The University of Tokyo
Graduate School of Information Science and Technology
Department of Creative Informatics

博士論文

A Study of Protocol-Checking and Memory-Management Techniques for Assisting Library Development

(ライブラリ開発を支援するためのプロトコル検査およびメモリ管理技術の研究)

Doctoral Dissertation of:
Tetsuro Yamazaki
山崎 徹郎

Academic Advisor:
Shigeru Chiba
千葉 滋

Abstract

Information technology has contributed to our lives a lot, and the limits are not visible yet. There still be an uncountably large number of applications that can improve our lives. Furthermore, the number of applications will increase because technological advances will discover new problem domains to apply information technology. Not only developing specific applications is important when we are trying to develop a large number of applications, but also it is important to reduce the costs to develop each application. Code reuse is one basic idea to reduce the cost of application developments. By sharing common program pieces in applications, we can avoid re-implementing duplicate parts. A set of programs organized for reuse is called a library. Today, we can access various kinds of libraries. By using an appropriate library, we can implement a complex application with a little effort. However, there is no library suitable for all applications. Applications in different problem domains need different libraries since not many program pieces are shared between them. Developing a specific library can assist only in limited application developments. To assist in a wider range of application developments, we aim to reduce the cost of library developments. Basically, developing a library is not a simple task. The main difficulty in library development comes from the abstract nature of libraries. Unfortunately, this complexity is inevitable. We aim to reduce the library development cost by assisting developers in extra tasks accompanied by library developments. There already exists many research and tools which assist in library development. In this thesis, we focus on two approaches to assist in library developments and propose we propose three additional techniques. The first approach is about library protocol checking. Libraries often provide many usages to adopt various kinds of applications. However, from a viewpoint of a library developer, the larger number of usages a library provides, the more difficult it is to control all the behavior. To prevent an overlooked unexpected behavior from introducing bugs, library developers can check the usages and prevent illegal usages from execution. A set of rules about how a library can be used is called a library protocol. We propose an embedding technique that makes type checkers emulate LR parsers. This technique can be used to check protocols of libraries that provide fluent interfaces. We experimented on how long our method took to check protocols and revealed that the time complexity is quadratic, although it is expected to be linear. The second approach is about Foreign Function Interfaces (FFIs). An FFI can reduce the library development cost since it enables engineers to use foreign libraries

without translating their programs. However, an FFI can cause memory leaks when both programming languages support garbage collection. When a cyclic reference goes through both languages, the cycle will not be collected since neither collector can determine whether the cycle is garbage or not. Note that our interest is in reusing foreign libraries. Thus we do not want to modify the language that the foreign library is written in. We propose a backup garbage collection algorithm that can collect cross-language cyclic references without customizing both of the collectors. Our algorithm copies reachability between objects from a language to the other language and reproduces it as references between objects. Once the copy finishes, the collector in the other language can detect all garbage objects correctly, and the cross-language garbage cycle will be broken. We experimented on how long our algorithm takes to collect cyclic garbage. We also propose a garbage collection algorithm for self-reflective garbage collectors. Experimenting with a garbage collection algorithm is time-consuming. Existing collectors are complex to customize, and it often saves our efforts to develop a whole new programming language. Customizing a garbage collector could be easier if the customization was self-reflective. However, it is not easy to design a self-reflective feature. We design a simple self-reflective interface to customize a copying garbage collector (named `copy-time callback`) to discuss what problem will occur in self-reflective collector customization. `Copy-time callback` allows registering a callback function which is called each time the collector copies an object. We discovered a problem that the callback function creates objects during garbage collection, and it can consume a huge amount of memory. Our garbage collection algorithm places objects created during collection in a special region and invokes minor garbage collection to compress them. And our algorithm will move the objects managed in the special region into the normal region in which other objects are managed when the collection finishes. We experiment on how small a memory region our algorithm can execute our microbenchmark and confirmed that our algorithm could save the memory.

Acknowledgements

I would like to take this opportunity to express my gratitude for people who helped to complete my doctoral course. First of all, I feel great gratitude to my supervisor Professor Shigeru Chiba. Although I was a capricious student, he led me to complete my doctoral course without preventing me from having my own way. I would not finish this thesis without his accurate guidance.

I would also like to express my gratitude to my colleagues in Computing Software Group. Especially, Kazuhiro Ichikawa gave me beneficial and patient advice on my studies even if the topic was not related to his research theme. Tomoki Nakamaru is a person who gave me a chance to begin my work on fluent interfaces. He gave me great inspiration, and I could not finish my work without that.

Finally, I would like to thank my parents for supporting me for a long time. Their long and patient support enables me to reach this place without getting into serious trouble.

Contents

Chapter 1

Introduction

In recent years, information technology spreads to our society, and computers have become important tools for our lives. At the same time, we have not discovered the limits of information technology; we do not know how much we can help ourselves by using computers. In other words, there are so many applications we should develop in the future. Not only developing each application is important but also reducing development costs is important.

One common approach to reducing the cost of developing applications is code reuse. By sharing common program pieces in applications, we can avoid reimplementing duplicated programs. Although we must pay an additional cost to reuse a code piece, code reuse can reduce the total cost.

For example, a series of program pieces organized for code reuse is called a library. Today, we can access many libraries to implement our applications. Each library provides each feature, and we can avoid writing a complex program by using an appropriate library. However, using a library is not free. It is time-consuming to find an appropriate library and installing a library may fail. You also have to learn the usage before using that library, and if a bug in a library disturbed your application, you had to look for a workaround. Of course, we should use a library only if it can reduce the total cost.

We aim to assist in developing libraries to indirectly reduce the cost of developing various applications. Developing a library can reduce the costs of developing applications. However, a specific library can reduce only the costs of a limited range of application developments. To cover a wide range of applications, we have to develop various libraries as problem domains. Moreover, the number of problem domains must increase in the future. Thus, not only developing each library is important, but also reducing the cost of developing libraries is important.

Here, how can we assist in library developments? We could reduce the library development cost if we discovered a general method to generate a library automatically. However, we suppose that this approach is not realistic. A library program is very different when the problem domain is different since requirements for a library vary depending on the

problem domain. Thus, the way to generate a library must change in a different problem domain. We suppose a library generator can reduce only the costs of a limited range of library developments since it can generate only a limited range of libraries.

Instead, we aim to reduce the costs to prevent problems that occur when you use a library. There are costs that engineers have to pay to use a library. Moreover, the costs could be huge if the library was not prepared for reuse. It is considered library developers' responsibility to prepare their library so that engineers do not need to pay a huge amount of effort. The preparation required for library developers is often the same regardless of the problem domain. We can save library developers' efforts by discovering an easier way to prepare libraries for reuse.

There are many existing pieces of research and tools which assist library developments. For example, to use a library, engineers must understand both the feature it provides and its usage. Organizing such information into documentation is one important task in library developments. Javadoc is a well-known tool that assists Java developers in documenting their library. Figure 2.1 shows an example document generated using Javadoc. Even with Javadoc, the amount of texts required to generate documents is not small. Moreover, library developers have to study how they can use Javadoc. However, Javadoc is used widely; since documentation takes a lot of time and effort, Javadoc can reduce the total cost.

1.1 Approach

We researched two approaches to reduce the cost of developing libraries. One approach is protocol checking which detects illegal uses of libraries so that library developers can easily prevent exceptional use cases. The other is foreign function interfaces which enable us to reuse libraries written in another programming language. Since the variety of programming languages is increasing, research on problems with a foreign function interface between recent languages is especially important today. This section shows the details and our motivation for these two approaches.

The first approach we researched is protocol checking. Libraries usually aim to provide a flexible interface to fit many applications. The more usages a library provides, the more application can use it. However, from a library developer's viewpoint, the large number of usages a library provides makes it difficult to control that library's behavior. If the library behaves unexpectedly to an engineer, it can cause bugs. Engineers will consider that it is the library developer's responsibility to prevent such unexpected behaviors.

By checking library protocols, library developers can detect illegal uses and prevent such applications from executing. A library protocol is a set of rules about what usage is legal for a library. Protocol checking examines a program whether it contains protocol violations or not. Thus, a library developer can avoid checking all the usages of his/her library to behave expectedly by applying a protocol checking. One typical format of protocol checking is type

checking. With type checking, library developers specify what kind of value their library receives and returns as types. Library users must also select types for their arguments passing to that library. By checking the consistency between the parameter types and the argument types, type checkers can detect illegal applications.

For example, in Java, even in a simple HTTP server, there appear ten or more classes and dozens of methods. Java standard HTTP libraries (i.e. `java.net.http`) provide more classes as other options for engineers. Engineers can use them in any combinations they like. A combination of library calls may work as an HTTP server, another combination may work as an HTTP client, and there are many other combinations work as other applications. Since Java is a statically typed programming language, no engineer can pass illegal arguments to the library. Type checking in Java reduces the variety of library calls. Thus library developers in Java need less effort to prevent unexpected behaviors.

A method is known that checks a protocol of fluent interface by making a type checker emulate parsing. A fluent interface is a library interface that is called via method chains. Since a method chain is a sequence of method calls, we can specify which method chain is legal in the form of grammar. This protocol checking method makes a type checker emulate a parsing corresponding to the given grammar. Library developers can prevent executing applications containing an illegal method chain by detecting as a type error.

However, no method can make a type checker emulate an LR parsing in non-exponential time, as far as we know. How complex protocol these methods can check changes by which parsing algorithm they make a type checker emulate. LR parsing is a standard parsing algorithm that can parse all deterministic context-free grammars. For example, Java grammar is designed to be analyzable by an LR parsing. We can specify any deterministic context-free grammar as a protocol for our fluent interface to make a type checker emulate an LR parsing. However, known methods that embed an LR parsing to a type checker need exponential time for a method chain length to verify it as far as we know. Originally, an LR parsing takes an only linear time for the input length. We can get one step closer to practical use if we can discover a faster method.

The second approach we researched is about foreign function interfaces (FFI). A mechanism is called an FFI when it enables us to call functions written in another programming language. We can use a library via an FFI, even if it is written in another language. Thus, we can reduce the cost to reimplement a library written in another language by using an FFI, as long as that library is not used very frequently. An FFI is especially important to young programming languages since they do not have enough libraries.

Cooperation between garbage collectors is important when an FFI connects two programming languages, and both languages support garbage collection. Garbage collection automatically detects unused memory regions and collects them for other data. When an FFI connects two programming languages, and both languages support garbage collection, it is difficult to detect a garbage object which is a part of cyclic reference going through both languages. Without communications between collectors, neither collector can determine whether a cyclic reference is garbage or not. From either collector's viewpoint, only

a part of the cycle is visible and cannot access the rest. Neither collector can determine whether an invisible part is garbage or not. Thus neither can determine a visible part as garbage because it is reachable from the invisible part.

It is desirable to avoid customizing both collectors connected by an FFI to cooperate. Usually, a source code of a garbage collector is large-scale and complex. Since customizing a garbage collector is difficult, it is desirable to avoid customizing collectors. It is especially desirable to avoid customizing the collector of the language that libraries are written in because our target is reusing libraries from a new language. A new language collector must be more simple to customize than a collector of an elder language. Although a similar problem is discussed in distributed garbage collection, no method can make garbage collectors cooperating without customizing both collectors, as far as we know.

We also researched a language feature that enables programmers to customize a garbage collector with a little effort. We could make garbage collectors cooperative by using existing methods to customize garbage collectors with a little effort. Thus, designing a language feature for collector customization can be another approach to prevent problems of FFI in the future.

One reason that makes garbage collector customization complex is that a programmer must consider two programming languages simultaniously to customize a collector. Most language processors are implemented in a lower-level language than the language processed by that language processor itself. Garbage collectors are also implemented in the lower-level language since they are a part of language processors. To customize a garbage collector, a programmer must consider not only the language the collector is written in but also the language managed by that collector to keep the semantics without changing.

By providing a self-reflective language feature, we can make garbage collector customization easier. A computation is self-reflective when it operates the language processor, which executes that computation itself. Via self-reflective computation, programs can modify the behavior of the language processor. By customizing a garbage collector in a self-reflective way, programmers no longer need to consider the programming language implementing that collector.

However, designing a self-reflective feature is not simple. First, no self-reflective feature allows customizing any part of the language processor, unfortunately. If a mechanism allows programmers to customize any part of the language processor, that mechanism also allows programmers to embed any kind of bugs. Such an interface must be more complicated than customizing the language processor directly in the lower-level language. We have to carefully consider what customization to accept before designing a self-reflective feature.

Infinite regression can be another designing issue of a self-reflective feature. Customizing a language processor can also change how the self-reflective feature works. A self-reflective feature has to behave as if the behaviors before and after the customization are consistent to avoid inconsistency. Naive implementation often causes *infinite regression*, an endless loop to find a fixpoint the behaviors become consistent. We have to design a self-reflective feature to avoid *infinite regressions* carefully.

1.2 Contributions

As shown in section 1.1, we researched two approaches. The former approach was about protocol checking, and the latter approach was about FFIs. Later in this thesis, we will present a proposed method for the former approach and two proposed methods for the latter approach. Our contributions for the first approach are summarized as follows:

- We present an algorithm to translate LR automata into the Fluent language, which we designed to express single-state non-realtime deterministic push-down automata. These automata are not jump-stack automata [9] , but they can pop multiple elements at once.
- We also present an implementation scheme of the automata described in Fluent. The implementing languages are Scala, Haskell, or C++. Our scheme uses function/method overloading that considers type arguments. It is not available in Java.
- We developed a code-skeleton generator of fluent API based on our approach. It generates code skeletons in Scala, Haskell, or C++. According to our experiments, the compilation time of the client code that accesses the generated fluent API was linear or quadratic.

Also, our contributions for the second approach are summarized as follows:

- We present a garbage collection algorithm to collect cyclic garbage, which goes through two programming languages connected via an FFI. This algorithm does not require customizing both garbage collectors of both languages. This algorithm is probabilistic; it possibly fails to collect garbage objects.
- We implemented our algorithm by customizing YARV, a virtual machine for Ruby, and developing an FFI library between Ruby and JavaScript. According to our experiments, the number of garbage objects in each cyclic garbage has a substantial impact on how long our algorithm takes to collect them. Our experiments also show that not many objects escape from garbage collection when there are no more than a hundred remote references.
- We present a simple self-reflective interface to customize a garbage collector to discover a problem in a self-reflective customization of a garbage collector. We also present a problem that a callback function can create a huge number of objects and turn them into garbage during a garbage collection.
- We present a garbage collection algorithm to manage objects created during garbage collection within a small amount of additional memory consumption. Our algorithm reserves a specific *buffer* region to allocate objects during collection. Our algorithm moves survival objects in *buffer* region into the same region to other objects to prevent inconsistent behavior.

- We developed an interpreter for a Scheme-like language implementing the self-reflective garbage collector customization interface and our garbage collection algorithm. 2KB is sufficient for the *buffer* region to reduce memory consumption for running our microbenchmark compared to a naive copying garbage collection according to our experiment.

1.3 Structure of This Thesis

From the next chapter, we present our research on reducing the costs of developing libraries. The rest of this thesis is organized as follows:

Chapter 2: Background

In this chapter, we show the background of our research. In this chapter, we first show existing techniques that we can use to reduce the cost to develop libraries to clarify what we can do to reduce the library development costs. Then, we show our motivation again in detail with examples.

Chapter 3: Check Complex Protocols of Fluent APIs

In this chapter, we present a technique to check a fluent interface protocol. We can generate a library-skeleton with this technique providing a fluent interface from a protocol expressed as an LR grammar. Since a fluent interface accepts a chain of method calls as its input, a grammar can describe a boundary of what method chaining is legal as a call to a fluent interface. Our technique encodes a protocol into types that each method call receives and returns. Thus a type checker will verify the client code to access the generated library by an LR parsing corresponding to the input grammar. Type checkers will report protocol violations as type errors.

In this chapter, we present our algorithm in two steps. Since our algorithm can generate a library-skeleton written in not only a specific programming language but also programming languages providing a particular kind of features, we introduce a pseudo-language named *Fluent* to explain the shared part in translations separately. Thus, we present a translation from an LR automaton to a *Fluent* program as the first translation, and then we present a translation from a *Fluent* program to a Scala program as the second translation in section 3.3.4. We also present translations from a *Fluent* program to a Haskell program and a C++ program in section 3.4.

Chapter 4: Collecting Cross-Language Cyclic Garbage References

In this chapter, we present a technique to manage cross-language garbage cycles that can occur when reusing a library written in another programming language via an FFI.

We first present a naive implementation of an FFI library and then present an example uncollectible cyclic reference created using the given naive FFI library. We then present a garbage collection algorithm that can collect such cross-language garbage cycles. Since this algorithm can cause long pauses, it could be better to use it as a backup garbage collection, a separate collection from a regular collection focusing on collecting specific garbage objects.

Our algorithm presented in section 4.3 is probabilistic, and it possibly fails to collect a part of garbage objects. We designed our algorithm to ensure that no object is mistakenly collected so that no memory error occurs and all garbage objects have at least a chance of being collected. We discuss how this probabilistic behavior affects our algorithm in section 4.3.2.

Chapter 5: Self-Reflective Garbage Collection for Customizing Garbage Collectors

In this chapter, we present a technique to implement a language feature to customize a garbage collector in a self-reflective way. Customizing a garbage collector requires a programmer to control two programming languages simultaneously, the language that collector is written in and the language that collector manages. Thus, the self-reflective approach is promising to make garbage collector customization easier. Making garbage collector customization easy can be another approach to prevent problems that an FFI between recent languages can cause since one reason why it is difficult to collect cross-language cyclic references is that it is difficult to customize a garbage collector. Since self-reflective garbage collector customization has not been researched well, we start by designing a simple self-reflective interface to discover what problem the interface can cause.

Chapter 6: Conclusion

This chapter concludes this thesis. In this chapter, we also discuss future works.

Chapter 2

Background

A series of programs organized for reusing them is called a library. Each library reduces software development costs for each specific problem domain. We aim to assist library development in reducing software development costs.

There are various approaches to assist in developing libraries. Originally, libraries can be any program since they are just programs focusing on reuse. However, reusing a program is not free. Engineers have to understand what feature the program piece to reuse provides. Engineers also incorporate the program piece into their application carefully to prevent them from behaving unexpectedly. It is considered a duty of library developers to reduce reuse costs. Thus, there is not a small amount of extra effort formed in library developments. We do not aim to reduce the complexity of library developments, but we aim to reduce the extra work's cost.

The rest of this chapter is composed as follows. In section 2.1, we clarify how we can assist in library developments by showing existing methods. In section 2.2, we focus on some approaches and look into the details to discuss possible improvements.

2.1 Assistance Methods for Library Development

This section shows several approaches to assist in library developments. Since there are so many existing methods, it is not easy to comprehensively cover them. In this section, we will present several existing methods in the order of what we think is important.

