Formalización de la semántica del lenguaje de programación Lua

Tesista: Mallku Ernesto Soldevila Raffa.
Director: Dr. Daniel Lima Ventura.

Índice

1

Resumen

Los lenguajes de “script” cumplen hoy un papel importante en el desarrollo de aplicaciones, ofreciendo conceptos para la programación, que pueden extender aquellos presentes en el lenguaje empleado para implementar la arquitectura de una aplicación. En ese sentido, Lua es un lenguaje pensado para ser utilizado como una herramienta para la programación, de propósito específico, para implementar código a ejecutarse embebido en una aplicación anfitrión. Presenta conceptos útiles para el desarrollo ágil de código, como tipado dinámico y gestión automática de memoria, como también mecanismos de reflexión, que permiten adaptarlo a dominios específicos.

Actualmente, existe un interés en disponer de un modelo formal del lenguaje, que permita el desarrollo de herramientas para verificar propiedades sobre programas escritos en Lua. En el presente trabajo, proponemos una semántica operacional para un subconjunto de los conceptos del lenguaje Lua y los servicios ofrecidos por su librería estándar. A su vez, empleando la herramienta PLT Redex, realizamos una transcripción del modelo obtenido y efectuamos pruebas de conformidad del mismo con respecto al intérprete de Lua, versión 5.2, empleando su propia suite de pruebas.

2

Abstract

Script languages have an important role today in the development of applications, offering concepts for programming, that can extend those present in the language used for the implementation of the application's architecture. In that matter, Lua is a language developed to be used as a tool for programming code of special purpose, to be executed embedded into a host application. It presents concepts that are useful for agile development of code, as dynamic typing and automatic memory management, and also reflection mechanisms, that bring the possibility of adapt the language to specific domains.

Currently, there is interest in having a formal model of the language, that allows the development of tools for verification of properties of programs written in Lua. In the present work, we propose an operational semantics for a subset of the concepts of Lua and the services offered by its standard library. Also, using the tool PLT Redex, we transcribe the obtained model and run tests of conformity of it, against Lua's interpreter, version 5.2, using its own test suite.

3

Introducción al presente trabajo

3.1 Motivación

Los lenguajes de "script" cumplen hoy un papel importante en el desarrollo de aplicaciones, introduciendo elementos dinámicos y siendo utilizados en la orquestación de las funcionalidades de tales aplicaciones. Lua es un lenguaje que se viene destacando en este sentido debido a la posibilidad que ofrece de extender y modificar su semántica, como también a lo compacto que es el lenguaje. En este trabajo deseamos formalizar y estudiar una semántica operacional para el lenguaje.

Para los lenguajes de programación, una formalización de su semántica puede ser empleada para garantizar que una implementación coincida con una especificación. Por ejemplo, proyectos como Compcert[1] y Concurrent C Minor Project[2] en donde se realizan verificaciones formales de compiladores reales, trabajando sobre la definición de la semántica del lenguaje fuente (lenguaje para el cual se construye el compilador), el intermediario (empleando internamente en el proceso de compilación) y el lenguaje objetivo (lenguaje hacia el cual se compila el programa en el lenguaje fuente).

3.1.1 Sobre Lua

Lua es un lenguaje de script desarrollado con el objetivo de ser extensible y que sirva como lenguaje de extensión [13]. El lenguaje fue pensado para ser compacto, con mecanismos que faciliten su uso con C y que pueda ser fácilmente embebido en las aplicaciones [14]. Lua es un lenguaje de extensión en el sentido de que va a proveer características como tipos dinámicos e mecanismos de reflexión que pueden ser utilizados para facilitar la programación de una aplicación. La habilidad de Lua de ser extensible significa que posee una serie de mecanismos que permiten modificar la semántica de algunas operaciones, para adaptar el lenguaje a distintos paradigmas de programación. A través del mecanismo provisto por las metatablas, las cuales definen el comportamiento de las operaciones que es posible de realizar sobre valores de cualquier tipo, es posible modelar otros paradigmas de programación.

Como ejemplo de las posibilidades que ofrece Lua para ser adaptado a dominios específicos, y a modo de introducción de la sintáxis y las posibilidades del lenguaje, veamos cómo es posible emplear el mecanismo de metatablas, junto a las tablas de Lua, para implementar la noción de clases e instancias, del paradigma orientado a objetos[3]. El empleo de tablas para representar objetos y clases resulta posible gracias a que todos los valores en Lua son de primera clase, permitiendo almacenar con una tabla tanto datos como procedimientos. La sintaxis del lenguaje es intuitiva y adecuada para este ejemplo, ya que provee notación extraída de lenguajes orientados a objetos:

```
local MyClass = {}    tabla que representará a la clase, y, a su vez, funcionará
                      de metatabla para las instancias
MyClass.__index = MyClass    intentos de indexado fallidos en instancias, deben remitirse
    a la tabla
                             que representa la clase, para obtener los métodos
```

[1] http://compcert.inria.fr/
[2] http://www.cs.princeton.edu/ appel/cminor/
[3] extraído de la wiki de Lua 5.2: http://lua-users.org/wiki/ObjectOrientationTutorial

```
azúcar sintactico equivalente a "MyClass.new = function..."
function MyClass.new(init)
  setmetatable define a MyClass como la metatabla de la tabla vacía {}, y
  retorna esta tabla como resultado de la llamada
  local self = setmetatable({}, MyClass)

  self.value = init
  return self
end

function MyClass.set_value(self, newval)
  self.value = newval
end

function MyClass.get_value(self)
  return self.value
end

local i = MyClass.new(5)
tbl:name(arg) es una abreviación para tbl.name(tbl, arg), salvo que tbl es evaluado una
  sola
vez
print(i:get_value())    > 5
i:set_value(6)
print(i:get_value())    > 6
```

Notar que, en el procedimiento MyClass.new (el constructor), la llamada al servicio interno setmetatable, define a MyClass como la meta-tabla del objeto recién creado, el cual, inicialmente, no es más que una tabla vacía (indicado por la notación {}). Para operaciones con el objeto que no se correspondan con lo provisto por el lenguaje, se recurre a consultar a su meta-tabla, indexando, para ello, ciertos campos con claves especiales. En particular, cuando, por ejemplo, indexamos una tabla con una clave inexistente, se procede a obtener su meta-tabla, y a consultar el campo de clave "__index". Si tal campo está definido en la meta-tabla, y contiene un procedimiento u otra tabla, se procede a emplearlos para resolver esta situación especial. En el código anterior, la línea MyClass.__index = MyClass (arriba de la definición del constructor) indica que, en caso de un intento de indexado de un objeto (que tenga a MyClass como meta-tabla) con una clave inexistente, se repetirá la misma operación de indexado sobre la tabla MyClass. Esto nos permite almacenar en esta tabla todos los métodos de clase, y delegar a esta tabla todos los intentos de llamadas a métodos, para un objeto dado.
Volviendo al análisis del constructor, a continuación de su primer línea, se define un atributo del objeto, mediante la asignación self.value = init, que agrega, a la tabla que representa al objeto, un campo de clave value y valor init. Luego del constructos, se definen 2 métodos de clase: set_value y get_value, que realizan lo esperado. Finalmente se crea una instancia, i, y se intenta llamar para la misma a sus métodos get_value y set_value. Por lo mencionado anteriormente, tales llamadas son delegadas a su meta-tabla, quien efectivamente tiene almacenada estos métodos.

Listamos, a continuación, las características de Lua 5.2. Para mayor información, referirse a [17]:

- Tipos de datos:
  - 8 tipos básicos: booleanos, nil (tipo especial con un único valor, **nil**, que es usado para representar la ausencia de información, en contextos en donde algún dato es requerido), números (de punto-flotante de doble precisión), strings (que representan secuencias inmutables de bytes), funciones, user-data (que representan datos C arbitrarios; permite entonces almacenar datos de C en variables de Lua), hilos (representan hilos de ejecución independiente, y son usados para implementar la noción de co-rutina: hilos que suspenden su ejecución únicamente mediante el llamado explícito de una función para ceder tiempo de ejecución a otros hilos) y tablas (arreglos asociativos).
  - Todos los tipos son de primera clase.

  - Funciones, hilos, tablas y user-data son manejados mediante referencias (las variables en realidad no almacenan estos valores, solo hacen referencia a ellos).
  - Lenguaje dinámicamente tipado.

- 3 tipos de variables: globales, locales, y campos de tablas:
  - Un identificador de variable se asume que se refiere a una variable global, a menos que se indique lo contrario. Toda variable global es considerada como un campo de una tabla especial, referida mediante el identificador _ENV. Es decir, internamente, una variable global es un caso particular de campo de tabla.
  - La definición de variables locales tienen alcance estático

  .

- Sentencias: presenta un repertorio usual de sentencias, típicas en lenguajes que soportan programación procedural. Entre ellas:
  - Concatenación de sentencias.
  - Asignación múltiple de variables.
  - Estructuras de control: condicional, bucles *while*, *repeat* y *for*.
  - Abstracción procedural.
  - Introducción de variables locales.

- Expresiones:
  - Operadores sobre números: +, -, *, /, % (módulo), ^ (exponenciación) y negación matemática (-).
  - Comparación: ==, ~= (distintos), <, <=, >, >=.
  - Operadores lógicos: **and**, **not** y **or**.
  - Operación sobre strings: concatenación de cadenas, longitud de cadenas.
  - Operaciones con tablas: longitud de tablas, constructor de tablas.
  - Definición de procedimientos (anónimos), y llamadas a procedimientos.

- Administración de memoria automática, mediante un recolector incremental "mark-and-sweep".

- Manejo de errores: provee mecanismos para generar explícitamente errores y atraparlos (a través de la posibilidad de ejecutar procedimientos en lo que se denomina "modo protegido"). Los mensajes de error pueden contener información sobre la traza de ejecución y el entorno.

- Mecanismo de meta-tablas: todo valor en Lua puede tener una meta-tabla, la cual es una tabla Lua ordinaria, que define el comportamiento del valor original, bajo circunstancias especiales. Tablas y userdata pueden tener meta-tablas individuales. Valores de los restantes tipos comparten meta-tablas, una por tipo. Las situaciones que pueden manejarse con el mecanismo de meta-tablas, incluyen:
  - Operaciones aritméticas: cuando el valor en cuestión es utilizado como operando de una operación aritmética, la cual no está definida usualmente para ese valor.
  - Operaciones con strings: cuando un valor diferente a una cadena es utilizado en lugar de la misma.
  - Comparaciones entre valores: para valores que no están naturalmente relacionados, es posible definir un procedimiento que indique cómo compararlos.
  - Operaciones con tablas: indexado de tablas con claves no existentes, asignación a tablas con claves no existentes.
  - Llamadas a procedimientos: operación de llamada a procedimiento sobre un valor que no es una función.

Lua viene siendo utilizado en una gran variedad de aplicaciones como Adobe Lightroom (40 % escrito en Lua[4]), en el *middleware* de la TV digital de Brasil (Ginga[5]), aparte de ser empleado en el área de desarrollo de juegos (por ejemplo, Grim Fandango[6]).

3.2 Formalización y trabajos relacionados

Los métodos formales todavía son asociadas a técnicas empleadas exclusivamente en la especificación de hardware, e.g. [9, 5], y en el desarrollo de software para sistemas críticos, e.g. [2]. El empleo de métodos formales en el desarrollo de sistemas en donde las fallas eventuales del mismo no representan un costo considerable, financiero o de vidas humanas, tiende a ser descartado por ser considerado costoso. Pero el desarrollo de herramientas asociadas al razonamiento automatizado (automated reasoning) permitió que la especificación formal y su verificación, pudiesen ser empleadas de una manera más amplia en el desarrollo de software, desde su modelado hasta la posible extracción de código a partir de esa formalización[7].

Así, asistentes de prueba como Coq [21] y PVS [18] tienen, aparte de su empleo en investigación académica como la aplicación en proyectos de lenguajes de programación[8], aplicación en la industria, por ejemplo en la certificación de Java Card EAL7[9]. Por lo tanto, la incorporación de la especifación formal en tales sistemas es esencial.

Guha et al., presentaron en [1] una formalización de JavaScript, utlizando PLT Redex [10], en donde la teoría de reescritura [4] es el fundamento teórico. En esa formalización, se presenta un núcleo del lenguaje, denominado λjs, que posee las características de ser reducido, pero aún así puede expresar todas las construcciones de JavaScript. Para λjs se define una semántica operacional *small-step* de Felleisen-Hieb con contextos de evaluación [10]. De esta forma se obtuvo una formalización de JavaScript más concisa en relación a el estándar [10] (de 200 páginas) y la derivación directa presenta en [19]. Inclusive la formalización a través de λjs es más completa, ya que apenas el operador *eval* fue excluido.

3.3 Propuesta de trabajo

Se propone utilizar el enfoque presentado en [1] para obtener una formalización del lenguaje Lua. A pesar de las semejanzas entre algunas construcciones de JavaScript y de Lua, tenemos en el segundo características que tienen una semántica distinta o no están presentes en el primero, como el mencionado mecanismo meta-tablas y co-rutinas. Las tareas que se van a realizar incluyen:

- Definición de un lenguaje que incluya los mecanismos básicos de Lua y formalización de una semántica operacional *small-step* del mismo.
- Mecanización de la semántica operacional del lenguaje, empleando la herramienta PLT Redex.
- Definición e implementación de una estrategia de compilación de programas en Lua a programas en nuestro lenguaje. A través de esta relación entre Lua y nuestro lenguaje es que se le dará significado a las construcciones de Lua.
- Elaboración de testeos de conformidad de la semántica de nuestro lenguaje con respecto al intérprete de Lua, versión 5.2. Vale aclarar que Lua posee un único intérprete.

3.4 Sobre semántica de reducciones y su mecanización con PLT Redex

Para describir la semántica de Lua, seguimos el enfoque propuesto por Landin en [8], para formalizar el significado de las construcciones de ALGOL 60: establecer una correspondencia entre

[4] es.wikipedia.org/wiki/Adobe_Photoshop_Lightroom

[5] www.ginga.org.br/es

[6] www.grimfandango.net/?page=articles&pagenumber=2

[7] La correspondencia proofs-as-programs, también conocida por propositions-as-types, que permite la extración de código correcto a partir de la especificación formal, es denominada Isomorfismo de Curry-Howard [20]

[8] Ver coq.inria.fr/cocorico/List of Coq PL Projects

[9] www.gemalto.com/php/pr_view.php?id=23

[10] www.ecma-international.org/publications/standards/Ecma-262.htm

este y un lenguaje más simple, que posea una semántica formalizable de un modo razonable (en este caso en particular, una semántica operacional). De esta forma, se sigue el paradigma científico consistente en explicar un fenómeno complejo (la semántica del lenguaje original) como resultado de la interacción de fenómenos más simples (la semántica de un lenguaje más simple, pero que alcanza a cubrir todos los aspectos del original). Este enfoque toma forma actual en trabajos como [1], [11] o [12], los cuales se asemejan tanto en el estilo de semántica operacional empleada (de Felleisen-Hieb [7]), como en la utilización de herramientas para automatizar el razonamiento sobre la semántica definida (mediante PLT Redex [10]).

La notación de Felleisen-Hieb permite describir una semántica operacional *small-step*, en donde toda la información necesaria para especificar el significado de un programa está incrustada en el mismo texto que describe al programa: no tenemos configuraciones en donde podemos reconocer, como piezas separadas, al código de control que describe al programa, al entorno, el almacenamiento y, por si fuera necesario, las continuaciones, sino que todas estas piezas han sido incorporadas al texto mismo que describe al programa, para describir una semántica basada puramente en la sintaxis del lenguaje. Los recursos de esta notación surgen con la intención de poder definir un cálculo para programas imperativos, es decir, una relación de equivalencia entre programas imperativos, que admita razonar sobre equivalencias entre los mismos, basándose puramente en el texto que describe a los programas. Concretamente, la noción de continuaciones ha sido reemplazada por la noción de "contextos": la porción de código que rodea a la sentencia que se va a reducir, la cual tiene una conexión natural con la idea de "continuación". El entorno se manipula ahora mediante un mecanismo de substitución de identificadores de variable por su significado (referencias o valores). La representación de almacenamientos ha sufrido diferentes transformaciones, desde el uso de un mecanismo similar al empleado para manejar entorno (es decir, incrustando la información del almacenamiento en el programa mismo, como en [6]), la incorporación del mismo como una sentencia más del lenguaje (interpretando al almacenamiento como una instrucción que realiza declaración y asignación de variables, como en [10]), o su descripción de forma separada al código del programa (como en [1] y [11]).

La semántica en sí es definida mediante relaciones entre estos programas enriquecidos con información para hacer posible la descripción de su semántica. Las relaciones, a su vez, son capturadas empleando reglas de inferencia. Para evitar confusión con respecto al uso usual de reglas de inferencia en otras áreas, vale la pena mencionar que, como es costumbre en la literatura sobre semántica operacional de Felleisen-Hieb, tanto el antecedente de una regla como cualquier otro predicado que restrinja a la relación que se quiere definir, se definen en el lugar del antecedente de la regla. Esto quiere decir que, en el antecedente de una regla de inferencia, podemos encontrarnos no solamente con texto que se refiere al cumplimiento de una relación entre 2 programas, sino que puede ocurrir cualquier otro predicado, como en:

$$\frac{\delta(==, sv_1, sv_2) = \textbf{true}}{(sv_1 \;==\; sv_2) \;\rightarrow^e\; \textbf{true}}\ \text{E-EqualitySuccess}$$

Regla que establece que la noción de reducción $\rightarrow^e$ relaciona a la frase $(sv_1 \;==\; sv_2)$ con **true**, de cumplirse el predicado $\delta(==, sv_1, sv_2) = \textbf{true}$ (donde δ es la función de interpretación de operadores primitivos, y sv_1, sv_2 representan a valores especiales en nuestro lenguaje).

En el presente trabajo nos interesamos por definir únicamente una relación que describa, para todo sentencia y expresión posible, qué cómputo representa. No buscamos el desarrollo de un cálculo para razonar sobre equivalencias entre programas en Lua. Por lo tanto, nos concentramos en la definición de 2 tipos de relaciones:

- Nociones de reducción: relaciones no transitivas, no simétricas y no reflexivas, que representan la noción de "un paso de cómputo" (llamado "reducción", en el contexto de semántica de reducciones), efectuado por una construcción en particular (llamada "redex").

- Relación estándar de reducción: clausura compatible sobre todas las nociones de reducción, la cual permite describir la semántica operacional para todo programa posible. Que sea una clausura compatible significa que indica, para todo programa, cómo se reducen sus subfrases (los redex, presentes en el programa, ya sean sentencias o expresiones), y por lo tanto, podemos interpretarla como la relación definitiva que especifica la semántica de programas completos. Estamos interesados en una semántica determinista, por lo cual, tenemos que definir una estrategia para poder identificar, para un programa dado, un único redex y su

contexto. En nuestra semántica basada en sintáxis, el recurso para lograr esto es mediante la definición de contextos especiales, llamados "contextos de evaluación", los cuales son frases que representan todos los contextos posibles que pueden rodear a un redex, y que describen a su vez una estrategia determinista de evaluación de un programa.

Tras la definición de la relación estándar de reducción, tenemos la posibilidad de definir una función evaluadora para nuestro lenguaje. Para esto, podemos identificar qué clase de frases representan resultados de la reducción sucesiva de expresiones y sentencias (esto es, expresiones y sentencias irreducibles, y que pueden, entonces, interpretarse cómo resultados). Luego, mediante la determinación de la clausura reflexiva-transitiva de nuestra relación estándar de reducción, podemos identificar qué resultado está relacionado con un programa dado. De esa forma, y si nuestra relación estándar resulta ser efectivamente determinista, tenemos un evaluador para programas en nuestro lenguaje, que se comportaría como una función: para cada programa P, nos devuelve un único programa P', que representa el resultado de P.

A partir de contar con un evaluador para programas en nuestro lenguaje, disponemos de una estrategia para "recabar evidencia" de que nuestro lenguaje y su semántica sirven como modelo del lenguaje Lua 5.2: compilar la suite de tests de Lua 5.2 a nuestro lenguaje, y emplear nuestro evaluador para determinar si se satisfacen tales tests. Naturalmente, sería adecuado poder disponer de una versión mecanizada de nuestra semántica para poder automatizar estos testeos. El enfoque actual para resolver esto, consiste en la utilización de PLT Redex: una serie de herramientas, entre las que se incluye un lenguaje específico para definir las piezas de una semántica de reducciones (gramática, reglas de reducción, meta-funciones), utilizando notación semejante a la descripción matemática usual, y mecanismos para la exploración y depuración de los modelos definidos (aplicación de reglas de reducción, motor de encaje de patrones basados en la sintaxis provista para describir la semántica, facilidades para la definición de suite de tests), basadas en el lenguaje Racket[11]. Esto se presentará en el capítulo 9.

[11] http://racket-lang.org/

4

Presentación del lenguaje núcleo de Lua

Introducimos nuestro lenguaje, al cual nos referiremos como "el lenguaje núcleo de Lua", en alusión a la idea de que engloba el conjunto mínimo de mecanismos subyacentes en Lua. Describimos en esta sección las características generales del lenguaje núcleo, junto con las decisiones de diseño subyacentes en tales características. En posteriores secciones, definimos formalmente su sintaxis y su semántica.

Nuestro lenguaje consiste en una versión de Lua 5.2, si sus azúcares sintácticos y abstracciones lingüisticas, pero, con nuevas construcciones agregadas para poner de manifiesto mecanismos internos de Lua. En el diseño de nuestro lenguaje y la definición de su semántica, siendo que este es el recurso que emplearemos para entender Lua, buscamos que la formalización se logre con economía de recursos, para facilitar su estudio. Para esto, la semántica de reducciones de Felleisen-Hieb, con contextos de evaluación, nos permite dividir la descripción de la semántica completa en partes menores, que pueden ser entendidas de manera aislada, y lograr definiciones concisas para cada pieza de la semántica, esto es, definiciones que no contienen piezas de información que no son necesarias para entender el concepto que se quiere exponer. Estos aspectos de la semántica van a ir siendo expuestos a medida que avance la presentación.

A continuación, introducimos las características de nuestro lenguaje, y delineamos el conjunto de conceptos de Lua 5.2 que serán tratados en nuestro modelo.

4.1 Tipos

El lenguaje núcleo tiene todos los tipos de Lua, excepto user-data e hilos (este primer enfoque de la semántica de Lua, no incluye el tratamiento de la noción de co-rutinas). Agregamos además 3 tipos más, necesarios para describir mecanismos internos de Lua: referencias a "valores simples" (values que representan el resultado de la evaluación de una expresión), referencias a objetos (valores que, en Lua, son manejados a través de una referencia: tablas y funciones), y tuplas de valores, para representar listas de valores retornados por una función, y lista de argumentos de una función, de longitud variable.

Otro aspecto de los tipos de datos del lenguaje núcleo, es que su definición debería permitirnos realizar introspección de tipos, como ocurre en Lua. Este no es un aspecto muy interesante para resaltar, pero por razones de completitud, mencionamos que tal aspecto esta trivialmente cubierto en nuestra semántica basada en sintaxis, al darle a cada tipo una representación textual diferente, lo que permite identificarlos por simple encaje de patrones.

4.2 Estado

Nuestro lenguaje núcleo modela las formas de estado en Lua: las variables y las tablas son imperativas. Para introducir estado en nuestro lenguaje agregamos las construcciones y procedimientos, descritos a continuación.

4.2.1 Valores de tipo referencia

Como fue mencionado en la sección 3.1.1, en Lua hay 3 tipos de variables: locales, globales y campos de tablas. Internamente, las variables globales son tratadas como campos de una tabla especial (referida mendiante el identificador _ENV). Resta entonces como lograr un modelo de las variables locales y de los campos de tablas de Lua. Definimos aquí cómo modelamos el aspecto imperativo de las variables locales en Lua. El manejo del alcance de las definiciones de variables locales es tratado en la sección 6.2. El modelo de los campos de tablas es discutido en la próxima sub-sección.

El desafío actual radica entonces en cómo modelar una variable imperativa. En la literatura sobre semántica de reducciones, nos encontramos con estas propuestas:

- Variables asignables: es el modelo empleado en [10] para agregar estado a una extensión del lenguaje ISWIM. Se mantiene un mapeo entre identificadores de variables y valores, para modelar la noción de estado. La definición de una nueva variable asignable agrega un nuevo mapeo, y una asignación altera ese mapeo.

- Variables funcionales con valores de tipo referencia: mantenemos variables funcionales, las cuales son mapeadas a valores de tipo referencia, mediante el entorno. Finalmente, la referencia es mapeada al valor asignado en concreto, mapeo que es mantenido por una representación de la noción de almacenamiento. Hay al menos dos enfoques diferentes alrededor de esta idea:

  - Los valores de tipo referencia son como cualquier otro tipo de valor: a veces, esta idea es referida como "the reference concept" [15]. En este enfoque, las referencias están disponibles para el programador, como un tipo de dato ordinario, el cual puede ser empleado por él o ella, en los contextos en los que sea requerido. A tal fin, el lenguage va a proveer mecanismos especiales para crear y des-referenciar valores de tipo referencia. Este es el enfoque empleado en [1].
  - Los valores de tipo referencia no están disponibles para el programador: en este enfoque, las referencias no están bajo el control del programador. Estas son solo parte de mecanismos internos del lenguaje, empleados para lograr efectos imperativos. El des-referenciado y la creación de nuevas referencias es realizado de manera implícita. Este es el enfoque seguido en [11].

El utilizar variables asignables, no hay distinción entre el entorno y el almacenamiento. El efecto de la ligadura de variables y la asignación de variables, es descrito para un único mapeo entre identificadores de variables y valores. En particular, no es posible modelar el fenómeno de "alias". No hay distinción entre la noción de identificador de variable y referencia al almacenamiento. Eso no es suficiente para modelar el comportamiento de Lua, en presencia de valores de tipo función, tabla o hilos (u "objetos", como se menciona en [17]), ya que las variables de Lua no contienen actualmente tales valores, solo contienen referencias a ellos, y son estas referencias las que actualmente se almacenan en variables, permitiendo la posibilidad de definir alias, o pasadas como parámetros en una llamada a procedimiento o retornadas desde un procedimiento. Por lo tanto, para describir la manipulación de objetos en Lua, necesitamos la presencia de un entorno, un almacenamiento y entidades diferentes de los identificadores de variables, que llamaremos referencias.

Al escoger un modelo con valores de tipo referencia, todavía tenemos que determinar cual de los modelos con valores de tipo referencia es más adecuado para nuestras necesidades. En Lua, las referencias aparecen como parte de un mecanismo interno para definir el manejo de objetos. Las referencias no están disponibles para el programador. Por otro lado, la adición de operadores especiales para crear referencias y des-referenciarlas vuelve más intrincado el código resultante, algo que, de ser posible, debe ser evitado. Adoptamos entonces el segundo enfoque descrito, y dejamos a los valores de tipo referencia, fuera del alcance del programador.

Agregamos el tipo simpvalref de referencias, para modelar variables locales. Estas referencias son creadas y des-referenciadas implícitamente. Las mismas referencian "valores simples" (aquellos que son el resultado de la evaluación de una expresión). Por ejemplo:

local X = *value* **in** *block* **end**

Si *value* no es un objeto, en la ejecución del programa anterior, un nuevo valor r, de tipo simpvalref, es creado, el mapeo $\mathsf{X} \to r$ es agregado al entorno, y el mapeo $r \to$ *value* es agregado al almacenamiento. En un intento de obtener el valor de X, se des-referencia de manera implícita r. Una asignación a X, hará que el mapeo $r \to$ *value* cambie.

Para modelar la forma en la que una variable en Lua (ya sea local, global o un campo de tabla) referencia a un objeto, agregamos el tipo objref, que representa referencias a objetos. Sea el siguiente programa:

local X = *object* **in** *block* **end**

En la ejecución de este programa, nuevamente una referencia r, de tipo simpvalref, es creada, y el mapeo $\mathsf{X} \rightarrow r$ es agregado al entorno. Luego, una referencia l, de tipo objref, es creada, y los mapeos $r \rightarrow l$ y $l \rightarrow$ *object* son agregados al almacenamiento. En situaciones en donde sólo se requiere la referencia a *object* (asignación, pasaje de parámetros o retorno de llamada a procedimiento) se realiza una des-referencia implícita sobre r, para obtener l. Si el valor *object* es requerido, entonces se des-referencia l.

4.2.2 Tablas mutables

Las tablas en Lua son imperativas. Nosotros agregamos tablas imperativas de manera directa en nuestro lenguaje. Por lo tanto, los campos de tabla, el tipo restante de variable, se comportan como variables imperativas.

Hay otra posibilidad explorada en la literatura sobre semántica de reducciones de Felleisen-Hieb, para modelar este tipo de estructuras, con efectos imperativos: utilizar una estructura funcional, con actualización funcional para implementar cualquier actualización en la estructura de datos (como lo realizado en [1]). De esta forma, cuando es necesario actualizar un campo de la tabla, se construye en su lugar otra tabla, que es igual a la primera, en todo campo, excepto aquel que se quiere modificar: en la nueva tabla creada, el campo en cuestión contendrá el valor que se está intentando asignar. Por lo tanto, una asignación de campo de tabla correcto, debería expresarse como:

variable := (*variable*[*field* → *new_value*])

Donde (*variable*[*field* → *new_value*]) denota la actualización funcional de la tabla, referenciada por *variable*. Este recurso es empleado en [1], en conjunción con la utilización de operadores para des-referenciar la referencia a la tabla, ya que se emplea el "reference concept". Aunque conceptualmente claro como modelo (analizado en términos de sus partes componentes), lo realizado allí deriva en un código muy intrincado, cuyo precio se paga al momento de analizar el código resultado de la traducción del lenguaje cuya semántica queremos explicar, hacia el lenguaje que estamos definiendo para tal fin. Inclusive si empleamos des-referencias implícitas, sigue siendo más sucinta, e igualmente clara, la adición de tablas mutables de manera directa en nuestro lenguaje. Con esto nos referimos a que, desde el punto de vista del lenguaje, las tablas son mutables, y su alteración se reduce a la sintaxis siguiente:

variable[*field*] := *value*

Luego, la descripción de su semántica sí hará uso, como es de esperar, de recursos similares a la actualización funcional, para modelar este cambio en el almacenamiento.

4.2.3 Entorno

El entorno mapea identificadores de variables con referencias, mientras que el almacenamiento mapea estas referencias con valores. Es de esta forma en que los identificadores de variables están relacionados con los valores que están representando. Se presenta una situación similar en lógica: la noción de entorno, que mapea las variables libres de una fórmula con sus significados. Es sabido que no es necesario mantener este mapeo, para determinar el valor de una fórmula. Todo lo que se requiere es reemplazar las ocurrencias de las variables libres por su significado (perdiendo así el mapeo), y evaluar la fórmula enriquecida resultante. Y este recurso también se emplea en la semántica de reducciones, para manipular entornos. Por lo tanto, en nuestro lenguaje, para describir las frases que alteran el entorno, emplearemos un mecanismo de substitución similar. Aquí, el "significado" de un identificador de variable libre, será la referencia a el cual esta mapeado. Por lo tanto, cuando una construcción altera el entorno, agregando un nuevo mapeo entre un identificador y una referencia, cada ocurrencia libre en el código de tal identificador será reemplazado por la referencia, de acuerdo a las reglas del lenguaje que definen el alcance de la declaración de una variable.

