first discuss the case where $\overline{X}$ or $\overline{Y}$ includes $subchain(B)$ for some B. In this case, by the definition of f, either $\overline{X} = subchain(A)$ or $\overline{Y} = subchain(A)$ holds although $\overline{X} = \overline{Y} = subchain(A)$ never holds. In the case where $\overline{X} = subchain(A)$ and $\overline{Y} \neq subchain(A)$, $first^{\Sigma}_{\mathrm{f(G)}}(subchain(A)) = \{subchain(A)\}$. Since $\overline{Y} \in (\Sigma \cup N)*$ and $first^{\Sigma}_{\mathrm{G}}(\overline{Y})$ does not include $subchain(A)$, $first^{\Sigma}_{\mathrm{f(G)}}(\overline{Y})$ does not include $subchain(A)$ unless $A \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{Y})$. In other words, $first^{\Sigma}_{\mathrm{f(G)}}(\overline{X}) \cap first^{\Sigma}_{\mathrm{f(G)}}(\overline{Y}) = \phi$ unless G is left recursive (recall $\overline{Y} \in \delta(A)$). A left recursive grammar is LL(1) only if the non-terminal symbol A, the cause of the left recursion, satisfies $\mathrm{L_G}(A) = \phi$. Therefore, the condition (1) holds for $f(G)$ except for this case. We will show a concrete example of this case in Section 5.

We next discuss the remaining case. In this case, $\overline{X}, \overline{Y} \in (\Sigma \cup N)*$. By the first case of formula (5), $first^{\Sigma}_{\mathrm{f(G)}}(\overline{X}) = first^{\Sigma}_{\mathrm{G}}(\overline{X}) \cup \{subchain(A) \mid A \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})\}$. This is the same for $\overline{Y}$. Thus, we can transform the given expression as follows:

$$first^{\Sigma}_{\mathrm{f(G)}}(\overline{X}) \cap first^{\Sigma}_{\mathrm{f(G)}}(\overline{Y}) =$$

$$(first^{\Sigma}_{\mathrm{G}}(\overline{X}) \cap first^{\Sigma}_{\mathrm{G}}(\overline{Y})) \tag{7}$$

$$\cup\ (\{subchain(A) \mid A \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})\} \cap \{subchain(B) \mid B \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{Y})\}) \tag{8}$$

$$\cup\ (first^{\Sigma}_{\mathrm{G}}(\overline{X}) \cap \{subchain(B) \mid B \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{Y})\}) \tag{9}$$

$$\cup\ (\{subchain(A) \mid A \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})\} \cap first^{\Sigma}_{\mathrm{G}}(\overline{Y})) \tag{10}$$

Since $subchain(A) \notin \Sigma$ for any non-terminal symbol A, both (9) and (10) are empty. (7) is also empty because of the assumption that G is LL(1), .

The remaining (8) is also empty in most cases, but there is an exception. Assume that there is a terminal symbol t_1 such that $t_1 \in \{subchain(A) \mid A \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})\} \cap \{subchain(B) \mid B \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{Y})\}$. t_1 must be of the form $subchain(C)$ for a non-terminal symbol C, and C satisfies both $C \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{X})$ and $C \in first^{\mathrm{N}}_{\mathrm{G}}(\overline{Y})$. A non-terminal symbol is an intermediate state in the derivation process and it will be replaced later with another symbol sequence. Thus, if there exists a terminal symbol t_2 such that $t_2 \in first^{\Sigma}_{\mathrm{G}}(C)$, t_2 must be included in both $first^{\Sigma}_{\mathrm{G}}(\overline{X})$ and $first^{\Sigma}_{\mathrm{G}}(\overline{Y})$. Since this violates the assumption that G is LL(1), no terminal symbol $t_2 \in first^{\Sigma}_{\mathrm{G}}(C)$ exists.

Therefore, if C exists, $first^{\Sigma}_{\mathrm{G}}(C) = \phi$. This is equivalent to $\mathrm{L_G}(C) \subseteq \{\epsilon\}$. (8) is empty unless G includes such C. We will show an concrete example of the exception in Section 5.

