Spring 5-13-2015

Operational Semantics for Featherweight Lua

Hanshu LIN
San Jose State University

Operational Semantics for Featherweight Lua

A Project

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Hanshu Lin

May 2015

The Designated Project Committee Approves the Project Titled

Operational Semantics for Featherweight Lua

by

Hanshu Lin

APPROVED FOR THE DEPARTMENTS OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2015

Thomas H. Austin	Department of Computer Science
Ronald Mak	Department of Computer Science
Chris Pollett	Department of Computer Science

ABSTRACT

Operational Semantics for Featherweight Lua

by Hanshu Lin

Lua is a small, embedded language to provide scripting in other languages. Despite a clean, minimal syntax, it is still too complex for formal reasoning because of some syntactic sugar or specific syntax structures in Lua.

This thesis develops *Featherweight Lua* (FWLua), following the tradition of languages like Featherweight Java[1] and Featherweight JavaScript[2]. The goal is to develop a core of language features that, while remaining simple enough for formal reasoning, also remain faithful to the central characteristics of the language. Specifically for Lua, the core features that are essential for our modeling include:

- First-class functions
- Tables as the central data construct
- *Metatables* that provide various "hooks" to change the behavior of tables

To further validate this approach, we show how an extensive set of features from the full Lua programming language can be reduced to FWLua. Finally, we include a reference implementation written in Haskell as a tool for further testing and experimenting with the language. With this research, we provide a basis for future research into the Lua programming language.

TABLE OF CONTENTS

CHAPTER

APPENDIX

LIST OF FIGURES

CHAPTER 1

Introduction of Lua

Lua is a scripting programming language. It is designed to support general procedural programming with data description facilities [3]. Since it is designed to be an embedded scripting language, it has a concise syntax. However, Lua is powerful and fast, making it useful as a general purpose scripting language. Our core goal is to develop a set of formal semantics for a minimal version of Lua. This set of semantics allows us to formally reason about properties of the Lua language without becoming needlessly mired in unimportant aspects of the full Lua language. In other words, we are attempting to remove those syntactic features we think are less important and thus keep the core features of Lua, facilitating future research into the language.

In our research on Lua [4], we have identified several features as central to Lua's character. First of all, Lua supports first-class functions. It provides syntax for functions to fulfill specific tasks from users. In addition, Lua's design is centered on tables. Tables are the primary data structure in Lua since they are lightweight, powerful, and can present many other data structures when combined with functions. Furthermore, Lua has a special feature named "metatables", which are reserved tables with a set of functions to control the behaviors of tables in Lua. All of these features make Lua different from other languages, and that is the reason why we identify them as the core features of Lua.

In this paper, we first introduce the background of Lua in this chapter. We will also introduce the syntax of Lua, and whose features we think are very important to Lua in Chapter 2. Then, we will discuss about the syntax and operational semantics

of Featherweight Lua – an exquisite language we create for further research of Lua – in Chapter 3 and Chapter 4. What is more, in case of leaning Lua, we show how we translate Lua to Featherweight Lua in Chapter 5. Finally, we will give a brief introduction about our implementation in Chapter 6, and discuss about conclusion and future work in Chapter 7.

1.1 History of Lua

Before talking about Lua, we give a brief introduction about the history of Lua to elaborate on the reason why we think Lua is interesting and valuable to research.

At the beginning of the history, there was a programming language called "Sol", which was created for fulfilling some mundane tasks in a Brazilian oil company called PETROBRAS. In 1993, there were already some lightweight languages, but none of them provided data description; variables in these languages were unable to define a data of key-value pair, such as tables or records. Therefore, Lua was created, without the need of type declarations, to satisfy this need at the time [5]. With more and more users leveraging Lua in their work, the demand for better performance increased. Lua version 2 was released in February 1995. It brought many important changes such as the simplification of the semantics for table constructors, the concept of "fallbacks"—a set of programmer defined functions that are called whenever Lua does not know what to do next, and some other important features. Version 3 of Lua was published in July 1997. In this version, the concept of fallbacks was replaced with tag methods. Because of this, developers using Lua 3 can now create their own tags, and even associate tables and userdata to the tags to make them have their own unique behaviors. Then, version 4 of Lua was released in November 2000. Besides a set of new APIs, the `for` loop statement was introduced in Lua. Also, in February

2002, Lua 4.1 introduced `metatables` and `metamethods` as the replacement for tag methods and fallbacks [6].

1.2 Where to use Lua

Lua is often embedded in many applications to fulfill some simple and small tasks due to its qualities. Usually, Lua is used to add extensions in large applications that would be hard to change. Also, Lua can be used in many programs in different domains that need extensions to make them flexible, especially for game developers and UI designers. [7].

For instance, Lua is used to create user interface add-ons in the popular online game called The World of Warcraft [8]. These add-ons are commonly used for assisting players by adding some simple functionality in the game, such as timers, alerts, chat reminders, action automations, dashboards and so on. Based on the add-ons website Curse [9], there are more than 4600 different add-ons used in the game.

In the aspect of commodities, Lua has been embedded in many devices, including the CHDK Lua interface for Canon cameras [10], UI designs for Photoshop Lightroom [11], and Logitech keyboards. Lua is also used for network security applications in Cisco [12] and for extensions and add-ons in the MySQL Workbench. CloudFlare, a content delivery network using Nginx, is trying further improve its performance using Lua because of the benefits of Lua [13] and the speed of the LuaJIT engine [14].

1.3 Interesting features of Lua

Analyzing the right features is always important for learning a programming language. In this section, we briefly introduce several features we think are interesting of important in Lua. As an embedded language, there are several characters making

Lua unique. Actually these mechanisms are all indispensable in Lua, beside basic data types and data structures.

First of all, Lua supports first-class functions. An anonymous function can be directly assigned to a variable in Lua just like any other value.

Also, metatables are used in Lua to control the behavior of tables. There are several reserved functions in a metatable, which we can call "Metamethods". Metatables delivers a fast and clean programming method by defining behaviors in tables and setting hooks for common operators, such as "+" or "∗".

In this paper, we keep these features and regard them as part of the core element. We will specifically discuss them in each chapter.

1.4 Lua and JavaScript

Lua is similar in its design to the JavaScript programming language. Since JavaScript is widely used in client-side scripting, it has received significantly more attention than Lua, but many of the challenges are very similar.

One of the inspirations for this work was Featherweight JavaScript (FWJS) [2], which also develops a core subset for a much larger language. JavaScript is similar to Lua in that it also has first-class functions, but it differs in that it has a prototype-based object system. Lua's combination of tables and metatables provide similar functionality, but with a noticeably different feel.

Guha et al. [16] show an alternate approach for reducing JavaScript to a core calculus. The authors also give several models, desugaring processes, a small-step operational semantics, and an implementation.

1.5 Operational semantics

Operational semantics [17] are a set of rules defining the behavior of a language. These rules are useful for verifying certain important properties for a programming language, such as security. Operational semantics can be classified into two different kinds: *small-step semantics* and *big-step semantics*. The former one describes each individual step in the computation, while the latter one shows how the final result of one statement is obtained after being fully executed. In this paper, we show the big-step semantics of Featherweight Lua according to the research on Lua. Generally speaking, writing operational semantics is more like programming in an abstract way. To help validate and experiment with our approach, we also implement an interpreter in Haskell that closely follows the semantics we have developed.

1.6 Featherweight Language

A *Featherweight* Programming Language is commonly used to research a programming language. This is a valuable method if the programming language an individual wants to research is so complex. For instance, Thomas H. Austin and et al. have pointed out that "The full JavaScript language is quite complex and difficult to reason about, and is a difficult foundation for proving formal security guarantee" [2], and they then choose to design Featherweight JavaScript and use it to research JavaScript in the aspects of security and information flows. This proves several advantages that motivate us to establish a simpler featherweight programming language in the research of Lua.

Generally speaking, featherweight languages are very simple in both syntax and semantics. However, they all have to retain the essence of the target languages in case of formal reasoning all features of original language. In the previous featherweight

programming languages, Atsushi Igarashi shows that there are some main features still remained in Featherweight Java, including classes, methods, fields, dynamic typecasts, and inheritance [18]. In addition, Jeremy Siek also has introduced Featherweight C programming language with stack-based allocation functions, pointers, and heap allocation remained [19]. All of these previous featherweight language are made for simplifying target language in case of research, and they provide complete feature modeling from target language.

CHAPTER 2

Full Version Lua

We introduce the syntax for the full version of Lua in this chapter. As mentioned previously, Lua is an extension programming language, and therefore its syntax is relatively simple. However, there are still several parts which can cause confusion in Lua.

2.1 Syntax in Lua

Lua is a small extension language [20] with many similarities to JavaScript in its structure. Every program of Lua can be thought as a `block` – a list of statements, which are executed sequentially. Return statements can only be used in the end of a block, although they are optional.

Next, we will discuss two important elements in Lua separately, which are expressions and statements. Lua splits statements and expressions. They are separated, but still partially independent. Although most programming languages allow expressions to be put in the code individually, Lua does not allow that. Expressions must be part of statements and cannot stand alone in a program. However, note that function application is considered both statements and expressions.

Figure 1 shows the complete list of Lua expressions, and Figure 2 shows the available statements for Lua. We give a brief introduction about the important parts of Lua in the next section. Those parts include expressions, statements, functions, tables, and metatables. We will talk about how the full version of Lua may be desugared to our core language in Chapter 5.

2.2 Expressions

As we mentioned above, expressions cannot be used individually. However, it still plays a key role in Lua since there are many evaluation rules for expressions. An expression evaluates to a value. We consider this as the main difference to statements, which might not produce a value.