4.2.4 Almacenamiento

Dado que pretendemos lograr la descripción de la semántica de nuestro lenguaje, basándonos puramente en la sintaxis del mismo, necesitamos agregar una construcción sintáctica que modele la noción de mapeo entre referencias y valores, e interpretar toda operación sobre ese mapeo como manipulaciones sintácticas, sobre la representación del almacenamiento. En lo que sigue, presentamos diferentes maneras de representar almacenamientos, empleadas en semánticas de reducciones de lenguages imperativos, en donde también se utilizan valores de tipo referencia.

Matthias Felleisen propone, en [6], incrustar la información del almacenamiento directamente en el código que representa al programa, utilizando valores etiquetados por las referencias que los refieren, y reemplazando los identificadores de variables por estos valores. Por lo tanto, el valor etiquetado (que está en representación de un mapeo referencia-valor) está repetido por cada ocurrencia del identificador de variable que corresponde. Hay que mencionar que este enfoque es propuesto en el intento del desarrollo de un cálculo para una variante del cálculo lambda con construcciones imperativas. Se intenta entonces apegarse al objetivo de lograr definir una manera de razonar sobre equivalencias entre programas imperativos, basándose únicamente en el texto que describe a los programas (posiblemente enriquecidos con más información, para hacer posible la descripción de su semántica).

En trabajos más recientes, como [1] o [11], la propuesta es extraer la información del almacenamiento y dejarla en un lugar, por fuera del código que representa al programa, empleando una estructura que sugiere la idea de un mapeo entre referencias y valores. Las ocurrencias de identificadores de variables en el código, son reemplazadas por referencias (es decir, se maneja el entorno de la misma propuesta anteriormente), y cualquier cambio en el mapeo entre referencias y valores es conducida sobre la representación del almacenamiento.

Aunque las razones para escoger esta última notación no nos resultan del todo claras, es evidente que se ha convertido en un recurso usual en trabajos recientes. Por lo tanto, apelaremos al mismo recurso. Por otro lado, como hay una distinción clara entre las situaciones en las que operamos con objetos y las situaciones en las que trabajamos con valores simples (como se verá en el desarrollo de la semántica del lenguaje), almacenaremos valores simples en una estructura diferente de aquella que emplearemos para almacenar objetos. Utilizaremos el símbolo σ, para referirnos al primer almacenamiento, y θ para referirnos al segundo. En particular, el separar la información de ese modo nos permite definir de manera más simple todas aquellas meta-funciones que trabajan sobre uno u otro tipo de almacenamiento (mantener los mapeos de tipo *simpvalref* y *objref* en el mismo lugar, nos obliga, al recorrer el almacenamiento, a considerar, en cada meta-función que opera con uno u otro tipo de referencia, un caso de su definición que se dedique a descartar el mapeo del tipo de referencia sobre el que que no opera).
Por lo mencionado, definimos almacenamientos como una construcción sintáctica que representa un conjunto finito de pares ordenados, según lo indicado por las siguientes producciones, de la gramática del lenguaje, que se definirá en la próxima sección:

σ ::= * | '(' simpvalref ',' simplevalue ')' ',' σ

θ ::= * | '(' objref ',' intrepobject ')' ',' θ

4.3 Manejo de errores

En esta propuesta, no modelaremos el mecanismo de errores de Lua en su totalidad. En concreto, modelaremos la capacidad de los objetos de error de descartar la continuación actual (o el contexto de evaluación, en términos de la semántica de reducciones), pero no incluiremos la posibilidad de incorporar a los mensajes de error información sobre la pila de llamadas que desembocaron en el error, lo cual requiere, entre otras cosas, mantener información sobre el entorno, la cual, en nuestra propuesta de manejos de entorno, es desechada durante el proceso de reducción. Futuras revisiones de este trabajo, que intenten modelar de manera completa el mecanismo de errores de Lua, deberán re-definir el manejo de entornos.

En esta formalización también incluiremos el mecanismo de ejecución de procedimientos en "modo protegido", lo que permite capturar errores que pudieran presentarse durante la ejecución de tales procedimientos.

4.4 Mecanismo de meta-tablas

En este trabajo también formalizamos el mecanismo de meta-tablas de Lua. El mismo será definido con 2 piezas: primero, habrá nociones de reducción que se encargarán de reconocer situaciones especiales que requieren de la intervención del mecanismo de meta-tablas. En el proceso de reducción, este paso resultará en expresiones o sentencias etiquetadas con información que indica el problema o la situación especial hallada. Luego, una noción de reducción diferente describirá cómo el mecanismo de meta-tablas trabaja para atender la situación especial. De esta forma, evitamos el repetir chequeos de condiciones que configuran condiciones manejadas por meta-tablas (algo que, aparte, tiene impacto positivo en el rendimiento de la mecanización de nuestra semántica con PLT Redex). La separación entre el reconocimiento de una situación que requiere la intervención del mecanismo de meta-tablas, y la forma en la que el mecanismo en sí opera, facilita también, a futuro, el agregar cualquier cambio que pudiera realizarse a futuro en Lua, en alguno de estos 2 aspectos.
Como ocurre en Lua, tenemos que modelar el hecho de que solamente las tablas tienen meta-tablas individuales. Para representar esto, cada tabla será almacenada junto con su meta-tabla, lo que configurará la forma interna de representación de las tablas. Valores de cualquier otro tipo, comparten meta-tablas, una por tipo. Recurriremos a almacenar tales meta-tablas en posiciones fijas (reservadas) dentro del almacenamiento.

4.5 Recolección de basura

En este trabajo, no incluiremos una formalización de la manera en que Lua realiza la administración automática de memora. A raíz de ello, no realizaremos ningún tipo de administración de la memoria (para mitigar la falta del mecanismo de recolección de basura de Lua), ya que esto podría alterar la semántica resultante, con respecto a la de la Lua.

4.6 Construcciones presentes en nuestro lenguaje

Lua en sí es un conglomerado de construcciones para atender diversas necesidades, entre las que se incluye un mecanismo versátil para describir datos, construcciones de programación estructurada, recursos para extender la semántica del lenguaje hacia dominios específicos y comodidades sintácticas pensadas para el público hacia el cual fue originalmente dirigido. La identificación de un repertorio mínimo de conceptos para incluir, requiere entonces revisar las componentes mencionadas del lenguaje, identificando azúcares sintácticos y abstracciones lingüísticas. El procedimiento en sí realizado no reserva ninguna sorpresa. La sintaxis resultante es presentada en el próximo capítulo. La forma en la que modelamos todo los conceptos de Lua empleando nuestro lenguaje, se describe en el capítulo 7.

5

Gramática del lenguaje núcleo

La gramática empleada para describir un lenguaje, en el contexto de una semántica de reducciones, no se limita únicamente a describir programas válidos, sino que debe generar (hablamos de gramáticas libres de contexto) todo tipo de construcción que se presenta durante la reducción de un programa (para poder chequear que, durante los pasos intermedios de reducción, obtenemos frases de un tipo esperado), y debe identificar como categorías sintácticas especiales a ciertas frases que constituyen casos particulares de otras (como, por ejemplo, qué consideramos como expresiones totalmente reducidas), categorías las cuales son necesarias mencionar al momento de describir las nociones de reducción.

Nuestro lenguaje pretende ser un modelo para razonar sobre Lua. Naturalmente, cuanto más semejante sea el primero al segundo, más fácil será adoptarlo como la herramienta que pretende ser. Como tal, queremos presentar a nuestro lenguaje como una versión de Lua, sin azúcares sintácticos ni abstracciones lingüísticas, pero enriquecidos con la menor cantidad de información necesaria para poder razonar directamente sobre las frases de nuestro lenguaje, en términos de una semántica operacional. En ese sentido, imaginamos que, características de la sintaxis Lua, como la distinción entre sentencias y expresiones, deben estar también presentes en nuestro lenguaje, si queremos mantener la proximidad de este con respecto a Lua. Sin embargo, por razones que iremos exponiendo, al momento de definir la gramática de un lenguaje cuya semántica será descrita puramente en términos de sintaxis, necesitamos ofuscar esa distinción entre expresiones y sentencias, a la vez que, como se mencionó, tenemos que enriquecer la gramática, para admitir otras construcciones, aparte de las que representan programas válidos.

Decidimos entonces ir presentando las distintas piezas de la gramática de manera separada, comenzando por la descripción de los programas válidos: aquellos programas que pretenden ser la representación directa de un programa Lua.

La notación empleada para describir la gramática será EBNF (Extended Backus-Naur Form). Los símbolos no terminales serán descritos en sans serif, las palabras del lenguaje estarán en **negrita**. Colocaremos entre comillas simples los símbolo terminales del lenguaje, que pueden llegar a confundirse con los caracteres de la notación EBNF empleada para describir las producciones de la gramática.

5.1 Gramática de programas válidos

El repertorio de sentencias de nuestro lenguaje, es el siguiente:

```
block ::=  statement
         | block ';' block

label ::= '::' name '::'

statement ::=  label '{' block '}'
             | do block end
             | break name tuple
             | functioncall
             | if exp then block else block end
```

| **while** exp **do** block **end**
| **local** namelist '=' explist **in** block **end**
| varlist '=' explist
| **void**

La gramática para expresiones:

exp ::= '...'
| var
| functioncall
| '(' exp ')'
| **nil**
| boolean
| number
| string
| tuple
| simpvalref
| objref
| tableconstructor
| functiondef
| exp binop exp
| unop exp

binop ::= normalbinop | shortcircuitbinop

normalbinop ::= '+' | '−' | '∗' | '/' | ' ^ ' | '%' | '..' | '<'
| '<=' | '=='

shortcircuitbinop ::= and | or

unop ::= '-' | not | '#'

functioncall ::= exp '(' [explist] ')' | exp : Name '(' [explist] ')'
| **$builtIn** name '(' [explist] ')'

functiondef ::= **function** name '(' [parlist] ')' block **end**

parlist ::= namelist [',' '...'] | '...'

namelist ::= name { ',' name }

var ::= name | exp '[' exp ']'

varlist ::= var { ',' var }

tableconstructor ::= '{' [fieldlist] '}'

fieldlist ::= field { ',' field }

field ::= '[' exp ']' '=' exp | exp

tuple ::= **empty** | '<' explist '>'

boolean ::= **false** | **true**

simplevalue ::= **nil** | boolean | number | string | objref

simplevaluelist ::= simplevalue { ',' simplevalue }

explist ::= exp { ',' exp }

La gramática que describe la representación de almacenamientos:

intreptable ::= '(' evaluatedtable ',' **nil** ')'
| '(' evaluatedtable ',' objref ')'

intrepobject ::= functiondef | intreptable

σ ::= * | '(' simpvalref ',' simplevalue ')' ',' σ

θ ::= * | '(' objref ',' intrepobject ')' ',' θ

Los valores simples (generados por las producciones de simplevalue) son solo una categoría sintáctica definida para identificar las construcciones que resultan de la evaluación de expresiones. Son almacenados en el almacenamiento de los valors simples (descrito por las producciones de σ), y son referenciados mediante las referencias generadas por simpvalref. Las cadenas generadas por las producciones de intrepobject describen la representación interna (en el almacenamiento) de los "objetos" (aquellos valores manipulados en Lua a través de las referencias a los mismos). Estos valores son almacenados en el "almacenamiento de objetos" (θ) y referenciados por las referencias objref.

5.2 Extensiones de la gramática

Introducimos ahora extensiones a la gramática, para adecuarla al uso en el contexto de la definición de una semántica de lenguajes, que está puramente basada en sintaxis.

Primero, añadiremos nuevas producciones para símbolos terminales ya presentes, para describir frases que aparecen durante el proceso de reducción (como frases etiquetadas, que indican situaciones especiales, a ser tratadas por el mecanismo de meta-tablas). La adición de producciones para un símbolo no-terminal v será descrita:

v ::= ... | (nueva producción)

Luego, para poder hacer referencia, en las reglas de reducción, a ciertos casos especiales de construcciones, agregaremos nuevos símbolos no terminales, con sus respectivas producciones, que describirán estas categorías sintácticas especiales (como constructores de tablas evaluados o listas de variables evaluadas).

Comenzamos permitiendo, desde la gramática, el generar sentencias en el lugar de expresiones. Este de frases podrían presentarse en el proceso de reducción de un programa como:

if f () **then** ... **else** ... **end**

Por la forma en la que se describe la reducción de la llamada a función presente en la guarda del condicional (ver sección 6.4.4), terminaríamos con las sentencias que forman parte del cuerpo de la función f, ubicadas en el lugar de la guarda del condicional. Hasta ahora, la gramática definida no admite ese tipo de construcciones, ya que mantiene la separación estricta, presente en Lua, entre sentencias y expresiones.

En el lenguaje tenemos también presentes expresiones cuya semántica operacional va a estar descrita en términos de la sentencia en la que aparecen. Es decir que, para describir su semántica,

será necesario mirar la sentencia completa en la que aparecen. A raíz de ello, la noción de reducción que describirá su comportamiento deberá ser una relación también sobre sentencias, a pesar del hecho de tratarse de una relación definida para describir la semántica de expresiones. Para cubrir estos casos, agregamos:

exp ::= ... | block

Presentamos esta extensión simple a la gramática, de esta manera, para evitar oscurecer la distinción presente en nuestro lenguaje (como en Lua), entre sentencias y expresiones.

Como fue mencionado en 4.4, uno de los pasos de reducción que describen el mecanismo de meta-tablas, consiste en el etiquetar una frase dada (sentencia o expresión), con información adecuada para indicar la situación especial que se presenta en esa frase. Las sentencias etiquetadas, que representan situaciones manejadas por el mecanismo de meta-tablas son:

block_abnormal ::= '(' objref '[' simplevalue ']' '=' simplevalue ')' **TableAssignmentWrongKey**
| '(' simplevalue '[' simplevalue ']' '=' simplevalue **TableAssignmentOverNonTableVal**

block ::= ... | block_abnormal

Las expresiones etiquetadas, que representan situaciones manejadas por el mecanismo de meta-tablas son:

exp_abnormal ::= '(' objref '[' simplevalue ']' ')'**KeyNotFound**
|'(' simplevalue '[' simplevalue ']' ')'**NonTableIndexed**
|'(' simplevalue '+' simplevalue ')'**AdditionWrongOperands**
|'(' simplevalue '-' simplevalue ')'**SubstractionWrongOperands**
|'(' simplevalue '*' simplevalue ')'**MultiplicationWrongOperands**
| '(' simplevalue ' ^ ' simplevalue ')'**ExponentiationWrongOperands**
| '(' simplevalue '/' simplevalue ')'**DivisionWrongOperands**
| '(' simplevalue '%' simplevalue ')'**ModuleWrongOperands**
| '(' '-' simplevalue ')'**NegationWrongOperand**
| '(' simplevalue '..' simplevalue ')'**StringConcatWrongOperands**
| '(' # simplevalue ')' **StringLengthWrongOperand**
| '(' simplevalue '==' simplevalue ')'**EqualityFail**
| '(' simplevalue '<' simplevalue ')'**LessThanFail**
| '(' simplevalue '<=' simplevalue ')'**LessThanOrEqualFail**
| '('simplevalue '(' simplevaluelist '))'**WrongFunctionCall**

exp ::= ... | exp_abnormal

5.2.1 Nuevas categorías sintácticas

Cuando definimos algunas reglas de reducción, necesitamos una manera de hacer referencia a casos especiales de construcciones ya definidas: para almacenar una tabla, los campos descritos en su constructor deben estar ya reducidos; la lista de valores retornados desde una función, es modelada empleando tuplas, y para describir el retorno de tal lista de resultados, los valores presentes deben estar completamente reducidos.

Agregamos las siguientes categorías sintácticas para describir estos y otros casos especiales:

evaluatedtable ::= '{' [evaluatedfieldlist] '}'

evaluatedfieldlist ::= evaluatedfield { ',' evaluatedfield }

evaluatedfield ::= '(' '[' simplevalue ']' '=' simplevalue ')' | simplevalue

evaluatedtuple ::= '(' **empty** ')' | '(' '<' simplevaluelist '>' ')'

evaluatedvar ::= simpvalref | objref '[' simplevalue ']'

evaluatedvarlist ::= evaluatedvar { ',' evaluatedvar }

var ::= ... | evaluatedvar

5.2.2 Contextos

Haremos uso de contextos para describir donde y en qué orden debe ocurrir la reducción de redexs presentes en sentencias y expresiones (contextos los cuales son generalmente referidos como "contextos de evaluación"). Esto nos servirá para describir, finalmente, cómo se reducen programas enteros, ya que, al momento de definir las nociones de reducción, nos concentraremos en indicar únicamente cómo reduce un redex, sin fijarnos en el contexto en el que ocurre.

Los contextos de evaluación serán empleados también para describir el comportamiento de sentencias o expresiones cuya semántica es sensible al contexto, o que operan sobre "el resto de cómputo" (o continuación), concepto el cual es naturalmente modelado por el contexto en el que ocurre un redex.

Iremos introduciendo cada categoría de contexto de evaluación, junto con una descripción breve de su uso. En lo que sigue, usaremos $[\![\ ]\!]$ para denotar el "hueco" (hole) de un contexto:

- Los siguientes contextos describen cómo una reducción ocurre en expresiones y sentencias, que no sean bloques etiquetados o sentencias **break**. Notar que estamos mezclando sentencias con expresiones. Nuevamente, estamos forzados a hacer esto para referirnos al tipo de contexto que podría aparecer en los pasos intermedios del proceso de reducción, en donde, por ejemplo, una sentencia podría aparecer en el lugar de una expresión. Todos estos contexto que describimos a continuación, serán útiles para modelar la noción de "resto de cómputo" o continuación, que puede ser descartado por la sentencia **break**:

  E_j ::= $[\![\ ]\!]$
  | **do** E_j **end**
  | '(' E_j ')' **ProtectedMode**
  | **if** E_j **then** block **else** block **end**
  | **local** namelist = E_{le} **in** block **end**
  | E_j ; block
  | evaluatedvar , ... , E_j [exp], var , ... = explist
  | evaluatedvar , ... , simplevalue [E_j], var , ... = explist
  | evaluatedvarlist = E_{le}
  | **$builtIn** Name (E_{le})
  | E_j (explist)
  | simplevalue (E_{le})
  | E_j ':' Name (explist)
  | (E_j)
  | E_j binop exp
  | simplevalue normalbinop E_j
  | unop E_j
  | < E_{le} >
  | { evaluatedfield , ... , [E_j] = exp , field ... }
  | { evaluatedfield , ... , [simplevalue] = E_j , field ... }

| { evaluatedfield , ... , E_j , field ... }
| E_j [exp]
| simplevalue [E_j]

- Algunas producciones de E_j están definidas en términos de E_{le}, que representa a una lista de expresiones. Estas son evaluadas de izquierda a derecha, comportamiento el cual es impuesto por la siguiente definición:

  E_{le} ::= simplevalue , ... , E_j , exp , ...

- Todos los contextos de evaluación posibles son incluidos en la siguiente categoría sintáctica:

  E ::= E_j
  | label { E }
  | **break** name < E_{le} >

  Al indicar que los contextos de E_j también pertenecen a los contextos de E, nos referimos a que agregamos los mismos reemplazando toda ocurrencia en ellos del símbolo no terminal E_j por E. También asumimos que en la definición de E_{le} realizamos el mismo reemplazo.

- Los valores de una tupla son descartados, salvo quizás el primero (truncar la lista de valores de una tupla), o agregados a una lista de valores, dependiendo del contexto inmediato en el que ocurren (esto es, contextos no anidados). Para lograr la descripción de este comportamiento sensible al contexto, distinguimos los contextos en los que se debe truncar la lista de valores de una tupla (E_t) de aquellos en donde la lista completa de valores de la tupla se debe agregar a otra (E_u):

  E_{tel} ::= simplevalue , ... , [[ ]] , exp, ...

  E_t ::= [[ ]] (explist)
  | simplevalue (E_{tel})
  | [[ ]] ':' Name (explist)
  | **$builtIn** name (E_{tel})
  | [[ ]] binop exp
  | simplevalue normalbinop [[ ]]
  | unop [[ ]]
  | < E_{tel} >
  | { evaluatedfield , ... , [[[ ]]] = exp , fieldlist }
  | { evaluatedfield , ... , [simplevalue] = [[ ]] , fieldlist }
  | {evaluatedfield , ... , [[ ]] , fieldlist }
  | [[ ]] [exp]
  | simplevalue [[[ ]]]
  | **if** [[ ]] **then** block **else** block **end**
  | **local** namelist = E_{tel} **in** block **end**
  | evaluatedvarlist = E_{tel}
  | evaluatedvar , ... , [[ ]] [exp], var , ... = explist
  | evaluatedvar , ... , simplevalue [[[ ]]], var , ... = explist
  | **break** Name < E_{tel} >

  E_{uel} ::= simplevalue , ... , simplevalue, [[ ]]

  E_u ::= < E_{uel} >
  | simplevalue (E_{uel})

| **$builtIn** name (E_{uel})
| { evaluatedfield, ..., evaluatedfield, ⟦ ⟧ }
| **local** namelist = E_{uel} **in** block **end**
| evaluatedvarlist = E_{uel}
| **break** Name < E_{uel} >

- Hay otras situaciones que involucran a las tuplas de valores, que para especificarlas, necesitamos estudiar el "resto de cómputo", o todo el contexto que rodea a una frase dada. Estas frases tienen que ver con la sentencia **break**, cuyo comportamiento es modificado por el contexto en el que aparece. Cuando la tupla que la sentencia **break** devuelve termina en el lugar de una expresión, entonces tal tupla es mantenida, y se aplican las reglas para truncar su lista de valores o agregarla a otra lista de valores. Pero si la tupla de valores va a aparecer en el lugar de una sentencia (como cuando empleamos una llamada a función como sentencia), entonces la tupla de valores debe ser descartada. Este comportamiento sensible al contexto puede ser formalizado de manera simple, describiendo primero los siguientes contextos de evaluación (en donde, los contextos E_d son aquellos en donde la tupla de valores es descartada, y los contextos de E_k son aquellos en donde la tupla es mantenida):

  E_d ::= ⟦ ⟧
  | E ⟦ **do** ⟦ ⟧ **end** ⟧
  | E ⟦ ⟦ ⟧ ; block ⟧
  | E ⟦ label '{' ⟦ ⟧ '}' ⟧

  E_k ::= E⟦ E_t ⟧ | E⟦ E_u ⟧

- Finalmente definimos a la categoría de contextos E_{np}, que incluye a todos los contexto de E, con excepción del contexto '(' E ')' **ProtectedMode**, el cual representa al modo protegido de Lua, dentro del se puede ejecutar una función, y atrapar cualquier mensaje de error resultado de la ejecución de esta. Estos contextos, como es de esperar, ayudan a describir la semántica del modo protegido, posibilitándonos hacer mención a cualquier contexto de evaluación posible, que sea diferente al que representa efectivamente el modo protegido.

Hay una característica de los contextos de evaluación, dado nuestro objetivo de definir noción de reducción estándar que describa la semántica de todo programa válido, que hace a su buena definición: que para toda sentencia y expresión, si esta no está totalmente reducida, entonces debería existir una única manera de particionarla entre un contexto de evaluación y un redex. En el actual trabajo, nos enfocamos en la experimentación con la mecanización del modelo. Queda fuera, entonces, el estudio analítico de las propiedades del modelo.

Hay también otras propiedades que quisiéramos de nuestros contextos de evaluación: concretamente, que el orden de reducción que ellos imponen sea el correcto, y que las reducciones que fuerzan a que ocurran (en los casos de frases con comportamiento sensible al contexto) sean las correctas. Estas propiedades son justamente parte de la semántica de Lua 5.2 que queremos capturar en nuestro modelo. Y, qué tan bien se corresponde nuestro modelo con respecto a Lua 5.2, recae en determinar si, dado un programa del primero, este arriba al mismo resultado que el correspondiente programa "equivalente", en nuestro lenguaje. Esto no podemos determinarlo de manera analítica, ya que para tal fin, necesitaríamos de un modelo de Lua 5.2, que admita un razonamiento matemático, modelo el cual no disponemos, sino que este que estamos presentando, pretende ser un primer paso en la construcción de tal modelo. Deberemos entonces estimar qué tan bien nuestro modelo se aproxima a los aspecto mencionados de Lua 5.2, recabando evidencia empírica de esa correspondencia, mediante el uso de PLT Redex, para mecanizar nuestro modelo y realizar pruebas de conformidad del mismo con respecto a Lua 5.2. Esto lo trataremos en el capítulo 9.

6

Semántica de reducciones

Emplearemos algunas convenciones para describir construcciones del lenguaje de manera sucinta. Variables cuantificadas sobre las frases generadas por un símbolo no terminal v, estarán indicadas como v, utilizando sub-índices, si fuera necesario, para distinguir entre varias frases generadas por las producciones de v. También se emplearán algunos identificadores breves, para referirse a cierto tipo particular de construcción: x e y para referirse a identificadores de variables, sv para referirse a valores simples, r y l para referirse a referencias a valores simples o referencias a objetos, respectivamente. Se mencionará cuando se utilice algún otro identificador, con la misma finalidad. Finalmente, la notación L(v) se referirá al conjunto de frases generadas por las producciones del símbolo no terminal v.

Las reglas de reducción que vamos a definir, relacionan frases del lenguaje en base a su sintaxis. En la definición de estas reglas, requeriremos emplear operaciones sobre las construcciones del lenguaje. Todos los objetos a los que nos referiremos, son simplemente cadenas de símbolos terminales derivadas de la gramática definida anteriormente. Por lo tanto, será común definir predicados sobre la estructura sintáctica de esas cadenas. Es costumbre, en la literatura del área, describir tales predicados de forma muy sucinta. En ciertas circunstancias, para mencionar que una cadena s tiene una subcadena s', comenzando en cualquier posición de s, emplearemos un notación ecuacional simple como s = ... s' ..., cuando, del contexto, está claro qué tipo de construcción es s y qué tipo de construcción s'.

También, tener presente la distinción entre el lenguaje que empleamos para describir las reglas de reducción (a veces referido como el "meta-lenguaje"), y el lenguaje cuya semántica estamos describiendo (a veces referido como el "lenguaje objeto"). Cuando pueda prestarse a confusión, encerraremos entre comillas simples a aquellos símbolos terminales del lenguaje objeto, para diferenciarlos de símbolos del meta-lenguaje.

6.1 Manipulación de almacenamientos

Cuando definimos reducciones que trabajan con los almacenamientos, necesitaremos emplear meta-funciones que los manipulen. La definición de tales meta-funciones asumirá que las cadenas de símbolos sobre las que trabajan representan correctamente a un almacenamiento: es decir, que no hay 2 pares ordenados que tengan la misma primer componente (se trata de una restricción que no describimos con nuestra gramática). Comenzaremos definiendo las meta-funciones que trabajan con el almacenamiento de valores simples, σ.
Siendo los almacenamientos, en nuestro lenguaje, una representación de un mapeo entre referencias y valores, se presentará la necesidad de hacer referencia al "dominio" de ese mapeo, también será necesario obtener el valor al cual está mapeada una referencia dada, y alterar ese mapeo. Emplearemos la notación $\mathsf{dom}(\sigma)$, para referirnos al dominio del mapeo representado por la cadena σ: esto es, si $\sigma = (r_1,sv_1),...,(r_n,sv_n),*$ entonces $\mathsf{dom}(\sigma) = \{r_1,...,r_n\}$. Para obtener el valor al cual una referencia dada está mapeada, según un almacenamiento σ, emplearemos la siguiente meta-función:

$$\_\,(\_) \;:\; L(\sigma)\ x\ L(\mathsf{simpvalref}) \;\rightarrow\; L(\mathsf{simplevalue})$$

$$\sigma\ (r) = sv,\ \mathit{if}\ \sigma\ =\ ...'('\ r\ ','\ sv\ ')'...$$

Para alterar el valor hacia al cual una referencia dada está mapeada, emplearemos la siguiente meta-función::

$$\_\,[\_ := \_] \;:\; L(\sigma)\ x\ L(simpvalref)\ x\ L(simplevalue) \;\rightarrow\; L(\sigma)$$

$$\sigma\ [r := v] = \sigma'\ \text{, where } r \in dom(\sigma),\ \mathsf{dom}(\sigma) = \mathsf{dom}(\sigma'),\ \forall\ r' \in \mathsf{dom}(\sigma'), [(r' \neq r \Rightarrow \sigma'(r') = \sigma(\mathsf{r}')) \wedge (r' = r \Rightarrow\ \sigma'(r') = v)]$$

Será útil disponer de una abreviación que denota la aplicación sucesiva de la meta-función anterior, a un almacenamiento dado:

$$\sigma\ [r_1\ :=\ v_1, ..., r_n\ :=\ v_n]\ = (...(\sigma\ [r_1\ :=\ v_1])...[r_n\ :=\ v_n])$$

Asumimos definidas (de la misma manera) a las correspondientes meta-funciones que operan sobre un almacenamiento de objetos, θ, para obtener el dominio su dominio ($\mathsf{dom}(\theta)$), obtener el valor al cual una referencia l está mapeada, según un almacenamiento θ ($\theta\ (l)$), y para alterar el mapeo de una referencia l, en un almacenamiento θ ($\theta\ [l\ :=\ o]$, donde o la representación interna de un objeto).

6.2 Manipulación de entornos

Los entornos son empleados para describir la correspondencia de un identificador de variable con una referencia (la "ligadura de variables") y el alcance de ese correspondencia. Como fue mencionado en 4.2.3, manipulamos entornos mediante el uso de un mecanismo de substitución de un identificador de variable por su correspondiente referencia, de acuerdo al alcance de la declaración de variable en cuestión, modelando así el significado de la declaración.

Debido al hecho de que tenemos construcciones que pueden tener ocurrencias ligadoras de varias variables, necesitamos una manera de realizar la substitución descrita, de manera simultánea, para todas las variables involucradas. Sea S el conjunto de todos los mapeos finitos que van desde identificadores de variables (miembros de L(Name)) y la expresión *vararg* ('...') [1], hacia otras expresiones. Para describir un miembro de S, que representa un mapeo particular entre identificadores x_i con expresiones exp_i, para $1 <= i <= n$, usaremos la notación:

$$(x_1 \setminus exp_1\ ,\ ...\ ,\ x_n \setminus exp_n)$$

El orden en el que el mapeo es descrito no tiene importancia, por lo que, cuando sea necesario, describiremos un orden particular, por conveniencias de notación. También, la substitución de la expresión *vararg* por otra expresión será denotada simplemente como $'...'\setminus exp$, en la lista de substituciones de un miembro S. Finalmente, un miembro de S será denotado simplemente con s.