2.1.1 Methods to Reduce the Cost of Using Libraries

Originally, using a library was not a simple task. To reuse a program, engineers must adjust their program to avoid inconsistency between their application and libraries. For example, if a program shares its memory region with a library program and uses it inconsistently, unexpected behavior can occur. Not only memories but also other computational resources can cause unexpected behavior when used in a conflicted way. Engineers have to avoid

such resource conflict to prevent their application from unexpected behavior. To enable engineers to avoid resource conflicts, library developers have to design their libraries in advance to provide a way to prevent them.

Module system is one of the most essential assistance methods for developing a library. A module system enables engineers to keep their source program separated into multiple program pieces. A language processor with a module system receives such separated program pieces, composes them, and executes them. A piece of the separated source program is called a module. Usually, a module system automatically assigns computational resources to each module to prevent resource conflicts. Thus, with a module system, library developers can avoid preparing to enable engineers to prevent resource conflicts by implementing their library as a module.

For example, C language allocates local variables into stack frames and access them based on base pointers so that local variables used in different modules are allocated into different memory regions. C language also adjust global variables' addresses at link time to allocate them to independent memory regions. C language also provides both `malloc` and `free` to enable programmers to allocate their dynamic variables independently. Thus, library developers in C language can avoid worrying about memory conflicts. In other words, C language's memory management strategy reduces the development cost of libraries.

For another example, recent programming languages often provide programmers with features that enable them to control namespaces. The names of functions or variables can also conflict, and that is one of the common causes of unexpected behavior. Some programming languages provide a mechanism called namespace to ease the name conflict problem. By considering names belonging to each namespace, functions and variables with the same name do not cause the name conflict as long as each name belongs to a different namespace. Library developers can reduce the name conflict risk by making each library to use each namespace. Namespaces release library developers from devising unique function names and variable names to prevent the name conflict problem.

For example, packages in Java, namespaces or classes in C++, module in Ruby, and object statements in Scala are language features that also have an aspect of the namespace. The following Scala program defines two `aFunction`s into different namespaces.

```
object namespace1 {
  def aFunction() = /* do something */
}
object namespace2 {
  def aFunction() = /* do another thing */
}
```

Programmers can specify which `aFunction` they refer to by writing its namespace explicitly as `namespace1.aFunction` or `namespace2.aFunction`. Programmers can also `import` a namespace to omit writing the namespace. In these languages, programmers specify namespaces with absolute paths. Thus, there remains some risk that namespace paths can conflict. In Python, there are namespaces, but no module can declare their namespace

by themselves. Python processors automatically assign independent namespaces to each module when a program imports that module. Although module paths can conflict, there are no namespace conflicts in Python as long as a program succeeds in importing modules.

2.1.2 Methods to Check How Libraries are Used

A library is a collection of reusable program pieces. Libraries often provide various kinds of usages to support a broader range of applications. Such a library is useful since engineers can use the library repeatedly without relearning how to use it.

From the viewpoint of a library developer, the larger number of usages a library provides, the more challenging it to control its behavior. Library developers cannot control how engineers use their libraries. Thus, if developers overlooked illegal usage that causes unexpected behavior, engineers could unfortunately embed bugs into their application. It is still better if the bug interrupts the execution, but it can be more annoying if it continues with an incorrect value.

Protocol checking is an approach that prevents such unexpected behavior by detecting illegal uses of a library and prevent programs from incorrect execution. A typical protocol checking for libraries is type checking. With type checking, libraries specify conditions that each input should follow as types, and a type checker checks application programs whether they observe the conditions or not. For example, object-oriented programming languages usually regard interfaces as types. In object-oriented languages, an interface is a set of valid manipulation an object accepts. A function call will be safe as long as it observes input objects' interfaces. From a library developer viewpoint, type checking is viewed as ensuring that nothing will input an illegal value. Thus, type checking releases library developers from handling edge cases, reducing the library development cost.

For example, assume an addition operator `+` which requires that both operands are numbers. If a program is interpreted to add a character string to a number, the execution can cause unexpected behavior. Type checking can detect such absurd programs.

2.1.3 Methods to Reduce the Cost of Preparing Documentation

Engineers have to know what feature a library provides and how they can use it before using that library. Libraries are required to summarize such information as documentation. Writing documents is one essential and time-consuming task in library development.

Javadoc is a tool that assists library developers in writing documents. Figure 2.1 shows an example input to Javadoc and figure 2.1 shows the output. Javadoc generates documentation from a smaller amount of description, although it fixes the documents' format. The fixed format is also desirable for library users since they can read it without worrying about what format it is written in. Recent programming language often provides documentation tools like Javadoc. For example, pydoc for Python, Dokka for Kotlin, RustDoc for Rust, and other tools are public.

Listing 2.1: An example input for Javadoc

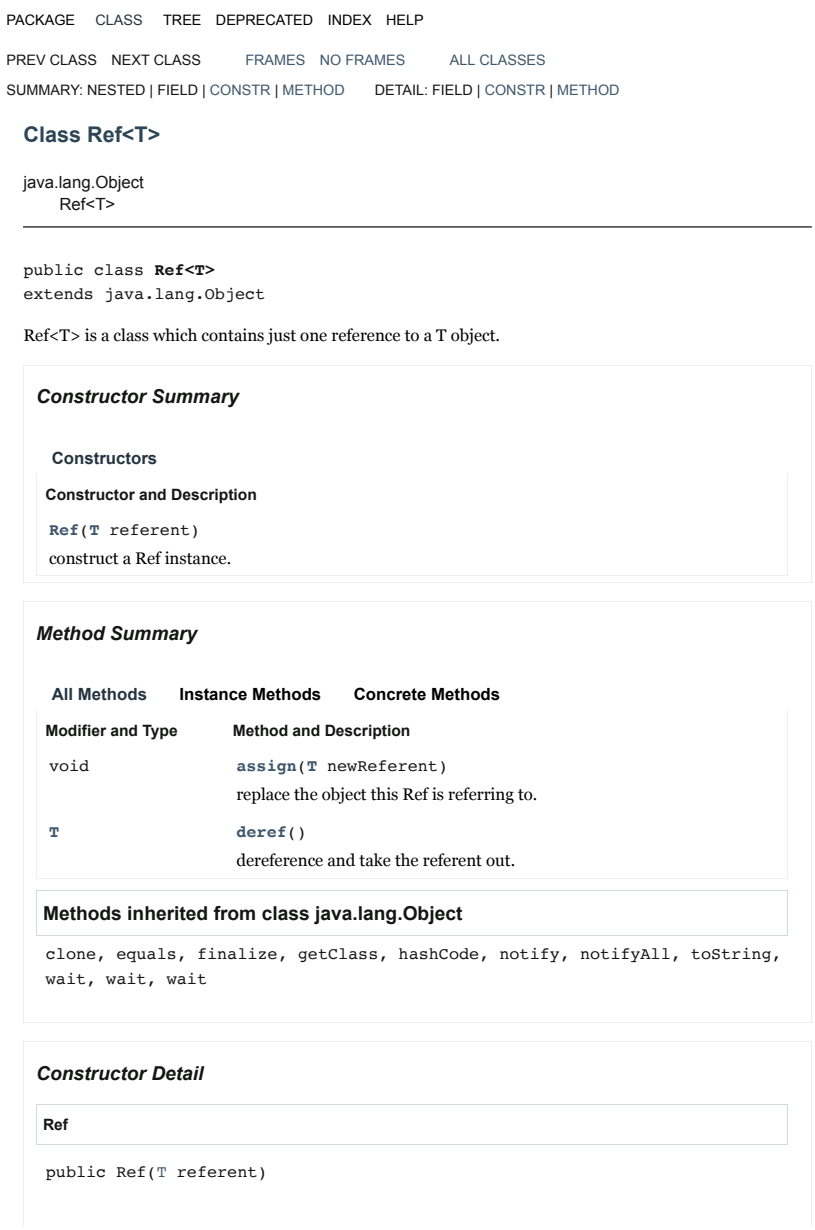
```
1 /**
2  *  {@literal Ref<T>} is a class which contains just
3  *  one reference to a T object.
4  *  @version 0.1
5  *  @author Tetsuro Yamazaki
6  */
7 public class Ref<T> {
8   private T referent;
9   /**
10   *  construct a Ref instance.
11   *  @param referent the initial object
12   *       this Ref instance refers to
13   */
14  public Ref(T referent) {
15    this.referent = referent;
16  }
17  /**
18   *  dereference and take the referent out.
19   *  @return the referent
20   */
21  public T deref() {
22    return referent;
23  }
24  /**
25   *  replace the object this Ref is referring to.
26   *  @param newReferent replace the reference to this object
27   */
28  public void assign(T newReferent) {
29    referent = newReferent;
30  }
31 }
```

2.1.4 Methods to Reuse Libraries in Other Languages

Libraries are written in a specific programming language, and they are usually used in the same language. In other words, usually, an application cannot use a library when it is written in a different programming language. To use such a foreign library, programmers have to translate the library program.

A mechanism is called a Foreign Function Interface (FFI) when it enables programmers to call functions written in another programming language without translating it. Programmers can use foreign libraries via an FFI since most libraries are used via function calls. By using an FFI, library developers can easily import foreign libraries.

Programming languages often provide an FFI between C language. There are two



PACKAGE CLASS TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

Class Ref<T>

java.lang.Object
Ref<T>

```
public class Ref<T>
extends java.lang.Object
```

Ref<T> is a class which contains just one reference to a T object.

Constructor Summary

Constructors

Constructor and Description
`Ref(T referent)` construct a Ref instance.

Method Summary

All Methods **Instance Methods** **Concrete Methods**

Modifier and Type	Method and Description
`void`	`assign(T newReferent)` replace the object this Ref is referring to.
`T`	`deref()` dereference and take the referent out.

Methods inherited from class java.lang.Object

`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructor Detail

Ref

```
public Ref(T referent)
```

Figure 2.1: an example documentation generated by Javadoc

```
using PyCall
plt = pyimport("matplotlib.pyplot")
x = range(0;stop=2*pi,length=100)
y = sin.(x)
plt.plot(x, y)
plt.show()
```

Figure 2.2: An example program using PyCall

```
// A matrix library with function call foramt
Matrix m = mat_mul(mat_add(matrix1, matrix2), matrix3);
// A matrix library with method call format
Matrix m = matrix1.add(matrix2).mult(matrix3);
// A matrix library with mathematical operation format
Matrix m = (matrix1 + matrix2) * matrix3;
```

Figure 2.3: Example interfaces of a matrix library

possible reasons for this. First, it is easy to support since most language processors are implemented in C language. Second, programs worth reuse are concentrated in C language.

In recent years, FFIs between script languages (i.e. Python, JavaScript, and so on) also attract attention. For example, PyCall is an FFI library in Julia which calls Python functions. Programmers can call Python functions from Julia via PyCall. Figure 2.2 shows an example program using PyCall.

2.1.5 Methods to Extend Library API

All libraries have their host language in which they are written. Thus, a developer cannot implement a library interface that is not interpretable in the host language. For example, matrix libraries can provide better readability and maintainability if engineers can use them in mathematical expressions. Figure 2.3 shows example interfaces of matrix libraries. However, it is challenging to implement a matrix library that provides such a mathematical interface without a host language feature that enables modifying mathematical operators' behavior.

Developing a preprocessor is a way to develop a library with an interface that is not interpretable in the host language. A preprocessor works as follows: it receives a source program before the host language processor receives it, converts it, and then returns it to the original processor. Thus, developers can design any format for a preprocessor's input as long as it can convert it to the original language.

However, developing a preprocessor introduces some problems, and it seems not popular to develop a preprocessor when a developer wants to extend a library API. The first problem

```
1 (defclass Range ()
2   ((start :accessor start :initarg :start)
3    (stop  :accessor stop  :initarg :stop)
4    (diff  :accessor diff  :initarg :diff)))
5 (defmethod next ((this Range))
6   (if (>= (start this) (stop this)) nil
7     (let ((result (start this)))
8       (setf (start this) (+ (start this) (diff this)))
9       result)))
```

Figure 2.4: An class declaration using CLOS.

is the development cost of preprocessors. Each preprocessor extending a language has its own parser as a component since its inputs are in its own language. However, a parser is one of the most complex components of a language processor to implement. It is difficult just implementing a parser that can analyze source programs correctly, but it is useless unless it can report appropriate error messages for illegal inputs. The second problem is the maintenance cost of preprocessors. When the language developers update the original language, preprocessors must follow the updates since they should not touch other programs from library calls. Especially when the language introduces new syntax, preprocessors must update their parsers, although it is a very complex component. The third problem is the lack of composability. When two libraries provide different preprocessors, no engineer can use them together because neither input is the same as the other output. This limitation is extreme.

We consider macro systems a better method for language extension than developing a preprocessor. A macro system allows programmers to transform the source program before executing it by registering functions that manipulate program pieces as data. With a macro system, library developers cannot design their library interfaces freely, unlike preprocessors. However they can design an interface that interprets each syntactic element of the host language differently. For example, Common Lisp Object System (CLOS) is an object system implemented just as a library. Figure 2.4 shows an example program that declares a class `Range` and a method `next` using CLOS. Since Common Lisp does not support object-oriented programming, there is no syntax for class declarations, method declarations, and member access. However, the library interface of CLOS looks as if Common Lisp is supporting object-oriented programming. The CLOS interface interprets some specific function calls as class declarations, member accesses, and so on.

One typical macro system is syntactic macro in lisp-family programming languages. Today, some programming languages provide similar macro systems (i.e. macro in Elixir, Scala macros in Scala, Template Haskell in Haskell). Although macrosystems seem to improve all the three problems in preprocessors, it is still said complex, and programmers must pay great care when they define a macro. Especially, error handling is a problem;

errors caused by macros are difficult to identify the cause because the error messages are usually complicated.

2.2 Our Motivation

As seen in section 2.1, various methods to assist in developing a library already exist. However, we believe additional assistance still reduces the development cost for libraries. We will develop a huge number of libraries since there will be countless applications for our lives in the future. Thus, discovering an assistance method can significantly contribute even if the improvement was still small.

For example, it is known that we can check a protocol of an interface that is used in method chaining style by associating receiver types of each method call with internal states of an automaton. How complex protocol this method can check depends on the class of automata that receiver types are associated with its internal states. As seen in section 2.1.2, protocol checking is one aspect in which we can assist library developments. Fluent API is an attractive library interface design that enables engineers to write their application program in similar word order to the natural language by providing a library interface used via method chains. For example, an integer sequence decreasing from 10 to 0 is written as following in object-oriented style:

```
IntegerSequence seq = new IntegerSequence(10, 0, -1);
```

With a fluent API, we can write the same sequence as follows:

```
IntegerSequence seq = IntegerSequence.from(10).to(0).by(-1);
```

We can make it closer to natural language as follows:

```
IntegerSequence seq = IntegerSequence.decreasing().from(10).to(0);
```

A simple way to implement a fluent interface is by defining a class that accumulates all method calls. Figure 2.5 shows an example implementation of the `IntegerSequence` class. Since each method returns the object itself, programmers can chain method calls. However, this straightforward implementation lacks the ability to detect illegal uses. For example, the `IntegerSequence` accepts broken sequences like `IntegerSequence.from(0).to(10).to(20).to(30)`.

It is known that we can detect such illegal chains by making each method return an object that has only methods legal as the next method call. For example, figure 2.6 shows an example implementation of such protocol checking. We can consider this protocol checking as emulating an automaton by using the type checker. From this viewpoint, each method call corresponds to each input token, and each receiver type corresponds to each state of an automaton. Since the number of classes is finite, the automaton emulated in this protocol checking is a deterministic finite-state automaton. Thus, the complexity of protocols that this method can check is limited to regular language.

```
1 class IntegerSequence {
2   private int from;
3   private int to;
4   private int by;
5   public static IntegerSequence from(int from) {
6     IntegerSequence self = new IntegerSequence();
7     self.from = from;
8     return self;
9   }
10  public IntegerSequence to(int to) {
11    this.to = to;
12    return this;
13  }
14  public IntegerSequence by(int by) {
15    this.by = by;
16    return this;
17  }
18 }
```

Figure 2.5: An example implementation of a fluent API

We can check more complex protocols by associating receiver types with internal states of a more sophisticated automaton. For example, it is known we can associate receiver types with the internal states of a deterministic pushdown automaton by representing a stack by a nested type argument [57, 39, 45]. Deterministic pushdown automata can analyze LL(1) languages. Thus, these methods can check fluent-style library APIs whenever you can write down valid method chains by an LL(1) grammar.

LR is a language class that is a proper superset of LL(1) languages. We can check more complex protocols by associating receiver types with internal states of an LR automaton [36, 9]. However, these methods consume exponential time for method chain length, although analyzing an LR language can be finished in linear time. Since this protocol checking method just emulates an automaton by using the type checker, analyzing an LR language must be finished in linear time. There should be a faster method that can check a complex protocol of fluent library interface.

As seen in section 2.1.4, FFIs can reduce the library development cost. It is known that an FFI between managed languages can cause memory leaks without cooperation between garbage collectors. Recent programming language often supports garbage collection which automatically detects and reuses memory regions that are no longer used. Such programming languages are said to be managed.

In contrast to traditional FFIs between C language, we have to consider cooperation between garbage collectors when an FFI connects two managed languages. For example, figure 2.7 shows a program that registers a `onclick` callback function to a button ele-

```
class IntegerInterval {
  private int from;
  private int to;
  private int by;
  public static AfterFrom from(int from) {
    IntegerInterval acc = new IntegerInterval();
    acc.from = from;
    return new AfterFrom(acc);
  }
}
class AfterFrom {
  private IntegerInterval acc;
  public AfterFrom(IntegerInterval acc) {
    this.acc = acc;
  }
  public AfterTo to(int to) {
    acc.to = to;
    return new AfterTo(acc);
  }
}
class AfterTo {
  private IntegerInterval acc;
  public AfterTo(IntegerInterval acc) {
    this.acc = acc;
  }
  public IntegerInterval by(int by) {
    acc.by = by;
    return acc;
  }
}
```

Figure 2.6: A safer implementation of a fluent API

```
class Button < React::Component
  def render
    @dom = React.createElement('button')
    @dom.onclick = proc do
      @status = 'clicked'
    end
    @dom
  end
end
```

Figure 2.7: An example Ruby program using JavaScript features via an FFI

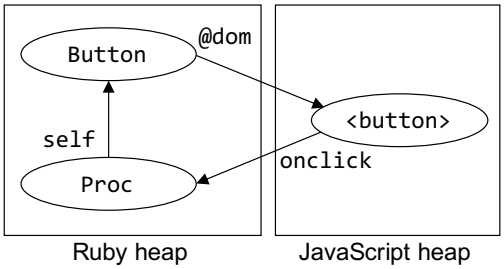


Figure 2.8: How objects are allocated by executing figure 2.7

ment using an FFI between Ruby and JavaScript. In line 1, the FFI library evaluates `'document'` as a JavaScript program, and it returns the `document` in JavaScript. Line 2 calls `getElementById` method in `document` to obtain the button object. And line 3-5 registers a callback function. Note that the substance of `button` is allocated in JavaScript memory, and the variable `button` references it remotely. Also, note that a `Proc` is a function object in Ruby, thus line 3-5 registers a callback remotely. Figure 2.8 shows how objects will be allocated. This simple program creates a cyclic reference between `Proc` and `<button>`.