Figure 1 shows the expressions for Lua. Values present some constant value like booleans or integers. Prefix expressions include variables, function calls, and parenthesized expressions. Since tables are the central data structure in Lua, getting variables from tables is an important aspect of the language. In Lua, variables can either be names (`x`), direct table selects (`e.x`), or computed table selects (`e[e]`)[1]. Lua supports the usual mathematical binary and unary operators. Finally, Lua includes blocks in function definitions, which are sequences of statements. However, we will give the specific definition of blocks in Figure 2, since they look more related to statements.

2.3 Statements

Figure 2 shows the syntax for statements in Lua. We can see many common types of statement that appear in lots of programming languages, such as assignment statements and function applications. One unusual aspect of Lua is that only statements may stand alone in a Lua program. Therefore, all functions are considered to be both statements and expressions.

There are 3 parts in the syntax of statements. They are statements, blocks, and functions. In the following, we will mainly introduce them.

[1]Note that direct table selects are syntactic sugar for computed table selects.

$e, f ::=$		*Expressions*
	`nil`	nil
	v	value
	p	prefix expression
	...	vararg expression
	`function` $(\overline{e_i})$ *block* `end`	lambda function definition
	$\{\overline{[v_i] = e_i}\}$	table constructor
	e *binop* e	binary operation
	unop e	unary operation
$v ::=$		*Values*
	n	number
	`true`	boolean true
	`false`	boolean false
	s	string
$p ::=$		*Prefix Expressions*
	x	variable
	$f(\overline{e_i})$	function call
	(e)	prefix expression
$x ::=$		*Variables*
	n	name
	$p[e]$	computed table select
	$p.s$	direct table select
binop $::=$	$+ \mid - \mid * \mid / \mid \hat{} \mid \% \mid .. \mid > \mid >= \mid$ $< \mid <= \mid == \mid = \mid$ `and` $\mid$ `or`	*Binary operators*
unop $::=$	$- \mid$ `not` $\mid *$	*Unary operators*
block		*Blocks*

Figure 1: Expressions in Lua

Assignment statements are used to assign a value to a specific variable in the global store. However, this syntax in Lua works a little differently than in other languages. First of all, it is not possible to chain assignments, so that `x = y = 3` is a syntax error. However, Lua allows users to assign multiple values to multiple

variables with only one statement by using expression lists on both side of '='. This looks like what we called "Destructuring Assignment" in JavaScript [21]. Section 5.2 rises an example showing this. It is very convenient and interesting, but is not that indispensable. Therefore we decide to remove it in FWLua, regard it as a syntactic sugar, and find a way to translate them.

In addition, Lua provides a bunch of statements for control flow. There is an `if` statement in the syntax that works as in most languages. In Lua, users have 3 different ways to implement and control a loop, which are `while` statements, `for` statements and `repeat` statements. Lua's `for` statement has a different but efficient syntax comparing to other common programming languages. We will talk about it in the Section 5.4. Since loops and conditional statement are both straightforward, we will not discuss them further.

As we mentioned, blocks are a sequence of statements, which may optionally include a return statement as the last statement in the block. Therefore, the return statement will be illegal outside of blocks, or before the last statement in a block,

2.4 Functions

A function is made up of a set of statements and expressions in Lua. The reason we choose to further introduce functions is that they could be used as either expressions or statements. This section discusses those differences in more detail.

Based on the research [20], we believe that there are at least two main purposes of using functions in programming languages: to produce some side effects, or to return a value after being computed. According to this, we can use a function application as a statement in the former case and regard it as an expression in the latter one. However, a given function application may serve both purposes, depending on the

s ::=		*Statements*
	;	empty statement
	$\overline{x_i} = \overline{e_j}$	assignment statement
	`local` $\overline{n_i} = \overline{e_j}$	local assignment statement
	`function` $f(\overline{n_i})$ b `end`	function definition
	$f(\overline{e_i})$	function call
	:: n ::	label
	`break`	break
	`goto` n	goto
	`do` b `end`	do
	`if` e `then` b `else` b `end`	if statement
	`while` e `do` b `end`	while statement
	`repeat` b `until` e	repeat statement
	`for` $n = e_1, e_2, e_3$ `do` b `end`	for statement
b ::=		*Blocks*
	$\overline{s_i}$ `return` $\overline{e_i}$	block statements
f ::=		*Function Names*
	n	name
	$n.f$	name2
	$f : n$	name3
n		*Names*
e		*Expressions*
x		*Variables*

Figure 2: Statements in Lua

function's definition.

Lua provides anonymous functions, which means that a function definition expression will return itself as the result. Basically, a function in this case is an expression that evaluates to a closure.

One important aspect of functions in Lua is that they are first-class values, meaning that it is possible to support functional programming. We see this behavior of Lua as one of its core characteristics.

2.5 Metatables and Metamethods

Metatables are one of the most unusual features in Lua. The functions in metatables are called "metamethods". Metamethods are automatically called when the specific condition is satisfied. They are usually called "hooks" in many programming languages. The primary goal of metatables is to control the basic behavior of tables. For instance, if we want to find a value with a key that does not exist in the table, the `__index` metamethod in the metatable of this table will automatically execute instead of reporting a crash.

There are several default functions in the metatable. They include hooks for behavior controlling (also called `Fallbacks` in the previous version of Lua), and functions for binary operations. In addition, these metamethods can be redefined by programmers. As a set of automatically executing functions, metatables can be very powerful and a nice convenience for Lua programmers.

Lua has a set of reserved functions for metatables such as `getmetatable()` and `setmetatable()`. By analyzing these, we can further understand the behavior of metatables. We use these functions for our research in the project since we think metatables plays a key role in the language design of Lua. Meanwhile, more rules about metatables can be found in the Lua Reference Manual [4].

2.6 Tables

From Figure 1, we can see that there is an expression for table constructors, which creates a new table as a value. Basically, there are many ways of telling the program how to deal with a table and also to specify the values in it. However, the purpose of the different syntax would all be the same: to store a key-value pair in the specific table.

Therefore, the basic syntax of tables is simple in Lua. Users can just assign a new empty table to a variable, assign a value to a variable in this table, or update some values in it, just like:

```
tbl = {}
tbl[1] = 2
```

Also, you can create a table by using a set of expressions like:

```
table = { x = 1; y = 2, z = 3 }
```

And this statement can be translated into a set of expressions; we will treat this as a part of translation in Section 5.3.

Tables in Lua are the primary data structure. Almost every type of data structure can be represented with tables in Lua in an efficient way, such as arrays, records or sets. One interesting thing is that arrays and lists are usually used as the basic data structure in other languages. We can easily implement these structures using Lua tables. However, we don't take this as necessary since tables are much more powerful than them.

In this section, we illustrate several examples showing how to implement other data structure with tables and metatables. These examples will also make both tables and metatables easier to be comprehended.

The first instance we want to show is about arrays. We know that an array is made up of a set of values and keys, where keys are index numbers. Therefore, giving an array using a table is not an issue, as we show in the following code:

```
array = {}
for i=1, 100 do
  array[i] = 0
end
```

In this program, the array would return the value of `Nil` instead of `0` if the index number is out of the range (1 to 100).

Objects are very important data structures in object-oriented programming languages. Lua does not provide objects by default, but it is easy to build them in Lua using a combination of tables and metatables.

Before we write the example, we first introduce some basic qualities about objects. According to Jung and Brown [26], objects and tables have a lot of similar qualities. Tables have a state, an identity that is independent of their values, and also have a specific life cycle when being created. All of these are behaviors shared with objects.

However, objects also have some unique qualities that make them different from tables such as inheritance. To properly implement this, we rely on metatables to change the behavior of tables to act like objects.

We create a table with the table constructor in the following code:

```
parent = {firstName = "Max", lastName = "Lin"}
child = {firstName = "Mary"}
```

And we now have two tables. These tables are similar to objects because they share a similar structure. Now, we use metatables to model inheritance using a prototype-based approach [27]. As we know, the child object inherits all the data from its parent. Therefore, when we miss a key from the child, we will get back the value from the parent. Here is the code:

```
setmetatable(child,{__index = parent})
print(child.lastName)
```

In the code, the method `setmetatble(t, mt)` sets the table `mt` to `t` as its metatable. In the table `mt`, the method `__index` tells the child table to index the specific table when the key is missing. Hence, we now deliver basic inheritance for our objects. In this program, the result would be the value `Lin` since the key `lastName` is available in the parent object.

CHAPTER 3

Featherweight Lua

Although Lua has a minimal syntax compared to most other programming languages, it is still too complex for formal reasoning. For instance:

```
x = 54
function foo(x)
  local x = 18
  print(x)
end;

foo(36); --what does this print?
```

In this example, it is not immediately obvious whether the local variable, the global variable, or the parameter will print.There are also some other problems like this, making Lua complex for analyzing.

Another instance will be adding information flow controls to Lua. In building these, we may think about whether our additions to the language present secrets from leaking. There might be some functions about this issue, for example, `makeSecret(e)` or something like this. While using FWLua, we only need to reason about FWlua.

Therefore, we further simplify the syntax from the full version Lua. We call the new, simplified language *Featherweight Lua* (FWLua). In FWLua, we remove unnecessary syntax and reduce the language to its essential.

In this chapter, we first introduce how we deal with several primary parts in Lua — expressions and statements. Then, we are going to talk about functions and metatables, which are both key elements of Lua, in more detail in the chapter.

Meanwhile, we give the full syntax and evaluation rules after all the elements in FWLua have been specifically introduced.

3.1 Expressions in FWLua

As mentioned before, Lua splits statements and expressions, though they may overlap due to function applications. As the consequence, we merge expressions and statements, and treat both of them as expressions in FWLua. In this section, we discuss expressions in FWLua and why we think these simplifications can remain the true spirit of Lua.

3.1.1 Lua's Expressions

We keep constant values in the language. These types include Number, String and Boolean representing values in the original syntax. There is also a special expression `nil`, similar to `null` in Java and to both `null` and `undefined` in JavaScript. For simplicity, we treat `nil` as a constant in our evaluation rules.

