博士論文

An Empirical Study and Code-Generation Techniques for Fluent Interfaces

(Fluent Interface のための実証研究とコード生成技術)

Doctoral Dissertation of:
Tomoki Nakamaru
(中丸 智貴)

Academic Advisor:
Shigeru Chiba
(千葉 滋)

Abstract

This dissertation presents our research to improve the user experiences of software libraries with *fluent interfaces*, interfaces designed to be used by chaining method invocations. The dissertation includes three studies: (1) An empirical study of the use of fluent interfaces in the real world, (2) an empirical study to discover desirable language designs for the use of fluent interfaces, and (3) the development of code-generation techniques to enable quick construction of safe fluent interfaces.

Study (1) is a background study to quantitatively reveal the significance of studying fluent interfaces. In previous studies on fluent interfaces, the significance has been claimed only qualitatively based on the abundance of existing fluent interfaces. To the best of our knowledge, no quantitative evidence has been provided to support the widespread use of fluent interfaces. For our goal, we conducted repository mining of numerous git repositories. Specifically, we collected 2,814 Java repositories on GitHub and analyzed historical trends in the use of fluent interfaces in those repositories.

Study (2) aims to help language developers to design their language appropriately for using fluent interfaces. To find such language designs, we analyzed Java code snippets in the real world and mined problematic code patterns for fluent interfaces. Our results are summarized as a list of desirable language designs and the statistically-estimated values of how effective it would be to introduce each language design into Java. The information we made is beneficial for language developers who attempt to improve the user experiences of fluent interfaces in their language because they can use the list and estimated impact values to smoothly discuss what language design to be adopted.

Study (3) aims to enable library developers to quickly give misuse-detection capabilities to their fluent interfaces, i.e., to quickly create safe fluent interfaces. Although safe fluent interfaces have been known to benefit their users, they are not widely developed in the real world due to their high development cost. While several code-generation techniques have been pro-

posed to reduce the cost, those existing techniques lack two essential features for practical use: generics support and sub-chaining support. In Study (3), we propose two novel code-generation techniques to address those problems.

All the studies presented in the dissertation include artifacts that offer values to society, not only to academia. In Study (1) and Study (2), we analyzed real-world Java source code to benefit real-world programmers and built a publicly available dataset to further investigate the real-world use of fluent interfaces. The code-generation techniques in Study (3) are demonstrated in tools named PROTOCOOL and Silverchain, which allow real-world library developers to test our techniques in real-world settings. Those tools are openly available at GitHub.

Acknowledgements

I would first like to express my sincere gratitude to my supervisor, Prof. Shigeru Chiba, for the continuous support throughout my Ph.D. program. What leads me to this point is his kind, encouraging, and insightful advice. Things I learned from him will be a compass for life as a researcher.

I would also like to thank the members of my thesis committee: Prof. Takeo Igarashi, Prof. Masayuki Inaba, Prof. Mary Inaba, Prof. Ryota Shioya, and Prof. Manabu Tsukada, for all the guidance and feedbacks that they gave me from the viewpoint of their specialties.

My gratitude extends to all the lab members I shared time with, notably Soramichi Akiyama, Kazuhiro Ichikawa, Tomomasa Matsunaga, Yutaro Orikasa, Daniel Perez, Antoine Tu, and Tetsuro Yamazaki. The conversations we had during coffee breaks were not only a precious memory but also a driving force of my research.

Last but not least, I want to thank my parents and girlfriend for their unlimited help and support outside of the lab. I would not have completed this dissertation without them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Enabling computer programmers to create high-quality application software in a shorter period of time, i.e., improving the productivity of application programmers, is a mission given to researchers of programming languages and software engineering. While approaches vary from study to study (e.g., exploration of theoretical properties, development of novel implementation techniques, and statistical analysis of real-world source code), every study in those fields aims to contribute to the mission in some way. The endeavor towards the mission is beneficial for the entire society, not only for programmers, since higher productivity enables programmers to create or update applications more quickly as requested by society.

Our research aims to contribute to the mission through improving the user experiences of software libraries whose interfaces are designed in a style called fluent interfaces. Fluent interfaces are library interfaces that are designed to be used by chaining method invocations [20, 32] as follows:

```
new Dialog().title("Warning").message("Are you sure?").show();
```

| | Method invocation | Method invocation | Method invocation |

Chain of method invocations

Fluent interfaces are a common design. There are a number of libraries with fluent interfaces in various object-oriented languages such as Polly[1] in C#, the Stream API[2] in Java, jQuery[3] in JavaScript, the Finder Component of Symfony[4] in PHP, and Pretty Tensor[5] in Python.

---

[1] http://www.thepollyproject.org
[2] https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html
[3] https://jquery.com
[4] https://symfony.com/doc/current/components/finder.html
[5] https://github.com/google/prettytensor

Language   Application   Library
Developer   Developer   Developer

| creates | creates | creates |

| Empirical Study | supports → | Language suitable for Fluent Intf. | Application Software | Library with Safe Fluent Intf. | ← supports | Code-gen. Technique |

Figure 1.1: Connection between our studies and application development

Our research does not directly improve the user experiences of fluent interfaces; we conducted research that helps language and library developers to improve it. Specifically,

1. we conducted an empirical study to reveal what language design helps the use of fluent interfaces and to what extent effective it would be. Such information helps language developers to appropriately design a suitable language for fluent interfaces.

2. we developed code-generation techniques for safe fluent interfaces, a form of fluent interface that is known to be user-friendly but also known to cost a lot for its construction. Such techniques enable library developers to construct safe (user-friendly) fluent interfaces with smaller effort, which leads to more safe (user-friendly) fluent interfaces to be developed in the real world.

Figure 1.1 illustrates the connection between our studies and application development. In addition to the studies for better user experiences,

3. we conducted a background suvery to reveal the importance of improving the user experiences of fluent interfaces in a quantitative manner. The importance have been argued only qualitatively in preceding studies.

In the following, we first illustrate the position of our studies in the context of research on programming languages and software engineering. We then introduce three studies (one background suvery, and two studies for better user experiences) presented in this dissertation. At the end of this chapter, we summarize our contributions and present the organization of this dissertation, along with the correspondence of the chapters and our publications.

1.1 Background

1.1.1 Importance of Libraries

No one would avoid using libraries when developing application software. Using libraries, one can save time for building common parts with other applications and concentrate on developing unique parts of their application. As well as development speed, the use of libraries also improves the quality of an application. A program collection provided as a library is usually tested and maintained by many programmers. Therefore, it is very likely to be more efficient, secure, and fault-tolerant than programs written on the fly during their development. We can indeed learn and deeply understand a lot of concepts and techniques by reinventing the wheel instead of using libraries. Still, that attitude is not appreciated when creating a fast and reliable application that everyone can use with confidence.

The importance of libraries can be observed in the language-selection tendency; programmers often select a programming language to be used in their development based on the presence (or absence) of suitable libraries for the application. Python's popularity in the machine learning field is a good example of such a selection. Most applications using machine learning are written in Python now, but Python is not a language developed for machine learning nor one with rich features for it. In fact, the popularity owes to well-maintained and well-documented libraries written in Python such as Keras[6] and PyTorch[7]. According to the survery presented in the literature [50], the availability of suitable libraries is the most influential factor for language selection.

1.1.2 User Experiences of Libraries

Since libraries are always used in application development, improving their user experiences benefits all programmers. If a library is designed to be user-friendly or if a language is designed to support the use of a certain sort of library, the users can quickly get started and build what they want using that library. Conversely, if not, the users need to spend much time on unessential tasks for application development, such as reading the documents, visiting QA sites, and writing code snippets to test the behavior.

---

[6]https://keras.io
[7]https://pytorch.org
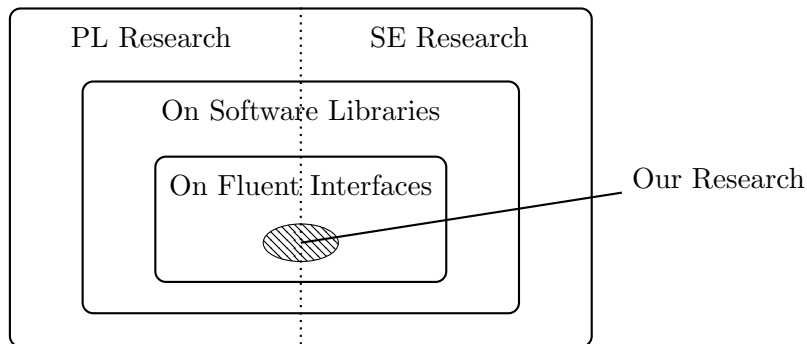
Figure 1.2: Position of our research

1.1.3   Our Scope

In this research, we focus on libraries designed in a certain style known as fluent interfaces and consider improving their user experiences, instead of trying to support all types of libraries. This scope limitation is our choice for better engineering research; we focus on specific cases and investigate highly effective solutions for those cases, rather than seeking a generic but less effective solution that works in all possible cases. Figure 1.2 illustrates the position of our research in research on programming languages (PL) and software engineering (SE).

Although we limit the scope of our research, the scope is broad to a certain degree. Fluent interfaces are not a design for specific domains such as database operations and machine learning. As we have listed at the very beginning of this chapter, there are fluent interfaces for various domains in the real world: Polly for describing fault-handling policies, the Stream API for processing data sequences, jQuery for HTML DOM traversal and manipulation, the Finder Component for querying files in a filesystem, and Pretty Tensor for building deep neural networks. Further, a fluent interface does not require exceptional language designs that only a small number of real-world languages provide. It only requires the method invocation syntax, which should be provided in an object-oriented language. Therefore, a fluent interface can be created in almost all object-oriented languages.

1.2   Acceptance of Fluent Interfaces

In the previous section, we argued the broadness of our scope by describing the high applicability of fluent interfaces. However, being able to adopt

fluent interfaces only indicates their potential and does not indicate their actual adoption in the real world. If fluent interfaces are less used in real software development, our research – improving the user experiences of fluent interfaces – hardly contributes to productivity improvement. The popularity of fluent interfaces has been claimed in the studies [28, 29, 42, 83, 84], but those claims are based on qualitative discussions about the advantages of fluent interfaces. Even worse, there are online materials that refer to fluent interfaces as a problematic design style [7, 65].

This meta-concern leads us to an empirical study to reveal the real-world acceptance of fluent interfaces in a quantitative manner. To dispel this concern, we collected Java repositories on GitHub[8] and analyzed historical trends and frequent code patterns of method-chaining expressions (expected expressions using fluent interfaces). The analysis of historical trends is crucial. Even if the use of fluent interfaces is seemingly high, our research soon becomes ineffective when the use is shrinking over time. We chose Java as the target language since Java has been widely used for a long time and there are a number of repositories on GitHub from more than ten years ago, which is a preferable property for analyzing the historical trends.

One may think that analyzing method-chaining expressions is an indirect or strange approach for revealing the real-world acceptance of fluent interfaces. However, other approaches have fatal problems:

Counting libraries with fluent interfaces. The main problem of this approach is that the increasing number of fluent interfaces does not indicate the increasing use of fluent interfaces. As we described above, our concern is whether fluent interfaces are increasingly used or not. Furthermore, without concrete use cases, it is difficult to objectively judge if a library provides a fluent interface since many fluent interfaces do not explicitly introduce themselves so. For instance, the documentation of the Stream API does not contain the word "fluent".

Interviewing real-world programmers about fluent interfaces. The primary problem of this approach is that it is difficult to answer a question about several years ago (e.g., how often were you using fluent interfaces five years ago?). As the quantitative observation of historical changes is essential to investigate real-world acceptance, this approach is inadequate. Moreover, the answers are likely to be noised by interviewees' subjective since the definition of a fluent interface is a controversial topic (see chapter 2 about the controversy).

---

[8]https://github.com

Note that we do not argue these alternative studies are completely useless. We discuss these studies as future work in chapter 3 and chapter 4.

1.3   Empirical Study of Language Desings

The user experience of a library can be improved by designing a programming language appropriately. As an example, consider the following code piece in Java:

```
// Retrieving top-left value of table
int i = table.rowAt(0).valueAt(0).asInt();
```

If `rowAt` returns `null` in case where no row exists at the given index, this line cause a null pointer exception. To avoid such a runtime error, a programmer needs to split the method chain into two chains and insert the guard as follows:

```
Value value;
Row row = table.getRowAt(0);
if (row != null) {
  value = row.getValueAt(0)
}
value = null;
```

While the runtime error can be eliminated by splitting the chain as shown above, such a separation is not appreciated in terms of the design concept of fluent interfaces. The nullsafe type system and safe call syntax in Kotlin [59] makes the legibility better for this code pattern.

```
return table.rowAt(0)? // Safe call on nullable type
    .valueAt(0).asInt();
```

Other than the above-described design in Kotlin, several designs in advanced (but less popular) languages have been known to help the use of fluent interfaces.

Our first study for better user experiences aims to reveal what language design is useful to support the use of fluent interfaces and how large the impact of introducing those designs is, through the examination of method-chaining expressions in the real world. Since we attempt to discover a language design, counting the use of some language designs is inappropriate. Such an approach would reveal the impact of a certain language design but does not lead to the discovery of unknown language designs.

To this end, we manually analyzed method-chaining expressions that are randomly sampled from collected Java repositories on GitHub. Specifically, we investigated possible reasons why those invocations are not chained in a code piece and looked for language designs to transform the piece into a single method-chaining expression. We chose Java since Java has a relatively simple syntax and does not provide any special designs for the use of fluent interfaces.

1.4   Code generation for Safe Fluent Interfaces

A safe fluent interfaces is a fluent interface that is designed to cause a type error when the user chains method invocations in an incorrect manner. No difference exists in the appearance of source code between a safe fluent interface and a (regular) fluent interface. The difference is apparent only when an incorrect method chain is type-checked. This feature of misuse detection greatly improves the user experiences since the users can find misuses at compile time without checking the documentation or writing test code.

The misuse-detection capability is achieved by setting the return type of each method based on which methods the user can invoke next. To illustrate this more concretely, consider developing a Java library for composing the SQL queries as follows:

```
// SELECT name FROM users
new SQL().select("name").from("users").execute();
```

Since this library is for emulating the SQL queries in Java, the following chain should be incorrect:

```
// Missing `from()`; Invalid SQL query
new SQL().select("name").execute();
```

This misuse can be detected in type-checking if the return type of `select()` is a type that only has `from()`. The users are forced to invoke `from()` right after the invocation of `select()`. If they invoke `execute()` right after `select()`, a type checker of a language emit a type error. Figure 1.3 shows this safe architecture in more detail and how a type error occurs by example.

Safe fluent interfaces also improve programmers' productivity while editing source code, not only at compile time. They cooperate well with a method-completion system in an integrated development environment (IDE), which usually suggests candidate methods based on the return type of a method. Since the return type of each method is selected appropriately in

```
class SQL {                        class SQL2 {
  SQL() { ... }                      SQL3 where() { ... }
  SQL1 select() { ... }              Result execute() { ... }
}                                  }
class SQL1 {                       class SQL3 {
  SQL2 from() { ... }                Result execute() { ... }
}                                  }
```

(a) Architecture

```
// Invalid statement causes compile error
new SQL()
  .select("name")  // Returns `SQL1`
  .where("id = 1") // `SQL1` does not have `where()`
                   // Type error!

// Valid statement causes no error
new SQL()
  .select("name")   // Returns `SQL1`
  .from("users")    // Returns `SQL2`
  .where("id = 1")  // Returns `SQL3`
  .execute();
```

(b) Usage examples

Figure 1.3: Architecture and examples of safe fluent query builder
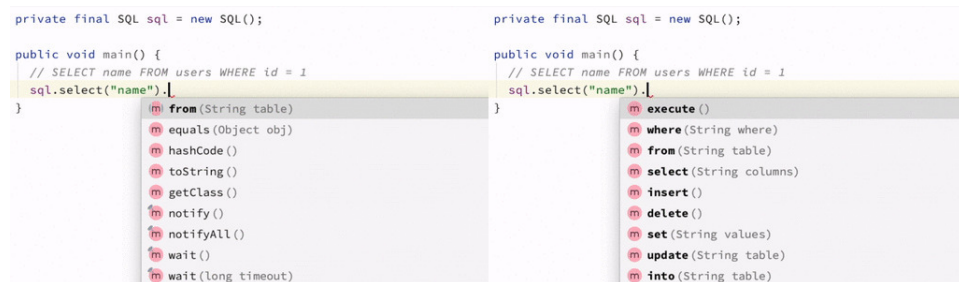
Figure 1.4: Method completion

```
class SQL {
  SQL() { ... }
  SQL select(String columns) { ...; return this; }
  SQL from(String table) { ...; return this; }
  SQL where(String expression) { ...; return this; }
  Result execute() { ... }
}
```

Figure 1.5: Architecture of naively-designed fluent query builder

the safe fluent design, the methods appearing in the candidate list are correct methods and the users can quickly build an expression just by selecting from those correct candidates. Figure 1.4 is the screenshot showing the difference of the candidate methods in the safe fluent design and in the naive design (shown in fig. 1.5).

Although safe fluent interfaces offer the above-mentioned advantages to the users, it is not widely used in the real world due to its development cost. The library developers need define a number of classes and wire them by carefully selecting the return type of each method, to give the safe property to the interfaces. As seen in the comparison of fig. 1.3 and fig. 1.5, the safe design requires four class definitions whereas the naive design requires only one class definition. When a library provides more features than our example SQL library, the number often becomes so large that the developers cannot manage by hand.

To address the problem of the development cost, several studies have been published on the code generation of safe fluent interfaces [28, 29, 32, 42, 83, 84]. The key idea is to regard the correct order of method invocations as the syntax of a method chain. From this viewpoint, the construction of a safe fluent interface can be modeled as the construction of a parser on top of a type system. The code generation of safe fluent interfaces is to generate source code of the type-level parser from a given grammar.

However, those studies rather focus on the theoretical aspects of this topic. Roughly speaking, they focus on extending the grammar class that can be generated by the algorithms. From the viewpoint of practical applicability, two problems remain unaddressed in the code-generation algorithms: Lack of sub-chaining support and generics support. In the rest of this section, we describe these problems.

1.4.1   Lack of Generics Support

It is popular to define generic methods – methods including type parameters in their definition – when creating a library in the real world. For instance, the Stream API in Java is an example of such a generic interface. It provides generic methods such as `map(Function<? super T, ? extends R> mapper)`. A generic method uses a type parameter to check the *semantic* constraints of the interfaces; for example, to check whether a correct value type is passed through a stream.

However, in preceding studies, the role of a type parameter was limited to the internal representation of a stack element of the parser built on the type system. That is, library developers cannot use type parameters for their library interfaces. The problem of the existing algorithms can be described as the lack of the binding-time analysis of type parameters, which we illustrate by example at the very beginning of chapter 5.

1.4.2   Lack of Sub-chaining Support

To take advantage of method chaining in various situations, a fluent interface should provide sub-chaining interfaces, commonly provided interfaces in real-world libraries to compose a part of a method chain as another method chain:

```
select("name").from("user").where(
  col("id").eq(1) // Sub-chain
);
```

Sub-chaining interfaces allow programmers to build an entire chain from semantically grouped partial chains. Without sub-chaining interfaces, programmers need to handle strange types and write less readable code when changing a part of a chains or when creating a function for reusing a part of a chain:

```
PartialQuery q = select("name").from("user").where();
PartialQuery r;
if (findById) { // findById: Given boolean value
  r = q.col("id").eq(1);
} else {
  r = q.col("name").eq("John");
}
r.execute();
```

Table 1.1: Organization and correspondence

	Topic	Publication
Chapter 1	Overview	–
Chapter 2	Review on fluent interfaces	–
Chapter 3	Acceptance of Fluent Interfaces	[53]
Chapter 4	Empirical study of language designs	[53]
Chapter 5	Code-generation for generics support	[52]
Chapter 6	Code-generation for sub-chaining support	[55, 56]
Chapter 7	Summary and future work	–

The above code can be a more readable form that reflects the essential structure of an expression if sub-chaining interfaces are provided:

```
WhereClause w1 = col("id").eq(1);
WhereClause w2 = col("name").eq("John");
select("name").from("user")
  .where(findById ? w1 : w2).execute();
```

However, the existing generators do not generate fluent interfaces with rich sub-chaining support. They are specialized to generate flat-chaining interfaces from a given grammar. If library developers separately generate every sub-chaining interface using those generators, it is possible to obtain fluent interfaces with rich sub-chaining support. However, library developers need to give the generators a lot of grammar definitions that overlap each other. To realize the generation of fluent interfaces with rich sub-chaining support, we need to develop a code-generation technique that automatically generates sub-chaining interfaces besides flat-chaining interfaces.

1.5 Organization and Contributions

This dissertation is a compilation of our publications on fluent interfaces, specifically the literatures [52, 53, 55, 56]. Table 1.1 summarizes the topic of each chapter and its correspondence to our publication, and fig. 1.6 illustrates the position of each chapter in the context of PL and SE research.

The next chapter details our discussion on fluent interfaces at the end of section 1.1 and the beginning of section 1.2. It includes the terminologies, review on existing opinions, and comparison to other technologies in terms

Figure 1.6: Positon of each chapter in PL and SE research

of embedded domain-specific languages (EDSLs) [37]. It also examines the safe fluent design in contrast to other misuse detection techniques.

Chapter 3 presents our meta-study that investigates the significance of the study of fluent interfaces, which we have introduced in section 1.2. This chapter is the first half of our paper published at the Mining Software Repositories conference 2020. The contribution presented in this chapter is as follows:

- We present, to the best of our knowledge, the first quantitative study on the use of method chaining and fluent interfaces that is based on a large set of source code in the real world.

- We empirically show the increasing use of method chaining and fluent interfaces in Java, which has been claimed without empirical evidence in preceding studies [28, 29, 42, 83, 84].

Chapter 4 statistically analyzes language designs that support the use of fluent interfaces. The chapter is the second half of our paper published at the Mining Software Repositories conference 2020. The contribution of this study, the importance of the concrete list, is as described as follows:

- We present a concrete list of language designs (and alternative interface designs) that support method chaining and fluent interfaces.

