# Models of Calculation Project
# The Language Lua

Serena Rietbergen, Frank Gerlings, Lars Jellema

April 2016

# Contents

# 1 Introduction

In this article, we're going to define axiomatic semantics for coroutines in Lua. We're going to start by defining the basics of Lua, including some basic data types which we'll use in our examples. Then follows a description of coroutines, which is the built-in concurrency tool of Lua. Coroutines in Lua are collaborative, which means a coroutine must explicitly yield its cpu to other coroutines. Only one coroutine ever runs at any time, which means coroutines do not offer parallelism. They still offer a useful abstraction though. As While doesn't offer any kind of concurrency, we believe Lua is sufficiently special.

## 1.1 Example code

```
-- from http://lua-users.org/wiki/CoroutinesTutorial

function odd(x)
  print('A: odd', x)
  coroutine.yield(x)
  print('B: odd', x)
end

function even(x)
  print('C: even', x)
  if x==2 then return x end
  print('D: even ', x)
end

co = coroutine.create(
  function (x)
    for i=1,x do
      if i==3 then coroutine.yield(-1) end
      if i % 2 == 0 then even(i) else odd(i) end
    end
  end)

count = 1
while coroutine.status(co) ~= 'dead' do
  print('----', count) ; count = count+1
  errorfree, value = coroutine.resume(co, 5)
  print('E: errorfree, value, status', errorfree, value, coroutine.status(co))
end
```

# 2 Description of the syntax

The following section describes the grammar of the subset of Lua that we'll be describing. Numerous Lua constructs have been left out for the sake of simplicity, like multiple assignment for example. Whitespace has also been left out of the grammar, as have the optional semicolons.

⟨*block*⟩ ::= ⟨*stmt*⟩ ⟨*block*⟩ | '`break`' | ''

⟨*expr*⟩ ::= ⟨*bool*⟩ | ⟨*number*⟩ | ⟨*string*⟩ | ⟨*table*⟩ | ⟨*func*⟩ | ⟨*thread*⟩ | ⟨*nil*⟩ | ⟨*table-access*⟩

⟨*nil*⟩ ::= '`nil`'

⟨*bool*⟩ ::= '`true`' | '`false`'

⟨*stmt*⟩ ::= ⟨*do*⟩ | ⟨*if*⟩ | ⟨*while*⟩ | ⟨*repeat*⟩ | ⟨*num-for*⟩ | ⟨*iter-for*⟩ | ⟨*assign*⟩ | ⟨*return*⟩

⟨*assign*⟩ ::= ⟨*var*⟩ '`=`' ⟨*expr*⟩

⟨*return*⟩ ::= '`return`' ⟨*expr*⟩

⟨*do*⟩ ::= '`do`' ⟨*block*⟩ '`end`'

⟨*if*⟩ ::= '`if`' ⟨*expr*⟩ '`then`' ⟨*block*⟩ ⟨*else-ifs*⟩ ⟨*else*⟩ '`end`'

⟨*else-ifs*⟩ ::= '`elseif`' ⟨*expr*⟩ '`then`' ⟨*block*⟩ ⟨*else-ifs*⟩ | ''

⟨*else*⟩ ::= '`else`' ⟨*block*⟩ | ''

⟨*while*⟩ ::= '`while`' ⟨*expr*⟩ '`do`' ⟨*block*⟩ '`end`'

⟨*repeat*⟩ ::= '`repeat`' ⟨*block*⟩ '`until`' ⟨*expr*⟩

⟨*num-for*⟩ ::= '`for`' ⟨*var*⟩ '`=`' ⟨*expr*⟩ '`,`' ⟨*expr*⟩ ⟨*num-for-step*⟩ '`do`' ⟨*block*⟩ '`end`'

⟨*num-for-step*⟩ ::= '`,`' ⟨*expr*⟩ | ''

⟨*iter-for*⟩ ::= '`for`' ⟨*var*⟩ '`,`' ⟨*var*⟩ '`in`' ⟨*expr*⟩ '`do`' ⟨*block*⟩ '`end`'

⟨*thread*⟩ ::= '`coroutine.create(`' ⟨*expr*⟩ '`)`'

⟨*table-access*⟩ ::= ⟨*var*⟩ '`[`'⟨*expr*⟩ '`]=`' ⟨*expr*⟩

⟨*table*⟩ ::= '`{`' '`}`' | ...

This grammar is still incomplete.

### 2.1 Expected Problems

## 3 Description of the semantics

The normal rules will be worked out in the nearby future, as seen in appendix A. We will, however, start some of the harder semantical expressions, namely the thread and the table rules.

### 3.1 Approach

For this project we have chosen to apply axiomatic semantics. This is because this allows us to properly prove a theorem, instead of just simulating a piece of a program.

## 3.2 Concepts

We will mainly be explaining tables. We will expand this with coroutines, yielding and resuming, pipes and filters. (also functions need to be described)

### 3.2.1 States

We are going to use Axiomatic Semantics and therefore we do not have states. We use pre- and postconditions instead.

### 3.2.2 Transitions

Our transitions are of the form:
$\{P\}S\{Q\}$

### 3.2.3 Types

Lua does not have a strong type system. Whenever a value is called the value is passed and the user doesn't need to apply which type this value has. In this model we will restrict ourselves to Coroutines, Tables, Booleans, Strings and Integers as types.

| | | |
|---|---|---|
| Boolean | ::= | **true** \| **false** |
| Integer | ::= | 0 \| 1 \| 2 \| ... |
| String | ::= | <Char,String> \|$\lambda$ |
| Char | ::= | a \| ... \| z \| A \| ... \| Z |
| Table | ::= | Value $\rightarrow$ Value |
| Value | ::= | Boolean \| Integer \| String \| Char \| Table \| Coroutine |
| Coroutine | ::= | [to be continued] |

## 3.3 Expected Problems

We expect some problems with describing how values are passed on. For example, if we want to call to print an Integer, Lua provides us with the toString value for that Integer. We need to figure out a good method for describing how Lua does this.

# 4 Analysis

We will prove that what is printed is actually the thing that needs to be printed.

## 4.1 Example Code

```
-- from http://lua-users.org/wiki/CoroutinesTutorial

function odd(x)
  print('A: odd', x)
  coroutine.yield(x)
  print('B: odd', x)
end

function even(x)
  print('C: even', x)
```

```
  if x==2 then return x end
  print('D: even ', x)
end

co = coroutine.create(
  function (x)
    for i=1,x do
      if i==3 then coroutine.yield(-1) end
      if i % 2 == 0 then even(i) else odd(i) end
    end
  end)

count = 1
while coroutine.status(co) ~= 'dead' do
  print('----', count) ; count = count+1
  errorfree, value = coroutine.resume(co, 5)
  print('E: errorfree, value, status', errorfree, value, coroutine.status(co))
end
```

## 4.2 Elaboration

To be added.

# A Planning

| Deadline | Summary |
|---|---|
| 29 April | Submit first version project |
| ? | Apply feedback |
| 2 May | Finish Syntax |
| 9 May | Finish Semantics |
| 16 May | Choose final program for analysis |
| 23 May | Finish Introduction |
| 27 Mei | Submit second version project |
| ? | Apply feedback |
| 13 June | Finish Analysis |
| 17 June | Submit final version project |