Both Ruby and JavaScript are managed languages. Without cooperation, no collector can collect garbage if it is a part of a cross-language cyclic reference. For example, the Ruby collector will delay collecting `Proc` since it is referred from `<button>` and Ruby collector cannot determine whether `<button>` is garbage or not. The same thing happens to the JavaScript collector. The JavaScript collector will also delay collecting `<button>` since it is referred from `Proc`. This kind of memory leak seems to be a result of a compromise. Therefore we call them "the last piece of cake."

A similar problem is reported between the JavaScript engine V8 and the renderer Blink in Google Chrome [10]. Thus, we can collect such cyclic garbage in a similar way to their method. However, their approach requires both collectors to provide an additional interface to cooperate. In the case that we apply it to an FFI, it is desirable to be able to use the collector of the language in which libraries are written without customizing it. There can be an improved garbage collection algorithm that can collect cross-language cyclic garbage, reusing existing garbage collectors as it is.

When experimenting on a garbage collection algorithm, there is difficulty in customizing garbage collectors. It would be easier to experiment on a collection algorithm if we could easily customize an existing garbage collector in a common programming language. We can test the collector by using existing test tools, and we can also use existing benchmark applications to measure the performance.

However, a garbage collector is usually a large-scale complex program. Programmers must consider both behaviors of the language the collector is written in and the language the collector manages. This fact makes the customization of a collector more difficult. If

we could customize a collector in the language managed by the collector itself, customizing the collector would become more manageable.

A mechanism is said self-reflective when it modifies the behavior of a language processor from a program processed by the processor itself. One typical self-reflective feature is the syntactic macro system in lisp-family languages. A syntactic macro modifies a program by a function defined in the program itself. Thus, a self-reflective garbage collector customization mechanism can make researches on garbage collection algorithms easier. However, it is challenging to design a self-reflective language feature. Since self-reflective features modify the language processor itself, inconsistency can frequently occur when it allows free customization. Usually, self-reflective features avoid inconsistency by restricting the range of customization. Thus, we need trial and error on restrictions to discover a good self-reflective feature design to customize a collector.

Chapter 3

Check Complex Protocols of Fluent APIs

3.1 Introduction

A Fluent API [17] is a promising design pattern for embedded domain-specific languages (embedded DSLs, or EDSLs) [29]. An embedded DSL is a DSL embedded in a general-purpose language, called the host language. It is usually implemented by a library for the host language and hence it can be considered as a library with a language-like programming interface. A naive technique for implementing such embedded DSLs is the string-embedding style, where the whole DSL code is embedded as a string literal in the host language. It is passed to the library function, parsed, and interpreted. Another technique adopted more often is to construct a language-like interface, or the *syntax* of the DSL, by using programming primitives such as method calls in the host language. A series of method or function calls to the library in the host language is regarded as the DSL code.

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

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

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

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

It is not easy work to implement an embedded DSL so that it will check the given method chain, which is a sequence of lexical tokens in the DSL, is valid or not. They only report an

error at runtime although standalone (or external) DSLs statically check the validity and report a syntax error when they find an invalid sequence of lexical tokens. The lack of static checking is a drawback of embedded DSLs since DSLs can compel somewhat programmers through the DSL syntax to follow semantically correct use of their functionalities. An invalid chain of method calls to fluent API should be considered as semantically wrong usage of the library. Note that detecting wrong usage of libraries is a significant topic and thus a number of static analysis tools have been developed, for example, in the security domain [38].

It is known that the validity of a method-call chain can be checked by using host-language types. In the example of the SQL library above, if the return type of the `from` method is a class providing only the `where` method, `from` is followed only by `where`. No other methods can follow `from`. We can declare a number of classes as the return types and thereby control valid chaining of method calls. To mitigate the costs of declaring a number of classes as return types, code-skeleton generators for fluent API, such as EriLex [57], have been already proposed.

This chapter presents a code-skeleton generator for fluent API that supports LR grammars in a widely-used programming language. As far as we know, existing generators support only grammars in classes smaller than LR. Although an algorithm has been proposed to support LR grammars within the ability of Java's type system [21], it requires exponential time for compiling a method chain accessing a fluent API library. A difficulty of supporting LR grammars is that LR parsing uses a push-down automaton; it does not use a simple finite-state automaton, which can be used only for parsing regular grammars. Since Java's type system is known as being Turing-complete [25], a push-down automaton can be encoded in principle in Java-like languages by using a type system if time and space overheads are ignored. Our challenge was to discover the kind of push-down automata that can be encoded in a widely-used language, which has a type system with limited capability. Using an advanced type system such as Plaid's [51] is an easy solution but we do not take this approach because we aim at developing a programming tool for existing popular languages.

Our contributions are summarized as follows:

- We present an algorithm to translate LR automata into the Fluent language, which we designed to express single-state non-realtime deterministic push-down automata. These automata are not jump-stack automata [9] but they can pop multiple elements at once.

- We also present an implementation scheme of the automata described in Fluent. The implementing languages are Scala, Haskell, or C++. Our scheme uses function/method overloading that considers type arguments. It is not available in Java.

- We developed a code-skeleton generator of fluent API based on our approach. It generates code skeletons in Scala, Haskell, or C++. The client code to access the

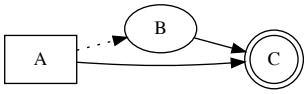


Figure 3.1: The graph drawn by the DOT program in Listing 3.1

generated fluent API was compiled in linear or quadratic time according to our experiments.

Since single-state deterministic push-down automata cannot perform LR parsing [23], our automata have the ability to pop multiple elements at once. To give the automata non-realtimeness, in other words, to allow ε-transitions, our implementation scheme needs function/method overloading (or type classes in Haskell). The ε-transitions are encoded by the methods that recursively call themselves. These methods are overloaded on their receiver types and return types with a type argument.

In the rest of this chapter, Section 3.2 mentions the background of this work. Section 3.3 proposes our technique to encode LR parsing in our Fluent language. It also presents our implementation of Fluent in Scala. Section 3.4 presents the implementations in Haskell and C++. Section 3.5 shows our experiments and section 3.6 compares our work to related work. Section 3.7 concludes this chapter.

3.2 Syntax Checking of Fluent Style Code

Since each problem domain has its own natural notation, the use of a domain specific language (DSL) is becoming widely accepted. For example, the DOT language [18] is a DSL designed for drawing a graph. The graph in Figure 3.1 is drawn by the program written in the DOT language shown in Listing 3.1. DOT provides simple and natural notation for describing a graph. However, since DOT is not general-purpose, we often want to write only part of the program in DOT and the rest of the program in a general-purpose language. In this approach, data exchanges between the two parts of the program tend to be awkward.

For better integration, we should use the DSL embedded in the general-purpose language. If the embedded DSL version of DOT is a library with the fluent API in the general-purpose language, it is straightforward to exchange data between the DSL and the general-purpose language, which is now the *host* language, since they are the same language. The DSL part of the program is constructed by chains of method calls to the library. The appearance of the DSL program is not far different from the original. Suppose the general-purpose language is Java. Listing 3.2 is the program written in the embedded

Listing 3.1: A DOT program

```
1 digraph small_graph {
2   A [shape = rectangle];
3   B;
4   C [shape = doublecircle];
5   A -> B [style = dotted];
6   {A B} -> C;
7 }
```

Listing 3.2: A fluent-style Java program

```
1 Graph fluentApiExample = beginDOT()
2   .digraph("small_graph")
3     .node("A").shape("rectangle")
4     .node("B")
5     .node("C").shape("doublecircle")
6     .edge("A").to("B").style("dotted")
7     .edge("A").and("B").to("C")
8   .endDOT();
```

DSL, which is equivalent to Listing 3.1. In Listing 3.2, method names and arguments can be regarded as lexical tokens in the DSL. We still see one-to-one correspondence between Listing 3.1 and 3.2 although Listing 3.2 is more verbose.

As the original standalone-DSL version of DOT applies static checking, for example, syntax checking to the source program, the embedded-DSL version of DOT can apply syntax checking at the compilation time by the host-language processor. Here, syntax checking is to check whether a chain of method calls is a valid sequence or not. Suppose that we have made a mistake when writing line 6 in Listing 3.2. Instead of this:

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

we have wrongly written line 6 as follows:

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

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

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

In the DOT language, the decoration of a node is specified by *shape* but that of an edge is by *style*. The library could also throw a runtime exception unless we change our *upright*

Listing 3.3: The return type of the `edge` method

```
1 public class AfterEdge {
2   private Graph  acc;
3   private String srcNode;
4   public AfterEdge(Graph a, String s){acc=a; srcNode=s;}
5   public AfterTo to(String dstNode) {
6     return new AfterTo(acc, srcNode, dstNode);
7   }
8   public AfterAnd and(String andNode) {
9     List<String> srcNodes;
10    srcNodes.add(srcNode);
11    srcNodes.add(andNode);
12    return new AfterAnd(acc, srcNodes);
13  }
14 }
```

notation to make *shape* and *style* interchangeable, but again, throwing a runtime exception is not convenient.

These invalid method chains can be detected by using the host-language type system. An advanced type system such as session types [27] obviously can deal with this detection but much simpler type systems such as Java's can also do so to a certain degree. For example, a fluent API generator based on this idea has been proposed for Java [45]. It generates a skeleton of library methods from a grammar definition written in the Backus-Naur form (BNF). The library methods report a type error when a chain of method calls is not valid. It regards each method call as a lexical token and reports an error when a sequence of the lexical tokens does not satisfy the given grammar. For example,

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

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

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

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

The definition of the return type of `edge` would be as shown in Listing 3.3. Here, the return type is the `AfterEdge` class and it has only two methods `to` and `and`. It also has

two fields `acc` and `srcNode`, which holds the accumulated results of the preceding method calls in a call chain. These values are passed to the next receiver object returned by the methods in `AfterEdge`.

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

Supporting a larger set of grammars without an advanced type system is a challenge for the developers of fluent API generators. As far as we know, no fluent API generators have been proposed to support LR grammars for widely-used programming languages. Although Gil and Levy proposed an algorithm to encode LR parsing by Java's types [21], an API generator based on that algorithm has not been developed as far as we know. Their paper [21] also reported that the compilation of a method chain based on their algorithm was extremely slow. Exploiting an advanced type system, such as Plaid's [51, 37] and context-free session types [52], might be another option. We did not choose the approach of extending these type systems either since we aimed at developing a programming tool for existing popular languages. Modifying the languages or waiting until the language supports our new type system was not acceptable.

LR grammars are typical grammars for programming languages. For example, it is known that a grammar is not LL(1) if it includes an `if` statement where an `else` clause is optional and the `if` statement may be nested in another statement. Therefore, the existing fluent-API generators, which do not support a parsing algorithm such as LALR(1), cannot generate a library skeleton for such a typical grammar. Otherwise, they generate a library skeleton that does not cause a compilation error even when the library is used as in Listing 3.4. Note that the last `else_` in line 7 does not correspond to any `if_`. Hence the method-call chain in Listing 3.4 is not syntactically valid. To detect this error, the library needs to track a stack in order to balance `if_`, `then_`, and `else_`.

3.3 Our Fluent API Generator

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

Our algorithm encodes an LR automaton by method overloading on the receiver type.

Listing 3.4: an invalid `if-else` statement

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

The encoding is fairly straightforward except the return types. Our trick is to leave the return type of the overloaded method unspecified and let the Scala compiler infer it depending on the type argument given to the receiver type. For formally describing our algorithm, we first translate the given LR automaton to a program written in our pseudo language named *Fluent*. In *Fluent*, the return type of a method is not explicitly specified; it is expected to be inferred. Then we translate the *Fluent* program into a Scala program. Since the return type cannot be omitted in Scala, we present our technique for expressing an unspecified return type in Scala. We also show the translation from *Fluent* to other languages C++ and Haskell later in Section 3.4. This chapter does not show how to construct an LR automaton from an LR grammar. We assume that the LR automaton has been already constructed in the parsing algorithm mentioned in the literature such as [26].

3.3.1 LR Automaton

We use following symbols to express an LR automaton.

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

Listing 3.5: The grammar of *oops*

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

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

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

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

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

$$\begin{array}{lcl} REDUCE(q\ \overline{qs}, nt, s\ \overline{ss}) & = & REDUCE(\overline{qs}, nt, \overline{ss}) \\ REDUCE(q\ \overline{qs}, nt, \varepsilon) & = & \delta_{goto}(q, nt)\ q\ \overline{qs} \end{array}$$

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

For example, the *oops* language (its grammar is shown in Listing 3.5) is converted to an LR automaton $M_{oops} = \langle \Sigma, N, R, Q, q_{init}, \delta_{action}, \delta_{goto}\rangle$ such that:

$$\begin{array}{lcl} \Sigma & = & \{\texttt{"o"}, \texttt{"ps"}\}, \\ N & = & \{oops, os\}, \\ R & = & \{oops \rightarrow os\ \texttt{"ps"},\ os \rightarrow \texttt{"o"}\ os,\ os \rightarrow \varepsilon\}, \\ Q & = & \{q_1, q_2, q_3, q_4, q_5, q_6\}, \\ q_{init} & = & q_1, \\ \delta_{action} & = & \text{The action table shown in Table 3.1,} \\ \delta_{goto} & = & \text{The goto table shown in Table 3.2.} \end{array}$$

Table 3.1: The action table for *oops*

Stack top	Input token		
	`"o"`	`"ps"`	$
q_1	*shift*	⟨*reduce*, `os -> eps`⟩	—
q_2	—	—	*accept*
q_3	—	*shift*	—
q_4	—	—	⟨*reduce*, `oops -> os "ps"`⟩
q_5	*shift*	⟨*reduce*, `os -> eps`⟩	—
q_6	—	⟨*reduce*, `os -> "o" os`⟩	—

Table 3.2: The goto table for *oops*

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

3.3.2 The Fluent Language

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

```
M  →  def <X> T.m() = e          method declarations
T  →  C    |   C[T]   |   X                types
e  →  new T()   |   new T().m()          expressions
```

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

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

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

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

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

Here, `X`, `Y`, and `Z` are concrete types and the method `m` is overloaded by the three declarations. Since the last declaration of `m` causes a recursive call, the return type of the last declaration `Z[Z[T]].m()` depends on the type parameter `T`. If `T` is `X[Unit]` or `Y[Unit]`, the return type is `X[Unit]` or `Y[Unit]`, respectively. If `T` is `Z[U]`, the return type depends on the form of the type argument `U`. If `U` is `Z[V]`, the return type further depends on the form of `V`. This causes infinite regression. Therefore, explicitly specifying the return type for each declaration is not feasible for this example.

This difficulty in the return-type inference appears in our encoding scheme of an LR automaton. The aim of *Fluent* is to deal with this difficulty in a separate phase of the translation. We will revisit this later in 3.3.4.

3.3.3 Translate LR Automata to Fluent

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

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

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

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

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

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

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

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

```
def <T> q1[q2[...[qn[qn+1[T]]]...]].σ() = new q'[qn+1[T]]().σ()
```

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

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

3.3.4 Translate Fluent to Scala

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

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

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

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

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

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

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

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

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

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

```
implicit def α[T]: MethodExists[T1,σ,T2] =
  new MethodExists[T1,σ,T2] {
    def invoke(receiver: T1): T2 = new T2()
  }
```

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

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

```
implicit def α[T, Ret](implicit t: MethodExists[T2, σ2, Ret])
    : MethodExists[T1, σ1, Ret] =
  new MethodExists[T1, σ1, Ret] {
    def invoke(receiver: T1): Ret = t.invoke(new T2())
  }
```

This Scala code also declares an implicit value α of type `MethodExists[`T_1`,`σ_1`,Ret]`. This α takes another implicit parameter `t` of type `MethodExists[`T_2`,`σ_2`,Ret]`. It requires that there exists a *Fluent* method σ_2 in T_2 that returns an instance of `Ret`. Since the implicit parameter ensures that the return type of the nested method call on `t` is `Ret`, we know the return type of `invoke` taking T_1 is also `Ret`. Hence the type of α is `MethodExists[`T_1`,`σ_1`,Ret]`.

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

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

3.3.5 Literals

Since every lexical token is encoded into a method name, our approach does not directly support number literals or string literals. We express such literal values by passing a runtime value as an argument to a method in a method-call chain. The examples are found in Listing 3.4. The arguments passed to `if_` and `return_` are regarded as literal values. For brevity, we do not present details of how we extend our translation to support literal values. As we later show in Listing 3.8, the input grammar to the API generator would specify some methods take a parameter. For example, `digraph` method takes a `String` object as its parameter. The translation algorithm is not largely extended. The σ method in *Fluent* is extended to take a parameter according to the input grammar and the corresponding Scala methods are also extended.

3.4 Translation to Haskell and C++

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

3.4.1 Haskell

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

Since Haskell is not an object-oriented language, we cannot directly write a method chain in Haskell. We design a similar fluent API library by using a pipe operator `|>`. The pipe operator is popular in several functional programming languages including F#, Ocaml, and Elixir. The definition of the pipe operator `|>` is as follows:

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

The pipe operator is an infix operator that takes a value for the left operand and a function for the right operand. It applies the function to the value and it is left-associative. This operator allows us to write the following chain for the fluent style DSL for SQL queries in Section 3.1.

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

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

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

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

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

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

```
class MethodExists_σ recv ret | recv -> ret where
  σ :: recv -> ret
```

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

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

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

```
instance MethodExists_σ Hs(T1) Hs(T2) where
  σ Hs(T1) = Hs(T2)
```

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

When a method in *Fluent* is the form of `def <T>` T_1`.`σ_1`() = new` T_2`().`σ_2`()`, we define the following instance of type class:

```
instance (MethodExists_σ2 Hs(T2) ret) =>
         MethodExists_σ1 Hs(T1) ret where
  σ1 Hs(T1) = σ2 Hs(T2)
```

This implementation of σ_1 constructs a value of type $Hs(T_2)$ from the given argument and then calls σ_2 with that value. The left-hand side of `=>` is a context. It requires that the instance `MethodExists_`σ_2 $Hs(T_2)$ `ret` exists and thus we can call the σ_2 function taking an argument of type $Hs(T_2)$.

Finally, we also define the `begin` function:

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

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

3.4.2 C++

Although C++ has not supported type classes yet, we can take a similar approach by using function templates. We below omit the code for memory management although the implementation used for the experiment in Section 3.5 exploits reference-counting garbage collection.