- We present our statistical estimation on how effective each design would be in the real world. This quantitative information is useful for language and library developers to objectively judge whether a design should be introduced into a language or library.

In chapter 5 and chapter 6, we present our code-generation techniques for safe fluent interfaces. As we have briefly discussed in section 1.4, the existing

studies do not support generics and sub-chaining, both of which are necessary to use the code-generation in the real world. Chapter 5 corresponds to the paper published at the Art, Science, and Engineering of Programming. Chapter 6 corresponds to our paper published as an article in Journal of Computer Languages, which is the extended version of our another paper published at the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences. The following summarizes our contibutions presented in chapter 5 and chapter 6:

- We have developed a code-generation technique that can generate generic fluent interfaces. To realize this, we developed an algorithm that analyzes binding time of type parameters in a deterministic finite-state automaton (DFA). Since the necessity of such an analysis is newly discovered by our study, there is no similar methods as far as we know.

- We have developed a code-generation technique that supports rich sub-chaining interfaces. The generation technique is modeled as the construction of single-state real-time deterministic pushdown automata (RPAs). Our RPA-construction method is different from the literatures [34, 64] in that it does not add or remove non-terminals from a given grammar, which is an essential property to generate sub-chaining interfaces as specified in the grammar.

Chapter 7 concludes our research. Specifically, we highlight the results of empirical studies and summarize the overview of our code-generation techniques. In this chapter, we also discuss possible directions of future work to complement our empirical results and bring the code-generation into the real world.

Chapter 2

Fluent Interface

2.1 Terminologies

The word "fluent interface" is coined by Evans and Fowler [20] in 2005. At that time, fluent interfaces were not a common design according to the literature [20]. In 2010, fluent interfaces were referred to as a promising design style of library interfaces that is known in industry but less known in the research community [83]. In 2020, fluent interfaces can be considered as a knwon term even in the research community thanks to the studies in recent years [5, 9, 28, 29, 32, 41, 42, 84].

However, the definition of a fluent interface and the ones of related terms have not been fixed yet. The meaning of each term differs from one material to another although they share the basic concepts. Such a fluctuation in terminologies is often seen in the definition of a design style of software components. Although the fluctuation may be inevitable since many designs were firstly born and raised in real-world software development as best practices and later named by some opinion leaders or researchers, the discussion about a style often fails to reach an agreement because of mutual misunderstandings.

To avoid misunderstandings, we begin this chapter with our terminologies in this dissertation. Concretely, we (roughly) define three terms: Method chaining, fluent interface, and safe fluent interface.

2.1.1 Method Chaining

Method chaining is a programming style in which multiple method invocations are chained in a single expression as follows:

```
Value topLeft = table.getRowAt(0).getValueAt(0);
```

Method chaining is often interchangeably used with method cascading, but
we distinguish one from the other: Method chaining is a chain of method in-
vocations connected by method invocation operators such as the dot symbol
in Java and the arrow symbol (`->`) in PHP; method cascading is a chain con-
nected by special operators for cascading such as the semicolon operator in
Smalltalk [3] and the double-dot operator (`..`) in Dart [49]. While method
chaining requires a method invocation to return an object, the method cas-
cading syntax allows programmers to chain a method that does not return
any object.

2.1.2   Fluent Interface

As we described at the very beginning of this dissertation, fluent interfaces
are a design style of library interfaces that is designed to be used by method
chaining. This definition is almost the same as the one introduced by Grig-
ore [32]:

> We say that it has a fluent interface when it encourages its users
> to chain method calls

Grigore's definition is slightly different from our understandings[1] of the def-
inition of Fowler [20]. Fowler's definition is more narrow than Grigore's in
that Fowler's rejects method names like `setFoo` in fluent interfaces. How-
ever, real-world fluent interfaces often contain method named like `setFoo`.
Therefore, we define fluent interfaces as mentioned above.

Note that, as mentioned in the blog post by Fowler [20], method chaining
and fluent interfaces are not equivalent although they are often confused in
many materials. Method chaining is about how programmers write source
code, whereas fluent interfaces is about how library developers design the
interface of a library. Technically, any method invocations can be chained
as long as they return objects. Therefore, most object-oriented libraries
can be used in the method-chaining style, but using a non-fluent library –
whose interfaces are not designed for the method-chaining style – would be
troublesome. On the other hand, a fluent interface can be used without
chaining method invocations.

---

[1]In our understanding, Fowler did not give a clear definition of a fluent interface. Instead,
he showed examples of what he think fluent interfaces are and describes the difference
from mere method chaining.

2.1.3 Safe Fluent Interface

In this dissertation, we refer to a fluent interface with the misuse-detectable architecture as a safe fluent interface. We use this term to explicitly distinguish ones with the misuse-detectable architecture (e.g., fig. 1.3) from naively-designed ones (e.g., fig. 1.5). When the safeness is obvious from the context, we refer to safe fluent interfaces as simply fluent interfaces. For example, we say "code-generation for fluent interfaces", not "code-generation for safe fluent interfaces", since the code-generation techniques discussed in this dissertation are always for safe fluent interfaces. In the studies [28, 29, 32, 42, 84], safe fluent interfaces are simply called fluent interfaces since the safe property is prerequisite for thier studies.

2.2 Existing Discussions on Fluent Interfaces

While the advantage of fluent interfaces is described from various perspectives, all arguments essentially claim the high readability of source code written with fluent interfaces, i.e., expressions in the method-chaining style. In other words, the advantage of fluent interfaces is actually the advantage of method chaining.

The readability of source code has a significant impact on the productivity of programmers. In the book [46], Martin puts the importance of readable code as follows:

> Indeed, the ratio of time spent reading versus writing is well over 10 to 1. We are constantly reading old code as part of the effort to write new code. ...(Therefore,) making it easy to read makes it easier to write.

While the precise ratio is arguable, the readability is certainly important as we do not write a line without reading code around that line. A lot of studies have been conducted to develop code-readability and code-quality metrics [8, 66, 71], but no metric system is designed for scoring the readability of a single expression, to the best of our knowledge. Existing metrics are rather for measuring code quality of a set of source code in a project. Due to this difficulty, the positive opinions on fluent interfaces are subjective and qualitative, which leads us to the background survey presented in the next chapter.

The negative opinions on fluent interfaces and method chaining are not only about code readability. For instance, in the thread on Stack Overflow [60], many programmers claim the disadvantages of fluent interfaces

and method chaining from the viewpoint of code readability, tool supports, and potential risks using fluent interfaces or method chaining. The online materials [7, 65] discusses the disadvantages from library developers' perspective.

In the rest of this section, we detail existing discussions of both sides. As we described in the previous section, the terminologies are often different from ours; Some materials blame fluent interfaces (their terminology), but they blame method chaining in our terminologies. To avoid confusion, we rephrase the original sentences and review the opinions.

2.2.1   Positive Opinions

Method chaining is a programming style to eliminate redundant temporary variables and code repetitions [60]. It assembles related method invocations in a single expression [83]. The following code snippet in Java illustrates these benefits:

```
// Method chaining
new AlertDialog()
  .setTitle("Warning")
  .setMessage("Are you sure?").show();

// Without method chaining
AlertDialog dialog = new AlertDialog();
dialog.setTitle("Warning");
dialog.setMessage("Are you sure?");
dialog.show();
```

As we see, the temporary variable `dialog` is not used in the method chaining style. All the method invocations for the `dialog` construction are assembled into a single expression. Fewer occurrences of temporary variables contribute to the code readability since they allow the programmers to keep only fewer variables in mind [60]. These apparently small improvements on source code affect a lot when dealing with many objects in the real-world settings.

The materials [20, 7, 83, 5] claim that a chain of method invocations are often easy to read from left to right as natural-language texts. For example, we can easily read the following code from left to right and understand that the elements of `strList` are filtered out and a function is applied to the remaining elements:

```
strList.filter(s -> s != "").map(s -> s + ".");
```

In the functional nesting style, the same computation is expressed in the reverse order, which possibly impose a cognitive burden on the programmers:

```
map(filter(strList, s -> s != ""), s -> s + ".");
```

Some libraries such as jOOQ[2] make good use of the left-to-right property to embed a domain-specific language (DSL, relatively small language designed for a specific domain) in an object-oriented language:

```
// "SELECT name FROM user WHERE id = 1" with jOOQ
select(field("name")).from(table("user"))
                      .where(field("id").eq(1));
```

As demonstrated by jOOQ, fluent interfaces is a popular means of implementing DSLs inside an (general-purpose) object-oriented language. However, it is not the only option that introduces a domain-specific notation into general-purpose programs, for which syntax extension is a well-known solution. SugarJ [17] provides a method for extending Java syntax. ProteaJ [38], Wyvern [58], and Honu [68] are programming languages that natively support syntax extension. Using these syntax extension mechanisms, domain-specific notation can be embedded as is. When embedding a language as a fluent interface, such notation needs to be transformed into a method chain that differs slightly from the original notation. However, their powerful features are realized by their underlying language mechanism such as type systems, so it is difficult to introduce a similar system to a language that is currently used in practice. A fluent interface, on the other hand, is a technique that can be applied to a number of general-purpose languages since it is just a class library. Moreover, it only requires method invocation syntax such as `obj.method(...)`, which is offered by most object-oriented languages.

Although the cognitive ease of the left-to-right readability is less insisted explicitly, library developers have been aware of its advantage before Fowler gives the name to fluent interfaces. Source code in Smalltalk, a programming language born in 1970s, heavily relies on method cascading that offers the left-to-right readability. The iostream library[3] uses the the shift operators (`<<` and `>>`) to send multiple messages to the same object.

2.2.2 Negative Opinions

The readability improvement by method chaining is, however, controversial. In the thread on StackOverflow [60], several posts claim that method

---

[2]https://www.jooq.org
[3]http://www.cplusplus.com/reference/iostream/

chaining rather worsens the readability. For example, a post says:

> If you do everything in a single statement then that is compact,
> but it is less readable (harder to follow) most of the times than
> doing it in multiple statements.

The same post also mentions that a long statement has to be split into multiple lines with indentations. The resulting code is not more readable than the non-chaining style. However, the readability is highly subjective as we mentioned at the beginning. The results of our background survey show the increasing use of the method-chaining style in the real world. Considering those results, most programmers would not see method chaining as a bad practice that should be avoided as much as possible.

Another post in the thread [60] mentions that method chaining is not reconcilable with the debuggers of most modern IDEs, which allows line-level breakpoints:

> You can't put the breakpoint in a concise point so you can pause
> the program exactly where you want it - If one of these methods
> throws an exception, and you get a line number, you have no
> idea which method in the "chain" caused the problem.

A simple workaround is to split a line to give a different line number to each method invocation. Another workaround is to set a breakpoint not on the caller site but on the callee site, which is the body of the called method. However, these workarounds complicate debugging. Our background survey that shows the increasing use of method chaining would be a quantitative basis for developing a special debugging feature for this situation.

Several posts on the thread [60] claim that method chaining violates a design guideline for developing a loosely coupled software known as the law of Demeter [43]:

> For all classes C, and for all methods M attached to C, all
> objects to which M sends a message must be
>
> - M's argument objects, including the self object or
> - The instance variable objects of C.
>
> (Objects created by M, or by functions or methods which M
> calls, and objects in global variables are considered as arguments
> of M.)

In the empirical study [33], the violation of this law has a negative impact on software quality. However, a fluent interface usually does not violate the law although it depends on the implementation of a fluent interface. Therefore, the law of Demeter is not a guideline that discourages fluent interfaces.

A post in the thread [60] says that method chaining often leads to unexpected a `NullPointerException`, an exception thrown on null dereferencing. This potential risk discourages method chaining, but not fluent interfaces since methods in naively-designed fluent interfaces (e.g., fig. 1.5) do no return a null pointer. Methods of safe fluent interfaces do not either. Furthermore, a special language design is implemented in advanced languages such as Kotlin or Swift to reduce this sort of exceptions. In section 4.1, we discuss the impact of introducing the design into Java in a statistical manner.

The blog posts [7, 65] point out the drawbacks of fluent interfaces: A fluent library cannot be extended using inheritance, a common extension mechanism in an object-oriented language [65]. The post [7] also blames fluent interfaces since the library developers need to pay more costs to maintain a fluent library than a non-fluent library. Those problems exists in many languages, but not in increasingly-used languagegs such as Kotlin and Swift. The solutions by language designs are discussed further in chapter 4.

2.3 Existing Code-generation Algorithms

"Keep it simple" is a famous motto in tool and system design. The UNIX philosophy [70] states it as follows: Write programs that do one thing and do it well. Although the statement is about command line programs, it can be applied to the design of class libraries. A small and simple method that performs only a single task is easier to understand for its users and is easier to manage for its creators, compared to a large and complex method that does multiple tasks at once. Furthermore, a collection of small methods is more flexible. Library users can express a combinatory number of processes or specifications by combining small methods, whereas they cannot execute a part of a large method. Method chaining is a suitable style for combining a lot of method invocations since it reduces temporary variables and code-repetitions (as disussed in section 2.2.1).

The problem of the keep-it-simple style is that a combination is not always correct; for example, some method needs to be invoked before another method. The safe design of a fluent interface is a technique to check the violation of rules on a combination of method invocations using a type system.

Table 2.1: List of existing algorithms and tools

	Grammar class	Output language
fluflu[4]	Regular grammar	Java
TS4J [5]	Regular grammar	
EriLex [83]	LL(1) grammar	Scala
Fajita [42]	LL(1) grammar	Java
Alg. of Gil and Levy [28]	LR grammar	Java
Alg. of Gil and Roth [29]	LR grammar	Java
TypelevelLR [84]	LR grammar	C++, Scala, Haskell
ScaLALR[5]	LR grammar	Scala
Alg. of Grigore [32]	(Turing-complete)	Java

Unfortunatelly, the true origin of the safe design is ambiguous. As far as we know, the material explicitly describing the idea is the post by Fowler [20].

The important idea in the code-generation of fluent interfaces is to express the rules on the combination by a grammar (or by describing state machines). Although the true origin of this idea is not clear either, the first academic study based on the view is published by Xu [83] in 2010. Since then, a number of studies and tool development have been conducted. Table 2.1 summarizes those studies and tools. In the rest of this section, we explain each of them and discuss the difference from our studies presented in chapter 5 and chapter 6.

fluflu and TS4J

Fluflu is the first – as far as we know – code generator for fluent interfaces. The input to fluflu is Java source code that describes finite-state machines, which is equivalent to the descriptions in the regular grammar. The project is deprecated now, but the successor Java::Geci[6] have been developed by the same developer.

TS4J [5] generates fluent interfaces from a regular grammar. For the generation, it builds a deterministic finite-state automaton (DFA) from a given grammar and encode the DFA into Java class definitions.

---

[4]https://github.com/verhas/fluflu
[5]https://github.com/phenan/scalalr
[6]https://github.com/verhas/javageci

EriLex

EriLex [83] is the first fluent interface generator presented as a research artifact. It generates fluent interfaces from a given grammar by encoding a real-time deterministic pushdown automaton (RDPA) into class definitions. As EriLex uses RDPA in its encoding process, the supported grammar class is the LL(1) grammar. Although the output is a set of Scala class definitions, the core technique used in EriLex-generated definitions can be exported into Java since the generated definitions do not use Scala-specific features. Fajita [42] is a tool that demonstrates it is in fact exportable into Java.

The drawback of EriLex is that the input needs to be LL(1) in Greibach normal form [31], which is a form that most of manually written grammars do not follow. Furthermore, there is no algorithm to rewrite a grammar into the form required by EriLex as far as we know. Whether given grammar can be rewritten into that form is undecidable [69].

Algorithm of Gil and Levy

Levy and Gil proposed an algorithm to translate an LR grammar to a fluent interface [28]. It first builds a jump-stack single-state real-time deterministic pushdown automaton (JRDPA) and then encodes the automaton into Java class definitions. Since a JRPA can recognize deterministic context-free languages [12] and this class of languages is larger than the class that RPAs can recognize [34].

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

Algorithm of Gil and Roth

In the literature [29], Gil and Roth have proposed another algorithm for an LR grammar to overcome the compilation time problem of their previous algorithm [28]. The newer algorithm uses a compact DAG representation of the tree encoding data structure to emulate a deterministic pushdown automaton on real-time devices such as the Java type system.

TypelevelLR and ScaLALR

TypelevelLR [84] is a tool for translating an LR grammar into a safe fluent interface in Scala, Haskell, and C++. The generated definitions in Scala and Haskell use type classes to encode ϵ transitions of an LR parser into type definitions. The generated definitions in C++ uses C++ templates to encode the ϵ transitions. ScaLALR[7] is the predecessor of TypelevelLR, which builds an LALR parser and generates Scala source code for safe fluent interface.

One drawback of TypelevelLR and ScaLALR is the clever use of advanced type systems. Since most widely-used languages do not offer type systems that are strong as Scala, Haskell, and C++, the technique in those tools is less portable to other languages.

Algorithm of Grigore

Grigore demonstrated that Java's type system is Turing complete [32]. Using this result, he illustrated that a CYK parser [10, 85, 40] can be constructed using Java types. Although he discovered the theoretical upper bound of grammar classes that can be checked by Java's type system, his technique requires a large memory size. This is because it builds the parser on top of the Turing machine implemented by the Java types. The similar approach can be applied to C++ since the type-system of C++ is also known to be Turing-complete [81].

Summary and Comparison to Our Techniques

As we have explained so far, recent research on the code-generation of safe fluent interface has focused on the syntax-checking capability. The study by Xu presented an algorithm to translate LL grammar into a safe fluent interface [83]; Gil and Levy proposed an algorithm to translate LR grammar into a safe fluent interfaces [28], which is later improved by Gil and Roth [29]. The study by Grigore [32] shows the Turing-coompleteness of of the Java type system, which indicates that the Java type system can even check context-sensitive grammar.

Their key concept for checking context-free structures was using nested generics (parameterized types) to represent a stack structure on top of the type system. The role of a type parameter is to represent a stack element.

---

[7]https://github.com/phenan/scalalr

Such a role differs from the ones in real-world fluent interfaces such as AssertJ[8], jOOQ, and j2html[9]. In those manually written fluent interfaces, 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 [16] and help the developers to implement methods that have the same signature but a different return type.

The essential differences from the preceding studies and our techniques is the support for generics and sub-chaining. As we have described in section 1.4, those two types of support is indispensable for real-world use of the code-generation techniques.

In theory, as we demonstrate in section 5.3.1, our technique can generate fluent interfaces with context-free rules by explicitly using type parameters to represent a stack structure. However, the generation of such fluent interfaces is excessively tedious since the users of our technique need to encode a pushdown automaton into the specification manually. (The encoding of a pushdown automaton has been automated in previous studies.)

2.4 Misuse Detection Techniques

Although we focused only on the syntax correctness of a DSL sentence in this chapter, techniques for semantic checking have also been studied. For example, AraRat [27] uses C++ template metaprogramming to allow its users to compose SQL queries that are syntactically correct and type-safe with respect to the database schema. The integration of such semantic checking and our library generation technique is left for future research. We also focused on fluent interfaces in this chapter, but other host language's mechanism such as operator overloading can be used to emulate a DSL sentence in a GPL program (e.g., sqlpp[10]. The advantage of method chaining is that it requires only method invocation syntax, which should exist in an object-oriented language, to the host language.

Typestate analysis [74] is a form of program analysis that verifies whether an operation sequence performed on an object follows specified rules (or protocols). Objects with typestates occur quite frequently. According to the literature [4], 7.2% of Java types define protocols. Various techniques have been proposed to realize typestate analysis. Plaid [75] is a language that inherently provides features for typestate analysis. Because typestate

---

[8]https://joel-costigliola.github.io/assertj/
[9]https://j2html.com
[10]https://github.com/rbock/sqlpp11

analysis is a type of static code analysis, it can be achieved by using general-purpose code analyzers such as FindBugs[11], PMD[12], and QL [2]. Techniques for mining typestate specifications have also been studied to overcome the difficulty of completely defining the specification by hand [13, 67, 25, 24].

Generating a fluent interface can be regarded as a technique for realizing typestate analysis with the type system of a language. It encodes each state of a type into a concrete type definition of a target language. The validity of the operation sequences is checked by the type system of that language. Such a generative approach causes two problems that do not occur when using external analyzers. Firstly, strange names are given to intermediate states, which may confuse API users. Secondly, performance deterioration may occur owing to the increase in the type definitions. These problems have not been studied yet as far as we know and their investigation will form part of our future work. Although the generative approach suffers from these disadvantages, it offers advantages that are not immediately provided by external analyzers. It aids API users in that a method completion system becomes state aware [83, 56], and it is also beneficial to library developers. As the type checker rejects code violating the protocol, developers do not need to add the implementation for handling such cases.

---

[11]http://findbugs.sourceforge.net
[12]https://pmd.github.io

Chapter 3

Method Chaining
in the Real World

As we described in chapter 1, we analyze method-chaining expressions in the real world to reveal the significance of studying fluent interfaces; we do not count the number of libraries that provide fluent interfaces since the increasing number of existing fluent interfaces does not indicate the increasing use of fluent interfaces. The analysis of the use of fluent interfaces is presented as a part of the analysis of method-chaining expressions in section 3.6.

This chapter is organized as follows: We first describe the materials and methods used in our analysis in section 3.1 and section 3.2. We then show and discuss the results of our analysis from section 3.3 to section 3.7. Section 3.8 argues the threats to validity, and section 3.9 relates our work and preceding studies. Finally, section 3.10 summarizes this chapter and discusses future work.

3.1   Dataset

To build our dataset, we collected 2,814 Java repositories on GitHub. Those repositories are the ones that were listed at least once in the most-starred 1,000 Java repositories on GitHub between Nov. 10th, 2019 and Dec. 21st, 2019. We collected them by monitoring the response of the GitHub API[1] every day during that period.

