Department of Creative Informatics
Graduate School of Information Science and Technology
THE UNIVERSITY OF TOKYO

Master's Thesis

A Study on a Translation Method from BNF-Style Grammars to Fluent Embedded Domain-Specific Languages with Subchaining

BNF 形式の文法から fluent な内部ドメイン固有言語への
変換手法の研究

Tomoki Nakamaru
中丸 智貴

Supervisor: Professor Shigeru Chiba

January 2018

Abstract

This thesis 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 thrown 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 thesis 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.

概要

本研究では, BNF 形式の文法から fluent な内部ドメイン特化言語 (fluent embedded domain-specific language, fluent EDSL) を生成するツール Silverchain を提案する. Fluent EDSL とは, ドメイン固有言語の文を汎用言語中にメソッドチェインとして埋め込むことを可能にするクラスライブラリのことである. 生成される fluent EDSL は, 不正な構文を型エラーとして検出でき, 状況に応じてメソッドチェインを効果的に使えるように設計されている. 第一の特徴は各メソッドの型を次に呼べるメソッドに応じて設定することで実現され, 第二の特徴は通常のチェイン API に合わせてサブチェイン API を提供することで達成される. 本研究の貢献は, BNF 形式の文法を前述の 2 つの特徴を持つ fluent EDSL に変換する手法を開発したことである. 提案する変換は single-state real-time deterministic pushdown automata (RPA) の構築としてモデル化される. Fluent EDSL は, それらの RPA をクラス定義にエンコードすることによって得られる. 本研究で用いる RPA の構築手法では, 与えられた文法へ非終端子を追加または削除をしない. この制限は, サブチェイン API を文法で定義された通りに生成するために必要な条件である.

Contents

Chapter 1 Introduction 1

Chapter 2 Background 4
- 2.1 Fundamental Background Topics 4
- 2.2 Motivation 7
- 2.3 Related Work 10
- 2.4 Summary 15

Chapter 3 Silverchain 16
- 3.1 RPA and Its Encoding 18
- 3.2 RPA Construction 20
- 3.3 Preprocessing 27
- 3.4 Limitation 29
- 3.5 Details of Encoding 30

Chapter 4 Use Cases 34

Chapter 5 Conclusion 38
- 5.1 Summary 38
- 5.2 Future Work 38

Publications 40

Acknowledgements 41

References 42

Chapter 1

Introduction

Domain-specific languages (DSLs) have been gaining popularity among both the programmers in industry and the researchers in software engineering. A DSL is a programming language that is designed to write the solutions or the expressions in its specific domain as simply as possible. Many DSLs are used in practice such as regular expressions for searching patterns in text documents, SQL [9] for handling relational data, and DOT [10] for drawing graphs. The simplicity achieved by DSLs lets programmers read and write code quickly and improves their productivity. This advantage is becoming more important these days because programmers need to be highly productive as the scale and the complexity of a software application gets larger and higher.

Many technologies to embed a DSL into a general-purpose language (GPL) has been proposed these decades. A GPL is a programming language that is not specialized to any specific domain such as C, Java, and Python. Since a DSL has the features and the syntax only for its specific domain, most of software applications can not be built only with DSLs. A GPL usually accounts for a large part of a software application, and DSLs are used to write code for relatively smaller parts of the application. The easiest embedding method is to write a DSL sentence as a string literal of a GPL and pass the literal to the execution system of the DSL as follows:

```
executeQuery("SELECT * FROM user WHERE id = 1");
```

This way of embedding works but does not provide good user experiences because the system causes runtime errors when passing invalid DSL sentences. Many embedding technologies have been proposed to address this issue of safety.

Creating an embedded DSL (EDSL) is one of the most popular technologies that allows programmers to use a DSL inside GPL program in a safer way [19]. An EDSL is a class library of a host GPL that makes it possible to emulate DSL sentences in a GPL program. Since an EDSL is just a library, its users do not have to learn new syntax and can get support from existing tools for the host GPL such as an integrated development environment (IDE). The developers of an EDSL also receive benefits. They do not need to create a compiler nor an editor just for a DSL. By embedding a DSL as an EDSL, the productivity improvement in application development is achieved with small overhead costs.

Method chaining has been drawing attention as a way of designing EDSLs. By using method chaining, sentences of a DSL can be embedded into a GPL program without changing the basic syntax of the DSL. 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 when it is designed to emulate sentences of a language as shown in the example above [12]. In this thesis, we refer to a class library emulating a DSL by method chaining as a fluent embedded DSL (fluent EDSL). jOOQ [14] and j2html [22] are examples of fluent EDSLs in Java that a number of programmers use in practice.

Several techniques have been studied to detect syntactic errors of embedded sentences as type errors of the host language [12, 2, 34]. Such errors indicate an arrangement of method calls is incorrect from the viewpoint of the grammar of a DSL, for example as follows:

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

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()`. This design of EDSLs improves user experiences especially when the host GPL has a static type system. Its users can find misuses of an EDSL from thrown type errors at compile-time, and method-completion of an IDE can show the list of necessary and sufficient candidates because it usually uses type information to narrow down that list.

Implementing such syntax-checking is a tedious task for the developers of a fluent EDSL. The developers need to define a large number of classes and relate them by choosing an appropriate return type for each method. This task is almost impossible to accomplish by hand because the number of required classes is often more than a hundred even when the grammar seems relatively simple. In many widely-used EDSLs, the developers relax syntactic rules to reduce the number of required classes, but this workaround worsens the user experiences. The users may accidentally compose a syntactically incorrect sentence, and method-completion provides unnecessary or insufficient candidates.

To solve the developers' side of the problem, we present Silverchain, a tool that translates a BNF-style grammar into a fluent EDSL. 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 c = col("year").eq().val(2018);
select("*").from("book").where().condition(c);
```

Although this API introduces redundant parts that do not appear in the original sentence (`condition` in this case), it provides good user experiences that can not be achieved through regular chaining API, especially when composing a complex sentence or changing a part of a sentence dynamically.

The contribution of this thesis is the development of a translation method from a BNF-style grammar to a fluent EDSL with both the syntax-checking and the multi-style APIs. Our translation method is modeled as the construction of a set 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 RPAs construction method is different from known ones [18, 29] in that it does not add or remove non-terminals from given grammar. This constraint on the construction is required to generate subchaining APIs as specified in the grammar. With our method, any context-free grammar can be translated into a fluent EDSL, but the EDSLs generated from some grammars require subchaining APIs – which introduce redundant parts to an embedded sentence – to compose certain parts of sentences. This limitation is due to our construction method of RPAs.

The organization of this thesis is as follows. We first explain background topics, the motivation, and related work of our research in Chapter 2. Chapter 3 presents Silverchain, the tool we developed to generate a fluent EDSL from a BNF-style grammar. Use cases of Silverchain are shown in Chapter 4. We also discuss the practical applicability of our tool qualitatively in Chapter 4. Chapter 5 summarizes our contribution and discusses possible directions of further research.

Chapter 2

Background

A DSL is a language specialized to its specific domain. It helps programmers keep code easy to understand and reduces the development cost of a software application. However, it imposes overhead costs on both its developers and users. The developers need to create the DSL and its support tools, ant the users have to learn how to use them. An EDSL, particularly one with a fluent interface, is a popular solution to reduce those overhead costs. However, creating a fluent EDSL still costs a lot though it is much easier than creating a stand-alone language. In the past few years, many translation methods have been proposed to generate a fluent EDSL from a BNF-style grammar but they have problems from the viewpoint of practice.

2.1 Fundamental Background Topics

We describe fundamental topics on DSLs in this section before proceeding to the explanation of our motivation and related work.

2.1.1 Domain-Specific Language

A DSL is a language designed to deal with the problems in its domain as simply as possible. Regular expressions are a well-known example of such languages, which aims at searching string patterns from a text document. The following regular expression matches a string made of zero or more consecutive occurrences of letter “a” and one or more consecutive occurrences of letter “b”:

```
// Matches strings such as ab, aabbb, and bbb
a* b+
```

SQL is another well-known example of DSLs for handling relational data. The following shows example SQL queries (SQL sentences) that retrieves records from the database named `users`:

```
SELECT * FROM users WHERE id = 1;
SELECT * FROM users WHERE age > 20;
```

Every DSL has their own syntax specialized to its domain. For example, in regular expressions, `*` and `+` indicate the repeated occurrences of the preceding letter while they are used to denote multiplication and addition in most GPLs. This reflects the fact that numeric calculation is not necessary when searching patterns in a text. In SQL, a query expresses what to be retrieved and does not describe how to retrieve records. This reflects that most operations on relational data can be described simply by such declarative notations.