La función de substitución que estamos buscando puede ser descrita mediante los mapeos:

$\_[\_]_{block}$: L(block) x S $\rightarrow$ L(block) (substitution in blocks)

$\_[\_]_{exp}$: L(exp) x S $\rightarrow$ L(exp) (substitution in expressions)

La construcción de nuestro lenguaje a la cual le aplicamos la substitución, será la primera sentencia o expresión que está inmediatamente a la izquierda del operador de substitución. Cuando haya posibilidad de interpretación amgigua, con paréntesis a la frase a la que se le aplica la substitución.

La función de substitución sobre expresiones puede ser definida de manera inductiva sobre la estructura de las expresiones. Comenzamos entonces por la substitución sobre expresiones sin estructura:

[1] En Lua, cuando en la lista de parámetros de una función, aparece la marca '...', entonces, los argumentos extra que reciba, en una llamada, tal función, se colocaran en una lista a la que se puede hacer referencia, en el cuerpo de la función, mediante la misma marca '...'

$$
\begin{array}{ll}
x\ [\mathsf{s}]_{\mathsf{exp}} & = \mathsf{s}(x) \\
\text{'...'}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathsf{s}\ (\text{'...'}) \\
\textbf{nil}\ [\mathsf{s}]_{\mathsf{exp}} & = \textbf{nil} \\
\mathit{boolean}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{boolean} \\
\mathit{number}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{number} \\
\mathit{string}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{string} \\
\mathit{simpvalref}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{simpvalref} \\
\mathit{objref}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{objref} \\
\textbf{empty}\ [\mathsf{s}]_{\mathsf{exp}} & = \textbf{empty}
\end{array}
$$

asd

La substitución en expresiones con estructura está definida como:

$$
\begin{array}{ll}
\mathit{exp_1}\ \text{'('}\ \mathit{exp_2}, ..., \mathit{exp_n}\ \text{')'}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{exp_1}\ [\mathsf{s}]_{\mathsf{exp}}\ \text{'('}\mathit{exp_2}\ [\mathsf{s}]_{\mathsf{exp}}, ..., \mathit{exp_n}\ [\mathsf{s}]_{\mathsf{exp}}\ \text{')'} \\
\mathit{exp_1}\ \text{':'}\ \mathit{Name}\ \text{'('}\ \mathit{exp_2}, ..., \mathit{exp_n}\ \text{')'}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{exp_1}\ [\mathsf{s}]_{\mathsf{exp}}\ \text{':'}\ \mathit{Name}\ \text{'('}\mathit{exp_2}\ [\mathsf{s}]_{\mathsf{exp}}, ..., \mathit{exp_n}\ [\mathsf{s}]_{\mathsf{exp}}\ \text{')'} \\
\textbf{\$builtIn}\ \mathit{Name}\ \text{'('}\ \mathit{exp_1}, ..., \mathit{exp_n}\ \text{')'}\ [\mathsf{s}]_{\mathsf{exp}} & = \textbf{\$builtIn}\ \mathit{Name}\ \text{'('}\mathit{exp_1}\ [\mathsf{s}]_{\mathsf{exp}}, ..., \mathit{exp_n}\ [\mathsf{s}]_{\mathsf{exp}}\ \text{')'} \\
\text{'('}\ \mathit{exp}\ \text{')'}\ [\mathsf{s}]_{\mathsf{exp}} & = \text{'('}\ \mathit{exp}\ [\mathsf{s}]_{\mathsf{exp}}\ \text{')'} \\
\mathit{exp_1}\ \text{'['}\ \mathit{exp_2}\ \text{']'}\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{exp_1}[\mathsf{s}]_{\mathsf{exp}}\ \text{'['}\ \mathit{exp_2}[\mathsf{s}]_{\mathsf{exp}}\ \text{']'} \\
<\ \mathit{exp_1}\ ,\ ...\ ,\mathit{exp_n}\ >\ [\mathsf{s}]_{\mathsf{exp}} & = <\ \mathit{exp_1}\ [\mathsf{s}]_{\mathsf{exp}}\ ,\ ...\ ,\ \mathit{exp_n}\ [\mathsf{s}]_{\mathsf{exp}}\ > \\
\{\ \mathsf{field}_1\ ...\ \mathsf{field}_n\ \}\ [\mathsf{s}]_{\mathsf{exp}} & = \{\ (\mathsf{field}_1\ [\mathsf{s}]_{\mathsf{field}}) \\
 & \qquad ... \\
 & \quad (\mathsf{field}_n\ [\mathsf{s}]_{\mathsf{field}})\ \} \\
(\mathit{exp_1}\ \mathit{binop}\ \mathit{exp_2})\ [\mathsf{s}]_{\mathsf{exp}} & = \mathit{exp_1}\ [\mathsf{s}]_{\mathsf{exp}}\ \mathsf{binop}\ (\mathit{exp_2}\ [\mathsf{s}]_{\mathsf{exp}}) \\
(\mathit{unop}\ \mathit{exp})\ [\mathsf{s}]_{\mathsf{exp}} & = \mathsf{unop}\ (\mathit{exp}\ [\mathsf{s}]_{\mathsf{exp}})
\end{array}
$$

En el caso de los constructores de tablas, definimos una función de substitución especializada:

$$\_[\_]_{\mathsf{field}} : \mathsf{L(field)x S} \rightarrow \mathsf{L(field)}$$

$$\mathit{exp}\ [\mathsf{s}]_{\mathsf{field}} = \mathit{exp}\ [\mathsf{s}]_{\mathsf{exp}}$$

$$(\text{'['}\mathit{exp_1}\text{']'}\ \text{' = '}\ \mathsf{exp}_2\ )[\mathsf{s}]_{\mathsf{field}} = \text{'['}\mathit{exp_1}[\mathsf{s}]_{\mathsf{field}}\ \text{']'}\ \text{' = '}\ (\mathsf{exp}_2\ [\mathsf{s}]_{\mathsf{field}})$$

La expresión restante, la definición de una función, es la única con ocurrencias ligadoras de variables. De los substitutos de identificadores definidos por un mapeo s, debemos descartar aquellos que se refieren a identificadores que también aparecen como parámetros de la función, antes de aplicar la substitución en el cuerpo de la función, para respetar el alcance de la definición de los parámetros de la función. Si s es el mapeo que describe la substitución a realizar, denotaremos con s - { x_1,...,x_n } el mapeo que resulta de eliminar de s todos las substituciones que define para los identificadores en { x_1,...,x_n }. Con estos recursos mencionados, la substitución sobre la definición de una función se define como:

$$
\begin{array}{ll}
(\textbf{function}\ \mathit{Name}\ \text{'('}x_1\ ,...,x_n\ \text{')'} = & \textbf{function}\ \mathit{Name}\ \text{'('}\ x_1,...,x_n\ \text{')'} \\
\quad \mathit{block} & \mathit{block}\ [(x_{1'} \setminus e_1,..., x_{m'} \setminus e_m) - \{\ x_1,...,x_n\ \}]_{\mathit{block}} \\
\textbf{end})\ [(x_{1'} \setminus e_1,..., x_{m'} \setminus e_m)]_{\mathit{exp}} & \textbf{end}
\end{array}
$$

El caso de una substitución que no define un substituto para la expresión vararg, pero es aplicada a una función varar, se define del mismo modo.

El caso restante, relacionado con la definición de una función, tiene que ver con la aplicación de una substitución que define un substituto para la expresión vararg. Cada vez que una tal substitución es aplicada, sea s, a a la definición de una función f, el substituto para la expresión vararg es descartado, ya que, si f no es una función vararg, entonces la sintaxis no permite la presencia de la expresión vararg, en el cuerpo de f (y, por lo tanto, no necesitamos mantener, en s, el substituto de la expresión vararg). Pero, si f sí es una función vararg, las reglas de alcance de las

declaraciones de variables, determina que el significado de una expresión vararg, en el cuerpo de una función, es aquel definido por el prototipo de f. Por lo tanto, nuevamente debemos descartar, de s, el substituto de la expresión vararg. Por lo planteado, podemos entonces definir el caso de substitución analizado, en términos de los casos definidos anteriormente, del siguiente modo:

$$\mathit{functiondef}\,[x_1 \backslash e_1, ..., x_m \backslash e_m, \text{'...'} \backslash e_{m+1}]_{exp} = \mathit{functiondef}\,[x_1 \backslash e_1, ..., x_m \backslash e_m]_{exp}$$

La substitución en sentencias que no poseen ocurrencias ligadoras de variables se define como:

void $[s]_{block}$	$=$ **void**
functioncall $[s]_{block}$	$=$ *functioncall* $[s]_{exp}$
(*statement* ; *block*) $[s]_{block}$	$=$ *statement* $[s]_{block}$; *block* $[s]_{block}$
label { *block* } $[s]_{block}$	$=$ *label* { *block* $[s]_{block}$ }
do *block* **end** $[s]_{block}$	$=$ **do** *block*$[s]_{block}$ **end**
if *exp* **then** $block_1$ **else** $block_2$ **end** $[s]_{block}$	$=$ **if** *exp* $[s]_{exp}$ **then** $block_1[s]_{block}$ **else** $block_2$ $[s]_{block}$ **end**
(**break** *Name tuple*) $[s]_{block}$	$=$ **break** *Name* (*tuple* $[s]_{exp}$)
while *exp* **do** *block* **end** $[s]_{block}$	$=$ **while** *exp* $[s]_{exp}$ **do** *block* $[s]_{block}$ **end**
$(var_1, ..., var_n$ ' = ' $exp_1, ..., exp_m)$ $[s]_{block}$	$= var_1$ $[s]_{exp}, ..., var_n$ $[s]_{exp}$ ' = ' $(exp_1$ $[s]_{exp}), ..., (exp_m$ $[s]_{exp})$

Finalmente, tratamos la substitución sobre la única sentencia que tiene ocurrencias ligadores de variables, la sentencia **local** para introducir variables locales a un bloque dado. Las consideraciones hechas para definir la substitución en este caso, son las mismas realizadas para con la substitución sobre la definición de una función. Notar que, en la lista de expresiones que están siendo asignadas, como valor inicial de las variables locales declaradas, la substitución es realizada sin ninguna alteración de los substitutos que define:

(**local** x_1,...,x_n '=' *explist* **in** block **end**)$[(x_{1'} \setminus e_{1'}, ..., x_{m'} \setminus e_{m'})]$	$=$	**local** x_1 ,...,x_n '='*explist* $[(x_{1'} \setminus e_{1'}, ..., x_{m'} \setminus e_{m'})]$ **in** *block*$[(x_{1'} \setminus e_{1'}, ..., x_{m'} \setminus e_{m'})$ - { x_1,...,x_n }] **end**

6.3 Diseño de las nociones de reducción

Como fue mencionado en la sección 4, queremos obtener una descripción de la semántica de nuestro lenguaje, sobre la que sea fácil razonar. Requerimos entonces definiciones concisas, logradas con la mínima cantidad de recursos. Del mismo modo, sería deseable que, sin comprometer los objetivos iniciales, la semántica pueda ser facilmente mecanizada, aprovechando las posibilidades que ofrece una herramienta como PLT Redex, para explorar nuestro modelo en busca de inconsistencias en las definiciones propuestas. La semántica de reducciones de Felleisen-Hieb, con contextos de evaluación, nos permite definir una relación principal (que engloba toda la maquinaria necesaria para describir la semántica de programas completos), como resultado de la composición de otras relaciones más simples, las cuales modelan cada una, un aspecto específico de lenguaje. Esto nos permite lograr definiciones en las que mencionamos solo las piezas de información que son necesarias para especificar la operación deseada, y aislar así todos los aspectos que puede ser analizados de manera aislada. Entonces, por ejemplo, preferimos evitar cualquier mención a un almacenamiento, cuando describimos el comportamiento de sentencias y expresiones que no operan sobre el mismo. Aparte de la ayuda que esto supone para describir la semántica del lenguaje, el definir nociones de reducción especializadas en cierto tipo específico de construcción de nuestro lenguaje, nos permita enriquecer la mecanización del modelo con información específica que puede ser empleada por PLT Redex para realizar chequeos de consistencia de las definiciones de todas las piezas de la semántica (nociones de reducción, meta-funciones y gramática).

Como se mencionó anteriormente, en Lua existe una distinción entre sentencias y expresiones, la cual mantenemos en nuestro lenguaje, y constituye entonces, el otro criterio a considerar para definir nociones de reducción especializadas en cierto tipo específico de construcciones. Por esto, y lo expuesto en el párrafo anterior, dividimos las construcciones del lenguaje en las siguientes categorías, a los fines de especificar por separado su semántica:

- Expresiones:
  - Expresiones que no interactúan con almacenamientos. Su semántica será describa mediante la relación $\rightarrow^{e}$.
  - Expresiones que interactúan el almacenamiento de valores simples. Su semántica será descrita mediante la relación $\rightarrow^{e\_\sigma}$.
  - Expresiones que interactúan con el almacenamiento de objetos. Su semántica será descrita con la relación $\rightarrow^{e\_\theta}$.
  - Expresiones que interactúan con ambos tipos de almacenamientos. Su semántica será descrita mediante la relación $\rightarrow^{e\_\sigma\_\theta}$.
  - Expresiones que requieren de la intervención del mecanismo de meta-tablas. Su semántica será descrita mediante la relación $\rightarrow^{e-abnormal}$.
- Sentencias:
  - Sentencias que no interactuan con los almacenamientos. Su semántica será definida mediante la relación $\rightarrow^{s}$.
  - Sentencias que interactúan con el almacenamiento de valores simples. Su semántica será descrita mediante la relación $\rightarrow^{s\_\sigma}$
  - Sentencias que interactúan con el almacenamiento de objetos. Su semántica será descrita mediante la relación $\rightarrow^{s\_\theta}$.
  - Sentencias que requieren de la intervención del mecanismo de meta-tablas. Su semántica será descrita mediante la relación $\rightarrow^{s-abnormal}$.

6.4 Semántica de expresiones

Introduciremos primero las nociones de reducción que definen la semántica de expresiones, comenzando por aquellas cuya comportamiento puede ser explicado con la menor cantidad de información, yendo hacia aquellas que requieren de la intervención de los almacenamientos. Finalmente, describiremos cómo el mecanismo de meta-tablas maneja las situaciones especiales que involucran a las expresiones .

6.4.1 Expresiones que no interactúan con contextos

Como se mencionó, la semántica de estas expresiones será descrita mediante la relación $\rightarrow_e \subseteq \mathsf{L(exp)xL(exp)}$.

Operadores primitivos

Definiremos el comportamiento de los operadores primitivos a través de una función de interpretación δ, como es costumbre en la literatura del área. El dominio de esta función está definido por la siguiente unión:

$$\text{L(binop) x L(simplevalue) x L(exp)} \cup \text{L(unop) x (L(simplevalue)} \cup \text{L(tableconstructor))}$$

Su imagen estará en $\mathsf{L(simplevalue)}$. Es decir, nuestra función de interpretación mapeará tuplas que representan la aplicación de operadores primitivos, hacia valores que representan el resultado de la operación indicada. En la definición de δ, nos concentramos en darle significado sólo a aplicaciones correctas de operadores primitivos. Cualquier otro aspecto relativo a la aplicación de operadores primitivos (como coerción o errores de tipo) será manejado por nociones de reducción definidas luego.

La interpretación de operadores aritméticos es definida de manera directa. Dado el objeto sintáctico $\mathsf{n} \in \mathsf{L(Number)}$, denotemos con $\hat{\mathsf{n}} \in \mathbb{R}$ el correspondiente número que es representado por n. Y, dado el número$\mathsf{r} \in \mathbb{R}$, sea $\ulcorner \mathsf{r} \urcorner$ el correspondiente objeto sintáctico que lo representa en nuestro lenguaje (es decir, $\ulcorner \mathsf{r} \urcorner \in \mathsf{L(Number)}$)[2]. Entonces, la interpretación de la aplicación de operadores aritméticos se puede describir simplemente cómo:

[2]Dejamos por ahora, fuera de esta presentación, aspectos relacionados al conjunto de números que efectivamente se pueden representar en Lua

$$\delta(+, n_1, n_2) = \ulcorner\ \hat{n_1} + \hat{n_2}\ \urcorner$$

$$\delta(-, n_1, n_2) = \ulcorner\ \hat{n_1} - \hat{n_2}\ \urcorner$$

$$\delta(*, n_1, n_2) = \ulcorner\ \hat{n_1} * \hat{n_2}\ \urcorner$$

$$\delta(/, n_1, n_2) = \ulcorner\ \hat{n_1} / \hat{n_2}\ \urcorner$$

$$\delta(\hat{}, n_1, n_2) = \ulcorner\ \hat{n_1}^{\hat{n_2}}\ \urcorner$$

$$\delta(\%, n_1, n_2) = \ulcorner\ \hat{n_1} \%\ \hat{n_2}\ \urcorner$$

$$\delta(-, n) = \ulcorner\ -\hat{n}\ \urcorner$$

En las ecuaciones previas, prestarle atención al hecho de que estamos describiendo la semántica de expresiones que involucran símbolos de la aritmética (en los términos izquierdos de las ecuaciones), en términos de las correspondientes operaciones aritméticas, a los cuales nos referimos empleando los mismos símbolos (en los términos derechos de las ecuaciones).

La comparación de orden natural, entre números y cadenas de caracteres, se puede describir también de forma directa, distinguiendo el tipo de operandos que se quiere comparar (nuevamente, recordar que, por el momento, nos limitamos a describir de manera denotacional la semántica de la aplicación de operadores primitivos a operandos del tipo correcto):

$$\delta(<, n_1, n_2) = \begin{cases} \textbf{true} & \text{si } \hat{n_1} < \hat{n_2} \\ \textbf{false} & \text{c.c.} \end{cases}$$

$$\delta(<=, n_1, n_2) = \begin{cases} \textbf{true} & \text{si } \hat{n_1} \leq \hat{n_2} \\ \textbf{false} & \text{c.c} \end{cases}$$

$$\delta(<, s_1, s_2) = \begin{cases} \textbf{true} & \text{si } s_1 < s_2 \text{ en orden lexicografico} \\ \textbf{false} & \text{c.c.} \end{cases}$$

$$\delta(<=, s_1, s_2) = \begin{cases} \textbf{true} & \text{si } s_1 \leq s_2 \text{ en orden lexicografico} \\ \textbf{false} & \text{c.c.} \end{cases}$$

La comparación de igualdad, opera sobre valores de cualquier tipo. En general, realiza solo comparación estructural. La excepción a esto se presenta cuando se comparan objetos. En tales casos, en Lua 5.2, la comparación de igualdad realiza comparación de tokens, estos es, compara las referencias a los objetos, y no los objetos mismos. En nuestra semántica, dejamos que la función δ describa el significado de la comparación solamente en términos de la comparación estructural, entre la cadena de símbolos que representa a los valores que se comparan. Los demás aspectos necesarios para modelar de manera completa la forma en la que opera la comparación de valores en Lua, es delegada para las correspondientes nociones de reducción:

$$\delta(==, sv_1, sv_2) = \begin{cases} \textbf{true} & \text{si } sv_1 = sv_2 \\ \textbf{false} & \text{c.c.} \end{cases}$$

En el caso particular de la comparación de valores de tipo función, como se verá en 6.4.3, por la forma en la que construimos clausuras (valores de tipo función), se garantiza que, diferentes referencias a clausuras, indican diferencias entre las funciones involucradas (en su definición o comportamiento) de entre las que que son tenidas en cuenta en Lua 5.2 al momento de construir nuevas clausuras[3], y que tiene impacto en la semántica de la comparación de clausuras.

Los operadores lógicos son interpretados de la siguiente forma:

$$\delta(\textbf{or}, sv, exp) = \begin{cases} sv & \text{si sv} \neq \textbf{false} \wedge \text{sv} \neq \textbf{nil} \\ exp & \text{c.c.} \end{cases}$$

$$\delta(\textbf{and}, sv, exp) = \begin{cases} sv & \text{si sv} = \textbf{false} \vee \text{sv} = \textbf{nil} \\ exp & \text{c.c.} \end{cases}$$

$$\delta(\textbf{not}, sv) = \begin{cases} \textbf{true} & \text{si sv} = \textbf{false} \vee \text{sv} = \textbf{nil} \\ \textbf{false} & \text{c.c.} \end{cases}$$

Notar que, en el caso de los operadores **and** y **or**, el segundo operando involucrado es una expresión, y no necesariamente un valor simple. La intención es la de otorgarle a los operadores una semántica de evaluación de cortocircuito, como ocurre en Lua. La pieza faltante, para completar la descripción de tal comportamiento, es la regla de reducción que describe cómo se aplica la función de interpretación, la cual impone tal aplicación inmediatamente después de que el primer operando ha sido completamente evaluado (regla definida más adelante).

Sólo resta describir la interpretación de los operados unarios:

$\delta(.., s_1, s_2) = s_1 s_2$ (es decir, la concatenación de la cadena s_1 con la cadena s_2)

$\delta(\#, s)$ = cantidad de caracteres de la cadena s (esto equivale a la cantidad de bytes de una codificación de la cadena s, en la que cada caracter es representado por un byte; valor que es en realidad lo retornado por el operador # de Lua)

El operador # puede ser también empleado para obtener la longitud de una tabla, cuando tal tabla es una "secuencia": una tabla cuyo conjunto de claves numéricas positivas es igual a { 1,...,m }, para algún número natural m[4]. En tal caso, siendo t una tabla que satisface tal condición:

$\delta(\#, t) = \text{m}$

Dejamos sin especificar la interpretación de este operador. Luego, cuando presentemos la mecanización de nuestra semántica, daremos una descripción del mismo.

Reglas reducción para operadores aritméticos

Definiremos las reglas de reducción de la noción de reducción $\rightarrow^e$, que describen la semántica de expresiones que representan la aplicación de operadores aritméticos. Comenzaremos tratando las situaciones en las que tales operaciones son aplicadas sobre operandos de tipo correcto. Posteriormente, describiremos cómo es realizada la coerción, y hacia el final de la sección, trataremos las situaciones especiales que son manejadas por el mecanismo de meta-tablas.

La semántica de expresiones que representan la aplicación de operadores aritméticos binarios, sobre argumentos de tipo esperado, se reduce a reducciones como la siguiente:

$$\frac{}{(Number_1 \ + \ Number_2) \ \rightarrow^e \ \delta(+, Number_1, Number_2)} \text{ E-Addition}$$

[3] ver [17], sección 8.1

[4] Extraído de [17], sección 3.4.6

Los casos restantes, que involucran a los demás operadores aritméticos binarios (-,*,/, ^ ,%), se definen del mismo modo.

La semántica de la aplicación de la negación matemática, bajo condiciones normales, se define de un modo similar:

$$\frac{}{(-\ \mathit{Number})\ \rightarrow^{e}\ \delta(-, \mathit{Number})}\ \text{E-Negation}$$

Reglas de reducción para operadores de relación

La semántica de expresiones que representan la comparación de orden natural, entre números y cadenas de caracteres, no depara sorpresas:

$$\frac{}{\mathit{Number}_1\ <\ \mathit{Number}_2\ \rightarrow^{e}\ \delta(<, \mathit{Number}_1, \mathit{Number}_2)}\ \text{E-LessThanNumber}$$

$$\frac{}{\mathit{String}_1\ <\ \mathit{String}_2\ \rightarrow^{e}\ \delta(<, \mathit{String}_1, \mathit{String}_2)}\ \text{E-LessThanString}$$

La descripción de la semántica de expresiones que involucran al operador "<=" es análoga.
Necesitamos incluir en nuestro lenguaje a los operadores de comparación > y >=, inclusive a pesar de que en Lua, internamente, las expresiones que involucran a los mismos son traducidas a las equivalentes expresiones que emplean las comparaciones < y <=, según corresponda. La necesidad en mantener en nuestro lenguaje a las comparaciones > y >=, radica en que, a pesar de ser expresadas, internamente, en términos de < y <=, la evaluación de las expresiones comparadas, sigue siendo de izquierda a derecha. Podemos, sin embargo, demorar la evaluación de la expresión, en términos de < y <=, hasta que las expresiones comparadas hayan sido completamente evaluadas, como se indica a continuación:

$$\frac{}{\mathit{sv}_1\ >\ \mathit{sv}_2\ \rightarrow^{e}\ \mathit{sv}_2\ <\ \mathit{sv}_1}\ \text{E-GreaterThan}$$

$$\frac{}{\mathit{sv}_1\ >=\ \mathit{sv}_2\ \rightarrow^{e}\ \mathit{sv}_2\ =<\ \mathit{sv}_1}\ \text{E-GreaterOrEqualThan}$$

En Lua es posible alterar la forma en la que tablas y user-data son comparados, a través del empleo del mecanismo de meta-tablas. Este mecanismo es disparado como resultado de una comparación de igualdad no exitosa, sobre valores que sean del mismo tipo, siendo estos tablas o user-data. Dejaremos su tratamiento para la sección final en donde se describe cómo trabaja el mecanismo de meta-tablas, sobre expresiones. Por el momento, desde la relación $\rightarrow^{e}$ podemos describir la semántica de la siguiente situación, que involucra a la comparación de igualdad:

$$\frac{\delta(==, \mathit{sv}_1, \mathit{sv}_2) = \textbf{true}}{(\mathit{sv}_1\ ==\ \mathit{sv}_2)\ \rightarrow^{e}\ \textbf{true}}\ \text{E-EqualitySuccess}$$

Semántica de operadores lógicos

Como en Lua, en nuestro lenguaje podemos asignarle significado a una expresión que involucra el uso de operadores lógicos, sobre operandos de cualquier tipo. La regla de reducción siguiente, que describe la aplicación de la función de interpretación δ, en el caso de una expresión que involucra al operador **and**, completa la descripción de la semántica de cortocircuito de este operador, como se mencionó al definir δ:

$$\frac{}{(\mathit{sv}\ \textbf{and}\ \mathit{exp})\ \rightarrow^{e}\ \delta(\textbf{and}, \mathit{sv}, \mathit{exp})}\ \text{E-And}$$

Notar cómo se hace uso de la categoría sintáctica que representa a los valores simples, para terminar de imponer la evaluación de cortocircuito del operador **and**. Si el operador involucrado es **or**, la regla de reducción es análoga.

La regla de reducción correspondiente para el operador **not**, es la esperada:

$$\frac{}{(\textbf{not}\ sv)\ \rightarrow^{e}\ \delta(\textbf{not},\ sv)}\ \text{E-Not}$$

Semántica de operadores sobre cadenas de caracteres

Nuevamente, las reglas son las esperadas:

$$\frac{}{(\#\ String)\ \rightarrow^{e}\ \delta(\#, String)}\ \text{E-StringLength}$$

$$\frac{}{(String_1\ ..\ String_2)\ \rightarrow^{e}\ \delta(.., String_1, String_2)}\ \text{E-StringConcatenation}$$

La aplicación del operador # sobre una tabla, comienza disparando el mecanismo de meta-tablas, por lo cual, será descrito posteriormente, en la sección dedicada a meta-tablas.

Coerción

Lua 5.2 realiza conversión de tipos de manera implícita, entre números y cadenas, en ambos sentidos. Cuando un operador aritmético es aplicado sobre una cadena, se realiza una conversión automática de la misma hacia un número. La cadena involucrada debe ser una representación textual de un número decimal o hexadecimal, la cual puede estar en notación científica. En Lua 5.2, la conversión de cadenas, con números decimales, hacia números es realizada por el procedimiento strtod, de la librería glibc. Este mecanismo, no forma, por lo tanto, parte de Lua, que requiera entonces ser modelado. Para cadenas con dígitos hexadecimales, se emplea un procedimiento propio de Lua[5]. En nuestra semántica, representaremos ambos procedimientos con la meta-función toNumber, la cual dejaremos por el momento sin especificar. Se experimentará con una definición de la misma cuando se mecanice la semántica, usando PLT Redex. En tal caso, es también posible aprovechar las posibilidades que brinda el lenguaje Racket, sobre el cual está definido PLT Redex.

Al describir el mecanismo de coerción, será necesario preguntar por el tipo de un valor dado. A tal fin, definimos la siguiente meta-función:

> simpvaltype : L(simplevalue) → {"nil", "boolean", "number", "string", "objref" } : retorna una descripción del tipo del valor recibido

Los objetos requieren primero, para determinar su tipo, observar el almacenamiento de objetos, esto a raíz de siempre manejamos objetos a través de referencias a ellos. Como aquí no estamos tratando con almacenamientos, y como en las siguientes reducciones no necesitamos referirnos al tipo de los objetos, definimos nuestra meta-función simpvaltype con la suficiente información como para que reconozca el tipo de cualquier valor, que no sea un objeto. En caso de recibir una referencia a un objeto, simpvaltype sólo indica "objref", como el tipo determinado. Para el objetivo de definir el mecanismo de coerción de Lua, es suficiente con manejar esa información, como quedará claro en las siguientes reglas de reducción.

En una expresión aritmética, cuando se aplica coerción a cadenas, debemos chequear si esta puede realmente ser convertida a un número. Si ese no es el caso, o si estamos en presencia de valores que no sean cadenas o números, en el lugar de un operando numérico, entonces, el mecanismo de meta-tablas es empleado para intentar darle algún significado a tal expresión. Ese proceso será descrito luego. El caso más simple de coerción de una cadena a número, cuando está como operando en una suma, se expresa cómo:

$$\frac{\begin{array}{c}\text{simpvaltype}(sv_1) \neq \text{"number"} \ \vee\ \text{simpvaltype}(sv_2) \neq \text{"number"} \\ sv_3 = \text{toNumber}(sv_1)\ \wedge\ sv_4 = \text{toNumber}(sv_2) \\ \text{simpvaltype}(sv_3) = \text{"number"}\ \wedge\ \text{simpvaltype}(sv_4) = \text{"number"}\end{array}}{(sv_1\ +\ sv_2)\ \rightarrow^{e}\ (sv_3\ +\ sv_4)}\ \text{E-AdditionCoercion}$$

[5]Ver el procedimiento lua_strx2number del archivo lobject.c, del código fuente de Lua 5.2

La coerción sobre operandos en las restantes expresiones aritméticas, se describe de manera análoga.

Finalmente, cuando un número es usado en el lugar de una cadena, en la concatenación de cadenas, nuevamente se realiza una coerción. En Lua 5.2, la conversión es realizada por el procedimiento sprintf, de la librería glibc. Dejamos que la meta-función toString denote tal procedimiento. Nuevamente, quedará sin ser especificada. Luego, en la mecanización de la semántica, se experimentará con una definición de la misma:

$$\frac{\begin{array}{c}\mathsf{simpvaltype}(sv_1) \neq \text{"string"} \vee \mathsf{simpvaltype}(sv_2) \neq \text{"string"} \\ sv_3 = \mathsf{toString}(sv_1) \ \wedge \ sv_4 = \mathsf{toString}(sv_2) \\ \mathsf{simpvaltype}(sv_3) = \text{"string"} \ \wedge \ \mathsf{simpvaltype}(sv_4) = \text{"string"}\end{array}}{sv_1 \ .. \ sv_2 \ \rightarrow^e \ sv_3 \ .. \ sv_4} \text{ E-StringConcatCoercion}$$

Nuevamente, si un valor que no sea numérico o una cadena, aparece en el lugar de un operando que debería ser una cadena, entonces el mecanismo de meta-tablas indica un camino para re-interpretar tal expresión. Esto comenzará a ser tratado en el próximo apartado.