4.2.2 Derivation of (2). If G is LL(1) and there is no non-terminal symbol A such that $\mathrm{L_G}(A) \subseteq \{\epsilon\}$, then the condition (2) is satisfied for $f(G)$. Let A be a non-terminal symbol such that $nullable_{\mathrm{f(G)}}(A)$. Obviously, $A \in (\Sigma \cup N)*$. By (5) and (6), we can transform the given expression as follows:

$$
\begin{aligned}
&first^{\Sigma}_{\mathrm{f(G)}}(A) \cap follow^{\Sigma}_{\mathrm{f(G)}}(A) = \\
&\quad (first^{\Sigma}_{\mathrm{G}}(A) \cap follow^{\Sigma}_{\mathrm{G}}(A)) && (11)\\
&\quad \cup (\{subchain(B) \mid B \in first^{\mathrm{N}}_{\mathrm{G}}(A)\} \\
&\qquad \cap \{subchain(C) \mid C \in follow^{\mathrm{N}}_{\mathrm{G}}(A)\}) && (12)\\
&\quad \cup (first^{\Sigma}_{\mathrm{G}}(A) \\
&\qquad \cap \{subchain(C) \mid C \in follow^{\mathrm{N}}_{\mathrm{G}}(A)\}) && (13)\\
&\quad \cup (\{subchain(B) \mid B \in first^{\mathrm{N}}_{\mathrm{G}}(A)\} \\
&\qquad \cap follow^{\Sigma}_{\mathrm{G}}(A)) && (14)
\end{aligned}
$$

By the same reasoning as in section 4.2.1, we can conclude that (11), (13), and (14) are empty and (12) is empty unless there exists a non-terminal symbol C such that $\mathrm{L_G}(C) \subseteq \{\epsilon\}$.

4.2.3 Derivation of (3). By the formula (4), a symbol sequence $\overline{X}$ is nullable under $f(G)$ if and only if $\overline{X} \in (\Sigma \cup N)$ and $nullable_{\mathrm{G}}(\overline{X})$ under G. Thus, equation (3) is true under $f(G)$ as well as it is true under G.

4.3 Is G LL(1) when $f(G)$ is LL(1)?

G is LL(1) when $f(G)$ is LL(1). For simplicity, we below discuss the contraposition: $f(G)$ is not LL(1) when G is not LL(1). Here, *"a grammar is not LL(1)"* means that at least one of the three conditions (1), (2), or (3) is not satisfied. We deduce $f(G)$ is not LL(1) by showing that each condition is false for $f(G)$ under the assumption that it is false for G.

4.3.1 Assume (1) to be false. For $\overline{X}$ and $\overline{Y}$ such that $\overline{X}, \overline{Y} \in \delta(A)$, $first^{\Sigma}_{\mathrm{G}}(\overline{X}) \cap first^{\Sigma}_{\mathrm{G}}(\overline{Y}) \neq \phi$. $\overline{X}$ and $\overline{Y}$ are symbol sequences in G. So, both $\overline{X} \in (\Sigma \cup N)*$ and $\overline{Y} \in (\Sigma \cup N)*$ hold. By the formula (5), both $first^{\Sigma}_{\mathrm{f(G)}}(\overline{X}) \supseteq first^{\Sigma}_{\mathrm{G}}(\overline{X})$ and $first^{\Sigma}_{\mathrm{f(G)}}(\overline{Y}) \supseteq first^{\Sigma}_{\mathrm{G}}(\overline{Y})$ hold. Hence, $first^{\Sigma}_{\mathrm{f(G)}}(\overline{X}) \cap first^{\Sigma}_{\mathrm{f(G)}}(\overline{Y}) \supseteq first^{\Sigma}_{\mathrm{G}}(\overline{X}) \cap first^{\Sigma}_{\mathrm{G}}(\overline{Y}) \neq \phi$. $f(G)$ is not LL(1) since it does not satisfy the condition (1).

4.3.2 Assume (2) to be false. By the assumption, there exists a non-terminal symbol A such that $nullable_{\mathrm{G}}(A)$ and $first^{\Sigma}_{\mathrm{G}}(A) \cap follow^{\Sigma}_{\mathrm{G}}(A) \neq \phi$. By the formulas (4), (5), and (6), we deduce $nullable_{\mathrm{f(G)}}(A)$, $first^{\Sigma}_{\mathrm{f(G)}}(A) \supseteq first^{\Sigma}_{\mathrm{G}}(A)$, and $follow^{\Sigma}_{\mathrm{f(G)}}(A) \supseteq follow^{\Sigma}_{\mathrm{G}}(A)$. Thus, we conclude that $f(G)$ is not LL(1) by the same reasoning in Section 4.3.1.

4.3.3 Assume (3) to be false. By the assumption, there exists a non-terminal symbol A and symbol sequences $\overline{X}$ and $\overline{Y}$ such that $\overline{X}, \overline{Y} \in \delta(A)$ and $nullable_{\mathrm{G}}(\overline{X})$ and $nullable_{\mathrm{G}}(\overline{Y})$. By the formula (4), $nullable_{\mathrm{f(G)}}(\overline{X})$ and $nullable_{\mathrm{f(G)}}(\overline{Y})$ hold whenever $nullable_{\mathrm{G}}(\overline{X})$ and $nullable_{\mathrm{G}}(\overline{Y})$ hold. Thus, $f(G)$ is not LL(1).