In the original syntax, there are several different ways for getting variables. To make it concrete, we show a program using all three ways for getting variables:

```
tbl
tbl.x
tbl["y"]
```

The first case references a variable name. The second is a direct table select, using '.'. This variable expression directly gets the value referenced by the key "x" in the specified table "tbl". The last case is an example of a computed table access. It is similar to the second case, except that Lua allows an expression to dynamically determine the key in the table.

The computed table access case can simulate the other approaches. Referring to `tbl.x` is identical to `tbl["x"]`. In the version of Lua 5.2, there is a new feature, which is using a reserved variable `_ENV` to represent the table of global environment. Therefore, instead of referring to `x`, we can use `_ENV.x`.

As a consequence, variables are now represented by a more common format: $e_1[e_2]$, which we call it table select. In this format, we only look up the variable e_2 in the table e_1 and return the value (or nil if it doesn't exist). And we can use reserved word "_ENV" to represent the global environment. Specifically,

```
tbl
```

equals to

```
_ENV["tbl"]
```

A table constructor constructs a new domain with some scoped mutable variables within. When we invoke "{}", the compiler allocates a new scope and creates a new anonymous table, similar to how `new` works in JavaScript.

Lua supports both unary operators and binary operators, though it is straightforward to treat unary operators as binary operators with two expressions e_1 and e_2 where e_1 is `nil`. As a result, we eliminate unary operators from the syntax of FWLua.

The syntax of expressions in Featherweight Lua is presented and discussed in Section 4.1

3.1.2 Lua's Statements

According to Roberto Ierusalimschy [20], statements in Lua include assignment, control structures, function calls, and variable declarations.

Our strategy for simplifying statements is to classify them into some syntactic groups, and then choose a basic syntactic format for each of them. We also remove blocks from our syntax since we can desugar sequences of statements easily, and implement block using lambda function.

We first discuss assignment statements. Every assignment statement will change the store since it assigns a value to a variable. Also, Lua is built to be a dynamic typed programming language, which means type declarations are not needed in assignment statements. Due to this factor, we can merge all types of assignment statements into one single form. According to the original syntax, there might be some differences in local assignment statements. For instance:

```
a = 3; --A single assignment statement
b, c, d = 1, 2, 3; --Destructuring assignment
e, f = 'hello', {}; --Destructuring assignment for different types
```

There is also a format of assignment that allows using a list of expressions as either variables or values to complete "Destructuring Assignment". However, we choose the basic assignment form without lists or local scope. The other forms can be produced with syntactic sugar. In Chapter 5 we review the desugaring process in more depth.

Lua also features a large number of control structures. However, these can be represented with functions, following patterns established for the lambda calculus [23]. For instance, the following are some different kinds of control structures in Lua:

```
x = 0
while x ~= 10 do --A while loop
  x = x + 1
  print(x)
end

i = 0
repeat --A repeat loop
  i = i + 1
  print(i)
until
  i == 10

if (x == 0) then
  print("HelloWorld") --An if statement
else
  print("ByeWorld")
end
```

Also, they can be represented using a set of lambda functions. We will further discuss it in the Chapter 5. Consequently, we eliminate these control statements from FWLua.

3.2 Integration of Expressions and Statements in Lua

Finally, we treat every statement as an expression for simplicity in FWLua. However, this means that FWLua is not a true subset of Lua. Consider the following expressions:

```
3; --A constant value
x; --A variable
a = b = 10; --Multiple statements
```

Lua does not allow constants or variables as stand-alone statements, and chained assignments are not supported in Lua. While these are departures from Lua, we feel that they are minor discrepancies between the languages.

Not being a proper subset of Lua may also cause a risk: the translated code in FWLua might not run in Lua. To avoid this issue, our implementation rules carefully produce code that is valid for both FWLua and Lua, though it may look strange. Also, another issue is that users basically have to understand "another programming language" when they are researching Lua using our way. Therefore, we try to make FWLua as simple as possible, in case of being valuable for research on Lua.

3.3 Functions

Functions are used to complete a set of computations and return a value. To define a function in Lua, each of the following statements can be used:

```
function foo(x) x = x + 1 return x end --normal function definition
foo = function (x) x = x + 1 return x end --assign a first-class function
```

We can invoke this function as follows:

```
foo(4);
```

In addition, functions in the full version of Lua provide multiple expressions in the function, both in defining arguments and returning values. For example:

```
function inc(x, y)
  x = x + 1;
  y = y + 1;
  return (x, y);
end
```

We can see that a function can both take more than one argument and return multiple values.

However, we want to keep the syntax for FWLua as simple as we can. Therefore, we represent multiple expressions through desugaring, which we discuss later in this chapter.

Based on the examples above, we then choose to use the syntax of anonymous functions as our primary function syntax in FWLua. Since we have already come up with the syntax of assignment statements, we will combine it with anonymous function definition to define a function with a name. Besides, we restrict functions to one argument and one return value for decreasing unnecessary complexity in the syntax.

However, we have also raised the issue against one special statement called "return statements". As we mentioned above, return statements cannot really be called statements. Although Lua allows it appear individually like statements, return statements can only appear as the last statement in a block, and that is the reason why we need to treat them as a special cases in functions. In solving this issue, we decide to make the reserved word `return` as the tail in the function and will use `return nil` instead of no return statement in the block.

As the result, the syntax of function would be like the following:

```
function x return e end
```

In the syntax, `x` represents the argument a function takes (only one allowed), and `e` represents the function body. Functions in FWLua still allow multiple expressions in the body of function through desugaring, outlined in Chapter 5.

We will talk about some key factors related to functions in the following sections, using some examples to make them easier to comprehend.

3.3.1 Scoping

Before talking about closures in Lua and FWLua, we will show some examples to help understand closures in functions.

```
x=42
function foo()
  print(x)
end
function bar()
  local x=100
  foo()
end
bar() --will print 42
```

The above example shows that Lua uses static or lexical scoping. Lexical scoping means that a stack containing variables in one's outer scope is defined once a function or a block is declared. On the other hand, with dynamic scoping, the compiler goes throughout the whole scope link, and then locates the variable that is being called. The above example is testing Lua's scoping mechanism. Once the function `foo()` is invoked in function `bar()`, Lua finds the value of variable `x` from the stack of function `foo()`, and then find its outer scope. It means the scope in Lua is static and immutable. That is the reason why we also call this "static scope". In contrast, if Lua used dynamic scoping, the above example would print 100.

3.3.2 Closures

Closures in a programming language enable functions to access variables in their outer scope. Closures is as important as scoping. It is commonly discussed together with scoping.

For a concrete understanding of closures, consider the following example:

```
function foo()
  local x = 0;
  return
    function()
      x = x + 1;
      return x;
    end;
end
bar = foo();
print(bar()) --return 1
print(bar()) --return 2
print((foo())()) --return 1
```

This shows closures in Lua. In the example, once the variable `bar` is declared, a stack storing scoping information is created. In the condition, calling `bar` and `foo()` seems the same. However, they will point to two independent stack in inner environment, just as the above instance shows. In other words, the function `foo()` returns a closure.

For setting variables, we need very exact directions of where FWLua should set values. This is a very interesting and important character in FWLua. We will discuss this issue more in Section 4.2, and then introduce our translation from Lua to FWLua in Chapter 5.

3.3.3 Recursion

Recursion means that a function can call itself in its body. Here, we give an example of a recursive function in Lua, `factorial(n)`.

```
function factorial(n)
  if n == 0 then
    return 1;
  else
    return n * factorial(n-1);
  end
end

print(factorial(5)); --results 120
```

In the example, we can see that only defining the function `factorial(n)` is not simply enough. For implementing the factorial, it must be called with different argument in the body of itself, and this would be the classic recursive function.

Lua uses a call stack to store local variables in one function, and thus implement recursive functions [26]. Theoretically, when a function itself is called in its body, the interpreter will automatically treat it as just a variable and will further finish it after the function is done. In other words, multiple calls to the same function can be active at the same time without the crashes reporting "undefined" in Lua.

This quality in Lua is very helpful for us to give the further abstract syntax in FWLua about function, since functions in FWLua would be more like lambda calculus. Next, we will briefly introduce lambda calculus.

3.3.4 Lambda calculus

The restriction of allowing a function in FWLua to only take one argument comes from the lambda calculus. Lambda calculus is a formal language representing computations based on functions. Function abstraction and function application are the two main parts in the syntax of lambda calculus.

There are several advantages using lambda calculus: it can capture the essence

of functional programming language; it has a minimal syntax while is able to build a very complex features. Functions in FWLua will use the system of lambda calculus. It is because lambda calculus is so powerful that enables us to remove many features, which can be represented using lambda calculus, from the full version of Lua. We will further show the detail in Section 5.4.

In the very basic lambda calculus, the symbol 'λ' means a function, the letter appears after 'λ' means the argument this function takes. Besides, lambda calculus uses another expression as the body of function to form the function abstraction. In representing a function application, there is also an extra letter behind the function abstraction meaning "the parameter this function applies". We can represent functions in FWLua using lambda calculus. For instance, the following is a function application in FWLua:

```
(function(x) return x end)(a)
```

This can be represented as a function application (λ`x.x a`) using lambda calculus. In the example, the expression λ`x.x` means the function abstraction, taking argument `x` and returning `x` as the result. Furthermore, the letter `a` shows the variable that this function applies.

Meanwhile, functions taking multiple arguments in Lua are very common. Lambda calculus also gives the solution about this. Based on the syntax, the function taking 2 arguments (`x` and `y`) can be shown as λ`x.`λ`y.y`. However, functions in Lua are far more complex than this, since there may be many features need to be taken into the consideration, such as scoping, closure, or outer environment. We will further talk about how we implement the syntactic desugaring according to the basic lambda calculus in Chapter 5.

