PROLUA

Working Draft

Jeremy OTHIENO
https://github.com/supranove

Chapter I. Introduction

[TODO] Add cycle diagram: input (lua code) > lua2prolog > prolua chunk > prolua evaluation > prolua output (result and environment)

I.a. Why Lua?

[TODO] Why Lua? Simple, powerful yet expressive, well documented, hasn't been done, etc.

I.b. Why Prolog?

[TODO] Why Prolog? Power of expression, ease of use, etc.

I.c. Constraints

The interpreter will work for Lua 5.1 and below. This is because I'm not familiar with some of the changes made in later versions of Lua, most notably the change in how the environment is managed. I'll have to do a bit more research to be able to update Prolua.

Of the eight basic types of values in Lua, only only **numbers**, **strings**, **booleans**, **tables**, **functions** and **nil** will be implemented. The **userdata** and **thread** types will be excluded, as well as the features that depend on these types, such as coroutines.

No garbage collection.

I.d. Notation

[TODO] Explain notation and how to read it.

Chapter II. Syntax

Our first order of business is to define an abstract syntax specific to Prolua so that we can have a general idea of the **form** that a Prolua program will take. To do so, we'll need to analyse Lua's concrete syntax and come up with an abstract syntax of our own. Below is Lua's concrete syntax in EBNF

```
chunk ::= {stat [`;´]} [laststat [`;´]]

block ::= chunk

stat ::=
    varlist `=´ explist |
    functioncall |
    do block end |
    while exp do block end |
    repeat block until exp |
    if exp then block {elseif exp then block} [else block] end |
    for Name `=´ exp `,´ exp [`,´ exp] do block end |
    for namelist in explist do block end |
    function funcname funcbody |
    local function Name funcbody |
    local namelist [`=´ explist]

laststat ::= return [explist] | break

funcname ::= Name {‘.’ Name} [‘:’ Name]

varlist ::= var {‘,’ var}

var ::= Name | prefixexp ‘[’ exp ‘]’ | prefixexp ‘.’ Name

namelist ::= Name {‘,’ Name}

explist ::= exp {‘,’ exp}

exp ::=
    nil | false | true | Number | String | ‘...’ | function |
    prefixexp | tableconstructor | exp binop exp | unop exp

prefixexp ::= var | functioncall | ‘(’ exp ‘)’

functioncall ::=  prefixexp args | prefixexp ‘:’ Name args

args ::=  ‘(’ [explist] ‘)’ | tableconstructor | String

function ::= function funcbody

funcbody ::= ‘(’ [parlist] ‘)’ block end

parlist ::= namelist [‘,’ ‘...’] | ‘...’

tableconstructor ::= ‘{’ [fieldlist] ‘}’

fieldlist ::= field {fieldsep field} [fieldsep]

field ::= ‘[’ exp ‘]’ ‘=’ exp | Name ‘=’ exp | exp

fieldsep ::= ‘,’ | ‘;’

binop ::=
    ‘+’ | ‘-’ | ‘*’ | ‘/’ |
    ‘^’ | ‘%’ | ‘..’ | ‘<’ | ‘<=’ | ‘>’ | ‘>=’ | ‘==’ | ‘~=’ | and | or

unop ::= ‘-’ | not | ‘#’
```

II.a. Sets

Let ***Expression*** be the set of all possible expressions in Lua, and ***Explist*** a list of expressions such that

$$\text{expressionlist}: \frac{}{[] \in \text{Explist}} \qquad \text{explist}: \frac{e \in \text{Expression},\ \ es \in \text{Explist}}{e::es \in \text{Explist}}$$

Let ***Name*** be the set of all possible identifer names in Lua, and ***Namelist*** a list of names such that

$$\text{namelist}: \frac{}{[] \in \text{Namelist}} \qquad \text{namelist}: \frac{n \in \text{Name},\ \ ns \in \text{Namelist}}{n::ns \in \text{Namelist}}$$

Let ***ParName*** be the set of all possible parameter names and an extension of ***Name*** to include the name "**...**", then let ***Parlist*** be a list of parameter names such that

$$\text{parname}: \frac{}{\text{Parname} = \text{Name} \cup \{"..."\}} \qquad \text{parlist}: \frac{}{[] \in \text{Parlist}}$$

$$\text{parlist}: \frac{p \in \text{Parname},\ \ ps \in \text{Parlist}}{p::ps \in \text{Parlist}}$$