Identificación de expresiones con operadores primitivos, que disparan el mecanismo de meta-tablas

Nuestro descripción del mecanismo de meta-tablas, se divide en 2 piezas: primero están las nociones de reducción que reconocen las situaciones en las cuales el mecanismo es empleado, e indican esto etiquetando la correspondiente sentencia o expresión. Segundo, una noción de reducción particular ($\rightarrow^{\mathsf{e\_abnormal}}$ para expresiones, y $\rightarrow^{\mathsf{s\_abnormal}}$ para sentencias) relaciona frases equipadas con la información necesaria para describir cómo efectivamente el mecanismo de meta-tablas opera para resolver. En particular, la información que necesitamos consiste en el almacenamiento de objetos, en donde las meta-tablas para los distintos valores (en el caso de tablas y user-data) o tipos están almacenadas, y la correspondiente etiquetado adosada a la expresión o sentencia, que nos informa de la situación anormal encontrada.

La definición de la segunda pieza de nuestro tratamiento del mecanismo de meta-tablas será presentada en la sección 6.4.5. Por el momento, desde la relación $\rightarrow^e$ vamos a identificar qué expresiones, con operadores primitivos, requieren de la intervención del mecanismo de meta-tablas. Tales frases, a las cuales no podemos darles significado sin el uso del mecanismo de meta-tablas, tienen que ver, en general, con la aplicación de operadores sobre operandos de tipo incorrecto (inclusive tras la intervención del mecanismo de coerción).

Si, por ejemplo, en una suma, alguno de sus operandos no es un número, ni una cadena que pueda ser convertida a un número, entonces alertamos sobre esta situación, según lo indicado por la siguiente regla, de nombre E-AlertAdditionWrongOperands:

$$\frac{\begin{array}{c}\mathsf{simpvaltype}(\ sv_1\ ) \neq \text{"number"} \vee \mathsf{simpvaltype}(\ sv_2\ ) \neq \text{"number"} \\ \mathsf{simpvaltype}(\ \mathsf{toNumber}(\ sv_1\ )\ ) \neq \text{"number"} \vee \mathsf{simpvaltype}(\ \mathsf{toNumber}(\ sv_2\ )\ ) \neq \text{"number"}\end{array}}{sv_1 \ + \ sv_2 \ \rightarrow^e \ (sv_1 \ + \ sv_2)_{\mathbf{AdditionWrongOperands}}}$$

Las restantes situaciones especiales, que involucran a las operaciones aritméticas, son manejadas del mismo modo.

Siguiendo la misma idea, la reducción que alerta sobre la aplicación de operadores de cadenas, sobre operandos de tipo incorrecto, se define como: Following the same pattern, the reduction that alerts about the application of string operators over operands of the wrong type, is defined as:

$$\frac{\begin{array}{c}\mathsf{simpvaltype}(\ sv_1\ ) \neq \text{"string"} \wedge \mathsf{simpvaltype}(\ sv_1\ ) \neq \text{"number"} \\ \vee \\ \mathsf{simpvaltype}(\ sv_2\ ) \neq \text{"string"} \wedge \mathsf{simpvaltype}(\ sv_2\ ) \neq \text{"number"}\end{array}}{sv_1 \ .. \ sv_2 \ \rightarrow^e \ (sv_1 \ .. \ sv_2)_{\mathbf{StringConcatWrongOperands}}} \text{ E-AlertStringConcatWrongOperands}$$

$$\frac{\mathsf{simpvaltype}(\ sv_1\ ) \neq \text{"string"}}{\#\ sv\ \rightarrow^e\ (\#\ sv)_{\mathbf{StringLengthWrongOperand}}}\ \text{E-AlertStringLengthWrongOperand}$$

Como fue mencionado en la sección 6.4.1, cuando una comparación de igualdad no resulta exitosa, entonces, se dispara el mecanismo de meta-tablas. Desde $\rightarrow^e$ etiquetamos entonces la correspondiente comparación que falló, para su tratamiento a posterior:

$$\frac{\delta(==, sv_1, sv_2) = \mathbf{false}}{sv_1\ ==\ sv_2\ \rightarrow^e\ (sv_1\ ==\ sv_2)_{\mathbf{EqualityFail}}}\ \text{E-AlertEqualityFail}$$

Finalmente, las situaciones especiales en donde están involucrados los operadores relacionales son las siguientes:

$$\frac{\begin{array}{c}\mathsf{simpvaltype}(sv_1) \neq \text{"string"} \vee \mathsf{simpvaltype}(sv_2) \neq \text{"string"} \\ \mathsf{simpvaltype}(sv_1) \neq \text{"number"} \vee \mathsf{simpvaltype}(sv_2) \neq \text{"number"}\end{array}}{sv_1\ <\ sv_2\ \rightarrow^e\ (sv_1\ <\ sv_2)_{\mathbf{LessThanFail}}}\ \text{E-AlertLessThanFail}$$

$$\frac{\begin{array}{c}\mathsf{simpvaltype}(sv_1) \neq \text{"string"} \vee \mathsf{simpvaltype}(sv_2) \neq \text{"string"} \\ \mathsf{simpvaltype}(sv_1) \neq \text{"number"} \vee \mathsf{simpvaltype}(sv_2) \neq \text{"number"}\end{array}}{sv_1\ <=\ sv_2\ \rightarrow^e\ (sv_1\ <=\ sv_2)_{\mathbf{LessThanOrEqualFail}}}\ \text{E-AlertLessThanOrEqualFail}$$

Tuplas

Como fue mencionado en 4.1, las tuplas modelarán listas de valores que son retornados desde una función, o que representan el valor que toma la expresión vararg. Hay 2 operaciones a modelar, sobre tuplas: extraer el primer valor de la tupla (lo que ocurre cuando la tupla aparece en contextos en donde en donde sólo un valor es requerido, como en la guarda de un condicional), o agregar los valores de la tupla al final de una lista de expresiones (como cuando la tupla aparece al final de una lista de expresiones que están por ser asignadas a una lista de variables).

Por lo tanto, la operación que se realiza sobre la tupla depende del contexto inmediato en el que aparece. Dar una descripción formal de ese comportamiento sensible al contexto puede ser sencillo, haciendo uso de contextos de evaluación que representen las situaciones en donde una u otra operación sobre tuplas debe ser realizada.

Cuando tenemos una tupla de valores no vacía, los pasos de reducción, que describen las operaciones mencionadas, son los siguientes:

$$\frac{}{E_t[\![< sv_1, ..., sv_n >]\!]\ \rightarrow^e\ E_t[\![sv_1]\!]}\ \text{E-TruncateNonEmptyTuple}$$

$$\frac{}{E_u[\![< sv_1, ..., sv_n >]\!]\ \rightarrow^e\ E_u[\![sv_1, ..., sv_n]\!]}\ \text{E-UnwrapNonEmptyTuple}$$

Luego, en cualquier contexto en donde, sobre una tupla no vacía, se realiza algunas de las 2 operaciones mencionadas, a una tupla vacía se la convierte al valor **nil**:

$$\frac{E \in \mathsf{L}(\mathsf{E_t}) \vee E \in \mathsf{L}(\mathsf{E_u})}{E[\![\mathbf{empty}]\!]\ \rightarrow^e\ E[\![\mathbf{nil}]\!]}\ \text{E-TruncateOrUnwrapEmptyTuple}$$

Recordar que el valor especial **nil** representa la "ausencia de un valor", en contextos en donde alguno es requerido (como observar lo que retorna una función, u observar el valor de una variable

que no ha sido asignada aún).

Hay otra situación, que no está directamente relacionada con tuplas, pero sí con valores retornados de una función, por lo que la describimos aquí, para mantener cierta cohesión. En Lua, es posible definir una función que no devuelve valores a través de la sentencia **return** (y, por lo tanto, la reducción de una llamada a tal función culminará en la sentencia especial **void**). En contextos en donde un valor es requerido, una llamada a una tal función no es distinguible de una llamada a una función que termina con una sentencia **return** con una lista de valores vacía (lista la cual, como se verá en el capítulo 7, se modelará usando una tupla vacía). Por lo tanto, en nuestro lenguaje, así como ocurre cuando trabajamos con tuplas vacías, una llamada a función que reduce a **void**, y es usada en el lugar de una expresión, es tratada como el valor **nil**:

$$\frac{E \in \mathsf{L}(\mathsf{E_t}) \vee E \in \mathsf{L}(\mathsf{E_u})}{E[\![\textbf{void}]\!] \ \rightarrow^{e} \ E[\![\textbf{nil}]\!]} \quad \text{E-ConvertVoidToNil}$$

Expresiones en paréntesis

El truncado que se realiza sobre una lista de expresiones, cuando esta está entre paréntesis, está contemplado como un caso especial de la operación general de extraer el primer elemento de los valores de una tupla (agregando a '(' hole ')' como contexto de $\mathsf{L}(\mathsf{E_t})$). Lo único que resta por describir, relativo al comportamiento de una expresión entre paréntesis, es qué ocurre cuando la expresión reduce a un único valor. En tal caso, nada interesante ocurre:

$$\frac{\mathsf{simpvaltype}(\mathit{simplevalue}) \neq \text{"Tuple"}}{(\mathit{simplevalue}) \ \rightarrow^{\mathsf{e}} \ \mathit{simplevalue}} \quad \text{E-ParenthesisAnyOtherValue}$$

Llamadas a métodos

Lua provee una forma sintáctica especial, *prefixexp* ':' *Name args*, la que puede ser interpretada como una llamada al método *Name*, del objeto *prefixexp*. La semántica de esa llamada dicta que el procedimiento *Name* recibe como primer argumento al objeto *prefixexp*, y este objeto es evaluado una única vez. En conjunción con clausuras de primera clase, lo que permite, entre otras cosas, que sean almacenadas en tablas, vemos que es posible modelar naturalmente conceptos del paradigma orientado a objetos, como la noción de clases, objetos y algún mecanismo de despacho de llamadas a métodos.

La forma especial de llamada a procedimiento mencionada, requiere de un tratamiento especial, ya que su semántica no puede ser simplemente descrita en términos de una traducción a una llamada ordinaria a procedimiento, pasando como primer argumento el objeto, debido a que este objeto debe ser evaluado una sola vez. Por esa razón, mantenemos en nuestro lenguaje esa forma sintáctica, y definimos su semántica, en parte, con la siguiente regla de reducción, que la re-interpreta como una llamada ordinaria, pero una vez que ha sido evaluado el objeto para el cual se está llamando el método :

$$\frac{}{\mathit{sv} \text{ " : " } \mathit{Name}\ (\mathit{exp}_1, ..., \mathit{exp}_n) \ \rightarrow^{\mathsf{e}} \ \mathit{sv}[\text{"}\mathit{Name}\text{"}](\mathit{sv}, \mathit{exp}_1, ..., \mathit{exp}_n)} \quad \text{E-MethodCall}$$

La semántica de la llamada a procedimiento en sí, será descrita en 6.4.4.

Servicios built-in

Identificamos los servicios principales provistos por la librería estándar de Lua, para agregarlos a nuestro lenguaje, con su respectiva semántica. Dejamos los restantes servicios provistos por la librería (e inclusive, ciertos detalles de aquellos que modelamos a continuación) como implementaciones en nuestro lenguaje, definidas en el entorno en el que un programa es ejecutado (lo cual será definido en el capítulo 8).

Para definir la semántica de los servicios de la librería estandar, abstraemos en meta-funciones aquellos principales, y brindamos una forma sintáctica especial para hacer referencia a los mismos. Expresiones de la forma **$builtIn** *Name* (*explist*) serán interpretadas como llamadas al servicio

básico *Name*. Al definirlos, nos concentraremos en la descripción de los mismos, en condiciones normales. También, esas formas sintácticas estarán dentro de definiciones de procedimientos, que servirán de envoltorio de las mismas, y que completarán su definición, agregando, de ser necesario, chequeo de tipo y cantidad de argumentos recibidos. También, estos procedimientos serán parte de las definiciones del entorno en donde un programa se ejecuta. Luego, se determinará por las reglas de alcance estático de la definición de variables, cuándo una llamada a un procedimiento se refiere a un servicio brindado por la librería estándar, o a un procedimiento definido por el usuario.

En esta sección daremos tratamiento formal a aquellos servicios que podemos describir sin emplear almacenamientos. Debido a que la intención es la de modelar el resultado de una llamada a procedimiento, necesitamos empaquetar en una tupla aquellos resultados que queremos devolver. Listamos los servicios que vamos a modelar, haciendo referencia al nombre que el servicio tiene en la librería estándar de Lua:

- error: la generación de mensajes de error se hará a través de este servicio. El comportamiento de un mensaje de error será descrito en 6.5.2:

$$\frac{}{\textbf{\$builtIn}\ \mathsf{error}\ (sv) \rightarrow^{\mathsf{e}}\ \textbf{error}\ sv}\ \text{E-BuiltInError}$$

- table.pack: si bien en esta propuesta inicial de la semántica de Lua, no aspiramos a una descripción exhaustiva de su librería estándar, debemos agregar ciertos procedimientos que, aunque no reportan un interés particular que justifique el considerarlos en este primer modelo, son muy usados en la suite de tests de Lua, lo que nos obliga a agregarlos para poder cubrir una mayor cantidad de casos de tests. Este es el caso del servicio pack, de la librería table. La definición que damos a continuación es lo suficientemente clara, respecto a qué servicio estamos definiendo. Comenzamos describiendo el mismo a través de una función:

$$\begin{array}{l}\mathsf{pack} : \mathsf{L(tuple)} \rightarrow \mathsf{L(evaluatedtable)} \\ \quad \mathsf{next}(< \mathsf{sv}_1,\ldots,\mathsf{sv}_k >) = \{\ [\ 1\ ] = \mathsf{sv}_1,\ \ldots,[\ \ulcorner k \urcorner\ ] = \mathsf{sv}_k,\ [\text{"n"}] = \ulcorner k \urcorner\ \} \\ \quad \mathsf{next}(\textbf{empty}) = \{\ [\text{"n"}] = 0\ \}\end{array}$$

Recordar que, dado $\mathsf{k} \in \mathbb{R}$, entonces $\ulcorner k \urcorner \in \mathsf{L(number)}$ es el objeto sintáctico que representa al número k, en nuestro lenguaje.
Definimos ahora las semántica de la correspondiente forma sintáctica, que nos permite acceder al servicio anterior:

$$\frac{\begin{array}{c} n >= 1\ \wedge\ \mathit{evaluatedtable} = \mathsf{pack}(\ < \mathit{sv}_1, \ldots, \mathit{sv}_n >) \\ \vee \\ n = 0\ \wedge\ \mathit{evaluatedtable} = \mathsf{pack}(\textbf{empty}) \end{array}}{\textbf{\$builtIn}\ \mathsf{tablePack}\ (\mathit{sv}_1, \ldots, \mathit{sv}_n) \rightarrow^{\mathsf{e}} < \mathit{evaluatedtable} >}\ \text{E-BuiltInTablePack}$$

- pcall: modelamos la llamada a este servicio, en condiciones normales. El mismo nos permite ejecutar una función en modo protegido, lo que significa que cualquier error que ocurra durante la ejecución de la función, es capturado e informado. La descripción de cómo opera el modo protegido, se dará en la sección 6.5.2:

$$\frac{}{\textbf{\$builtIn}\ \mathsf{pcall}\ (\mathit{sv}_1, \mathit{sv}_2, \ldots, \mathit{sv}_n) \rightarrow^{\mathsf{e}}\ (\mathit{sv}_1\ (\mathit{sv}_2, \ldots, \mathit{sv}_n))_{\textbf{ProtectedMode}}}\ \text{E-BuiltInAssertFail}$$

- rawequal: este servicio nos permite realizar una comparación de igualdad, sin disparar el mecanismo de meta-tablas. La definimos simplemente como una forma de acceder de manera directa a la interpretación de la comparación de igualdad, hecha por δ:

$$\frac{\mathit{sv}_3 = \delta(==,\ \mathit{sv}_1,\ \mathit{sv}_2)}{\textbf{\$builtIn}\ \mathsf{rawEqual}\ (\mathit{sv}_1,\ \mathit{sv}_2) \rightarrow^{\mathsf{e}} < \mathit{sv}_3 >}\ \text{E-BuiltInRawequal}$$

- select: este procedimiento de la librería estándar sirve como selectora de la lista de argumentos que recibe, y es útil para la definición de modismos que trabajan con la expresión vararg. Abstraemos su comportamiento en la siguiente función:

$$\text{select} : \mathsf{L}(\text{tuple}) \to \mathsf{L}(\text{tuple})$$

$$\text{select}(< sv_1, sv_2, ..., sv_n >) = \begin{cases} < sv_{s\hat{v}_1}, ..., sv_n > & \textbf{if } \text{simpValType}(\text{sv}_1) = \text{"number"} \\ & \quad \textbf{and} \\ & \quad 1 <= \text{s}\hat{\text{v}}_1 <= \text{n} - 1 \\ < \textbf{nil} > & \textbf{if } \text{simpValType}(\text{sv}_1) = \text{"number"} \\ & \quad \textbf{and} \\ & \quad n - 1 < s\hat{v}_1 \\ < sv_{n-1+s\hat{v}_1}, ..., sv_n > & \textbf{if } \text{simpValType}(\text{sv}_1) = \text{"number"} \\ & \quad \textbf{and} \\ & \quad -(\text{n} - 1) <= \text{s}\hat{\text{v}}_1 <= -1 \\ < sv_2, ..., sv_n > & \text{sv}_1 = \text{"\#"} \end{cases}$$

Finalmente, permitimos acceder a este servicio, de la forma descrita por la regla siguiente, de nombre E-BuiltInSelectNormal:

$$\frac{\begin{array}{c}(\text{simpValType}(sv_1) = \text{"number"} \\ \wedge \\ [(1 <= s\hat{v}_1 <= n - 1) \ \vee \ (n - 1 < s\hat{v}_1) \ \vee \ (-(n - 1) <= s\hat{v}_1 <= -1)]) \\ \vee \\ sv_1 = \text{"\#"}\end{array}}{\textbf{\$builtIn} \ \text{select} \ (sv_1, sv_2, ..., sv_n) \to^{\text{e}} < sv_{s\hat{v}_1}, ..., sv_n >}$$

Con motivo de la dificultad para saber, en el procedimiento que envuelve a esta forma sintáctica, la cantidad de argumentos que recibiremos, agregamos una regla (de nombre E-BuiltInSelectError), que maneja las situaciones de error que pueden producirse al usar este servicio:

$$\frac{\begin{array}{c}sv_1 \neq \text{"\#"} \\ \text{simpValType}(sv_1) = \text{"number"} \ \Rightarrow \ sv_1 = 0 \ \vee \ s\hat{v}_1 < -(n - 1)\end{array}}{\textbf{\$builtIn} \ \text{select} \ (sv_1, sv_2, ..., sv_n) \to^{\text{e}} \ \textbf{\$builtIn} \ \text{error} \ (\text{"bad argument \#1 to 'select'"})}$$

- tonumber: proveemos de una manera de acceder a la función de conversión hacia números:

$$\frac{sv_2 = \text{toNumber}(sv_1)}{\textbf{\$builtIn} \ \text{toNumber} \ (sv_1) \to^{\text{e}} < sv_2 >} \ \text{E-BuiltInToNumber}$$

- tostring: realizamos lo mismo con la función de conversión hacia cadenas de caracteres. El servicio tostring de la librería de Lua, es más complejo que esto, aunque los detalles restantes pueden describirse con código en nuestro lenguaje, por lo que será definido luego, en la sección 8:

$$\frac{sv_2 = \text{toString}(sv_1)}{\textbf{\$builtIn} \ \text{toString} \ (sv_1) \to^{\text{e}} < sv_2 >} \ \text{E-BuiltInToString}$$

6.4.2 Expresiones que interactúan con el almacenamiento de valores simples

Agregamos una pieza de información, el almacenamiento de valores simples (σ), para estar en condiciones de darle tratamiento a las expresiones cuya semántica operacinal depende de estos almacenamientos. Introducimos la relación $\rightarrow^{e\_\sigma}$, que relaciona frases de la forma "σ ; *exp*".

Des-referenciado de referencias simpvalref

Como se enunció en 4.2, creamos y des-referenciamos de manera implícita referencias a valores simples. La necesidad de definir nuevas referencias de este tipo, aparece en la descripción de las llamadas a función, para precisar describir el pasaje de argumentos, y en la introducción de variables locales. Definiremos posteriormente la semántica de esas operaciones.

El desreferenciado implícito ocurre cada vez que un identificador de variable es utilizado en el lugar de una expresión, es decir, cada vez que anunciamos que lo que precisamos es el valor representado por la variable en cuestión. El mismo es descrito simplemente como sigue:

$$\frac{}{\sigma;\ r\ \rightarrow^{e\_\sigma}\ \sigma;\ \sigma(r)} \quad \text{E-RefDeref}$$

Naturalmente, para completar la definición del mecanismo de desreferenciado implícito, realizado cada vez que queremos el valor de una variable, falta aún describir todos los lugares en donde puede ocurrir, lo cual es realizado por los contextos de evaluación.

Notar que, si cada vez que una referencia simpvalref aparece en el lugar de una expresión, se la desreferencia, entonces tales referencias nunca son el resultado de la reducción de una expresión. Es decir, estas referencias no están en L(simplevalue).

6.4.3 Expresiones que interactúan con el almacenamiento de objetos

Describimos la semántica de las expresiones cuyo comportamiento depende de la información que tenemos en el almacenamiento de objetos. Introducimos para ello la relación $\rightarrow^{e\_\theta}$, que relaciona frases de la forma "θ ; *exp*".

Definición de funciones

Si tenemos una definición de función como próxima expresión activa a reducir, entonces, la correspondiente función debe ser almacenada. Queremos clausuras de alcance léxico, por lo que debemos tener la certeza de que la función no tiene en su cuerpo identificadores libres. Dicho de otro modo, que la función tiene capturado en su cuerpo el entorno, estático, en el que fue definida. Esto significa que debería valer que, cuando la definición de la función se vuelve la expresión activa, los identificadores libres en su cuerpo, han sido reemplazados por las correspondientes referencias[6]. También, como en Lua cada objeto es manejado a través de referencias a los mismos, queremos almacenar la clausura, y que la referencia que apunta a la misma sea el resultado de la evaluación de la definición. De aquí en adelante, toda operación sobre la función, será realizada a través de esta referencia, la cual, en nuestro modelo, es de tipo objref, y es creada de manera implícita.

Desde Lua 5.2, la creación de clausuras ha sido optimizada, como se menciona en [17], sección 8.1. Tal optimización tiene impacto en la semántica de operaciones que manipulan clausuras, por lo que la incluimos en nuestro modelo. Para modelar la forma en la que Lua 5.2 crea nuevas clausuras, debemos identificar de manera unívoca a cada definición de función. Naturalmente, siendo lo nuestro un modelo de la semántica basado en sintaxis, el recurso que usaremos para esto es puramente sintáctico: agregamos a cada prototipo de la definición de una función, una etiqueta única. Sea f una función con etiqueta t. Cuando la definición de f es ejecutada, comparamos la etiqueta t con la etiqueta de las clausuras almacenadas. A su vez, para clausuras de funciones que tengan la t, observamos el cuerpo de las mismas, para comparar las referencias que podrían tener, a variables locales externas a las funciones (denominadas upvalues). En base a estos elementos, reconocemos 2 situaciones en donde f sería almacenada:

- Si no hay ninguna clausura almacenada con la etiqueta t.

[6] Para esto ocurra debe darse que la reducción comience sobre un término cerrado, y que la función de substitución esté correctamente definida

- Si hay una clausura almacenada con la etiqueta t, pero los upvalues presentes en su entorno (aquel capturado en el cuerpo de la misma) difieren de aquellos presentes en el entorno en el que f está definida. En nuestro modelo, determinar esto se reduce a comparar el cuerpo de f con el de cada clausura almacenada de etiqueta t. Si difieren, entonces podemos asumir que sus upvalues difiere.

Estas 2 situaciones son contempladas en la próxima regla de reducción, de nombre E-StoreClosure, que describe cuándo, una nueva clausura, es almacenada:

$$\frac{\begin{array}{c}(\forall\ l_1 \in \mathsf{dom}(\theta_1),\ \theta_1(l_1) = \textbf{function}\ \mathit{Name_1}\ (x_1, ..., x_m)\ \mathit{block_1}\ \textbf{end} \Rightarrow \mathit{Name_1} \neq \mathit{Name_2} \\ \vee \\ \exists\ l_2 \in \mathsf{dom}(\theta_1), \theta_1(l_2) = \textbf{function}\ \mathit{Name_2}\ (y_1, ..., y_n)\ \mathit{block_1}\ \textbf{end} \wedge \mathit{block_1} \neq \mathit{block_2}) \\ l_3\ \notin\ \mathsf{dom}(\theta_1) \\ \theta_2 = (l_3, \textbf{function}\ \mathit{Name_2}\ (y_1, ..., y_n)\ \mathit{block_2}\ \textbf{end}), \theta_1\end{array}}{\theta_1;\ \textbf{function}\ \mathit{Name_2}\ (\mathsf{y_1}, ..., \mathsf{y_n})\ \mathit{block_2}\ \textbf{end}\ \rightarrow^{\mathsf{e\_}\theta}\ \theta_2;\ \mathsf{l_3}}$$

Finalmente, si hay una clausura f' almacenada, con etiqueta t, cuyo entorno coincide con el de f, entonces no se almacena una nueva clausura. En su lugar, recuperamos la referencia a f', y la empleamos como resultado de la evaluación de la definición de f:

$$\frac{\exists\ l \in \mathsf{dom}(\theta),\ \theta(l) = \textbf{function}\ \mathit{Name}\ (x_1, ..., x_n)\ \mathit{block}\ \textbf{end}}{\theta;\ \textbf{function}\ \mathit{Name}\ (\mathsf{x_1}, ..., \mathsf{x_n})\ \mathit{block}\ \textbf{end}\ \rightarrow^{\mathsf{e\_}\theta}\ \theta;\ \mathsf{l}}\ \text{E-ReuseClosure}$$

Constructores de tablas

Tomamos los mismos constructores como la representación de las tablas. Hay una excepción a esto, que está relacionada con la posibilidad de tener presentes, en el constructor, a campos que no posean claves, como en el constructor { 4, 5, 6 }. La semántica de Lua 5.2 dicta que tales valores van a tener, como claves, números naturales consecutivos, comenzando desde 1. Por lo tanto, el constructor anterior generaría la tabla { [1] = 4 , [2] = 5 , [3] = 6 }. Sobre este comportamiento hay algunos casos especiales a contemplar:

- Cuando una clave numérica de un campo, presente originalmente en el constructor, cae dentro del intervalo [1 ; max. clave numérica para agregar], entonces, tal campo es eliminado de la tabla final:

  $> a = \{4, [1] = 1, 5\}$
  $> print(a[1])$
  4
  $> print(a[2])$
  5

- Si el valor **nil** está presente en el constructor, pero sin clave, es tenido en cuenta al momento de agregar las claves numéricas:

  $> a = \{4, nil, 5\}$
  $> print(a[1])$
  4
  $> print(a[2])$
  nil
  $> print(a[3])$
  5

La meta-función addFields : L(tableconstructor) → L(tableconstructor) describirá estos últimos detalles de la semántica del constructor de tabla. Su definición no será especificada. Se experimentará primero con una definición cuando se mecanice la semántica con PLT Redex.

Internamente, almacenaremos una tabla junto a su meta-tabla, para facilitar la definición la operación del mecanismo de meta-tablas de Lua. Emplearemos, como representación interna final de una tabla, a un par ordenado, cuya primer componente será la tabla en sí, y su segunda componente contendrá una referencia a su meta-tabla, si esta está definida. Por defecto, al ser construida una tabla, esta no posee meta-tabla, por lo tanto, la segunda componente del par ordenado tendrá inicialmente el valor especial **nil**.