We built our dataset by extracting syntactically valid `.java` files from

---

[1]https://api.github.com/search/repositories?q=language:java&sort=stars

---

Algorithm 3.1 Dataset construction

Input: Set of repositories *Repositories*
Output: Dataset *Dataset*
  1: **for each** *repository* ∈ *Repositories* **do**
  2:       *Name* ← Name of *repository*
  3:       *Revisions* ← Year-end revisions of *repository*
  4:       **for each** *revision* ∈ *Revisions* **do**
  5:             *year* ← Year of *revision*
  6:             **for each** `.java` file *file* in *revision* **do**
  7:                   *code* ← Content of *file*
  8:                   Add (*code*, *name*, *year*) to *Dataset*

---

the year-end revisions of each repository in the most-starred repository set. The year-end revision of a year is the latest revision made in that year. We classified a `.java` file as syntactically valid when JavaParser[2], a parser often used both in industry and academia, successfully parses the content of that file. To find year-end revisions, we use the command `git rev-list <branch>`, which lists all the revisions reachable from `<branch>`. We set `<branch>` to the default branch[3] of the repository. The default branch of a repository is different depending on the configuration of the repository on GitHub, but it is `master` in most repositories.

An entry of the dataset is a tuple (*code*, *name*, *year*), where *code* is the content of a source file, *name* is the name of the repository to which the file belongs, and *year* is the year of the revision to which the file belongs. Algorithm 3.1 shows pseudocode for constructing our dataset from a given set of repositories. The set of the most-starred repositories were used as the input to this algorithm.

Our dataset contains over three million Java files (approximately seven hundred million source-code lines) in total. Figure 3.1 shows the number of repositories, files, and lines in each year. As seen in the figures, the amount of the collected code considerably varies from year to year. This property of our dataset indicates that it is inappropriate to directly compare the raw numbers in each year.

---

[2]https://javaparser.org
[3]https://help.github.com/en/articles/setting-the-default-branch

(a) Number of repositories

(b) Number of files
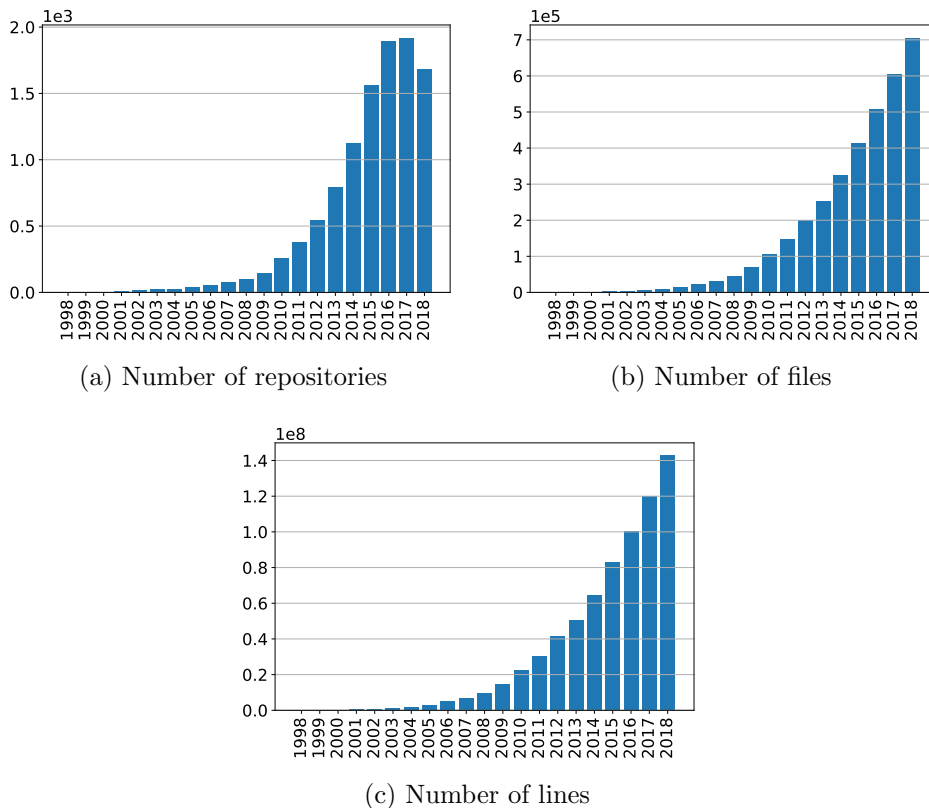
(c) Number of lines

Figure 3.1: Number of repositories, files, and lines

3.2   Definition of Chain Length

We define a method chain as a sequence of one or more method invocations joined by the "." symbol. We define the length of a method chain as the number of invocations in the sequence. For example, the Java code snippet shown in fig. 3.2 contains five chains of length 1, one chain of length 2, and one chain of length 3. We used JavaParser to parse a Java file and mined the parsing result for method chains.

We first enumerated the chains of length 1 (non-chained method invocations) from our dataset to obtain the baseline values. If the number of non-chained invocations increases at the same pace as the number of method chains longer than one, we cannot argue that the use of method chaining is growing. Note that, for convenience, we below regard a non-chained invocation as a method chain of length 1.

```
1  List<String> list = new ArrayList();
2  list.add(
3     createRandomString() // Length = 1
4  ); // Length = 1
5  list.stream().map(s -> {
6     return s.replace("foo", "bar")
7              .replace("baz", "qux"); // Length = 2
8  }).forEach(s ->
9     int n = s.split("\n").length // Length = 1
10    String t = String.format("%d", n); // Length = 1
11    System.out.println(t); // Length = 1
12 ); // Length = 3
```

Figure 3.2: Method chains and their lengths

3.3 Overall Trend

We measure the frequency of method chaining to judge whether it is widely used in the real world. In our analyses, we use only the code that is newer than or in 2010. 2010 is the year in which the number of repositories exceeds 250 and in which the number of files exceeds 10^5 for the first time. We adopt this criterion to avoid that the programming styles of a small number of old repositories overly affect our analysis.

We use the following two indicators f_n and r to measure the frequency:

$$
\begin{aligned}
f_n &= m_n/m_1, \\
r &= (\Sigma_{2 \geq n} n m_n)/(\Sigma_{1 \geq n} n m_n),
\end{aligned}
$$

where m_n is the raw number of method chains of length n. The indicator f_n is the relative occurrence of chains of length n. The indicator r is the ratio of the method invocations that are part of method chains longer than 2, among all the method invocations. For example, the f_2 value of the code in fig. 3.2 is $1/5$ ($= 0.2$) since the code includes five chains of length 1 and one chain of length 2. The r value of the example code is $5/10$ ($= 0.5$) since the code includes ten method invocations in total and five of them constitute the method chains of length 2 or 3.

Figure 3.3a shows the plot of f_n for the code in 2010 and 2018. (The f_n values are computed over the total dataset and not the averages of per-file values.) The horizontal axis shows the value of n and the vertical axis shows

(a) f_n in 2010 and 2018

(b) r
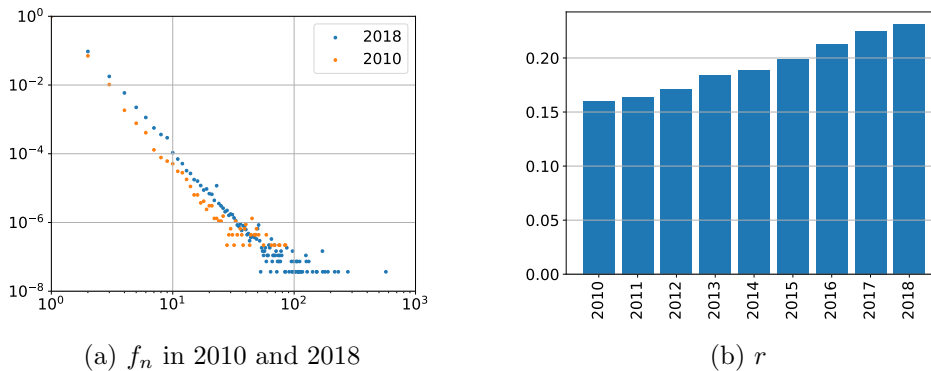
Figure 3.3: Frequency of method chaining

the value of f_n. Note that we use logarithmic scales for both vertical and horizontal axes. Figure 3.3b shows the plot of r from 2010 to 2018. Although we omit the plot of f_n values from 2011 to 2017, similar distributions are observed in those years.

Both charts in fig. 3.3 show the increasing use of method chaining. The relative number of method chains has increased from 2010 to 2018 in almost all lengths. The r value has increased from 16.0% to 23.1%. Further, the maximum length of a chain has also been increased from 2010 to 2018.

3.4 Power-law Distribution

As seen in fig. 3.3, f_n decreases almost linearly in the log-log scale plot. Such a linear decrease is observed when a distribution has a heavy tail [57]. The heavy tail indicates that extremely long chains often appear and their occurrences are not exceptional, unlike a normal distribution. Such distributions are found in several measures of source code such as change sizes [30, 44, 82], component sizes [35], in-degree and out-degree in dependency networks [45], and the number of subclasses [79].

We performed the Kolmogorov-Smirnov (KS) test to see whether the right tail of f_n in 2018 is generated by a power-law distribution, a well-known heavy-tailed distribution. The test reported 9 for x_{min} and 0.937 for p-value. The p-value is greater than the commonly used significance level 0.05. These results of the KS test indicate that it is consistent to assume that the observed values for $n \geq 9$ are generated by a power-law distribution. Although it is interesting to discover the generation model of such a distribution, we leave it for future work.

(a) Distribution

(b) Trend of quartiles
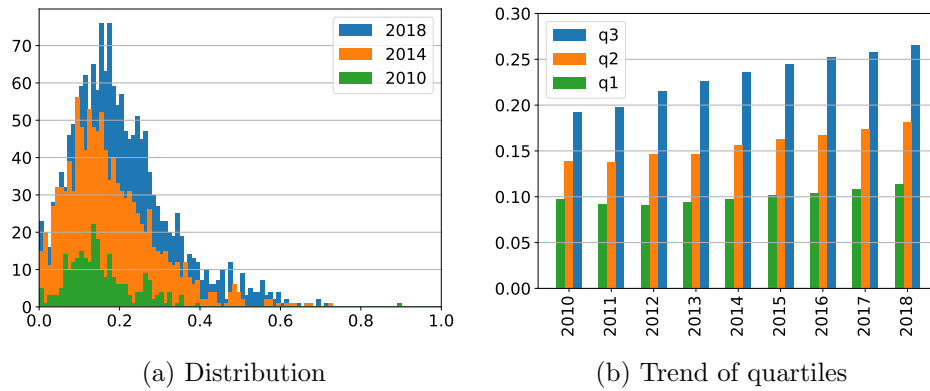
Changes of quartiles from 2010 to 2018

1st quartile	2nd quartile	3rd quartile	Average
+1.71%	+4.27%	+7.34%	+7.08%

Figure 3.4: Distribution and trend of per-repository r

3.5 Bias in Frequency

Since the two indicators shown in fig. 3.3 represent average values for the repositories in our dataset, we observed only average trends in our dataset. In this section, we examined the following hypothesis: Only a small number of large repositories in our dataset might contain a large number of method chains and increase the total number of chains in our dataset, while others contain a small number of chains. To reveal the bias of the increase found in fig. 3.3, we computed the r value for each repository and carried out the analysis on their distribution.

Figure 3.4a shows the histograms of per-repository r in 2010, 2014, and 2018. The horizontal axis shows the value of r and the vertical axis shows the number of repositories. Figure 3.4b shows the trend of their quartiles.

All the information in Figure 3.4 shows that the overall r value is increased not only by a few repositories. If the increase has occurred only in a small number of repositories, we would not have observed changes in the first and second quartiles. However, the increase of overall r value is increased significantly by repositories containing a lot of method chains since the change in the third quartile is much larger than the change in the first quartile.
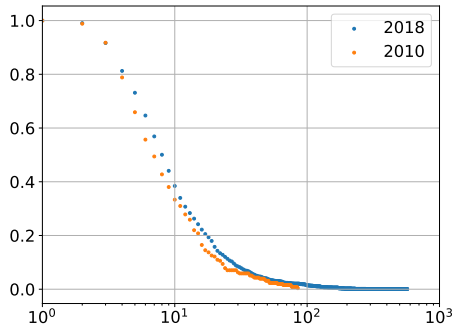
n	u_n in 2018	u_n in 2010
1	100%	100%
8	50.1%	42.7%
9	44.1%	38.0%
41	5.10%	4.31%
42	4.98%	4.31%

Figure 3.5: Ratio containing chains longer than or equal to n

To see the widespread use from a different perspective, we calculated the following value:

u_n: The ratio of repositories that contain one or more method chains whose length is longer than or equal to n.

The left chart in fig. 3.5 shows the plot of u_n in 2010 and 2018. The horizontal axis shows the value of n and the vertical axis shows the value of u_n. Note that we use a logarithmic scale for the horizontal axis. The table on the right shows the values that we refer to in later analyses.

Figure 3.5 shows that more than 50% of repositories contain at least one chain longer than length 7. Since chains of length 8 are unlikely to be composed by programmers who tend to avoid method chaining, this result is another supportive evidence for the widespread use of method chaining.

We also investigated the difference in the use of method chaining between testing code and non-testing code. Figure 3.6 shows the plot of f_n values and the changes in r values in those code sets.

The figures show that the testing code contains more method chains longer than 2. The increasing amount in the testing code (+8.57%) is larger than the one in the non-testing code (+5.37%). On the other hand, the maximum length in the non-testing code is longer than the one in the testing code. Although there are those differences, the increasing trends and the heavy-tailed distributions can be seen in both code sets. From these results, we concluded that method chaining is used not only in either code set but in both sets.

(a) f_n of non-testing code

(b) f_n in testing code

(c) r
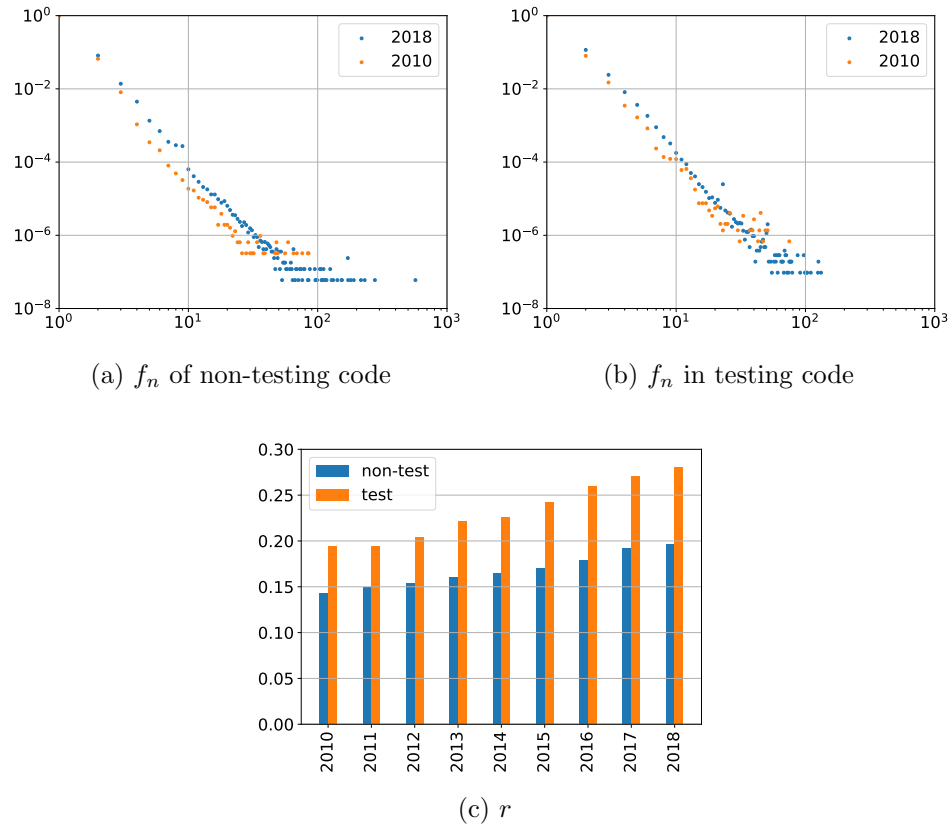
Figure 3.6: Non-testing code vs. Testing code

Table 3.1: Groups of Method Chains

	SHORT	LONG	EXTLONG
Length range	$1 < $ len. ≤ 8	$8 < $ len. < 42	$42 \leq $ len.
# of chains in 2018	3,343,781	19,084	280
# of chains in 2010	384,549	1,106	27

3.6   Categories

To better understand the trends in method chaining, we manually catego-
rized the method chains in 2018 and 2010 by their behaviors (the action
that a method chain performs). The manual inspection is required since it
is hard to automatically categorize a method chain by its behavior.

```
// Access
jfc.getCategoryPlot().getRangeAxis();
miniCluster.getNameNode().getNamesystem()
    .getBlockManager().getDatanodeManager()
    .getNumStaleNodes();
histogram.getBuckets().get(0).getKey();
getSubscriptionAttributes()
    .getInterestPolicy().isCacheContent();

// Access for operation
index.getLibraries().add(libIndex);
getSupportActionBar()
    .setDisplayShowTitleEnabled(false);
```

ACCESSOR. A chain where all methods perform data access except for the last method. Although a chain in this category violates the law of Demeter [43], it frequently appears in the real world code.

Figure 3.7: ACCESSOR chains

For this analysis, we divided the set of method chains into three groups by their length: SHORT, LONG, and EXTLONG. Table 3.1 shows the range of lengths and the number of chains for each group. We chose the border length 8 and 42 in consideration of u_n values, the ratio of repositories that contain one or more method chains whose length is longer than or equal to n. In 2018, more than 50% of repositories contain chains longer than 8, and less than 5% of repositories contain chains longer than 42, as shown in fig. 3.5.

Since it is not feasible to manually inspect all the chains in SHORT and LONG, we analyze randomly-sampled 280 method chains in each of those groups. We analyze all the chains in EXTLONG.

We categorized a method chain into either of ACCESSOR, BUILDER, ASSERTION, and OTHERS. Figure 3.7, fig. 3.8, and fig. 3.9 describe the first three of them by example. OTHERS is the category for chains that do not match any of ACCESSOR, BUILDER, and ASSERTION.

Figure 3.10 illustrates the ratios of each category in SHORT and LONG. The black bars in the figure indicate the margin of errors due to the sampling at a 95% level of confidence. Since we did not find any ACCESSOR chain in the samples of LONG, no blue bars are drawn in LONG. The error bar for

```
// java.lang.StringBuilder
sb.append("New").append(kindName).append("Array");

// com.google.common.base.MoreObjects
MoreObjects.toStringHelper(this)
    .add("iLine", iLine)
    .add("lastK", lastK)
    .add("spacesPending", spacesPending)
    .add("newlinesPending", newlinesPending)
    .add("blankLines", blankLines)
    .add("super", super.toString())
    .toString()
```

BUILDER. A chain that builds an object. It often ends with the invocation
of a method named like `buildFoo` or `toFoo`.

Figure 3.8: BUILDER chains

```
// Mockito
when(myHttpClient.execute(capt.capture()))
    .thenReturn(myHttpResponse);
verify(map).containsKey("testOk");

// AssertJ
assertThat(kunaTimeTicker.getTicker()).isNotNull();
Assertions
  .assertThat(actualObj.has("outline_colors"))
  .isTrue();
```

ASSERTION. A chain that describes expected behaviors of an object. Un-
derstandably, such chains are written in test code. We found usages of the
two libraries Mockito and AssertJ in the sampled set for this category.

Figure 3.9: ASSERTION chains
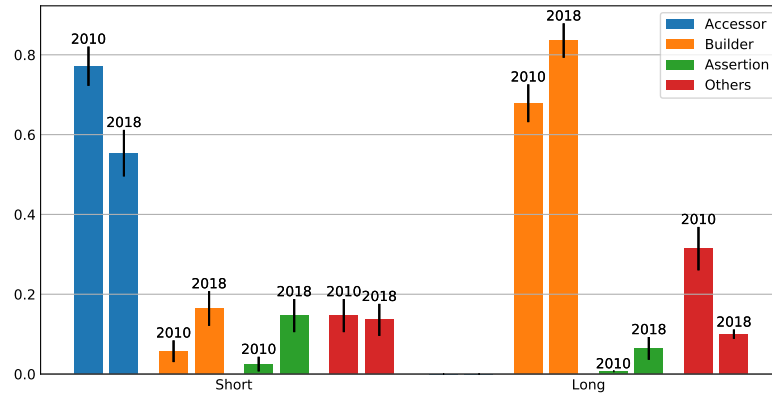
Figure 3.10: Constitution ratio of each category

ASSERTION in LONG in 2010 is not drawn since the number of chains is too low to compute its valid margin. All the chains in EXTLONG are categorized into BUILDER in both 2010 and 2018.

Figure 3.10 shows that approximately 80% of chains in SHORT are categorized into ACCESSOR in 2010. In 2018, the ratio of ACCESSOR decreases to approximately 55%, and the ratios of BUILDER and ASSERTION increase accordingly. These changes in the ratios could be explained by the general acceptance of fluent interfaces, API design that encourages its users to chain method invocations [20, 32]. An ACCESSOR chain can be composed even when the library is not fluent. On the other hand, a BUILDER/ASSERTION chain needs a fluent interface to compose. In Java, object building and assertions are used to be written in the non-chaining style as follows:

```java
// new Builder().setFoo("a").setBar("b").build();
Builder b = new Builder();
b.setFoo("a");
b.setBar("b");
Object o = b.build();

// assertThat(list).contains("a").contains("b");
assert list.contains("a");
assert list.contains("b");
```

Thus, the increase in the ratio of BUILDER/ASSERTION chains indicates

the increasing use of fluent interfaces. The same trends can be seen in the changes in Long. The ratio of Others considerably decreases as the use of Builder and Assertion chains increases. The increasing use of fluent interfaces is supportive evidence for the wide acceptance of method chaining.