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

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

```
};
```

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

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

```
template<typename T>
auto invoke_σ(T1* stack) {
  return new T2();
}
```

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

When a method in *Fluent* is the form of `def <T>` T_1`.`σ_1`() = new` T_2`().`σ_2`()`, we declare:

```
template<typename T>
auto invoke_σ1(T1* stack) {
  return invoke_σ2(new T2());
}
```

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

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

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

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

3.5 Experiment

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

Listing 3.6: The grammar of the if/else language

```
1 syntax ifElse (Stmt) {
2   Return     : Stmt -> "return_(String)"
3   Throw      : Stmt -> "throw_(String)"
4   IfThen     : Stmt -> "if_(Boolean)" Then
5   IfThenElse : Stmt -> "if_(Boolean)" Then Else
6   ThenClause : Then -> "then_" Stmt
7   ElseClause : Else -> "else_" Stmt
8   TryCatch   : Stmt -> "try_" Stmt "catch_(String => Stmt)"
9 }
```

3.5.1 Fluent API Generator

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

TypelevelLR can generate a library in Scala, Haskell, and C++. The program in Haskell uses the pipe operator. The generated library provides a fluent API for the deep embedding style [19]. Thus, a chain of method calls to the library constructs a parse tree representing the chain. Each method call in the chain incrementally constructs the parse tree. Giving the semantics to the method-call chain is the responsibility of the library developer, or in other words, the user of TypelevelLR. The developer could implement an interpreter that executes the parse tree constructed by the generated library.

For example, TypelevelLR can generate a fluent-API library from the grammar shown in Listing 3.6. TypelevelLR reads a source file containing Listing 3.6 and generates a library that constructs a parse tree from a method-call chain if the chain is valid in the grammar. The identifiers before the colons, such as `Return` and `Throw`, are used as the type names of the parse-tree nodes constructed by the library.

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

As we mentioned in Section 3.2, the grammar in Listing 3.6 is an LR grammar. The library generated by TypelevelLR from this grammar consists of 903 lines of Scala code

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

Listing 3.7: A syntactically incorrect program using the if/else language in Listing 3.6

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

with 82 implicit functions. When compiling the user code shown in Listing 3.7, the Scala compiler reports a compilation error in line 11; the call to `else_` is not acceptable there since that `else_` does not match any `if_`.

3.5.2 Compilation Time of DOT-Like DSL

We next present the compilation time of the user code of the fluent-API library generated by TypelevelLR. We show that the compilation time is significantly shorter than Gil's approach in 2016, which was reported as being impractically slow [21]. Gil's paper [21] reported that the compilation of a chain of 26 method calls took around 30 seconds in Java. It also mentioned that the compilation time exponentially grows probably due to a design flaw of the Java compiler. We also observed similar results of our own experiment with the newer JVM.

For the experiment, we used the grammar similar to the DOT language shown in Listing 3.8. In this grammar, the iteration is right-recursive. Since the form of recursion may affect the compilation time, we also used the equivalent grammar except that the iteration is left-recursive. The differences between the two grammars are line 3, 6, 10, and 12. The followings are the syntax rules for left-recursion:

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

Listing 3.8: The grammar of our DOT-like language

```
1 Directed       : Graph     -> "digraph(String)" Stmts
2 Undirected     : Graph     -> "graph(String)" Stmts
3 StmtsCons      : Stmts     -> Stmt Stmts
4 StmtsNull      : Stmts     -> eps
5 NodeStmt       : Stmt      -> "node(String)" Ands NodeAttrs
6 AndsCons       : Ands      -> "and(String)" Ands
7 AndsNull       : Ands      -> eps
8 EdgeStmt       : Stmt      -> "edge(String)" Ands "to(String)"
9                               Ands EdgeAttrs
10 NodeAttrsCons : NodeAttrs -> NodeAttr NodeAttrs
11 NodeAttrsNull : NodeAttrs -> eps
12 EdgeAttrsCons : EdgeAttrs -> EdgeAttr EdgeAttrs
13 EdgeAttrsNull : EdgeAttrs -> eps
14 NodeAttrColor : NodeAttr  -> "color(String)"
15 NodeAttrShape : NodeAttr  -> "shape(String)"
16 EdgeAttrColor : EdgeAttr  -> "color(String)"
17 EdgeAttrStyle : EdgeAttr  -> "style(String)"
```

Then we ran TypelevelLR to generate libraries for each grammar and target language: Scala, Haskell, or C++. The programs we compiled were like Listing 3.9. The graph drawn by Listing 3.9 is shown in Figure 3.2. We wrote programs like Listing 3.9 for the three host languages and for the different numbers of graph nodes from 1 to 100. The number of the graph nodes changes the length of the method-call chain in the programs.

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

Figures 3.3, 3.4, and 3.5 present the results. We observed that the compilation time was not exponential; it looked linear or quadratic for the number of the method calls in the chain in any host language. The grammar with left recursion showed shorter compilation time. This would be because the stack does not grow deep under the left-recursive grammar and hence a fewer types were generated. Even in C++, the user program with more than 200 method calls in the chain of the right-recursive grammar was compiled within 30 seconds. We believe that our approach can generate a library for an LR grammar so that the user's code can be compiled in polynomial time. Furthermore, our compilation speed was significantly faster than Gil's approach in 2016 [21]. On the other hand, our approach is not applicable to Java since it exploits language features that are not available in Java although Gil's approach can be used in Java and maybe other languages.

Listing 3.9: A user program of our DOT library

```
1 object test {
2   import dot.rightRec._
3   def main(args: Array[String]) = {
4     val graph = begin()
5           .digraph("test")
6             .node("A")
7             .node("B")
8             .node("C")
9             .node("D")
10            .node("E")
11            .edge("A").to("B")
12            .edge("B").to("C")
13            .edge("C").to("D")
14            .edge("D").to("E")
15            .edge("E").to("A")
16          .end()
17    println(graph)
18  }
19 }
```

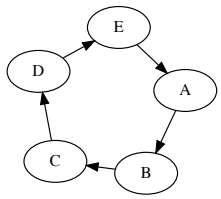


Figure 3.2: The graph drawn by Listing 3.9

3.5.3 Compilation Time of Randomly Generated Chain

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

For further investigation, we did similar experiments for other DSLs. Since we needed a large variety of method-call chains that are valid in the DSL grammars, we implemented a random method-call chain generator based on the literature [43]. The generator takes a context-free grammar and that length of a chain. Then it randomly generates a method-call chain of that length so that it will be valid in that grammar.

The experiments used the following four DSLs[2]:

- *expr*: a small subset of arithmetic expressions. It consists of only two binary operators, addition and multiplication, and parentheses. Multiplication takes precedence over addition.

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

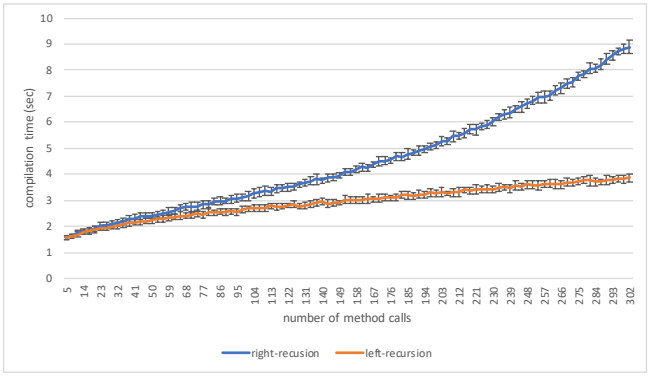


Figure 3.3: Compilation time in Scala

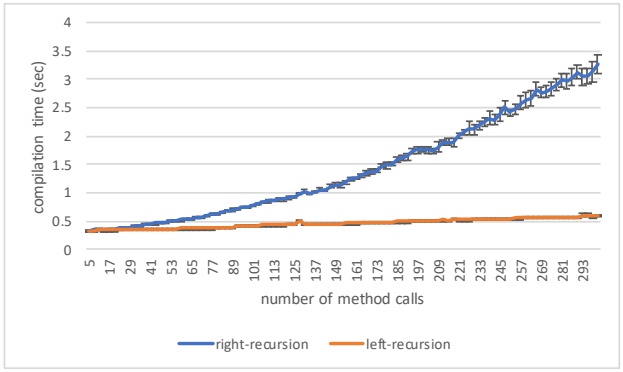


Figure 3.4: Compilation time in Haskell

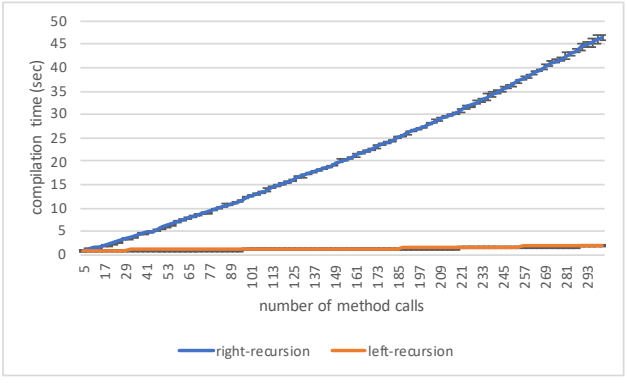


Figure 3.5: Compilation time in C++

Table 3.3: The DSL grammars

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

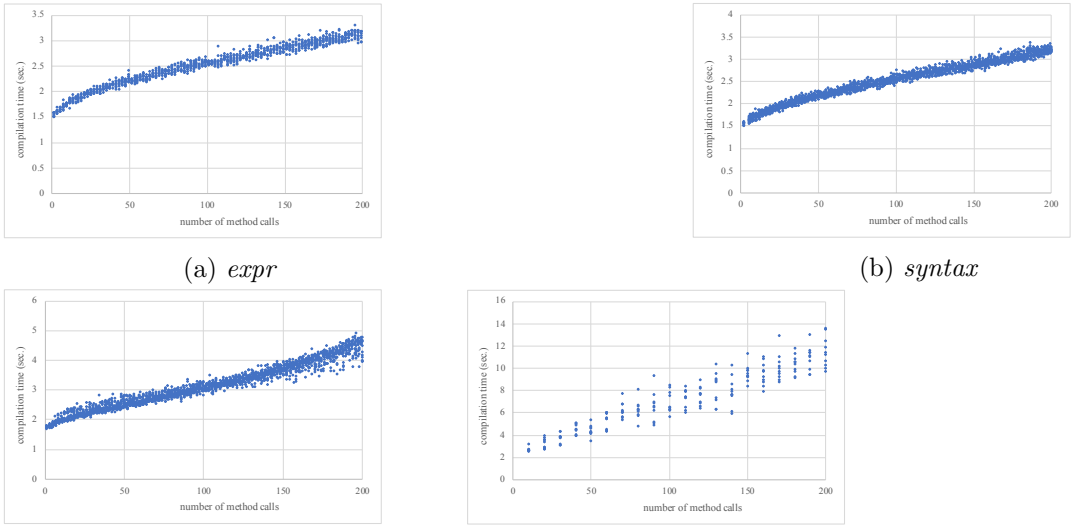
- *syntax*: the DSL we designed to express a grammar definition for TypelevelLR. A program in this DSL consists of several derivation rules. Each derivation rule consists of three parts: a name, a non-terminal symbol derived by the rule, and a sequence of terminal or non-terminal symbols. Listing 3.8 is written in this DSL.
- *while-lang*: a DSL with a similar grammar to the `while` language described in the literature [46]. It consists arithmetic expressions, boolean operations, variable assignments, `if` statements, and `while` statements. No operator precedence is given. The operators are right-associative.
- *SQL*: a subset of the SQL language. We designed this DSL by referring to the SQL grammar publicized by Apache Phoenix project [15]. This DSL supports only a subset of the `select` statement.

The sizes of the grammars of these DSLs are summarized in Table 3.3.

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

Figures 3.6, 3.7, and 3.8 show the results of our measurements for *expr*, *syntax*, *while-lang*, and *SQL*, respectively. The compilation time was the means of five runs after two warm-up runs. During the experiment for each DSL, we randomly generated 10 different instances of method-call chains for every chain-length from 1 to 200. For the *SQL* DSL, the lengths of the generated chains were every 10 from 10 to 200. All the figures are scatter plots. The horizontal axis represents the number of method calls in the chain. The vertical axis represents the compilation time in seconds.

In all the figures, we do not observe the exponential growth of the compilation time. They rather look linear. As in the experiment with the DOT-like language, the C++ compiler was slower than the other compilers. The compilation speed, however, still looks linear. Although the Scala compiler showed steady performance in these experiments, it slowed down by a factor of ten when it compiled short method-call chains in *SQL* on a different machine with a similar processor but a smaller amount of memory. The slow-down



(a) *expr* (b) *syntax*

(c) *while-lang* (d) *SQL*

Figure 3.6: Compilation time of random chains in Scala.

of the other compilers was a factor of two on this machine. This would be because the size of the *SQL* grammar was large and hence a large library was generated from that grammar and it caused frequent garbage collection. The size of the generated library was 1.8 MB (37K lines).

3.5.4 Error Messages

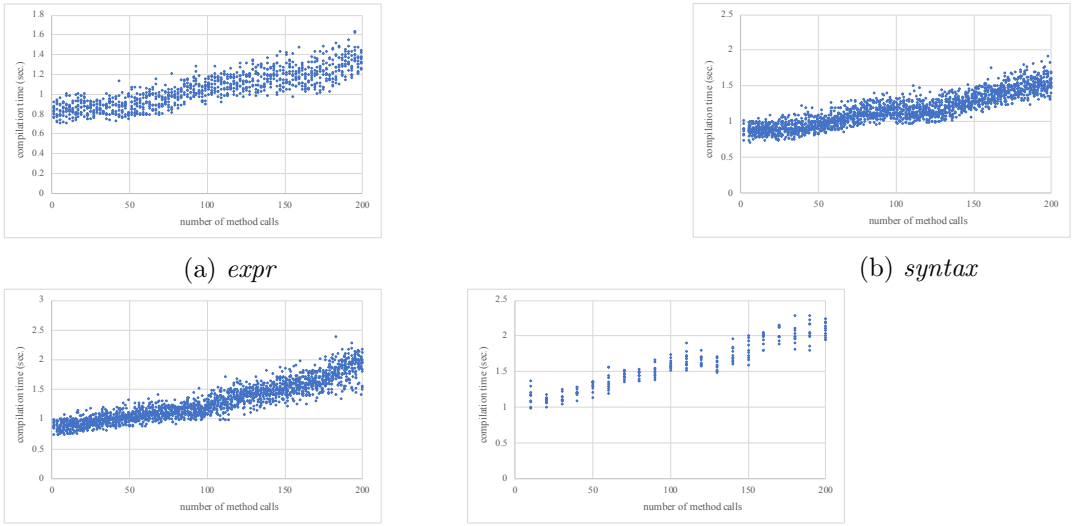
The readability of the error messages is often raised as a drawback of the type-based encoding that we adopted for TypelevelLR. An error message is printed when an invalid chain of method calls is detected, but the content of the message depends on the implementation of the host-language compiler. As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

We below show example error messages printed when the library generated by TypelevelLR detects an invalid chain of method calls. We showed the DOT-like language in Section 3.2 and implemented a Scala library for this DSL in Section 3.5.2. When the programmer using this library wrongly calls the shape method instead of the style method as follows:

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

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

```
test.scala:10: error: value shape is not a member of dotDSL.Node6[
dotDSL.Node18[dotDSL.Node7[dotDSL.Node16[dotDSL.Node16[dotDSL.Node16[
```



(a) *expr* (b) *syntax*

(c) *while-lang* (d) *SQL*

Figure 3.7: Compilation time of random chains in Haskell

```
dotDSL.Node9[dotDSL.Node1[Unit]]]]]]]]
      .edge("A").to("B").shape("dotted")
                         ^
one error found
```

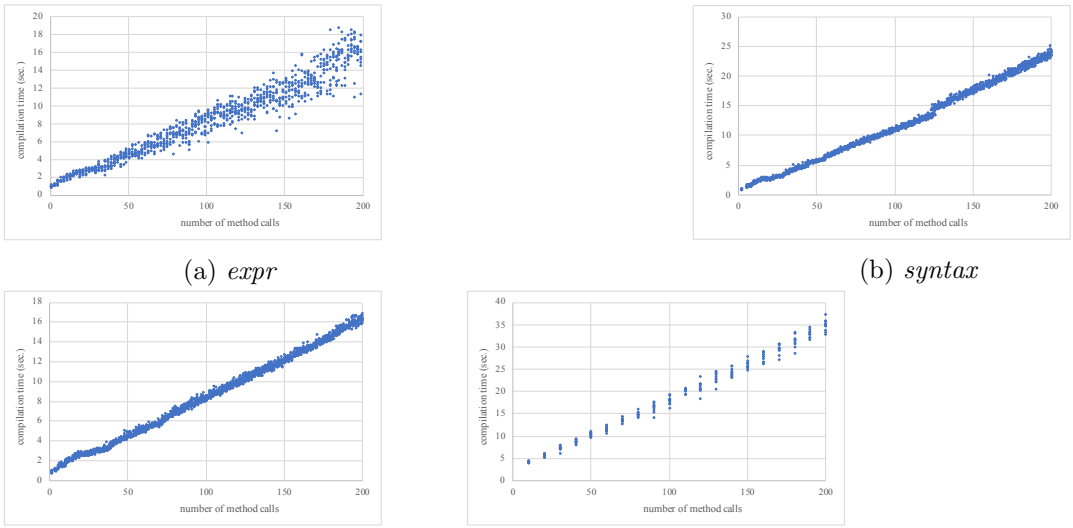
Although the state of the LR automaton is exposed in the message, we can still read the line number and the method name that causes an error. This line number and the method name indicate the correct position of the error. Only the reason of the error is difficult to read. The programmer has to know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

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

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

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

```
Test.hs:5:8:
    No instance for (Show r0) arising from a use of 'print'
    The type variable 'r0' is ambiguous
    Note: there are several potential instances:
      instance Show a => Show (Maybe a) -- Defined in 'GHC.Show'
      instance Show Ordering -- Defined in 'GHC.Show'
      instance Show Integer -- Defined in 'GHC.Show'
      ...plus 30 others
    In the expression: print
-- omit 28 lines --
Test.hs:9:34:
    No instance for (ShapeTransition
                       (Node5 (Node9 (Node6 (Node10 (Node11 (Node1 ()))))))) s0)
```



(a) *expr* (b) *syntax*

(c) *while-lang* (d) *SQL*

Figure 3.8: Compilation time of random chains in C++.

```
      arising from a use of 'shape'
    In the second argument of '(|>)', namely 'shape "dotted"'
    In the first argument of '(|>)', namely
      'begin |> digraph "small_graph" |> node "A" |> shape "rectangle"
       |> node "B"
       |> node "C"
       |> shape "doublecircle"
       |> edge "A"
       |> to "B"
       |> shape "dotted"'
    In the first argument of '(|>)', namely
-- omit 9 lines --
Test.hs:10:12:
    No instance for (EdgeTransition s0 s1) arising from a use of 'edge'
    The type variables 's0', 's1' are ambiguous
-- omit the rest and three other errors --
```

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

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

```
In file included from test.cpp:2:
In file included from ./dotDSL.hpp:627:
./dotDSL.hpp.impl:48:10: error: function 'shape_transition<dotDSL::Node6,
      dotDSL::Node18, dotDSL::Node7, dotDSL::Node16, dotDSL::Node16,
      dotDSL::Node16, dotDSL::Node9, dotDSL::Node1>' with deduced return type
      cannot be used before it is defined
  return shape_transition( this_.lock(), arg1 );
         ^
```

```
test.cpp:13:29: note: in instantiation of member function
      'dotDSL::State<dotDSL::Node6, dotDSL::Node18, dotDSL::Node7,
      dotDSL::Node16, dotDSL::Node16, dotDSL::Node16, dotDSL::Node9,
      dotDSL::Node1>::shape' requested here
      ->edge("A")->to("B")->shape("dotted")
                           ^
./dotDSL.hpp:607:6: note: 'shape_transition<dotDSL::Node6, dotDSL::Node18,
      dotDSL::Node7, dotDSL::Node16, dotDSL::Node16, dotDSL::Node16,
      dotDSL::Node9, dotDSL::Node1>' declared here
auto shape_transition( std::shared_ptr< State< Stack... > > const& src, ...
     ^
1 error generated.
```

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

3.6 Related Work

The syntax checking in any grammar class (including the grammars that are not context-free) is possible in principle if the type system of the host language is Turing-complete such as Java [25] and C++ [53]. For instance, when a given grammar is context-free, the syntax checking can be achieved by creating a CYK parser [7, 59, 34] using a Turing machine emulated on a type system [25]. However, using a Turing machine in this way is overly complicated for most DSLs and causes practical problems. The technique requires a significant amount of memory and time to check even the syntax of a small chain [25]. Our encoding technique does not cause such practical problems as shown in the experiment in Section 3.5.