5 Exceptional Cases

As we show in the previous section, $f(G)$ is not LL(1) if G is LL(1) but it includes a non-terminal symbol A such that $\mathrm{L_G}(A) \subseteq \{\epsilon\}$. This section shows three examples of this exception. Each example represents a different type of exception. In all the three examples, the grammar includes at least one non-terminal symbol that does not derive any symbol sequence except an empty sequence.

The first example is G_1.

$$
G_1 = \langle \phi, \{S\}, \delta_1, S \rangle \quad \text{where} \quad \left\{ \begin{array}{l} \delta_1(S) = \{S\} \end{array} \right.
$$

G_1 is a grammar representing a left-recursive but LL(1) language. Although this language includes no symbol sequences, G_1 satisfies all the three conditions (1), (2), and (3), for LL(1). However, $f(G_1)$ does not satisfy the condition (1). $f(G_1)$ is:

$$
f(G_1) = \langle \{subchain(S)\}, \{S\}, \delta'_1, S \rangle \quad \text{where} \quad \left\{ \begin{array}{l} \delta'_1(S) = \{subchain(S), S\} \end{array} \right.
$$

and hence $first^{\Sigma}_{\mathrm{f(G_1)}}(subchain(S)) = first^{\Sigma}_{\mathrm{f(G_1)}}(S) = \{ subchain(S)\}$.

The second example is G_2. It is a finite language consisting of only two terminal symbols a and b.

$$
G_2 = \langle \{a, b\}, \{S, A\}, \delta_2, S \rangle \quad \text{where} \quad \left\{ \begin{array}{l} \delta_2(S) = \{A\,a, A\,b\} \\ \delta_2(A) = \{\epsilon\} \end{array} \right.
$$

G_2 is an LL(1) grammar, for example, it satisfies the condition (1). $\delta_2(S)$ includes only two derivations and their first sets have no common element. $\delta_2(A)$ includes only one derivation. Since there is only one first set for $\delta_2(A)$, the condition (1) holds. Note that $first^{\Sigma}_{\mathrm{G_2}}(A\,a) = \{a\}$ and $first^{\Sigma}_{\mathrm{G_2}}(A\,b) = \{b\}$. However, $f(G_2)$ violates the condition (1) since $first^{\Sigma}_{\mathrm{f(G_2)}}(A\,a)$ and $first^{\Sigma}_{\mathrm{f(G_2)}}(A\,b)$ share a terminal symbol $subchain(A)$.

The third example is G_3. It represents an infinite language a^i $(i \geq 0)$.

$$
G_3 = \langle \{a, b\}, \{S, A\}, \delta_3, S \rangle \quad \text{where} \quad \left\{ \begin{array}{l} \delta_3(S) = \{aSA, A\} \\ \delta_3(A) = \{\epsilon\} \end{array} \right.
$$

G_3 is an LL(1) grammar and it satisfies the condition (2). Both $nullable_{\mathrm{G_3}}(S)$ and $nullable_{\mathrm{G_3}}(A)$ hold. Since $follow^{\Sigma}_{\mathrm{G_3}}(S) = follow^{\Sigma}_{\mathrm{G_3}}(A) = \phi$, these follow sets do not share any elements

with their first sets. However, $f(G_3)$ does not satisfy the condition (2) since $nullable_{f(G_3)}(S)$ holds and $first^{\Sigma}_{f(G_3)}(S)$ and $follow^{\Sigma}_{f(G_3)}(S)$ shares a common element $subchain(A)$. The language $L(f(G_3))$ is $\{a^i\ subchain(S)\ subchain(A)^j \quad (i \geq j)\} \cup \{a^i\ subchain(A)^j \quad (i \geq j-1)\}$. This is a variant of well-known example of non-LL(1) (but LR) language.

6 Related Work

Nakamaru et al. [10] present a method to generate a fluent interface supporting the two styles of chaining. However, as we discussed in Section 2, they give only a complex procedure for the generation, and it is difficult to reveal the ability and limitation of their method. For a better generation of fluent interfaces with the two-style support, we presented yet another generating method that can be formally analyzed.

While the subchaining interface is useful in practice, its support is not discussed in most papers about fluent interface generation [2–4, 8, 12, 14, 15]. In this paper, we propose the formally-defined function f that *embeds* the subchaining style into the framework for generating the flat-chaining style interface. Although this paper focuses on LL(1) grammars, the function f would contribute to integrating the subchaining support into the existing techniques for LR grammars.

Nakamaru and Chiba [9] point out another problem on safe fluent interface generation that methods constituting method chaining cannot have type parameters and proposes a countermeasure. We also find this problem interesting and if we can find a solution based on grammar rewriting, as our method does, it will be easier to combine with other safe fluent interface generation methods. At the same time, however, we have found that the grammars of fluent interfaces supporting type parameters cannot be expressed in the context-free grammar used in this paper, because the acceptable language changes depending on how the type parameters are filled. Thus, a more expressive fluent interface generator may be needed as the latter step in our method to develop such a fluent interface generator.