Con todos los detalles cubiertos, estamos en condiciones de difinir cómo los valores de tipo tabla son creados y almacenados. En la siguiente regla notar que indicamos que almacenamos tablas que estén completamente evaluadas (es decir que, todas las expresiones pertenecientes en sus campos han sido completamente evaluadas). Notar también que, como las tablas son manejadas a través de referencias, el resultado de la evaluación de un constructor de tabla, será una referencia fresca, que será empleada en lugar de la tabla:

$$\frac{\begin{array}{c} l \notin \mathsf{dom}(\theta_1) \\ \mathit{tablevalue} = \mathsf{addFields}(\mathit{evaluatedtable}) \\ \theta_2 = (l,\ (\mathit{tablevalue},\ \mathbf{nil})\ ), \theta_1 \end{array}}{\theta_1;\ \mathit{evaluatedtable}\ \rightarrow^{\mathsf{e\_}\theta}\ \theta_2;\ l}\ \text{E-CreateTable}$$

Indexado de tablas

El indexado de tablas requiere, primero, obtener la referencia de tipo objref a la tabla. Esto es realizado por el des-referenciado implícito sobre la correspondiente referencia simpvalref, descrito en 6.4.2. Con la referencia objref obtenida, observamos el almacenamiento para recuperar la tabla e indexarla con la clave indicada. El indexado de una tabla, en condiciones normales, se formula simplemente como:

$$\frac{\theta(l) = (\{\ \ldots\ [\ \mathit{sv}_1\ ] =\ \mathit{sv}_2\ \ldots\ \}\ \ldots\ )}{\theta;\ l\ [\ \mathit{sv}_1\ ]\ \rightarrow^{\mathsf{e\_}\theta}\ \theta;\ \mathit{sv}_2}\ \text{E-IndexTable}$$

A continuación vamos a identificar las situaciones, relacionadas con el indexado de tabla, que disparan el mecanismo de meta-tablas. En la descripción de estos escenarios, necesitaremos de las siguientes meta-funciones:

- gettable : L(intreptable) → L(tableconstructor)

  gettable("(" *table* "," *metatable* ")") = *table*

- getkeys : L(table) → P(L(simplevalue))

  getkeys('{' '}') = { }
  getkeys('{' $[\mathit{sv}_1] = \mathit{sv}_{1'}$,...,$[\mathit{sv}_n] = \mathit{sv}_{n'}$ '}') = $\{\mathit{sv}_1, \ldots, \mathit{sv}_n\}$

También necesitaremos extender la función simpvaltype, definida originalmente en 6.4.1, para hacer posible el diferenciar también entre el tipo de los objetos. Definimos una nueva función type con este fin:

type : L(simplevalue) x L(θ) → {"nil", "boolean", "number", "string", "function", "table" }

type(*objref*, θ) = "table", if θ(*objref*) ∈ L(intreptable)

type(*objref*, θ) = "function", if θ(*objref*) ∈ L(functiondef)

type(*sv*, θ) = simpvaltype(*sv*), if *sv* ∉ L(Objref)

Con las herramientas definidas, procedemos a identificar las situaciones anormales, que involucran el indexado de tablas. Cuando intentamos indexar una tabla con una clave que no pertenece a la misma, se debe obtener opera entonces el mecanismo de meta-tablas: se obtiene la correspondiente meta-tabla de la tabla involucrada, y se le delega la tarea de manejar esta situación. Desde la relación $\rightarrow^{e\_\theta}$ sólo chequearemos el cumplimiento de la condición que dispara el mecanismo de meta-tablas. La relación $\rightarrow^{e\_abnormal}$ hará el resto:

$$\frac{\begin{array}{c}\mathsf{type}(l,\theta) = \text{"table"} \\ sv \notin \mathsf{getkeys}(\mathsf{gettable}(\theta(l)))\end{array}}{\theta;\ l\ [\ sv\ ]\ \rightarrow^{e\_\theta}\ \theta;\ ((l\ [sv])_{\mathbf{KeyNotFound}})}\ \text{E-AlertKeyNotFound}$$

Intentar una operación de indexado, sobre un valor que no es una tabla, constituye otra situación manejada por el mecanismo de meta-tablas:

$$\frac{\mathsf{type}(sv_1,\theta) \neq \text{"table"}}{\theta;\ sv_1\ [\ sv_2\ ]\ \rightarrow^{e\_\theta}\ \theta;\ ((sv_1\ [sv_2])_{\mathbf{NonTableIndexed}})}\ \text{E-AlertNonTableIndexed}$$

Servicios built-in

Agregamos a nuestro modelo una serie de servicios de la librería estándar de Lua, cuya semántica requiere, para ser descrita, de la información en el almacenamiento de objetos:

- rawget: el servicio rawget, indexa una tabla con una clave dada, sin invocar el mecanismo de meta-tablas, en caso de presentarse alguna situación anormal. Abstraemos este servicio en la siguiente meta-función:

  $$\mathsf{rawget} : \mathsf{L(table)} \times \mathsf{L(simplevalue)} \rightarrow \mathsf{L(simplevalue)}$$

  $$\mathsf{rawget}(table, sv_i) = \begin{cases} sv_{i+1} & \textbf{si} \quad table = \{[sv_1] = sv_2, ..., [sv_i] = sv_{i+1}, ...\} \\ \textbf{nil} & \textbf{caso contrario} \end{cases}$$

  Finalmente, para acceder al servicio descrito por la anterior meta-función, definimos la siguiente forma sintáctica:

  $$\frac{\begin{array}{c}\mathsf{type}(sv_1,\theta) = \text{"table"} \\ table = \mathsf{gettable}(\theta(sv_1)) \\ sv_3 = \mathsf{rawget}(table,\ sv_2)\end{array}}{\theta\ ;\ \textbf{\$builtIn}\ \mathsf{rawGet}\ (sv_1,\ sv_2) \rightarrow^{e\_\theta}\ \theta\ ;\ <\ sv_3\ >}\ \text{E-BuiltInRawGetNormal}$$

- getmetatable: abstraeremos, en una meta-función, el comportamiento necesario para obtener la meta-tabla de una valor dado. Luego describiremos, en términos de esta, a la semántica de la forma sintáctica correspondiente, que nos permitirá disponer, desde el lenguaje, del mecanismo de obtención de una meta-tabla.
  Como sólo las tablas tienen meta-tablas individuales (en nuestro modelo, en donde no incluimos user-data), la representación interna de las tablas es una estructura que engloba a la tabla misma junto con su meta-tabla. De esta forma, podemos extraer de manera simple la meta-tabla de una tabla dada. En el caso de valores de los restantes tipos, asumimos que sus meta-tablas, si efectivamente están definidas, deben estar almacenadas en ciertas posiciones reservadas del almacenamiento de objetos, fijadas de antemano. Por lo tanto, intentar obtener sus meta-tablas resultará en el desreferenciado de ciertas referencias objref conocidas de antemano, cuando las mismas forman parte del dominio del almacenamiento de objetos. Si este no es el caso, entonces podemos asumir que no hay meta-tablas para el tipo indicado. Dejamos las referencias concretas a utilizar para este fin, como un detalle de la mecanización de la semántica.

En cualquier caso, en Lua 5.2, una vez que se dispone de la meta-tabla de un valor dado, se intenta indexarla con la clave "__metatable", y retornar el valor asociado en el lugar de la meta-tabla (ver [17], sección 6.1; esto permite re-definir la semántica del intento de acceder a la meta-tabla de un valor, como también al intento de sobre-escribirla, como se verá más adelante, en la definición del servicio setmetatable). Todos estos mecanismos que operan para obtener una meta-tabla, los describimos con la siguiente meta-función:

- getmetatable : L(simplevalue) x L(θ) $\rightarrow$ (L(objref) $\cup$ {**nil**}):
  - si type(sv, θ) = "table", sea $\theta(sv)$ = '(' *table* ',' *metatable* ')':
    - si *metatable* $\neq$ **nil**, sea θ(*metatable*) = '(' $table_2$ ';' $metatable_2$ ')':

$$\text{getmetatable}(sv,\ \theta) = \begin{cases} metatable & \textbf{si } \text{"\_\_metatable"} \notin \text{getkeys}(table_2) \\ \text{rawget}(table_2,\ \text{"\_\_metatable"}) & \textbf{caso contrario} \end{cases}$$

    - si *metatable* = **nil**:

$$\text{getmetatable}(sv,\ \theta) = \textbf{nil}$$

  - si type(sv, θ) $\neq$ "table", sea l la correspondiente referencia a la meta-tabla del tipo de sv. Entonces:
    - si $l \in \text{dom}(\theta)$, sea $\theta(l)$ = '(' *table* ',' *metatable* ')':

$$\text{getmetatable}(sv,\ \theta) = \begin{cases} l & \textbf{si } \text{"\_\_metatable"} \notin \text{getkeys}(table) \\ \text{rawget}(table,\ \text{"\_\_metatable"}) & \textbf{caso contrario} \end{cases}$$

    - si $l \notin \text{dom}(\theta)$ (entonces, el tipo no tiene una meta-tabla definida):

$$\text{getmetatable}(sv,\ \theta) = \textbf{nil}$$

Finalmente, incorporamos este servicio al lenguaje:

$$\frac{sv_2 = \text{getmetatable}(sv_1,\ \theta)}{\theta\ ;\ \textbf{\$builtIn}\ \text{getMetatable}\ (sv_1) \rightarrow^{\text{e\_}\theta}\ \theta\ ;\ <\ sv_2\ >}\ \text{E-BuiltInGetMetaTable}$$

- type: agregamos al lenguaje el servicio que nos permite observar el tipo de una valor dado, y lo definimos en términos de la meta-función type, definida en 6.4.3:

$$\frac{sv_2 = \text{type}(sv_1, \theta)}{\theta\ ;\ \textbf{\$builtIn}\ \text{type}\ (sv_1) \rightarrow^{\text{e\_}\theta}\ \theta\ ;\ <\ sv_2\ >}\ \text{E-BuiltInType}$$

- next: la librería estándar de Lua ofrece una serie de mecanismos para iterar sobre los campos de una tabla. Uno de ellos, sobre el cual otros mecanismos están definidos, es el procedimiento next. Por ofrecer un servicio básico, que podemos describir cómodamente mediante una meta-función, lo agregamos a nuestro lenguaje. La correspondiente función, que describe el criterio para recorrer los campos de una tabla, es la siguiente:

  next : L(tableconstructor)xL(simplevalue) $\rightarrow$ L(tuple)

$$\mathsf{next}(\{\ ...,([\mathsf{sv}_n] = \mathsf{sv}_{n+1}),\ ([\mathsf{sv}_{n+2}] = \mathsf{sv}_{n+3}),...\},\ \mathsf{sv}_n\ ) = <\ \mathsf{sv}_{n+2},\ \mathsf{sv}_{n+3}\ >$$

$$\mathsf{next}(\{\ ([\mathsf{sv}_1] = \mathsf{sv}_2),...,([\mathsf{sv}_n] = \mathsf{sv}_{n+1})\ \},\ \mathsf{sv}_n\ ) = <\ \mathbf{nil}\ >$$

$$\mathsf{next}(\{\ ([\mathsf{sv}_1] = \mathsf{sv}_2),...\ \},\ \mathbf{nil}\ ) = <\ \mathsf{sv}_1,\ \mathsf{sv}_2\ >$$

$$\mathsf{next}(\{\ \},\ \mathbf{nil}\ ) = <\ \mathbf{nil}\ >$$

Y la forma sintáctica para acceder a este servicio, desde un programa, es:

$$\frac{\begin{array}{c}\mathsf{type}(sv_1, \theta) = \text{"table"} \\ tableconstructor = \mathsf{gettable}(\theta(sv_1)) \\ sv_2 \in \mathsf{getkeys}(tableconstructor)\ \vee\ sv_2 = \mathbf{nil} \\ tuple = \mathsf{next}(tableconstructor, sv_2)\end{array}}{\theta\ ;\ \$\mathbf{builtIn}\ \mathsf{next}\ (sv_1, sv_2) \rightarrow^{e\_\theta}\ \theta\ ;\ tuple}\ \text{E-BuiltInNext}$$

$$\frac{\begin{array}{c}\mathsf{type}(sv_1, \theta) = \text{"table"} \\ tableconstructor = \mathsf{gettable}(\theta(sv_1)) \\ sv_2 \notin \mathsf{getkeys}(tableconstructor)\ \wedge\ sv_2 \neq \mathbf{nil}\end{array}}{\theta\ ;\ \$\mathbf{builtIn}\ \mathsf{next}\ (sv_1, sv_2) \rightarrow^{e\_\theta}\ \theta\ ;\ \$\mathbf{builtIn}\ \mathsf{error}\ (\text{"invalid key to 'next'"})}$$

Si bien, dado un servicio de la librería estándar de Lua, nos hemos propuesto agregar a nuestro lenguaje los aspectos elementales del mismo, dejando aspectos, como el chequeo de cantidad y tipo de argumentos, para ser implementados luego, en procedimientos definidos en el entorno, lo descrito por la segunda regla, de nombre E-BuiltInNextError, no lo podemos enunciar solamente con código en nuestro lenguaje. De allí que agreguemos esa regla.
El orden en el que los índices son enumerados, no forma parte de la especificación de Lua 5.2, por lo que tenemos la libertad de retornarlos en el orden que nos resulte más conveniente. Nosotros mantenemos el orden en el que los campos son mencionados en el constructor (recordando qué ocurre con los campos del constructor que no poseen clave; ver 6.4.3). En el caso del comportamiento de la implementación original del procedimiento next, este depende de la forma en la que internamente son almacenadas las tablas (dividiendo estas en una porción que se almacenará como un arreglo, para campos con índices numéricos consecutivos, y una porción con campos cuyas claves son pasadas por una función de hash, cuando estas no son índices numéricos consecutivos; ver [16]).

- rawlen: este procedimiento permite obtener la longitud de una cadena de caracteres, o una tabla, sin la intervención del mecanismo de meta-tablas (ver en la sección 6.4.1, qué ocurre cuando se usa el operador # sobre un valor diferente de una cadena de caracteres). Aquí lo definiremos en términos de la, ya introducida, función de interpretación de operadores primitivos δ:

$$\frac{\begin{array}{c}\mathsf{type}(sv_1, \theta) = \text{"table"} \\ \mathsf{gettable}(\theta(sv_1)) = evaluatedtable \\ sv_2 = \delta(\#, evaluatedtable)\end{array}}{\theta\ ;\ \$\mathbf{builtIn}\ \mathsf{rawLen}\ (sv_1) \rightarrow^{e\_\theta}\ \theta\ ;\ <\ sv_2\ >}\ \text{E-BuiltInRawLenTable}$$

$$\frac{\begin{array}{c}\mathsf{type}(sv_1, \theta) = \text{"string"} \\ sv_2 = \delta(\#, sv_1)\end{array}}{\theta\ ;\ \$\mathbf{builtIn}\ \mathsf{rawLen}\ (sv_1) \rightarrow^{e\_\theta}\ \theta\ ;\ <\ sv_2\ >}\ \text{E-BuiltInRawLenString}$$

- rawset: este servicio nos permite asignar un campo de una tabla, sin invocar al mecanismo de meta-tablas cuando ocurre alguna situación excepcional (como intentar asignar un campo con clave inexistente). El servicio provisto por rawset podemos describirlo de manera simple, con la siguiente función (en donde $ef \in$ L(evaluatedfield)):

  assignTableField : L(evaluatedtable)xL(simplevalue)xL(simplevalue) $\rightarrow$ L(evaluatedtable)

  assignTableField({ ef_1,..., [sv_j] '=' sv_{j+1},... }, sv_j, sv_k) = { ef_1,..., [sv_j] '=' sv_k,... }
  assignTableField({ ef_1,..., ef_n }, sv_j, sv_k) = { [sv_j] '=' sv_k, ef_1,...,ef_n },
  si $sv_j \notin$ getkeys({ ef_1,..., ef_n }

  Con la meta-función definida, podemos describir cómo acceder a este servicio, desde nuestro lenguaje:

$$\frac{\begin{array}{c}\theta_1(sv_i) = (evaluatedtable_1\ ','\ metatable) \\ sv_j \neq \textbf{nil} \\ evaluatedtable_2 = \text{assignTableField}(evaluatedtable_1,\ sv_j,\ sv_k) \\ \theta_2 = \theta_1[sv_i := (evaluatedtable_2\ ','\ metatable)]\end{array}}{\theta_1\ ;\ \textbf{\$builtIn}\ \text{rawSet}\ (sv_i,\ sv_j,\ sv_k) \rightarrow^{\text{e\_}\theta}\ \theta_2\ ;\ <sv_i>}\ \text{E-BuiltInRawSet}$$

- setmetatable: este servicio nos permite definir una meta-tabla, para una tabla dada. De entre los detalles a definir de su semántica, se encuentran los siguientes:
  - Si el valor que representa la meta-tabla a asignar, es **nil**, entonces debemos remover la meta-tabla original de la tabla.
  - Si la tabla originalmente tiene una meta-tabla definida, con un campo de clave __metatable, entonces la operación debe terminar en error, ya que en tal caso, se interpreta que la meta-tabla está protegida contra escritura (y, como se mostró en el servicio getmetatable, es el valor asociado a __metatable, el que se emplea en lugar de la meta-tabla, cuando se intenta acceder a la misma).

  Nos será de utilidad, tanto para definir la semántica de setmetatable, como para describir el mecanismo de meta-tablas en general (en la sección 6.4.5), el contar con la siguiente función:

  - indexmetatable : L(simplevalue) x L(simplevalue) x L(θ) $\rightarrow$ L(simplevalue):

$$\text{indexmetatable}(sv, key, \theta) = \begin{cases} \text{rawget}(table,\ key) & \textbf{si} & (key \neq \text{"\_\_metatable"}) \wedge (\text{getmetatable}(sv, \theta) = sv_2) \wedge sv_2 \in dom(\theta) \wedge (\theta(sv_2) = \text{'(' } table \text{ ';' ... ')'}) \\ sv_2 & \textbf{si} & (key = \text{"\_\_metatable"}) \wedge (\text{getmetatable}(sv, \theta) = sv_2) \\ \textbf{nil} & \textbf{caso contrario} & \end{cases}$$

La intención de la anterior definición, es la de contar con una manera breve de referirnos al proceso de intentar indexar la meta-tabla de una valor dado, con una cierta clave. Recordar que, cuando la meta-tabla de un valor tiene definida un campo de clave __metatable, entonces getmetatable nos devuelve el valor asociado a ese campo.

Con los recursos descritos, podemos describir cómo acceder a este servicio, desde nuestro lenguaje:

$$\frac{\begin{array}{c}\text{type}(sv_1, \theta_1) = \text{"table"} \\ \text{type}(sv_2, \theta_1) = \text{"table"} \ \vee sv_2 = \textbf{nil} \\ \text{indexmetatable}(sv_1, \text{"\_\_metatable"}, \theta_1) = \textbf{nil} \\ \theta_1(sv_1) = (\mathit{evaluatedtable}, \mathit{metatable}) \\ \theta_2 = \theta_1[sv_1 := (\mathit{evaluatedtable}, sv_2)]\end{array}}{\theta_1 \ ; \ \textbf{\$builtIn}\ \text{setMetatable}\ (sv_1, \ sv_2) \rightarrow^{\text{e\_}\theta} \ \theta_2 \ ; \ < sv_1 >} \ \text{E-BuiltInSetMetatable}$$

Las situaciones erróneas a contemplar, incluyen el intentar asignar una meta-tabla, para un valor con meta-tabla protegida o intentar asignar como meta-tabla, una valor diferente a una tabla o a **nil**. Ambas situaciones las contemplamos en el siguiente caso, de nombre E-BuiltInSetMetatableErrorNoMetaTable:

$$\frac{\begin{array}{c}\text{type}(sv_1, \theta_1) = \text{"table"} \\ \wedge \\ (\ [\text{type}(sv_2, \theta_1) = \text{"table"} \ \vee sv_2 = \textbf{nil} \\ \text{indexmetatable}(sv_1, \text{"\_\_metatable"}, \theta_1) \neq \textbf{nil} \\ \mathit{message} = \text{"cannot change a protected metatable"}] \\ \vee \\ [\text{type}(sv_2, \theta) \neq \text{"table"} \ \wedge \ sv_2 \ \neq \ \textbf{nil} \\ \mathit{message} = \text{"bad argument \#2 to 'setmetatable' (nil or table expected)"}]\ )\end{array}}{\theta \ ; \ \textbf{\$builtIn}\ \text{setMetatable}\ (sv_1, \ sv_2) \rightarrow^{\text{e\_}\theta} \ \theta \ ; \ \textbf{\$builtIn}\ \text{error}\ (\mathit{message})}$$

6.4.4 Expresiones que interactúan con ambos almacenamientos

Finalmente, tratamos la semántica de las expresiones cuyo comportamiento se describe en términos de la información presente en ambos almacenamientos. Introducimos, para esto, la relación $\rightarrow^{\text{e\_}\sigma\text{\_}\theta}$, que relaciona frases de la forma $\sigma \ ; \ \theta \ ; \mathit{exp}$.

Llamada a función

Debido a que los objetos los debemos manejar a través del uso de las referencias a los mismos, en el caso de una llamada a función, requerimos del almacenamiento de objetos, para buscar la función involucrada, y del almacenamiento de valores simples, para almacenar los argumentos que se le pasan a la función. Relacionado a los argumentos de una función, otro aspecto que tenemos que modelar es el mecanismo de pasaje de parámetros por valor. A través de una semántica basada en sintaxis, imponemos ese comportamiento definiendo la semántica de una llamada a función solamente sobre listas de argumentos totalmente reducidos. Finalmente, una llamada a función modifica el entorno, agregando nuevos mapeos entre identificadores y referencias (sus parámetros) con un alcance definido (el cuerpo de la función). Usamos entonces nuestro mecanismo de substitución para modelar esa operación sobre el entorno.

En base a lo expuesto, la semántica de una llamada a función, en circunstancias normales, se define cómo:

$$\frac{\begin{array}{c}\theta\ (l) = \textbf{function}\ \mathit{Name}\ (x_1\ , ..., \ x_n\ )\ \mathit{block}\ \textbf{end} \\ r_1 \notin \text{dom}(\sigma) \ \wedge \ ... \ \wedge \ r_n \notin \text{dom}(\sigma) \\ \sigma' = (r_1, sv_1), ..., (r_n, sv_n), \sigma\end{array}}{\sigma \ ; \ \theta \ ; \ \text{l}\ (sv_1 \ , \ ... \ , \ sv_n\ ) \ \rightarrow^{\text{e\_}\sigma\text{\_}\theta} \ \sigma' \ ; \ \theta \ ; \ \mathit{block}\ [x_1 \setminus r_1 \ , \ ... \ , \ x_n \setminus r_n]} \ \text{E-Apply}$$

Cuando intentamos llamar a una función, con más argumentos que aquellos especificados en el prototipo de la misma (una función con una cantidad fija de parámetros), evaluamos igualmente todos los parámetros, y luego descartamos los restantes

$$\frac{\begin{array}{c}\theta\ (l) = \textbf{function}\ \mathit{Name}\ (x_1, ..., x_n)\ \mathit{block}\ \textbf{end} \\ r_1 \notin \mathsf{dom}(\sigma)\ \wedge\ ...\ \wedge\ r_n \notin \mathsf{dom}(\sigma) \\ \sigma' = (r_1, sv_1), ..., (r_n, sv_n), \sigma\end{array}}{\sigma\ ;\ \theta\ ;\ \mathsf{l}\ (sv_1, ..., sv_n, sv_{n+1}, ...)\ \rightarrow^{\mathsf{e\_}\sigma\_\theta}\ \sigma'\ ;\ \theta\ ;\ \mathit{block}\ [x_1 \backslash r_1, ..., x_n \backslash r_n]}\ \text{E-ApplyDiscardArgs}$$

Si llamamos a una función con menos argumentos que aquellos especificados en su prototipo, aquellos parámetros para los cuales no se pasaron valores, toman el valor que por defecto tiene toda variable, y que acusa la ausencia de un valor:

$$\frac{\begin{array}{c}\theta\ (l) = \textbf{function}\ \mathit{Name}\ (x_1, ..., x_m, x_{m+1}, ..., x_{m+l})\ \mathit{block}\ \textbf{end} \\ r_1 \notin \mathit{dom}(\sigma)\ \wedge\ ...\ \wedge\ r_{m+l} \notin \mathit{dom}(\sigma) \\ \sigma' = (r_1, v_1), ..., (r_m, v_m), (r_{m+1}, \textbf{nil}), ..., (r_{m+l}, \textbf{nil}), \sigma\end{array}}{\sigma\ ;\ \theta\ ;\ \mathsf{l}\ (v_1, ..., v_m)\ \rightarrow^{\mathsf{e\_}\sigma\_\theta}\ \sigma'\ ;\ \theta\ ;\ \mathit{block}\ [x_1 \backslash r_1, ..., x_{m+l} \backslash r_{m+l}]}\ \text{E-ApplyFewArgs}$$

Atención especial requiere el tratamiento de las situaciones en las que tenemos funciones que pueden recibir una cantidad variable de argumentos extras. Cuando se llaman a tales funciones, con al menos tantos argumentos como indica su prototipo, colocamos en una tupla la lista de aquellos que se corresponden con la posición de la expresión vararg, en el prototipo de la función. El marcador que indica que la función puede recibir una cantidad extra arbitraria de parámetros (es decir, "..."), cuando es empleado en el cuerpo de la función como expresión, solo está en representación de la lista de argumentos extras recibidos, no es una variable más. Por lo tanto, no necesitamos almacenar el contenido de esta expresión en el almacenamiento. Directamente reemplazamos, en el cuerpo de la función, toda ocurrencia de la expresión "..." por la tupla de valores correspondientes:

$$\frac{\begin{array}{c}\theta\ (l) = \textbf{function}\ \mathit{Name}\ (x_1, ..., x_n,\ '...')\ \mathit{block}\ \textbf{end} \\ r_1 \notin \mathsf{dom}(\sigma)\ \wedge\ ...\ \wedge\ r_n \notin \mathsf{dom}(\sigma) \\ \sigma' = (r_1, v_1), ..., (r_n, v_n), \sigma \\ \mathit{tuple} = \text{'}<\text{'} v_{n+1}, ..., v_{n+k} \text{'}>\text{'}\end{array}}{\sigma\ ;\ \theta\ ;\ l\ (v_1, ..., v_n, v_{n+1}, ..., v_{n+k})\ \rightarrow^{\mathsf{e\_}\sigma\_\theta}\ \sigma'\ ;\ \theta\ ;\ \mathit{block}[x_1 \backslash r_1, ..., x_n \backslash r_n, '...' \setminus \mathit{tuple}]}\ \text{E-ApplyVararg}$$

Cuando hacemos una llamada a una función, con menos argumentos que aquellos especificados en su prototipo, y esta función puede recibir argumentos extras (es decir, en su prototipo tiene el marcador "..."), entonces, en el cuerpo de la función, la expresión "..." está en representación de una lista vacía de argumentos, o, en nuestro modelo, una tupla vacía. La siguiente regla, de nombre E-ApplyVarargFew, describe esto:

$$\frac{\begin{array}{c}\theta\ (l) = \textbf{function}\ \mathit{Name}\ (x_1, ..., x_m, x_{m+1}, ..., x_{m+l},\ '...')\ \mathit{block}\ \textbf{end} \\ r_1 \notin \mathsf{dom}(\sigma)\ \wedge\ ...\ \wedge\ r_{m+l} \notin \mathsf{dom}(\sigma) \\ \sigma' = (r_1, sv_1), ..., (r_m, sv_m), (r_{m+1}, \textbf{nil}), ..., (r_{m+l}, \textbf{nil}), \sigma\end{array}}{\sigma\ ;\ \theta\ ;\ l\ (sv_1, ..., sv_m)\ \rightarrow^{\mathsf{e\_}\sigma\_\theta}\ \sigma'\ ;\ \theta\ ;\ \mathit{block}[x_1 \backslash r_1, ..., x_m \backslash r_m, x_{m+1} \backslash r_{m+1}, ..., x_{m+l} \backslash r_{m+l},\ '...' \backslash \textbf{empty}]}$$

Finalmente, identificamos las situaciones que involucran una llamada a función, y requieren de la intervención del mecanismo de meta-tablas. En Lua 5.2, hay una única situación de este tipo: cuando empleamos la forma sintáctica de llamada a función, pero sobre un valor que no es un función. Desde la relación $\rightarrow^{\mathsf{e\_}\sigma\_\theta}$ identificamos esta situación, para su posterior tratamiento:

$$\frac{\mathsf{type}(sv_1,\ \theta) \neq \text{"Function"}}{\sigma\ ;\ \theta\ ;\ \mathsf{sv}_1\ (\mathsf{sv}_2, ..., \mathsf{sv}_n) \rightarrow^{\mathsf{e\_}\sigma\_\theta}\ \sigma\ ;\ \theta\ ;\ (\mathsf{sv}_1\ (\mathsf{sv}_2, ..., \mathsf{sv}_n))_{\textbf{WrongFunctionCall}}}\ \text{E-AlertWrongFunctionCall}$$

Notar que la semántica de Lua dicta que todos los argumentos de la llamada son evaluados, antes de determinar si la llamada fue realizada efectivamente sobre una función.

6.4.5 Mecanismo de meta-tablas

Finalmente, tratamos la descripción de cómo opera el mecanismo de meta-tablas, en este caso en particular, sobre expresiones. Introducimos la noción de reducción $\rightarrow^{\mathsf{e\_abnormal}}$, que relaciona frases de la forma "θ ; *exp_abnormal*" con frases como "θ ; *exp*". Es decir, relaciona expresiones etiquetadas, que representan situaciones anormales, con expresiones que representan el intento del mecanismo de meta-tablas de darle significado, a las situaciones especiales en cuestión.

La descripción del mecanismo de meta-tablas dependerá de una serie de procedimientos que observan el almacenamiento de objetos, en busca de la correspondiente meta-tabla y manejador de evento (valor presente en una meta-tabla, asociado a una clave especial con la que se indexa la meta-tabla, en ocasión de una situación anormal). Tales procedimientos los abstraeremos aquí, como meta-funciones:

- getbinhandler : $\mathsf{L(exp)} \times \mathsf{L(exp)} \times \mathsf{L(String)} \times \mathsf{L}(\theta) \rightarrow \mathsf{L(simplevalue)}$, meta-función que describe cómo se determina un manejador para una situación anormal, que involucre a un operador binario[7]:

$$\mathsf{getbinhandler}(sv_1, sv_2, event, \theta) = \begin{cases} \mathsf{indexmetatable}(sv_1, event, \theta) & \textbf{if} \quad \mathsf{p}(sv_1, \theta, event) \\ \mathsf{indexmetatable}(sv_2, event, \theta) & \textbf{si} \quad \textbf{no } \mathsf{p}(sv_1, \theta, event) \wedge \mathsf{p}(sv_2, \theta, event) \\ \textbf{nil} & \textbf{c.c.} \end{cases}$$

  donde $\mathsf{p}(sv, \theta, event)$ denota al predicado:

  $$\mathsf{indexmetatable}(sv, event, \theta) \neq \textbf{nil} \wedge \mathsf{indexmetatable}(sv, event, \theta) \neq \textbf{false}$$

  El cual enuncia que que el evento "*event*", tiene un manejador definido.

- getunaryhandler : $\mathsf{L(exp)} \times \mathsf{L(String)} \times \mathsf{L}(\theta) \rightarrow \mathsf{L(simplevalue)}$, meta-función que describe cómo determinar un manejador, para una situación anormal que involucre a un operador unario:

$$\mathsf{getunaryhandler}(sv, event, \theta) = \begin{cases} \mathsf{indexmetatable}(sv, event, \theta) & \textbf{si} \quad \mathsf{p}(sv, \theta, event) \\ \textbf{nil} & \textbf{c.c.} \end{cases}$$

- getequalhandler : $\mathsf{L(exp)} \times \mathsf{L(exp)} \times \mathsf{L}(\theta) \rightarrow \mathsf{L(simplevalue)}$, meta-función que describe cómo determinar a un manejador, para una situación anormal que involucre a una comparación de igualdad:

[7] Basado en el procedimiento del mismo nombre, de www.lua.org/manual/5.2/manual.html#2.4

$$\text{getequalhandler}(sv_1, sv_2, \theta) = \begin{cases} sv_3 \quad \textbf{si} & \text{usemetamethod}(sv_1,\ sv_2,\ \theta) \\ & \qquad \wedge \\ & sv_3 = \text{indexmetatable}(sv_1,\ \text{"\_\_eq"},\ \theta) \\ & \qquad \wedge \\ & sv_4 = \text{indexmetatable}(sv_2,\ \text{"\_\_eq"},\ \theta) \\ & \qquad \wedge \\ & sv_3 = sv_4 \\ \textbf{nil} \quad \textbf{if} & \text{usemetamethod}(sv_1,\ sv_2,\ \theta) \\ & \qquad \wedge \\ & sv_3 = \text{indexmetatable}(sv_1,\ \text{"\_\_eq"},\ \theta) \\ & \qquad \wedge \\ & sv_4 = \text{indexmetatable}(sv_2,\ \text{"\_\_eq"},\ \theta) \\ & \qquad \wedge \\ & sv_3 \neq sv_4 \\ \textbf{nil} \quad \textbf{si} & \textbf{no}\ \text{usemetamethod}(sv_1,\ sv_2,\ \theta) \end{cases}$$

Donde $\text{usemetamethod}(sv_1,\ sv_2,\ \theta)$ denota al predicado:

$$\text{type}(sv_1,\theta) = \text{type}(sv_2,\theta) \wedge \text{type}(sv_1,\theta) = \text{"table"}$$

Situaciones anormales con operadores primitivos

En general, el tratamiento de estas situaciones se reduce a determinar si tenemos un manejador definido, para delegarle la tarea de resolver situación, o, en caso contrario, terminar con un error. Las meta-funciones definidas previamente servirán como manera abreviada de referirnos al procedimiento común consistente en buscar las meta-tablas de los valores involucrados en la operación, e intentar indexar las mismas con determinadas claves, en busca de los manejadores. Describiremos los casos representativos de la forma en la que el mecanismo de meta-tablas opera, acompañando cada caso con una breve descripción.