```
def match_a_star_b_plus(text):
    prev = None
    for c in text:
        if prev is None:
            if prev not in (’a’, ’b’):
                return False
            prev = c
        elif prev == ’a’:
            if prev not in (’a’, ’b’):
                return False
            prev = c
        elif prev == ’b’:
            if c == ’b’:
                prev = c
            else:
                return False

    if prev != ’b’:
        return False
    else:
        return True
```

Fig. 2.1: Searching patterns using Python

To show the effect of using DSLs in more detail, consider using Python, an example of GPLs, to search string patterns in a text document. Figure 2.1 shows the Python code that checks if a given text document contains the pattern shown above by the regular expression. The code shown in the figure is obviously longer and more complex than the single line regular expression. When a pattern is more complicated, it is almost impossible to implement a matching function only with a GPL. The situation is the same in SQL. Programmers need to write many lines to do the same as a single SQL query.

Using a DSL affects not only the readability but also the speed of program execution because the DSL compiler can optimize users' code by using the domain knowledge. In contrast, the GPL compiler can only optimize the code in general way when a program is written in GPL. SQL is a good example of DSLs that executes user queries with highly developed optimization. It achieves fairly good performance even when handling relational data that contains millions of records.

Although a DSL reduces the cost to read and write application code, it requires two overhead costs of both its users and its developers. The first overhead is the learning cost imposed on its users. The users have to spend a certain amount of time since the syntax is often unique and much different from the one of a GPL. The situation gets worse when a building application uses multiple DSLs. Frequent language switching would worsen the productivity. The second overhead is the development cost of a DSL. To make a DSL actually work, a lot of effort needs to be made such as building a parser and a compiler. Further, the developers often need to write detailed documents and create many support tools such as syntax-highlighting and debugger to let programmers use the DSL in practice.

```
// SELECT * FROM user WHERE id = 1;
User.findById(1);

// UPDATE user SET name = "John" WHERE id = 1;
User u = User.findById(1);
u.set("name", "John");
u.saveIt();

// DELETE FROM user WHERE id = 1;
User u = User.findById(1);
u.delete();
```

Fig. 2.2: Usage examples of ActiveJDBC

2.1.2 Embedded Domain-Specific Language

An EDSL is a form of DSLs that is implemented as a library of a GPL. ActiveJDBC [30] is an example of EDSLs that is used to emulate SQL queries in Java. Unlike ActiveJDBC, Some EDSLs do not have any external DSLs and works completely inside a GPL such as jMock [23] for creating mock objects and Hamcrest [17] for testing values and objects. The line between ordinal libraries and EDSLs is not clear yet but many libraries can be regarded as an EDSL, an internal language for a certain domain. Writing a DSL sentence as a string literal is an embedding method that works without any overhead costs but this method is not safe:

```
String.format("SELECT * FROM users WHERE id = %d", userId);
String.format("SELECT WHERE id = %d", userId); // Incorrect query
```

The users can write syntactically incorrect sentences and occasionally make an application causes runtime errors. By creating EDSLs, the developers can control what can be written by the users and prevent runtime errors.

An EDSL provides more benefits to both its users and its developers compared to a DSL that is implemented as a stand-alone external language. Users can use a DSL without learning new syntax or how to use new tools and editors since an EDSL is just a library. Developers can create a DSL with smaller amount of efforts. They can use the type system of the host GPL so they do not need to re-define basic types such as integers. They do not have to develop new tools such as new editors just for that DSL. Further, the developers' work would be reduced by the support of the debuggers and the optimizers for the host GPL.

The interface design is important when creating an EDSL. The API of an EDSL should keep the domain-specific syntax because the syntax of a DSL is designed to fit its domain. Figure 2.2 shows usage examples of ActiveJDBC. As seen in the figure, the appearance of the expression is different from its original queries. This way of emulation might be acceptable, but often confuses programmers who know much about original SQL especially when programmers need to write more complicated queries. They need to read the documentation of an EDSL to use the EDSL even when they know how to write an expression in the original DSL.

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

Fig. 2.3: Grammar[*1]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 literal

Fig. 2.4: Usage example and three embedded forms into Java

2.2 Motivation

Using a fluent EDSL, sentences of a DSL can be emulated nicely by using method chaining inside programs written in a GPL. As an example, consider a simple DSL for writing itemized documents whose syntax is defined by the grammar shown in Figure 2.3. Figure 2.4a shows a usage example of this DSL, and Figures 2.4b, 2.4c, and 2.4d show three em-

[*1] We use `*` for zero or more occurrences of the preceding element, `+` for one or more occurrences, `?` for zero or one occurrences, and `()` to group elements.

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

(a) With subchaining

```
IncompleteChain1 c1 = begin().text("Item");
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

Fig. 2.5: Changing a part of a chain dynamically

bedded forms of that example into Java. In Figure 2.4b, 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 basic syntax helps programmers make effective use of a DSL. On the other hand, the same example is embedded without method chaining in Figure 2.4c. This form does not resemble the original sentence, and the code tends to contain more intermediate variables. In Figure 2.4d, our example is written as a string literal. Although this form keeps the basic syntax, embedded sentences become hard to read when they contain escape sequences. Furthermore, no checking is applied to embedded sentences at compile-time unless programmers use special IDE plugins [25]. Extending the host syntax is another technique to embed DSL sentences into a GPL program. By extending Java syntax, programmers can compose a DSL sentence in Java as it is, but introducing such an extension mechanism is still a research topic [11, 21]. Fluent EDSLs can be activated quickly by importing them since they are just class libraries.

The syntax of embedded sentences 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();
```

This design of EDSLs improves user experiences in two ways. Firstly, users can find misuses at compile-time because incorrectly chained calls cause a type error. Secondly, IDE method-completion can provide necessary and sufficient candidate methods because it usually uses type information to narrow down the list of candidates. Users can easily compose a syntactically correct sentence by following provided hints, and they do not have to pay attention to syntactic errors. In other words, an existing IDE for a host language also serves as an IDE for the DSL.

To take advantage of method chaining in various situations, a fluent EDSL should

```
// Each method corresponds
// to a token of the DSL.
begin()
  .text("1")
    .begin()
      .text("1.1")
        .begin()
          .text("1.1.1")
        .end()
    .end()
.end();
```

(a) Without subchaining

```
// "list" does not appear
// in the original sentence.
begin()
  .text("1").list(
    begin()
      .text("1.1").list(
        begin()
          .text("1.1.1")
        .end())
    .end())
.end();
```

(b) With subchaining

Fig. 2.6: Simple itemization

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 keep code easy to understand especially when changing a part of a sentence dynamically as shown in Figure 2.5a. Without subchaining, intermediate variables tend to be meaningless, and users cannot make good use of method chaining as shown in Figure 2.5b. Subchaining API, however, introduces redundant parts that do not appear in original sentences of a DSL. This prevents programmers from composing short or simple sentences briefly as it originally is. Figure 2.6a and 2.6b shows the chains representing the same document but the chain without subchaining resembles more the original sentence. As demonstrated, the style of chaining API that should be applied depends on the situation. Fluent EDSLs with only subchaining APIs (or only regular chaining APIs) prevent their users from making effective use of method chaining.

However, creating a well-designed fluent EDSL (a fluent EDSL with the syntax-checking and the multi-style chaining APIs) is a burdensome task for the developers of the EDSL. 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 several hundreds in 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 many widely-used EDSLs relax the rules to follow and reduce the number of classes to define. This workaround, however, allows the users of an EDSL 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" with jOOQ
// (Missing ‘FROM’ clause)
select().orderBy(BOOK.id).getSQL();

// "<div src=‘./image.png’></div>" with j2html
// (Incorrect attribute ‘src’)
```

```
div().withSrc("./image.png").render();
```

Such incomplete checking is not only unsafe but also allows method-completion to suggest unnecessary or insufficient methods for hints. 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.

2.3 Related Work

This section summarizes existing tools and algorithms that generate fluent EDSLs and discusses their problems in terms of practical use. In addition, we also describe the advantage of EDSLs compared to syntax extension mechanisms.

2.3.1 Algorithm of Gil et al.

Gil et al. proposed an algorithm to translate a BNF-style grammar to a fluent EDSL with the syntax-checking feature although it does not support subchaining APIs [13, 27]. 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 [7] and this class of languages is larger than the class that RPAs can recognize [18], the algorithm of Gil et al. can generate non-subchaining APIs for grammars that Silverchain cannot generate them for.

However, with an EDSL generated by the algorithm of Gil et al., the compilation time of a method chain grows exponentially to the length of the chain in the worst case. This is caused by the exponential growth of the size of the type at the end of a chain, as Gil et al. showed in their experiment using Java 8. 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.

To investigate the results shown in the paper by Gil et al., we performed the same experiment as theirs using Java 8, Java 9, and C++. Two versions of Java were used in our experiments since Java 9 has a new type-checking strategy [8, 32]. Figure 2.7 shows the class definitions and example chains used in our experiments. `Exp` is a worst-case model of generic types used in a fluent EDSL generated by the algorithm of Gil et al. (`Exp` in Java is actually from the experiment in their paper.) Type size produced by repeated calls of `m()` is larger than any other types produced by a fluent EDSL by Gil et al. The time for the code in Figure 2.7 is measured as the result of a 1-sized chain. 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`, `javac 9` and `clang 900.0.39.2`. We used `javac -verbose` to compile chains and extracted total compilation time from its output. We also used `clang -ftime-report` to compile chains and extracted total compilation time from the output.