Although we expected to frequently encounter the use of the Stream API in Others, we found only two such chains (0.71%) in Short and three chains (1.07%) in Long in 2018. We found no Stream API usages in 2010 as the API is not yet introduced into Java in 2010.

3.7   Extremely Long Chains

Since we did not immediately see why and how such long chains exist in the real-world code, we conducted further inspection of the chains in ExtLong in 2018.

How much of the ExtLong chains are in testing code?

We found 140 chains (50% of ExtLong) in testing code. As mentioned above, all the chains in ExtLong are composed to build an object. Thus, half of the ExtLong chains build objects for testing.

Are the ExtLong chains machine-generated?

We found only five generated chains (1.79% of ExtLong). All of those chains are generated by `aws-java-sdk-code-generator`. Most chains in ExtLong are very likely to be written by human programmers, as far as we can see from the git history.

Which libraries produce the ExtLong chains?

To answer this question, we checked the package name of the first method invocation of each chain. We found 71 different package names, which indicates that various libraries are used to compose extremely long chains. However, a large bias exists in the number of appearances of each library: 53.5% of the libraries are used only once; three libraries constitute 43.9% of the appearances. The following summarizes the most-used three libraries:

> **Elasticsearch**[4] We found 75 chains (26.8%) using `XContentFactory` or `XContentBuilder` in this library. Those classes are for building data

---

[4]https://github.com/elastic/elasticsearch

used in Elasticsearch, which is a distributed search and data-analytics engine.

Guava[5] We found 30 chains (10.7%) using immutable collection builders (e.g., `ImmutableSet.Builder` and `ImmutableList.Builder`) in this library.

Java Std. Lib. We found 18 chains (6.43%) using `StringBuilder` or `StringBuffer` in `java.lang`, both of which represent a mutable sequences of characters in Java.

Are the ExtLong chains styled nicely?

To improve the readability of extremely long chains, programmers often introduce semantical indentations as follows:[6]

```
String jsonString = new PrettyJSON()
  .array()
    .object()
      .key("method")  .value("POST")
      .key("to")      .value("...")
      .key("body")
        .object()
          .key("key").value("ID")
          .key("value").value("fra")
          ... # Our comment: Omitted
        .endObject()
      .key("id")      .value(0)
    .endObject()
    ... # Our comment: Omitted
  .endArray().toString();
```

Another technique is to insert empty lines and comments to group semantically related part as follows:[7]

```
return new SpacingBuilder(
  settings, BallerinaLanguage.INSTANCE)
```

---

[5]https://github.com/google/guava

[6]https://github.com/neo4j/neo4j/blob/a43b26fac61c59da813ec9302a24dd86f6657537/co mmunity/server/src/test/java/org/neo4j/server/rest/BatchOperationIT.java#L501

[7]https://github.com/ballerina-platform/ballerina-lang/blob/27292e84b9f661da89b6c 66840802f2196decb0d/tool-plugins/intellij/src/main/java/io/ballerina/plugins/idea/for matter/BallerinaFormattingModelBuilder.java#L328

```
// Keywords
.around(IMPORT).spaceIf(true)
.around(AS).spaceIf(true)
.around(CHECK).spaceIf(true)
    # Our comment: Empty line for segmentation
.around(ABORTED).spaceIf(true)
.around(COMMITTED).spaceIf(true)
.around(LISTENER).spaceIf(true)
... # Our comment: Omitted
```

We counted the number of chains that are styled nicely as shown above. Our inspection revealed that 184 chains are nicely-styled, which is 66.9% of handwritten chains.

3.8   Threats to Validity

Internal Validity

The validity of ratios shown in section 3.6 and section 3.7 highly depends on our manual inspection of method chains. To openly discuss the validity, we made our results of the inspection publicly available at Zenodo, a general-purpose open-access repository. The details on the publicly-available data is provided at the end of this chapter.

External Validity

We did not apply any filter (e.g. filter by project domains) to the collected repositories. This supports the generalizability of our results. However, the trends in other languages would be different especially when a language provides special constructs to build a domain-specific language (DSL). Since method chaining is often regarded as a technique to design a DSL embedded in a host language, method chaining may not be used if the language provides such special constructs (e.g., the literature [14, 36]). Our results are more likely to be applied to a language that does not provide such a construct (e.g. PHP and JavaScript). The empirical study of this hypothesis is import future work.

3.9   Related Work

Heavy-tailed distributions are found in a number of source code measures [44, 45, 79, 82, 35, 30, 11, 51]. For example, the study [30] shows that such

distributions are found in the lexical properties of source code such as the number of lines and changed lines. The studies [45, 79] show that they are found in the structural properties such as the number of subclasses and the in-degree and out-degree in dependency networks. The paper [44] says

> if one were to analyze the distribution of another measure of software, it would be most surprising to find it *not* following a power law or other heavy-tailed distribution.

However, none of those empirical results can describe the power-law distribution we observed in the number of method chains.

While a number of power-law distributions have been reported, the generation model of those distributions is less studied. Turnu et al. proposed a modified Yule process to model the evolution of object-oriented system properties [79]. Lin and Whitehead proposed a model based on preferential attachment and self-organized criticality [44]. The model proposal for the number of method chains is future work that is needed to deeply understand method chaining.

The use of language features is often empirically studied. In Java, the use of generics [15, 62], lambda expressions [48], annotations [15], and cast operators [47] has been studied. In the study on lambda expressions [48], Mazinanian et al. investigated not only the use in the real-world code but also the reason by interviewing the authors of the source code. Such a study would be beneficial to better understand the advantages and disadvantages of method chaining. In the literature [76], Tanaka et al. analyze the use of method chaining from the view point of functional idioms in Java.

The study [33] empirically shows that the violation of the law of Demeter has a negative impact on software quality. As pointed out in the Stackoverflow thread [60], chaining method invocations violates the law in most cases. Considering these facts, our results imply that real-world software increasingly becomes error-prone. However, method chaining is said to cooperate well with method completion systems in IDEs and let programmers write code easily and quickly [83, 56]. Further research needs to be carried out that inspects problems and merits in method chaining for the future development of language features and static analyzers addressing those problems.

High development cost is a well-known drawback of a fluent interface [7]. The cost becomes significantly higher when the developers choose to implement typed chaining [22] since a number of class definitions are required to achieve typed chaining. The tools and techniques proposed in the paper [5, 9, 83, 28, 42, 56, 29] help library developers to create a fluent interface instantly from grammar definitions of method chains.

Table 3.2: Materials published at Zenodo

File name	Description
data.txt	List of all the collected chains
metadata.txt	List of files and the number of lines for each file
rq1_short_2010.md	SHORT chains (2010)
rq1_short_2018.md	SHORT chains (2018)
rq1_long_2010.md	LONG chains (2010)
rq1_long_2018.md	LONG chains (2018)
rq1_extlong_2010.md	EXTLONG chains sampled (2010)
rq1_extlong_2018.md	EXTLONG chains sampled (2018)
rq1_2010.csv	Result of our manual inspections (2010)
rq1_2018.csv	Result of our manual inspections (2018)

3.10   Summary

This chapter presented our empirical study of method chaining in Java. Our analysis quantitatively revealed the widespread and increasing trend of method chaining, which indicates the potential demands of fluent interfaces.

The collected method chains and the results of our manual inspections are publicly available as an archive file [54]. Table 3.2 lists the files related to this chapter in that archive file.

Highlights of Our Results

Method chaining is increasingly used in the real world. In 2018, 23.1% of method invocations are part of chains longer than 2, while 16.0% are such invocations in 2010. More than 50% of repositories contain at least one chain that is longer than seven. Approximately 5% of repositories contain chains that are longer than 42. Further, the increase is not caused by a few repositories that heavily use method chaining.

We also observed the increasing use of fluent interfaces, which is a supportive result for the wide acceptance of method chaining. In 2010, approximately 80% of chains are accessor chains, those that can be composed without fluent interfaces. The ratio of accessor chains decreased to approximately 55% in 2018, and the ratio of builder/assertion chains – those that require fluent interfaces to compose – increased accordingly.

All the chains longer than 42 are builder chains. 98.2% of them are very likely to be handwritten, and 65.7% are styled nicely with indentations,

empty lines, and comments. The variety of such extremely long chains is unexpectedly wide. We found that 71 packages are used for composing 280 extremely long chains.

Implications

Our results are supportive evidence for the wide acceptance of method chaining in the real world. If method chaining is commonly considered as a bad practice, the use of method chaining would be the same or decreasing. However, to clearly state that method chaining is accepted, user studies need to be conducted. Such studies are our primary future work.

The above-mentioned implication will motivate the developers of fundamental software such as languages, libraries, and IDEs. It will be a supportive and quantitative ground in the discussion of adding new functions for method chaining. For example, library developers can claim that adding fluent interfaces is beneficial for their users; IDE developers can discuss the priority of supporting breakpoints between method invocations in a chain and of a code formatting feature for long chains. The answer also motivates researchers of tools for developing safe fluent interfaces [28, 29, 42, 56, 52, 83]. The researchers can quantitatively state that their tools and further studies on the tools are beneficial for a number of real-world programmers.

Chapter 4

Desirable Language Designs for Fluent Interfaces

This chapter presents our empirical study of desirable language designs that support the use of fluent interfaces (i.e., that support method chaining). Although we aim to present a concrete list of such language designs first and foremost in this study, we also present alternative API designs (the designs of library interfaces). Changing API costs less than changing a language.

We attempt to find language designs for fluent interfaces by investigating real-world Java code pieces that contain non-chained method invocations. Specifically, we (1) pick a code piece with non-chained invocations, (2) find possible reasons why those invocations are not chained, and (3) look for language designs to transform the piece into the method-chaining style. We selected Java for this study since its syntax is relatively simple and does not provide any special language designs for method chaining. Note that the step (2) and (3) are not always feasible; we could not find clear reasons or language designs for some code pieces.

The code pieces that we investigate are randomly sampled from the dataset described in the last chapter. We manually analyzed randomly-sampled 385 chains and the code around them. Since we are interested in why method invocations are not chained and how to support by language designs, the population of the random sampling was the chains of all lengths (including non-chained invocations) in 2018. Since our analysis process includes reasoning of code pieces, it cannot be done mechanically for all the chains in our dataset.

In the following four sections, we present code patterns that we found in our analysis with their examples and discuss appropriate language designs

or API design for those patterns. In section 4.5, we summarize the statistically estimated ratios of those patterns in the population. Section 4.6 and section 4.7 discuss the validity of our results and related work, respectively. We summarize language and API designs that we find in our investigation in section 4.8.

4.1   NullExceptionAvoidance

When a method invocation may return `null`, a programmer cannot chain all related method invocations; for example, as follows:[1]

```
JAXBMapping mapping = jaxbModel.get(qname);
if (mapping == null){
  return null;
}
return mapping.getType().getTypeAnn();
```

A `NullPointerException` may be thrown by `get(qname)` if one simply chains all the method invocations `get(qname)`, `getType()`, and `getTypeAnn()`. We classified a chain into NULLEXCEPTIONAVOIDANCE when `null`-checking has to be performed on the receiver side of the first invocation in a chain. We found nine chains (2.34%) of this pattern in the sampled dataset.

   The safe call syntax in Kotlin [59] is helpful for this pattern.

```
return jaxbModel.get(qname)? // Safe call
    .getType().getTypeAnn();
```

The code above invokes `getType()` only when `get(qname)` returns a non-null object. With this syntax, a programmer can group semantically related invocations into a single chain. Furthermore, the following negative opinions in the thread on StackOverflow [60] can be addressed by introducing the safe call syntax:

> Chaining different objects can also lead to unexpected null errors. ...there's no guarantee (as an outside developer looking at the code) that getSchedule will actually return a valid, non-null schedule object.

Equivalent syntax also exists in Swift [61] and TypeScript [80]. The syntax is called optional chaining syntax in those languages.

---

[1]https://github.com/corretto/corretto-8/blob/32a35a24e2791bc810a0b4d89ad685c97e 4485fa/src/jaxws/src/share/jaxws_classes/com/sun/tools/internal/ws/processor/model er/wsdl/JAXBModelBuilder.java#L119

Library developers can also provide better user experience by using `Optional<T>` for methods that possibly return `null`. The example code of this pattern can be transformed into the following if `get(qname)` returns `Optional<JAXBMapping>`:

```
return jaxbModel
    .get(qname) // returns Optional<JAXBMapping>
    .map(qname -> qname.getType().getTypeAnn())
    .orElse(null);
```

Since all the related invocations are grouped into a chain, the code above would be easier to understand.

4.2 RepeatedReceiver

Different methods are often invoked on the same object as follows:[2]

```
event.getPresentation().setEnabled(true);
event.getPresentation().setVisible(true);
```

This code repeats the expression `event.getPresentation()`. This repetition is not preferable in terms of code readability. We classified a chain as REPEATEDRECEIVER when the receiver of the last invocation is the same as the previous/next chain. We excluded a chain from this pattern if the receiver is `this` or a class since no code repetition exists in such cases. We also excluded chains where programmers avoid chaining on purpose; for example as follows:[3]

```
// These two statements can be written as a chain,
// however, they are written separately.
// sb: java.lang.StringBuilder
sb.append("hasFilter:");
sb.append(this.hasFilter);
```

In the sampled dataset, we found 33 chains (8.57%) of this pattern.

The method cascading syntax in Smalltalk [3] and Dart [49] is useful for removing such repetitions. With this syntax, the example code of this pattern can be written as follows:

---

[2]https://github.com/eclipse/che/blob/e4d0f9987db58f3d46a3a727b88e601e84a5749b/ide /che-core-ide-app/src/main/java/org/eclipse/che/ide/processes/actions/StopProcessA ction.java#L73

[3]https://github.com/apache/incubator-pinot/blob/09eb0150dec47a28d5a4517e4930183e b5dfd0af/pinot-common/src/main/java/com/linkedin/pinot/common/request/Query Type.java#L567

```
event.getPresentation()
    .setEnabled(true)
    ..setVisible(true); // Dart-style syntax
```

The repetition of a receiver object can also be removed by setting the return value of `setEnabled(...)` to `this`. However, when considering the descriptive role of return types, some may think that it is not desirable to return a value in a method named like `setFoo`. In that case, it might be a good convention to name a normal setter (returning nothing) as `setFoo` and a fluent setter (returning `this`) as `withFoo`. A notable library using that naming convention is `TemporaryCredential` in AWS SDK[4].

4.3   DownCast

When the invocation of a method needs downcasting, a chain is frequently split as follows[5] to make code easily understandable:

```
firstBtn = (Button) findViewById(R.id.firstBtn);
firstBtn.setText(START);
```

If a programmer wants to write a single chain for this operation, one needs to write nested parentheses as follows:

```
((Button) findViewById(R.id.firstBtn)).setText(START);
```

However, the nested parentheses worsen the readability. We classified a chain as DOWNCAST when it contains cast operations. Six chains (1.56%) of this category are found in the sampled dataset.

   Providing a method for downcasting relieves the problem in the DOWN-CAST chains. The following lines show example usage of the downcasting methods:

```
// When destination types are practically known
findViewById(R.id.firstBtn).asButton().setText(START);
// When destination types are unknown
findViewById(R.id.firstBtn).as(Button.class)
    .setText(START);
```

---

[4]https://github.com/aws/aws-sdk-java/blob/dafccf5a1241b5655c542a45eae05a582c
3225de/aws-java-sdk-opsworks/src/main/java/com/amazonaws/services/opsworks/mo
del/TemporaryCredential.java#L186

[5]https://github.com/yaowen369/DownloadHelper/blob/a27944d175cc48ddbe06151db8a
d7cb415e9fa60/sample/src/main/java/com/yaoxiaowen/download/sample/MainActivi
ty.java#L144

As seen above, an expression can be read from left to right easily. Although the solution described here is for library developers, it would be unnecessary if the top type `Object` provides the downcasting method `as(...)`. This is a candidate for language extension to Java for supporting method-chaining style.

4.4 ConditionalExecution

Some methods are invoked only when certain conditions are satisfied; for example as follows:[6]

```
if (buildLogger.isInfoEnabled()) {
  buildLogger.info(message, throwable);
}
```

We classified a chain as CONDITIONALEXECUTION when it is conditionally executed as shown above. We found nine chains (2.34%) of this pattern in the sampled dataset.

The code in this pattern can be transformed into a single chain if the library provides a method that takes a lambda expression as its argument:

```
buildLogger.ifInfoEnabled(
  logger -> logger.info(message, throwable));
```

This workaround is largely adopted in the JavaParser library, and other Java libraries could adopt this workaround. Although we could not find syntax for this pattern in other languages, a syntactic sugar like the following would be useful for this pattern:

```
// Possible syntactic sugars
//    (not in any existing languages)
buildLogger
  .isInfoEnabled() => .info(message, throwable);
buildLogger
  .isInfoEnabled()?? .info(message, throwable);
```

4.5 Estimated Ratios

We statistically estimated the ratio in the population (i.e., whole dataset) from the results obtained from the analysis of the randomly-sampled dataset.

---

[6]https://github.com/raphw/byte-buddy/blob/9364421492e830b883d10d9718d6586480e
35747/byte-buddy-dep/src/main/java/net/bytebuddy/build/BuildLogger.java#L535

Table 4.1: Estimated Ratio of Pattern

Pattern	Ratio	$c = 95\%$	$c = 99\%$
NULLEXCEPTIONAVOIDANCE	2.34%	±1.51%	±1.98%
REPEATEDRECEIVER	8.57%	±2.80%	±3.67%
DOWNCAST	1.56%	±1.24%	-
CONDITIONALEXECUTION	2.34%	±1.51%	±1.98%
	14.8%	±3.55%	±4.66%

Table 4.1 summarizes the estimated ratios. The columns "$c = n\%$" show margins of errors at $n\%$ level of confidence. We put the symbol "-" when the ratio in the sampled dataset is too low to compute the valid margin of errors at that level of confidence. The last row of the table shows the values for the sum of those discovered patterns.

As shown in the table, approximately 15% of chains can be combined into a single chain with other invocations around that chain if the appropriate language designs or API design are provided. We counted the number of chains conservatively. For example, we did not classify the following chain as NULLEXCEPTIONAVOIDANCE:

```
JAXBMapping mapping = jaxbModel.get(qname);
updateFoo(); // Possibly changes states
if (mapping == null){
  return null;
}
return mapping.getType().getTypeAnn();
```

By exchanging the first and second statements, the above code can be transformed into a single chain if the safe call syntax is available. However, this exchange possibly changes the semantics of the code, so we did not count such code for NULLEXCEPTIONAVOIDANCE. This conservative counting implies that the real ratios might be larger than the values shown in the table.

4.6   Threats to Validity

The internal validity of our results (shown in section 4.5) highly depends on our manual inspection of method chains. To publicly discuss the validity, we made our results of the inspection available at Zenodo (see table 4.2 at the end of this chapter for more details). The external validity is the same

as the one in the last chapter (section 3.8) since the dataset we use in this chapter is the same as the one used in the last chapter.

4.7 Related Work

There are more features that help programmers to chain method invocations although only a small number of languages provide those features. D [23] and Nim [63] have the uniform function call syntax, which allows chaining the invocation not only of methods but also functions. The scope functions in Kotlin [72] can be used to compose a chain with conditional statements and loops without storing an intermediate state into a temporary variable.

Class-extension mechanisms without inheritance help programmers to use a fluent library to mitigate the extensibility problem pointed out in the blog posts [7, 65]. Assume that we provide the library for counting numbers implemented as follows:

```
class Counter {
  n = 0;
  Counter increment() { n++; return this; }
}
```

The user of this library can count up a number by chaining the method call `increment()`. Suppose that the user need to add the method `decrement()` to this library. Programmers would use inheritance, a feature implemented in most object-oriented languages, to extend existing code:

```
class ExtCounter extends Counter {
  ExtCounter decrement() { n--; return this; }
}
```

However, this approach does not work as expected since `increment()` returns an instance of `Counter`:

```
new ExtCounter().increment() // Returns Counter
               .decrement(); // Method not found!
```

The user can avoid this problem by overwriting the existing methods in `Counter`, but it is too tedious to overwrite all the existing methods when a given library provides tens of methods. With class-extension mechanisms provided in Swift [19] and Kotlin [18], the user can add `decrement()` as expected.

Table 4.2: Materials published at Zenodo

File name	Description
rq2.md	Sampled chains for manual inspection
rq2.csv	Result of our manual inspections

4.8   Summary

We found four code patterns that disturb programmers to chain more method invocations and listed language designs that eliminate the disturbers. Some readers might think it is obvious that those designs are useful for method chaining. However, what is important in our study is that our analyses quantitatively revealed how effective they are if adopted. Out results shows that at least 10% of non-chained invocations can be transformed into a chain. This quantitative result will be a basis for the design decision when adding those features or adopting the library design.

The results of our manual inspections in this chapter are publicly available at Zenodo [54] as an archive file. Table 4.2 describes the files related to this chapter in that archive file.

Highlights of Our Results

Language designers can effectively save the development effort of their users by implementing the safe call syntax and method cascading syntax. The users can write compact code by chaining method invocations with such syntax. In Java, which is a language that does not support the syntax, approximately 10% of chains suffer from `null`-checking and writing receiver objects repeatedly.

The following summarizes best practices that we propose for Java library developers:

- Return `Optional<T>` when a method may return `null`.

- When creating a method returning a boolean value such as `isFoo()`, create the method `ifFoo(...)` that takes lambda expressions.

- When creating a setter method such as `setFoo(...)`, return `this` in that method instead of returning nothing.

- When creating a public non-final class, provide methods for downcasting such as `as(...)`.

Tool developers can save these efforts of library developers by creating code generators for the boilerplate code.

Chapter 5

Generating Generic Fluent Interfaces

In this chapter, we present a code-generation technique of generic fluent interfaces. As we discussed in chapter 1, the support for generics is indispensable to use the generative approach in the real world since the use of type parameters is common in real-world library development.