Gil and Levy proposed an algorithm to translate a grammar into a DSL embedded in Java with syntax checking by the type checker [21]. It supports LR grammars. The algorithm generates the type definitions for encoding a jump-stack single-state real-time deterministic push-down automaton (JRPA), a variant of push-down automaton that can recognize deterministic context-free languages [9]. However, the algorithm by Levy and Gil suffers from the exponential growth of compilation time to the length of the chain in the worst case. That exponential growth of compilation time is caused by the exponential growth of the size of the types appearing in the chain, as Levy and Gil showed in their experiment in [21]. Our technique does not suffer from that problem as we mentioned above.

In parallel to our work, Gil and Roth proposed another algorithm to translate an LR language into a DSL embedded in Java [22]. A main difference between their work and ours is that the approaches are totally different. Their approach uses only Java generics

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

EriLex [57] and Fajita [39] are fluent API generators. These existing tools can take only an LL(1) grammar as their input while our generator supports LR(1) grammars as we described earlier. Silverchain [45] is a tool that generates an embedded DSL with sub-chaining API support. Sub-chaining API improves the user experience in that the API allows programmers to compose a chain by combining several sub-chains. The support of subchaining API is our future work to make our generator more practically applicable. However, Silverchain can take only an LL(1) grammar.

Several techniques for semantic checking have been also studied although our work focuses only on the syntax correctness of DSL code. For example, AraRat [20] uses C++ template metaprogramming to check the syntax of SQL queries and the type-safety with respect to the database schema. The integration of such semantic checking and our fluent API generator is also future work. Other host language mechanisms such as operator overloading can be used to emulate a DSL sentence in a program written in a general-purpose language [5]. Although we focused on fluent APIs in our work, mixing those mechanisms with our technique would be a possible direction for future research.

Checker Framework [12] is a framework to extend Java's type system. A number of static checking including syntax checking can be implemented by using Checker Framework. Standalone static analyzers such as Android Lint [31] can statically check the syntactic correctness of method chains. However, since those external standalone checkers are separately developed from the library, it might not be easy to maintain the checkers up-to-date to be compatible to the latest version of the library. Our generator would not cause this problem since the syntax checker is included in the library and they are developed together.

If we can freely extend the syntax of host languages, we can implement an embedded DSL that provides a more natural API without using a chain of method calls. ProteaJ [30] and Wyvern [47] are programming languages that natively support syntax extension. Since their extensibility is achieved by their underlying language mechanism such as their new type systems, this approach is not directly available to widely-used current programming languages without modifying these languages. A fluent API using method-call chains is a technique easily applicable to these languages.

3.7 Concluding Remarks

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

Chapter 4

Collecting Cross-Language Cyclic Garbage References

4.1 Introduction

Today, we can use various kinds of programming languages. Each language has their own libraries, and we sometimes develop an application which is written using multiple languages since there are different problem domains each library specializes in. For example, in web applications, the server-side is often developed in Java or PHP, and the client-side is usually developed in JavaScript. In machine learning, many developers use Python to write a program, but its internal calculation is developed in C language. In graphical applications, developers may use C bindings because programming languages may not support graphics libraries by the standard.

A mechanism that enables us to call a function written in another programming language is called Foreign Function Interface (FFI). An FFI is especially important for a new programming language since it does not have enough libraries yet. Previously, it was enough if there exists an FFI between C language since most programs to reuse were concentrated in C language. However, today's popularity of script languages like Python and JavaScript has increased the importance of FFIs between them. For example, PyCall is an FFI library in Julia, which enables us to call Python functions. PyCall also provides Ruby implementation. There is also PyV8, an FFI library in Python, which enables us to call JavaScript functions. Note that script languages are usually managed languages; they have their own garbage collector, and objects are managed by them.

An FFI between managed languages raises a cooperation problem between garbage collectors. Usually, a garbage collector is designed to manage the entire heap memory, but an FFI between managed languages introduces per-language memory regions managed by per-language collectors. That makes collectors unable to correctly determine whether each object is garbage or not since collectors cannot access objects in other languages. A

memory leak can occur when a collector mistakenly determines an object alive, even if it is garbage. Also, a memory error can occur, and the error can crash the VM when a collector mistakenly determines an object as garbage even if it is used later.

It is possible to customize garbage collectors to cooperate, but the customization should be as small as possible. Since a garbage collector is a large-scale complicated program, customizing it is not easy. Moreover, customizing a garbage collector makes the VM customized. If you want to use a customized VM for a while, you have to apply updates and security patches by yourself since applying an official patch may cause inconsistency between your customization. Since we are considering a new programming language that supports FFIs to enable programmers to reuse functions written in another language, we aim to keep the existing language without customizing it.

We propose a garbage collection algorithm for FFI, which improves the accuracy of garbage detection. Our method can be implemented without customizing both garbage collectors cooperating with each other. Our method is based on the idea that a collector can provide the missing information of another collector by cloning the object graph. Here, we mean object graph, the graph of objects where the edges are reference relations. The reason why collectors cannot correctly detect garbage objects was the inaccessibility of objects in the opposite region. By cloning the object graph, a collector can supply the missing information of reference relations to the other collector. Since cloning all objects is memory-consuming, our method compresses the information before cloning. Our algorithm depends on bloom filters [4] during the analysis of the object graph. Since a bloom filter is a probabilistic data structure, the object graph constructed by our method possibly contains some errors. The inaccuracy may leave garbage objects without collecting them but never determines an object is garbage and collect it if it is not really a garbage object. In section 4.3.2 we will discuss the effect of false-positives in bloom filters.

In the rest of this chapter, in section 4.2 we show what will happen when the cooperation between garbage collectors has failed. In section 4.3 we show our algorithm. In section 4.4 we show our experiments. Finally, section 4.5 concludes this chapter.

4.2 Foreign Function Interface and Distributed Garbage Collection

Mechanisms that enable us to call functions written in another programming language are called Foreign Function Interface (FFI). FFIs are important, especially for new programming languages, since an FFI enables them to reuse libraries written in other languages. Previously, FFIs were enough if they could call C functions like Java Native Interface [48], libffi [41], Python ctypes [16], and so on. Today, FFIs are also paid attention which can call functions written in other languages from the C language like PyCall [32] and PyV8 [24].

For example, listing 4.1 shows a Ruby program which contains library calls to *React.js*, a JavaScript library for interactive web pages. We can use React.js by defining a class

Listing 4.1: An example Ruby program which calls a JavaScript library via an FFI

```
1 class Button < React::Component
2   def render
3     @dom = React.createElement('button')
4     @dom.onclick = proc do
5       @status = 'clicked'
6     end
7     @dom
8   end
9 end
```

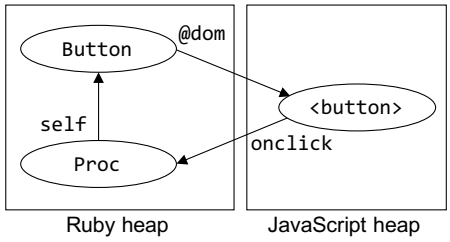


Figure 4.1: Objects allocated by listing 4.1

which extends `React::Component`. Programmers specify what will be displayed on the screen by what `HTMLElement` instance the `render` method returns. The `render` method in `Button` defined in line 2-8 first creates a `button` element by calling `React.createElement` and store it as `@dom` as shown in line 3. Then register an `onclick` callback function to that `button` element. The callback function changes `@status` in the `Button` object to `'clicked'`. Finally, it returns the `button` element as the result.

Figure 4.1 shows how objects are allocated in the memory. The `button` element created by `React.createElement` is allocated in the JavaScript heap since an `HTMLElement` instance is a JavaScript object. In contrast to the `button` element, the `Button` object and the callback function are allocated in the Ruby heap. Thus, the reference `@dom` is a *remote* since it crosses the border between languages. We say an object is *public* when at least one remote reference points it.

Cooperation between garbage collectors can cause a problem when multiple programming languages connected by FFIs perform distinct garbage collections. A garbage collector is usually designed for single programming language thus their algorithm assume that all objects are known and managed by the collector itself. An FFI between managed languages breaks this assumption. There are several garbage collectors and each collector manage each memory region, no collector knows there exists objects managed by other collectors, and also cannot access them. As the result, the collectors' determination of whether an object is garbage or not become incorrect since no collector can follow a path which goes

through another memory region. If an object is mistakenly determined that it will be used later although it is unreachable, the object will not be collected and a memory leak can occur. On the other hand, if an object is mistakenly determined as a garbage although it will be used later, a memory error will occur and the VM may crush. We will show an example in later section.

It is possible to customize garbage collectors to be cooperative, but the customization is very difficult. Since an FFI calls functions written in an existing programming language, there also already exists a garbage collector. Customizing garbage collectors is very hard since usually it is a very complicated program. For example, the garbage collector of YARV (a common Ruby VM) is very large; it consists of 12,000 lines of C program. Bugs in a garbage collector is very difficult to fix. Usually bugs in a garbage collector writes a broken value into somewhere on the memory, function invocation or variable assignment or something else copies the broken value, and at some point, the VM notices that a value is broken and reports a memory error. Here, there are nothing which indicates where the broken value came from. It is desirable to keep collector customization as small as possible. We are considering about new programming language and to call functions written in existing programing language from the new language. Thus, our aim is to discover a method which keeps the existing language from customizing.

Distributed garbage collection [49, 54] handles a similar problem. In distributed environment, both memory space and computational resources are separated. In such environment, garbage objects are collected by per-machine local collectors. Since local collectors also cannot access objects in another machine, local collectors communicate to each other and try to detect garbage objects correctly. As far as we know, distributed garbage collections require distributed-specific garbage collection algorithm. That is, in other words, no distributed collection algorithm can reuse existing garbage collector.

Naive FFI implementation and memory leaks

In an application that has multiple garbage collectors working cooperatively, a memory leak or a memory error may occur when the cooperation fails. This section shows an example of such memory leaks. First, we show a naive FFI design that provides only two interfaces: `eval` function and remote references. Then, we show how remote references are managed by naive garbage collection. Finally, we show how a memory leak occurs.

`Eval` is a function which evaluates an argument string as an expression in the opposite language. Here, we say *host* to mean the language calling `eval` and *guest* to mean the language it evaluates the argument string. The computation in the host language is blocked until the requested evaluation in the guest language finishes.

Since data representation depends on the programming language, our library converts arguments and return values. The conversion is bi-directional. We apply the same conversion strategy for both directions. When a value is primitive (i.e. a numerical value, a character string), the conversion just creates a copy in the opposite language. When a

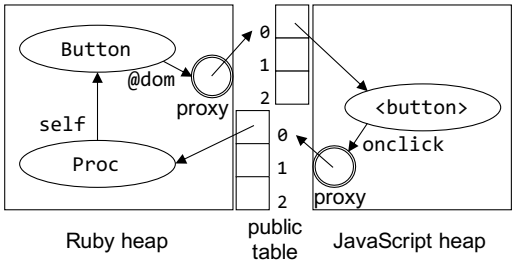


Figure 4.2: An image of remote references implemented by using public tables

value is not primitive (i.e. an instance of a user defined class), the conversion creates a reference pointing the non-primitive object remotely instead of converting it.

Our FFI library maintains a *public table* to implement remote references. References cannot point an object in another programming language directly. Instead of creating a true remote reference, our library registers the referent object to the public table and creates a *proxy* to the referent. Since an object registered to the public table is referred from another language, it is a *public object*. References pointing to proxy objects behave as they are remote references, thus operations on a proxy object are transferred to the referent. A proxy object identifies the referent by looking up the public table in the opposite language. Figure 4.2 shows the same object graph to figure 4.1, but implementing remote references by using a public table.

This FFI library can collect remote references by using finalizers. Basically, garbage collectors never determine public objects as garbage since they are globally accessible via public table. Since we cannot determine when a public object is accessed via a remote reference, it is better to leave remote references from collecting. On the other hand, collectors treat proxies in the same way to normal objects. By deleting proper entry from public table when a proxy is collected, we can enable collectors to collect public objects since there are no reference from public table.

We can consider this garbage collection on public table as a reference counting, thus this approach cannot collect cyclic references. The basic idea was "if all remote references pointing a public object is removed, the public object is no longer public." In other words, the collector is counting how many remote references are pointing each public object, and make the object non-public when the count decreases to zero. Reference counting methods cannot collect cyclic garbage objects. Reference counting collectors handle cyclic garbage by techniques like backup tracing collection or trial deletion, however both approaches does not fit our problem. We call this kind of cyclic garbage *"the last piece of cake"* since collectors seem to be waiting for each other to start collecting them.

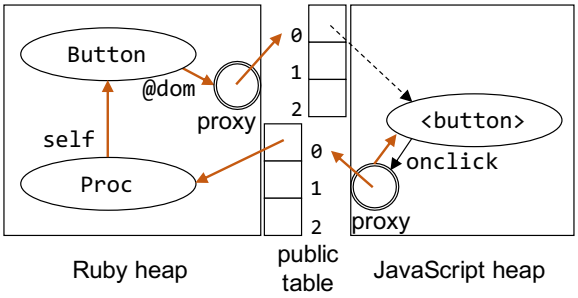


Figure 4.3: After reflecting reachability in Ruby heap to JavaScript heap
The reference from a proxy to `<button>` represents the path going through Ruby heap. The dotted line represents a weak reference. Thus, the `<button>` is collectible.

4.3 Our Object Graph Cloning Method

We propose an object graph cloning method that can collect *the last piece of cake* without customizing both of the two garbage collectors. Our method assumes an FFI library to be implemented in a naive way shown in section 4.2. Our idea is that when we clone all objects in a language to the other language, the collector in the other language can simply collect all garbage cycles in the usual way. The reason why no collector could detect a cross-language garbage cycle was due to a lack of information. We can make a collector inform the other collector of its objects to make the other collector able to collect the last piece of cake. Furthermore, we can inform a collector by objects managed by the collector itself so that that collector needs no customization to trace informed reference relation.

Since it is memory consuming if we really clone the whole object graph, we compress it before cloning. The minimum information the opposite collector requires to determine correctly whether each object is garbage or not is reachability from root-set to proxies and reachability from public objects to proxies. Thus our method collects reachability by a similar technique to marking of tracing GC, and clone it to the opposite language as references from proxies to public objects. Figure 4.3 shows an image of heap memory after cloning object graph. Since the reference from public table to `<button>` is weak, the collector can collect it if there are no path from root objects.

Our method stops the world during collection. This is for preventing user programs from breaking invariants in our algorithm. Although there are parallel garbage collection algorithms, it is not clear whether we can implement them without customizing the VM. If it is stop-the-world, we can implement without customizing a VM if the VM has global interpreter locks like Python or Ruby, or the concurrency model is an event loop like JavaScript.

Listing 4.2: pseudo code of collection phase

```
1 stack := φ
2 for pid, p ← public_table do
3   mark(p) ← { pid }
4   push(stack, p)
5 end
6 for root ← rootset do
7   mark(root) ← Ω
8   push(stack, root)
9 end
10 while stack ≠ φ do
11   cur ← pop(stack)
12   for next ← pointers(cur) do
13     if mark(cur) ⊈ mark(next) then
14       mark(next) ← mark(cur) ∪ mark(next)
15       push(stack, next)
16     end
17   end
18 end
```

4.3.1 Detailed algorithm

Our algorithm consists of two phases; (1) reachability collection and (2) reflection. The first phase traverses the object graph and collects the reachability from public objects to proxies. Listing 4.2 shows a pseudo-code. The algorithm is similar to the marking algorithm in tracing GC, but there are some differences. Our algorithm does not collect binary information of whether each object is garbage or not, but the reachability from public objects to proxies. Thus, the root set is a set of public objects, and the mark of each object is not a bit but a set of publication ids. The difference between marks makes the algorithm also different. Since there are more than two states for each mark value, the collector must update them as many times until they converge. Our collection phase also examines reachability from root to prevent collectors from collecting objects remotely reachable from roots. If a public object is found as root-reachable, the algorithm no longer needs to collect the information about which proxies the public object is reachable from. Thus, in Listing 4.2 we show this idea by representing marks of root objects as Ω, the set which contains all elements. Note that Ω contains not only publication ids but also a special element "root." This is because collectors should not collect root-reachable objects even if no object references proxies.

The second phase of our algorithm is reachability reflection phase. This phase reflects collected reachability information in the opposite language and collect parts of cyclic garbage. Listing 4.3 shows a pseudo code. The pseudo code is expected to run in the

Listing 4.3: pseudo code of reflection phase

```
1 for p ← proxies do
2   refs(p) ← φ
3 end
4 for pid, mark ← collected_relation do
5   if mark ≠ Ω then
6     for p ← proxies do
7       if dest(p) ∈ mark then
8         push(refs(p), publib_table[pid])
9       end
10    end
11  end
12 end
13 for pid, pub ← public_table do
14   if mark ≠ Ω then
15     make_weak(public_table, pid)
16   end
17 end
18 local_gc()
19 for pid in keys(public_table) do
20   make_strong(public_table, pid)
21   refs(p) ← φ
22 end
```

opposite language to the first phase. Be careful with the public objects and proxies because they are viewed as swapped in the opposite language. The algorithm consists of five steps. First, initialize *refs* of each proxy to an empty set. Second, reflect the collected information; create references from proxies to public objects. In this step, no reference is created when the public object is reachable from the root since the object will not be collected without any references. Third, replace references from public tables to public objects by weak reference. Here, there become no global strong reference to public objects is guaranteed, thus collectors have chance to collect public objects. Fourth, invoke the local garbage collection. The collection is expected to collect parts of cyclic garbage. Finally, restore references from public tables to public objects by strong reference.