We have adopted an extended definition of first sets (*first*) and follow sets (*follow*) from the well-known definitions presented in [1, 5] for ease of use in section 4. Although we do not consider this extension as our contribution since this kind of extensions have been discussed *ad nauseam*, the idea will be still promising when considering similar problems dealing with grammatical rewriting.

7 Conclusion

We proposed a method to generate a safe fluent interface that accepts both flat- and sub-chaining styles. Our method is based on grammar rewriting. Our method first transforms a given grammar by the rewriting function f shown in Section 3 and then generates a safe fluent interface from the grammar. The key idea is to embed sub-chaining style calls into the grammar which specifies a set of valid flat-chaining style calls. Thus, the latter step can be done by an existing method for generating a safe fluent interface supporting only the flat-chaining style.

An important feature of our method is that the grammar class it can generate a fluent interface for is proven. If there is no non-terminal symbol such that it does not derive any symbol sequences other than an empty one, our method can generate a fluent interface for any LL(1) grammars. Our proof is shown in Section 4.

There are few LL(1) grammars our method cannot generate a fluent interface for. Such exceptional grammars are shown in Section 5. All such grammars include at least one non-terminal symbol such that no sequence is derived from it, or if a sequence is derived, it is an empty one. The presence or absence of such a non-terminal symbol does not change the language derived. It would not be a problem for library developers to remove them.

Since we considered it important to reveal the grammar class acceptable to our method, the generated interfaces may be redundant. For example, [10] shows that it is possible to improve generated interfaces by merging a method that receives a sub-chain with its previous method if the previous method takes no argument. We would be able to incorporate such interface improvements into our method by considering them as grammar rewriting. However, the discussion in Section 4 does not consider such rewriting. We must revisit the grammar class acceptable to the method when incorporating such additional functionality into our method.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Number JP20H00578.

References

[1] Alfred Aho, Jeffrey Ullman, Ravi Sethi, and Monica Lam. 2006. *Compilers: Principles, Techniques, and Tools* (2 ed.). Addison Wesley.

[2] Yossi Gil and Tomer Levy. 2016. Formal Language Recognition with the Java Type Checker. In *Proceedings of 30th European Conference on Object-Oriented Programming*. https://doi.org/10.4230/LIPIcs.ECOOP.2016.10

[3] 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

[4] Radu Grigore. 2017. Java Generics Are Turing Complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. https://doi.org/10.1145/3009837.3009871

[5] Dick Grune, Kees van Reeuwijk, Henri E. Bal, Ceriel J.H. Jacobs, and Koen Langendoen. 2012. *Modern Compiler Design* (2 ed.). Springer New York, NY. https://doi.org/10.1007/978-1-4614-4699-6

[6] jMock 2000. jMock - An Expressive Mock Object Library for Java. http://jmock.org/.

[7] jOOQ 2009. jOOQ: The easiest way to write SQL in Java. https://www.jooq.org/.

[8] Tomer Levy. 2017. *A Fluent API for Automatic Generation of Fluent APIs in Java*. Master's thesis. Israel Institute of Technology.

[9] Tomoki Nakamaru and Shigeru Chiba. 2020. Generating a Generic Fluent API in Java. *The Art, Science, and Engineering of Programming* 4, 3 (2020), 9. https://doi.org/10.22152/programming-journal.org/2020/4/9

[10] 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*. https://doi.org/10.1145/3136040.3136041

[11] Tomoki Nakamaru, Tomomasa Matsunaga, Tetsuro Yamazaki, Soramichi Akiyama, and Shigeru Chiba. 2020. An Empirical Study of Method Chaining in Java. In *Proceedings of the 17th International Conference on Mining Software Repositories* (Seoul, Republic of Korea) *(MSR '20)*. Association for Computing Machinery, New York, NY, USA, 93–102. https://doi.org/10.1145/3379597.3387441

[12] Ori Roth. 2021. Study of the Subtyping Machine of Nominal Subtyping with Variance. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 137 (oct 2021), 27 pages. https://doi.org/10.1145/3485514

[13] StreamAPI 2014. Java Platform Standard Edition 8 Documentation - `interface Stream<T>`. https://docs.oracle.com/javase/jp/8/docs/api/java/util/stream/Stream.html.

[14] Hao Xu. 2010. *EriLex: An Embedded Domain Specific Language Generator*. Springer-Verlag, 192–212. https://doi.org/10.1007/978-3-642-13953-6_11

[15] 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 (oct 2019), 24 pages. https://doi.org/10.1145/3360560