En el caso de los operadores aritméticos binarios, el procedimiento general es ejemplificado por el siguiente caso, de una suma sobre operandos de tipo incorrecto. La primer regla se denomina E-AdditionWrongOperandsWithHandler, y la segunda E-AdditionWrongOperandsNoHandler:

$$\frac{sv_3 = \text{getbinhandler}(sv_1, sv_2, \text{"\_\_add"}, \theta) \qquad sv_3 \neq \textbf{nil}}{\theta\ ;\ (sv_1\ +\ sv_2)_{\textbf{AdditionWrongOperands}}\ \rightarrow^{\text{e\_abnormal}}\ \theta\ ;\ \ sv_3\ (sv_1,\ sv_2)}$$

$$\frac{\textbf{nil} = \text{getbinhandler}(sv_1, sv_2, \text{"\_\_add"}, \theta) \qquad string = \text{"attempt to perform arithmetic on operands of the wrong type"}}{\theta\ ;\ (sv_1\ +\ sv_2)_{\textbf{AdditionWrongOperands}}\ \rightarrow^{\text{e\_abnormal}}\ \theta\ ;\ \textbf{\$builtIn}\ \text{error}(string)}$$

Con los restantes operadores binarios, se procede del mismo modo.

El mecanismo de meta-tablas opera de forma análoga para resolver situaciones especiales que involucren a la negación matemática. La única diferencia radica en el uso del procedimiento descrito por getunaryhandler, para obtener el correspondiente manejador. De las siguientes, la primer regla se denomina E-NegationWrongOperandWithHandler, y la segunda E-NegationWrongOperandNoHandler:

$$\frac{sv_2 = \text{getunaryhandler}(sv_1,\ \text{"\_\_unm"},\ \theta) \qquad sv_2 \neq \textbf{nil}}{\theta\ ;\ (-sv_1)_{\textbf{NegationWrongOperand}}\ \rightarrow^{\text{e\_abnormal}}\ \theta\ ;\ sv_2\ (sv_1)}$$

$$\frac{\begin{array}{c}\textbf{nil} = \text{getunaryhandler}(sv, \text{"\_\_unm"}, \theta) \\ string = \text{"attempt to perform arithmetic on a"} \mathbin{..} type(sv, \theta) \mathbin{..} \text{"value"}\end{array}}{\theta \; ; \; (-sv)_{\textbf{NegationWrongOperand}} \rightarrow^{\text{e\_abnormal}} \theta \; ; \; \textbf{\$builtIn}\; \text{error}(string)}$$

El manejo de situaciones especiales, que involucren al operador de concatenación de cadenas, no presenta sorpresas. Nuevamente, dividimos entre las situaciones en las que tenemos un manejador, y cuando no es el caso.

De las siguientes, la primer regla se denomina E-StringConcatWrongOperandsWithHandler, y la segunda E-StringConcatWrongOperandsNoHandler:

$$\frac{\begin{array}{c}sv_3 = \text{getbinhandler}(sv_1, sv_2, \text{"\_\_concat"}, \theta) \\ sv_3 \neq \textbf{nil}\end{array}}{\theta \; ; \; (sv_1 \mathbin{..} sv_2)_{\textbf{StringConcatWrongOperands}} \rightarrow^{\text{e\_abnormal}} \theta \; ; \; sv_3 \; (sv_1, \; sv_2)}$$

$$\frac{\begin{array}{c}\textbf{nil} = \text{getbinhandler}(sv_1, sv_2, \text{"\_\_concat"}, \theta) \\ string = \text{"attempt to apply string concatenation over operands of the wrong type"}\end{array}}{\theta \; ; \; (sv_1 \mathbin{..} sv_2)_{\textbf{StringConcatWrongOperands}} \rightarrow^{\text{e\_abnormal}} \theta \; ; \; \textbf{\$builtIn}\; \text{error}(string)}$$

El manejo de situaciones anormales que involucren al operador de longitud de cadena y tablas, introduce algunas novedades. Si un manejador para la situación excepcional es encontrado, entonces se le delega la tarea de resolver el problema:

$$\frac{\begin{array}{c}sv_2 = \text{getunaryhandler}(sv_1, \text{"\_\_len"}, \theta) \\ sv_2 \neq \textbf{nil}\end{array}}{\theta \; ; \; (\# \; sv_1)_{\textbf{StringLengthWrongOperand}} \rightarrow^{\text{e\_abnormal}} \theta \; ; \; sv_2 \; (sv_1)} \; \text{E-StringLengthWrongOperandWithHandler}$$

Pero si un manejador no es encontrado, y el valor, al cual se le aplicó originalmente el operador de longitud, es una tabla, entonces la operación es re-interpretada como el cálculo de la "longitud" de la tabla (recordar de 6.4.1, que la simple aplicación del operador # a un valor diferente de una cadena, dispara el mecanismo de meta-tablas, lo que comienza con el etiquetado de la expresión correspondiente, la cual estamos tratando ahora). La siguiente regla, de nombre E-StringLengthWrongOperandTableLength, expresa esto:

$$\frac{\begin{array}{c}\textbf{nil} = \text{getunaryhandler}(sv, \text{"\_\_len"}, \theta) \\ \text{type}(sv, \theta) = \text{"table"} \\ table = \text{gettable}(\theta(sv))\end{array}}{\theta \; ; \; (\# \; sv)_{\textbf{StringLengthWrongOperand}} \rightarrow^{\text{e\_abnormal}} \theta \; ; \; \delta \; (\#, table)}$$

Finalmente, si el operando involucrado no es una tabla, y no disponemos de una manejador para la situación, entonces la aplicación del operador # termina en error. La siguiente regla, de nombre E-StringLengthWrongOperandNoHandler, expresa esto:

$$\frac{\begin{array}{c}\textbf{nil} = \text{getunaryhandler}(sv, \text{"\_\_len"}, \theta) \\ \text{type}(sv, \theta) \neq \text{"table"} \\ string = \text{"attempt to get length of a"} \mathbin{..} \text{type}(sv, \theta) \mathbin{..} \text{" value"}\end{array}}{\theta \; ; \; (\# \; sv)_{\textbf{StringLengthWrongOperand}} \rightarrow^{\text{e\_abnormal}} \theta \; ; \; \textbf{\$builtIn}\; \text{error}(string)}$$

Para el tratamiento de las situaciones especiales que involucran a la comparación de igualdad, haremos uso del procedimiento descrito por la meta-función getequalhandler, el cual define el criterio empleado en Lua 5.2 para escoger un manejador para la circunstancia anormal. Está basado en el procedimiento homónimo, descrito en [17], sección 2.4:

$$\frac{\begin{array}{c}\mathsf{getequalhandler}(sv_1,\ sv_2,\ \theta) = sv_3 \\ sv_3 \neq \textbf{nil}\end{array}}{\theta\ ;\ (sv_1\ ==\ sv_2)_{\textbf{EqualityFail}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta\ ;\ sv_3\ (sv_1,\ sv_2)}\ \text{E-EqualityFailWithHandler}$$

$$\frac{\mathsf{getequalhandler}(sv_1,\ sv_2,\ \theta) = \textbf{nil}}{\theta\ ;\ (sv_1\ ==\ sv_2)_{\textbf{EqualityFail}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta\ ;\ \textbf{false}}\ \text{E-EqualityFailNoHandler}$$

El tratamiento de las situaciones anormales que involucran al operador de relación $<$, es análogo a los casos anteriores:

$$\frac{\begin{array}{c}sv_3 = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_lt"},\ \theta) \\ sv_3 \neq \textbf{nil}\end{array}}{\theta\ ;\ (sv_1\ <\ sv_2)_{\textbf{LessThanFail}}\ \rightarrow^{e\_abnormal}\ \theta\ ;\ sv_3\ (sv_1,\ sv_2)}\ \text{E-LessThanFailWithHandler}$$

$$\frac{\begin{array}{c}\textbf{nil} = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_lt"},\ \theta) \\ string = \text{"attempt to compare "} \mathbin{..} \mathsf{type}(sv_1,\ \theta) \mathbin{..} \text{" with "} \mathbin{..} \mathsf{type}(sv_2,\ \theta)\end{array}}{\theta\ ;\ (sv_1\ <\ sv_2)_{\textbf{LessThanFail}}\ \rightarrow^{e\_abnormal}\ \theta\ ;\ \textbf{\$builtIn}\ \mathsf{error}(string)}\ \text{E-LessThanFailNoHandler}$$

Finalmente, el tratamiento de las situaciones que involucran al operador $<=$ resulta más interesante. De manera usual, se comienza intentando ubicar a un manejador para la situación:

$$\frac{\begin{array}{c}sv_3 = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_le"},\ \theta) \\ sv_3 \neq \textbf{nil}\end{array}}{\theta\ ;\ (sv_1\ <=\ sv_2)_{\textbf{LessThanOrEqualFail}}\ \rightarrow^{e\_abnormal}\ \theta\ ;\ sv_3\ (sv_1,\ sv_2)}\ \text{E-LessThanOrEqualFailWithHandler}$$

Pero si se falla en la búsqueda de una manejador para $<=$, se intenta con uno que esté definido para el operador $<$, y se re-interpreta la comparación errónea original, como "**not** $<$" (naturalmente, intercambiando de lugar los valores comparados). La siguiente regla, de nombre E-LessThanOrEqualFailWithAltHandler, expresa esto:

$$\frac{\begin{array}{c}\textbf{nil} = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_le"},\ \theta) \\ sv_3 = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_lt"},\ \theta) \\ sv_3 \neq \textbf{nil}\end{array}}{\theta\ ;\ (sv_1\ <=\ sv_2)_{\textbf{LessThanOrEqualFail}}\ \rightarrow^{e\_abnormal}\ \theta\ ;\ \textbf{not}\ sv_3\ (sv_2,\ sv_1)}$$

Recién cuando, inclusive, el recurso anterior falla, la comparación termina en error. Esto expresa la siguiente regla, de nombre E-LessThanOrEqualFailNoHandler:

$$\frac{\begin{array}{c}\textbf{nil} = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_le"},\ \theta) \\ \textbf{nil} = \mathsf{getbinhandler}(sv_1, sv_2, \text{"\_\_lt"},\ \theta) \\ string = \text{"attempt to compare "} \mathbin{..} \mathsf{type}(sv_1,\ \theta) \mathbin{..} \text{" with "} \mathbin{..} \mathsf{type}(sv_2,\ \theta)\end{array}}{\theta\ ;\ (sv_1\ <=\ sv_2)_{\textbf{LessThanOrEqualFail}}\ \rightarrow^{e\_abnormal}\ \theta\ ;\ \textbf{\$builtIn}\ \mathsf{error}\ (string)}$$

Situaciones en las que una llamada a función dispara el mecanismo de meta-tablas

Recordemos, de la sección 6.4.4, que el mecanismo de meta-tablas de Lua puede ser empleado para re-interpretar operaciones de llamadas a función sobre valores que no sean funciones. En una tal llamada, si el valor que aparece en el lugar de la referencia a la función tiene una meta-tabla con campo de clave "__call", entonces, la llamada a función es repetida, ahora sobre el valor asociado a esa clave. Esto expresa la siguiente regla, de nombre E-WrongFunctionCallWithHandler:

$$\frac{\begin{array}{c}\mathsf{indexmetatable}(sv_1,\ \text{"\_\_call"},\ \theta) = sv_{n+1} \\ sv_{n+1} \neq \textbf{nil} \wedge sv_{n+1} \neq \textbf{false}\end{array}}{\theta;\ (sv_1\ (sv_2, \ldots, sv_n))_{\textbf{WrongFunctionCall}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ sv_{n+1}(sv_1, sv_2, \ldots, sv_n)}$$

Si no hay un manejador definido, entonces la operación de llamada a función termina en error, como se describe a continuación. Esto expresa la siguiente regla, de nombre E-WrongFunctionCallNoHandler:

$$\frac{\begin{array}{c}\mathsf{indexmetatable}(sv_1,\ "\_\_call",\ \theta) = sv_{n+1} \\ sv_{n+1} = \mathbf{nil} \vee sv_{n+1} = \mathbf{false} \\ string = "\mathsf{attempt\ to\ call\ a\ "} \mathbin{..} \mathsf{type}(sv_1,\ \theta) \mathbin{..} "\ value"\end{array}}{\theta;\ (sv_1\ (sv_2, ..., sv_n))_{\mathbf{WrongFunctionCall}} \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ \mathbf{\$builtIn}\ \mathsf{error}(string)}$$

Indexado de tabla erróneo

Cuando realizamos una operación de indexado, empleando una clave inexistente, el mecanismo de meta-tablas toma control de esta situación. Y, si la tabla involucrada tiene una meta-tabla con un campo de clave "__index" y con una función como valor, entonces, tal función es empleada para resolver la situación, del modo que se indica a continuación:

$$\frac{\begin{array}{c}sv_2 = \mathsf{indexmetatable}(l,\ "\_\_index",\ \theta) \\ \mathsf{type}(sv_2, \theta) = "\mathsf{function}"\end{array}}{\theta;\ (l\ [sv_1])_{\mathbf{KeyNotFound}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ sv_2\ (l,\ sv_1)}\ \text{E-KeyNotFoundWithHandlerNormal}$$

Si, en la meta-tabla de la tabla involucrada en la operación de indexado errónea, hay un campo de clave "__index", pero con un valor asociado que no es una función, entonces se repite la operación de indexado, ahora sobre este valor:

$$\frac{\begin{array}{c}sv_2 = \mathsf{indexmetatable}(l,\ "\_\_index",\ \theta) \\ sv_2 \neq \mathbf{nil} \wedge \mathsf{type}(sv_2, \theta) \neq "\mathsf{function}"\end{array}}{\theta;\ (l\ [sv_1])_{\mathbf{KeyNotFound}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ sv_2\ [sv_1]}\ \text{E-KeyNotFoundWithHandlerRepeat}$$

Finalmente, si la meta-tabla no tiene un campo de clave "__index", o la tabla involucrada en el indexado erróneo no tiene meta-tabla definida, entonces la operación de indexado resulta en el valor **nil**:

$$\frac{\mathsf{indexmetatable}(l,\ "\_\_index",\ \theta) = \mathbf{nil}}{\theta;\ (l\ [sv_1])_{\mathbf{KeyNotFound}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ \mathbf{nil}}\ \text{E-KeyNotFoundNoHandler}$$

Notar que esto explica por qué, una operación de indexado de tabla, nunca resulta en error.

Indexado sobre un valor que no es una tabla

Las situaciones contempladas son las mismas que se consideraron cuando se trató el indexado de tablas con claves inexistentes. Si la tabla involucrada tiene meta-tablas definidas:

$$\frac{\begin{array}{c}\mathsf{indexmetatable}(sv_1,\ "\_\_index",\ \theta) = sv_3 \\ \mathsf{type}(sv_3, \theta) = "\mathsf{function}"\end{array}}{\theta;\ (sv_1\ [sv_2])_{\mathbf{NonTableIndexed}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ sv_3\ (sv_1,\ sv_2)}\ \text{E-NonTableIndexedWithHandlerNormal}$$

$$\frac{\begin{array}{c}\mathsf{indexmetatable}(sv_1,\ "\_\_index",\ \theta) = sv_3 \\ \mathsf{type}(sv_3, \theta) \neq "\mathsf{function}" \wedge sv_3 \neq \mathbf{nil}\end{array}}{\theta;\ (sv_1\ [sv_2])_{\mathbf{NonTableIndexed}}\ \rightarrow^{\mathsf{e\_abnormal}}\ \theta;\ sv_3\ [sv_2]}\ \text{E-NonTableIndexedWithHandlerRepeat}$$

Finalmente, cuando no hay meta-tabla definida, o esta no tiene la información necesaria para manejar la situación, entonces, la operación de indexado termina en error:

$$\frac{\begin{array}{c}\mathsf{indexmetatable}(sv_1,\ \text{"\_\_index"},\ \theta) = \textbf{nil} \\ string = \text{"attempt to index a "} \mathbin{..} \mathsf{type}(sv_1,\ \theta) \mathbin{..} \text{" value"}\end{array}}{\theta;\ (sv_1\ [sv_2])_{\textbf{NonTableIndexed}} \rightarrow^{\text{e\_abnormal}} \theta;\ \textbf{\$builtIn}\ \mathsf{error}\ (string)}\ \text{E-NonTableIndexedNoHandler}$$

6.5 Sentencias

Describimos ahora la semántica de sentencias. El orden de la presentación será el mismo que el empleado para describir la semántica de expresiones: comenzaremos dando la semántica de sentencias que no interactúan con los almacenamientos, para ir hacia sentencias que operan con los mismos.
Modelaremos cómo opera el mecanismo de meta-tablas, de la misma manera que se empleó al tratar con expresiones: primero reconociendo, con ciertas nociones de reducción, las situaciones erróneas que son manejadas por este mecanismo, para luego describir, en una noción de reducción separada, cómo el mecanismo de meta-tablas opera para intentar re-interpretar el significado de las sentencias en cuestión.

6.5.1 Sentencias que no interactúan con almacenamientos

De entre las sentencias cuya semántica no requiere de la información presente en los almacenamientos, destacan aquellas que operan descartando la continuación actual, o, en términos de semántica de reducciones, operan descartando al contexto de evaluación que las rodea. Es decir que, para describir el comportamiento de estas sentencias, tenemos que mirar al programa completo (salvo la información del almacenamiento). Requerimos de describir su semántica en una noción de reducción separada, que denotaremos con $\rightarrow^{\text{s\_c}}$, para poder expresar luego, al momento de definir la relación de reducción estándar, que no se debe construir la clausura compatible sobre $\rightarrow^{\text{s\_c}}$, con respecto a nuestros contextos de evaluación, ya que esto arrojaría una relación estándar de reducción que no estaría describiendo una semántica determinista. Para ver esto, consideremos la descripción de la forma en la que un mensaje de error descarta la totalidad de lo que resta por computar:

$$\frac{}{E[\![\ \textbf{error}\ sv\ ]\!] \rightarrow^{\text{s\_c}} \textbf{error}\ sv}\ \text{E-Error}$$

La intención es que E denote la totalidad del resto del programa en cuestión (salvo almacenamientos), de forma tal de que ocurra que, siempre que durante la reducción de un programa se genera el mensaje de error **error** sv, se concluya la reducción de la totalidad del programa, con ese mismo objeto de error. Si, a su vez, definiéramos la relación de reducción estándar en términos de la clausura compatible de $\rightarrow^{\text{s\_c}}$, entonces, para un programa, $E[\![\ \textbf{error}\ sv]\!]$, para cada forma de particionar E en $E'[\![E'']\!]$, tendríamos el redex $E''[\![\ \textbf{error}\ sv]\!]$ sobre el cual aplicar la regla E-Error para reducirlo. De esta forma, sobre la relación de reducción estandar así definida, no se cumpliría la propiedad de unicidad de descomposición de todo programa en redex y contexto de evaluación (cuando tal programa no representa un resultado o un mensaje de error).

6.5.2 Sentencias que operan sobre el contexto actual

Describimos aquí la semántica de estas sentencias. Introducimos la relación
$\rightarrow^{\text{s\_c}} \subseteq \mathsf{L(block)xL(block)}$

Sentencia Break

La instrucción "**break** *Name tuple*" altera el flujo de ejecución usual, descartando el bloque etiquetado más inmediato (que rodea a la sentencia), que posea la etiqueta *Name*, y entregando fuera del mismo a la tupla *tuple*. Esta tupla puede ser mantenida o descartada, de acuerdo al contexto que

la rodea. Definiremos ese comportamiento sensible al contexto, empleando los contextos de $\mathsf{L}(\mathsf{E_d})$ (aquellos en donde una tupla es descartada) y de $\mathsf{L}(\mathsf{E_k})$ (aquellos en donde la tupla es mantenida). También, para poder hacer mención al bloque etiquetado más próximo a una instrucción **break**, haremos mención a contextos de $\mathsf{L}(\mathsf{E}_j)$, los cuales son todos los contextos posibles, diferentes a bloques etiquetados u otra sentencia **break**.

Las siguientes reglas, de nombres E-BreakKeepTuple, E-BreakDiscardTuple y E-BreakLabelDiscarded, respectivamente, formalizan la semántica de la instrucción **break** en condiciones normales y describen cómo interactúa la instrucción con bloques etiquetados con etiquetas distintas de la señalada en la instrucción:

$$\frac{}{E_k[\![\ \text{'::'}\ name\text{'::'}\ \{\ E_j[\![\ \textbf{break}\ name\ evaluatedtuple\ ]\!]\}[\![\ ]\!]\ ]\ \rightarrow^{\text{s-c}}\ E_k[\![\ evaluatedtuple\ ]\!]\ ]}$$

$$\frac{}{E_d[\![\ \text{'::'}\ name\ \text{'::'}\ \{\ E_j[\![\ \textbf{break}\ name\ evaluatedtuple\ ]\!]\}\ ]\!]\ \rightarrow^{\text{s-c}}\ E_d[\![\ \textbf{void}\ ]\!]}$$

$$\frac{name_1\ \neq\ name_2}{E[\![\ \text{'::'}\ name_1\ \text{'::'}\ \{E_j[\![\textbf{break}\ name_2\ evaluatedtuple]\!]\}\ ]\!]\ \rightarrow^{\text{s-c}}\ E[\![\ \textbf{break}\ name_2\ evaluatedtuple\ ]\!]}$$

Finalmente, si se concluye la reducción del cuerpo de un bloque etiquetado, sin que una instrucción **break** descarte al bloque, entonces se lo remueve:

$$\frac{}{E[\![\ label\ \{\textbf{void}\}]\!]\ \rightarrow^{\text{s-c}}\ E[\![\ \textbf{void}\ ]\!]}\qquad \text{E-BreakLabelledBlockEnd}$$

Errores

Como se mencionó en la sección 4.3, no modelaremos por completo el mecanismo de errores de Lua. En particular, los aspectos del mecanismo que quedarán fuera de nuestro modelo, tienen que ver con la posibilidad de obtener información sobre el historial de llamadas a función que llevó a la ocurrencia del error, e información del entorno, relativas al alcance de variables que aparecen en la sentencia o expresión que provocó el error.

Sí modelaremos, al menos, la capacidad de un mensaje de error de descartar la totalidad del resto de cómputo, lo cual se puede formular de manera sucinta, empleando contextos de evaluación:

$$\frac{Enp \neq [\![\ ]\!]}{Enp[\![\ \textbf{error}\ sv]\!]\ \rightarrow^{\text{s-c}}\ \textbf{error}\ sv}\qquad \text{E-Error}$$

Recordar, de la sección 5.2.2, que $\mathsf{L}(\mathsf{Enp})$ incluye a todos los contextos, salvo aquel que representa al modo protegido, $([\![\ ]\!])_{\textbf{ProtectedMode}}$. Este modo es otro aspecto del mecanismo de errores de Lua, que sí será incluido en nuestro modelo. Consiste en un recurso que el lenguaje ofrece para capturar mensajes de error. Dentro del mismo, podemos ejecutar una función y capturar cualquier error que ocurra durante su ejecución. La semántica del funcionamiento de este modo (ver [17], sección 6.1), se resume en las siguientes reglas, de nombres E-ProtectedModeErrorCatched, E-ProtectedModeNoErrorWithoutReturnedValues y E-ProtectedModeNoErrorWithReturnedValues, respectivamente:

$$\frac{}{E[\![\ (Enp[\![\ \textbf{error}\ sv]\!])_{\textbf{ProtectedMode}}]\!]\ \rightarrow^{\text{s-c}}\ E[\![\ <\textbf{false}, sv>]\!]}$$

$$\frac{}{E[\![\ (\textbf{void})_{\textbf{ProtectedMode}}]\!] \ \rightarrow^{s\text{-}c} \ E[\![< \textbf{true}, \textbf{empty} >]\!]}$$

$$\frac{}{E[\![\ (\mathit{tuple})_{\textbf{ProtectedMode}}]\!] \ \rightarrow^{s\text{-}c} \ E[\![< \textbf{true}, \mathit{tuple} >]\!]}$$

6.5.3 Sentencias que no operan con almacenamientos, ni descartan el contexto

Llamaremos a estas como "sentencias simples". Describiremos su semántica mediante la relación $\rightarrow^{s\text{-}s} \subseteq$ L(block)xL(block).

Condicional

Forzamos el orden de evaluación de un condicional, definiendo reglas de reducción que primero requiere que su guarda haya sido completamente reducida. En Lua, **nil** y **false** son evaluados como falso. Cualquier otro valor es considerado como verdadero:

$$\frac{\mathit{simplevalue} \neq \textbf{nil} \wedge \mathit{simplevalue} \neq \textbf{false}}{\textbf{if } \mathit{simplevalue} \textbf{ then } \mathit{block}_1 \textbf{ else } \mathit{block}_2 \textbf{ end } \rightarrow^{s\text{-}s} \ \mathit{block}_1} \quad \text{E-IfTrue}$$

$$\frac{\mathit{simplevalue} = \textbf{nil} \vee \mathit{simplevalue} = \textbf{false}}{\textbf{si } \mathit{simplevalue} \textbf{ then } \mathit{block}_1 \textbf{ else } \mathit{block}_2 \textbf{ end } \rightarrow^{s\text{-}s} \ \mathit{block}_2} \quad \text{E-IfFalse}$$

Concatenación de sentencias

Interpretamos a la sentencia **void** como indicación de que la primera sentencia ha sido completamente reducida:

$$\frac{}{\textbf{void}; \ \mathit{block} \ \rightarrow^{s\text{-}s} \ \mathit{block}} \quad \text{E-ConcatBehavior}$$

Bucle "While"

Tomamos de [1] la descripción del bucle **while** enunciada por la siguiente regla, de nombre E-While:

$$\frac{}{\textbf{while } \mathit{exp} \textbf{ do } \mathit{block} \textbf{ end } \rightarrow^{s\text{-}s} \textbf{ si } \mathit{exp} \textbf{ then } (\mathit{block}; \textbf{while } \mathit{exp} \textbf{ do } \mathit{block} \textbf{ end}) \textbf{ else void end}}$$

En este caso, no esperamos a la evaluación de la condición del bucle. Introducimos un condicional que evalúa la guarda del bucle, y, de cumplirse, ejecuta una vez el cuerpo del bucle, para, a continuación volver a evaluar la sentencia completa que representa el bucle. Repetimos el proceso, hasta que deje de valer la guarda del bucle.

Esta regla de reducción no entra en conflicto con la forma en la que modelamos la sentencia **break** de Lua: si tenemos un bucle en Lua como el siguiente:

```
while ... do ... break ... end
```

Entonces este es expresado en nuestro lenguaje como un bloque etiquetado, de la siguiente forma:

:: $ret :: { **while** ... **do** ... **break** $ret **empty** ... **end** }

Luego, la reduccion del bucle ocurre dentro del cuerpo del bloque etiqeutado, lo cual no interfiere con el comportamiento que hemos descrito de la sentencia **break**. También notar que estamos usando etiquetas con símbolos que están permitidos en las etiquetas de Lua. Esto nos permite disponer de un espacio de nombres privado, con identificadores que no colisionan con aquellos presentes en programas en Lua.

Bloque do...end

En relación a bloques de sentencias delimitados de manera explícita con la construcción **do - end**, sólo necesitamos describir cómo tal bloque completa su ejecución:

$$\frac{}{\textbf{do void end} \ \rightarrow^{s\_s} \ \textbf{void}} \text{ E-DoEnd}$$

Reglas para ajustar la longitud de listas de expresiones a asignar

Lua dispone de una sentencia de asignación simultanea de múltiples variables. La semántica del lenguaje dicta que, antes de que se realice la asignación, deben ser evaluadas todas las expresiones que van a a ser asignadas. Luego de ello, se aplican reglas para igualar la cantidad de elementos a ambos lados del símbolo de asignación, las cuales son simples: si hay más valores del lado derecho que variables del lado izquierdo, entonces los valores sobrantes son descartados. Si faltan valores, entonces la lista de los mismos es completada con valores **nil** (en lo que sigue, $ev \in \mathsf{L}(\mathsf{evaluatedvar})$):

$$\frac{\begin{array}{c} k >= 1 \\ sv_{n-k+1} = \textbf{nil} \ \wedge \ ... \ \wedge \ sv_n = \textbf{nil} \end{array}}{ev_1, ..., ev_n = sv_1, ..., sv_{n-k} \ \rightarrow^{s\_s} \ ev_1, ..., ev_n = sv_1, ..., sv_{n-k}, sv_{n-k+1}, ..., sv_n} \text{ E-AssignCompleteValues}$$

$$\frac{k >= 1}{ev_1, ..., ev_n = sv_1, ..., sv_n, ..., sv_{n+k} \ \rightarrow^{s\_s} \ ev_1, ..., ev_n = sv_1, ..., sv_n} \text{ E-AssignDiscardValues}$$

Cuando se concluye con este proceso, la asignación es realizada. El mecanismo que subyace a la asignación de una variable ordinaria, difiere de aquel que implementa la modificación de un campo de tabla (mientras que podemos expresar asignaciones para ambos tipos de variables, en la misma sentencia de asignación). Notar que, a raíz de que las expresiones del lado derecho son primero evaluadas, entonces, una simple alteración secuencial del almacenamiento alcanza para describir el efecto final de una asignación simultanea. Esto nos ayudará a describir (luego) la asignación de una variable ordinaria en una reducción diferente de aquella con la describiremos la asignación de un campo de tabla. Para ello, comenzaremos por separar una asignación múltiple en las correspondientes asignaciones simples, del siguiente modo::

$$\frac{n >= 2}{ev_1, ..., ev_n = sv_1, ..., sv_n \ \rightarrow^{s\_s} \ ev_n = sv_n; \ ev_1, ..., ev_{n-1} = sv_1, ..., sv_{n-1}} \text{ E-AssignSplit}$$

Notar que esto significa que las asignaciones son realizadas de derecha a izquierda.

Otra sentencia que se comporta de la misma forma que la anterior, es aquella que nos permite introducir múltiples variables locales. Por lo tanto, aparte de evaluar de izquierda a derecha todas las expresiones que queremos asignar inicialmente a las variables (comportamiento impuesto por los contextos de evaluación que definimos), aplicamos reglas como las anteriores, para igualar la cantidad de variables locales a introducir y sus valores iniciales. Lo enunciamos con las siguientes reglas, de nombres E-LocalDiscardValues y E-LocalCompleteValues, respectivamente:

$$\frac{k >= 1}{\textbf{local } x_1, ..., x_n \ = \ sv_1, ..., sv_n, ..., sv_{n+k} \textbf{ in } block \textbf{ end} \rightarrow^{s\_s} \textbf{local } x_1, ..., x_n \ = \ sv_1, ..., sv_n \textbf{ in } block \textbf{ end}}$$

$$\frac{\begin{array}{c} k >= 1 \\ sv_{n-k+1} = \textbf{nil} \ \wedge \ ... \ \wedge \ sv_n = \textbf{nil} \end{array}}{\textbf{local } x_1, ..., x_n = sv_1, ..., sv_{n-k} \textbf{ in } block \textbf{ end} \rightarrow^{s\_s} \textbf{local } x_1, ..., x_n = sv_1, ..., sv_{n-k}, sv_{n-k+1}, ..., sv_n \textbf{ in } block \textbf{ end}}$$

Habiendo resuelto este aspecto de la semántica de una asignación múltiple, estamos en condiciones de poder concentrarnos en la descripción únicamente efecto de una asignación en sí, lo que haremos en el próximo apartado.