We first illustrate the problem of existing code-generation algorithms by example in section 5.1. We then overview our technique and its demonstration tool named PROTOCOOL in section 5.2. The detailed description of our technique follows the overview from section 5.2.1 to section 5.2.4. To evaluate our technique, we generated several generic fluent interfaces and discuss their use cases. We also measured the number of generated lines using our demonstration tool. The results of our evaluation are discussed in section 5.3.1 and section 5.3.2. Section 5.4 summarizes this chapter and briefly discusses future work.

5.1   Problem of Existing Algorithms

Consider creating a generic fluent interface for constructing an instance of `Map<K, V>` in Java. Our example API allows its users to construct a map with any key/value type by chaining the method invocations, as follows:

```
Map<Integer, String> map
  = OurAPI.newMap()
    .put(1, "foo") // Associate "foo" with key 1
    .put(2, "bar") // Associate "bar" with key 2
    .build();
```

In our API, the key and value types are inferred from the types of the argument provided to the first invocation of `put(...)` in the chain. Users cannot create an entry with an inconsistent key/value type. A type error is reported if such an entry is created:

```
OurAPI.newMap()
  .put(1, "foo") // key: Integer, value: String
  .put("bar", 2) // key: String, value: Integer
                 // => Type error!
  .build();
```

Our API also has syntactical rules regarding the order of the method invocations in a chain. Users first need to invoke `newMap()` to begin a map construction. Thereafter, they can create entries by chaining an arbitrary number of `put(...)`. They need to invoke `build()` to complete the construction and obtain an instance of `Map<K, V>`. The syntax described above can be summarized as follows, in the form of a regular expression:

$$\text{newMap() } \big(\text{put(K key, V value)}\big) * \text{ build()}$$

The asterisk denotes zero or more occurrences of the preceding element. `K` and `V` are type parameters that represent the key and value types, respectively.

Consider reporting the violation to the syntax as a type error so that users can identify their misuse at compile-time. This safe property can be achieved by setting the return type of each API method based on what the users can chain next. For example, a duplicate invocation of `newMap()` can be prevented by setting the return type of `newMap()` to a class providing only `put(...)`:

```
OurAPI.newMap()   // Returns a type
                  //   that provides only `put(...)`
     .newMap(); // This line causes a type error;
                  //   Cannot resolve method 'newMap'
```

It is known that such a safe property can be achieved by: (1) building a state machine that accepts only a syntactically correct method sequence, and (2) encoding the machine into Java class definitions [5, 28, 83]. Recall that the syntax of our API is expressed by a regular expression. Therefore, a DFA is capable of recognizing the syntax. Figure 5.1 illustrates the DFA that accepts method sequences conforming to the syntax. The left-most state in the figure is the initial state of the DFA. The double circle represents an
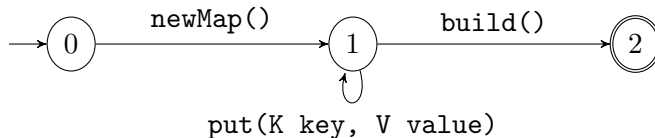
Figure 5.1: DFA that accepts correct method sequences of `OurAPI`

accepting state. The number in each state is simply an index of the state for later references.

A problem arises when encoding the DFA into class definitions. According to the previous studies [5, 28, 83], we obtain the class definitions by encoding each state into a class, and encoding each transition into a method. Consider encoding the loop transition consuming `put(...)` into a method. A naive idea is to encode it into the following:

```
// Corresponds to the state 1 in the diagram
class State1<K, V> {
  State1 put(K key, V value) { /* method body */ }
  ...
}
```

Unfortunately, this encoding does not generate our example API as expected. In this encoding, all invocations of `put(...)` refer to types that are bound to the type parameters `K` and `V`. However, no invocations of `put(...)` bind the argument types to `K` or `V`. The type parameters are never bound to types. Thus, the API generated by this encoding is broken.

Another idea is to encode the loop transition as follows:

```
class State1 {
  <K, V> State1 put(K key, V value) { ... }
  ...
}
```

Using this encoding, an invocation of `put(...)` binds the argument types to `K` and `V`. However, this encoding does not generate our example API either. In this encoding, `K` and `V` are bound at every invocation of `put(...)`. As no invocations refer to a previously bound type, the map entry types become inconsistent. The users of the generated API can put any type of item into the map.

The problem is that, in the state machine naively constructed from syntactical rules, it is not clear which type parameters are already bound in each state. A type parameter in a method chain is bound at the first method

invocation that uses the type parameter. The successive method invocations refer to that bound type. In our API, the type parameters `K` and `V` are bound to the argument types provided to the first invocation of `put(...)`. Successive invocations of `put(...)` refer to those bound types for their arguments.

The binding rule of a type parameter in a chain is not specific to our API. For example, the type parameter in the Stream API is bound to a type, as described above:

```
Stream
  .of("a", "aa", "aaa")
      // Bind String to T of Stream<T>
  .filter(s -> s.length() > 1)
      // Refer to the bound type (= String)
  .forEach(s -> System.out.println(s));
      // Refer to the bound type (= String)
```

To generate a generic fluent interface correctly, an algorithm needs to construct a state machine that knows which type parameters are bound in each state. If the bound type parameters in each state are clear, the encoding algorithm can identify whether or not a generated method newly binds types to type parameters.

5.2   Our Code-generation Technique

To address the problem discussed in the previous section, we have developed a code-generation technique that correctly handles type parameters in a fluent interface. As described in chapter 1, the code generation of fluent interfaces is the translation from a grammar into class definitions. In the translation, a grammar is first converted into a state machine that accepts sequences derived from the grammar, and the machine is encoded into class definitions for fluent interfaces. Although our technique also follows this process, it constructs a special DFA for generics support.

Our technique is implemented as a demonstration tool named PROTO-COOL, and we describe our technique as the process implemented in PRO-TOCOOL. Although the core contribution is the development of an algorithm for the binding-time analysis of type parameters, which we explain in section 5.2.2, we describe all parts from the start for readers to better understand the core contribution.

PROTOCOOL receives the API specification written in Java-like syntax. Figure 5.2 illustrates the specification of `OurAPI`, which is the example API

```
1  class OurAPI {
2    // Defines syntax
3    static Map<K,V> newMap() put(K key, V value)* build();
4    // Define type parameters for this class
5    K;  V;
6  }
```

Figure 5.2: Specification of `OurAPI`
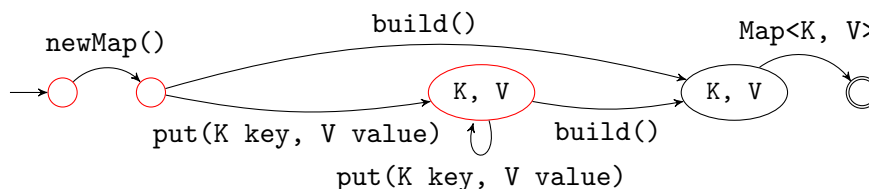
Figure 5.3: DFA constructed from class declaration in fig. 5.2

described in the previous section. The statement on line 3 in fig. 5.2 is a chain declaration, which defines the syntactically correct chaining of the API methods. The keyword `static` merely indicates that the first method of a chain is a static method. The generated API is used as follows if we remove `static`:

```
// "new OurAPI().newMap()"
//    instead of "OurAPI.newMap()"
Map<Integer, String> map
  = new OurAPI().newMap().put(1, "foo").build();
```

The statements on line 5 declare the type parameters used in the API.

According to the given specification, PROTOCOOL constructs a DFA in which each state is annotated with the type parameters bound in that state. Figure 5.3 illustrates the DFA constructed by PROTOCOOL from the specification in fig. 5.2. The symbols indicated inside a state circle are the type parameters bound in that state. The DFA consumes a method or type at each step. It reaches an accepting state by consuming the method sequence defined in the chain declaration, and then by consuming the return type. A transition consuming a type identifies which type is instantiated by chaining methods. The construction of such a DFA consists of two steps. The first step is the naive construction of a DFA from the given syntactical rules. The second step is the modification of the naively constructed DFA. These two

```
1  // Corresponds to the initial state
2  class OurAPI {
3    static State1 newMap() { ... }
4  }
5  // Corresponds to the second state from the left
6  class State1 {
7    <K, V> State2<K, V> put(K key, V value) { ... }
8    <K, V> Map<K, V> build() { ... }
9  }
10 // Corresponds to the third state from the left
11 class State2<K, V> {
12   State2<K, V> put(K key, V value) { ... }
13   Map<K, V> build() { ... }
14 }
```

Figure 5.4: Class definitions generated from DFA in fig. 5.3

steps for constructing a DFA are described in section 5.2.1 and section 5.2.2.

PROTOCOOL generates a safe fluent interface by encoding the constructed DFA into Java class definitions. Figure 5.4 illustrates the class definitions generated from the DFA in fig. 5.3. A state is encoded into a class if the state does not have a transition consuming a type and the state is not an accepting state. In fig. 5.3, only those states colored with red are encoded into classes, as the others have a type-consuming transition or are an accepting state. The initial state is encoded into a class with the same name as the class declaration. In our example, the initial state is encoded into a class named OurAPI, as indicated on line 2 in fig. 5.4. Other states are encoded into classes named, for example, StateN. A transition is encoded into a method if the transition consumes a method. The return type of a method depends on the destination state of the original transition. If the destination state includes a type-consuming transition, the return type is that consumed type. In our example, the return type of build() is Map<K, V>, as indicated on line 8 and line 13. Otherwise, the return type is the class corresponding to the destination state. The generated method includes its type parameter declaration when the type parameters bound in the source state differ from those that are bound in the destination state. Note that, if no put(...) is invoked in a chain, K and V are inferred from the type information outside of the chain. For example, if the return value of OurAP.newMap().build()

(a) Set of chain automata

(b) NFA
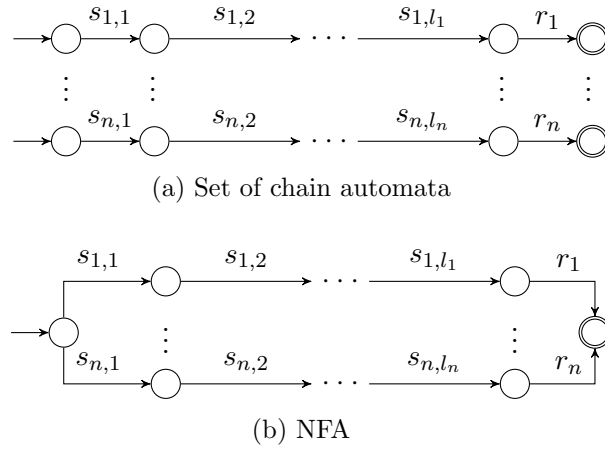
Figure 5.5: NFA construction

is assigned to a variable, K and V are inferred from the type of that variable:

```
// K and V are bound to Integer and String
Map<Integer, String> m = OurAP.newMap().build();
```

5.2.1 DFA Construction

Suppose that a given specification has the following class declaration:

```
class c {
    r₁ s₁,₁ s₁,₂ ... s₁,ₗ₁;
    ... ;
    rₙ sₙ,₂ sₙ,₂ ... sₙ,ₗₙ;
}
```

Here, c is a class name, r_i is a return type, and $s_{i,j}$ is a method signature. In fig. 5.2, OurAPI is c, Map<K,V> is r_1, newMap() is $s_{1,1}$, put(K key, V value) is $s_{1,2}$, and build() is $s_{1,3}$. Note that c, r_i, and $s_{i,j}$ include type parameters. For instance, r_1 corresponds to Map<K,V> on line 3 in fig. 5.2, not only Map. Later in section 5.2.2, we define a function $\pi(s_{i,j})$ that retrieves the set of type parameters appearing in $s_{i,j}$. We omit static and type parameter declarations (line 5 in fig. 5.2) since they are irrelevant to our DFA construction. The keyword static only indicates that the first method of a chain should be a static method, and type parameter declarations is to distinguish type parameter names (e.g., K and V) from type names (e.g.,

`OurAPI` and `Map`). We also omit regular-expression operators such as `*` here. The support for those operators are discussed at the end of this sub-section.

PROTOCOOL first constructs a non-deterministic finite-state automaton (NFA) from the class declaration. It achieves this by combining a set of chain automata, each of which is constructed from a chain declaration. A chain automaton reaches an accepting state by consuming the method sequence of its corresponding chain declaration, and then by consuming the return type. Figure 5.5a illustrates the set of chain automata obtained from the above class declaration. Our algorithm constructs the NFA presented in fig. 5.5b by merging the initial states and accepting states of the chain automata in fig. 5.5a. Such an NFA is always constructed successfully from a given class declaration.

PROTOCOOL then converts the constructed NFA into a DFA using Brzozowski's algorithm [6]. To judge the equality between two transitions, our algorithm uses the function that takes two transitions as its input and returns a boolean value, as follows:

- returns true if both transitions consume methods and have the same signature;

- returns true if both transitions consume the same type; and

- returns false otherwise.

This conversion is necessary to generate a valid Java class definition, as non-determinism produces duplicate method definitions in a class. The conversion always succeeds because any NFA can be converted into a DFA.

The use of regular-expression operators in a input grammar can be supported by using Thompson's algorithm [78] to construct a chain automaton. Although a chain automaton constructed by the algorithm may not be deterministic, PROTOCOOL still can obtain a DFA from a NFA that is constructed by merging chain automata.

5.2.2   Binding-time Analysis

PROTOCOOL finally analyzes the binding times of the type parameters to determine which type parameters are already bound in a state. It incrementally assigns a set of type parameters to each state of the DFA, and *modifies* the DFA constructed at the previous step during the analysis if necessary.

The modification of the DFA is essential to successfully encode a DFA into class definitions with type parameters. Without the modification, the encoding of the DFA causes problems discussed in section 5.1.

(a) Assign empty set to initial state

(b) Assign empty set to second state

(c) Clone second state

(d) Change loop transition destination
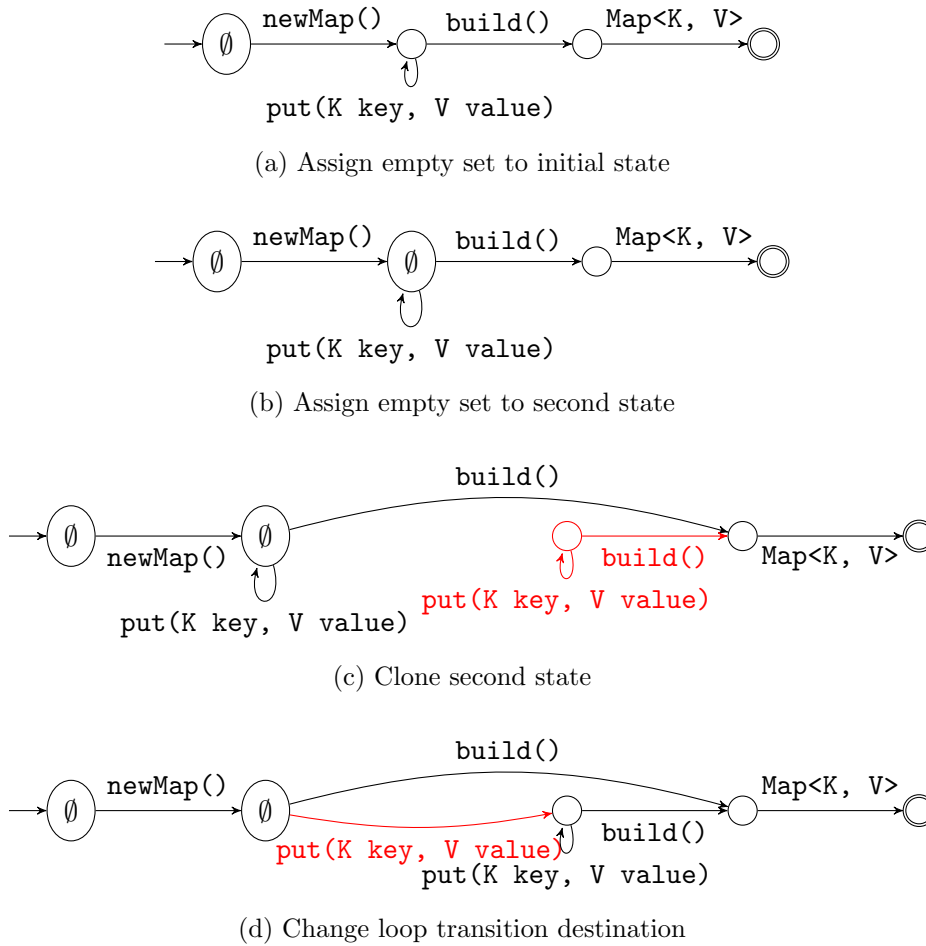
Figure 5.6: Incremental assignment of type parameters

The algorithm for our binding time analysis can be described as follows. The algorithm first assigns the set of type parameters appearing in the class name to the initial state. In the example case, it assigns an empty set $\emptyset$ to the initial state, as illustrated in fig. 5.6a, because `OurAPI` does not have any type parameters in its name. Suppose that a set of type parameters P_i is already assigned to a state q_i, and q_i includes a transition $t : q_i \xrightarrow{s} q_j$. If any set has not been assigned to q_j, our algorithm assigns $P_i \cup \pi(s)$ to q_j, where $\pi(s)$ is a function to retrieve the set of type parameters appearing in s. In the example case, the algorithm assigns $\emptyset$ to the second state, as illustrated in fig. 5.6b, because no set is yet assigned to the second state. If a set of

---

Algorithm 5.1 Binding time analysis of type parameters

Input: A DFA D

Output: A DFA with type parameter information

  1: $Q \leftarrow$ An empty queue
  2: $q_0 \leftarrow$ The initial state of D
  3: Assign the set of type parameters of the declared class to q_0
  4: Enqueue all transitions outgoing from q_0 to Q
  5: **while** Q is not empty **do**
  6:     $(t : q_i \xrightarrow{s} q_j) \leftarrow$ Dequeue from Q
  7:    **if** s is a method **then**
  8:       $P_i \leftarrow$ The set assigned to q_i
  9:      **if** No set is assigned to q_j **then**
10:          Assign $P_i \cup \pi(s)$ to q_j
11:          Enqueue all transitions outgoing from q_j to Q
12:      **else**
13:          $P_j \leftarrow$ The set assigned to q_j
14:          **if** $P_j \neq P_i \cup \pi(s)$ **then**
15:             **if** $\pi(s) = \emptyset$ **then**
16:               Assign $P_i \cup \pi(s)$ to q_j
17:             **else**
18:               $q'_j \leftarrow \gamma(q_j, D)$
19:               Assign $P_i \cup \pi(s)$ to q'_j
20:               Change the destination of t to q'_j
21:               Enqueue all transitions outgoing from q'_j to Q

---

type parameters P_j has already been assigned to q_j and $P_j \neq P_i \cup \pi(s)$, our algorithm changes the destination of t to a state q'_j and assigns $P_i \cup \pi(s)$ to q'_j. Here, q'_j is a newly added state that is obtained by cloning (copying a state including the transitions outgoing from that state) q_j. The cloned state q'_j has the same set of transitions as that of q_j. Figure 5.6c and fig. 5.6d illustrate the cloning process in our example case. The algorithm continues to assign the bound type parameters to a state until all states are annotated with their bound type parameters.

    Algorithm 5.1 presents the pseudo-code of our algorithm. It takes a DFA D as its input and emits a modified DFA as its output. The function $\gamma(q, D)$ on line 18 is the function that clones a state q (in a DFA D) and the transitions outgoing from q. The red state in fig. 5.6c is the state made by the function *gamma*, which is cloned from the second state on the left

```
                          Object_Map
                              /|\

Method_newMap  Method_put  Method_put  Method_build
                   /\           /\

              1  "foo"       2  "bar"
```
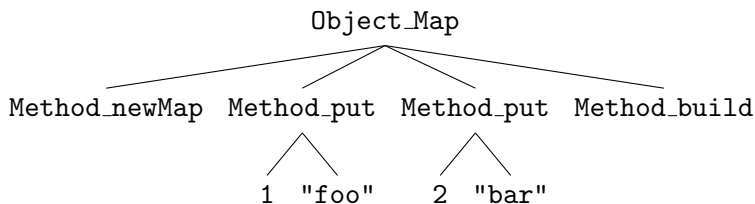
Figure 5.7: Tree construction in generated library

of the red state. Because the cloning process does not add a transition, but only changes its destination, the modified automaton is still finite and deterministic. The binding time analysis does not repeat infinitely. Only a finite number of type parameters exist in a class declaration. A state is mapped to a subset of the set of those type parameters. Therefore, the number of states is also finite.

5.2.3 Bodies of Generated Methods

PROTOCOOL generates method bodies, not only source code for library interfaces, as do the existing fluent interface generators [29, 84]. The generation of method bodies helps generator users to implement the actions of a generated API. Without the body generation, users need to deal with the laborious task of restoring previously implemented actions when they regenerate their API to update the API specification. This problem in code-regeneration is well-kown and the separation of generated code and handwritten code is known as the pattern called generation gap [21].

The generated method bodies construct a tree that represents a method chain composed by the API user. For example, the tree illustrated in fig. 5.7 can be constructed from the following chain:

```
OurAPI.newMap().put(1, "foo").put(2, "bar").build();
```

Each node of the tree represents either an object construction or a method invocation. In fig. 5.7, the root node `Object_Map` represents an object construction, while the child nodes such as `Method_newMap` represent a method invocation. The child nodes of an object construction node are method invocation nodes, each of which represents a method that is invoked to construct the object. The child nodes of a method invocation node are the arguments passed to that method.