4.3.2 Data representation of mark sets

As shown in section 4.3.1, our algorithm prepares a number of sets for marks of objects. We chose bloom filter [4] as the data representation of them. Common data representations of sets (i.e. hash tables, binary search trees, and so on) are complex, thus unacceptable memory- and time-consumption are supposed. Using bitmaps is another memory-efficient option, but changing their size will cause large time-consumption. Since it is not clear how we can assume the maximum number of an application will create, we do not adopt bitmaps.

A bloom filter is a memory efficient data representation of a set. The bit width of a bloom filter does not depend on the number of elements – it is fixed. The union operation is done by just one bitwise OR operation. The drawback of the memory efficiency is false-positiveness. A bloom filter possibly answers an element is contained even if the element is not inserted to it. However, a bloom filter has no false-negativeness; a bloom filter never answers an element is not contained if the element was inserted to it.

In our algorithm, false-positives of bloom filters are viewed as false-reachability. Our algorithm may determine reachable from a public object to a proxy even if they are not really reachable, and may leave some garbage objects without collecting them. The probability of false-positives depends on how many bits a bloom filter contains and how many public objects are created. Note that, the false-negatives which bloom filters never raises may make it worse; false-negatives may cause wrong collection and may crush the VM. In our algorithm, false-negatives are viewed as false-unreachability. Thus, an object can be collected even if it is still accessible.

We can avoid false-positives from continuing over GC cycles by choosing hash functions used in bloom filters randomly for each GC cycle. A bloom filter uses several hash functions in it. If the hash functions are the same, false-positives appears with the same combination of elements. Thus, the same false-positives are seen if the object graph has not been changed and the hash functions are the same. We use a series of hash functions generated from a single seed number randomly selected for each GC cycle. Our hash function is based on *xorshift* [44]. Listing 4.4 shows a pseudo code of our hash function. The first hash value

```
1 #define N /* the number of hash functions */
2 int seed;
3 void calc_N_hashes(int result[], int x) {
4     result[0] = seed ^ x;
5     for(int i = 1; i < N; i++) {
6         result[i] = xorshift(result[i - 1]);
7     }
8 }
```

Figure 4.4: A pseudo code of our hash function

Ruby VM	customized
JavaScript Engine	not customized
FFI library	developed
microbenchmark application	developed

Table 4.1: What we developed for our experiments

depends only on the seed value and the value to be hashed. The $(n+1)$-th hash value depends only on the n-th hash value. By updating the seed by applying another xorshift function, we avoid the same false-positives from appearing over GC cycles.

We use a bloom filter where all bits are one to represent Ω, the mark which means the object is reachable from the root-set. Such bloom filter is considered to contain any element thus the corresponding remote references will not be collected anyway.

4.4 Experiment

We had an experiment to understand the performance of our algorithm. There are two criteria we are interested in. The first criterion is how long our algorithm takes for each collection. Since our method interrupts user programs during each collection, it is important to understand how long the pause time is. The second criterion is how many objects escape from the collection because of the false-positive errors. Since a bloom filter is a probabilistic data structure, our algorithm possibly leaves garbage objects from collecting them as discussed in section 4.3.2. The possibility increases as many remote references exist at the same time. It is also important to understand how many garbage objects will escape from the collection since the objective of our algorithm is to reduce memory consumption.

Table 4.1 shows what we developed for this experiment. We customized the Ruby interpreter and implemented our method as a backup garbage collection which is independent from the normal garbage collection. Our backup garbage collection starts when a user program calls a specific function and the collection interrupts the interpreter until it

Listing 4.4: A pseudo code of our micro benchmark

```
1 T.times do
2   cycles = (1..N).collect { make_cycle }
3   cycles = []
4   collect_garbage_cycles
5 end
```

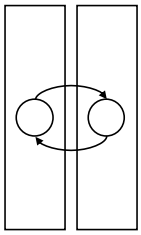


Figure 4.5: An image of cyclic reference

finishes. The version of the Ruby interpreter which we customized is `ruby 2.8.0dev`. The bit width of bloom filters used in our algorithm is 64 and the number of hash functions used in bloom filters is 5.

Also, we developed an FFI library which enables Ruby program to call function written in JavaScript. We adopt the naive FFI design shown in section 4.2 to our library. All our experiments use this FFI library.

In the rest of this section, section 4.4.1 shows an experiment on how long our garbage collection algorithm takes to collect garbage cycles. Section 4.4.2 shows another experiment on how our algorithm fails to collect garbage cycles due to the false-positiveness of bloom filters. Both experiments run on a machine with Intel® Core™ i7-4770S, 16 GB memory, and Ubuntu-18.04.5 LTS.

4.4.1 Garbage Collection Time

We experimented on how long our algorithm takes to collect garbage cycles. Usually, how long a collection takes depends on how objects are connected in the heap. Thus, our objective is to reveal the relation between the collection time and the shape of the object graph in the heap. First, we experimented on the relation between the collection time and the number of cyclic garbage.

We implemented a micro benchmark for this experiment. The benchmark repeats following three steps: create cyclic references, throw them away, and then invoke the object graph cloning collection. Figure 4.4 shows a pseudo code of this benchmark. First, line 2 creates N cyclic references and store them as `cycles`. Each cyclic reference consists of both one Ruby object and one JavaScript object which refer to each other. Figre 4.5

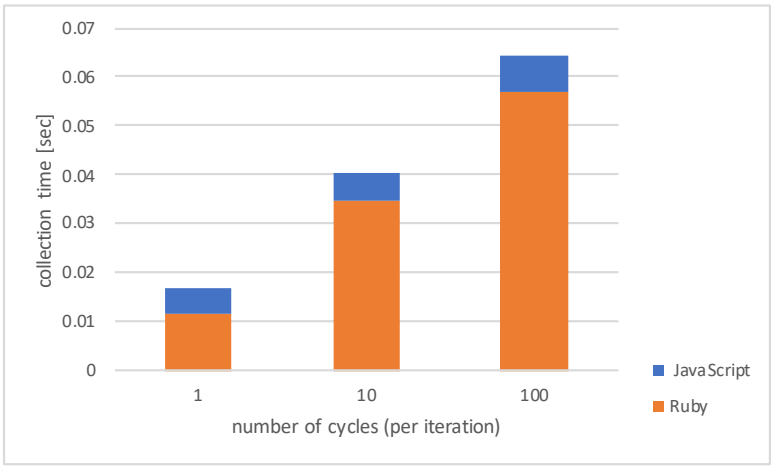


Figure 4.6: The relation between the collection time and the number of cycles

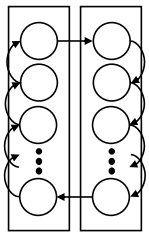


Figure 4.7: An image of a *long* cycle

shows an image of a cyclic reference. Next, line 3 throws all cyclic references away and turns them into garbage. Then, line 4 invokes our algorithm and collect cyclic garbage. In this experiment, we run the micro benchmark and measure how long the garbage collection (in line 4) takes in average over eleven iterations.

Figure 4.6 shows the result. The horizontal axis represents N, how many cycles the micro benchmark creates in each iteration. The vertical axis represents how long the garbage collection takes. The orange region represents how long does it take in the Ruby collector and the blue region is for the JavaScript collector.

Garbage collection time increases as the number of garbage cycles increases. However, the increase of garbage collection time was less than ten times even though the number of garbage cycles was increased ten times. In a more detailed result, more than 90% of the computation time in the Ruby collector is reachability collection phase. Most of the computation time in the JavaScript collector was garbage collection.

We also experimented on the relation between the collection time and the shape of cyclic garbage. In this experiment, we run the same micro benchmark to the previous experiment. Its pseudo code is shown in figure 4.4. However, we change the shape of cyclic

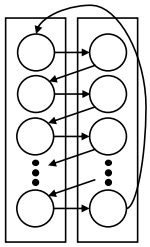


Figure 4.8: An image of a *coming-and-going* cycle

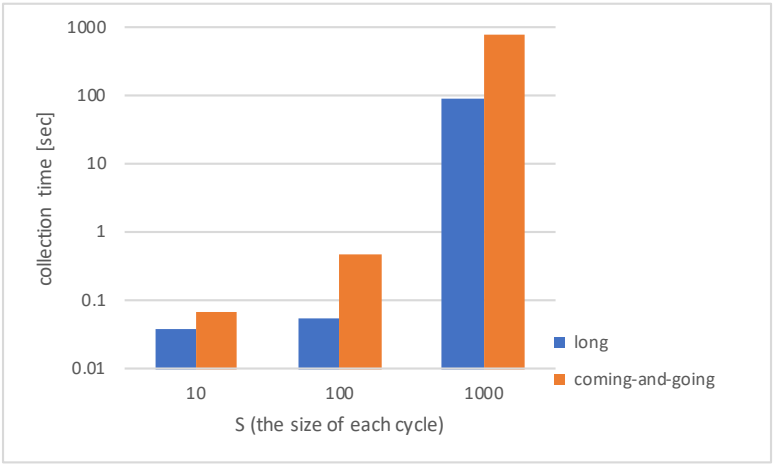


Figure 4.9: The relation between the collection time and the shape of cycles

references created by `make_cycle` in line 2. We also measure how long garbage collection takes which is invoked in line 4.

We experimented on three shapes of cyclic references for this experiment. The first shape is *simple* shape shown in figure 4.5. The second shape is *long* which consists of both S Ruby objects and S JavaScript objects where S denotes the size and the objects are connected in order. Figure 4.7 shows an image of a *long* cycle. The third shape is *coming-and-going* which also consists S Ruby and JavaScript objects, but connected alternately. Figure 4.8 shows an image of a *coming-and-going* cycle. Note that, a *coming-and-going* cycle contains S times more remote references than a *long* cycle.

Figure 4.9 shows the result. The horizontal axis represents the S which denotes how many objects each cycle contains. The vertical axis represents how long garbage collection takes. In this experiment, N is fixed to 10. The blue bar represents the result for *long* cycles, and orange bar is for *coming-and-going* cycles. Note that, the vertical axis is logarithmic. Thus, the leftmost blue bar (the result of *long* cycles when $S = 10$) is almost 0.04 seconds although the rightmost orange bar (the result of *coming-and-going* cycles

Listing 4.5: A pseudo code of modified micro benchmark

```
1 cycles = []
2 T.times do
3   N.times do
4     cycles << make_cycle
5   end
6   cycles = cycles.sample(N)
7   collect_garbage_cycles
8 end
```

when $S = 1000$) is about 765 seconds.

Except the case when $S = 10$, the garbage collection for *coming-and-going* cycles takes about ten times longer than it for *long* cycles when S is equal. This result suggests that both the number of objects and the number of remote references has an impact on garbage collection time, and the impact of number of objects is stronger. The result of *long* cycles when $S = 1000$ is more than 1,000 times slower than the result of $S = 100$ although S is only ten times larger. We suspect that this result is due to a secondary effects of heavy memory consumption.

4.4.2 Effect of False-Positives

We also experimented on how our algorithm fails to collect garbage cycles due to the false-positives caused by bloom filters. The effect of false-positives also depends on how objects are connected in the heap. We use a similar microbenchmark to listing 4.4, but we modify it. The difference is that in each iteration, we choose N cycles randomly and prevent them from thrown away. Listing 4.5 shows a pseudo code. Line 3-5 creates N cycles and append them into the end of `cycles`. Each call to `make_cycle` produces a *simple* cycle shown in figure 4.5, the same shape to the first experiment. Line 6 randomly selects N cycles from `cycles` and throw other cycles away. Then line 7 invokes our garbage collection. Since it is a specific scenario that all cycles are thrown away simultaneously, the modified microbenchmark is expected to give a more realistic result.

In this experiment, we count how many remote references each garbage collection. Since the possibility of false-positive errors increases as many remote references exist, we expect that the number of remote references which survive each collection start to increase linearly from a certain point. Note that, then number of remote references must not be less than N since we keep N cycles over each iteration.

Figure 4.10 shows the result. The horizontal axis represents the number of iterations. The vertical axis represents the number of cycles survived each iteration. The vertical axis is normalized by dividing by N. Each line represents the results for each N of 1, 10, and 100.

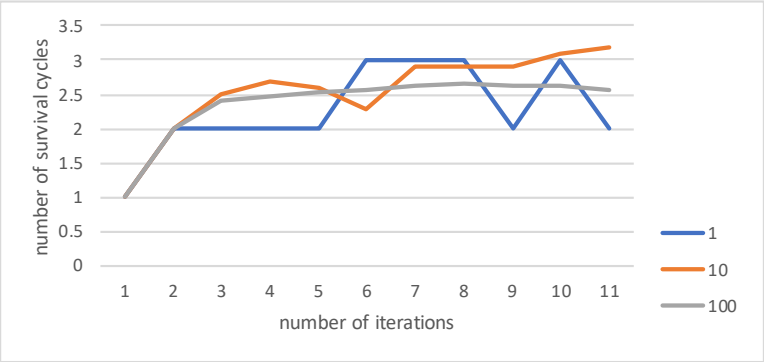


Figure 4.10: How many cycles survive for each iteration

The first garbage collection does not collect any cycle. Since the first iteration creates N cycles and throw no cycle away, this is an expected result. The second garbage collection also does not collect any cycle. This is unexpected result since the second iteration also creates N cycles and throws N cycles away. We suspect that there are some references to cycles remaining in somewhere like a stack frame or a register. In the remaining iterations, the number of remote references increases and decreases and it is about between 2 and 3 times N. This result suggests that the possibility of false-positive errors is not large enough to prevent garbage cycles from collected when N is not larger than 100.

4.5 Concluding Remarks

In this chapter, we discussed a memory leak problem which we call *the last piece of cake problem*. This problem is caused by an FFI between programming languages when both languages support garbage collection. Since script languages are getting more popular in recent years, FFIs targeting programming languages that support garbage collection will increase. Studying on the last piece of cake problem is important to prevent problems in the future.

We proposed an object graph cloning algorithm that can collect cross-language cyclic references without customizing both collectors connected via an FFI. Our algorithm copies an object graph in a language to another language as a reduced object graph in that language, thus the garbage collector in the other language can detect all cross-language cyclic references without customization. To obtain an object graph within a smaller amount of memory consumption, we adopt a bloom filter to represent a set of remote references. Although the false-positiveness of bloom filters can remain garbage objects from the collection, it never causes illegal collection.

We had two experiments to understand the performance of our algorithm. The first experiment was on the relationship between how long our algorithm takes to collect garbage

cycles and the number and the shape of garbage cycles. This experiment reveals that collection time increases as the number of cycles increases and the number of objects a cycle contains increases. Especially the number of objects a cycle contains is observed to have a strong impact on the collection time. The second experiment was on the impact of the false-positiveness of bloom filters on the collection rate. This experiment reveals that false-positive errors do not have a fatal impact on collection rate when there are only a hundred simple shape cycles.

It is a limitation of our method that our garbage collection algorithm is stop-the-world, thus its pause time may be very long. Long pause time due to garbage collection is not desirable. Reducing pause time is future work. There is room for improvement not only in the implementation but also in the algorithm. We consider that techniques in incremental or concurrent garbage collection may be applicable to our algorithm. There is also room for improvement in false-positive error handling. Improving garbage collection rate is another future work.

Chapter 5

Self-Reflective Garbage Collection for Customizing Garbage Collectors

5.1 Introduction

A garbage collector (GC) is a runtime component that is often customized for heap analysis [33, 40] or runtime object evolution [56, 8]. For example, string deduplication [28] is typical runtime object evolution. String deduplication is a technique to save memory by replacing duplicated string objects with a representative string object when they are immutable. The usual way to replace an object is to rewrite all references pointing to the object, but it costs a lot of time to find all such references. Since a garbage collector often rewrites all references to do compaction, customizing the garbage collector is a well-known technique for implementing string deduplication without large overheads.

The customization is however not a simple task. A typical approach to customize a collector is to rewrite the program that implements the collector. Since the collector is a part of a programming language system, the customization does not goes like customizing applications. Developers have to modify a low-level implementation of the GC component, and most implementations do not provide clean programming interfaces for such customization.

GC customization via computational self-reflection [50] is a promising approach, although self-reflection has not been well investigated for garbage collection as far as we know. Suppose that we run a program in a language L_{base} supporting garbage collection. A reflective programming interface for customizing the garbage collector allows developers to modify the behavior of that collector via a program written in the *base* language L_{base} instead of the language the collector is implemented in. That program can intercept the garbage collection during collection time and run as if it is part of the implementation of

the garbage collector. For clarification, we call such a program a *meta* program from this point on.

A design problem of such a reflection interface for garbage collectors is how to manage objects created by a meta program. Since a meta program is a normal program, it may create objects and also turn them into garbage during runtime. The garbage collector customized by the meta program should also collect these garbage objects, but naive implementation may cause infinite regression. Note that a meta program may create an object that will substitute another live (base-level) object when it implements object evolution. Hence an object created by a meta program should not be distinguished from normal (base-level) objects.

One possible approach is to allocate some special heap memory that only a meta program can use. Then we can separately collect garbage in that heap memory by a dedicated collector. However, this approach does not satisfy our motivating requirement. The customized garbage collector never collects the objects created by the collector itself. A different, uncustomized collector c_2 collects them. We could customize that different collector just like the original one. However, the customization would introduce the third heap space to place objects created by the second customized collector c_2 and the third space is managed by the third collector c_3 that uses the fourth heap space. Thus, this approach causes infinite regression.

Another approach would be to let a meta program allocate objects in the regular heap memory where the garbage collector is concurrently collecting garbage. It seems promising but its heap memory consumption would be a problem. If placing an object into the heap, the collector has to traverse the whole heap later to test whether the object is alive or dead. Thus, if a meta program allocates objects in the regular heap, these objects will not be collected during the current GC cycle. Since a meta program may create a large number of objects when a live object is found and copied, the meta program may create more objects in total than the existing ones in the heap. Such huge memory consumption is unacceptable.

This chapter proposes a novel algorithm for reflective garbage collection, *buffered garbage collection*. This algorithm allows a meta program customizing a garbage collector to create objects that are also collected by that customized collector while avoiding infinite regression of garbage collection. Buffered garbage collection is based on copying garbage collection [14, 6] but it manages the third space named *buffer* to buffer objects created by a meta program. The buffer space is similar to the nursery space of the generational algorithm. An advantage of this algorithm is that it will consume a smaller amount of heap memory than other approaches. Through a reflection interface, a program can register a callback function that is invoked whenever an object in the old heap is copied to the new heap during GC time. This callback function can customize the collector as it is a meta program in our algorithm. The objects created by the meta program are stored in the buffer space and effectively collected by the customized collector. Since our algorithm introduces staged collection, those objects are not collected until the garbage collection moves into a

stable state. We have implemented an interpreter for a subset of Scheme with the proposed garbage collector.

The rest of this chapter is as following. In Section 2, we detail a problem that occurs when we apply computational self-reflection to garbage collector; why infinite regression and unacceptable memory consumption occurs. In Section 3, we propose Buffered Garbage Collection; our novel garbage collection algorithm that avoids both infinite regression and unacceptable memory consumption. In section 4, we compare our method to other possible approaches to clarify advantages of our method. In section 5, we show a result of an experiment to confirm our method avoids unacceptable memory consumption in our exprimental implementation. In section 6, we present related works and in section 7, we concludes this chapter.