3.4 Metatables and metamethods

Metatables allow us to define behaviors as fallbacks when Lua cannot handle an operation. These operations even include symbols whose behavior we think of as being fixed, such as "+" or "-". Also, metamethods in metatables can be automatically invoked once the relative conditions are satisfied. In other words, the behaviors of tables can be further controlled to follow our new rules in the program by using metatables. That is the reason why metatables are such an important characteristic of Lua.

However, metatables can only be set for tables. We cannot assign a metatable to some constant values, like numbers or strings. Those data types are too basic, and changing behaviors of some operator for them will cause the chaos. At least, the authors do not want an equation "1 + 1" results in 3.

Although metatables are a core element in Lua, we represent their behavior as part of the desugaring process, rather than bringing it into the basic semantics of FWLua. It is because we want to make FWLua as clean and simple as possible. We encode metatables in the translation phase, which means metatables can be further translated as a special inner table when we desugar Lua into FWLua.

3.4.1 Introduction about metatables

Lua constructs a table without any metatable by default. However, we can set a metatable to tables. There are two reserved functions related to metatables: `setmetatable()` and `getmetatable()`. These two functions are used for setting and getting a metatable of a table. Lua allows users to set any table as a metatable of other tables. Or, there is a need of C code to manipulate the metatables using other types. However, we will skip studying C in the paper and only discuss the former

condition.

3.4.2 What's in metatable

Metamethods are reserved functions with a set of special names in a metatable. These functions can be automatically triggered; hence they are called "hooks". Actually, all the computations during programming can be thought as hidden functions. One of the purposes of metamethods is to let developers change the behavior of the programming language to a certain degree.

Furthermore, there are 4 different kinds of metamethods in Lua [20]: arithmetic, relational, library-defined, and table-access. According to their names, each of them carries functions toward different fields. The arithmetic and relational metamethods are mostly responsible for binary operations, and the other two are often for tables and reserved functions. Generally, values in Lua can only take arithmetic and relational metamethods, since what they defined would not change the normal behavior of the programming language. On the other hand, table-access metamethods will possibly change the behavior of tables for several situations.

Figure 3 introduces several metamethods we think are important for the research of Lua. In this figure, the name of reserved functions are shown at the left hand side, while the information flows about specific functions are shown at the right hand side. Additionally, information flow means what type of value this function will take and result – the last word of type in each equation shows its result, and others will show its arguments.

Table-Access Metamethods

$$\begin{aligned}
\texttt{__index}(get) &:: table \rightarrow string \rightarrow value \\
\texttt{__newindex}(set) &:: table \rightarrow string \rightarrow value \rightarrow \texttt{nil}
\end{aligned}$$

Arithmetic Metamethods

$$\begin{aligned}
\texttt{__add}(+) &:: value \rightarrow value \rightarrow value \\
\texttt{__sub}(-) &:: value \rightarrow value \rightarrow value \\
\texttt{__mul}(*) &:: value \rightarrow value \rightarrow value \\
\texttt{__div}(/) &:: value \rightarrow value \rightarrow value \\
\texttt{__mod}(\texttt{mod}) &:: value \rightarrow value \rightarrow value \\
\texttt{__pow}(\wedge) &:: value \rightarrow value \rightarrow value
\end{aligned}$$

Relational Metamethods

$$\begin{aligned}
\texttt{__eq}(==) &:: value \rightarrow value \rightarrow value \\
\texttt{__ne}(\sim=) &:: value \rightarrow value \rightarrow value \\
\texttt{__lt}(<) &:: value \rightarrow value \rightarrow value \\
\texttt{__le}(<=) &:: value \rightarrow value \rightarrow value \\
\texttt{__gt}(>) &:: value \rightarrow value \rightarrow value \\
\texttt{__ge}(>=) &:: value \rightarrow value \rightarrow value
\end{aligned}$$

Figure 3: Metatable Event Types

CHAPTER 4

Operational semantics for Featherweight Lua

The operational semantics of Featherweight Lua are deliberately minimal. No hooks, fallbacks or control structures are included in the semantics.

FWLua can be very helpful in analyzing Lua, since it provides easy way of transferring important features, which are first-class functions, table structures and metatables in Lua into FWLua. In FWLua, functions are treated as first-class values, meaning they can be assigned to a variable. Also, tables in FWLua are the primary data structure, just the same as in Lua. Finally, we don't provide any default metatables built in FWLua. Instead, we can easily build a structure like metatables and metamethods in FWLua via translation, which we will discuss in Chapter 5.

4.1 Syntax for Featherweight Lua

We now give the complete syntax of FWLua. In this section, we introduce the syntax for FWLua in Section 4.1 and the corresponding semantics in Section 4.2.

We have given some figures showing the syntax of expressions and statements separately, and have introduced them in detail before. Statements do not return any value after being evaluated, while expressions always evaluate to a value. We simplify our language by having statements always return a value just like expressions. While this is a difference from the full version of Lua, we feel it is a minor change that greatly simplifies the complexity of our semantics.

The full syntax of Featherweight Lua is given by Figure 4. We can see that there are both old version expressions such as constants and binary operations, and

statements. For expressions, we keep registers, constants, and new allocations. Instead of variable-like syntax `e1[e2]`, we use a function-like syntax for getting variable `rawget(e1, e2)`, where e_1 is the table and e_2 is the key in that table. The reason is that `rawget` is an important reserved function in Lua, which can get a variable without any hooks. Therefore, table select in Lua can be desugared using `rawget` in FWLua. A set of hooks also exists in the table assignment statements, so we instead use the raw function `rawset(e1, e2, e3)`, where e_1 also means the table, e_2 means the key, and e_3 means the value that we want to assign.

Finally, we add the syntax for functions, that is `function x return e end` with only one argument `x` and body of function `e`. However, for brevity we adopt the lambda calculus symbol λ for a simpler syntax representation to make it easier to read. In other words, the syntax "λ`x.e`" means "`function x return e end`" in FWLua. We define it as the function abstraction being used as function definition for the reasons we have mentioned before.

Also, there is the syntax of function application, and the structure is pretty simple: `e1(e2)`, where e_1 presents the name of function and e_1 means the value this function applies.

Not that there is no Boolean value in the list of constants. Because functions are similar to lambda calculus, the value of true and false can be encoded into specific functions. Rojas [22] talks about what these functions look like.

4.2 Semantics for Featherweight Lua

The full big-step semantics of Featherweight Lua (evaluation rules) is shown by Figure 5. In the semantics, there are 5 kinds of runtime variables: store, table, function, value, and name. The variable "store" is mapping of registers to tables.

$e ::=$		*Expressions*
	a	register
	c	constant
	$\{\}$	new allocation
	$\texttt{rawget}(e_1, e_2)$	table select
	$\texttt{rawset}(e_1, e_2, e_3)$	table update
	e `binop` e	binary operation
	$\lambda x.e$	abstraction
	$(e)(e)$	function application
$c ::=$		*Constants*
	n	number
	s	string
	nil	nil value
a		*Registers*
x		*Variables*
$op ::=$	$+ \mid - \mid * \mid / \mid .. \mid > \mid >= \mid < \mid <= \mid == \mid = \mid and \mid or$	*Binary operators*

Figure 4: Full Syntax of Featherweight Lua

Since we have mentioned that the table is the primary data structure in FWLua, all tables will be stored in the store with specific addresses, which are also "registers". Secondary, the variable "table" is a table that maps keys and values. "Function" is used as functions. The variable "value" means all types of values in FWLua. There is also a variable called "name" to represent some names using strings.

In the basic evaluation rules, we can see FWLua takes current expressions and stores before evaluating and will return a value and manipulated store when it is done.

FW-VALUE: This rule takes the value and returns it, with no other side effects.

FW-NEW: This rule allocates a new memory location in the store for a new, empty table `{}`. This table is added to the store (register to table), with the key of the new address. The new address is the resulting value.

FW-RAWGET: This semantic shows an interesting character of FWLua because it is very different with many common syntax in other language. In this evaluation rule, FWLua takes 2 arguments: table and key, for getting a variable in the right place. The first expression in this rule presents a direction, and the second expression is string type token, representing the name of variable in this table. What is more, because there are not any reserved words in FWLua, users may possibly need to use a recursive `rawget()` syntax in the first argument for locating the target table.

FW-RAWSET: `rawset()` is similar to `rawget()`: it evaluates its 3 arguments in the body and returns them as a table, key, and value at the first step and then attempts to find that table in the store as the second step. The difference is that the executer then adds the pair with key and value into this table if this table exists in the store. Since Lua merges variable declaration and assignment statement into one, `rawset()` now can represent both as the user needs: if the key in the `rawset` function does not already exist in the table, it will be created automatically.

FW-BINOP: The evaluation rule for binary operations takes the two operands of the operator, evaluates them to values, and then calculates the final result depending on the expected behavior of the operator[1]. However, different operators may be used in different data types, like `..` for strings and `+` for numbers. Therefore, there are two functions used for the type checking against operands: `validL(c, op)` for the left hand side and `validR(c, op)` for another side. The evaluation will be done

[1]The behavior for most of these operators is straightforward; for the sake of brevity, we do not provide exact definitions.

only if the types of both operands are valid. What is more, the operands in FWLua must be a constant (Number, Boolean, String). It is because tables in Lua need metatables to control their behavior in binary operations. We will place this issue in the desugaring phase and further talk about how we transfer binary operations for table in Lua into FWLua in the section 5.3.

FW-FUNC-APP: This rule evaluates a function application in FWLua. At the beginning, it evaluates an expression to a function. Then it evaluates the argument of this function. Finally, it evaluates the function body with the substitution from arguments to values.

The substitution step is also like evaluating. Actually, it walks through every node in an expression, and then substitutes values to arguments based on different type of node. A recursive call might be needed due to some specific type of node in an expression. What is more, there is possibly a special condition called `function renaming` appearing in a function when the nested function has some same-name arguments with the outer function. The substitution function considers this condition by a set of algorithms. The Appendix B shows the details of substitution function written in Haskell.