Figure 2.8 shows the results of our experiment. We measured the time 25 times for each length of a chain, and the averages are shown in the figures. As seen the figures on the left, the exponential growth of compilation time occurs not only in Java 8 but also in Java 9 and C++. The figures on the right show the same result with the different horizontal axis. These figures experimentally show that compilation time increases linearly to the size of the type at the end of a chain. (Repeated calls of `m()` produces a type of size $2^{n+2} - 1$ in an n-sized method chain.) Although Gil et al. concluded that the exponential growth is caused by “a design flaw in the compiler” in their paper [13], we observe that, from the exponential growth in C++ and Java 9, checking an extremely large type essentially takes a long time. The improvement of compiler implementation is unlikely to solve this

```java
class Exp<T,S> {
    Exp<Exp<T,S>,Exp<T,S>> m() { return null; }
}

class Main {
    public static void main(String[] args) {
        Exp<Exp<Object,Object>,Exp<Object,Object>> exp
            = new Exp<Object,Object>().m();
    }
}
```

(a) In Java

```cpp
template <class T, class S> class Exp {
    Exp<Exp<T, S>, Exp<T, S> > *m() { return NULL; }
};

int main(int argc, const char * argv[]) {
    (new Exp<X, X>())->m();
    return 0;
}
```

(b) In C++

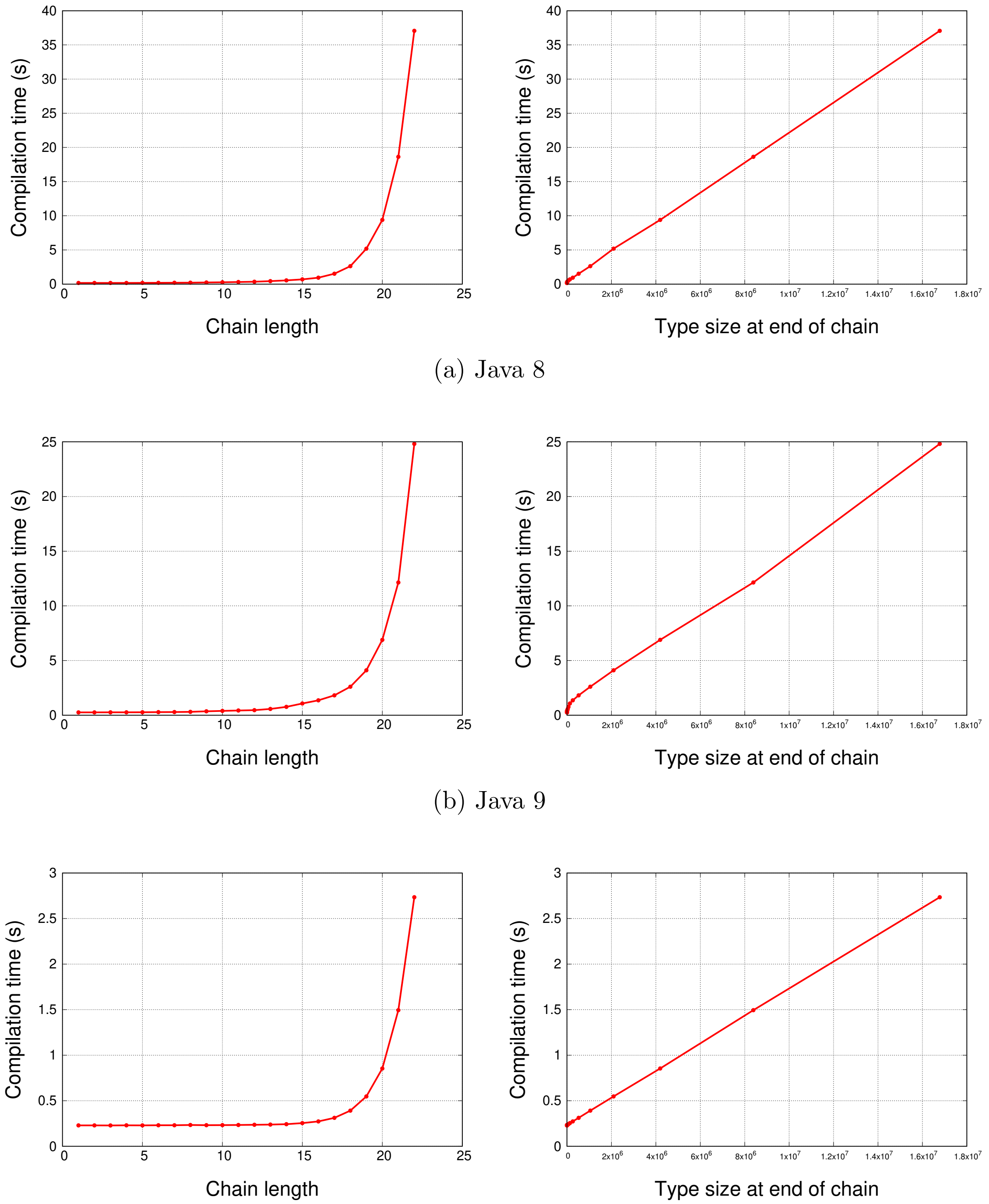
Fig. 2.7: Source code used in experiments

Table 2.1: Fitted parameters in $y = ax + b$

	a	b
Java 8	$1.73 \times 10^{-4} \pm 1.63 \times 10^{-6}$ (±0.944%)	$1.78 \times 10^{-1} \pm 4.73 \times 10^{-4}$ (±0.266%)
Java 9	$1.78 \times 10^{-4} \pm 1.87 \times 10^{-6}$ (±1.051%)	$2.66 \times 10^{-1} \pm 5.41 \times 10^{-4}$ (±0.203%)
C++	$4.14 \times 10^{-4} \pm 2.92 \times 10^{-6}$ (±0.722%)	$2.15 \times 10^{-1} \pm 8.44 \times 10^{-4}$ (±0.393%)

problem, but further investigation on the compiler implementation should be performed to find out the real cause of this problem.

To show that our encoding scheme does not cause the exponential growth, we performed the same experiment using `Lin` shown in Figure 2.9. `Lin` is a worst-case model of generic types used in a fluent EDSL generated by our method. Figure 2.10 shows the results of this experiment. The blue line in each figure is the linear regression line of data recorded in `Lin`. Fitted parameters are summarized in Table 2.1. As seen in these figures, the compilation time in `Lin` is much shorter than the time in `Exp`. Since repeated calls of `Lin.m()` produce a type of the size $n+2$ in an n-sized method chain, the compilation time increases linearly to chain sizes in `Lin`. This indicates that, with an EDSL generated by Silverchain, the size of the type at the end of a chain grows linearly even in the worst case, and the compilation time grows at most linearly to the length of a chain.



(a) Java 8

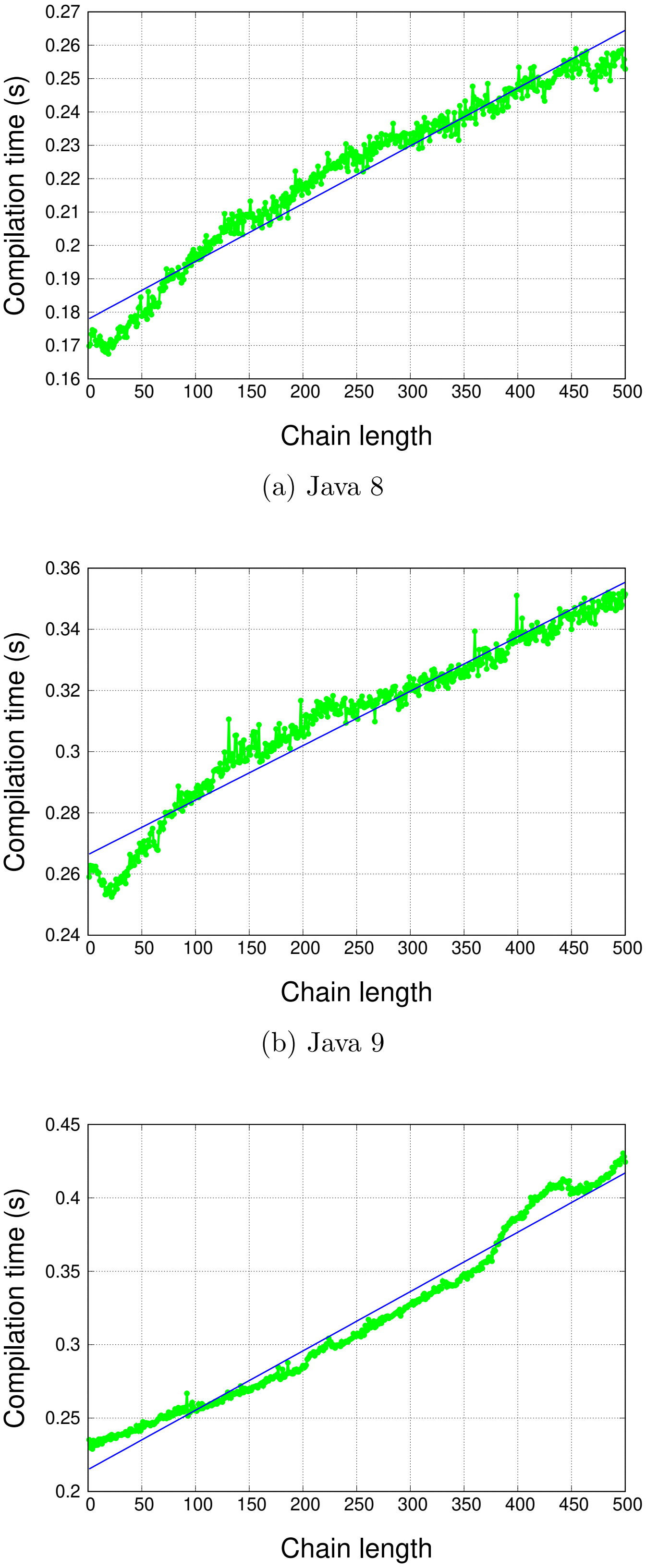
(b) Java 9

(c) C++

Fig. 2.8: Result of experiments using Exp

```
class Lin<T> {
    Lin<Lin<T>> m() { return null; }
}
```

Fig. 2.9: Model of our encoding scheme



(a) Java 8

(b) Java 9

(c) C++

Fig. 2.10: Result of experiments using `Lin`

2.3.2 Theoretical Results

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 [16] and C++ [37]. For instance, if given grammar is context-free, such an EDSL can be built by creating a CYK parser [5, 40, 24] implemented by a Turing machine emulated on a type system [16]. However, using a Turing machine on the back is overly complex for most DSLs and often causes practical problems. For example, `javac` throws a `StackOverflowError` even for a small chain of two or three methods with an EDSL generated by Grigore's algorithm.