Let ***Variable***, a subset of ***Expression***, be the set of all possible variables in Lua, and ***Varlist*** a list of variables such that

$$\text{variablelist}: \frac{}{[] \in \text{Varlist}} \qquad \text{variablelist}: \frac{v \in \text{Variable},\ \ vs \in \text{Varlist}}{v::vs \in \text{Varlist}}$$

Let ***Value***, a subset of ***Expression***, be the set of all possible values in Lua, and ***Valuelist*** a list of values such that

$$\text{valuelist}: \frac{}{[] \in \text{Valuelist}} \qquad \text{valuelist}: \frac{v \in \text{Value},\ \ vs \in \text{Valuelist}}{v::vs \in \text{Valuelist}}$$

Let ***Statement*** be the set of all possible statements (instructions) in Lua, and ***Statementlist*** a list of statements such that

$$\text{statementlist}: \frac{}{[] \in \text{Statementlist}} \qquad \text{statementlist}: \frac{s \in \text{Statement},\ \ ss \in \text{Statementlist}}{s::ss \in \text{Statementlist}}$$

Let a ***Reference*** be a positive non-zero integer, and ***Referencestack*** a stack of references such that

$$\text{referencestack}: \frac{}{[] \in \text{ReferenceStack}}$$

$$\text{referencestack}: \frac{r \in \text{Reference},\ \ rs \in \text{ReferenceStack}}{r::rs \in \text{ReferenceStack}}$$

II.b. Values and Types

Nil is a type of value whose main property is to be different from any other value; it usually represents the absence of a useful value

$$\text{value}: \frac{v \in \{\text{nil}\}}{\text{niltype}(v) \in \text{Value}}$$

Boolean values are defined as **false** and **true**.

$$\text{value}: \frac{v \in \{\text{false}, \text{true}\}}{\text{booleantype}(v) \in \text{Value}}$$

Number represents **real** numbers.

$$\text{value}: \frac{v \in \mathbb{R}}{\text{numbertype}(v) \in \text{Value}}$$

A **string** represents arrays of 8-bit characters. There's no **character** type in Lua but to be able to define the syntax of a string, we need to define what a character is. Unfortunately, the character set is too large to enumerate so we'll simplify by supposing that it's the set of all 8-bit ASCII characters. A string is then considered to be a concatenation of characters

$$\text{string}: \frac{}{[] \in \text{String}} \qquad \text{string}: \frac{c \in \text{Character},\ \ s \in \text{String}}{c::s \in \text{String}} \qquad \text{value}: \frac{s \in \text{String}}{\text{stringtype}(s) \in \text{Value}}$$

The type **table** implements associative arrays, i.e. arrays that can be indexed with any value except nil. Tables can contain values of all types including **nil**, in which case the table field is deleted

$$\text{table}: \frac{}{[] \in \text{Table}} \qquad \text{table}: \frac{k \in \text{Expression} \setminus \{\text{niltype}(\text{nil})\},\ \ v \in \text{Expression},\ \ t \in \text{Table}}{\langle k, v \rangle ::t \in \text{Table}}$$

$$\text{value}: \frac{t \in \text{Table}}{\text{tabletype}(t) \in \text{Value}}$$

In Lua, tables are not passed by value but by **reference**: a positive non-zero integer that indexes a table's position in the environment

$$\text{value}: \frac{n \in \mathbb{N}_+^*}{\text{referencetype}(n) \in \text{Value}}$$

Functions are defined syntactically as

$$\text{value}: \frac{ps \in \text{Parlist},\ \ ss \in \text{Statementlist},\ \ rs \in \text{Referencestack}}{\text{functiontype}(ps, ss, rs) \in \text{Value}}$$

II.c. Expressions

Variables store values. To be able to retrieve a value of a variable, we need the variable's name

$$\text{expression}: \frac{n \in \text{Name}}{\text{variable}(n) \in \text{Expression}}$$

We define the table **index** expression which, like the variable expression, retrieves the value for a given key in a given table

$$\text{expression}: \frac{t, k \in \text{Expression}}{\text{index}(t, k) \in \text{Expression}}$$

Lua defines a **variadic expression** represented by three dots "**...**" which is a placeholder for a list of values.