6.5.4 Sentencias que interactúan con el almacenamiento de valores simples

Introducimos la relación $\rightarrow^{s\text{-}\sigma}$, que relaciona frases de la forma σ ; *block*.

Asignación de una variable

Cuando tenemos, en el lado izquierdo del símbolo de asignación, a un identificador de variable ordinario, el efecto de la asignación será el alterar el valor hacia el cual está mapeada, la referencia que está en correspondencia con el identificador, según el entorno. Tratados en el apartado anterior, los demás aspectos que describen la operación de asignación, nos concentramos ahora en describir la alteración final que provoca la asignación, en el mapeo entre referencia y valor:

$$\frac{\sigma' = \sigma[r := sv]}{\sigma;\ r = sv\ \rightarrow^{s\text{-}\sigma}\ \sigma';\ \textbf{void}}\ \text{E-RefMapChange}$$

Introducción de variables locales

La sentencia **local** permite introducir variables, con un valor inicial y un alcance especificados en la misma sentencia. Al introducir varias variables, hay reglas para igualar la cantidad de variables con respecto a los valores iniciales que se pretenden asignar, análogas a las de una simple asignación múltiple. Estas reglas fueron descritas en la sección 6.5.3. Por lo cual, podemos restringir la descripción de la introducción de variables locales, al caso en el que coinciden la cantidad de variables declaradas con la cantidad de valores iniciales. Para efectivamente introducir estas variables, se crean referencias frescas en el almacenamiento, una por cada variable declarada, y se las liga con los correspondientes valores asignados. Finalmente, substituimos estas referencias por las ocurrencias libres de los identificadores de variables introducidos por la sentencia, en la porción de programa que corresponda (o, equivalentemente, modificamos el entorno, agregando mapeos entre los identificadores de variables introducidos y las referencias frescas)

$$\frac{\begin{array}{c} r_1 \notin \mathsf{dom}(\sigma)\ \wedge\ \ldots\ \wedge\ r_n \notin \mathsf{dom}(\sigma) \\ \sigma' = (r_1, sv_1), \ldots, (r_n, sv_n), \sigma \end{array}}{\sigma;\ \textbf{local}\ x_1, \ldots, x_n\ =\ sv_1, \ldots, sv_n\ \textbf{in}\ block\ \textbf{end}\ \rightarrow^{s\text{-}\sigma}\ \sigma';\ block[x_1 \backslash r_1, \ldots, x_n \backslash r_n]}\ \text{E-Local}$$

De la forma propuesta, podemos garantizar que las declaraciones de variables anteriores, valgan sólo en la porción de programa indicada. De todos modos, en Lua, es posible mantener las referencias a estas variables, y usarlas fuera del alcance de la declaración de las mismas. Por ejemplo, si las capturamos en el cuerpo de una función. Como ejemplo, en el siguiente código en Lua (una versión alterada de un programa de ejemplo presente en [17]) estamos manteniendo la referencia a variables locales, de la forma mencionada:

```
a = {}
do
   local x = 20
   for i=1,10 do
       local y = 0
       a[i] = function () y=y+1; return x+y end
   end
end
```

```
print(a[1]())
print(a[1]())
print(a[2]())
print(a[2]())
```

Luego de ejecutar ese código, podemos ver que se mantiene válida la referencia a la variable x y a cada variable declarada en cada iteración del bucle:

```
21
22
21
22
```

En nuestro lenguaje, como no estamos eliminando del almacenamiento a las variables introducidas por la sentencia **local**, al concluirse la reducción del bloque que representa el alcance de la declaración de las variables, junto con nuestras clausuras de alcance léxico, se puede ver entonces que tales definiciones imitan el comportamiento de Lua observado.

6.5.5 Sentencias que interactúan con el almacenamiento de objetos

Introducimos la relación $\rightarrow^{\mathsf{s\_}\theta}$, para darle semántica a las sentencias que interactúan con el almacenamiento de objetos.

Asignación de campos de tablas

La definición de una asignación de campo de tabla, resta sobre el hecho de que, en nuestro lenguaje, tenemos tablas mutables, en contraste con las tablas funcionales, mencionadas en **ref**. La manipulación de tablas que vamos a describir se realiza, como toda operación con tablas, a través de la referencia objref que apunta a la misma. En circunstancias normales, la asignación de un campo de tabla se define cómo:

$$\frac{\begin{array}{c} sv_3 \neq \textbf{nil} \\ \theta_1(l) = (\{\ evaluatedfield_1, ..., [\ sv_i\ ] = \ sv_{i+1}, ..., evaluatedfield_n\ \}\ ,\ metatable\ ) \\ \theta_2 = \theta_1[l := (\{\ evaluatedfield_1, ..., [\ sv_i\ ] = \ sv_j, ..., evaluatedfield_n\ \}\ ,\ metatable\ )] \end{array}}{\theta_1;\ l\ [sv_i]\ =\ sv_j\ \rightarrow^{\mathsf{s\_}\theta}\ \theta_2;\ \textbf{void}}\ \text{E-AssignTable}$$

En las pre-condiciones de la regla anterior, pedimos que el valor a asignar, y la clave empleada, deben ser ambos diferentes de **nil**. Esto es en relación a la definición de tablas en Lua: arreglos asociativos, que pueden ser indexados con cualquier valor, excepto **nil**, y que pueden contener cualquier valor, excepto **nil**. Una asignación con **nil** como el valor asignar, resulta en la eliminación del campo referido de la tabla. Estos es descrito por la siguiente regla, de nombre E-DeleteTableField:

$$\frac{\begin{array}{c} \theta_1(l) = (\{\ evaluatedfield_1, ..., evaluatedfield_{i-1},\ [\ sv_i\ ] = \ sv_{i+1}, evaluatedfield_{i+1}\ , ...,\ evaluatedfield_n\ \}\ ,\ metatable\ ) \\ \theta_2 = \theta_1[l := (\{\ evaluatedfield_1, ..., evaluatedfield_{i-1}, evaluatedfield_{i+1}, ...,\ evaluatedfield_n\ \}\ ,\ metatable\ )] \end{array}}{\theta_1;\ l\ [sv_i]\ =\ \textbf{nil}\ \rightarrow^{\mathsf{s\_}\theta}\ \theta_2;\ \textbf{void}}$$

Finalmente, un intento de realizar una asignación de campo de tabla, empleando **nil** como la clave, resulta en error:

$$\frac{\begin{array}{c} \mathsf{type}(l, \theta) = \text{"table"} \\ string = \text{"table index is nil"} \end{array}}{\theta;\ l\ [\textbf{nil}] = sv\ \rightarrow^{\mathsf{s\_}\theta}\ \theta;\ \textbf{\$builtIn}\ \mathsf{error}(string)}\ \text{E-TableAssignmentIndexNil}$$

Situaciones anormales, que disparan el mecanismo de meta-tablas

Las situaciones especiales que son manejadas por el mecanismo de meta-tablas, incluyen la operación de asignación de un campo de tabla, pero con clave inexistente:

$$\frac{\begin{array}{c}\mathsf{type}(l,\theta) = \text{"}\mathsf{Table}\text{"} \\ sv_1 \neq \textbf{nil} \\ sv_1 \notin \mathsf{getkeys}(\mathsf{gettable}(\theta(l)))\end{array}}{\theta;\ l_1\ [sv_1] = sv_2 \rightarrow^{\mathsf{s\_}\theta}\ \theta;\ (l_1\ [sv_1] = sv_2)_{\textbf{TableAssignmentWrongKey}}}\ \text{E-AlertTableAssignmentWrongKey}$$

Y la asignación de un campo de tabla, sobre un valor que no sea una tabla:

$$\frac{\mathsf{type}(sv_1,\theta) \neq \text{"}\mathsf{Table}\text{"}}{\theta;\ sv_1\ [sv_2] = sv_3 \rightarrow^{\mathsf{s\_}\theta}\ \theta;\ (sv_1\ [sv_2] = sv_3)_{\textbf{TableAssignOverNonTableVal}}}\ \text{E-AlertTableAssignOverNonTableVal}$$

6.5.6 Mecanismo de meta-tablas

Introducimos la última de las nociones de reducción, la cual completa la maquinaria necesaria para describir la semántica operacional de nuestro lenguaje. Esta relación, denotada con $\rightarrow^{\mathsf{s\_abnormal}}$, relacionará frases de la forma θ ; block_abnormal con frases de la forma θ ; block.

Tablas

El tratamiento dado a las sentencias anormales que involucran el uso de tablas, es el usual: se comienza intentando obtener un manejador para resolver la situación, y, si un tal manejador no es encontrado, alguna otra acción es realizada.

Cuando ocurre una asignación sobre un campo de tabla no existente, el mecanismo de meta-tablas intenta primero obtener un manejador para delegarle la tarea de resolver la situación. La acción que se ejecuta sobre el manejador, dependerá de su tipo:

$$\frac{\begin{array}{c}sv_3 = \mathsf{indexmetatable}(l, \text{"}\_\_\mathsf{newindex}\text{"}, \theta) \\ \mathsf{type}(sv_3,\theta) = \text{"}\mathsf{function}\text{"}\end{array}}{\theta;\ (l\ [sv_1] = sv_2)_{\textbf{TableAssignmentWrongKey}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta;\ sv_3\ (l,\ sv_1,\ sv_2)}\ \text{E-TableAssignmentWrongKeyNormal}$$

$$\frac{\begin{array}{c}sv_3 = \mathsf{indexmetatable}(l, \text{"}\_\_\mathsf{newindex}\text{"}, \theta) \\ sv_3 \neq \textbf{nil}\ \wedge\ \mathsf{type}(sv_3,\theta) \neq \text{"}\mathsf{function}\text{"}\end{array}}{\theta;\ (l\ [sv_1] = sv_2)_{\textbf{TableAssignmentWrongKey}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta;\ sv_3\ [sv_1] = sv_2}\ \text{E-TableAssignmentWrongKeyRepeat}$$

Si, en cambio, no se encuentra manejador alguno, entonces se hace uso del servicio rawset. La siguiente regla, de nombre E-TableAssignmentWrongKeyNoHandler), describe esta operación:

$$\frac{\begin{array}{c}\mathsf{indexmetatable}(l, \text{"}\_\_\mathsf{newindex}\text{"}, \theta_1) = \textbf{nil} \\ \theta_1(l_1) = (\{evaluatedfield_1, ...,\ evaluatedfield_n\}\ ,\ metatable\ ) \\ \theta_2 = \theta_1\ [\ l_1 := (\{[\ sv_1\ ] =\ sv_2, evaluatedfield_1, ..., evaluatedfield_n\}\ ,\ metatable\ )\ ]\end{array}}{\theta_1;\ (l_1\ [sv_1] = sv_2)_{\textbf{TableAssignmentWrongKey}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta_2;\ \textbf{void}}$$

$$\frac{\mathsf{indexmetatable}(l, \text{"}\_\_\mathsf{newindex}\text{"}, \theta_1) = \textbf{nil}}{\theta;\ (l_1\ [sv_1] = sv_2)_{\textbf{TableAssignmentWrongKey}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta;\ \textbf{\$builtIn}\ \mathsf{rawset}\ (l,\ sv_1,\ sv_2)}$$

Finalmente, una operación de asignación de campo de tabla, sobre un valor que no sea una tabla, es manejado por el mecanismo de meta-tablas, de la manera usual:

$$\frac{sv_4 = \mathsf{indexmetatable}(sv_1, \text{"}\_\_\mathsf{newindex}\text{"}, \theta) \quad \mathsf{type}(sv_4, \theta) = \text{"}\mathsf{function}\text{"}}{\theta;\ (sv_1\ [sv_2] = sv_3)_{\mathbf{TableAssignOverNonTableVal}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta;\ sv_4\ (sv_1,\ sv_2,\ sv_3)}\ \text{E-TableAssignOverNonTableValNormal}$$

$$\frac{sv_4 = \mathsf{indexmetatable}(sv_1, \text{"}\_\_\mathsf{newindex}\text{"}, \theta) \quad sv_4 \neq \mathbf{nil} \ \wedge\ \mathsf{type}(sv_4, \theta) \neq \text{"}\mathsf{function}\text{"}}{\theta;\ (sv_1\ [sv_2]\ =\ sv_3)_{\mathbf{TableAssignOverNonTableVal}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta;\ sv_4\ [sv_2] = sv_3}\ \text{E-TableAssignOverNonTableValRepeat}$$

Si no se encuentra un manejador para esta situación, entonces la operación termina en error, como lo indica la siguiente regla, de nombrea E-TableAssignOverNonTableValNoHandler:

$$\frac{\mathbf{nil} = \mathsf{indexmetatable}(sv_1, \text{"}\_\_\mathsf{newindex}\text{"}, \theta) \quad string = \text{"}\mathsf{attempt\ to\ index\ a\ "}\ ..\ \mathsf{type}(\ sv_1,\ \theta)\ ..\ \text{"}value\text{"}}{\theta;\ (sv_1\ [sv_2] = sv_3)_{\mathbf{TableAssignOverNonTableVal}} \rightarrow^{\mathsf{s\_abnormal}}\ \theta;\ \mathbf{\$builtIn}\ \mathsf{error}\ (string)}$$

6.6 Relación de reducción estándar

Definimos aquí la relación de reducción estándar: aquella que describirá la semántica de programas completos, y que podremos emplear para determinar, dado un programa, qué resultado arroja. Estará esta relación definida como la clausura compatible sobre todas las nociones de reducción anteriores, excepto, como se indicó, de la relación $\rightarrow^{\mathsf{s\_c}}$, e incorporará ambos tipos de almacenamientos. La denotaremos con $\mapsto$:

$$\frac{e\ \rightarrow^{e}\ e'}{\sigma\ ;\ \theta\ ;\ \mathsf{E[e]}\ \mapsto\ \sigma\ ;\ \theta\ ;\ \mathsf{E[e']}}\ \text{E-simpleExpressions}$$

$$\frac{\sigma\ ;\ e\ \rightarrow^{e\_\sigma}\ \sigma'\ ;\ e'}{\sigma\ ;\ \theta\ ;\ \mathsf{E[e]}\ \mapsto\ \sigma'\ ;\ \theta\ ;\ \mathsf{E[e']}}\ \text{E-simpValStoreExpressions}$$

$$\frac{\theta\ ;\ e\ \rightarrow^{e\_\theta}\ \theta'\ ;\ e'}{\sigma\ ;\ \theta\ ;\ \mathsf{E[e]}\ \mapsto\ \sigma\ ;\ \theta'\ ;\ \mathsf{E[e']}}\ \text{E-objStoreExpressions}$$

$$\frac{\sigma\ ;\ \theta\ ;\ e\ \rightarrow^{e\_\sigma\_\theta}\ \sigma'\ ;\ \theta'\ ;\ e'}{\sigma\ ;\ \theta\ ;\ \mathsf{E[e]}\ \mapsto\ \sigma'\ ;\ \theta'\ ;\ \mathsf{E[e']}}\ \text{E-simpValObjStoreExpressions}$$

$$\frac{\theta\ ;\ e\ \rightarrow^{e\_abnormal}\ \theta'\ ;\ e'}{\sigma\ ;\ \theta\ ;\ \mathsf{E[e]}\ \mapsto\ \sigma\ ;\ \theta'\ ;\ \mathsf{E[e']}}\ \text{E-abnormalExpressions}$$

$$\frac{s \ \rightarrow^{s\_s} \ s'}{\sigma \ ; \ \theta \ ; \ \mathsf{E[s]} \ \mapsto \ \sigma \ ; \ \theta \ ; \ \mathsf{E[s']}} \ \text{E-simpleStatements}$$

$$\frac{s \ \rightarrow^{s\_c} \ s'}{\sigma \ ; \ \theta \ ; \ \mathsf{s} \ \mapsto \ \sigma \ ; \ \theta \ ; \ \mathsf{s'}} \ \text{E-breakStatements}$$

$$\frac{\sigma \ ; \ s \ \rightarrow^{s\_\sigma} \ \sigma' \ ; \ s'}{\sigma \ ; \ \theta \ ; \ \mathsf{E[s]} \ \mapsto \ \sigma' \ ; \ \theta \ ; \ \mathsf{E[s']}} \ \text{E-simpValStoreStatements}$$

$$\frac{\theta \ ; \ s \ \rightarrow^{s\_\theta} \ \theta' \ ; \ s'}{\sigma \ ; \ \theta \ ; \ \mathsf{E[s]} \ \mapsto \ \sigma \ ; \ \theta' \ ; \ \mathsf{E[s']}} \ \text{E-ObjStoreStatements}$$

$$\frac{\theta \ ; \ s \ \rightarrow^{s\_abnormal} \ \theta' \ ; \ s'}{\sigma \ ; \ \theta \ ; \ \mathsf{E[s]} \ \mapsto \ \sigma \ ; \ \theta' \ ; \ \mathsf{E[s']}} \ \text{E-abnormalStatements}$$

7

Compilación de un programa en Lua

Describiremos cómo traducir un programa en Lua, a un programa en nuestro lenguaje, completando así la descripción de la semántica de Lua.

Hay una gran proximidad entre Lua y nuestro lenguaje, con respecto a la sintaxis y (esperamos) la semántica. Por lo cual, el proceso de compilación se reduce a una traducción código fuente a código fuente, directa. En un solo paso, podemos traducir directamente a nuestro lenguaje los azúcares sintácticos, también aquellas construcciones de Lua que consideramos como abstracciones lingüísticas y podemos identificar también el alcance de las variables, para traducirlas como corresponda. Para esto, haremos uso de un esquema simple de traducción dirigida por sintaxis. Hay algunas pocas construcciones de Lua que requieren de algún trabajo para ser traducidas, incluyendo la sentencia que introduce variables locales (ya que, en nuestro lenguaje, la sentencia equivalente incluye el alcance de la declaración, a diferencia de la sentencia de Lua), las sentencias **return** y **break** (las cuales son modeladas con nuestra sentencia **break**), y la definición de funciones (las cuales, en nuestro lenguaje, incluyen en su prototipo etiquetas que las identifican de manera única).

La intención de este capítulo es la de presentar una descripción breve de la correspondencia entre las construcciones de Lua y las de nuestro lenguaje, la cual es, de todos modos, bastante directa. De todos modos, a pesar de ser bastante sencilla la correspondencia entre ambos lenguajes, la implementación de la compilación depende de herramientas cuya presentación acomplejarían innecesariamente la presentación, por lo que preferimos plantear la relación entre ambos lenguajes, en términos más informales, pero directos. Cabe mencionar que la implementación del proceso de compilación no está totalmente basado en código de nuestra autoría.

7.1 Entornos

En el proceso de traducción, necesitaremos mantener una representación de el alcance de cada declaración de variable, o, las variables presentes en el entorno, en cada sección de un programa. Emplearemos la idea consistente en mantener, para cada nueva declaración de variables, una tabla de símbolos simplificada, que contenga un registro de las variables declaradas y una referencia a una tabla de símbolos que se refiera al entorno externo al actual (es decir, tablas de símbolos encadenadas [3]). Emplearemos luego esa información para distinguir las ocurrencias de identificadores de variables de alcance local, de las ocurrencias de identificadores de variables globales, las cuales, en Lua, son consideradas como campos de la table _ENV, y, por lo tanto, requieren un tratamiento diferente al ser traducidas a nuestro lenguaje. También la información en las tablas de símbolos será útil para determinar nuevos identificadores de variables auxiliares, que serán empleados en la descripción de aquellas construcciones de Lua que las interpretamos como abstracciones lingüísticas.

Para describir las tablas de símbolos encadenadas, definiremos al conjunto de tablas de símbolos encadenadas S como:

$$S = \{ \text{empty} \} \cup \{ (s,i) : s \in S, i \in P(L(\text{Name})) \}$$

donde empty representará a una tabla de símbolos vacía, y, en el par ordenado (s,i), i será un conjunto de identificadores, representando a las variables introducidas por una construcción con

ocurrencias ligadoras de variables. En el proceso de traducción, en cada paso, mantendremos una referencia s a un miembro de S, que contendrá un registro de todos los identificadores de variables cuyo alcance incluya a la actual posición del programa Lua que estamos traduciendo. Para cada nueva construcción del lenguaje, que introduzca nuevas variables, definiremos una nueva tabla s' = (s , i), en donde s es la anterior tabla, e i contendrá los identificadores recién introducidos. Al mantener referencia a tablas externas, podemos recuperar información sobre entornos externos.

Sobre la estructura de datos anterior, nos será útil definir el predicado isLocal, el cual determina si un identificador dado se refiere a una variable local, ya sea presente en el alcance más interno de una declaración, o en uno externo:

$$\text{isLocal}(\mathit{Name};\ \text{empty}) = \text{false}$$

$$\text{isLocal}(\mathit{Name};\ (s,i)) = \begin{cases} \text{true} & \textbf{if} \quad \mathit{Name} \in i \\ \text{isLocal}(\mathit{Name};s) & \textbf{otherwise} \end{cases}$$

También necesitaremos definir un modo de agregar nuevos identificadores a nuestras tablas de símbolos. Esto será realizado por la siguiente función:

$$\text{addIdentifiers} : \mathsf{P}(\mathsf{L}(\text{namelist})) \times \mathsf{S} \to \mathsf{S}$$

$$\text{addIdentifiers}(\{\mathit{Name}_1,...,\mathit{Name}_n\},\ s\ ) = (s,\ \{\mathit{Name}_1,...,\mathit{Name}_n\})$$

7.2 Traducción código fuente a código fuente

La traducción será descrita mediante una función $[\![\_]\!] : \mathsf{L}(\mathsf{G_L}) \times \mathsf{S} \to \mathsf{L}(\mathsf{G_C})$, donde $\mathsf{G_L}$ denota a la gramática de Lua, y $\mathsf{G_C}$ denota a la gramática de nuestro lenguaje. Para capturar esta función haremos uso de ecuaciones dirigidas por sintaxis, sobre frases en Lua.

Mantendremos las mismas convenciones de notación empleadas en capítulos previos, con el agregado de que las frases en Lua estarán en fuente serif, y aquellas en el lenguaje núcleo estarán en sans serif.

Traducción de expresiones

La función transExp : $\mathsf{L}(\text{exp}_\mathsf{L}) \times \mathsf{S} \to \mathsf{L}(\text{exp}_\mathsf{C})$ traducirá expresiones en Lua, a las equivalentes expresiones en nuestro lenguaje:

- Comenzamos describiendo los casos de traducción de expresiones que ni involucran ningún desarrollo:

  transExp("..." ; s) = "..."
  transExp(**nil** ; s) = **nil**
  transExp(**false** ; s) = **false**
  transExp(**true** ; s) = **true**
  transExp(*number* ; s) = representación en el lenguaje núcleo
  transExp(*string* ; s) = representación en el lenguaje núcleo
  transExp(*prefixexp args* ; s) = transExp(*prefixexp* ; s) transArgs(*args* ; s)
  transExp(*prefixexp* ':' *Name args* ; s) = transExp(*prefixexp* ; s) ':' transExp(*Name* ; s) transArgs(*args* ; s)
  transExp('(' *exp* ')' ; s) = '(' transExp(*exp* ; s) ')'

  Donde transArgs se define como:

  transArgs('(' exp_1 ',' ... ',' exp_n ')' ; s) = '(' transExp(exp_1 ; s) ',' ... ',' transExp(exp_n ; s) ')'
  transArgs(*tableconstructor* ; s) = '(' transExp(*tableconstructor* ; s) ')'
  transArgs(*string* ; s) = '(' transExp(*string* ; s) ')'

- Definición de funciones:
  un aspecto que requiere cierto trabajo, es la representación en nuestro lenguaje, de la sentencia **return** de Lua. En nuestro lenguaje, la modelaremos empleando la sentencia **break**.

Para lograr el mismo comportamiento a través de esta sentencia, al traducir la definición de una función en Lua, colocaremos su cuerpo dentro de un bloque etiquetado, con la etiqueta específica $ret, la cual no puede coincidir con alguna otra presenten en el código Lua original, ya que los identificadores en Lua no pueden contener el símbolo $. Por lo tanto, este procedimiento puede mantenerse inclusive cuando se considere el conjunto completo de construcciones de Lua, con sentencias **goto** y bloques etiquetados.
Al traducir el cuerpo de la función, cada sentencia **return** presente será reescrita en términos de nuestra sentencia **break**, la cual expresará un salto fuera del bloque etiquetado con $ret, entregando en tal punto la lista de valores (traducidos) que la sentencia **return** original tenía, lista, la cual, la representaremos usando nuestras tuplas.
Por otro lado, como el prototipo de una función declara variables cuyo alcance es el cuerpo mismo de la función (es decir, los parámetros), debemos alterar la representación de entorno que emplearemos al traducir el cuerpo de la función. Definimos entonces s' = (s, i), donde s es la tabla de símbolos que teníamos al momento de comenzar a traducir la definición de función, e i es el conjunto de parámetros declarados en el prototipo de la función. Finalmente, en nuestro lenguaje necesitaremos definir una etiqueta única *Name*, para identificar de manera única a esta función. Dejamos el mecanismo que emplearemos para definir tal etiqueta, como un detalle propio de la implementación de la traducción. Solo se requiere que, efectivamente, cada función traducida resulte con una etiqueta única:

transExp(**function** '(' *parlist* ')' *block* **end** ; s) = **function** *Name* '(' transParlist(*parlist*) ')'
:: $ret :: {
transBlock(*block*; s')
}
end

Donde s' es la tabla de símbolos descrita anteriormente, que contiene los identificadores de *parlist*.
La función transParlist usada en la definición, traduce la lista de parámetros:

transParlist : $L(parlist_L) \to L(parlist_C)$
transParlist(*namelist*) = transNamelist(*namelist*)
transParlist('...') = '...'
transParlist(*namelist* ',' '...') = transNamelist(*namelist*) ',' transParlist('...')

La función transNamelist traduce listas de identificadores de variables:

transNamelist : $L(namelist_L) \to L(namelist_C)$
transNamelist(*name*) = representación en el lenguaje núcleo
transNamelist(*name* ',' *namelist*) = transNamelist(*name*) ',' transNamelist(*namelist*)

- **Constructores de tabla**: definimos su traducción en términos de una función auxiliar:

  transExp('{' *fieldlist* '}' ; s) = '{' transFieldlist(*fieldlist*; s) '}'

  La función transFieldlist traduce inclusive los azúcares sintácticos de la sintaxis de constructores de tablas de Lua, con la excepción de los campos sin clave, los cuales son traducidos a nuestro lenguaje sin un tratamiento especial, ya que es requerido primero evaluar de manera completa al constructor de tabla, para luego determinar qué claves tendrán estos campos.

transFieldlist : $L(fieldlist_L) \times S \to L(fieldlist_C)$
transFieldlist(*exp* ; s) = transExp(*exp* ; s)
transFieldlist('[' exp_1 ']' '=' exp_2 ; s) = '(' '[' transExp(exp_1 ; s) ']' '=' transExp(exp_2 ; s) ')'
transFieldlist(*name* '=' *exp* ; s) = '(' '[' " transNamelist(*name*) " ']' '=' transExp(*exp* ; s) ')'
transFieldlist($field_1$ $fieldsep_1$ ... $fieldsep_{n-1}$ $field_n$; s) = transFieldlist($field_1$; s)
',' ... ','
transFieldlist($field_n$; s)
transFieldlist($field_1$ $fieldsep_1$ ... $fieldsep_{n-1}$ $field_n$ $fieldsep_n$; s) = transFieldlist($field_1$; s)
',' ... ','
transFieldlist($field_n$; s)

Notar que, cuando se considera un campo como *name* '=' *exp*, la intención es traducirlo como un campo cuya clave es una cadena de carácteres, en donde estos son justamente la traducción de *name*.

- **Identificadores de variables**:
  Aquí emplearemos la información del entorno, para discriminar la ocurrencia de variables locales, de la ocurrencia de variables globales:

  $\mathsf{transVar} : \mathsf{L}(\mathsf{var}_L) \to \mathsf{L}(\mathsf{var}_C)$

  $$\mathsf{transVar}(Name\ ;\ \mathsf{s}) = \begin{cases} \mathsf{transNamelist}(Name) & \textbf{si} \quad \mathsf{isLocal}(Name\ ;\mathsf{s}) \\ \_\mathsf{ENV}[\text{"}\ \mathsf{transNamelist}(Name)\ \text{"}] & \textbf{caso contrario} \end{cases}$$

  $\mathsf{transVar}(\mathit{prefixexp}\ \text{'['}\ \mathit{exp}\ \text{']'}\ ;\ \mathsf{s}) = \mathsf{transExp}(\ \mathit{prefixexp}\ ;\ \mathsf{s})\ \text{'['}\ \mathsf{transExp}(\mathit{exp}\ ;\ \mathsf{s})\ \text{']'}$
  $\mathsf{transVar}(\mathit{prefixexp}\ .\ \mathit{Name}\ ;\ \mathsf{s}) = \mathsf{transExp}(\ \mathit{prefixexp}\ ;\ \mathsf{s})\ \text{'['}\ \text{"}\ \mathsf{transNamelist}(\mathit{Name})\ \text{"}\ \text{']'}$

  Finally, we extend the definition of transExp, for list of variables:

  $$\mathsf{transExp}(\mathit{var}\ ;\ \mathsf{s}) = \mathsf{transVarlist}(\mathit{var}\ ;\ \mathsf{s})$$

- **Operadores binarios**:
  La traducción de operadores binarios no depara sorpresas. Solo recordemos que intentamos incluir en nuestro lenguaje a un conjunto restringido de operadores:

  $\mathsf{transExp}(\mathit{exp}_1\ \text{'+'}\ \mathit{exp}_2\ ;\ \mathsf{s}) = \mathsf{transExp}(\ \mathit{exp}_1\ ;\ \mathsf{s})\ \text{'+'}\ \mathsf{transExp}(\mathit{exp}_2\ ;\ \mathsf{s})$
  $\mathsf{transExp}(\mathit{exp}_1\ \text{'}\sim\text{='}\ \mathit{exp}_2\ ;\ \mathsf{s}) = \textbf{not}\ \mathsf{transExp}(\mathit{exp}_1\ ;\ \mathsf{s})\ \text{'=='}\ \mathsf{transExp}(\mathit{exp}_2\ ;\ \mathsf{s})$
  $\mathsf{transExp}(\mathit{exp}_1\ \text{'>'}\ \mathit{exp}_2\ ;\ \mathsf{s}) = \mathsf{transExp}(\mathit{exp}_1\ ;\ \mathsf{s})\ \text{'>'}\ \mathsf{transExp}(\mathit{exp}_2\ ;\ \mathsf{s})$

  La traducción de expresiones que involucran a los restantes operadores binarios se realiza el mismo modo.
  En [17] se menciona que, en Lua, una expresión como exp_1 '>' exp_2, es traducida internamente empleando la comparación '<', y lo mismo vale para '>='. De todos modos, el orden en el que las expresiones comparadas son evaluadas, sigue siendo de izquierda a derecha, en el orden de la expresión original. Para reproducir ese comportamiento, en nuestra semántica basada en sintaxis, debemos mantener el orden de ocurrencia de las expresiones comparadas, y, por lo tanto, incluir '>' y '>=' en nuestro lenguaje.