2.3.3 EDSL Generators

ScaLALR [20] generates an EDSL only with non-subchaining APIs in Scala 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 but that the EDSL internally uses uncommon features 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 [39] 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 [15], which is a form that most of manually written grammars do not follow. Since the users of EriLex need to rewrite grammar into that form, EriLex cannot generate a subchaining style API for each non-terminal. Furthermore, there is no algorithm to rewrite grammar into the form required by EriLex as far as we know. Whether given grammar can be rewritten into that form is undecidable [33]. There are several tools known as fluent API generators such as CLARA [2], TS4J [3], and fluflu [38]. 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 [6], jOOQ [14], and j2html [22]. 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 [26] and help the developers to implement methods that have the same signature but a different return type. On the other hand, Silverchain uses type parameters to express an infinite number of states that are produced from CFG. The algorithm of Gil et al. and EriLex also use type parameters in the same way as the one of Silverchain.

2.3.4 Syntax Extension Mechanism

SugarJ [11] is a syntax extension mechanism of Java and allows programmers to write DSL sentences as they originally are. The appearance of a written sentence is much better compared to the emulation by method chaining, but the cost of introducing SugarJ is higher than introducing EDSLs. Further, the parsing time gets longer when using SugarJ since it converts written DSL sentences into plain Java sentences before compiling a program. Such a problem does not occur with EDSLs since EDSLs are just Java libraries as we mentioned earlier.

ProteaJ [21], Wyvern [28], and Honu [31] are programming languages that natively support syntax extension. However, their powerful features are realized by their underlying language mechanism such as type systems so it is hard to introduce 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.

2.4 Summary

As we have described in Section 2.1, DSLs are indispensable to build a software application with as little effort as possible. Particularly, fluent EDSLs have been gathering attentions as a way of implementing DSLs with relatively smaller costs compared to implementing DSLs as external stand-alone languages. Creating well-designed fluent EDSLs, however, still requires a lot of developers' effort. To provide the syntax-checking feature using the host type system, the developers need to a number of classes.

Many translation method from a BNF-style grammar to a fluent EDSL has been proposed to ease that developers' task, but none of them generates a fluent EDSL that provides the multi-style APIs. As we have demonstrated in Section 2.2, both regular chaining APIs and subchaining APIs should be provided in a single fluent EDSL to let EDSL users to write code in a way that fits their situation. Silverchain, the tool we propose and describe in the next chapter, generates a fluent EDSL with the multi-style APIs and the syntax-checking feature.

Chapter 3

Silverchain

Figure 3.1 shows the overview of our translation method. Silverchain first constructs a set of RPAs, each of which corresponds to a non-terminal of a given grammar. For example, Silverchain constructs one RPA for `<list>` from the grammar in Figure 2.3. A constructed RPA accepts sequences consisting of terminals and non-terminals produced from its corresponding non-terminal. For instance, the RPA for `<list>` accepts the following sequences:

```
// Direct productions
begin <text> end
begin <text> <list> end
// Indirect productions
begin <text> begin <text> end end
begin <text> begin <text> <list> end end
```

The set of accepted sequences contains ones produced not only directly but also indirectly from the non-terminal. A direct production of a non-terminal is a sequence that matches the regular expression on the right-hand side of the production rule for that non-terminal. An indirect production is a sequence derived by inlining a non-terminal occurrence of a direct production with a production of that non-terminal.

Silverchain then encodes those RPAs into class definitions in a way that a sequence is emulated by chaining methods of the generated classes. The sequences in the above

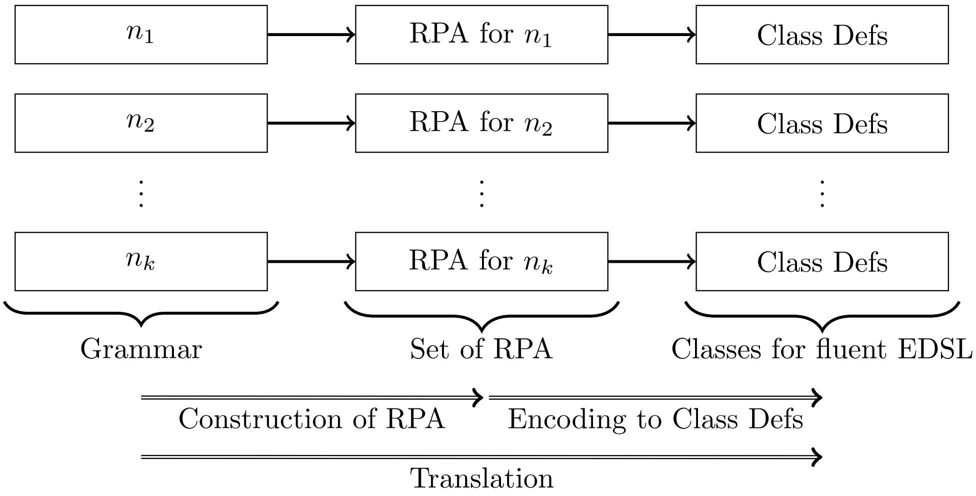


Fig. 3.1: Overview of our translation method. (n_i is a non-terminal.)

example are emulated by the following chains using the classes generated from the RPA for `<list>`:

```
// Direct productions
begin().text("Str").end();
begin().text("Str").list(...).end();
// Indirect productions
begin().text("S").begin().text("S").end().end();
begin().text("S").begin().text("S").list(...).end().end();
```

In our encoding scheme, a non-terminal in a sequence is encoded into a method that takes an argument. The argument is an instance of the type specified on the right-hand side of the rule if the non-terminal has a rule with `::` such as `<text>`. Otherwise, the argument is a subchain that emulates a production of that non-terminal as follows:

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

As we mentioned earlier, the return type of every method only has methods that may be invoked next so that a syntactic error causes a type error. In addition, the return type of the last method in a chain inherits the class corresponding to the source non-terminal if the chain represents a syntactically correct sequence. For example, the last method returns a type inheriting `List`, the class corresponding to `<list>`, only when a chain emulates a production of `<list>`:

```
// No type error
List list1 = begin().text("Str").end();
// Type error on assignment
List list2 = begin().text("Str");
```

This design helps the type checker to find syntactic errors in subchains. For example, the following chain causes a type error by setting the argument type of `list` method to `List`:

```
// Type error on passing argument to list(List)
begin().text("Str").list(begin().text("Str")).end();
```

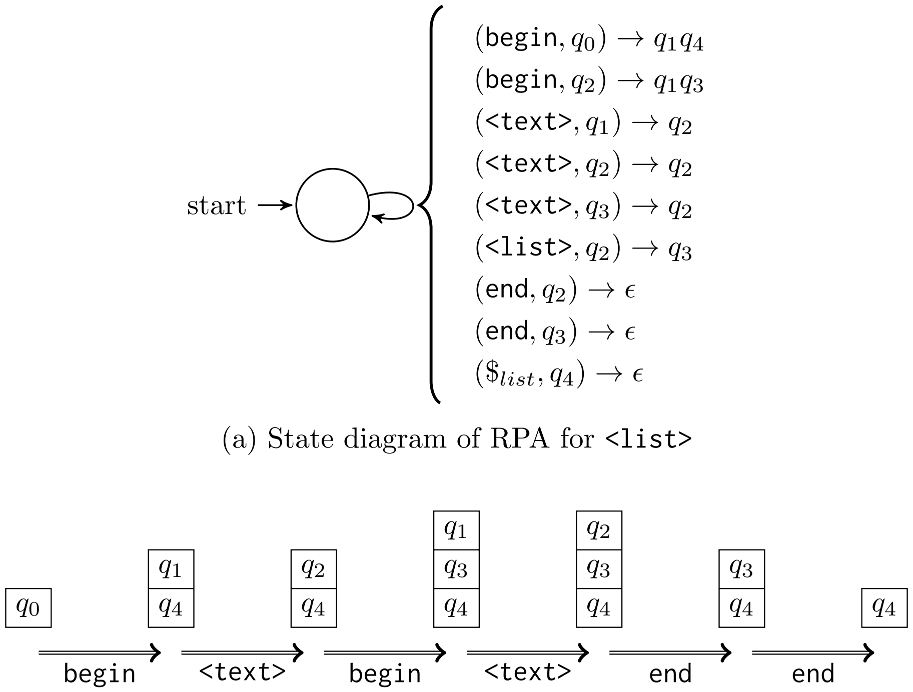
In Section 3.1, we describe RPAs and our encoding scheme in more detail using the table representation of RPAs.