$$\text{expression}: \frac{}{\text{"..."} \in \text{Expression}}$$

Calling a **unary operator** requires the operator's name and the expression to be evaluated

$$\text{expression}: \frac{n \in \{\text{negative, not, length}\}, e \in \text{Expression}}{\text{unop}(n, e) \in \text{Expression}}$$

Almost like a unary operator, calling **binary operators** requires the name of the operator and two expressions to be evaluated

$$\text{expression}: \frac{n \in \{\text{add, subtract, multiply, divide, modulo, exponent}\}, \quad e_{lhs}, e_{rhs} \in \text{Expression}}{\text{binop}(n, e_{lhs}, e_{rhs}) \in \text{Expression}}$$

$$\text{expression}: \frac{n \in \{\text{equal, lt, le, gt, ge, and, or, concatenate}\}, \quad e_{lhs}, e_{rhs} \in \text{Expression}}{\text{binop}(n, e_{lhs}, e_{rhs}) \in \text{Expression}}$$

Function definitions are defined as

$$\text{expression}: \frac{ps \in \text{Parlist}, \quad ss \in \text{Statementlist}}{\text{function}(ps, ss) \in \text{Expression}}$$

Function calls are defined as

$$\text{expression}: \frac{e \in \text{Expression}, \quad es \in \text{Explist}}{\text{functioncall}(e, es) \in \text{Expression}}$$

II.d. Statements

The unit of execution in Lua, and therefore Prolua, is called a **chunk** which is simply a sequence of statements that are executed sequentially. Lua handles a chunk as the body of an anonymous function with a variable number of arguments, and the same is done in Prolua

$$\text{chunk}: \frac{\text{ss} \in \text{Statementlist}}{\text{chunk(ss)}}$$

Lua allows multiple **assignments**. Therefore, the syntax for assignment defines a list of variables on the left side and a list of expressions on the right. Syntactically, the assignment operator takes two lists of expressions which it will evaluate into appropriate values

$$\text{statement}: \frac{es_{lhs}, es_{rhs} \in \text{Explist}}{\text{assign}(es_{lhs}, es_{rhs}) \in \text{Statement}}$$

Function calls were previously defined as expressions but can also be executed as statements, in which case all return values are thrown away

$$\text{statement}: \frac{e \in \text{Expression}, \quad es \in \text{Explist}}{\text{functioncall}(e, es) \in \text{Statement}}$$

The **do** statement allows us to explicitly delimit a block of statements to produce a single statement

$$\text{statement}: \frac{ss \in \text{Statementlist}}{\text{do}(ss) \in \text{Statement}}$$

A **while** loop is defined as

$$\text{statement}: \frac{e \in \text{Expression}, \quad s \in \text{Statement}}{\text{while}(e, s) \in \text{Statement}}$$

A **repeat-until** loop is similar to a while loop

$$\text{statement}: \frac{e \in \text{Expression}, \quad s \in \text{Statement}}{\text{repeat}(e, s) \in \text{Statement}}$$

An **if-else** condition takes an expression that it evaluates into a boolean value, a statement that will be executed if the aforementioned value is true, and another if it is false

$$\text{statement}: \frac{e \in \text{Expression}, \quad s_{true}, s_{false} \in \text{Statement}}{\text{if}(e, s_{true}, s_{false}) \in \text{Statement}}$$

We can then define an **if** condition (without the **else**) as "if(e, s_{true}, do([]))", an **if-elseif** condition recursively as "if(e, s_{true}, $\text{if}_i(e_i, s_{truei}, \text{do}([]))$)", and finally an **if-elseif-else** condition as "if(e, s_{true}, $\text{if}_i(e_i, s_{truei}, s_{false})$)".

The **numeric for** loop is defined syntactically as

$$\text{statement}: \frac{n \in \text{Name}, \quad e_{initial}, e_{end}, e_{increment} \in \text{Expression}, \quad s \in \text{Statement}}{\text{for}(n, e_{initial}, e_{end}, e_{increment}, s) \in \text{Statement}}$$

while the **generic for** loop is defined as

$$\text{statement}: \frac{ns \in \text{Namelist}, \quad es \in \text{Explist}, \quad s \in \text{Statement}}{\text{for}(ns, es, s) \in \text{Statement}}$$

Declaring explicitly local variables requires the name of the variable as well as a value that will be assigned to the variable. If no value is specified, then **nil** is implied.