Runtime Syntax:

$$
\begin{array}{rcll}
\sigma & \in & \mathit{Store} & = & \mathit{register} \rightarrow \mathit{table} \\
T & \in & \mathit{Table} & = & \mathit{string} \rightarrow \mathit{value} \\
f & \in & \mathit{Function} & ::= & \mathit{function} \\
a & \in & \mathit{Register} & ::= & \mathit{string} \\
c & \in & \mathit{Constant} & ::= & \mathit{number} \mid \mathit{string} \\
v & \in & \mathit{Value} & ::= & \mathit{constant} \mid \mathit{register} \mid f \\
x & \in & \mathit{Name} & ::= & \mathit{string}
\end{array}
$$

Evaluation Rules: $\boxed{e, \sigma \Downarrow v, \sigma'}$

[FW-VALUE]
$$\frac{}{v, \sigma \Downarrow v, \sigma}$$

[FW-NEW]
$$\frac{a \notin dom(\sigma) \quad \sigma' = \sigma + (a := \{\})}{\{\}, \sigma \Downarrow a, \sigma'}$$

[FW-RAWGET]
$$\frac{e_1, \sigma \Downarrow a, \sigma_1 \quad e_2, \sigma_1 \Downarrow x, \sigma' \quad T = \sigma'(a) \quad v = T(x)}{\texttt{rawget}(e_1, e_2), \sigma \Downarrow v, \sigma'}$$

[FW-RAWSET]
$$\frac{e_1, \sigma \Downarrow a, \sigma_1 \quad e_2, \sigma_1 \Downarrow x, \sigma_2 \quad e_3, \sigma_2 \Downarrow v, \sigma_3 \quad T = \sigma_3[a] \quad T' = T \ + \ (x := v) \quad \sigma' = \sigma_3[a := T']}{\texttt{rawset}(e_1, e_2, e_3), \sigma \Downarrow a, \sigma'}$$

[FW-BINOP]
$$\frac{e_1, \sigma \Downarrow c_1, \sigma_1 \quad e_2, \sigma_1 \Downarrow c_2, \sigma' \quad \texttt{validL}(c_1, op) \quad \texttt{validR}(c_2, op) \quad c = c_1 \ op \ c_2}{e_1 \ op \ e_2, \sigma \Downarrow c, \sigma'}$$

[FW-FUNC-APP]
$$\frac{e_1, \sigma \Downarrow \lambda x.e, \sigma_1 \quad e_2, \sigma_1 \Downarrow v_1, \sigma_2 \quad e[x := v_1], \sigma_2 \Downarrow v, \sigma'}{(e_1)(e_2), \sigma \Downarrow v, \sigma'}$$

Figure 5: Full Semantics of Featherweight Lua

CHAPTER 5

Lua to Featherweight Lua

We have introduced the full syntax and semantics of FWLua in Chapter 4. We can see that FWLua is a much smaller language than the full version of Lua, with many differences between them. However, our purpose is to let FWLua capture the essence of Lua with only these core features. In this chapter, we discuss how FWLua works like Lua by showing how syntactic desugaring can be used to convert between FWLua and Lua. Since there are many additional features in Lua, we first discuss how to desugar in Lua to let the code fit FWLua. Secondly we give some translation rules from Lua to FWLua.

There are several important parts we focus on in desugaring Lua. First of all, getting variables is taken into consideration. Secondly, Lua allows many different ways of representing variable updates and declarations in different conditions and scopes. We also consider functions because there are some different qualities for functions in the full Lua language. Tables, in addition, need to be discussed since they are the primary data structure with metamethods and metatables. Finally, we discuss how we may desugar control statements.

Finally, we will discuss those parts in detail, along with examples, in the following sections. All the figures about desugaring below are not translated into FWLua completely, but can be finally translated by following the all the figures. The only reason doing this is to make the translation rules clearer to understand.

5.1 Variable select

Selecting variables is used for computations to get a user defined value. Generally speaking, getting variables in a common programming language is easy and straightforward: users simply write a variable as the key for getting its value. For example, the following expressions used in Lua would stand for "getting a value":

```
x ---get the value of x
t["x"] ---value of key "x" in table t
t.y ---value of key "y" in table t
```

Every variable in Lua has its own field. The first line of the example has no associated table. To normalize it, we need to give a table name to that variable to make it clear. Lua provides a reserved table, `_ENV` to represent the global environment. Therefore, every variable now can have a table.

The normal format, in Lua, is `e["e"]`. The expression `t.y` is syntactic sugar with key `y` since the key is the string type in the table. After we desugar these expressions, all variable select expressions in Lua can be represented as the format `e[e]`, such as the following:

```
_ENV["x"] ---get the value of x
t["x"] ---value of key "x" in table t
t["y"] ---value of key "y" in table t
```

Lua provides metamethods and metatables in table constructors. The metamethod `__index` points to a fallback table. If the key that users want to index is not available in the current table, this metamethod is triggered. However, there is a reserved function `rawget(t, k)` in Lua, which will index the key `k` in the table `t` and ignore the hook in its metatable.

5.2 Variable update & declaration

Variable update and declaration are two key procedures in a programming language. In these two different procedures, one or more variables are declared or updated in the specific scope. There are many ways of defining variables in Lua, even including declaring multiple variables in one statement. Before we discuss desugaring, the following example program gives each way of doing variable declaration and updating.

```
x = 11 ---declare a variable x
x = 12 ---update x since it has existed
t.y = 63 ---key variable declarations in table
a,b,c,d = 1,2+3,4,5 ---destructuring assignment statement
local f = "Hello" ---local variable declaration in scope
```

Direct table select in Lua is treated as a syntactic sugar, and we desugar it to computed table select. Multiple variable declarations can be represented using a set of single assignment statements. The local variable declaration is more complicated. Lua actually finds out which scope, or which table, should the variable be set in a fixed order.

Local variable is a very interesting point in setting variable. Actually, local variable, as its name, is a key-value pair that only can be accessed in the local or inner scope. There is a need to allocate a table for recording every local variable in a specific scope. What is more, there are only 2 places allowed for setting a local variable using reserved word "`local`" in Lua: blocks and functions, while we can easily translate a block into a set of lambda functions. As the result, we can handle the local variable in translating functions in Lua and thus will mainly discuss it in Section 5.4.

There is also a reserved function in Lua, rawset, to get rid of effects of metatables in setting variables, which is `rawset(t, k, v)`. This is the reason we build our syntax in FWLua using this format, and will then desugar Lua to FWLua, followed by telling mechanisms in Lua. An example of decoding assignments in Lua to FWLua is shown below.

```
t.x = 12 -- Lua, in the table of t
rawset(rawget(_ENV, "t"), "x", 12) --translated in FWLua
```

Figure 6 shows how we desugar syntax about variables in Lua, using FWLua. We can also call this step "Normalizing" since this is the first step of handling variables in tables for the translation. The main purpose is to remove all syntactic sugars in dealing with variables. Next, we will talk about translating tables and metatables based on this section.

5.3 Tables and Metatables

Based on the full version of Lua, table constructors are used as expressions to assign a set of pairs with key and value in the specific table. Instead of it, there is only one syntax in FWLua called "new allocation" to allocate and then be evaluated as a new address in the store. However, the syntax `rawset(t, s, v)` in FWLua will also be evaluated as an address, just as this function does in Lua. We thus can assign pairs with values and keys, or changing behaviors using some kinds of expressions about this table. In other words, we will completely decompose the whole syntax of "table constructor" in to a set of single expressions in FWLua.

To make it more concrete, we will show several examples about variable handling of table in Lua. First, consider the following example code:

```
table = {x=1, y=2}
```

Based on the above example, we first need to allocate a memory for this table. The table constructor "{}" in FWLua is regarded as a type of value. We therefore can just assign a table constructor to the variable, just like assign a constant, by using variable setting syntax in FWLua. The next step will be pretty straightforward. Only we need to do is to assign every key-value pair to into this table by using the same syntax. As the result, following shows the translated code, written in FWLua:

```
rawset(_ENV, "table", {}) ---new allocation
rawset(rawget(_ENV, "table"), "x", 1)
rawset(rawget(_ENV, "table"), "y", 2)
```

Or, because rawset in FWLua can also return that address, we can use a sequence of rawset expression, like a recursive function application:

```
rawset(rawset(rawset(_ENV, "table", {}) , "x", 1), "y", 2)
```

However, metatables and metamethods have to be considered. In Section 2.6, we have discussed that tables can be used as different data structures by changing metatables or defining some functions. Therefore, we reserve a table with the name of `_metatable` to represent metatables in Lua. There are also two functions, `setmetatable(t, mt)` and `getmetatable(t)`, highly related to table constructor. We can desugar them by using `rawset`, `rawget`, and `_metatable` in FWLua. Just like the following example:

```
t = {};
mt = {};
setmetatable(t, mt);
getmetatable(t);
```

This can be translated into FWLua like the following:

```
rawset(_ENV, "t", {})
rawset(_ENV, "mt", {})
rawset(rawget(_ENV, "t"), "_metatable", rawget(_ENV, "mt"))
rawget(rawget(_ENV, "t"), "_metatable")
```

What is more, we have mentioned the reserved metamethods about binary operations for table. To finish binary operations for table in FWLua, we can easily get the reserved functions in the table `_metatable` and then apply it with the two arguments in an binary operation.

Finally, getting and setting an variable in table is a key point. As we mentioned, some metamethod, like `__index` or `__newindex`, will automatically be triggered when we are missing the key in getting or setting variables. Therefore, in Lua, getting or setting variables in Table is actually a set of algorithms, which will look for the fallback table when there is no matched key-value pairs in current table.