5.2 Self-reflective customization of Garbage Collector

Sometimes, garbage collectors are customized. One typical purpose is to make collector to manage objects using application specific knowledge so that both memory efficiency and calculation performance become better. For example, string deduplication [28] replaces duplicated string objects with a representative string object when they are immutable. Applying string deduplication for web server applications does not only save memory but also speed up calculation because comparison between string objects become faster.

A typical approach to customize a garbage collector is to implement the customization in the language that implements that collector. For example, we can customize the garbage collector of the Java virtual machine in C++ since it is implemented in C++, but such a customization has to consider a number of low-level implementation issues of Java. First, the presentation of Java objects is more complicated from the perspective of the GC implementation in C++. The developers have to deal with C++ data structures implementing Java objects; they have to be aware of the objects' meta data, memory layout, and how references are implemented.

Furthermore, various low-level invariants must be preserved in the virtual machine. Since C++ code can access hidden data such as meta data and accidentally break a memory layout, the developers have to carefully implement GC customization to satisfy the low-level invariants. Note that it is not a problem that the implementation language is C++. Even if the Java virtual machine is implemented in Java as the Jikes RVM [1] and the Maxine VM [55] are, those problems will be observed.

Computational self-reflection is a promising approach to customize garbage collectors. Sometimes a program controls something that is not a data structure in the programming language: file, interprocess connection, or that program itself. Computational self-reflection is a design pattern for interface of such out-of-language data. Computational self-reflection allows to operate out-of-language target as if it was a normal data structure by reflecting operations to the real target.

```
1 (define intern-table
2   (make-hashtable string-hash
3                   string=?))
4
5 (define (intern str)
6   (if (hashtable-contains?
7         intern-table str)
8     (hashtable-ref intern-table str)
9     (begin (hashtable-set! str str)
10           str)))
11
12 (define (callback-function obj)
13   (if (string? obj) (intern obj) obj))
14
15 (register-on-copy callback-function)
```

Figure 5.1: String deduplication implemented with reflection

For example, *copy-time callback* is a simple computational self-reflection. *Copy-time callback* allows to register a callback function which is called each time the collector copies a live object. Garbage collectors often copy a live object to move it during compaction phase. In *copy-time callback*, live objects are passed to registered callback function before it is copied. The callback function investigates the received object and returns either the received object itself or another one. If the callback function returns the same object as the received one, the garbage collector simply copies it for heap compaction. If it otherwise returns a different object, the collector copies and uses that object as a substitute for the object originally received. All the references to the original object are updated to refer to the returned object. Since a callback function is a normal function, it receives a normal object and can return another normal object. It never accesses the low-level C++ data structure implementing those objects.

Although *copy-time callback* is simple, it is enough able to implement dynamic object evolution [8, 56] or heap analysis. For example, Listing 5.1 shows an example implementation of string deduplication by using *copy-time callback* in a Scheme-like language. It is a normal program written in Scheme. In line 17-18, a callback function `string-deduplication-callback` is registered by a special form `register-on-copy`. `string-deduplication-callback` is a normal scheme function defined in line 12-15. `string-deduplication-callback` just passes the argument to the `intern` function if the argument is a string object. `intern` is also a scheme function defined in line 5-10 that returns the representational string object memorized in a hashtable `intern-table` defined in line 1-3. Since `intern` is a scheme function, `intern-table` is also a normal global variable. The callback function is invoked when the garbage collector of the Scheme inter-

preter moves a Scheme object for compaction. Thus, when a string object is moved and another object representing the same character string has been already moved, that object is deleted and all the references to that object are modified to point to the equivalent object that was already moved.

`register-on-copy` is a special form for reflection. To implement this special form, we need to address garbage collection for the objects created by the copy-time callback function registered through `register-on-copy`. Since the callback function is a normal Scheme function, it will create objects during the execution. However, it concurrently runs as part of the garbage collector while the garbage collector is running. Depending on the result of the callback function, the collector needs to substitute a new object for the object being currently moved, and appropriately update the references to that object. Furthermore, the heap size that the callback function can use is limited since the garbage collector starts running when the size of the remaining heap reaches below a certain threshold. We need a sophisticated concurrent garbage collector that can run with a small amount of heap memory.

One possible approach is to allocate some special heap memory that only a copy-time callback function can use. Then we can separately collect garbage in that heap memory by a dedicated collector. However, this approach does not satisfy our motivating requirement. The garbage collector customized by the callback function never collects the objects created by the callback function itself. A different, uncustomized collector collects them. For example, when the callback function `string-deduplication-callback` is modified to create string objects for returning, these string object are never inspected for string deduplication. They are not passed to `string-deduplication-callback` when they are later collected by a garbage collector c_1. To avoid this, we might have to register another callback function for the garbage collector c_2 managing the heap memory where `string-deduplication-callback` creates objects. That callback function, however, needs a different special heap memory where it creates objects, which are not managed by c_2 but another collector c_3. Thus, this approach causes infinite regression.

Another approach would be to let a copy-time callback function allocate objects in the regular heap memory where the garbage collector is concurrently collecting the garbage. It seems promising but its heap memory consumption would be a problem. Since our `string-deduplication-callback` is invoked for every live object during garbage collection, it might rapidly consume the remaining heap memory and then require to collect the objects that were created by `string-deduplication-callback` but are already dead. The feasibility of collecting these dead objects, for example by an existing concurrent collector, is not clear since the collection recursively invokes `string-deduplication-callback`, which will create objects.

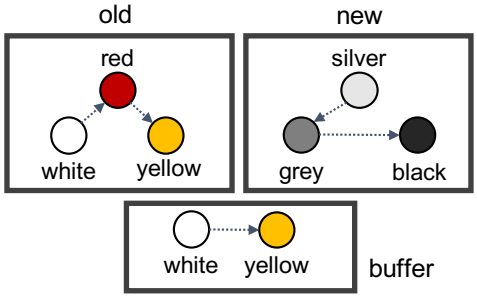


Figure 5.2: The colors of objects and the three memory spaces

5.3 Buffered Garbage Collection

In this section we propose an algorithm for garbage collection, *Buffered garbage collection*, which enables computational self-reflection on garbage collection while keeping extra memory consumption within a practical amount. Buffered garbage collection is based on Cheney's copying garbage collection [6] and it enables meta programs to create objects while avoiding both infinite regression and unacceptable memory consumption.

The reflection mechanism of Buffered garbage collection is simple; it just supports the *copy-time callback*. Once a callback function is registered, the collector invokes it each time before copying a live object into new space to determine whether the object should be replaced or not. The callback function can read/write to the given copied object and the global environment. It can also create new objects.

In our algorithm, the objects created by the callback function are allocated in a dedicated small memory region, called *buffer*. The garbage in the buffer space is frequently collected by minor copying collection between the buffer space and the new space. Those objects in the buffer space are copied into the new space if they are alive when the garbage collector reaches a certain safe point, for example, when each invocation of the callback function finishes. Since all the live objects created by the callback function are moved into the new space, they are processed by the callback function at the next major collection between the new and old spaces. The next major collection will copy them as it copies other normal objects if they are alive.

5.3.1 The colors of objects

Before presenting our algorithm, we introduce several states for objects. These states are based on the tri-color marking abstraction [13], but we use more colors. The tri-color marking abstraction categorizes objects into three groups: white, black, and grey, but for Buffered garbage collection we add yellow, silver, and red (see Figure 5.2). Yellow denotes that the object is already copied into the new region, it contains a forward pointer to the

copy, and it may be destroyed. Silver denotes that the object may contain pointers to the buffer region. Red is a special color to detect cyclic object replacements and prevent infinite recursion during the GC.

Since our algorithm is an extension to copying collection, the memory space is divided into three regions: old, new, and buffer. At the beginning, objects are in the old space. They are white and their liveness is uncertain. When the garbage collector finds a live white object, it makes a copy of that object in the new space. The copy is colored grey. The grey objects become black if they do not contain any references to white or yellow objects. After making the copy, the collector modifies the copied white object to contain a forward pointer to the copy. We assign the color yellow to the object containing a forward pointer.

Some objects are created by a copy-time callback function and allocated in the buffer space. At first, these objects are white. Our algorithm performs copying collection from the buffer space to the new space as well as one from old to new. Hence a copy made in the new space may contain a reference to an object in the buffer space. Such an object is colored silver. A silver object may contain a reference to the old space.

Finally, we assign the color red to an object in the old space while it is processed by a callback function. This is for avoiding an infinite loop in our algorithm.

5.3.2 The algorithm

Buffered garbage collection splits memory space into three regions: old, new, and buffer. An object newly created is allocated in the old space. When the garbage collection starts running, the objects in the old space are moved to the new space if they are alive. The collection continues until all live objects have been moved.

First, the collector colors all objects white in the old space. Then, it makes a copy of every root object in the new space. The copy is colored grey. The references in the root set are updated to point to the copies. The original object in the old space is changed into yellow to contain a forward pointer to its copy. Before making a copy, the collector invokes a callback function. Details of this procedure named *callback-and-copy* are described in section 5.3.2.

The collector examines all grey objects to find a reference to a white or yellow object in the old space. If the object is white, a copy of the object is created in the new space by *callback-and-copy*. The resulting copy is (usually) a grey object. The reference is modified to refer to this grey object. If the reference points to a yellow object, it is modified to refer to the object that the forwarding pointer in the yellow object points to. After this examination, the object is turned into black since it does not contain a reference to a white or yellow object.

The garbage collection finishes when all the objects in the new space become black. Then the old space is cleared and the roles of the old and new spaces are swapped. The program execution is resumed.

Procedure callback-and-copy

callback-and-copy makes a copy of a live white object in the old space. The copy is stored in the new space. Then, the garbage collector invokes a callback function if it is registered. The white object being copied is passed to the function as an argument. The callback function can access any objects except yellow since yellow objects are already copied and may be destroyed. To avoid accesses to yellow, the algorithm introduces read barriers. When it creates a new object, the object is allocated in the buffer space and colored white. *callback-and-copy* finishes by returning any object except yellow ones. The returned object may be in either the old, new, or buffer space.

When the callback function finishes, the procedure named *flush-buffer* is executed (its detailed are described in section 5.3.2). It performs copying collection from the buffer space to the new space. After the execution of *flush-buffer*, there exist no references to an object in the buffer space. If the callback function returns the given white object *as is*, the collector makes a new copy of that white object. The copy is created in the new space and colored grey. If the callback function returns a white object different from the given white one, the collector recursively invokes *callback-and-copy* to make a copy of that different white object. Otherwise, if the object returned by the callback function is grey or black, the returned object is regarded as a new copy that this invocation of *callback-and-copy* is supposed to make. In either case, the collector finally gives the yellow color to the white object passed to the callback function. It modifies the white object to contain a forward pointer to the copy of that object created by *callback-and-copy*.

As shown above, *callback-and-copy* may recursively invoke itself. To avoid infinite regression, the white object being copied is changed into red before being passed to the callback function. If *callback-and-copy* is invoked later to make a copy of a red object, the collector throws an error.

Procedure flush-buffer

flush-buffer performs copying collection from the buffer space to the new space. We call this *minor collection*. The buffer space contains the objects created by the callback function. *flush-buffer* does not invoke the callback function when it copies a live object from the buffer space to the new space.

Since *flush-buffer* is invoked after the callback function finishes, *flush-buffer* does not consider stack frames as the root set. The root set for this copying collection is only the remembered set constructed by the write barriers. The reference value returned by the callback function is included in the root set. Hence, if it points to an object in the buffer space, it will be updated to a reference to an object in the new space after the minor collection.

During this minor collection, a copy of an object in the buffer space is made in the new space. The copy is at first colored silver. The minor collector modifies a reference in the silver object only if the reference points to an object in the buffer space. A copy of this

object is made in the new space and the reference is updated to point to that copy. This modification is repeated until all the references into the buffer space are updated. A silver object containing no reference into the buffer space is changed into a grey object, which may contain a reference into the old space.

5.4 Comparison

Although there are several approaches to customizable garbage collectors, we designed a copy-time callback function and Buffered garbage collection to satisfy our requirements presented in Section 5.2. The first requirement is the use of reflection because of its ease of customization. The other is to avoid large memory overheads due to the customizability. In this section, we compare Buffered garbage collection to other approaches, which include ones we have briefly presented in Section 5.2.

5.4.1 Dynamically linked library

One of the simplest approaches to implementing a garbage collector with a copy-time callback function is to have the interpreter dynamically link a callback function written in C++ (for the clarity of the presentation, we assume that we are implementing a Scheme interpreter in C++).

Most C++ execution environments support dynamically linked libraries. We can build a library module containing a callback function and load it on demand to be linked with the garbage collector of the interpreter. The callback function can flexibly customize the garbage collector since both the callback function and the collector are written in C++.

A problem of this approach is that a callback function has to be written at the level of abstraction of the C++ implementation of the garbage collector. Figure 5.3 shows an example of the callback function implementing string deduplication in this approach. It is equivalent to Figure 5.1, which our approach enables. `SchemeObject` (or `SO` in short) and `SchemeObjectRef` (or `SORef` in short) are utility data types provided by the interpreter implementation for the developers of callback functions. `SO` is a C++ object implementing a Scheme object. `SORef` is also a C++ object but implementing a reference value in Scheme. It is a smart pointer that points to an `SO` and encapsulates the maintenance of the root set for garbage collection. Without these utility data types, implementing a callback function would be extremely error-prone for developers who do not know the detailed implementation of the interpreter. For example, developers need careful attention for appropriately removing reference values from the root set when an exception is thrown.

The main part of the callback function is `onCopy` in lines 27–34. Note that it inspects the meta data of `obj` to determine whether it is a string object or not. Although line 28 inspects the meta data of a given reference value, line 29 inspects the meta data of the Scheme object that the reference value points to. The developers have to be aware of these details.

`onCopy` calls `intern`, which returns a canonical representation for the given string object. It looks into the hash table `intern_table`. This hash table is declared in lines 10–17. `schemeHash` and `schemeEqual` are provided by the interpreter implementation. The former computes a hash value of the string object and the latter compares two string objects. These string objects are not ones in C++ but the C++ objects implementing Scheme's string objects. This *meta* perception often causes errors.

5.4.2 Single language

Another approach is to implement an interpreter in the same language that the interpreter interprets. If we are implementing a Scheme interpreter, the interpreter is implemented in Scheme. Implementing such an interpreter is feasible; we can run a Scheme interpreter written in Scheme on top of the Scheme interpreter written in C++.

Developers can now write a copy-time callback function in Scheme since the interpreter is written in Scheme. Although Scheme provides a much higher-level programming abstraction than C++, they still encounter the problem of the low-level abstraction used by the interpreter implementation. If the interpreter directly exposes the implementations of the garbage collector to a callback function, the developers have to write a Scheme program similar to the C++ program in the previous approach. A callback function would have to process a given object from the perspective of the interpreter implementation. The programming might be confusing and worse than the previous approach since the developers have to distinguish Scheme objects processed by the garbage collector from Scheme objects used in the callback function.

Figure 5.4 shows a callback function written in this approach for string deduplication. We can observe a one-to-one similarity between Figure 5.3 and Figure 5.4 while Figure 5.4 has to deal with lower-level abstractions than Figure 5.1 that our approach enables. The callback function `on-copy` in Figure 5.4 uses `ref-type` and `type-of` in lines 21 and 23 for accessing the meta data of `obj`. The developers cannot use the standard function `string?` to determine whether `obj` is a string object or not. If we call `string?` with an object implementing a string object in the interpreted Scheme, `string?` will return `false`. A similar problem is seen for the hash table.

This single-language approach was adopted by a more practical system, Jikes Research Virtual Machine (Jikes RVM) [1]. It is an implementation of the Java virtual machine written in Java. Jikes RVM provides the Memory Management Toolkit (MMTk) [3] for implementing a new garbage collector as easily as our copy-time callback function. However, implementing a new garbage collector with MMTk for Jikes RVM causes the problem mentioned above.

5.4.3 Other garbage collectors

As we have shown in Section 5.2, a copy-time callback function will need a large amount of extra memory space if the objects created by the callback function are allocated directly in the new space, which live objects are being moved into. According to our requirement, the callback function is invoked for every live object in the old space and thus the total amount of objects created by the callback function is proportional to the number of the live objects. In some interpreter implementations, a callback function may implicitly create an object for its stack frame and so on. If so, collecting every small live object consumes a bigger memory block than the object's.

Our Buffered garbage collection first allocates those objects in the buffer space and then moves only live objects to the new space. The move from the buffer space to the new space is frequently performed every time when the callback function finishes. We can expect that our collector consumes a smaller amount of memory.

A key idea of our approach is to reclaim garbage objects created by a callback function while the garbage collector is still running. Hence, a concurrent garbage collector [2] might seem to be able to reclaim the garbage objects created by the callback function if they are allocated in the old space. An issue with this approach however is whether garbage is reclaimed faster than it is produced. Recall that the callback function may create several larger objects when it collects one live object. We will also need to study the execution of the callback function during the second pause time (or the remark phase) of the concurrent collection.

Buffered garbage collection can be regarded as a variant of the generational collection [42]. Like the generational collection, the minor collection from the buffer space to the new space exploits the fact that most objects created by the callback function are garbage when the function finishes. Buffered garbage collection uses the buffer space to identify such short-lived objects. On the other hand, the generational collection exploits the fact that most of the recently created objects are short-lived. It does not provide multiple regions where objects are initially allocated. All objects are initially allocated in the young space and they are equally treated.

We see a similarity to the regional garbage collection [11, 35] in the fact that objects are initially allocated in two regions, the old space or the buffer space, and that they are separately scavenged. Although our aim is not to reduce the pause time related to garbage collection, it would be possible to emulate our algorithm by customizing a regional collector. The customized collector would use one region as the buffer space and, at the first time, scavenge that region without invoking a copy-time callback function. Then that region would be changed into part of the normal space where the callback function does not create objects. When that region is scavenged next time, the collector invokes the callback function. The collector uses another fresh region as the buffer space where the invoked callback creates objects.

5.5 Experiment

To evaluate Buffered garbage collection, we implemented an interpreter for a simple Scheme-like language in C++. The interpreter supports not only the Buffered garbage collector but also the normal copying collector, where the objects created by a copy-time callback function are allocated directly in the new space. The garbage collection is initiated when the memory consumption exceeds the preset threshold. In this Scheme-like language, a symbol value is not unique (the same-looking symbols may not be identical), numeric values are not unboxed, and every stack frame is allocated as an object in the heap memory.

We ran a micro benchmark program on this interpreter and examined whether the Buffered garbage collector could run with a smaller amount of memory than the normal copying collector when a copy-time callback function was registered. The benchmark program counted bi-gram frequencies in a long character string. This character string contained 10240 letters randomly selected among four letters: `a`, `b`, `c`, and `d`. The copy-time callback function implemented string deduplication and its code was shown in Figure 5.1. The interpreter was run on a machine with the Intel® Core i7-4770S processor (eight 3.10 GHz cores) and 16 GB of memory. Its operating system was Ubuntu16.04 LTS. We used GNU gcc 5.4.0 for compiling the interpreter.