- **Operadores unarios**
  La traducción de expresiones que involucran a los operadores unarios, no depara sorpresas. Ilustramos con un ejemplo:

  $$\mathsf{transExp}(\text{'-'}\ \mathit{exp}\ ;\ \mathsf{s}) = \text{'-'}\ \mathsf{transExp}(\mathit{exp}\ \ ;\ \mathsf{s})$$

  Los restantes operadores unarios son traducidos del mismo modo.

Finalmente, definimos una función que será útil en definiciones posteriores:

$$\mathsf{transExpList}(\mathit{exp}_1,...,\mathit{exp}_n;\ \mathsf{s}) = \mathsf{transExp}(\mathit{exp}_1;\ \mathsf{s}),...,\mathsf{transExp}(\mathit{exp}_n;\ \mathsf{s})$$

Traducción de sentencias

Definimos la función transBlock, que traducirá a nuestro lenguaje, bloques enteros de sentencias en Lua. Como se menciona en [17], un programa en Lua puede contener sentencias **return** al final de cualquier bloque de sentencias, no necesariamente dentro de una función (ya que, de hecho, un programa Lua es compilado como el cuerpo de otra función, con lo cual, empleamos la sentencia **return** para devolver valores desde una función, pero también desde un bloque de instrucciones cualquiera). Como tal, colocaremos, la traducción un bloque de instrucciones dado, en un bloque etiquetado, desde el que se puede salir, con la ejecución de una sentencia **break**:

transBlock : $L(G_L) \rightarrow L(G_C)$
transBlock(*block*) = '::' $ret '::' '{'
transBlockAux(*block*, empty)
'}'

Finalmente, la traducción que cada sentencia será delegada a la función auxiliar transBlockAux : $L(G_L) \times S \rightarrow L(G_C)$, que recibe inicialmente el bloque completo que queremos traducir, y una cadena vacía inicial de tablas de símbolos.
Damos una definición recursiva de transBlockAux, la cual presentaremos en diferentes piezas:

- Comenzamos por la traducción de sentencias que no requieren ningún tratamiento especial:

  transBlockAux(var_1 ',' ... ',' var_n '=' exp_1 ',' ... ',' exp_m ; s) = transExplist(var_1 ',' ... ',' var_n ; s)
  '='
  transExplist(exp_1 ',' ... ',' exp_m ; s)

  transBlockAux(*prefixexp* '(' exp_1 ',' ... ',' exp_n ')' ; s) = transExp(*prefixexp* '(' exp_1 ',' ... ',' exp_n ')' ; s)

  transBlockAux(**if** *exp* **then** $block_1$ **else** $block_2$ **end** ; s) = **if** transExp(*exp* ; s) **then**
  transBlockAux($block_1$; s)
  else
  transBlockAux($block_2$; s)
  end

  transBlockAux(**do** *block* **end** ; s) = **do** transBlockAux(*block* ; s) **end**

  transBlockAux(';' ; s) = **void**

- **Bucle "while"**:
  Su traducción es prácticamente directa. El único detalle que requiere de una explicación, tiene que ver con la forma en la que manejamos la posibilidad de tener, en el cuerpo del bucle en Lua, una sentencia **break**. Convertimos el proceso de saltar fuera del bucle, al proceso, expresado en nuestro lenguaje, de saltar fuera de un bloque etiquetado, que contiene completamente al bucle. La traducción propuesta es la siguiente:

  transBlockAux(**while** *exp* **do** *block* **end** ; s) = '::' $while '::' '{'
  while transExp(*exp* ; s) **do**
  transBlockAux(*block* ; s)
  end
  '}'

  Completamos el tratamiento de la traducción del bucle **while**, con la correspondiente traducción de la sentencia **break**:

  transBlockAux(**break** ; s) = **break** $while **empty**

  Bucle for numérico:
  Este tipo de bucle es tratado como una abstracción lingüística, que puede ser descrita empleando un bucle **while**. La traducción es la que se propone en [17], por lo que la omitimos aquí. Para implementar la traducción, necesitamos generar identificadores frescos de variables, lo que se hace empleando la información que tenemos en las tablas de símbolos encadenadas. De todos modos, más allá de este detalle, no se requiere ninguna nueva idea para esta traducción, que las ya expuestas hasta aquí.

  Bucle for genérico:
  Al igual que con el bucle **for** numérico, la traducción del bucle **for** genérico sigue la misma descripción de [17].

  Variables locales:
  Trataremos aquí el caso en el que tenemos una sentencia **local**, seguida de un bloque de instrucciones con una o más sentencias. El caso en el que tenemos sólo una sentencia **local**, se tratará de la misma manera, sólo que agregando, en la traducción, a la sentencia de nuestro lenguaje **void**, como el cuerpo de la sentencia:

transBlockAux(**local** *namelist* '=' *explist block* , s) = **local** transNamelist(*namelist*)
'='
transExplist(*explist*, s)
in
transBlockAux(*block*, s')
end

Donde s' = addIdentifiers(transNamelist(*namelist*), s).

Definición de funciones, como sentencia
La declaración de una función local, se traduce de manera directa, como se muestra a continuación:

transBlockAux(**local function** *Name funcbody block* , s) = **local** $Name_2$ '=' **nil**
in
$Name_2$ '=' transExp(**function** *funcbody*, s') ';'
transBlock(*block*, s')
end

Donde $Name_2$ = transNamelist(*Name*) y s' = addIdentifiers($Name_2$, s). La traducción propuesta para la definición de función, intenta generar código que resuelva de modo correcto cualquier llamada recursiva presente en el cuerpo de la función.
La restante sintaxis para definir funciones, es traducida como:

transBlockAux(**function** *funcname funcbody* , s) = transFuncname(*funcname* ; s)
'='
transExp(**function** *funcbody*, s')

La función transFuncname, traduce nombres de funciones, lo que involucra consultar la tabla de símbolos, para determinar si estamos asignando la función a una variable local o global, y tratar algunos azúcares sintácticos de nombres de funciones, que son introducidos con la intención de proveer soporte para la programación orientada a objetos (como, por ejemplo, el nombre var.Name, que debe ser traducido a la forma var["Name"].

La sentencia return
Esta sentencia es empleada en Lua para retornar valores desde una función o un bloque de sentencias. Podemos modelar su comportamiento en términos de nuestra sentencia **break**. La traducción de un bloque de instrucciones, desde el cual podemos retornar valores usando la sentencia de Lua **return**, será colocado dentro un bloque etiquetado, con una etiqueta específica. Luego, la sentencia **return** será traducida hacia una sentencia **break**, que provoca un salto, fuera del bloque etiquetado, entregando una tupla con los valores que la sentencia **return** original estaba entregando:

transBlockAux(**return** exp_1,...,exp_n ; s) = **break** $ret '<' transExp($exp_1$; s),...,transExp(exp_n) '>'
transBlockAux(**return** exp_1,...,exp_n ';' ; s) = **break** $ret '<' transExp($exp_1$; s),...,transExp(exp_n) '>'
transBlockAux(**return** ';' ; s) = **break** $ret **empty**
transBlockAux(**return** ; s) = **break** $ret **empty**

Concatenación de sentencias:
Ya habíamos considerado el caso de una concatenación de sentencias, cuando la primera de ellas es una sentencia que introduce variables locales, o una función local. Los casos restantes de bloques de instrucciones, los traducimos como sigue;:

transBlockAux(*stat block* ; s) = transBlockAux(*stat* ; s) ';' transBlockAux(*block* ; s)
transBlockAux(*stat* ';' *block* ; s) = transBlockAux(*stat* ; s) ';' transBlockAux(*block* ; s)

8

Entorno de ejecución

Tras el proceso de compilación de un programa Lua a nuestro lenguaje, terminamos con un término probablemente abierto. Los identificadores libres en ese término van a referirse a servicios, provistos por la librería estándar de Lua, o a variables globales. En 6.4.3, describimos un conjunto de servicios básicos, de la librería estándar de Lua, que decidimos incluir en nuestro modelo. Para acceder a esos servicios, definimos una forma sintáctica especial. Como se mencionó al momento de definir estos servicios, la intención es la de mantener el lenguaje lo más reducido posible, por lo que incluimos en el mismo funcionalidades básicas de la librería estándar, y dejamos cualquier otra característica (como por ejemplo, chequeo de tipo y cantidad de argumentos) como una implementación con procedimientos en nuestro lenguaje, a definirse como parte del entorno en el que se ejecuta un programa. Esto también nos permite proveer el acceso a los servicios básicos desde procedimientos almacenados en la tabla especial _ENV, que es lo que justamente Lua ofrece. Describimos a continuación tales procedimientos.

8.1 Procedimientos de envoltorio

Hay ciertos servicios de la librería estándar de Lua, que hemos agregado a nuestro lenguaje, y para los cuales sólo es requerido definir un procedimiento de envoltorio, que sea asignado a la tabla _ENV, para hacerlos disponibles a todos los programas, a través del mecanismo usual de llamadas a función, como ocurre en Lua:

- getmetatable:

```
_ENV["getmetatable"] = function $getmetatable (value)
                        :: $ret :: {
                            break $ret < $builtIn getMetatable (value) >
                        }
                       end
```

- rawequal:

```
_ENV["rawequal"] = function $rawequal (value1, value2)
                    :: $ret :: {
                        break $ret < $builtIn rawEqual (value1,value2) >
                    }
                   end
```

- select: notar el uso de la expresion "vararg":

```
_ENV["select"] = function $select (index ,...)
                  :: $ret :: {
                      break $ret < $builtIn select (index ,...) >
                  }
                 end
```

- tonumber:

```
_ENV["tonumber"] = function $tonumber (e, base)
                                :: $ret :: {
                                        break $ret < $builtIn toNumber (e, base) >
                                }
                                end
```

- type:

```
_ENV["type"] = function $type (value)
                    :: $ret :: {
                        break $ret < $builtIn type (value) >
                    }
                end
```

8.2 Servicios descritos como implementaciones en nuestro lenguaje

Hay otro repertorio de servicios que deseamos que estén disponibles para programas en nuestro lenguaje, y que podemos describir como implementaciones en nuestro lenguaje. La intención es que sean servicios sin ninguna complejidad, para que nuestra implementación no requiera de mayores justificaciones.

A continuación describimos algunos de ellos, junto a su implementación.

8.2.1 ipairs

La semántica de una llamada a ipairs(t) es descrita en [17] como:

> `If t has a metamethod __ipairs, calls it with t as argument and returns the first three results from the call. Otherwise, returns three values: an iterator function, the table t, and 0, so that the construction`
>
> `for i,v in ipairs(t) do body end`
>
> `will iterate over the pairs (1,t[1]), (2,t[2]), ..., up to the first integer key absent from the table.`

Esta descripción es suficiente para justificar el siguiente código, que implementa el servicio usando otros ya definidos en nuestro lenguaje. La siguiente función será almacenada en _ENV["ipairs"], para hacer este servicio disponible a cualquier programa:

```
function $ipairs (t)
 :: $ret :: {
      local (metatable,handler, iter ,type_t = $builtIn getMetatable (t), nil , nil , $builtIn type(
          t))
      in
         if (metatable == nil)
         then
           void
         else
           handler = $builtIn rawGet(metatable, " __ipairs ")
         end;
         if (handler == nil)
         then
            if (type_t == "table")
            then
              void
            else
```

```
            $builtIn error ("bad argument #1 to 'ipairs' (table expected, got " .. type_t
                .. ")")
        end;
        iter = function $iPairsIter (t, var)
                :: $ret :: {
                  if ($builtIn type(t) == "table")
                  then
                    void
                  else
                    $builtIn error ("expected a table value")
                  end;
                  local (result == nil)
                  in
                    var = var + 1;
                    result = $builtIn rawGet(t,var);
                    if (result == nil)
                    then
                      (break $ret < nil >)
                    else
                      (break $ret < var, result >)
                    end
                  end
                }
              end;
        (break $ret < iter,t,0 >);
      else
        local (v1,v2,v3 = (handler (t)))
        in
          (break $ret < v1,v2,v3 >)
        end
      end
    end
}
end
```

Notar, en el código anterior, que, cuando usamos los servicios de la librería estándar incluidos en nuestro lenguaje, estamos empleando directamente la forma sintáctica definida para ello, sin depender en el mecanismo usual de llamada a procedimiento, a través de un identificador conocido de antemano para invocar tales servicios. La implementación de ipairs en Lua emplea de manera directa al servicio rawget, por lo tanto, no es posible alterar el comportamiento de ipairs asignándole a la variable _ENV["rawget"] otro valor. La misma razón justifica los restantes usos de la forma sintáctica **$builtIn**.

8.2.2 pairs

Su semántica es similar a la de ipairs, con la diferencia de que pairs puede ser empleado para iterar sobre todos los campos de una tabla, y no sólo aquellos con claves numéricas. Procedemos como en la descripción del servicio ipairs, definiendo una función que implementa lo ofrecido por pairs. Tal función deberá ser asignada al campo _ENV["pairs"], para hacer disponible el servicio a todo programa:

```
function $pairs (t)
 :: $ret :: {
      local (metatable,handler, iter = ($builtIn getMetatable (t)), nil , nil )
      in
        if (metatable == nil)
        then
          void
        else
          handler = $builtIn rawGet (metatable," __pairs")
```

```
            end;
            if (handler == nil)
            then
               if ($builtIn type (t) == "table")
               then
                 void
               else
                 $builtIn error ("expected a table value")
               end;
               iter = function $pairsIter (t, var)
                          :: $ret :: {
                                (break $ret < $builtIn next (t,var) >);
                          }
                      end;
               (break $ret < iter,t,0 >);
            else
               local (v1,v2,v3 = (handler (t)))
               in
                  (break $ret < v1,v2,v3 >)
               end
            end
         end
 }
end
```

8.2.3 tostring

Si bien hemos incluido en el lenguaje una forma sintáctica para acceder a la conversión de valores a cadenas de caracteres, el servicio provisto por Lua agrega cierta información, que bien puede ser descrita como una implementación en nuestro lenguaje, que se apoya en otros servicios ya definidos. La siguiente función debe asignarse al campo _ENV["tostring"], para hacer disponible el servicio:

```
function $tostring (v)
  :: $ret :: {
        local (metatable,handler, string ,type = $builtIn getMetatable (v), nil , nil , nil )
        in
          if (metatable == nil)
          then
            void
          else
            handler = ($builtIn rawGet (metatable," __tostring "))
          end;
          if (handler == nil)
          then
             type = $builtIn type(v);
             string = $builtIn toString (v);
             if (type == "table")
             then
               string = "table: " .. string
             else
               if (type == "function")
               then
                  string = "function: " .. string
               else
                 void
               end
             end;
             break $ret < string >
          else
```

```
            break $ret < handler (v) >
        end
    end
  }
end
```

8.2.4 Servicios que

Finalmente, describimos la implementación de una serie de servicios, que se reducen a procedimientos de envoltorio para las llamadas **$builtIn**, pero agregan chequeos de tipos. Los listamos a continuación:

- next:

```
_ENV["next"] = function $next (table, index)
                :: $ret :: {
                        local type_table = $builtIn type (table) in
                            if (not (type_table == "table")) then
                                $builtIn error ("bad argument #1 to 'next' (table expected
                                    , got " .. type_table .. ")")
                            else
                                break $ret < $builtIn next (table, index) >
                            end
                        end
                }
               end
```

- rawget:

```
_ENV["rawget"] = function $rawget (table, index)
                :: $ret :: {
                        local type_table = $builtIn type (table) in
                            if (not (type_table == "table")) then
                                $builtIn error ("bad argument #1 to 'rawget' (table
                                    expected, got " .. type_table .. ")")
                            else
                                break $ret < $builtIn rawGet (table, index) >
                            end
                        end
                }
               end
```

- rawlen:

```
_ENV["rawlen"] = function $rawlen (v)
                :: $ret :: {
                        local type_v = $builtIn type (v) in
                            if (not (type_v == "table") and (not (type_v == "string"))
                                then
                                $builtIn error ("bad argument #1 to 'rawlen' (table or
                                    string expected)")
                            else
                                break $ret < $builtIn rawLen (v) >
                            end
                        end
                }
               end
```

- rawset:

```
_ENV["rawset"] = function $rawset (table, index, value)
                 :: $ret :: {
                        local type_table = $builtIn type (table) in
                            if (not (type_table == "table")) then
                                $builtIn error ("bad argument #1 to 'rawset' (table
                                    expected, got " .. type_table .. ")")
                            else
                                if (index == nil) then
                                    $builtIn error ("table index is nil")
                                else
                                    break $ret < $builtIn rawSet (table, index, value) >
                                end
                            end
                        end
                 }
                 end
```

- setmetatable:

```
_ENV["setmetatable"] = function $setmetatable (table, metatable)
                       :: $ret :: {
                              local type_table = $builtIn type (table) in
                                  if (not (type_table == "table")) then
                                      $builtIn error ("bad argument #1 to 'setmetatable' (table
                                          expected, got " .. type_table .. ")")
                                  else
                                      break $ret < $builtIn setMetatable (table, metatable) >
                                  end
                              end
                       }
                       end
```

9

Mecanización con PLT Redex

La semántica propuesta ha sido mecanizada empleando PLT Redex[1]. La herramienta nos provee de facilidades para describir todas las piezas de nuestra semántica, para explorar nuestro modelo y para realizar testeos de conformidad de la semántica con respecto a la implementación de Lua 5.2.

Comenzaremos describiendo los distintos recursos que ofrece PLT Redex para describir un lenguaje junto a su semántica de reducciones, y los mecanismos que permiten explorar la definición descrita.

9.1 Gramática

La gramática del lenguaje puede definirse mediante una notación que impone el uso de una versión de nuestra gramática completamente parentizada. En la figura 9.1 se puede observar una porción del módulo grammar, en donde está definida toda la gramática. Las líneas mostradas en la imagen, corresponden a la definición de las producciones del símbolo no terminal statement, de nuestra gramática.

La notación empleada difiere en algunos aspecto de la que usamos en este texto (EBNF). En particular, difiere en los recursos empleados para indicar la posibilidad de múltiples ocurrencias de una cierta frase. En la figura 9.2, se puede ver, en la línea numerada 112, el uso de una notación que sugiere la idea de "ellipses" matemática, para indicar que una lista de nombres (namelist) puede contener 0 o más ocurrencias de frases generadas por el símbolo no terminal Name. El lenguaje de patrones que se emplea para describir la gramática, también será usado para describir las restantes piezas de la semántica.

9.2 Meta-funciones

PLT Redex nos permite definir meta-funciones a través de ecuaciones en las que podemos emplear el mismo lenguaje de patrones utilizado para describir la gramática y las nociones de reducción. Si

[1] http://redex.racket-lang.org/

```
[statement (label \{ block \})
           (do block end)
           (err simplevalue)
           ((block)ProtectedMode)
           functioncall
           (if exp then block else block end)
           (while exp do block end)
           (local namelist = explist in block end)
           (varlist = explist)
           void]
```

Figure 9.1: Gramática

```
[varlist (var ...)]

[(evaluatedvar ev) simpvalref
              (objref \[ simplevalue \])]

[evaluatedvarlist (evaluatedvar ...)]

[tableconstructor (\{ field ... \})]

[evaluatedtable (\{ evaluatedfield ... \})]
```

Figure 9.2: Patrones para describir la cantidad de ocurrencias de una frase

nuestras meta-funciones poseen entonces una definición recursiva, sobre la estructura de las frases del lenguaje, su transcripción a PLT Redex es directa. Como ejemplo, en la figura 9.3 se puede ver el comienzo de la definición de la función de substitución para expresiones.

9.3 Nociones de reducción

La transcripción de nuestra definición a un modelo en Redex es directa. Por cada noción de reducción de nuestro modelo, definimos módulo de Redex. En la figura 9.5 se puede ver el comienzo de la definición del módulo en donde describimos la noción de reducción $\rightarrow^{s\text{-}s}$. Más allá de los detalles específicos de la notación empleada, es intuitiva la correspondencia directa entre esta definición y la que presentamos en este texto.

Con respecto a las nociones de reducción, es posible explorarlas empleando la herramienta para aplicar sucesivas veces una noción de reducción dada, sobre una frase especificada, y obtener una representación visual del historial de reducción. En la figura 9.4 se puede ver el historial de reducción de la sentencia "while true do void end", la cual configura un bucle infinito, lo que es señalado por la herramienta.

9.4 Verificación de la semántica

Naturalmente, es también posible aplicar sucesivas veces una noción de reducción dada, y obtener una representación del historial de reducciones que no sea visual, sobre la cual poder definir predicados para testear la semántica. Este recurso se empleó para verificar la semántica con respecto a la suite de tests de Lua 5.2[2]. La suite de tests está implementada como una colección de archivos Lua, en donde cada uno contiene una colección de chequeos de aserciones, relativas a algún aspecto específico del lenguaje. Por lo tanto, verificar que nuestro modelo satisface tales tests, se reduce a compilarlos a nuestro lenguaje, e intentar reducir el término resultante, observando que todos los chequeos de aserciones resulten satisfactorios. Los tests que actualmente se corren, conforman una selección restringida de entre el total de tests que componen a la suite. Los casos de tests que no son contemplados actualmente incluyen:

- Construcciones del lenguaje: verificación de sentencia **goto**, co-rutinas y user-data.
- Mecanismo de administración automática de memoria.
- Tests que verifican la calidad del código intermedio generado. Lua compila a un código de arquitectura neutra, que es el código de una máquina virtual, en términos de la cual se implementa el lenguaje. Naturalmente, esto queda fuera del modelo de Lua que pretendemos definir.
- Test que estresa el manejo de las estructuras de datos sobre las que se implementa el lenguaje: manejo de tablas con gran cantidad de campos, recursión de cola (realizando gran cantidad de llamadas recursivas). Estos tests quedan fuera por la dificultad de ejecutarlos sobre los equipos con los que contamos.

[2]Ver en http://www.lua.org/tests/, suite para Lua 5.2.2

```
#lang racket
(require redex
         "../grammar.scm")

; Substitution function over expressions
; PARAMS:
; exp : the expression to which the substitution is applied
; parameters : a list of identifiers to be subtituted
; explist : a list of expressions to substitute the identifiers in
; parameters. The identifier in the position i of parameters will
; be replaced by the expression in the position i of expressions
(define-metafunction core-lang
  ;substExp : exp parameters explist -> exp

  ; Variable identifier or vararg expression
  [(substExp parameter parameters explist)
   (applySubst parameter parameters explist)]

  ; Function call
  [(substExp (exp \( explist_1 \)) parameters explist_2)
   ((substExp exp parameters explist_2) \(
                                        (substexplist explist_1 parameters explist_2)
                                        \))]

  [(substExp (exp : Name \( explist_1 \)) parameters explist_2)
   ((substExp exp parameters explist_2) : Name \(
                                        (substexplist explist_1 parameters explist_2)
                                        \))]
  ; Protected mode
  [(substExp ((exp)ProtectedMode) parameters explist_2)
   (((substExp exp parameters explist_2))ProtectedMode)]
```

Figure 9.3: Implementación de la función de substitución

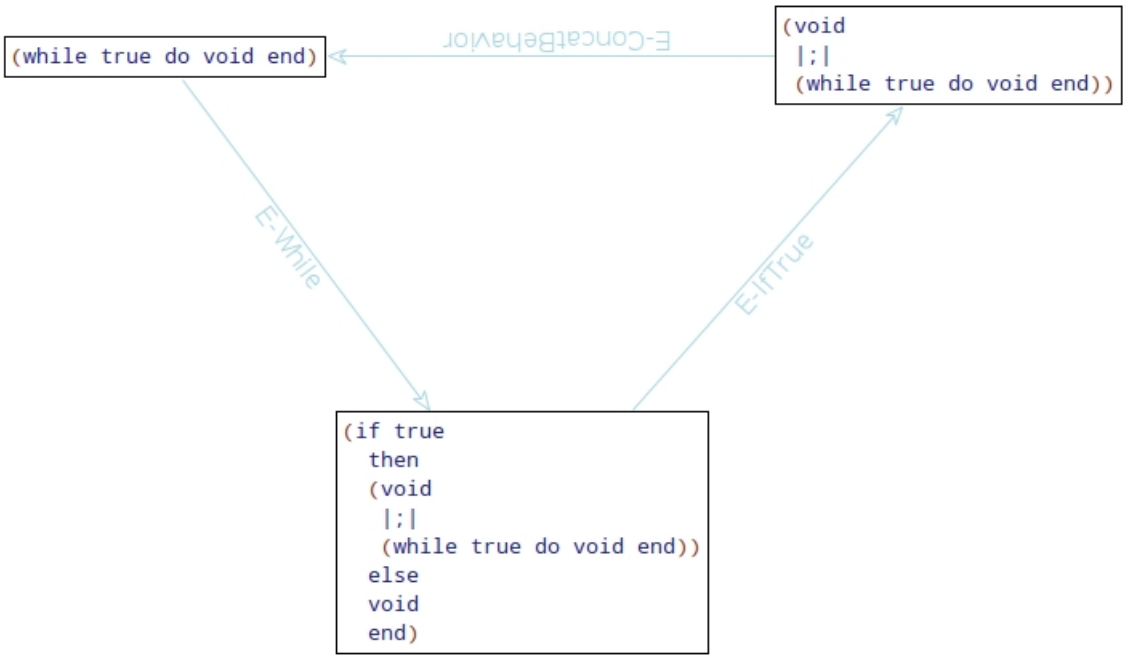

Figure 9.4: Reducción de la sentencia "while true do void end", mediante la noción de reducción $\rightarrow^{s\text{-}s}$

- Tests para servicios de la librería que no han sido incorporados a nuestro lenguaje

Aparte de confrontar nuestro modelo con la suite de tests de Lua 5.2, se implementó una suite de tests propia para nuestra mecanización, esta última, apoyándose en las herramientas que provee PLT Redex. Contamos con una colección de tests para cada noción de reducción y meta-función de nuestro modelo.

Disponibilizamos el código de la mecanización, a través de un repositorio de público acceso, cuya dirección es https://mallku@bitbucket.org/mallku/mecanizacion_lua.git. Por cualquier sugerencia o duda, escribir a mallkuernesto@gmail.com.

```
#lang racket
; Statements that don't interact with some store and whose behaviour is not
; context sensitive

(require redex
         "../grammar.scm"
         "../Meta-functions/simpValStoreMetafunctions.scm")

(define core-lang-simply-stat-red
  (reduction-relation
   core-lang
   #:domain block
   ; If statement
   [--> (if simplevalue then block_1 else block_2 end)
      block_1
      E-IfTrue
      (side-condition (and (not (eqv? (term simplevalue) (term nil)))
                           (not (eqv? (term simplevalue) (term false)))))]

   [--> (if simplevalue then block_1 else block_2 end)
      block_2
      E-IfFalse
      (side-condition (or (eqv? (term simplevalue) (term nil))
                          (eqv? (term simplevalue) (term false))))]
   ; While statement
   [--> (while exp do block end)
      (if exp then (block \; (while exp do block end)) else void end)
      E-While]
```

Figure 9.5: Definición de la noción de reducción $\rightarrow^{s\text{-}s}$

Bibliografía

[1] C. Saftoiu A. Guha and S. Krishnamurthi. The essence of javascript. In *ECOOP 2010, Lecture Notes in Computer Science 6183*, pages 126–150, 2010.

[2] C. Muñoz A. L. Galdino and M. Ayala-Rincón. Formal verification of an optimal air traffic conflict resolution and recovery algorithm. In *WoLLIC, Lecture Notes in Computer Science*, pages 177–188, 2007.

[3] Ravi Sethi Jeffrey D. Ullman Alfred V. Aho, Minica S. Lam. *"Compilers Principles, Techniques and Tools"*. Pearson Education Inc., second edition, 2007.

[4] F. Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[5] M. Ayala-Rincón C. Morra, J. Becker and R. W. Hartenstein. Felix: Using rewriting-logic for generating functionally equivalent implementations. In *IEEE FPL'05*, pages 25–30, 2005.

[6] Matthias Felleisen. *The calculi of Lambda-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages*. PhD thesis, Indiana University, 1987.

[7] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103:235–271, 1992.

[8] P. J. Landin. Correspondence between algol 60 and church's lambda-notation: part i. *Communications of the ACM*, 8:89–101, 1965.

[9] R. P. Jacobi M. Ayala-Rincón, C. H. Llanos and R. W. Hartenstein. Prototyping time- and space-efficient computations of algebraic operations over dynamically reconfigurable systems modeled by rewriting-logic. *ACM Trans. Design Autom. Electr. Syst.*, 11(2):251–281, 2006.

[10] Matthew Flatt Matthias Felleisen, Robert Bruce Finlder. *Semantics Engineering with PLT Redex*. The MIT Press, 2009.

[11] Joe Gibbs Politz, Matthew J. Carroll, Benjamin S. Lerner, Justin Pombrio, and Shriram Krishnamurthi. A tested semantics for getters, setters, and eval in javascript. In *DLS '12 Proceedings of the 8th symposium on Dynamic languages*, pages 1–16, 2012.

[12] Joe Gibbs Politz, Alejandro Martinez, Matthew Milano, Sumner Warren, Daniel Patterson, Junsong Li Beijing, Anand Chitipothu Bangalore, and Shriram Krishnamurthi. Python: the full monty. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages and applications*, pages 217–232.

[13] L. H. de Figueiredo R. Ierusalimschy and W. Celes. Lua–an extensible extension language. *Software: Practice and Experience*, 26(6):635–652, 1996.

[14] L. H. de Figueiredo R. Ierusalimschy and W. Celes. Passing a language through the eye of a needle. *ACM Queue*, 9(5):20–29, 2011.

[15] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 2009.

[16] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. "the evolution of lua". In *HOPL III Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 2–1–2–26. ACM New York, NY, USA, 2007.

[17] Waldemar Celes Roberto Ierusalimschy, Luiz Henrique de Figueiredo. *Lua 5.2 Reference Manual*, 2011-2013. Available at www.lua.org/manual/5.2/manual.html.

[18] J. M. Rushby S. Owre and N. Shankar. Pvs: A prototype verification system. In *CADE'92, Lecture Notes in Computer Science 607*, pages 748–752, 1992.

[19] J. C. Mitchell S. S. Maffeis and A. Taly. An operational semantics for javascript. *In Asian Symposium on Programming Languages and Systems*, 2008.

[20] M. H. B. Sorensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism. Studies in Logic and the Foundations of Mathematics 149.* Elsevier Science, 1998.

[21] The Coq Development Team. *The Coq Proof Assistant Reference Manual, Version V8.0.* 2004.