Silverchain constructs the set of RPAs by modifying a set of RPAs each accepts only the direct productions of a non-terminal. This way of constructing RPAs does not add or remove a non-terminal from a given grammar. This property lets Silverchain users (i.e., the developers of an EDSL) specify where the users of a generated EDSL can use subchaining APIs. Suppose that Silverchain rewrites the grammar in Figure 2.3 as follows:

```
<list> -> "begin" <n>;
<n>    -> (<text> <list>?)+ "end"; // Added non-terminal
<text> :: String;
```

From this grammar, our encoding scheme generates `n(N)`, which is a method corresponding to `<n>`:

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



(a) State diagram of RPA for `<list>`

(b) Example stack transition

Fig. 3.2: State diagram and example stack transition

As seen above, adding a non-terminal adds an unexpected subchaining API to the generated EDSL. Removing a non-terminal, on the other hand, removes expected subchaining API from the EDSL. We explain our RPAs construction method in Section 3.2.

As we mentioned, Section 3.1 describes an RPA and its encoding into class definitions and Section 3.2 describes our RPAs construction method. Section 3.3 describes the preprocessing that is applied before the construction of RPAs. The preprocessing is implemented because our RPAs construction method works correctly only when a grammar satisfies several uncommon conditions. The preprocessing rewrites a given grammar so that the grammar satisfies those conditions as much as possible, without adding or removing any non-terminal. Section 3.4 discusses the limitation of Silverchain, which is different from the limitation of our RPAs construction method because of the preprocessing.

3.1 RPA and Its Encoding

An RPA, single-state real-time deterministic pushdown automaton, is a deterministic pushdown automaton that has only one state and no ϵ-transitions. Figure 3.2a shows the state diagram of the RPA for `<list>`. At every step, an RPA takes an action that consumes one input symbol, pops the top of its stack, and pushes zero or more elements into its stack. We denote each action as $(s, q) \to Q$, where s is an input symbol to consume, q is the top of the stack to pop, and Q is a sequence of stack elements to push. All possible actions of the RPA for `<list>` are listed on the right half of Figure 3.2a. When pushing elements, the rightmost element of Q is pushed into the stack first. ϵ on the right-hand side indicates that no element is pushed into the stack on that action. At the beginning, the stack is filled with only one element. In the RPA for `<list>`, the stack is filled with q_0 at the beginning.

Table 3.1: Table representation of RPA. ("-" indicates an undefined action.)

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

Figure 3.2b shows how the content of the stack changes at each step when the input sequence is as follows:

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

The RPA for a non-terminal n stops when it consumes all input symbols and the end-of-input symbol $\$_n$, or when it become unable to consume an input symbol anymore with its defined actions. An input sequence is accepted if the stack becomes empty by consuming $\$_n$. A sequence is, on the other hand, rejected if the RPA stops before consuming $\$_n$. The example sequence above is accepted because the stack is filled with q_4 after consuming the second `end` and the stack will become empty by taking the action that consumes $\$_{list}$ and pops q_4 on the stack top.

An RPA can be described by a table. Table 3.1 is the table representation of the RPA for `<list>`. In the table representation, all possible input symbols (including $\$_n$) and all stack elements are enumerated across the rows and the columns, respectively. A defined action $(s, q) \to Q$ is represented by placing Q in row s of column q. The initial stack element is indicated by adding the non-terminal name on the column top of that element.

A fluent EDSL is obtained by encoding the table into a set of class definitions. Figure 3.3 shows the classes generated from Table 3.1. The generated classes are categorized into two. One is a set of classes each 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 other category consists of two auxiliary classes: the class corresponding to a non-terminal and the class representing the stack bottom. In Figure 3.3, `List` and `Bottom` belong to this category. We describe the usage of these auxiliary classes in the next paragraph.

Each method in a 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 static import. Instead of encoding a cell in column $\$_n$ into a method, the cell is encoded into an `extend` clause. In the case of the example, `Q4` inherits `List` as shown on Line 22 because column q_4 has a value in column $\$_{list}$. 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

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

Fig. 3.3: Class definitions generated from Table 3.1

the type parameter of the owner class. For example, $q_1 q_4$ in column q_0 is encoded into `Q1<Q4<Bottom>>` as shown on Line 3. 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 is determined by how the non-terminal is defined in the given grammar. The argument type is as specified on the right of `::` if the symbol has a rule with `::`. The type is the class corresponding to the symbol such as `List` otherwise. In Figure 3.3, the method bodies are omitted, but they are also generated by Silverchain. We describe them later in Section 3.5.

With the classes in Figure 3.3, programmers can compose a syntactically correct sequence as shown in Figure 3.4. Each comment on the right shows the return type of the method invocation. As seen from the comparison of Figure 3.2b and 3.4, the return type of each method represents the content of the RPA stack at that time.

3.2 RPA Construction

Our method first constructs RPAs each accepts only direct productions, then modifies them so that they can also accept indirect productions. In the example case, it first constructs an RPA accepting the direct production of `<list>`, then modifies the RPA into

```
import static Q0.begin;

List ls = begin()      // Q1<Q4<Bottom>>
    .text("Item 1")    // Q2<Q4<Bottom>>
    .begin()           // Q1<Q3<Q4<Bottom>>>
    .text("Item 1.1") // Q2<Q3<Q4<Bottom>>>
    .end()             // Q3<Q4<Bottom>>
    .end();            // Q4<Bottom> extends List
```

Fig. 3.4: Example chain using classes in Figure 3.3

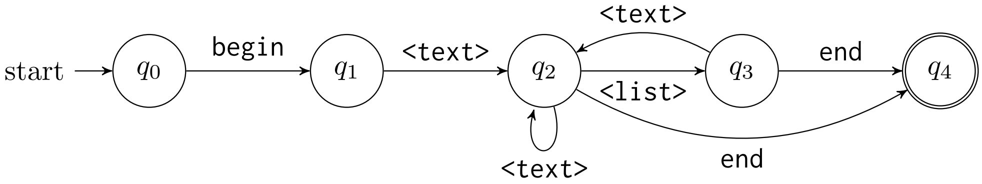


Fig. 3.5: State diagram of DFA for <list>

Table 3.2: Table constructed from Figure 3.5

	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

the one accepting both the direct and indirect productions. Such RPAs accepting only direct productions can be constructed easily from a given grammar by the following steps:

(1) Construct a set of deterministic finite automata (DFAs) by converting the regular expression on the right-hand side of each production rule. (Several conversion algorithms are known such as Brzozowski's [4] and Thompson's [36].) Each DFA accepts the direct productions of the corresponding non-terminal since a direct production is a sequence that matches the regular expression. In the example case, this step constructs the DFA shown in Figure 3.5, which accepts the direct productions of <list>.

(2) Convert each constructed DFA to an equivalent RPA by the following steps. This step constructs Table 3.2, which represents an RPA accepting the direct productions, by converting the DFA in Figure 3.5.

   (2a) Enumerate all possible input symbols (including $\$_n$) and all states of the DFA across the rows and the columns, respectively. A state of the DFA is converted to a stack element of the RPA.

   (2b) Put a non-terminal name n on the top of the column whose header is the initial state of the DFA. The initial state of the DFA is converted to the initial stack

Table 3.3: Modifications made in Step (3) and (4)

(a) At Step (3)

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

(b) At Step (4)

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

element of the RPA.

(2c) Put q_i in row s of column q_j if the DFA can transition from q_j to q_i by consuming s. A transition of the DFA from q_j to q_i is converted to an action of the RPA that pops q_j and pushes q_i into the stack.

(2d) Put ϵ in row $\$_n$ of a column whose header is an accepting state of the DFA. An accepting state of the DFA is converted to an element that allows RPA to make its stack empty by consuming $\$_n$

Step (1) constructs a set of DFAs from any grammar since a regular expression can always be converted into the DFA. Step (2) can always convert a DFA into an RPA since a DFA can be regarded as an RPA whose stack depth is limited to one.

Our modification of RPAs is modeled as the update on the table constructed by Step (1) and (2). Recall that an indirect production is a sequence derived by inlining a non-terminal's occurrence of a direct production with a production of a non-terminal. This implies that, for RPAs to accept indirect productions, the modification should be made so that those RPAs can consume a production of n when they can consume n. To this end, our method updates the table obtained at Step (2) as follows:

(3) Add new actions so that RPAs can start consuming a production of n when they can consume n:

(3a) Collect actions that consume n and let the collection be C_n. In the case where n is `<list>`, C_n is $\{(\texttt{<list>}, q_2) \to q_3\}$.

(3b) Collect actions that pop the initial stack element for n and let the collection be P_n. P_n is $\{(\texttt{begin}, q_0) \to q_1\}$ in the example case.

(3c) For each $((*, q_i) \to q_j, (s, *) \to q_k) \in C_n \times P_n$, put $q_k q_j$ in row s of column q_i. In the example case, this step adds $(\texttt{begin}, q_2) \to q_1 q_3$ as shown in Table 3.3a.