Library developers (that is, PROTOCOOL users) can access the tree by specifying a tree evaluator through a `return` clause; for example, as follows:

```
1  class Evaluator {
2    static <K, V> Map<K, V> buildMap(
3      Object_Map<K, V> node) {
4        BuildMapVisitor<K, V> visitor
5          = new BuildMapVisitor<K, V>();
6        visitor.visit(node);
7        return visitor.map;
8    }
9  }
```

Figure 5.8: Handwritten evaluator implementation

```
1   class BuildMapVisitor<K, V> extends Visitor {
2     Map<K, V> map = new HashMap();
3     void visitMethod_put(Method_put<K, V> node) {
4       map.put(node.key, node.value);
5     }
6     void visitObject_Map(Object_Map<K, V> node) {
7       super.visitConstruction_Layer(node);
8     }
9     void visitMethod_newMap(Method_newMap node) {}
10    void visitMethod_build(Method_build node) {}
11  }
```

Figure 5.9: Handwritten visitor implementation

```
class OurAPI {
  static Map<K, V>
    newMap() put(K key, V value)* build()
    return Evaluator.buildMap;
  K; V;
}
```

In this case, `Evaluator.buildMap` is a static method defined by hand outside of the generated code. The constructed tree is passed to the static method placed after the keyword `return`:

```
Map<K, V> build() {
  // Create and store a new method node
```

```
   Method_build method_build = new Method_build();
   ...
   // Create a new object construction node
   Object_Map<K, V> object_map = new Object_Map<K, V>();
   ...
   // Pass the tree to the evaluator method
   //   and return the return value of the evaluator
   return Evaluator.buildMap(object_map);
}
```

Using this design, library developers can implement the actions of the generated API separately from the generated code. The generated tree nodes support the visitor pattern. Figure 5.8 and fig. 5.9 present example implementations of the actions that construct a `HashMap<K, V>` instance using the visitor pattern.

5.2.4 Specification Validation

PROTOCOOL throws an error and does not generate Java class definitions when it detects that a state with a type-consuming transition also has another transition. This is because the specification producing such a DFA cannot be translated into valid Java class definitions, or the specification is translated into unexpected Java classes. In the following, we describe the two cases where PROTOCOOL throws an error along with the reasons for such behaviors.

Multiple Type-consuming Transitions

Figure 5.10a presents an example specification that produces a state with multiple type-consuming transitions. The specification states that the users of the generated API can write both of the following:

```
List<String> list = Collection.of("foo").create();
Set<String> set = Collection.of("bar").create();
```

Figure 5.10b illustrates the DFA constructed from the specification presented in fig. 5.10a. The state colored in red has two transitions that consume types.

Although PROTOCOOL can construct a DFA from the specification in fig. 5.10a, this DFA cannot be translated into valid Java classes. The second state is encoded into the following Java class:

```
class SingletonCollection {
  static List<E> of(E elem) build();
  static Set<E> of(E elem) build();
  E;
}
```

(a) Specification
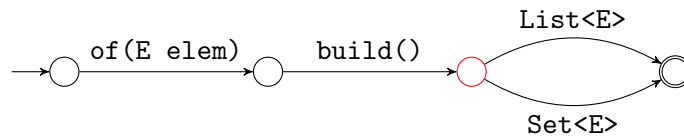
(b) DFA

Figure 5.10: Invalid spec. with multiple type-consuming transitions

```
class State1<E> {
  List<E> build() { ... }
  Set<E> build() { ... }
}
```

In Java, methods with the same signature must return the same type. Therefore, the class definition above is invalid Java code. PROTOCOOL throws an error not to generate a broken API. Note that this error is due to the incompatibility between the given specification and the Java type system, not to our DFA-construction method.

Type-consuming and Method-consuming Transitions

Figure 5.11a presents an example specification that produces a state with both type-consuming and method-consuming transitions. It states that the users can write the following:

```
Map<String, String> map = StrMapBuilder
  .newMap().add("foo", "bar").add("bar", "baz");
```

Figure 5.11b illustrates the DFA constructed from the specification presented in fig. 5.11a. The state colored in red has both type-consuming and method-consuming transitions.

Although a DFA can be constructed by PROTOCOOL, the encoding of this DFA is problematic. The initial state is encoded into the following

```
class StrMapBuilder {
  static Map<String, String>
    newMap() add(String k, String v)*;
}
```

(a) Specification

```
            add(String k, String v)
        newMap()  ↺  Map<String, String>
   ───○──────────→●───────────────────→◎
```
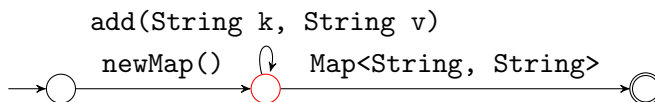
(b) DFA

Figure 5.11: Invalid spec. with type- and method-consuming transitions

class, as the destination of the `build()` transition has a type-consuming transition:

```
class StrMapBuilder {
  static Map<String, String> newMap();
}
```

However, the users of the generated API cannot chain the method `add(...)`, because `Map<String, String>` in Java does not provide `add(...)`:

```
Map<String, String> map = StrMapBuilder
  .newMap() // Returns Map<String, String>
  .add("foo", "bar"); // Type error!
                      // Cannot resolve method 'add'
```

PROTOCOOL throws an error and do not generate code to prevent the generated API from being an unexpected API. Note that, if the output language provides a mechanism to extend existing classes (e.g., the mechanism provided in Kotlin [18] and Swift [19]), this problem can be addressed, and PROTOCOOL does not need to throw an error to this case.

5.3 Evaluation

5.3.1 Use Cases

In this section, we illustrate the ability and limitations, as well as several features that are introduced for the practical applicability of PROTOCOOL, through the generation of three example APIs.

```
<spec>
  → <class>+ ;
<class>
  → <class-head> <class-body> ;
<class-head>
  → "class" NAME <type-param-list>? ;
<type-param-list>
  → "<" <type-param> ( "," <type-param> )* ">" ;
<class-body>
  → "{" <chain-or-type-param>* "}" ;
<chain-or-type-param>
  → ( <chain> | <type-param> ) ";" ;
<type-param>
  → NAME <type-param-bound>? ;
<type-param-bound>
  → "extends" <type-ref-list> ;
<type-ref-list>
  → <type-ref> ( "," <type-ref> )* ;
<chain>
  → "static"? <type-ref> <chain-expr> <tree-eval>? ;
<chain-expr>
  → <chain-term> ( "|" <chain-term> )* ;
<chain-term>
  → <chain-fact>+ ;
<chain-fact>
  → <chain-elem> ( "?" | "*" | "+" )? ;
<chain-elem>
  → <method> | "(" <chain-expr> ")" ;
<method>
  → NAME "(" <method-param-list>? ")" <method-action>? ;
<method-param-list>
  → <method-param> ( "," <method-param> )* ;
<method-param>
  → <type-ref> "..."? NAME ;
<method-action>
  → "{" <qual-name> ";" "}" ;
<type-ref>
  → <qual-name> ( "<" <type-ref-list> ">" )? "[]"* ;
<tree-eval>
  → "return" <qual-name> ;
<qual-name>
  → NAME ( "." NAME )* ;
NAME : [a-zA-Z_][a-zA-Z0-9_]* ;
```

Figure 5.12: Syntax of our API specification language

The concrete syntax and semantics of our API specification language are presented in fig. 5.12. Parentheses are used to group elements. An asterisk represents zero or more occurrences, a plus sign represents one or more occurrences, and a question sign represents zero or one occurrence of the preceding element. A colon is used to define a lexical token. The left-hand side of a colon is the name of the token, while the right-hand side is the regular expression that the token should follow.

Matrix Computation API

The support for type parameters enables examination of a relatively complex API protocol. As an example, we demonstrate the generation of a matrix computation API that reports an incompatible computation as a type error.

Our matrix computation API provides two classes, namely `IntMat` and `FltMat`, which represent an integer matrix and a float matrix, respectively. The API supports matrix addition and multiplication. Only a matrix computation between integer matrices returns an integer matrix. Other computation (e.g., addition between an integer matrix and a float matrix) returns a float matrix.

Figure 5.13 illustrates the specification of our matrix computation API. The type parameters `ROW` and `COL` are the row and column sizes of a matrix, respectively. The boundings for these parameters are written in a similar manner to that of Java on line 10 and line 11 in fig. 5.13. (The keyword `extends` defines the upper bound of a type parameter.) The boundary type `Size` is a type that is defined manually, as follows:

```
abstract class Size { abstract int getIntVal(); }
```

The API users can define any matrix size by subclassing `Size` outside of the PROTOCOOL-generated code. For example, they can use 128-by-256 matrices and 256-by-128 matrices in their computation by defining the two classes `Size128` and `Size256`. The abstract method `getIntVal` is used in the tree evaluator (i.e., visitor; see section 5.2.3 for details) to obtain the integer value represented by a concrete `Size` class.

The users of our matrix computation API proceed as follows:

```
// class Size128 extends Size { ... }
Size128 size128 = new Size128();
// class Size256 extends Size { ... }
Size256 size256 = new Size256();

FltMat<Size128, Size128> matrix1
```

```
 1  class MatrixBuilder {
 2    // Size is upper bound of parameter ROW
 3    ROW extends Size;
 4    COL extends Size;
 5    static IntMat<ROW, COL>
 6      randInt() row(ROW row) col(COL col);
 7    static FltMat<ROW, COL>
 8      randFlt() row(ROW row) col(COL col);
 9  }
10  class IntMat<ROW extends Size, COL extends Size> {
11    NEW_COL extends Size;
12    IntMat<ROW, COL> plus(IntMat<R, COL> m);
13    FltMat<ROW, COL> plus(FltMat<R, COL> m);
14    IntMat<ROW, NEW_COL> mult(IntMat<COL, NEW_COL> m);
15    FltMat<ROW, NEW_COL> mult(FltMat<COL, NEW_COL> m);
16    int[][] toArray() return Evaluator.toIntArray;
17  }
18  class FltMat<ROW extends Size, COL extends Size> {
19    NEW_COL extends Size;
20    FltMat<ROW, COL> plus(IntMat<ROW, COL> m);
21    FltMat<ROW, COL> plus(FltMat<ROW, COL> m);
22    FltMat<ROW, NEW_COL> mult(IntMat<COL, NEW_COL> m);
23    FltMat<ROW, NEW_COL> mult(FltMat<COL, NEW_COL> m);
24    float[][] toArray() return Evaluator.toFloatArray;
25  }
```

Figure 5.13: Specification of our matrix library

```
  = MatrixBuilder.randFlt().row(size128).col(size128);
IntMat<Size128, Size256> matrix2
  = MatrixBuilder.randInt().row(size128).col(size256);
FltMat<Size128, Size256> matrix3
  = matrix1.mult(matrix2);
```

The following statements throw type errors, as they do not conform to the protocol of our API:

```
FltMat<Size128, Size128> matrix1
  = MatrixBuilder.randFlt().row(size128).col(size128);
IntMat<Size128, Size256> matrix2
```

```
  = MatrixBuilder.randInt().row(size128).col(size256);

// Cause type errors (incompatible sizes)
matrix2.mult(matrix1); // [128, 128] * [256, 128]
matrix1.plus(matrix2); // [128, 256] + [128, 128]

// Cause type errors (incompatible element types)
// FltMat * IntMat returns FltMat
IntMat f2x3 = matrix1.mult(matrix2);
```

When using our matrix computation API in Scala 2.13, users can avoid
defining a custom-sized class by using literal singleton types [73], which
allows programmers to use a literal value as a type. To use the litaral types
in our API, the following helper function and class need to be defined by
hand:

```
def size[T <: Singleton](v: T): SizeForScala[T] {
  new SizeForScala[T](t)
}
class SizeForScala[T <: Singleton](v: T) extends Size {
    override def getIntVal: Int = v.asInstanceOf[Int]
}
```

The function `size` creates an instance of `SizeForScala[T]`, which extends
the abstract class `Size` illustrated above. As the parameter `T` is bounded to
`Singleton`, the type of `v` is inferred as a literal singleton type. For example,
the return type of `size(100)` is the literal type 100, which is a subclass of
`scala.Int`. Using the helper function and class illustrated above, users can
write their computation as follows:

```
val matrix1 = MatrixBuilder
  .randFlt().row(size(128)).col(size(128));
val matrix2 = MatrixBuilder
  .randInt().row(size(128)).col(size(256));
val matrix3 = matrix1.mult(matrix2); // No type error
matrix2.mult(matrix1); // Causes a type error
```

Itemized Document API

As PROTOCOOL allows its users to specify how to use type parameters in
their API, PROTOCOOL users can use type parameters to check context-free

grammar. As an example of APIs with context-free grammar, consider an API that emulates itemization of LaTeX, as follows:

```
begin()                    // \begin{itemize}
  .item("Item A")          //    \item Item A
  .begin()                 //    \begin{itemize}
    .item("Item A.1")      //       \item Item A.1
  .end()                   //    \end{itemize}
.end()                     // \end{itemize}
.asTeXStr();
```

The API requires its users to invoke only a pair of `begin()` and `end()` in a chain. A type error occurs when the API users attempt to obtain a string from the itemization with an unbalanced invocation of `begin()` and `end()`:

```
begin()
  .item("Item A")
  .begin()
    .item("Item A.1")
  .end()
.asTeXStr(); // Causes a type error;
             // Cannot resolve method 'toTeXStr'
```

In the API, the item type is inferred from the first argument provided to `item(...)`. A type error occurs when users input an item with an inconsistent type:

```
begin()
  .item(100)
  .item("200") // Causes a type error
.end().asTeXStr();
```

The itemized document API described above can be generated from the specification illustrated in fig. 5.14. The API uses the first type parameter to represent the stack of a pushdown automaton, in a matter that is described in the literature [56]. The second type parameter represents the type of the items in the document.

Although it is possible to check context-free rules in the PROTOCOOL-generated API, it is often excessively tedious to specify such checking in the specification. To achieve such checking, PROTOCOOL users need to encode a pushdown automaton into the specification manually. The encoding of a pushdown automaton has been automated in previous studies.

However, this limitation will not greatly degrade the practical applicability of PROTOCOOL, because context-free rules are not common in fluent

```
class API {
  ITEM;
  static Nested<EndOfDoc, ITEM> begin(ITEM item) ;
}
class Nested<X, ITEM> {
  Nested<Nested<X, ITEM>, ITEM> begin(ITEM item) ;
  X end(ITEM item) return Evaluator.end ;
}
class EndOfDoc {
  String asTeXStr();
}
```

Figure 5.14: Specification for itemized document API

interfaces. The nesting structure is often emulated with a feature that is available in the host language, such as method invocation syntax:

```
// Nesting is emulated by passing sub-chains
// With varargs:
itemize(
  item("Item A"),
  itemize(item("Item B"))
).asTeXStr();
// Without varargs:
itemize()
  .elem(item("Item A"))
  .elem(itemize(item("Item B"))).asTeXStr();
```

This type of sub-chaining technique is frequently used in a real-world library such as j2html[1]. One problem with techniques using variable-length arguments is that all arguments need to be of the same type. The problem does not occur when using techniques without variable-length arguments, but the latter emulation requires somewhat redundant invocations of `elem(...)`. Finding a succinct emulation with the ability to take different argument types can be investigated in future work. In relatively new languages, special syntactic sugars are provided that can be used to emulate nesting structures [36, 14].

---

[1]https://j2html.com

```
 1  class Assertions {
 2    PredicateAssert assertThat(String s);
 3  }
 4  class PredicateAssert {
 5    PredicateAssert startsWith(String s) {
 6      Action.startsWith;
 7    }
 8    PredicateAssert endsWith(String s) {
 9      Action.endsWith;
10    }
11  }
```

Figure 5.15: Specification of subset of AssertJ

Assertion API

As described in section 5.2.3, the semantics of a PROTOCOOL-generated API
are designed to be added by creating a tree evaluator (i.e., visitor). This
style is known as the deep embedding style [26]. In this style, the execution
is postponed until the API user invokes a tree evaluation method, such as
`build()` in `OurAPI`. However, an API often does not provide such a tree
evaluation method. AssertJ[2] is an example of such an API. It does not
require its users to invoke a method such as `runAssertions()` at the end
of the chain: every method call immediately runs an assertion.

```
String str = ... ;
assertThat(str)
  .startsWith("A") // Runs assertion immediately
  .endsWith("Z"); // Runs assertion immediately
```

This style of implementing semantics is known as the shallow embedding
style [26].

PROTOCOOL supports the shallow embedding style. Figure 5.15 illus-
trates the specification of the subset of AssertJ that uses the feature for
the shallow embedding style. The {} block written after a method specifies
an action invoked in that method. PROTOCOOL inserts the action directly
before the `return` statement in the generated method body, as follows:

```
PredicateAssert startsWith(String s) {
```

---
[2]http://joel-costigliola.github.io/assertj/

Table 5.1: Quantitative information of generated code

	Spec.	Gen. (API)	Gen. (TREE)	Gen.
NeuralNetwork	12	158	141	299
Matrix	24	257	238	495
EBNF	21	138	198	336
DOT	54	443	629	1072

	Gen./ Spec.	Gen. (TREE) / Gen.
NeuralNetwork	24.9	0.47
Matrix	20.6	0.48
EBNF	16.0	0.59
DOT	19.9	0.59
Average	19.8	0.54

```
  // Tree construction
  Object_PredicateAssert node = ... ;
  // Inserted action
  Action.startsWith(node);
  return new PredicateAssert(node);
}
```

The constructed tree is provided to the inserted action. Unlike the `return` clause described in section 5.2.3, the action return value is discarded even if the inserted method returns a value.

5.3.2   Reduction of Development Cost

Table 5.1 summarizes the number of lines of several specifications and the generated code. NeuralNetwork and Matrix in the table refer to the specifications we have shown in fig. 5.2 and fig. 5.13. EBNF and DOT refer to the specifications in fig. 5.16 and fig. 5.17, respectively. The latter two specifications are translated into the libraries that allow their users to compose EBNF/DOT programs inside Java code. We counted the number of lines for two groups of classes: API and TREE. The group API contains the classes that the library users interact with and the code for the tree construction. The group TREE contains only the tree node classes. The last two columns of table 5.1 shows the ratio obtained from the values in the first four columns of table 5.1. The column "Gen./ Spec." shows the

```
class API {                        class Term extends Expr {}
  Grammar rules(Rule... rules);    class Fact extends Term {}
  Expr expr(Term... terms);        class Sym extends Fact {
  Term term(Fact... facts);          Fact repeat0();
  TSym tsym(String text);            Fact repeat1();
  NSym nsym(String text);          }
}                                  class TSym extends Sym {}
class Grammar {}                   class NSym extends Sym {
class Rule {}                        Rule eq(Expr expr);
class Expr {}                      }
```

Figure 5.16: Specification for EBNF emulation

magnification of the generated code over its original specification. The column "Gen. (TREE) / Gen." shows the proportion of tree node classes in the generated code.

As shown in table 5.1, a specification is translated into a considerably large amount of Java code. This magnification is due not only to many class definitions for protocol inspection but also to the generation of boilerplate code such as tree node classes and their construction. The large proportion (approximately one half of the generated code) of the tree classes indicates that our tree generation would help library developers a lot in terms of the development cost.

5.4   Summary

In this chapter, we have proposed PROTOCOOL, a tool for generating a generic fluent interface in Java. The contribution of this study is the development of the translation algorithm implemented in PROTOCOOL. Unlike the methods of previous studies, our algorithm analyzes the binding times of type parameters in a method chain. The analysis is the key technique enabling the generation of a generic fluent interface. Support for generic methods is essential for introducing safe fluent interface generation into the real world, as generic methods are frequently used to make an API statically type safe. It is possible to check context-free rules in the PROTOCOOL-generated API, as the use of type parameters is completely the decision of the PROTOCOOL users.

Further investigation into experience with using PROTOCOOL will be our primary future work. In particular, studying the effects on library user

```
class API {
  Graph graph(Stmt... stmts);
  Graph digraph(Stmt... stmts);
  Graph strict() graph(Stmt... stmts);
  Graph strict() digraph(Stmt... stmts);
  Graph graph(String id, Stmt... stmts);
  Graph digraph(String id, Stmt... stmts);
  Graph strict() graph(String id, Stmt... stmts);
  Graph strict() digraph(String id, Stmt... stmts);
  Port port(String id);
  Port port(String id, CompassPt pt);
  Port port(CompassPt pt);
  CompassPt NORTH();
  CompassPt NORTHEAST();
  ... // Other 8 CompassPt directions
  Node node(String id);
  Node node(String id, Port port);
  Attr attr(String k, String v);
  AttrStmt node(Attr... attrs);
  AttrStmt graph(Attr... attrs);
  AttrStmt edge(Attr... attrs);
  Subgraph subgraph(String id, Stmt... stmts);
}

class Stmt {}
class Graph {
  Graph stmts(Stmt... stmts);
}
class Subgraph extends Stmt {
  Subgraph stmts(Stmt... stmts);
}
class Node {
  NodeStmt attrs(Attr... attrs);
  Edge to(Node node);
  Edge to(Subgraph subgsraph);
}
class EdgeStmt extends Stmt {
  Edge attrs(Attr... attrs);
}
class NodeStmt extends Stmt {}
class Attr extends Stmt {}
class AttrStmt extends Stmt {
  AttrStmt attrs(Attr... attrs);
}
class Port {}
class CompassPt {}
```

Figure 5.17: Specification for DOT emulation

experiences caused by strange type names, and the time and space overheads caused by the code bloat, will be useful for discussing the practical applicability of PROTOCOOL and other fluent interface generators.

Chapter 6

Generating Fluent Interfaces with Sub-chaining

This chapter discusses our code-generation technique that supports the generation of sub-chaining APIs. As well as the generics support described in the last chapter, the sub-chaining support is favorable to use of the generative approach in the real world. Our technique is implemented as a tool named Silverchain.

The chapter is organized as follows: We first describe our key idea for sub-chaining support in section 6.1. We then overview our translation method from a give grammer into class definitions (i.e., code-generation technique) in the beginning of section 6.2. In section 6.2.1, we introduce the formal definition and the table representation of an single-state real-time deterministic pushdown automaton (RPA), which we need for describing our technique in more detail from section 6.2.2 to sec:rpa-encoding. Section 6.2.2 describes the central part of our technique, and section 6.2.3 describes the preprocessing part, which rewrites a given grammar into a form that the central part can process. In section 6.2.4, we describe the encoding of RPAs and discuss how to implement the semantics of a generated library. In section 6.2.5, we discuss the limitation of our construction method and describes Silverchain's approach to address that limitation. The last two sections present the use cases for evaluating our technique and summary of this chapter, respectively.