$$\text{statement}: \frac{n \in \text{Name}, \quad v \in \text{Expression}}{\text{localvariable}(n, v) \in \text{Statement}}$$

The **return** statement returns one or more values from a function or a chunk

$$\text{statement}: \frac{es \in \text{Expressionlist}}{\text{return}(es) \in \text{Statement}}$$

The **break** statement breaks the flow of a loop

$$\text{statement}: \frac{}{\text{break} \in \text{Statement}}$$

Now consider the following Lua program[1] and its generated Prolua counterpart

```
function toCelsius(fahrenheit)
   return (fahrenheit - 32)*(5 / 9);
end;

t = {min = 0, 0, 0, 0, max = 0}

t.min = toCelsius(5);

local i = 1;

while (i < 4) do
   t[i] = toCelsius(5^(i + 1));
   i = i + 1;
end;

t.max = toCelsius(5^5);

return t.min, t[1], t[2], t[3], t.max;
```

```
chunk([
assign([variable('toCelsius')], [functionbody(['fahrenheit'],
[return([binop(multiply, binop(subtract, variable('fahrenheit'),
numbertype(32)), binop(divide, numbertype(5), numbertype(9)))])])]),

assign([variable('t')], [tabletype([[stringtype('min'), numbertype(0)],
[numbertype(1), numbertype(0)], [numbertype(2), numbertype(0)], [numbertype(3),
numbertype(0)], [stringtype('max'), numbertype(0)]])]),

assign([access(variable('t'), stringtype('min'))],
[functioncall(variable('toCelsius'), [numbertype(5)])]),

localvariable('i', numbertype(1)),

while(binop(lt, variable('i'), numbertype(4)),
do([assign([access(variable('t'), variable('i'))],
[functioncall(variable('toCelsius'), [binop(exponent, numbertype(5), binop(add,
variable('i'), numbertype(1)))])]), assign([variable('i')], [binop(add,
variable('i'), numbertype(1))])])),

assign([access(variable('t'), stringtype('max'))],
[functioncall(variable('toCelsius'), [binop(exponent, numbertype(5),
numbertype(5))])]),

return([access(variable('t'), stringtype('min')), access(variable('t'),
numbertype(1)), access(variable('t'), numbertype(2)), access(variable('t'),
numbertype(3)), access(variable('t'), stringtype('max'))])
]).
```

We can see that the output is essentially a Prolog program with predicates that resemble the previously defined abstract syntax.

1 This is the **temperature.lua** program provided in the samples.

Chapter III. Semantics

As I mentioned before, the syntax dictates the form of a Prolua program, but says nothing about its behavior.

III.a. The environment

<ts, is, make(keys, values)> $=>_{env}$ <t::ts, i::is>
<ts, gettable(reference)> $=>_{env}$ Table
<ts, getvalue(reference, key)> $=>_{env}$ Value
<ts, setvalue(reference, key, value)> $=>_{env}$ ts_1
<ts, keyexists(reference, key)> $=>_{env}$ {true, false}

III.b. Expression evaluation

When evaluating expressions, we should take note that the evaluation can result in left and right values. Take for example the following assignment

$$x = 3$$

The variable x should evaluate into a memory address where the value '3' will be stored. The variable is also known as a left value and the value '3' is known as a right value.

By this train of thought, only two of the defined expressions can be left-hand expressions:
variable
index

All expressions are right-hand expressions, even if they don't return results.

Evaluating a left-hand side **variable** means retrieving its memory address, i.e. the table and key that corresponds to the variable. If the variable name cannot be found in a given scope, a query is made in the outer scope. If the name is not found yet again, then the process is repeated until we reach the global scope, at which point we return the reference to the global scope and the name, even if the variable does not exist in it.

If a query is made in the global environment, always return the reference to said environment and the name of the variable, even if the variable does not exist in the environment

$$\frac{}{\langle E,\ r_1::[],\ \text{variable}(n)\rangle \overset{\text{lhs}}{\Rightarrow} \langle E,\ \langle r_1,\ n\rangle\rangle}$$

If the variable name exists in a given environment, return the reference to that environment and the name of the variable

$$\frac{\langle E,\ \text{keyexists}(r_i,\ n)\rangle \overset{\text{env}}{\Rightarrow} \text{true}}{\langle E,\ r_i::R,\ \text{variable}(n)\rangle \overset{\text{lhs}}{\Rightarrow} \langle E,\ \langle r_i,\ n\rangle\rangle}$$