Since s is a symbol that may appear first in a production of n, RPAs become able to consume the first symbol of an inlined sequence by the modification at Step (3c). Furthermore, they can also consume the second and subsequent symbols since those added actions push q_k, the element that allows RPAs to consume the second symbol, on the stack top. The second pushed symbol q_j represents the element that should be on the stack top after consuming all symbols of an inlined sequence. We explain the usage of q_j in the next step.

(4) Modify existing actions so that RPAs can consume symbols following an inlined sequence of n as specified in the grammar:

(4a) Find an element whose column contains ϵ in row $\$_n$ and let the element be e_n. e_n is q_4 in the example case.

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

Fig. 3.6: Example grammar containing multiple non-terminals

(4b) Replace all occurrences of e_n with ϵ. In the example case, this step replaces all occurrences of q_4 with ϵ as shown in Table 3.3b.

(4c) Append e_n to cell values in the column of the initial stack element for n. This step appends q_4 to the value in row `begin` of column q_0 as shown in Table 3.3b.

By the update in Step (4b), the stack top gets back to q_j, which is pushed into the stack when starting an inlined sequence, after consuming the last symbol of an inlined sequence. This allows RPAs to consume symbols following the inlined sequence as specified in the grammar. Step (4c) is modification that fixes the side effect caused by Step (4b). The stack of the RPA for n becomes empty before consuming $\$_n$ since e_n is replaced with ϵ at Step (4b). By the update in Step (4c), the RPA for n becomes able to make its stack empty by consuming $\$_n$ at the end of input sequence.

Table 3.3b is exactly the same as Table 3.1, which represents the RPA accepting both direct and indirect productions of `<list>`.

To show how our method works when an input grammar contains multiple non-terminals, consider a new example grammar shown in Figure 3.6. From the grammar, our method first constructs Table 3.4a by Step (1) and (2). In Table 3.4a, two RPA for `<doc>` and `<list>` are described as one table, but the representation is the same as the case where the table describes only one RPA. In Step (3) and (4), our method defines C_n, P_n, and e_n as follows:

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

where ϕ denotes the empty set. At Step (3c), our methods adds two actions $(\texttt{beginLst}, q_1) \to q_4q_1$ and $(\texttt{beginLst}, q_5) \to q_4q_6$, which are derived from $C_{list} \times P_{list}$. (No new actions are derived from $C_{doc} \times P_{doc}$ because $C_{doc} \times P_{doc}$ is ϕ.) At Step (4b), q_2 and q_7 are replaced with ϵ. Step (4c) appends q_2 and q_7 to cell values in column q_0 and q_3, respectively. Table 3.4b is the table representation of two RPAs for `<doc>` and `<list>`, which is obtained by updating Table 3.4a with Step (3) and (4).

Limitation of RPA Construction

Our method constructs RPAs correctly when the table obtained at Step (2) satisfies the following conditions:

(a) No value exists in row s of column q for all $((*, q) \to *, (s, *) \to *) \in C_n \times P_n$.

(b) A column containing ϵ does not contain any other values.

In the following two paragraphs, we describe how the violation of Condition (a) and (b)

Table 3.4: Tables appearing in RPAs construction when input grammar is Figure 3.6

(a) Table constructed from Figure 3.6

	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) Table obtained by updating Table 3.4a

	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

affect our RPAs construction.

If a table obtained at Step (2) does not satisfy Condition (a), Step (3) cannot put a value into the cell in row s of column q. Table 3.5 is an example of such tables, which is constructed from the following grammar:

```
<list>      -> "begin" (<text> | <text-list>)+ "end";
<text-list> -> <text> <list>;
```

From Table 3.5, Step (3a) and (3b) constructs $C_{text-list}$ and $P_{text-list}$ as follows:

$$\begin{aligned} C_{text-list} &= \{(\texttt{<text-list>}, q_1) \to q_2, (\texttt{<text-list>}, q_2) \to q_2\}\,, \\ P_{text-list} &= \{(\texttt{<text>}, q_4) \to q_5\}\,. \end{aligned}$$

Step (3c) then tries to put q_5q_2 to the cell in row `text` of column q_1 and q_2 but cannot put them since q_2 has already existed in that cell.

Table 3.6a is an example table that does not satisfy Condition (b), which is constructed from the following grammar:

Table 3.5: Table violating Condition (a)

	list				text-list		
	q_0	q_1	q_2	q_3	q_4	q_5	q_6
begin	q_1	-	-	-	-	-	-
<text>	-	$\boldsymbol{q_2}$	$\boldsymbol{q_2}$	-	q_5	-	-
<list>	-	-	-	-	-	q_6	-
<text-list>	-	q_2	q_2	-	-	-	-
end	-	-	q_3	-	-	-	-
$\$_{list}$	-	-	-	ϵ	-	-	-
$\$_{text-list}$	-	-	-	-	-	-	ϵ

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

(a) Example grammar

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

(b) Rewritten grammar

Fig. 3.7: Grammar rewriting performed in our preprocessing

```
<list> -> "begin" <item>+ "end";
<item> -> <text> <list>*;
```

Step (3) updates Table 3.6a as shown in Table 3.6b, and Step (4) updates Table 3.6c. In this case, all updating processes are made without problems, but the constructed RPAs do not accept sequences as specified in the grammar. For example, the RPA for `<list>` in Table 3.6c does not accept the following sequence:

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

This incorrect construction occurs because our method uses the stack to store the element to be on the top after consuming an inlined sequence.

Note that Condition (a) and (b) describe the limitation of our RPAs construction method. Silverchain can translate the example grammars shown above into fluent EDSLs as expected because it preprocesses those grammars as described in Section 3.3. The limitation of Silverchain is discussed in Section 3.4 after describing the preprocessing.

Table 3.6: Table violating Condition (b) and its modification

(a) Table constructed at Step (2)

	list			item		
	q_0	q_1	q_2	q_3	q_4	q_5
begin	q_1	-	-	-	-	-
<item>	-	q_2	q_2	-	-	-
<text>	-	q_2	q_2	-	q_5	-
<list>	-	-	-	-	-	q_5
end	-	-	q_3	-	-	-
$\$_{list}$	-	-	-	ϵ	-	-
$\$_{item}$	-	-	-	-	-	ϵ

(b) Table updated from Table 3.6a by Step (3)

	list			item		
	q_0	q_1	q_2	q_3	q_4	q_5
begin	q_1	-	-	-	-	-
<item>	-	q_2	q_2	-	-	-
<text>	-	$\boldsymbol{q_5 q_2}$	$\boldsymbol{q_5 q_2}$	-	q_5	-
<list>	-	-	-	-	-	q_5
end	-	-	q_3	-	-	-
$\$_{list}$	-	-	-	ϵ	-	-
$\$_{item}$	-	-	-	-	-	ϵ

(c) Table updated from Table 3.6b by Step (4)

	list			item		
	q_0	q_1	q_2	q_3	q_4	q_5
begin	$q_1 \boldsymbol{q_3}$	-	-	-	-	-
<item>	-	q_2	q_2	-	-	-
<text>	-	$\boldsymbol{q_2}$	$\boldsymbol{q_2}$	-	$\boldsymbol{q_5}$	-
<list>	-	-	-	-	-	$\boldsymbol{\epsilon}$
end	-	-	q_3	-	-	-
$\$_{list}$	-	-	-	ϵ	-	-
$\$_{item}$	-	-	-	-	-	ϵ

Table 3.7: Table constructed from rewritten grammar

	doc			list					item		
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7	q_8	q_9	q_{10}
`beginDoc`	q_1	-	-	-	-	-	-	-	-	-	-
`beginLst`	-	-	-	q_4	-	-	-	-	-	-	-
`<list>`	-	q_2	q_2	-	-	q_6	-	-	-	q_{10}	-
`<text>`	-	-	-	-	q_5	q_5	q_5	-	q_9	-	-
`<item>`	-	-	-	-	q_6	q_6	q_6	-	-	-	-
`endLst`	-	-	-	-	-	q_7	q_7	-	-	-	-
`endDoc`	-	-	q_3	-	-	-	-	-	-	-	-
$\$_{doc}$	-	-	-	ϵ	-	-	-	-	-	-	-
$\$_{list}$	-	-	-	-	-	-	ϵ	ϵ	-	-	-
$\$_{item}$	-	-	-	-	-	-	-	-	-	ϵ	ϵ

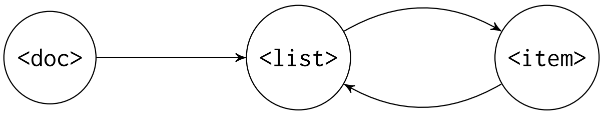


Fig. 3.8: The initial reference relation graph

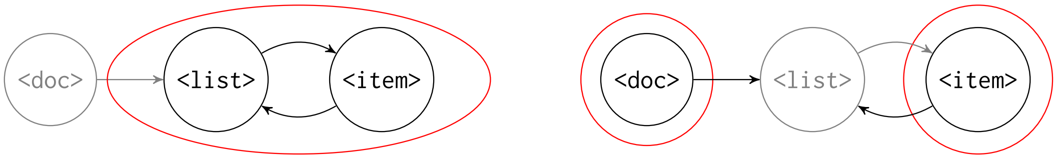
3.3 Preprocessing

Silverchain preprocesses an input grammar so that the grammar satisfies Condition (a) and (b) as much as possible. As an example, consider the grammar shown in Figure 3.7a. Our preprocessing rewrites the grammar as shown in Figure 3.7b. As demonstrated in Figure 3.7, our preprocessing replaces a non-terminal occurrence with the union of that non-terminal and its production rule. Note that this inline-expansion does not add or remove any non-terminals from the grammar. Replaced occurrences of non-terminals are ignored when building C_n at Step (3a) since those occurrences have already been inlined by the preprocessing. For example, Step (1) and (2) construct Table 3.7 from the rewritten grammar. Step (3a) builds the set of actions as follows:

$$\begin{aligned} C_{list} &= \{(\texttt{<list>}, q_1) \to q_2, (\texttt{<list>}, q_2) \to q_2\}, \\ C_{text} &= \{(\texttt{<text>}, q_4) \to q_5, (\texttt{<text>}, q_5) \to q_5, (\texttt{<text>}, q_6) \to q_5\}, \\ C_{item} &= \phi. \end{aligned}$$

C_{item} is ϕ since `<item>` on the right-hand side of `<list>` is ignored at Step (3a) as we described above. By ignoring the inlined occurrences, the problem described in Condition (b) does not occur at Step (3c).