6.1   Key Idea for Sub-chaining Support

Recall that the code generation of fluent interfaces is the translation from a grammar into class definitions. More specifically, a grammar is first turned into a state machine that accepts sequences derived from the grammar, and then the machine is encoded into class definitions for fluent interfaces. Our technique also follows this process from a grammar into class definitions, but it constructs a special state machine for sub-chaining support. In the reset of this section, we describe our idea and the constraint on the state-machine construction caused by applying our idea to the code-generation.

A sub-chaining API allows programmers to devide a chain into semantically grouped pieces. As we described in chapter 1, such an API is useful when composing a long chain or changing a part of chain dynamically; for example as follows:

```
// Sub-chains
WhereClause w1 = col("id").eq(1);
WhereClause w2 = col("name").eq("John");

boolean findById = ... ;
select("name").from("user")
  .where(findById ? w1 : w2) // Switch where-clause
  .execute();
```

Since a sub-chaining API corresponds to a semantic group for human programmers, it is impossible to automatically determine where a sub-chaining API should be provided in a fluent interface. The point where a sub-chaining API is provided is up to the (human) library developers. Therefore, a code-generation algorithm needs to retrieve information about where to create a sub-chaining API from a grammar given by (human) library developers.

Our key idea for the sub-chaining support is to map a non-terminal in a given grammar to a sub-chaining API. By mapping so, the library developers can tell a code-generation algorithm where to generate a sub-chaining API by defining (or not defining) a non-terminal in their grammar. The following indicates users of the generated fluent interface can use the sub-chaining API for where-clause:

```
// `?` indicates zero or one ocurrence
//   of preceding element
<query> -> select from <where-clause>? ;
<where-clause> -> col eq ;
```

If library developers do not want to provide the sub-chaining API, they can specify it by not defining the non-terminal `<where-clause>` in the grammar:

```
<query> -> select from (col eq)? ;
```

Technically, a non-terminal is just a group of syntactical elements and does not always represent a *semantical* group. However, the library developers can write an input grammar so that a non-terminal represents a semantical group.

Mapping a non-terminal to a sub-chaining API imposes a constraint on the construction of a state machine: An algorithm cannot add or remove a non-terminal from a given grammar during the construction of a state machine. In the ordinal setting, it is allowed to rewrite grammar (i.e., add or delete non-terminals) in the translation from a grammar to a state machine. However, since a non-terminal is mapped to a sub-chaining API, the addition of a non-terminal produces a sub-chaining API that is not specified by library developers. The deletion, on the other hand, does not produce a sub-chaining API that is specified by the developers. To allow library developers to control where to create sub-chaining APIs, a code-generation algorithm need to keep the set of non-terminals as specified by library developers. This constraint leads us to the development of a new code-generation algorithm.

6.2 Our Code-generation Technique

To illustrate our translation method concretely, we use a DSL for writing itemized documents whose syntax is defined by the grammar in fig. 6.1a. Parentheses are used to group elements, and a plus sign represents one or more occurrences of the preceding element. Figure 6.1b shows a usage example of this DSL.

Figure 6.2 illustrates the overview of our translation. Our technique first constructs a set of RPAs, each of which corresponds to a non-terminal of a given grammar. It then encodes those RPAs into class definitions to obtain a fluent interface. For clarity, in the following, we denote an RPA for a non-terminal n by R_n.

R_n accepts all the symbol sequences derived from a non-terminal n, including a sequence that contains a non-terminal, not only sequences of terminals. For example, R_{list} (the RPA for `<list>` in fig. 6.1a) accepts the following sequences:

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

```
<list> -> "begin" (<text> <list>?)+ "end";
<text> :: String; // Refers to a group of values
                  //   i.e., type
```

(a) Grammar

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

(b) Example sentence

Figure 6.1: Grammar and example sentence of our DSL
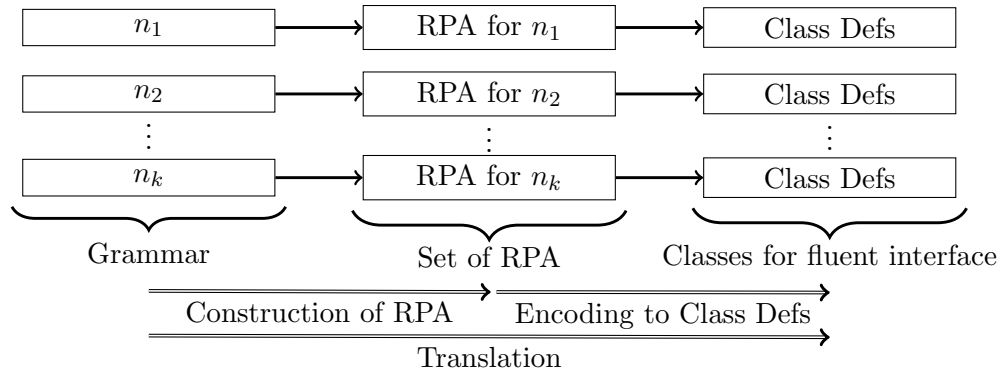
Figure 6.2: Overview of our translation method (n_i is a non-terminal).

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

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

Table 6.1: Table representation of R_{list}

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

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

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

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

6.2.1 RPA and Its Table Representation

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

An RPA can be described by a table since the transition function of an RPA is a binary function. Table 6.1 shows the table representation of
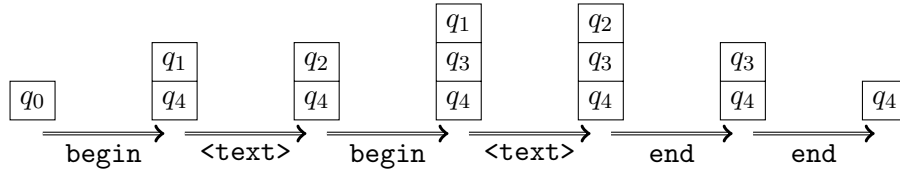
Figure 6.3: Example stack transition

R_{list}. The table has all input symbols in its rows and all stack elements in its columns. A transition $(s, q) \to Q$ is represented by the cell value Q in row s of column q. When pushing elements, the rightmost element of Q is pushed into the stack first. The symbol - in a cell indicates that no transition is defined, and the symbol ϵ indicates that no element is pushed into the stack on that transition. The initial stack element is indicated by adding the non-terminal name on the column top of that element.

R_n, an RPA for a non-terminal n, conforms to the following rules on the number of elements to push into its stack: When R_n consumes the first symbol of a nested construct in a sequence, it pushes two elements; When R_n consumes the last symbol of a nested construct, it pushes no element; R_n pushes one element otherwise. Figure 6.3 shows how the stack content of R_{list} changes at each step when the input sequence is as follows:

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

More specifically, R_{list} pushes two elements when consuming `begin`, pushes no element when consuming `end`, and pushes one element when consuming other symbols such as `<text>`. When consuming the first symbol of a nested construct, R_{list} firstly pushes an element that will appear on the stack top after consuming the last symbol of that nested construct. For instance, when R_{list} consumes the first `begin` in fig. 6.3, it first pushes q_4, which is the element on the top after consuming the last `end`.

6.2.2   Construction of RPAs

The central part of our RPA construction consists of two steps: For each non-terminal n, our method first constructs the RPA that accepts only direct productions of n; It then extends that RPA into R_n. Here, a direct production of n refers to a sequence that matches the regular expression on the right-hand side of n's production rule. Note that the right-hand side can be considered as a regular expression on symbol sequences. For clarity, we denote the RPA accepting only direct productions of n by R'_n.
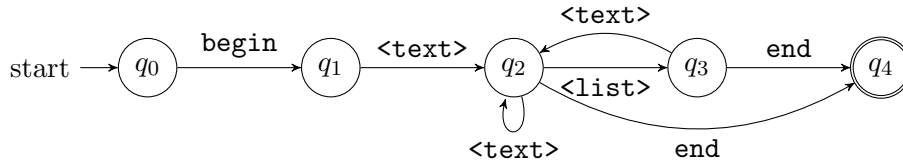
Figure 6.4: State diagram of D_{list}

Table 6.2: Table representation of our RPA construction

(a) Table of R'_n (Initial table)

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

(b) Intermediate table

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

(c) Table of R_n (Final table)

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

The first step constructs R'_n from a deterministic finite automaton (DFA) that accepts n's direct productions. Such a DFA (hereinafter denoted by D_n) can be obtained by using well-known algorithms [6, 78] since the expression on the right-hand side is a regular expression that matches the direct productions. Figure 6.4 shows the state diagram of D_{list}. R'_n is constructed by converting D_n to an equivalent RPA. The following describes the conversion from D_n to R'_n:

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

a stack element of R'_n.

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

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

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

Table 6.2a is the table representation of R'_{list}, which is obtained from D_{list}. Since a DFA can be regarded as an RPA whose stack depth is limited to one, the table of R'_n can always be constructed from D_n.

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

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

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

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

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

In the example case, e_n is q_4 and Step (2.1) replaces all occurrences of q_4 with ϵ. Step (2.2) appends q_4 to the value in row `begin` of column q_0. Table 6.2b shows the intermediate table obtained by applying the above process to table 6.2a.

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

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

When m is `<list>`, C_m is $\{(\texttt{<list>}, q_2) \to q_3\}$ and P_m is $\{(\texttt{begin}, q_0) \to q_1\}$. Step (2.3) adds $(\texttt{begin}, q_2) \to q_1 q_3$. Table 6.2c shows the table obtained by applying the above process to table 6.2b.

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

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

From this grammar, our method first constructs table 6.3a. In table 6.3a, R_{doc} and R_{list} are shown as one table, but the representation is the same as the case where the table describes only one RPA. Our method then defines e_m, C_m, and P_m as follows:

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

where $\emptyset$ denotes an empty set. In Step (2.1), our method first replaces q_2 and q_7 with ϵ. In Step (2.2), our method then appends q_2 and q_7 to the cell values in column q_0 and q_3, respectively. In Step (2.3), our method finally

Table 6.3: Tables appearing in RPAs construction

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

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

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

	doc			list				
	q_0	q_1	q_2	q_3	q_4	q_5	q_6	q_7
beginDoc	$q_1\mathbf{q_2}$	-	-	-	-	-	-	-
beginLst	-	$\mathbf{q_4q_1}$	-	$q_4\mathbf{q_7}$	-	$\mathbf{q_4q_6}$	-	-
<text>	-	-	-	-	q_5	q_5	q_5	-
<list>	-	q_1	-	-	-	q_6	-	-
endLst	-	-	-	-	-	$\boldsymbol{\epsilon}$	$\boldsymbol{\epsilon}$	-
endDoc	-	$\boldsymbol{\epsilon}$	-	-	-	-	-	-
$\$_{doc}$	-	-	ϵ	-	-	-	-	-
$\$_{list}$	-	-	-	-	-	-	-	ϵ

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 $C_{doc} \times P_{doc}$ since $C_{doc} \times P_{doc}$ is $\emptyset$.) table 6.3b is the table representation of R_{doc} and R_{list}, which is obtained by modifying table 6.3a.

6.2.3   Preprocessing

The preprocessing part rewrites a given grammar into a form where every non-terminal has a direct recursion, as we assumed in the previous subsection . To obtain that form, the preprocessing part expands all the occurrences of non-terminals in the right-hand side of a production rule. They

are replaced with the right-hand side of their production rule. For example, consider the following grammar that has an indirect recursion:

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

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

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

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

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

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

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

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

   (ii) Decompose G' into strongly connected components (SCCs) and let the component containing n_d be C [77].
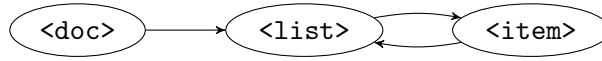
(a) Initial reference relation graph

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

(c) The edge (`<list>`,`<item>`) is selected since the SCC containing `<item>` consists only of one node without a self-loop.
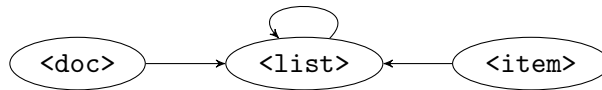
(d) Updated graph after expanding `<item>` in the rule for `<list>`

Figure 6.5: Selecting an edge for inline expansion

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

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

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

In Step (I), the preprocessing part constructs the graph shown in fig. 6.5a from the example grammar. In Step (II), it then selects (`<list>`,`<item>`) through the sub-processes shown in fig. 6.5b and fig. 6.5c. In Step (III), `<list>` will be expanded into the following as we have seen before:

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

By Step (IV), G is updated to the graph in fig. 6.5d. Our preprocessing repeats the above process until no edges found in Step (II). An edge further from the node for the start symbol is examined earlier.

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

Figure 6.6: Without decomposition into SCCs

The 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 fig. 6.5a is selected for expansion. The dotted edge is removed, and the red edge is added as shown in fig. 6.6a. Then the edge (`<doc>`, `<item>`) in fig. 6.6a is selected for expan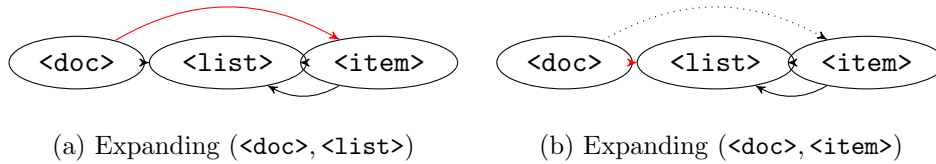sion and the graph is updated to fig. 6.6b, where the dotted edge is removed and the red edge is added. Figure 6.6b is equivalent to fig. 6.5a. Further inline expansions will cause infinite regression. By the decomposition, Silverchain selects an edge composing a mutual-recursion cycle earlier than other edges. Such an edge is expanded, and the cycle is transformed into non-cyclic edges and self-loops. Which edge is first selected among cyclic edges does not matter to resolve infinite regression.

6.2.4   Encoding into Class Definitions

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

The class definition shown in fig. 6.7 shows the classes generated from table 6.1. (For better readability, we put table 6.1 on the bottom of fig. 6.7.) The generated classes are categorized into two kinds. One is a set of classes each of which corresponds to a column of the table. The classes named `Qn` in the figure belong to this category. A class in this category has one type parameter except the class corresponding to the initial stack element. The type parameter of `Qn` is used to represent the stack content below the stack element corresponding to `Qn`. The other category consists of two auxiliary classes: the class corresponding to a non-terminal and the class representing the stack bottom. In fig. 6.7, `List` and `Bottom` belong to this category.

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

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

Figure 6.7: Class definitions generated from table 6.1

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

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

As a manually developed fluent interface often provides several convenience methods using variable-length arguments, Silverchain also generates such methods to improve the user experiences of the generated library. The method on Line 11 is such a convenience method and takes multiple `String` objects as its arguments. The following lines show the example usage of that convenience method:

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

Silverchain generates such a method with variable length arguments when

it finds two actions $(s, q_i) \to q_j$ and $(s, q_j) \to q_j$.

Figure 6.8 shows the generated code for `Q2` including the method bodies. Each method appends the object representing an invoked method to a list shared among state instances. Each method first appends a `Method` instance, which is an instance that holds the name and arguments of the method invocation, to the list. Line 9 in fig. 6.8 is the line that appends such an instance. A method appends multiple `Method` instances at once when it takes a sub-chain as its argument as shown on Line 16. The method then creates and return an instance that is used as the receiver of the next method invocation. The list of invoked methods is shared by passing it as the argument to the constructor as shown on Line 11. We need a little trick for generated methods to return a nested parametric type as shown from Line 26 to Line 31. Instead of executing `new T()`, Silverchain calls `newInstance` on a class object representing `T`. Since a type parameter is not a first-class entity in Java, the generated library uses the reflection API and explicitly passes a type parameter as a class object.

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

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

Library developers (i.e., the Silverchain users) can add semantics to the generated library by implementing an evaluation method, a method that interprets a written chain. Figure 6.9 shows an example implementation that constructs an itemized document in TEX from a written chain. In this implementation, each method in `methodList` is converted into a token of TEX. The Silverchain users can put all semantic actions into one method rather than editing method bodies scattered over the generated code. This approach to implement the semantics reduces the library developers' task when updating and re-generate a modified library. They do not have to edit

```
 1  class Q2<T> {
 2    ArrayList<Method> methodList;
 3    Stack<Class<?>> classStack;
 4    Q2(ArrayList<Method> list, Stack<Class<?>> stack) {
 5      methodList = list;
 6      classStack = stack;
 7    }
 8    Q1<Q3<T>> begin() {
 9      methodList.add(new Method("begin", null));
10      classStack.push(Q3.class);
11      return new Q3<>(methodList, classStack);
12    }
13    Q2<T> text(String text, String... textArray) {
14      methodList.add(new Method("text", text));
15      for (String t: textArray) {
16        methodList.add(new Method("text", t));
17      }
18      return new Q2<>(methodList, classStack);
19    }
20    Q3<T> list(List list) {
21      methodList.addAll(list.methodList);
22      return new Q3<>(methodList, classStack);
23    }
24    T end() {
25      methodList.add(new Method("end", null));
26      try {
27        return (T) classStack.pop()
28          .getDeclaredConstructor(
29            ArrayList.class, Stack.class)
30          .newInstance(methodList, classStack);
31      } catch (Exception e) {}
32    }
33  }
```

Figure 6.8: Definition of `Q2` including method bodies

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

Figure 6.9: Example implementation of library semantics

a number of parts of the generated code.

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

6.2.5   Limitation

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

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

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

of such a syntax component:

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

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

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

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

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

When given grammar represents a visibly pushdown language (VPL) [1], Condition (a) and (b) is satisfied. A VPL has the property that symbols to begin/end a nesting structure are not used anywhere else.

When the table of R'_n violates Condition (a), our method cannot perform Step (2.2) successfully. Table 6.4a is an example table that does not satisfy Condition (a), which is constructed from the following grammar:

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

Table 6.4a has two columns that contain ϵ in row $\$_{nest}$. To fix the number of pushed elements in Step (2.2), our method needs to find e_n, the element to append to the value in row `begin`. When the table contains multiple columns that contain ϵ in row $\$_n$, our method needs to choose either of those elements since a cell can contain only one value. However, the RPA obtained by appending one those elements accepts only a part of sequences that should be accepted. If our method chooses q_2 as e_n and perform Step (2.1) and Step (2.2) as shown in table 6.4b, the RPA does not accept the following:

```
<text>
```

On the other hand, if our method chooses q_3 as e_n and perform Step (2.1) and Step (2.2) as shown in table 6.4c, the RPA does not accept the following:

Table 6.4: Table violating Condition (a) and its modification

(a) Table of R_{nest}

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

(b) $e_n = q_2$

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

(c) $e_n = q_3$

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

```
begin <nest >
```

When the table of R'_n violates Condition (b), our method cannot construct R_n correctly. Table 6.5a is an example of such tables, which is constructed from the following grammar:

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

Our method fixes the number of pushed elements in Step (2.1) and Step (2.2) as shown table 6.5b. It then constructs C_{nest} and P_{nest} as follows:

$$C_{nest} = \{(\texttt{<nest>}, q_1) \rightarrow q_2\}, P_{nest} = \{(\texttt{begin}, q_0) \rightarrow q_1 q_4\}.$$

In Step (2.3), our method put $q_1 q_2$ in row begin of column q_1 as shown in table 6.5c. However, the RPA described by table 6.5c does not accept the following:

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

Table 6.5: Table violating Condition (b) and its modification

(a) Table of R_{nest}							(b) Fixed table of R_{nest}				

(a) Table of R_{nest}

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

(c) Incorrect update

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

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

Silverchain skips the second step in our RPA construction method when it finds the violation of Condition (a) or (b). Since the second step is the process to make RPAs accept indirect productions, RPAs remain to accept only direct productions. This results that the generated fluent interface allows its users to use only sub-chaining APIs for certain parts of a chain. For example, consider encoding table 6.5b into class definitions. Figure 6.10 shows the classes generated from the table and the library users can write the following chains:

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

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

```
 1  class Q0 {
 2      static Q1<Q4<Bottom>> begin() { ... }
 3  }
 4  class Q1<T> {
 5      Q3<T> begin() { ... }
 6      Q2<T> nest(Nest nest)  { ... }
 7  }
 8  class Q2<T> {
 9      T end() { ... }
10  }
11  class Q3<T> {
12      T text(String text) { ... }
13  }
14  class Q4<T> extends Nest {}
15  class Nest {}
16  class Bottom {}
```

Figure 6.10: Class definitions generated from table 6.5b

Table 6.6: Number of classes and methods

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

	#Classes (Hand-written)	#Methods (Hand-written)
LINQ	16	44
DOT	25	337

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

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

6.3 Use Cases

In this section, we compare Silverchain-generated fluent interfaces and popular hand-written libraries, 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 library for LINQ and graphviz-java[4] as one for DOT. Besides, we experimentally investigate how the length of a composed chain affects the compilation time since a Silverchain-generated library heavily uses generics.

Table 6.6 summarizes the numbers of classes and methods generated from the grammars of those DSLs. The values in the first and second columns are the numbers of unique symbols and non-terminals in the grammar, respectively. The values in the third column to the sixth column are the numbers of classes and methods in the generated libraries and the hand-written libraries. When counting classes and methods, we picked up only public ones since our primary concern is API but not non-public implementations. As seen from the table, the numbers of generated definitions are too large to handle by hand although the grammars of those languages are relatively simple. Those numbers are significantly small in hand-written libraries. This smallness is mainly because the hand-written libraries allow their users to compose syntactically incorrect chains as follows:

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

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

Figure 6.11 and fig. 6.12 show example sentences and their embedded versions of LINQ and DOT, respectively. As seen from these examples, the code written with the generated libraries are similar to the one written with hand-written libraries in most parts. Figure 6.11c is similar to fig. 6.11b, and fig. 6.12c and fig. 6.12d are similar to fig. 6.12b.