If the variable name cannot be found in the given scope, a query is made in the outer scope. This process is repeated until the variable name is found or we reach the global scope

$$\frac{\begin{array}{c}\langle E,\ \text{keyexists}(r_i,\ n)\rangle \overset{\text{env}}{\Rightarrow} \text{false} \\ \langle E,\ R,\ \text{variable}(n)\rangle \overset{\text{lhs}}{\Rightarrow} \langle E,\ \langle r_k,\ n\rangle\rangle \quad \forall\ i, k \in \mathbb{N} \text{ st. } 1 \leq k < i\end{array}}{\langle E,\ r_i::R,\ \text{variable}(n)\rangle \overset{\text{lhs}}{\Rightarrow} \langle E,\ \langle r_k,\ n\rangle\rangle}$$

On the other hand, evaluating a right-hand side variable means retrieving its value. If the variable does not exist, then **nil** is returned.

$$\frac{\langle E,\ \text{getvalue}(r_1,\ n)\rangle \overset{\text{env}}{\Rightarrow} V}{\langle E,\ r_1::[],\ \text{variable}(n)\rangle \overset{\text{rhs}}{\Rightarrow} \langle E,\ V\rangle}$$

If the variable exists in the given scope, then its value is returned

$$\frac{\langle E,\ \text{keyexists}(r_i,\ n)\rangle \overset{\text{env}}{\Rightarrow} \text{true} \quad \langle E,\ \text{getvalue}(r_i,\ n)\rangle \overset{\text{env}}{\Rightarrow} V}{\langle E,\ r_i::R,\ \text{variable}(n)\rangle \overset{\text{rhs}}{\Rightarrow} \langle E,\ V\rangle}$$

If the variable does not exist in the given scope, then we check the outer scope

$$\frac{\langle E,\ \text{keyexists}(r_i,\ n)\rangle \overset{\text{env}}{\Rightarrow} \text{false} \quad \langle E,\ R,\ \text{variable}(n)\rangle \overset{\text{rhs}}{\Rightarrow} \langle E,\ V\rangle}{\langle E,\ r_i::R,\ \text{variable}(n)\rangle \overset{\text{rhs}}{\Rightarrow} \langle E,\ V\rangle}$$

Accessing a value in a table via an **index**

Variadic expressions evaluate into a list of values if they exist. If the name "**...**" is defined in the current scope, then we return its list of values

$$\frac{\langle E,\ \text{keyexists}(r_i,\ \text{"..."})\rangle \overset{\text{env}}{\Rightarrow} \text{true} \quad \langle E,\ \text{getvalue}(r_i,\ \text{"..."})\rangle \overset{\text{env}}{\Rightarrow} V}{\langle E,\ r_i::R,\ \text{"..."}\rangle \overset{\text{rhs}}{\Rightarrow} \langle E,\ V\rangle}$$

But if it does not exist, then the current function is not a vararg function, and this is an error

$$\frac{\langle E,\ \text{keyexists}(r_i,\ \text{"..."})\rangle \overset{\text{env}}{\Rightarrow} \text{false}}{\langle E,\ r_i::R,\ \text{"..."}\rangle \overset{\text{rhs}}{\Rightarrow} \langle E,\ \textbf{ERROR}\rangle}$$

Unary operators
Binary operators

When a **function is defined**, it inherits the environment of the function that created it

$$\frac{}{\langle \text{ETS},\ R,\ \text{function}(ps,\ ss)\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS},\ \text{functiontype}(ps,\ ss,\ R)\rangle}$$

On the other hand, a **function call** creates a new scope in which it will evaluate a function body. The evaluation returns a value which can be an error

$$\frac{\langle \text{ETS}, \text{R}, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{functiontype(ns, ss, R)}\rangle \quad \langle \text{ETS}_1, \text{R}, \text{es}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \text{vs}\rangle \quad \langle \text{ETS}_2, \text{R}, \text{make(ns, vs)}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_3, \text{R}_1\rangle \quad \langle \text{ETS}_3, \text{R}_1, \text{ss}\rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_4, \text{C}, \text{V}\rangle \text{ st C} \in \{\text{return, error}\}}{\langle \text{ETS}, \text{R}, \text{functioncall(e, es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_4, \text{V}\rangle}$$