Our preprocessing does not inline expand all every non-terminal occurrence since infinite regression will occur when a non-terminal is defined recursively. It selects non-terminals to be expanded by examining reference relations among non-terminals. A reference 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



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

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

Fig. 3.9: Selecting an edge for inline expansion. (A red circle indicates a SCC.)

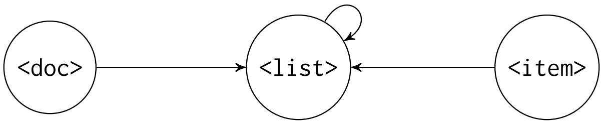


Fig. 3.10: The updated graph after expanding <item> in the rule for <list>

three references from <doc> to <list>, from <list> to <item>, and from <item> to <list>. Our preprocessing is described as follows:

(i) Create a graph G. A node in G represents a non-terminal in a given grammar. An edge (n_s, n_d) in G represents a relation from n_s to n_d. Figure 3.8 shows a graph created from the example grammar.

(ii) Find an edge (n_s, n_d) such that the following subprocess returns true:

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

(ii-b) Decompose G' into strongly connected components (SCCs) and let the component containing n_d be C [35].

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

In the case of G in Figure 3.8, Silverchain selects (<list>,<item>) through the subprocesses shown in Figure 3.9.

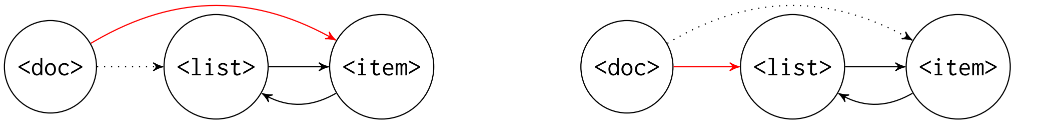
(iii) Apply inline expansion to the production rule for n_s. All the occurrences of n_d in the rule are expanded. For example, when n_s is <list> and n_d is <item>, then the rule for <list> will be expanded into the following as we have seen before:

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

(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 reference relations in the expanded production rule. G is updated to the graph in Figure 3.10 when (<list>, <item>) is selected for expansion.

Our preprocessing repeats from Step (i) to Step (iv) until no edges found in Step (ii). It examines edges in the order that an edge further from the node for the start symbol is examined earlier. (Silverchain regards the first non-terminal of an input grammar as the start symbol.)

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 3.8 is selected for expansion. The dotted edge is removed and the red edge is added as shown in Figure 3.11a. Then the edge (<doc>, <item>) in



(a) Expanding (<doc>, <list>)

(b) Expanding (<doc>, <item>)

Fig. 3.11: Without decomposition into SCCs

Figure 3.11a is selected for expansion and the graph is updated to Figure 3.11b, where the dotted edge is removed and the red edge is added. Figure 3.11b is equivalent to Figure 3.8. Further inline expansion 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 Limitation

Flat chaining (non-subchaining) APIs are not always supported in a fluent EDSL generated by Silverchain. This is because our translation method fails to construct RPAs correctly as described in Section 3.2. Our translation method skips Step (3) and (4) if the table obtained at Step (2) does not satisfy Condition (a) and (b). Since Step (3) and (4) are the process to make RPAs accept indirect productions, RPAs remain to accept only direct productions. This results that generated fluent EDSL allows its users to use only subchaining APIs for certain parts of a chain. For example, consider encoding Table 3.6a into class definitions by the encoding scheme in Section 3.1. Figure 3.12 shows the classes generated from the table and the EDSL users can write the following chains:

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

However, the users cannot write the following chains that use flat chaining APIs:

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

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

Silverchain can generate flat chaining APIs for all parts when every recursively defined non-terminal has explicit symbols that begin/end a nesting structure. 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>)?;
<stmts>   -> (<if-else> | <stmt>)*;
```

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>)*;
```

```
class Q0 {
    static Q1<Bottom> begin() { ... }
}

class Q1<T> {
    Q2<T> item(Item item) { ... }
}

class Q2<T> {
    Q2<T> item(Item item) { ... }
    Q3<T> end()           { ... }
}

class Q3<T> extends List {}

class Q4 {
    Q5<Bottom> text(String text) { ... }
}

class Q5<T> extends Item {
    Q6<T>      list(List list) { ... }
    Q1<Q6<T>> begin()         { ... }
}

class Q6<T> extends Item {}

class List {}

class Item {}

class Bottom {}
```

Fig. 3.12: Class definitions

Silverchain fails to generate non-subchaining APIs since our method constructs an RPA so that it can push an element when a new nesting begins, and pops an element when the current nesting ends. A recursive non-terminal without explicit ending symbols introduces conflicts when popping an element at the end of an inlined sequence.

When given grammar represents a visibly pushdown language [1] (VPL), the cases Silverchain cannot generate non-subchaining API will not happen. This is because a VPL has explicit unique symbols to begin/end a nesting structure. However, further formal discussion and proof are necessary to posit this claim.

3.5 Details of Encoding

Silverchain generates method bodies that build a list of invoked method. Figure 3.13 shows the generated code for `Q2` including the bodies of methods. Each method first appends a `Method` instance, an object that packs the name and the argument, to the list. Line 11 in Figure 3.13 is the line that appends such packed information. A method appends

```
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) {
        }
    }
}
```

Fig. 3.13: Definition of Q2

multiple `Method` instances at once when it takes a subchain as its argument as shown on Line 25. A method then creates and return an instance that is used as the receiver of the next method invocation. The list of invoked methods are shared by passing it as the argument of the constructor as shown on Line 13. We need a little trick for generated methods to return a nested parametric type as shown from Line 30 to Line 39. 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.

As a manually developed fluent API often provides several convenience methods, Silverchain also generates convenience methods. The method on Line 16 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 $(s, q_j) \to q_j$.

We show the generated code only for `Q2`, but the method bodies of other classes are very similar to the ones in `Q2`. For example, the body of `text(String)` in `Q1` is exactly 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 invoked methods and a stack to store classes:

```
static Q1<Q4<Bottom>> begin() {
    ArrayList<Method> methodList = new ArrayList<>();
    methodList.add(new Method("begin", null));

    Stack<Class<?>> classStack = new Stack<>();
    classStack.push(Q4.class)

    return new Q1<>(methodList, classStack);
}
```

To add semantic actions to the generated fluent EDSL, Silverchain users have only to implement an evaluation method, a method that interprets a written chain. Figure 3.14 shows an example implementation that constructs an itemized document in TEX from a written chain. By fitting all semantic actions into one method, Silverchain users can avoid implementing method bodies that are scattered over the generated code. However, this implementation of semantic actions is tedious when building actions for a DSL that does not have an external execution system. If the EDSL is just an embedded interface to an external DSL such as SQL and TEX, the implementation is much easier since Silverchain users have only to convert a method into a token of that external DSL.

```
String toTeX() {
    Sttring 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;
}
```

Fig. 3.14: Example implementation of EDSL semantics

Chapter 4

Use Cases

In this chapter, we compare fluent EDSLs generated by Silverchain 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. Table 4.1 summarizes the numbers of classes and methods generated from the grammars of those DSLs. The first and second columns in the table show the numbers of unique symbols and non-terminals in the grammar, respectively. The third and forth columns show the numbers of generated classes and methods, respectively. The grammars of those languages are relatively simple, but the numbers of generated definitions are too large to handle by hand.

Figure 4.1 and 4.2 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 4.1c is similar to Figure 4.1b, and Figure 4.2c and 4.2d are similar to Figure 4.2b.

The first drawback is that code written with the generated EDSLs tends to contain redundant method calls. For instance, `from().collection(...)` in Figure 4.1c is expressed in a shorter way by `from(...)` in Figure 4.1b. Similarly, `beginGraph()` in Figure 4.2c is omitted in Figure 4.2b. These redundant methods are generated because 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 can hardly edit those classes to make APIs better because generated EDSLs use mechanically named classes such as `Q1` and `Q2`. The

Table 4.1: Number of classes and methods generated by Silverchain

	#Symbols	#Non-terminals	#Classes	#Methods
LINQ	32	17	132	323
DOT	40	15	389	1687

[*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

Fig. 4.1: Comparison in LINQ

APIs of coollection and graphviz-java are designed using the domain-specific knowledge that is not represented in the grammar. For instance, coollection uses `enum` to specify the order as shown in Figure 4.1b and graphviz-java uses `enum` to specify attributes of a node as shown in Figure 4.2e. 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 because our translation method naively encodes every token of a sentence into a method. Finding a smarter way of encoding is important future work to put Silverchain into practical use.