Figure 6 show how to translate Lua into FWLua about table constructors. It is noteworthy that, rawset expressions in both Lua and FWLua return an address, while an assignment statement in Lua returns nothing.

5.4 Functions and control structures

Before desugaring functions in Lua, we first introduce those differences between functions in Lua and in FWLua. Since functions in FWLua are similar to lambda calculus, there is an obvious difference, which is that each function in FWLua always takes exactly one argument as its perimeter. Therefore, we are going to make every function in Lua as the style of multiple functions in FWLua, or in lambda calculus. For instance, consider a simple Lua function:

```
function (x,y) return e end
```

Variable Select:

$$e.x \stackrel{def}{=\!=} e[\text{“}x\text{”}]$$

Variable Declaration & Update:

$$e_1.x = e_2 \stackrel{def}{=\!=} e_1[\text{“}x\text{”}] = e_2$$
$$\overline{e_i} = \overline{e_j} \stackrel{def}{=\!=} \overline{e_i = e_j;}$$

Table:

$$T.t = \{\overline{e_i = v_j}\} \stackrel{def}{=\!=} \texttt{rawset}(...\texttt{rawset}(\texttt{rawset}(T, \text{“}t\text{”}, \{\}), e_1, v_1)...)$$
$$\texttt{setmetatable}(t, mt) \stackrel{def}{=\!=} \texttt{rawset}(t, \text{“}\_metatable\text{”}, mt)$$
$$\texttt{getmetatable}(t) \stackrel{def}{=\!=} \texttt{rawget}(t, \text{“}\_metatable\text{”})$$
$$t_1\ op\ t_2 \stackrel{def}{=\!=} ((\texttt{rawget}(\texttt{rawget}(a, \text{“}\_metatable\text{”}),\ op))(t_1))(t_2)$$

$$e_1[e_2] \stackrel{def}{=\!=} \lambda e_1.\lambda e_2.$$

```
if rawget(e1, e2) ~= nil then
  rawget(e1, e2)
else
  local f =
    rawget(getmetatable(e1), __index);
  if f == nil then
    nil
  else
    f(e1, e2)
  end
end
```

$$e_1[e_2] = e_3 \stackrel{def}{=\!=} \lambda e_1.\lambda e_2.\lambda e_3.$$

```
if rawget(e1, e2) == nil then
  rawset(e1, e2, e3)
else
  local f =
    rawget(getmetatable(e1), __newindex);
  if f == nil then
    nil
  else
    f(e1, e2, e3)
  end
end
```

Figure 6: Translation rules of Variables and Tables

An equivalent function can be written in a style closer to FWLua as follows:

```
function (x) return
  function (y) return
    e
  end
end
```

Return statements in Lua functions are optional, which means a function might not return a value. However, return statements are required in every function in FWLua. Furthermore, a function in Lua might have its own scope and supports local variables. Since we don't provide local variables to functions in FWLua, we now treat the local scope as a temporary table. Now, we discuss details about functions in Lua.

To capture other additional information about Lua functions, we translate every function in Lua to a table with fields for basic information about the function. To implement the scope, there are four arguments that must be taken into the consideration when we translate a function in Lua into FWLua: `_local` for variables in the local environment; `_arg` for storing the arguments of the function; `_outer` for the outer scope; and `_call` is a function to store the body of the function. There is an example in Appendix A.

In the structure, we only bring two new keys in the table, which are argument and body, at the time when we declare a function. However, we keep the outer scope and local scope as the inner variable in the body function to decrease the complexity in translating function application. In translating function body, we will assign a table as the local scope and will record the outer scope and argument. In other words, the information about outer scope and argument names should be recorded along with local variables in the local scope. By doing this, we can easily roll back to find further outer scopes by a recursive function application.

Finally, one of the key points in the function application is to distinguish the scope of the current variable, for example:

```
x = 40
function (x) return
  local x = 10
  x = 20
  return x
end
```

Based on our test, when get a variable in a function without any prefix. There is a fixed order in locating the scope of this variable. This order is exactly like the following algorithm, and it will stop going when there is match key-value pairs:

```
1. Check local scope
2. Check argument
3. Repeat this algorithm in the outer scope.
```

In the step 3, this algorithm will be repeated. Therefore there is a recursive function application in getting and setting variables in the function. Figure 7 shows details about this algorithm in FWLua. To get more details in FWLua, see Appendix C.

Also, global environment is very interesting based on this algorithm. Consider the following code in Lua:

```
local a = 11; ---local variable in _ENV
print (a); ---result nil
a = 20
print(a) ---result 20
local a = 14
print(a) ---still result 20
```

Therefore, local variables in global environment can be declared, but can never be goten. This is a very interesting point in Lua and is worth to notice.

Reserved Functions:

```
getValue = λ_local.λ_var.
             (λ_outer.λ_arg.
               if rawget(_local, _var) ~= nil then
                 rawget(_local, _var)
               else
                 if rawget(_arg, _var) ~= nil then
                   rawget(_arg, _var)
                 else
                   if _outer == _ENV then nil
                     nil
                   else
                     getValue(_outer, _var)
                   end
                 end
               end)(_local._outer, _local._arg)

setValue = λ_local.λ_var.λ_value
             (λ_outer.λ_arg.
               if rawget(_local, _var) ~= nil then
                 rawset(_local, _var, _value)
               else
                 if rawget(_arg, _var) ~= nil then
                   rawset(_arg, _var, _value)
                 else
                   if _outer == _ENV then nil
                     nil
                   else
                     setValue(_outer, _var, _value)
                   end
                 end
               end)(_local._outer, _local._arg)
```

Figure 7: Reserved Functions in Translation rules

Lambda function applications can be used to represent a sequence of statements. We therefore translate sequences of statements into functions in FWLua. Lambda functions can also be used to represent control structures and boolean values, which we discuss in the next section.

Figure 8 gives the rules for translating functions. As above, the results of the translations are all in FWLua syntax, or in other constructs that we define.

5.4.1 Control structures

Control structures can be defined in terms of a set of lambda functions. Following the standard pattern used in the lambda calculus, we can treat the constant `true` as a function taking 2 parameters[1] and returning the first value, and treat `false` as a function taking 2 parameters and returning the second value. The syntax is as follows:

```
function (x) return
  function (y) return
    x
  end
end --True

function (x) return
  function (y) return
    y
  end
end --False
```

Using these constructs for true and false, we now can translate the basic control structure in Lua, the `if` statement, into a complicated reserved lambda function in FWLua. Pierce [23] shows us the details about translating control structure statements into lambda functions.

According to our analysis, using a set of conditional statements and recursive function calls can represent all forms of loop statements in Lua. For example:

```
x = 0;
```

[1] More strictly speaking, `true` is a function that takes one parameter and returns another function that takes one parameter and returns the argument passed to the outer function.

```
while x <= 10 do x = x+1; end
```

The above code is a loop statement. It increments the variable `x` by one each time, until it equals 10. We can see that, a recursive computation and a conditional statement are in the loop: the program tests whether `x` is equal to 10, and then recursively executes this computation. Therefore, they can be represented as the following:

```
function while()
  if x <= 10 do
    x = x+1; while()
  else
    return nil
  end
end; while()
```

Hence all different kind of loop statements can be desugared using conditional statements. Figure 8 gives other desugaring rules.

Function:

$$
\begin{array}{rcl}
e_1;\ e_2 & \stackrel{def}{=\!=} & (\lambda d.e_2)(e_1)\ \textit{where}\ d \notin e_2 \\
\texttt{function}\ f(\overline{x_i})\ e\ \texttt{end} & \stackrel{def}{=\!=} & f = \{\text{“}\texttt{\_arg}\text{”} = \{\},\ \text{“}\texttt{\_call}\text{”} = \lambda\_arg. \\
 & & \quad (\lambda\_outer.\lambda\_local.e)(\\
 & & \quad \{\text{“}\texttt{\_outer}\text{”} = \_outer,\ \text{“}\texttt{\_arg}\text{”} = \_arg\}, \\
 & & \quad \_local' \\
 & & \quad) \\
 & & \} \\
f(\overline{v_i}) & \stackrel{def}{=\!=} & (f.\_call)(\{\overline{x_i = v_i}\}) \\
 & & \\
\texttt{local}\ x = e & \stackrel{def}{=\!=} & \texttt{rawset}(\_local, \text{“}x\text{”}, e) \\
x & \stackrel{def}{=\!=} & \texttt{getValue}(\_local, x) \\
x = e & \stackrel{def}{=\!=} & \texttt{setValue}(x, e, \_local)
\end{array}
$$

Control Structure:

$$
\begin{array}{rcl}
\texttt{True} & \stackrel{def}{=\!=} & \lambda x.\lambda y.x \\
\texttt{False} & \stackrel{def}{=\!=} & \lambda x.\lambda y.y \\
\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{end} & \stackrel{def}{=\!=} & (((e_1)(\lambda d.e_2))(\lambda d.e_3))(\lambda x.x)\ \textit{where}\ d \notin e_2, e_3 \\
\texttt{while}\ e_1\ \texttt{do}\ e_2\ \texttt{end} & \stackrel{def}{=\!=} & \texttt{function}\ f() \\
 & & \quad \texttt{if}\ e_1\ \texttt{then}\ e_2; f()\ \texttt{else return nil end} \\
 & & \texttt{end}; f() \\
\texttt{repeat}\ e_1\ \texttt{until}\ e_2 & \stackrel{def}{=\!=} & \texttt{while not}\ e_2\ \texttt{do}\ e_1\ \texttt{end} \\
\texttt{for}\ x = e_1\ ,\ e_2\ ,\ e_3\ \texttt{do}\ e_4\ \texttt{end} & \stackrel{def}{=\!=} & \texttt{function}\ f() \\
 & & \quad \texttt{local}\ x = e_1; \\
 & & \quad \texttt{while}\ x <= e_2\ \texttt{do}\ x = x + e_3;\ e_4\ \texttt{end} \\
 & & \texttt{end}; f()
\end{array}
$$

Figure 8: Translation Rules of Lua's Functions

CHAPTER 6

Implementation: Interpreter in Haskell

Our implementation is an interpreter written in Haskell, which is a popular functional programming language.