Figure 5.5 shows the results of the experiments with the buffered garbage collector and the normal copying collector, respectively. Each column represents the total heap size excluding the buffer space. The rows in the upper chart of Figure 5.5 represents the size of the buffer space in Bytes (not KBytes). The rows in the lower chart of Figure 5.5 represents the threshold when the garbage collection is initiated. Since the interpreter needs an extra margin in the heap memory to run a copy-time callback function with the normal collector, we examined different thresholds. For the buffered collector, we chose 100% for the threshold. The color of each cell shows the result of executing our micro-benchmark program. Black cells denote a heap memory shortage and the failure of execution. White cells denote that no heap memory shortage or garbage collection happened. Gray cells denote a successful execution with garbage collection and string deduplication.

The results of our experiments reveal that our collector could run the micro benchmark with a smaller amount of heap memory than the normal copying collector. When more than 2 KB was given to the buffer space, the Buffered garbage collector could run the program with only 4 MB of heap memory (the total heap size including the buffer space was 8194 KB). The normal copying collector could not run the program with this amount of heap memory; it needed at least 12 MB. Note that the interpreter with this size of heap memory did not have to perform garbage collection. The garbage collection was initiated when the threshold was set from 20 to 40%. Under this configuration, however, the normal copying collector required more than 16 MB of heap memory.

5.6 Concluding Remarks

This chapter proposed Buffered garbage collection, which allows us to customize a garbage collector via computational self-reflection. The experiment showed that the buffered garbage collector could run our benchmark program without consuming unacceptable huge memory. A limitation is that our garbage collection is based on copying algorithm. Therefore, the collector stops the world during garbage collection and only the half of an available memory space is used. To avoid these problems, applying our idea to regional collectors is a future work.

```
#include <unordered_map>
#include <functional>
#include "SchemeObject.hpp"
#include "SchemeHash.hpp"
using namespace std;

typedef SchemeObject    SO;
typedef SchemeObjectRef SORef;

unordered_map<
  SORef,
  SORef,
  function<size_t(SORef)>,
  function<bool(SORef, SORef)>
> intern_table(0,
               schemeHash,
               schemeEquals);

SORef intern(SORef str){
  if(!map_contains(intern_table, str)){
    return intern_table[str] = str;
  }else{
    return intern_table[str];
  }
}

SORef onCopy(SORef obj){
  if(obj.ref_type == SORef::OBJECT &&
     obj->type    == SO::STRING){
    return intern(obj);
  }else{
    return obj;
  }
}
```

Figure 5.3: String deduplication by a dynamically-linked callback function

```
(#%module-begin
  (library gc-customization
    (export on-copy)
    (import scheme-interpreter))

(define intern-table
  (make-hashtable scheme-hash
                  scheme-equal?))

(define intern
  (lambda (str)
    (if (hashtable-contains?
          intern-table str)
      (hashtable-ref intern-table str)
      (begin (hashtable-set!
               intern-table str str)
             str))))

(define on-copy
  (lambda (obj)
    (if (and (eq? (ref-type obj)
                  'OBJECT)
             (eq? (type-of  obj)
                  'STRING))
      (inetrn obj)
      obj))))
```

Figure 5.4: String deduplication by a callback function written in Scheme

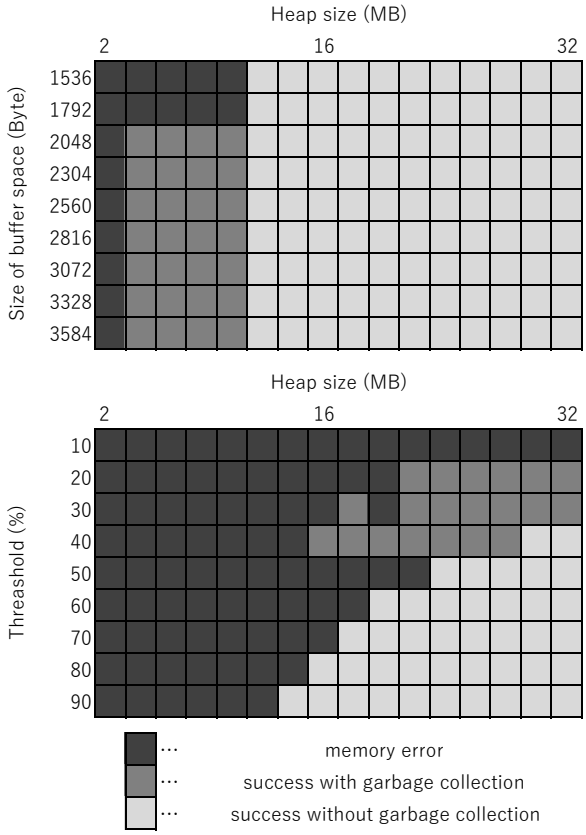


Figure 5.5: Bi-gram counting with the buffered garbage collector (top) and with the normal copying garbage collector (bottom)

Chapter 6

Conclusion

In this thesis, we presented our researches to assist library developments. People will develop countless applications for our lives in the future, and the number of problem domains must increase. Developing libraries is a promising approach to reduce the cost of developing applications. However a specific library can reduce only the cost of a limited range of application developments. To reduce the cost of a wide range of application developments, we have to develop libraries for each problem domain. We aim to reduce the cost of a wide range of application developments by assisting library developments.

We focused on two approaches to assist in library developments and proposed three methods. The first approach we focused on was protocol checking, and we proposed a novel protocol checking method for fluent interfaces. Our method makes a type checker emulate an LR parser to detect illegal method chains. We revealed that our method would consume quadratic time to check a program by our experiment. The result suggests that it is possible to improve a type checker for other uses from type checking since an LR parsing can be done in linear time even if it is by emulation.

One limitation of our work is that our method can check a library protocol only if it is a fluent interface. A method to check a protocol of another type of library interface is left for future work. Also, we have not discovered how we can measure how much a method reduces the cost to develop libraries. It is not easy to evaluate how much a method reduces the cost of library developments since it is not enough to measure how much it costs to develop a library, but it is also necessary to measure the quality of that library. Measuring how much it costs to develop applications using that library possibly implies the library quality. Considerations of how we can estimate the impact of a method on the cost to develop libraries are also left for future work.

The second approach to assist library developments we focused on was FFIs which enables us to reuse libraries written in another programming language, and we proposed two methods. The first method was a garbage collection algorithm that can detect and collect cross-language cyclic references when it is garbage. We experimented on its computation

time overhead and revealed the relationship between shapes of cycles and computation time. The second method was also a garbage collection algorithm, but for a garbage collector, which allows self-reflective customization. Our experiment confirmed that our algorithm could reduce memory consumption caused by self-reflective customization.

The first method can stop a user program for a long time since it is a stop-the-world algorithm. Techniques in incremental or concurrent garbage collection might reduce the pause time. It is also unclear how much preventing "the last piece of cake" problem reduces the cost of library developments. Discovering a method to measure how a method reduces the cost to develop libraries is left for future work. The self-reflective interface to customize a garbage collector the second method focusing on is still simple and cannot implement the first method. Researches on a more expressive interface and measures against problems that interface can cause are left for future work.

Each of our proposals is still a small progress, and they cannot contribute our lives for now. However, we have to do trial and error on methods to assist in library developments since evaluating them takes years, and we cannot efficiently implement only the technologies we needed. We believe that accumulating little progress will enrich our lives in the future.

Bibliography

[1] Bowen Alpern et al. "Implementing JalapeÑO in Java". In: *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*. ACM, 1999, pp. 314–324.

[2] Henry G. Baker Jr. "List Processing in Real Time on a Serial Computer". In: *Commun. ACM* 21.4 (Apr. 1978), pp. 280–294. ISSN: 0001-0782. DOI: 10.1145/359460.359470. URL: http://doi.acm.org/10.1145/359460.359470.

[3] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. "Oil and Water? High Performance Garbage Collection in Java with MMTk". In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 137–146. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.999420.

[4] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: 10.1145/362686.362692. URL: %5Curl%7Bhttps://doi.org/10.1145/362686.362692%7D.

[5] Roland Bock. *rbock/sqlpp11: A type safe SQL template library for C++*. https://github.com/rbock/sqlpp11. 2016.

[6] C. J. Cheney. "A Nonrecursive List Compacting Algorithm". In: *Commun. ACM* 13.11 (Nov. 1970), pp. 677–678. ISSN: 0001-0782. DOI: 10.1145/362790.362798. URL: http://doi.acm.org/10.1145/362790.362798.

[7] John Cocke. *Programming Languages and Their Compilers: Preliminary Notes*. 1969.

[8] Tal Cohen and Joseph (Yossi) Gil. "Three Approaches to Object Evolution". In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. PPPJ '09. Calgary, Alberta, Canada: ACM, 2009, pp. 57–66. ISBN: 978-1-60558-598-7. DOI: 10.1145/1596655.1596665. URL: http://doi.acm.org/10.1145/1596655.1596665.

[9] Bruno Courcelle. "On Jump-Deterministic Pushdown Automata". In: *Mathematical systems theory* (1977).

[10] Ulan Degenbaev et al. “Cross-Component Garbage Collection”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018). DOI: 10.1145/3276521. URL: %5Curl%7Bhttps://doi.org/10.1145/3276521%7D.

[11] David Detlefs et al. “Garbage-first Garbage Collection”. In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM ’04. Vancouver, BC, Canada: ACM, 2004, pp. 37–48. ISBN: 1-58113-945-4. DOI: 10.1145/1029873.1029879. URL: http://doi.acm.org/10.1145/1029873.1029879.

[12] Werner Dietl et al. “Building and Using Pluggable Type-checkers”. In: *Proceedings of the 33rd International Conference on Software Engineering*. ICSE ’11. Waikiki, Honolulu, HI, USA: ACM, 2011, pp. 681–690. ISBN: 978-1-4503-0445-0. DOI: 10.1145/1985793.1985889. URL: %5Curl%7Bhttp://doi.acm.org/10.1145/1985793.1985889%7D.

[13] Edsger W. Dijkstra et al. “On-the-fly Garbage Collection: An Exercise in Cooperation”. In: *Commun. ACM* 21.11 (Nov. 1978), pp. 966–975. ISSN: 0001-0782. DOI: 10.1145/359642.359655. URL: http://doi.acm.org/10.1145/359642.359655.

[14] Robert R. Fenichel and Jerome C. Yochelson. “A LISP Garbage-collector for Virtual-memory Computer Systems”. In: *Commun. ACM* 12.11 (Nov. 1969), pp. 611–612. ISSN: 0001-0782. DOI: 10.1145/363269.363280. URL: http://doi.acm.org/10.1145/363269.363280.

[15] Apache Software Foundation. *Apache Phoenix*. https://phoenix.apache.org/. 2014.

[16] Python Software Foundation. *Python Documentation*. https://docs.python.org/ja/3/library/ctypes.html.

[17] Martin Fowler. *FluentInterface*. https://www.martinfowler.com/bliki/FluentInterface.html. 2005.

[18] Emden R. Gansner and Stephen C. North. “An open graph visualization system and its applications to software engineering”. In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[19] 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.

[20] Joseph (Yossi) Gil and Keren Lenz. “Simple and safe SQL queries with C++ templates”. In: *Science of Computer Programming* 75.7 (2010). Generative Programming and Component Engineering (GPCE 2007), pp. 573–595.

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

[22] Yossi Gil and Ori Roth. "Fling - A Fluent API Generator". In: *33rd European Conference on Object-Oriented Programming (ECOOP 2019)*. 2019. DOI: 10.4230/LIPIcs.ECOOP.2019.13. URL: %5Curl%7Bhttp://drops.dagstuhl.de/opus/volltexte/2019/10805%7D.

[23] Jonathan Goldstine, John K. Price, and Detlef Wotschke. "On reducing the number of states in a PDA". In: *Mathematical systems theory* 15.1 (Dec. 1981), pp. 315–321. ISSN: 1433-0490. DOI: 10.1007/BF01786988. URL: %5Curl%7B%22https://doi.org/10.1007/BF01786988%22%7D.

[24] *Google Code Archive - pyv8*. https://code.google.com/archive/p/pyv8/.

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

[26] Stephan Heilbrunner. "A parsing automata approach to LR theory". In: *Theoretical Computer Science* 15.2 (1981), pp. 117–157. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(81)90067-0. URL: %5Curl%7B%22http://www.sciencedirect.com/science/article/pii/0304397581900670%22%7D.

[27] Kohei Honda. "Types for dyadic interaction". In: *CONCUR'93*. Ed. by Eike Best. Berlin, Heidelberg: Springer Berlin Heidelberg, 1993, pp. 509–523. ISBN: 978-3-540-47968-0.

[28] Michihiro Horie et al. "String Deduplication for Java-based Middleware in Virtualized Environments". In: *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. VEE '14. Salt Lake City, Utah, USA: ACM, 2014, pp. 177–188. ISBN: 978-1-4503-2764-0. DOI: 10.1145/2576195.2576210. URL: http://doi.acm.org/10.1145/2576195.2576210.

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

[30] Kazuhiro Ichikawa and Shigeru Chiba. "User-Defined Operators Including Name Binding for New Language Constructs". In: *The Art, Science, and Engineering of Programming* 1.2 (2017), 15:1–15:25.

[31] Google Inc. *Improve your code with lint checks*. https://developer.android.com/studio/write/lint. 2011.

[32] Steven G. Johonson. *Calling Python functions from the Julia language*. https://github.com/JuliaPy/PyCall.jl.

[33] Maria Jump and Kathryn S. McKinley. "Cork: Dynamic Memory Leak Detection for Garbage-collected Languages". In: *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '07. Nice, France: ACM, 2007, pp. 31–38. ISBN: 1-59593-575-4. DOI: 10.1145/1190216.1190224. URL: http://doi.acm.org/10.1145/1190216.1190224.

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

[35] Felix S. Klock II and William D. Clinger. "Bounded-latency Regional Garbage Collection". In: *Proceedings of the 7th Symposium on Dynamic Languages*. DLS '11. Portland, Oregon, USA: ACM, 2011, pp. 73–84. ISBN: 978-1-4503-0939-4. DOI: 10.1145/2047849.2047859. URL: http://doi.acm.org/10.1145/2047849.2047859.

[36] Donald E. Knuth. "On the translation of languages from left to right". In: *Information and Control* 8.6 (1965), pp. 607–639. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(65)90426-2. URL: %5Curl%7B%22http://www.sciencedirect.com/science/article/pii/S0019995865904262%22%7D.

[37] Dimitrios Kouzapas et al. "Typechecking protocols with Mungo and StMungo: A session type toolchain for Java". In: *Science of Computer Programming* 155 (2018), pp. 52–75. DOI: https://doi.org/10.1016/j.scico.2017.10.006. URL: http://www.sciencedirect.com/science/article/pii/S0167642317302186.

[38] Stefan Krüger et al. "CrySL: An Extensible Approach to Validating the Correct Usage of Cryptographic APIs". In: *European Conference on Object-Oriented Programming (ECOOP)*. July 2018, 10:1–10:27.

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

[40] Du Li and Witawas Srisa-an. "Quarantine: A Framework to Mitigate Memory Errors in JNI Applications". In: *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*. PPPJ '11. Kongens Lyngby, Denmark: ACM, 2011, pp. 1–10. ISBN: 978-1-4503-0935-6. DOI: 10.1145/2093157.2093159. URL: http://doi.acm.org/10.1145/2093157.2093159.

[41] *libffi*. https://sourceware.org/libffi/.

[42] Henry Lieberman and Carl Hewitt. "A Real-time Garbage Collector Based on the Lifetimes of Objects". In: *Commun. ACM* 26.6 (June 1983), pp. 419–429. ISSN: 0001-0782. DOI: 10.1145/358141.358147. URL: http://doi.acm.org/10.1145/358141.358147.

[43] Harry G. Mairson. "Generating Words in a Context-free Language Uniformly at Random". In: *Inf. Process. Lett.* (1994). DOI: 10.1016/0020-0190(94)90033-7. URL: http://dx.doi.org/10.1016/0020-0190(94)90033-7.

[44] George Marsaglia. "Xorshift RNGs". In: *Journal of Statistical Software, Articles* 8.14 (2003), pp. 1–6. ISSN: 1548-7660. DOI: 10.18637/jss.v008.i14. URL: %5Curl%7Bhttps://www.jstatsoft.org/v008/i14%7D.

[45] 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.

[46] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: A Formal Introduction*. 1992. ISBN: 0-471-92980-8.

[47] 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*. MASPEGHI '13. Montpellier, France: ACM, 2013, pp. 9–16.

[48] Oracle. *Java Native Interface Specification⊢Contents*. `https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html`.

[49] David Plainfossé and Marc Shapiro. "A Survey of Distributed Garbage Collection Techniques". In: *Proceedings of the International Workshop on Memory Management*. IWMM '95. Berlin, Heidelberg: Springer-Verlag, 1995, pp. 211–249. ISBN: 3540603689.

[50] Brian Cantwell Smith. "Reflection and Semantics in LISP". In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '84. Salt Lake City, Utah, USA: ACM, 1984, pp. 23–35. ISBN: 0-89791-125-3. DOI: `10.1145/800017.800513`. URL: `http://doi.acm.org/10.1145/800017.800513`.

[51] 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*. OOPSLA '11. Portland, Oregon, USA: ACM, 2011, pp. 713–732. ISBN: 978-1-4503-0940-0. DOI: `10.1145/2048066.2048122`. URL: `http://doi.acm.org/10.1145/2048066.2048122`.

[52] Peter Thiemann and Vasco T. Vasconcelos. "Context-free Session Types". In: *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*. ICFP 2016. Nara, Japan: ACM, 2016, pp. 462–475. ISBN: 978-1-4503-4219-3. DOI: `10.1145/2951913.2951926`. URL: `%5Curl%7Bhttp://doi.acm.org/10.1145/2951913.2951926%7D`.

[53] Todd Veldhuizen. *C++ Templates are Turing Complete*. Tech. rep. Indiana University Computer Science, 2003.

[54] S. C. Vestal. "Garbage Collection: An Exercise in Distributed, Fault-Tolerant Programming". PhD thesis. USA, 1987.

[55] Christian Wimmer et al. "Maxine: An Approachable Virtual Machine for, and in, Java". In: *ACM Trans. Archit. Code Optim.* 9.4 (2013), 30:1–30:24.

[56] Thomas Würthinger, Christian Wimmer, and Lukas Stadler. "Dynamic Code Evolution for Java". In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. PPPJ '10. Vienna, Austria: ACM, 2010, pp. 10–19. ISBN: 978-1-4503-0269-2. DOI: `10.1145/1852761.1852764`. URL: `http://doi.acm.org/10.1145/1852761.1852764`.

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

[58] Tetsuro Yamazaki and Shigeru Chiba. "Buffered Garbage Collection for Self-reflective Customization". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: ACM, 2018, pp. 1256–1259. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167416. URL: http://doi.acm.org/10.1145/3167132.3167416.

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