A function doesn't necessarily return a value

$$\frac{\langle \text{ETS}, \text{R}, \text{e}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_1, \text{functiontype(ns, ss, R)}\rangle \quad \langle \text{ETS}_1, \text{R}, \text{es}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_2, \text{vs}\rangle \quad \langle \text{ETS}_2, \text{R}, \text{make(ns, vs)}\rangle \overset{\text{env}}{\Rightarrow} \langle \text{ETS}_3, \text{R}_1\rangle \quad \langle \text{ETS}_3, \text{R}_1, \text{ss}\rangle \overset{\text{evaluate}}{\Rightarrow} \langle \text{ETS}_4, \text{continue}, []\rangle}{\langle \text{ETS}, \text{R}, \text{functioncall(e, es)}\rangle \overset{\text{rhs}}{\Rightarrow} \langle \text{ETS}_4, []\rangle}$$

III.c. Statement evaluation

Evaluating a statement requires an environment, a reference to the current scope and the statement to evaluate. It returns the modified environment, a control value which signals whether or not to continue execution, and possibly a return value.

Controls are return, break, continue, error.

Before assigning variables, we need to ???

If there's an error while evaluating the left-hand expressions, propagate it

$$\frac{\langle E, R, es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle E_1, R, \textbf{ERROR}\rangle}{\langle E, R, assign(es_{lhs}, es_{rhs})\rangle \overset{evaluate}{\Rightarrow} \langle E_1, error, \textbf{ERROR}\rangle}$$

Similarly, if there's an error while evaluating the right-hand expressions, do the same

$$\frac{\langle E, R, es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle E_1, R, vs_{lhs}\rangle \quad \langle E_1, R, es_{rhs}\rangle \overset{rhs}{\Rightarrow} \langle E_2, R, \textbf{ERROR}\rangle}{\langle E, R, assign(es_{lhs}, es_{rhs})\rangle \overset{evaluate}{\Rightarrow} \langle E_2, error, \textbf{ERROR}\rangle}$$

If no errors occured, then assign the values to the variables

$$\frac{\langle E, R, es_{lhs}\rangle \overset{lhs}{\Rightarrow} \langle E_1, R, vs_{lhs}\rangle \quad \langle E_1, R, es_{rhs}\rangle \overset{rhs}{\Rightarrow} \langle E_2, R, vs_{rhs}\rangle \quad \langle E_2, setvalues(vs_{lhs}, vs_{rhs})\rangle \overset{env}{\Rightarrow} \langle E_3\rangle}{\langle E, R, assign(es_{lhs}, es_{rhs})\rangle \overset{evaluate}{\Rightarrow} \langle E_3, continue, []\rangle}$$

Function calls can be statements too in which case all return values are discarded

$$\frac{\langle ETS, R, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, functiontype(ns, ss, R)\rangle \quad \langle ETS_1, R, es\rangle \overset{rhs}{\Rightarrow} \langle ETS_2, vs\rangle \quad \langle ETS_2, R, make(ns, vs)\rangle \overset{env}{\Rightarrow} \langle ETS_3, R_1\rangle \quad \langle ETS_3, R_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle ETS_4, C, V\rangle \text{ st } C \in \{return, continue, break\}}{\langle ETS, R, functioncall(e, es)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_4, continue, []\rangle}$$

except when an error occurs

$$\frac{\langle ETS, R, e\rangle \overset{rhs}{\Rightarrow} \langle ETS_1, functiontype(ns, ss, R)\rangle \quad \langle ETS_1, R, es\rangle \overset{rhs}{\Rightarrow} \langle ETS_2, vs\rangle \quad \langle ETS_2, R, make(ns, vs)\rangle \overset{env}{\Rightarrow} \langle ETS_3, R_1\rangle \quad \langle ETS_3, R_1, ss\rangle \overset{evaluate}{\Rightarrow} \langle ETS_4, error, \textbf{ERROR}\rangle}{\langle ETS, R, functioncall(e, es)\rangle \overset{evaluate}{\Rightarrow} \langle ETS_4, error, \textbf{ERROR}\rangle}$$

The **break** statement simply breaks a loop and does not return any values or modify the environment. In terms of its evaluation, all it does is return the break control.

$$\frac{}{\langle E, R, \text{break} \rangle \overset{\text{evaluate}}{\Rightarrow} \langle E, \text{break}, [] \rangle}$$