Haskell provides a static type system, a fast runtime speed; since it treats computations as a set of expressions of mathematical functions, and this will possibly avoid mutable variables.

Haskell also has simple memory allocation due to the research [28]. As the trade off, Haskell does not allow mutable variables, and this would obviously differ in building interpreter between object-oriented language and functional language.

In this chapter, we discuss the structure of our interpreter, and introduce how each component works in the structure. In addition, Appendix D shows an instance program using FWLua and the result interpreted by our implementation.

6.1 Structure

According to the reference [29], we want our interpreter to be loosely-coupled [30]. This term means that the program can be treated as several components. In addition, each component in the program is totally independent, with little or no shared information or definitions with other parts. This term was introduced in case of keeping program adaptive, especially in designing compiler of interpreter. In other words, a loosely-coupled interpreter can handle different kinds programming language by only changing specific key components in it.

Therefore, we design our interpreter to be loosely-coupled by splitting it into

several parts. Basically, there are three main parts in the program, and we will make files for each of them. The file `ParserTD.hs` is built for the front tier of our interpreter. Its purpose is to parse the code we write from the top down, and thus translate it into an abstract syntax tree (AST) with all defined node by us.

Secondly, the file `Executor.hs` is the backend of the interpreter and is used for evaluating the AST that the parser produces. In doing this, all the results Lua returns will be from the executer, after being evaluated.

There is also an intermediate tier that we call the "Symbol Table". This tier stores any information we need in the interpreter such as variable types, reserved words, type definitions and so on. While either the parser or the executer is running, they will visit the symbol table and get information they need to run.

Above all, the structure of our FWLua interpreter is shown in the Figure 9. We also introduce the detail in the following sections.

6.2 ParserTD.hs

Fortunately, there is a package named `Text.ParserCombinators.Parsec` in Haskell, providing a set of functions for parsing tokens and thus building parsers. We will use this package to build the parser. Reference [31] shows the details of this package.

The key role in a functional programming language is recursive functions. Basically, this package allows us to parse token one by one recursively and thus control the data flow. According to this, the parser uses `parsec` for passing tokens and it takes strings as tokens. Then it returns a tree –an abstract syntax tree (AST). The AST is made up of different types of nodes. Technically, all nodes in the AST will

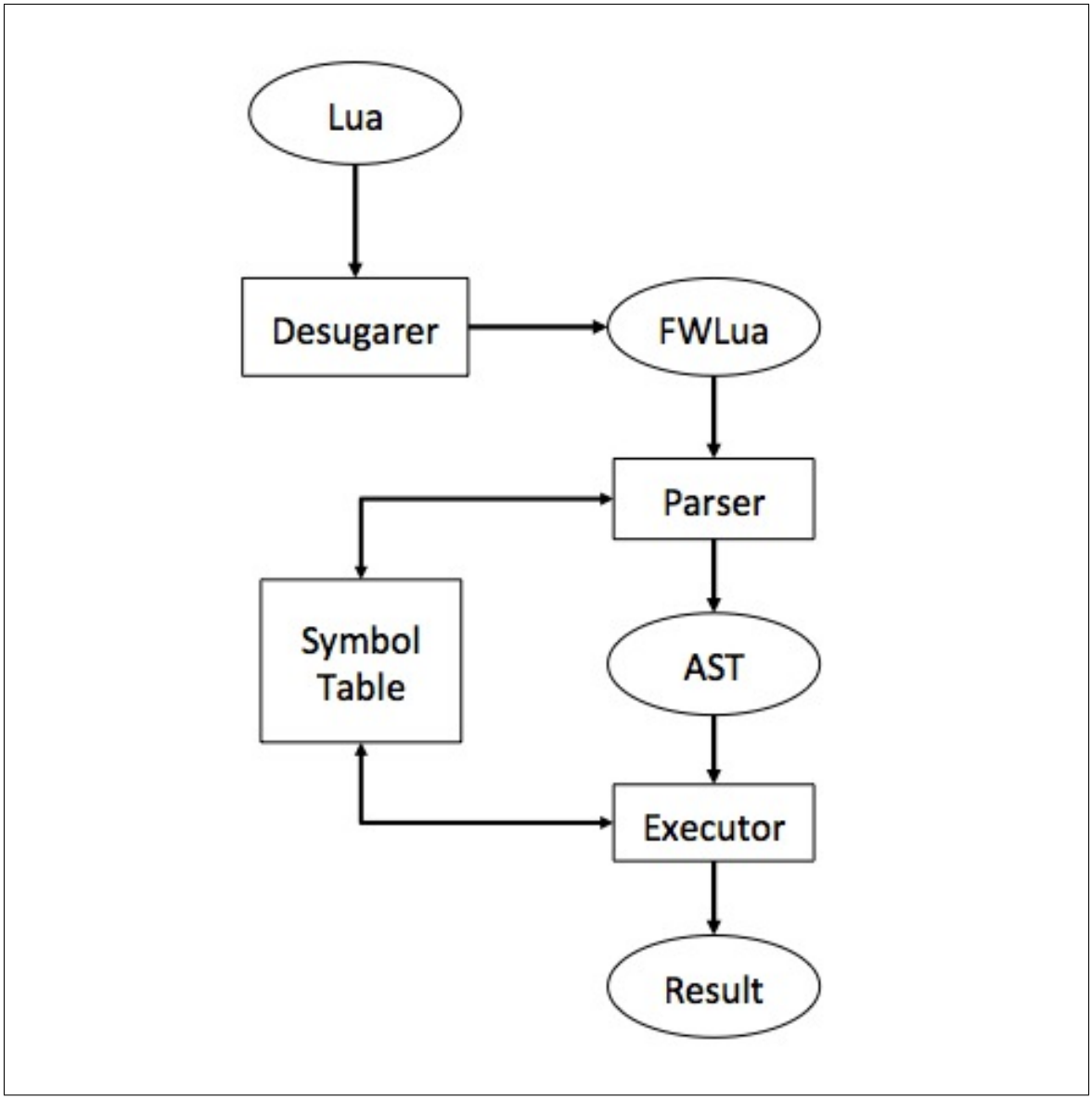

Figure 9: Structure of the interpreter

exactly represent the whole program. Every time it parses a token, AST should be modified (even could be "do nothing").

The parser cares about the syntax of FWLua. In other words, there are no evaluations or storage manipulations in the parsing phase. The only purpose in this block is to sort the inputs by using the structure of the tree, and thus bring the

cleaner AST to the executer. The AST is completely created by parser. It also means that we can handle different kinds of inputs by using different parsers with the same executor.

6.3 Symbol Table

Haskell is a strongly-typed programming language. Everything we want to define needs to have a type, and Haskell strictly follows this type. The tier of the symbol table stores all new types we define and it is used as a user-defined library.

There are two ways declaring new data types in Haskell: `data` and `type`. The token `type` allows us to define a new type with one single format, while the token `data` defining a new variable type in multiple formats with different tokens. For instance, the part of code in the symbol table is shown:

```
type Store = Map Register Table
type Argument = String
data Value =
VNil
| VArg String
| VFunc Argument Expression
| VResFunc String
| VReg Register
| VInt Integer
| VBool Bool
| VStr String
| VTrue
| VFalse
deriving (Show)
```

The type `Store` and `Argument` have the single typing paradigm. They don't need a specific token to be distinguished. On the contrary, the data `Value` has multiple cases and thus needs different tokens (such as `VArg`, `VFunc` and `VInt` in the code above).

6.4 Executor.hs

`Executor.hs` represents the semantics of FWLua that we have given above. The function `evaluate()` is the main function in the file. It takes different kinds of nodes in the AST for evaluating, then returns values and the manipulated store as the final result. There are also some other assisting reserved functions for helping the evaluation rules such as address allocation, key pointing, binary operation application and so on. At the beginning, the global store is supposed to be empty.

6.5 Run and Runfile

We have also built a run file to test if our interpreter works as intended. What is more, this file links all components that we introduced before. In the run file, we treat inputs as a string flow. Then it will parse it, execute the AST and finally output the results. The results include the desugared code, detail of the AST, the summary of store, and final results. Since it shows all the information that we need, we can debug our interpreter due to the outputs from this run file.

CHAPTER 7

Conclusion & Future work

This paper shows the syntax and semantics of the full version of Lua, and thus a simpler version of Lua called *Featherweight Lua* (FWLua). FWLua removes a lot of elements and only retains the key components from Lua. We have focused on the exotic and important features, which are metatables and metamethods. During the research, we found that those factors are very powerful for Lua since they can change the behavior of the programming language. They can even model many different kinds of data structures combining the primary data structure in Lua — tables.

In addition, we also give an approach to desugar Lua into FWLua. Due to this, users can gain a clearer understanding of the full version of Lua through this core language. We also give an interpreter implemented in Haskell as the implementation. This implementation proves to be very useful during our studying about the relationship between Lua and FWLua. With this implementation, we are able to test and verify our translations and semantics.

LIST OF REFERENCES

[1] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight java: a minimal core calculus for java and GJ. *ACM Trans. Program. Lang. Syst., 23(3):396âĂŞ450*, 2001. URL: http://doi.acm.org/10.1145/503502.503505, doi:10.1145/503502.503505.

[2] Thomas H. Austin, Tim Disney, Alan Jeffrey, and Cormac Flanagan. Dynamic information flow analysis for featherweight javascript, 2011.

[3] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. Lua an extensible extension language. *Softw., Pract. Exper., 26(6):635âĂŞ652*, 1996. URL: http://dx.doi.org/10.1002/(SICI)1097-024X(199606)26:6<635::AID-SPE26>3.0.CO;2-P.