As another use case, we gave the grammar of the DSL used by Silverchain for describing input grammar. The generated EDSL is a fluent EDSL that can be used to define fluent EDSLs. Using this EDSL, the grammar in Figure 3.7a can be written as shown in Figure 4.3. Figure 4.3a shows a chain that expresses the grammar without subchaining. Figure 4.3b shows a chain that uses subchaining to group semantically related method calls. The users of this EDSL can use any style that suits their usage.

```
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) Using generated EDSL with subchaining

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

(d) Using generated EDSL 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

Fig. 4.2: Comparison in DOT

```
String src = grammar()

    .nsym("doc").arrow().tsym("beginDoc").nsym("list").star().tsym("endDoc")

    .nsym("list").arrow().tsym("beginLst").nsym("item").plus().tsym("endLst")

    .nsym("item").arrow().nsym("text").nsym("list")

    .nsym("text").is().type("String")

    .toJavaSrc();
```

(a) Without subchaining

```
Rule doc =
    nsym("doc").arrow().tsym("beginDoc").nsym("list").star().tsym("endDoc");

Rule list =
    nsym("list").arrow().tsym("beginLst").nsym("item").plus().tsym("endLst");

Rule item = nsym("item").arrow().nsym("text").nsym("list");

Rule text = nsym("text").is().type("String");

String src = grammar().rule(doc, list, item, text).toJavaSrc();
```

(b) With subchaining

Fig. 4.3: Usage example of generated EDSL for EDSLc

Chapter 5

Conclusion

5.1 Summary

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. Firstly, it throws a type error when a built chain represents a syntactically incorrect sentence of the original DSL. This is realized by setting the return type of each method based on which methods can be invoked next. The problem on the developers' side, the burdensome task to achieve the syntax-checking feature, is the primary reason why we proposed a fluent EDSL generator. Secondly, the generated EDSL supports subchaining APIs in addition to regular chaining APIs. This helps programmers express their solutions in writing style fitting the situation. Silverchain can generate fluent EDSLs for existing practical DSLs as we showed in Chapter 4.

Our translation method is modeled as the RPAs construction as we explained in Chapter 3. The construction method is different from known ones in that our method keeps a given grammatical structure by avoiding addition and removal of non-terminals. This constraint is required to let Silverchain users specify where subchaining APIs are available. Our construction method is described as the construction of the table representation of RPAs and the update on that table. More specifically, it first constructs a set of RPAs accepting only direct productions and then modifies them into the ones accepting both direct and indirect productions.

Since our construction method is limited to a grammar that satisfies the uncommon conditions described in Section 3.2, Silverchain rewrites a given grammar to make it satisfy the conditions as much as possible. This makes our translation method complicated but fairly widen the range of grammars that Silverchain can translate. Our translation method is designed to be portable to other object-oriented languages although Silverchain currently supports only Java as its output general-purpose language. Other output languages can be supported by adding the encoders to class definitions in those languages.

5.2 Future Work

Silverchain fails to generate regular chaining APIs for some grammars as we mentioned in Section 3.4. Future work is to clarify the range of grammars that our current translation method can handle and to find its relation to 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 the 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.

Publications

(1) Tomoki Nakamaru, Kazuhiro Ichikawa, Tetsuro Yamazaki, Shigeru Chiba. 2017. Silverchain: A Fluent API Generator. 16th International Conference on Generative Programming: Concepts and Experience.

(2) 中丸 智貴, 市川 和央, 山崎 徹郎, 千葉 滋. 2017. Java用Fluent API生成システム“B2F”の設計と開発. ポスター発表, 第19回プログラミングおよびプログラミング言語ワークショップ.

(3) 松永 智將, 市川 和央, 山崎 徹郎, 中丸 智貴, 千葉 滋. 2017. 型検査を用いたコンパイル時LR構文解析手法の提案. 日本ソフトウェア科学会第34回大会.

Acknowledgements

テーマの決定からその後の研究の方向性, 論文の執筆にあたって注意する点・意識すべき点についてアドバイスをくださった千葉教授, 市川さん, 山崎さん, 英語での論文執筆において支援してくれた Antoine, Daniel, Joe にまず初めに深く感謝の意を表します. そして, 26歳になってからの唐突な修士課程進学という目的地不明の選択を支えてくれた家族, 友人にも深く感謝の意を表します.

References

[1] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*, STOC '04, pages 202–211, New York, NY, USA, 2004. ACM.

[2] Eric 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, pages 5–14, New York, NY, USA, 2010. ACM.

[3] Eric 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, pages 1–6, New York, NY, USA, 2014. ACM.

[4] Janusz Brzozowski. Canonical Regular Expressions and Minimal State Graphs for Definite Events. pages 529–561, 1962.

[5] John Cocke. *Programming Languages and Their Compilers: Preliminary Notes.* Courant Institute of Mathematical Sciences, New York University, 1969.

[6] Joel Costigliola. AssertJ / Fluent assertions for java, December 2014.

[7] Bruno Courcelle. On jump-deterministic pushdown automata. *Mathematical systems theory*, 11(1):87–109, Dec 1977.

[8] Joseph Darcy. JDK 9 Language, Tooling, and Library Features, September 2016.

[9] C. J. Date and Hugh Darwen. *A Guide to the SQL Standard (4th Ed.): A User's Guide to the Standard Database Language SQL.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[10] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. *Graphviz— Open Source Graph Drawing Tools*, pages 483–484. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[11] Sebastian Erdweg, Tillmann Rendel, Christian Kästner, and Klaus 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, pages 391–406, New York, NY, USA, 2011. ACM.

[12] Martin Fowler. FluentInterface, December 2005.

[13] Yossi Gil and Tomer Levy. Formal Language Recognition with the Java Type Checker. In Shriram Krishnamurthi and Benjamin S. Lerner, editors, *30th European Conference on Object-Oriented Programming (ECOOP 2016)*, volume 56, pages 10:1–10:27, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[14] Data Geekery GmbH. jOOQ: The easiest way to write SQL in Java, September 2017.

[15] Sheila Greibach. A new normal-form theorem for context-free phrase structure grammars. *J. ACM*, 12(1):42–52, January 1965.

[16] Radu Grigore. Java generics are turing complete. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 73–85, New York, NY, USA, 2017. ACM.

[17] Hamcrest. Hamcrest, August 2012.

[18] Michael Harrison and Ivan Havel. Real-time strict deterministic languages. *SIAM*

Journal on Computing, 1(4):333–349, 1972.

[19] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es), December 1996.

[20] Kazuhiro Ichikawa. phenan/scalalr: ScaLALR : LALR parser generator for embedded DSLs in Scala, December 2016.

[21] Kazuhiro Ichikawa and Shigeru Chiba. User-defined operators including name binding for new language constructs. *The Art, Science, and Engineering of Programming*, 1(2), 2017.

[22] j2html. Fast and fluent Java HTML5 builder - Java HTML builder, September 2017.

[23] jMock. jMock - An Expressive Mock Object Library for Java, December 2012.

[24] Tadao Kasami. An efficient recognition and syntax-analysis algorithm for context-free languages. Technical report, DTIC Document, 1965.

[25] KiwiSoft. SQL Query Plugin :: JetBrains Plugin Repository, November 2006.

[26] Ansgar Konermann. Emulating ”self types” using Java Generics to simplify fluent API implementation, November 2010.

[27] Tomer Levy. *A Fluent API for Automatic Generation of Fluent APIs in Java.* PhD thesis, Israel Institute of Technology, 2017.

[28] Ligia Nistor, Darya Kurilova, Stephanie Balzer, Benjamin Chung, Alex Potanin, and Jonathan 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, pages 9–16, New York, NY, USA, 2013. ACM.

[29] Jan Pittl and Amiram Yehudai. Constructing a realtime deterministic pushdown automaton from a grammar. *Theoretical Computer Science*, 22(1):57 – 69, 1983.

[30] Igor Polevoy. Javalite - activejdbc. http://javalite.io/activejdbc, July 2017. (Accessed on 01/13/2018).

[31] Jon Rafkind and Matthew Flatt. Honu: Syntactic extension for algebraic notation through enforestation. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering*, GPCE '12, pages 122–131, New York, NY, USA, 2012. ACM.

[32] Vicente Romero and Maurizio Cimadamore. JEP 215: Tiered Attribution for javac, July 2016.

[33] Daniel Rosenkrantz and Richard Stearns. Properties of deterministic top down grammars. In *Proceedings of the First Annual ACM Symposium on Theory of Computing*, STOC '69, pages 165–180, New York, NY, USA, 1969. ACM.

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

[35] Robert Tarjan. Depth-first search and linear graph algorithms. pages 114–121, Oct 1971.

[36] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.

[37] Todd Veldhuizen. C++ Templates are Turing Complete. Technical report, Indiana University Computer Science, 2003.

[38] Peter Verhas. verhas/fluflu: Fluent Api Creator, July 2014.

[39] Hao Xu. *EriLex: An Embedded Domain Specific Language Generator.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

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