The first drawback is that code written with the generated libraries tends to contain redundant method calls. For instance, `from().collection(...)` in fig. 6.11c is expressed in a shorter way by `from(...)` in fig. 6.11b. Similarly, `beginGraph()` in fig. 6.12c is omitted in fig. 6.12b. These redundant

---

[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                 from(users)
  where u.age > 2                 .where("age", gt(2))
  orderby u.age descending        .orderBy("age", Order.DESC)
  select u;                       .all();
```

      (a) Example sentence   (b) With coollection

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

(c) With generated library

Figure 6.11: Comparison in LINQ

methods are generated since our translation method encodes every token of a DSL into a method. Those methods can be omitted by using information obtained from the table representation of RPAs. A method can be omitted if it is the only transition that exists between states and no ambiguity arises after omitting that transition. However, if such automatic omission is applied naively, a method chain with the generated libraries 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 since generated libraries use mechanically named classes such as `Q1` and `Q2`. The 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 fig. 6.11b and graphviz-java uses `enum` to specify attributes of a node as shown in fig. 6.12e. However, to add such methods reflecting domain-specific knowledge, the developers of a library need to fix scattered parts of generated classes. The generation of such methods is difficult since our translation method naively encodes every token of a sentence into a method. Finding a smarter way of encoding is important future work to put Silverchain into practical use.

To investigate the relation between the compilation time and the length of a chain, we measured the compilation time of a chain of various length, using the generated library for DOT. In our experiments, we compiled chains

```
digraph G {                    graph("G").directed().with(
  A -> B                         node("A").link(node("B")),
  B -> C                         node("B").link(node("C")),
  C -> A                         node("C").link(node("A"))
}                              );
```

(a) Example sentence   (b) Using graphviz-java

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

(c) Usage with sub-chaining

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

(d) Usage without sub-chaining

```
// With graphviz-java        // With generated library
node("X").with(              node("X").beginAttr()
  Shape.RECTANGLE,             .shape().eq().rectangle()
  Style.FILLED                 .style().eq().filled()
);                           .endAttr();
```

(e) Difference between generated library and graphviz-java

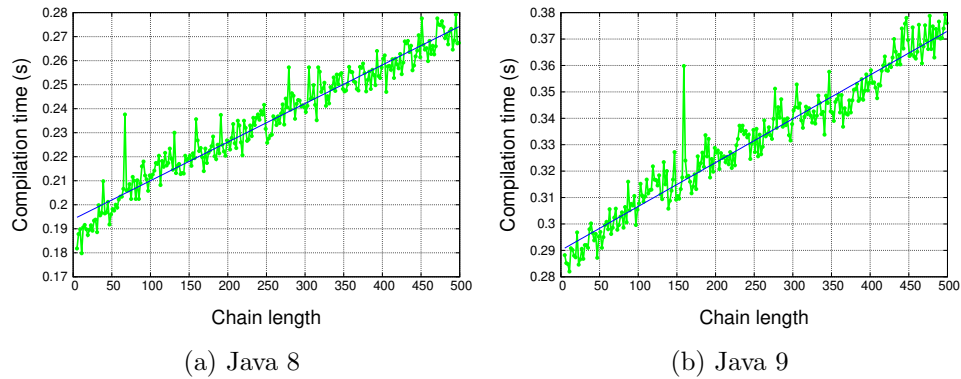Figure 6.12: Comparison in DOT

(a) Java 8   (b) Java 9

Figure 6.13: Result of experiments

Table 6.7: Fitted parameters in $y = ax + b$

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

that contain various numbers of `subgraph().beginSubgraph()` as follows:

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

Since a call to `beginSubgraph` starts a new nested construct, the type size at the end of a chain increases as the chain becomes longer. Here, the size of a type is defined by the number of type names in the textual representation of the type. (The size of `G<T,S>` is three for example.) Our experiments are performed on a machine with Intel Core i7 3.3 GHz processor and 16 GB memory, using `javac 1.8.0_114` and `javac 9`. Two versions of Java were used in our experiments since Java 9 has a new type-checking strategy [39]. We used `javac -verbose` to compile chains and extracted total compilation time from its output.

Figure 6.13 shows the results of this experiment. We measured the compilation time five times for each length of a chain, and the averages are shown in the figures. The blue line in each figure is the linear regression line of data. The fitted parameters are summarized in table 6.7. As seen from

these results, with a library generated by Silverchain, the compilation time grows linearly to the length of a chain even in the worst case.

6.4 Summary

We presented Silverchain, which generates safe fluent interfaces that support both sub-chaining and non-subchaining style. The input to Silverchain is the grammar of a generating fluent interface. Since the DSL for the input grammar is similar to BNF, a fluent interface for an external DSL can be easily generated from the grammar of the DSL with Silverchain. However, for some grammars, Silverchain fails to generate a fluent interface that fully supports non-subchaining style.

First future work is to extend the range of grammars that Silverchain can generate non-subchaining style API for without exponential growth of compilation time. Another future work is to improve the emulation of 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 mapping a part of DSL's syntax to the similar syntax in a host general-purpose language (e.g. if-else syntax in a DSL to the same syntax in a host language). 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 host language mechanism.

Chapter 7

Conclusions

This dissertation presented our studies to improve the user experiences of fluent interfaces. As we have discussed in chapter 2, fluent interfaces are considered as a promising design style of library interfaces. Therefore, the scope of our studies is broad to a certain degree. However, the increasing use of fluent interfaces in the real world is not empirically verified as far as we know. Chapter 3 addresses this lack of empirical evidence by mining 2,814 Java repositories on GitHub. Our results shown in chapter 3 empirically reveal that the method-chaining style and fluent interfaces are increasingly used in the real world.

In chapter 4, we discovered desirable language designs for the use of fluent interfaces, through mining real-world repositories hosted on GitHub. We also statistically estimated the impact of introducing those designs. As we summarized in section 4.8, we found four language/library designs and their introduction would help method chaining in 14.8% of method invocations. The primary contribution of the study discussed in the chapter is that the impact is measured in a quantitative manner. Since language development often proceeds conservatively, such quantitative data is important to claim the need for the designs in language development.

Chapter 5 proposed a code-generation technique to generate generic fluent interfaces. Our technique is demonstrated by the code generator PROTOCOOL. Unlike the previous study, the users of PROTOCOOL can include generic methods in an input grammar. Our translation algorithm implemented for PROTOCOOL is modeled as the construction of deterministic finite automaton (DFA) with type parameter information. Each state of the DFA holds information about which type parameters are already bound in that state. The information is used to identify whether a method invocation

109

in a chain newly binds a type to a type parameter, or refers to a previously bound type. The identification is required since a type parameter in a chain is bound at a particular method invocation, and that bound type is referred to in the following method invocations. Our algorithm constructs the DFA by analyzing the binding time of type parameters and their propagation among the states in a DFA that is naively constructed from the given grammar.

Chapter 6 presented another code-generation technique to realize the sub-chaining support in the fluent interface generation. We implemented a tool named Silverchain to demonstrate our technique. Our translation is modeled as the construction of deterministic pushdown automata without ϵ-transitions called single-state real-time deterministic pushdown automata (RPAs). The class definitions of a fluent interface are generated by encoding those RPAs. Our RPA-construction method is different from the literature [34, 64] in that it does not add or remove non-terminals from the given grammar. This property is essential to generate sub-chaining APIs as specified by Silverchain users (i.e., library developers). Our technique can generate a fluent interface from any context-free grammar, but the interface generated from some grammars require sub-chaining APIs to compose certain parts of sentences. This limitation is due to our RPA-construction method.

Future Work

The results presented in chapter 3 are supportive evidence for the acceptance of the method-chaining style and fluent interfaces in the real world. However, true acceptance cannot be derived directly from them. To claim so, we need to conduct interviews with real-world programmers whether they accept the style and interfaces. Such interviews are helpful to strengthen the background of our studies in chapter 4, chapter 5, and chapter 6. It is also beneficial for a better understanding of method chaining to investigate why real-world programmers prefer (or do not prefer) method chaining. The issues mentioned in the StackOverflow thread [60] would be good starting points of such an investigation. Although we conducted analyses only of Java code for the study in chapter 3, conducting the same study on other object-oriented languages would also be valuable to strengthen the background of studying fluent interfaces.

In chapter 4, we presented several language/library designs but did not implement them into Java. The desirable language designs for the patterns NULLEXCEPTIONAVOIDANCE, REPEATEDRECEIVER, and DOWNCAST are

implemented in other languages such as Kotlin., Therefore, there would not be serious problems in the actual implementation of those designs although the implementation would cost a lot. The pattern CONDITIONALEXECU-TION can be relieved by introducing syntactic sugar into Java. Thus, the implementation would not include a serious problem either. However, these discussions are based on our (optimistic) observation. It is beneficial to implement them into Java, to discover new problems that PL/SE researchers need to tackle.

While a lot of studies have been done for fluent interface generation (including our studies), the implementation side is not well studied yet. By generating interface code, the programmer experiences would be improved. However, the code bloat would decrease the runtime performance and would increase the size of a program. Related to this point, how the semantics of libraries are implemented is interesting enough to study. When creating a relatively small library, the deep embedding style is too tedious. On the other hand, the shallow embedding makes the implementation complicated when developing relatively large libraries.

The user studies of our demonstration tools PROTOCOOL and Silverchain are another important future work to test the availability of the generative approach in the real world. To the best of our knowledge, there is no fluent interface generator that is widely used in the real world, i.e., the generative approach has never been truly evaluated in real-world settings. (Note that many use cases are shown in the papers discussing the approach, including our papers.) The generative approach would not be completely useless since there are widely-used code generators for specific libraries such as AssertJ and jOOQ. The user studies would help the promotion of the approach and the discovery of new problems that we need to solve.

Bibliography

[1] Rajeev Alur and P. Madhusudan. "Visibly Pushdown Languages". In: *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing*. 2004.

[2] Pavel Avgustinov et al. "QL: Object-oriented Queries on Relational Data". In: *30th European Conference on Object-Oriented Programming*. 2016.

[3] Kent Beck. "Smalltalk Best Practice Patterns". In: 1997, pp. 183–188.

[4] Nels E. Beckman, Duri Kim, and Jonathan Aldrich. "An Empirical Study of Object Protocols in the Wild". In: *Proceedings of the 25th European Conference on Object-oriented Programming*. 2011.

[5] 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*. 2014.

[6] Janusz A Brzozowski. "Canonical regular expressions and minimal state graphs for definite events". In: *Mathematical theory of Automata* (1962).

[7] Yegor Bugayenko. *Fluent Interfaces Are Bad for Maintainability*. https://www.yegor256.com/2018/03/13/fluent-interfaces.html. (Accessed on 01/07/2021).

[8] R. P. L. Buse and W. R. Weimer. "Learning a Metric for Code Readability". In: *IEEE Transactions on Software Engineering* (2010).

[9] Arvid Butting et al. "Deriving Fluent Internal Domain-specific Languages from Grammars". In: *Proceedings of the 11th ACM SIGPLAN International Conference on Software Language Engineering*. 2018.

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

[11]   S. Cook, R. Harrison, and P. Wernick. "A simulation model of self-organising evolvability in software systems". In: *IEEE International Workshop on Software Evolvability*. 2005.

[12]   Bruno Courcelle. "On jump-deterministic pushdown automata". In: *Mathematical systems theory* (1977).

[13]   Valentin Dallmeier et al. "Generating Test Cases for Specification Mining". In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. 2010.

[14]   *Domain-Specific Languages*. http://docs.groovy-lang.org/docs/latest/html/documentation/core-domain-specific-languages.html. (Accessed on 01/07/2021).

[15]   Robert Dyer et al. "Mining Billions of AST Nodes to Study Actual and Potential Usage of Java Language Features". In: *Proceedings of the 36th International Conference on Software Engineering*. 2014.

[16]   *Emulating "self types" using Java Generics to simplify fluent API implementation*. https://gist.github.com/esfand/dd79511ede7e48a0e88e. (Accessed on 01/07/2021).

[17]   Sebastian Erdweg et al. "SugarJ: Library-based Syntactic Language Extensibility". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 2011.

[18]   *Extensions - Kotlin Programming Language*. https://kotlinlang.org/docs/reference/extensions.html. (Accessed on 01/07/2021).

[19]   *Extensions - The Swift Programming Language (Swift 5.1)*. https://docs.swift.org/swift-book/LanguageGuide/Extensions.html. (Accessed on 01/07/2021).

[20]   Martin Fowler. *FluentInterface*. https://www.martinfowler.com/bliki/FluentInterface.html. (Accessed on 01/07/2021).

[21]   Martin Fowler. *Generation Gap*. https://martinfowler.com/dslCatalog/generationGap.html. (Accessed on 01/07/2021).

[22] Steve Freeman and Nat Pryce. "Evolving an Embedded Domain-specific Language in Java". In: *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. 2006.

[23] *Functions - D Programming Language*. https://dlang.org/spec/function.html. (Accessed on 01/07/2021).

[24] Mark Gabel and Zhendong Su. "Symbolic Mining of Temporal Specifications". In: *Proceedings of the 30th International Conference on Software Engineering*. 2008.

[25] Carlo Ghezzi, Andrea Mocci, and Mattia Monga. "Synthesizing Intensional Behavior Models by Graph Transformation". In: *Proceedings of the 31st International Conference on Software Engineering*. 2009.

[26] Jeremy Gibbons and Nicolas Wu. "Folding domain-specific languages: Deep and shallow embeddings (Functional Pearl)". In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*. 2014.

[27] Yossi Gil and Keren Lenz. "Simple and safe SQL queries with C++ templates". In: *Science of Computer Programming* (2010).

[28] Yossi Gil and Tomer Levy. "Formal Language Recognition with the Java Type Checker". In: *30th European Conference on Object-Oriented Programming*. 2016.

[29] Yossi Gil and Ori Roth. "Fling — A Fluent API Generator". In: *Proceedings of 30th European Conference on Object-Oriented Programming*. 2019.

[30] AA Gorshenev and Yu M Pis'mak. "Punctuated Equilibrium in Software Evolution". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* (2005).

[31] Sheila A. Greibach. "A New Normal-Form Theorem for Context-Free Phrase Structure Grammars". In: *Journal of the ACM* (1965).

[32] Radu Grigore. "Java Generics Are Turing Complete". In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017.

[33] Y. Guo et al. "An Empirical Validation of the Benefits of Adhering to the Law of Demeter". In: *18th Working Conference on Reverse Engineering*. 2011.

[34] Michael A. Harrison and Ivan M. Havel. "Real-Time Strict Deterministic Languages". In: *SIAM Journal on Computing* (1972).

[35] L. Hatton. "Power-Law Distributions of Component Size in General Software Systems". In: *IEEE Transactions on Software Engineering* (2009).

[36] *Higher-Order Functions and Lambdas - Kotlin Programming Language.* https://kotlinlang.org/docs/reference/lambdas.html. (Accessed on 01/07/2021).

[37] Paul Hudak. "Building Domain-Specific Embedded Languages". In: *ACM Computing Surveys* (1996).

[38] Kazuhiro Ichikawa and Shigeru Chiba. "User-Defined Operators Including Name Binding for New Language Constructs". In: *The Art, Science, and Engineering of Programming* (2017).

[39] *JEP 215: Tiered Attribution for javac.* http://openjdk.java.net/jeps/215. (Accessed on 01/07/2021).

[40] Tadao Kasami. *An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages.* Tech. rep. DTIC Document, 1965.

[41] Martin Kellogg et al. "Verifying Object Construction". In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering.* 2020.

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

[43] K. Lieberherr, I. Holland, and A. Riel. "Object-oriented Programming: An Objective Sense of Style". In: *ACM SIGPLAN Notices* (1988).

[44] Z. Lin and J. Whitehead. "Why Power Laws? An Explanation from Fine-Grained Code Changes". In: *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories.* 2015.

[45] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. "Power Laws in Software". In: *ACM Transactions on Software Engineering and Methodology* (2008).

[46] Robert C Martin. *Clean code: a handbook of agile software craftsmanship.* Pearson Education, 2009.

[47] Luis Mastrangelo, Matthias Hauswirth, and Nathaniel Nystrom. "Casting about in the Dark: An Empirical Study of Cast Operations in Java Programs". In: *Proceedings of the ACM on Programming Languages* (2019).

[48] Davood Mazinanian et al. "Understanding the Use of Lambda Expressions in Java". In: *Proceedings of the ACM on Programming Languages* (2017).

[49] *Method Cascades in Dart.* http://news.dartlang.org/2012/02/method-cascades-in-dart-posted-by-gilad.html. (Accessed on 01/07/2021).

[50] Leo A. Meyerovich and Ariel S. Rabkin. "Empirical Analysis of Programming Language Adoption". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications.* 2013.

[51] Chris Myers. "Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs". In: *Physical review. E, Statistical, nonlinear, and soft matter physics* (2003).

[52] Tomoki Nakamaru and Shigeru Chiba. "Generating a Generic Fluent API in Java". In: *The Art, Science, and Engineering of Programming* (2020).

[53] Tomoki Nakamaru et al. "An Empirical Study of Method Chaining in Java". In: *Proceedings of the 17th International Conference on Mining Software Repositories.* 2020.

[54] Tomoki Nakamaru et al. *Data - An Empirical Study of Method Chaining in Java.* Version 1.0.0. Zenodo, Mar. 2020. DOI: 10.5281/zenodo.3697939. URL: %5Curl%7Bhttps://doi.org/10.5281/zenodo.3697939%7D.

[55] Tomoki Nakamaru et al. "Generating fluent embedded domain-specific languages with subchaining". In: *Journal of Computer Languages* (2019).

[56] Tomoki Nakamaru et al. "Silverchain: a fluent API generator". In: *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences.* 2017.

[57] Mark EJ Newman. "Power laws, Pareto distributions and Zipf's law". In: *Contemporary physics* (2005).

[58] Ligia Nistor et al. "Wyvern: A Simple, Typed, and Pure Object-oriented Language". In: *Proceedings of the 5th Workshop on MechAnisms for SPEcialization, Generalization and inHerItance.* 2013.

[59] *Null Safety - Kotlin Programming Language.* https://kotlinlang.org/docs/reference/null-safety.html. (Accessed on 01/07/2021).

[60]   *OOP - Method chaining - why is it a good practice, or not? - Stack Overflow.* https://stackoverflow.com/questions/1103985/method-chaining-why-is-it-a-good-practice-or-not. (Accessed on 01/07/2021).

[61]   *Optional Chaining - The Swift Programming Language (Swift 5.1).* https://docs.swift.org/swift-book/LanguageGuide/OptionalChaining.html. (Accessed on 01/07/2021).

[62]   Chris Parnin, Christian Bird, and Emerson Murphy-Hill. "Java Generics Adoption: How New Features are Introduced, Championed, or Ignored". In: *Proceedings of the International Working Conference on Mining Software Repositories*. 2011.

[63]   Dominik Picheta. "Nim in Action". In: 2017, pp. 3–21.

[64]   Jan Pittl and Amiram Yehudai. "Constructing a realtime deterministic pushdown automaton from a grammar". In: *Theoretical Computer Science* (1983).

[65]   Marco Pivetta. *Fluent Interfaces are Evil.* https://ocramius.github.io/blog/fluent-interfaces-are-evil/. (Accessed on 01/07/2021).

[66]   Daryl Posnett, Abram Hindle, and Premkumar Devanbu. "A Simpler Model of Software Readability". In: *Proceedings of the 8th Working Conference on Mining Software Repositories*. 2011.

[67]   Michael Pradel and Thomas R. Gross. "Automatic Generation of Object Usage Specifications from Large Method Traces". In: *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*. 2009.

[68]   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*. 2012.

[69]   D. J. Rosenkrantz and R. E. Stearns. "Properties of Deterministic Top Down Grammars". In: *Proceedings of the First Annual ACM Symposium on Theory of Computing*. 1969.

[70]   Peter H Salus. *A quarter century of UNIX*. ACM Press/Addison-Wesley Publishing Co., 1994.

[71]   Simone Scalabrino et al. "A comprehensive model for code readability". In: *Journal of Software: Evolution and Process* (2018).

[72]   *Scope Functions - Kotlin Programming Language.* https://kotlinlang.org/docs/reference/scope-functions.html. (Accessed on 01/07/2021).

[73] *SIP-23 - LITERAL-BASED SINGLETON TYPES.* https://docs.scala-lang.org/sips/42.type.html. (Accessed on 01/07/2021).

[74] Robert Strom and Shaula Yemini. "Typestate: A programming language concept for enhancing software reliability". In: *IEEE Transactions on Software Engineering* (1986).

[75] Joshua Sunshine et al. "First-class State Change in Plaid". In: *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. 2011.

[76] Hiroto Tanaka, Shinsuke Matsumoto, and Shinji Kusumoto. "A Study on the Current Status of Functional Idioms in Java". In: *IEICE Transactions on Information and Systems* (2019).

[77] Robert Tarjan. "Depth-first search and linear graph algorithms". In: *SIAM journal on computing* (1972).

[78] Ken Thompson. "Programming Techniques: Regular Expression Search Algorithm". In: *Communications of the ACM* (1968).

[79] I. Turnu et al. "A modified Yule process to model the evolution of some object-oriented system properties". In: *Information Sciences* (2011).

[80] *TypeScript 3.7 · TypeScript.* https://www.typescriptlang.org/docs/handbook/release-notes/typescript-3-7.html. (Accessed on 01/07/2021).

[81] Todd L. Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. 2003.

[82] J. Wu, R. C. Holt, and A. E. Hassan. "Empirical Evidence for SOC Dynamics in Software Evolution". In: *2007 IEEE International Conference on Software Maintenance*. 2007.

[83] Hao Xu. "EriLex: An Embedded Domain Specific Language Generator". In: *Objects, Models, Components, Patterns*. 2010.

[84] Tetsuro Yamazaki et al. "Generating a fluent API with syntax checking from an LR grammar". In: *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*. 2019.

[85] Daniel Younger. "Recognition and parsing of context-free languages in time n^3". In: *Information and Control* (1967).