[4] Lua 5.2 Reference Manual, *Roberto Ierusalimschy, Luiz Henrique de Figueiredo, Waldemar Celes*, 2013.

[5] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes. The evolution of an extension language: A history of lua, *In Proceeding of V Brazilian Symposium on Programming Language, pages 14âĂŞ28*, 2001.

[6] Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho. The evolution of lua. *In Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007, pages 1âĂŞ26*, 2007. URL: http://doi.acm.org/10.1145/1238844.1238846, doi:10.1145/1238844.1238846.

[7] Matheson, Ash, An Introduction to Lua. *GameDev.net. Accesat la 3 ianuarie 2013*, 2003.

[8] Paul Emmerich, Beginning Lua with World of Warcraft Addons, *ISBN: 9781430223719*, 2009.

[9] Curse Inc. Curse.com, Recently accessed in 2015. URL: http://www.curse.com/addons/wow.

[10] CloudFlare, Lua: the world's most infuriating language, 2013.

[11] Adobe Photoshop Lightroom & Lua, 2007, Recently accessed in 2015. URL: http://blog.kodekabuki.com/post/29578340/adobe-photoshop-lightroom-lua.

[12] Cisco Inc, Cisco ASA Software SharePoint RAMFS Integrity and Lua Injection Vulnerability, 2014.

[13] Pushing Nginx to its limit with Lua, 2012. URL: http://blog.cloudflare.com/pushing-nginx-to-its-limit-with-lua/, Matthieu Tourne.

[14] Mike Pall, Introduction to LuaJIT, Recently accessed in 2015. URL: http://wiki.luajit.org/New-Garbage-Collector.

[15] Well House Consultants LTD. Functions are first class variables in Lua and Python, Recently accessed in 2015. URL: http://www.wellho.net/mouth/3695_Functions-are-first-class-variables-in-Lua-and-Python.html.

[16] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of javascript. *In ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings, pages 126âĂŞ150*, 2010. URL: http://dx.doi.org/10.1007/978-3-642-14107-2_7, doi:10.1007/978-3-642-14107-2_7.

[17] Hayo Thielecke, University of Birmingham, An introduction to operational semantics and abstract machines, 2012.

[18] Atsushi Igarashi, Benjamin C. Pierce, Philip Wadler, Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 23 Issue 3, May 2001, Pages 396-450*, 2001.

[19] Jeremy Siek, The Semantics of a Familiar Language: Featherweight C, 2012. URL: http://siek.blogspot.com/2012/07/the-semantics-of-familiar-language.html.

[20] Roberto Ierusalimschy, Programming in Lua, Third Edition, 2013.

[21] Mozilla Developer Network, Destructuring assignment, Recently accessed in 2015. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment.

[22] Raul Rojas, A Tutorial Introduction to the Lambda Calculus, 1997.

[23] Benjamin C. Pierce. Types and programming languages: The next generation. *In 18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings, page 32*, 2003. URL: http://dx.doi.org/10.1109/LICS.2003.1210042, doi:10.1109/LICS.2003.1210042.

[24] Lua Type Checking, Recently accessed in 2015. URL: http://lua-users.org/wiki/LuaTypeChecking.

[25] Lua Scoping, Recently accessed in 2015. URL: http://lua-users.org/wiki/LuaScoping.

[26] Kurt Jung, Aaron Brown, Beginning Lua Programming, 2012.

[27] Douglas Crockford, Prototypal Inheritance in JavaScript, 2008. URL: http://javascript.crockford.com/prototypal.html.

[28] Graham Hutton, Programming in Haskell, 2007.

[29] Ronald Mak, Writing Compilers and Interpreters: A Software Engineering Approach, 2009.

[30] Doug Kaye, Loosely Coupled: The Missing Pieces of Web Services, 2008.

[31] Instruction of Haskell, 2008. URL: https://hackage.haskell.org/package/parsec-3.1.0/docs/Text-ParserCombinators-Parsec.html.

APPENDIX A

Lua to FWLua: functions

A.1 Code in Lua

```
function foo()
  local x = 0;
  return
    function()
       x = x + 1;
       return x;
    end;
end;
bar = foo();
bar();
```

A.2 Code in FWLua

```
rawset(_ENV, "foo", rawset(rawset({}, "_arg", {}), "_call",
  function (_arg) return
    (function (_outer) return
      (function (_local) return

          (function () return

            rawset(rawset({}, "_arg", {}), "_call",
              function (_arg) return
                (function (_outer) return
                  (function (_local) return
                    (function () return
                      ((rawget(_ENV, "getValue"))("x"))(_local)
                    end)(
                      (((rawget(_ENV, "setValue"))("x"))
                        (
                          (
                            (
                              (
                                rawget(_ENV, "getValue")
                              )("x")
                            )(_local) + 1
                          )
                        )(_local)
                      )
                    end
                    )(rawset(
                        rawset({}, "_outer", _outer), "_arg", _arg))
                end)(_local)
              end)

          end)(
            rawset(_local, "x", 10)
          )
      end)(rawset(rawset({}, "_outer", _outer), "_arg", _arg))
    end)(_ENV)
  end)
);

rawset(_ENV, "bar", (rawget(rawget(_ENV, "foo"), "_call"))({}));
(rawget(rawget(_ENV, "bar"), "_call"))({});
```

APPENDIX B

Substitute function in Haskell

```
substitute :: Expression -> String -> Value ->
              Either ErrorMsg Expression
substitute expr a v = do
  case expr of
    Seq expr1 expr2 -> do
      expr1' <- substitute expr1 a v
      expr2' <- substitute expr2 a v
      return $ Seq expr1' expr2'
    Val value -> do
      case value of
        VFunc arg body -> do
          body' <- substitute body a v
          return $ Val $ VFunc arg body'
        VArg arg -> do
          case (arg == a) of
            True -> return $ Val v
            False -> return $ Val value
        otherwise -> return expr
    New -> return expr
    Rget table key -> do
      table' <- substitute table a v
      key' <- substitute key a v
      return $ Rget table' key'
    Rset table key val -> do
      table' <- substitute table a v
      key' <- substitute key a v
      val' <- substitute val a v
      return $ Rset table' key' val'
    Opraw expr1 op expr2 -> do
      expr1' <- substitute expr1 a v
      expr2' <- substitute expr2 a v
      return $ Opraw expr1' op expr2'
    Funcall func expr -> do
      case func of
        (Val (VFunc arg e)) -> do
          case (arg == a) of
            True -> do
```

```
      expr' <- substitute expr a v
      return $ Funcall func expr'
    False -> do
      func' <- substitute func a v
      expr' <- substitute expr a v
      return $ Funcall func' expr'
otherwise -> do
  func' <- substitute func a v
  expr' <- substitute expr a v
  return $ Funcall func' expr'
```

APPENDIX C

Reserved functions in FWLua

```
rawset(_ENV, "if",
  function(cond) return
    function(t) return
      function(f) return
        ((cond)(t))(f)
      end
    end
end);

rawset(_ENV, "getValue",
  function(_var) return
    function(_local) return
      (function(_arg) return
        (function(_outer) return

          ((((rawget(_ENV, "if"))
           (rawget(_local, _var) ~= nil))(
            function() return
              rawget(_local, _var)
            end))(
            function() return
              ((((rawget(_ENV, "if"))
               (rawget(_arg, _var) ~= nil))(
                function() return
                  rawget(_arg, _var)
                end))(
                function() return
                  ((((rawget(_ENV, "if"))
                   (_local == _ENV))(
                    function() return
                      nil
                    end))(
                    function() return
                      ((rawget(_ENV, "getValue"))
                       (_var))
                        (_outer)
                    end)
```

```
                )()
              end)
            )()
          end)
        )()

      end)(rawget(_local, "_outer"))
    end)(rawget(_local, "_arg"))
  end
 end
);

rawset(_ENV, "setValue",
  function(_var) return
    function(_value) return
      function(_local) return
        (function(_arg) return
           (function(_outer) return

            ((((rawget(_ENV, "if"))
             (rawget(_local, _var) ~= nil))(
              function() return
                rawset(_local, _var, _value)
              end))(
              function() return
                ((((rawget(_ENV, "if"))
                 (rawget(_arg, _var) ~= nil))(
                  function() return
                    rawset(_arg, _var, _value)
                  end))(
                  function() return
                    ((((rawget(_ENV, "if"))
                     (_local == _ENV))(
                      function() return
                        rawset(_local, _var, _value)
                      end))(
                      function() return
                        (((rawget(_ENV, "setValue"))
                         (_var))
                          (_value))
                           (_outer)
                      end)
```

```
                    )()
                  end)
                )()
              end)
            )()


          end)(rawget(_local, "_outer"))
        end)(rawget(_local, "_arg"))
      end
    end
  end
);
```

APPENDIX D

Lua to FWLua: functions

D.1 Program of Featherweight Lua

```
rawset(_ENV, "x", 1);
rawset(_ENV, "foo", function (x) return x + 1 end);
(rawget(_ENV, "foo"))(rawget(_ENV, "x"));
```

D.2 Abstract Syntax Tree

```
Seq (Rset (Val (VReg "_ENV")) (Val (VStr "x")) (Val (VInt 1)))
(Seq (Rset (Val (VReg "_ENV"))
  (Val (VStr "foo"))
  (Val (VFunc "x" (Opraw (Val (VArg "x")) Plus (Val (VInt 1))))))
    (Funcall (Rget (Val (VReg "_ENV")) (Val (VStr "foo")))
       (Rget (Val (VReg "_ENV")) (Val (VStr "x")))))
```

D.3 Store Information

```
"x":=VInt 1
"foo":=VFunc "x" (Opraw (Val (VArg "x")) Plus (Val (VInt 1)))
```