Semántica operacional y su aplicación para el estudio de recolección de basura, en Lua 5.2

por Mallku Ernesto Soldevila Raffa

Presentado ante la Facultad de Matemática, Astronomía, Física y Computación, como parte de los requerimientos para la obtención del grado de Doctor en Ciencias de la Computación de la

UNIVERSIDAD NACIONAL DE CORDOBA

5 de abril de 2020

Director: Dr. Fridlender, Daniel
Co-director: Dr. Ziliani, Beta

Resumen

Lua es un lenguaje de programación imperativo de scripting, que ofrece tipado dinámico, manejo automático de memoria, facilidades para la descripción de datos, y mecanismos de meta-programación para adaptar el lenguaje a dominios específicos. Es utilizado en proyectos de diversa naturaleza, desde desarrollo de juegos, de manera notable en juegos "AAA", desarrollo de *plugins*, *firewall* de aplicaciones web, y en sistemas embebidos. Gracias al éxito de Lua es posible encontrar diversas implementaciones alternativas y analizadores estáticos. Sin embargo, la naturaleza informal de la especificación del lenguaje implica que quienes desarrollan esas herramientas no pueden proveer garantías formales de corrección para las mismas.

En este trabajo presentamos una formalización de la semántica operacional de Lua 5.2, incluyendo recolección de basura (GC) y sus interfaces (*finalizadores* y *tablas débiles*; una forma de implementar referencias débiles). Validamos la semántica mediante su mecanización, utilizando PLT Redex, y el testeo de la misma con respecto a la suite de tests del intérprete oficial de Lua. A su vez, utilizamos las facilidades ofrecidas por PLT Redex para la mecanización de sistemas formales y testeo aleatorio de propiedades, para obtener evidencia de la propiedad de progreso de la semántica. Para GC proveemos un framework para razonar formalmente sobre propiedades de cualquier algoritmo de GC basado en un criterio sintáctico. Dentro del framework podemos formalizar y esbozar la demostración de propiedades sobre GC, incluyendo su corrección (sin considerar sus interfaces).

La semántica formalizada y su mecanización podrían ayudar a proveer garantías formales de corrección para herramientas que realicen análisis estático de programas Lua, como también en el prototipado de nuevos conceptos de programación y extensiones para Lua.

Abstract

Lua is a lightweight imperative scripting language, featuring dynamic typing, automatic memory management, data description facilities, and metaprogramming mechanisms to adapt the language to specific domains. It is extensively used in projects ranging from game development, most notably by "AAA" games, plugin development, web application firewalls, and embedded systems. Thanks to Lua's success, several alternative implementations and static analyzers can be found in the wild. However, the informal nature of the language's specification means that developers of those tools cannot provide formal guarantees of correctness for them.

In this work we present a formalization of Lua 5.2's operational semantics, including garbage collection (GC) and its interfaces (finalizers and weak tables; a particular implementation of weak references). We validate our model by mechanizing it, using PLT Redex, and testing it against the test suite of the reference interpreter of Lua. Also, we use the features provided by PLT Redex for the mechanization of formal systems and random testing of properties, to gather evidence for the progress property of the semantics. For GC we provide a framework for formal reasoning of properties of any GC algorithm based on a syntactic criterion. Within the given framework we are able to formalize and sketch the proof of several claims about GC, including its correctness (without its interfaces).

The formalized semantics and its mechanization could help to provide formal guarantees of correctness for tools that perform static analysis of Lua programs, as well as for the prototyping of new features and extensions to Lua.

Contenido

Figuras

Esta página ha sido intencionalmente dejada en blanco

1 Introducción

Lua* es un lenguaje de programación imperativo de *scripting*[1][1] , que ofrece tipado dinámico, manejo automático de memoria, facilidades para la descripción de datos, y mecanismos de meta-programación para adaptar el lenguaje a dominios específicos [2] . Está implementado como una librería en "clean C"[2], y ofrece una interfaz poderosa con C, C++[3] y otros lenguajes[3].

El caso de uso típico de una aplicación Lua se da en la forma de una librería embebida en una aplicación *host*, comúnmente escrita en C o C++. En ese escenario, podemos disponer de las facilidades ofrecidas por un lenguaje dinámico como Lua (flexibilidad, posibilidad de desarrollar prototipos velozmente, recolección automática de basura, etc), allí donde no sean necesarias las garantías estáticas y optimizaciones de lenguajes más estrictos como C o C++.

Lua es utilizado en proyectos de diversa naturaleza, desde desarrollo de juegos, de manera notable en juegos "AAA" [4] , juegos para celulares, frameworks de desarrollo de juegos [4], desarrollo de *plugins*[5], firewalls de aplicaciones web[6], y en sistemas embebidos[7].

1.1. Objetivos

Lua posee una especificación informal, provista por su manual de referencia[8], con una implementación oficial desarrollada y mantenida por un grupo reducido de programadores[9]: Roberto Ierusalimschy, Luiz Henrique de Figueiredo y Waldemar Celes. Sin embargo, gracias al éxito de Lua, es posible encontrar diversas implementaciones alternativas[10] y analizadores estáticos[11]. La naturaleza informal de la especificación del lenguaje implica que quienes desarrollan esas herramientas deben guiarse por un entendimiento intuitivo sobre el comportamiento del lenguaje, intuición formada por el estudio del manual de referencia, la inspección del código fuente del intérprete y la experimentación.

En este trabajo presentamos una formalización de la semántica operacional de Lua 5.2, incluyendo recolección de basura (GC) y sus interfaces (*finalizadores* y *tablas débiles*; una forma de implementar referencias débiles). Validamos la semántica mediante su mecanización, utilizando PLT Redex, y el testeo de la misma con respecto a la suite de tests del intérprete oficial de Lua. A su vez, utilizamos las facilidades ofrecidas por PLT Redex para la mecanización de sistemas de tipos y testeo aleatorio de propiedades, para obtener evidencia de la propiedad de progreso de la semántica. Para GC proveemos un framework para razonar formalmente sobre propiedades de cualquier algoritmo de GC basado en un criterio sintáctico. Dentro del framework podemos formalizar y esbozar la demostración de propiedades sobre GC, incluyendo su corrección (sin considerar sus interfaces).

[1]: R. Ierusalimschy y col. (2011), «Passing a language through the eye of a needle»

[2]: R. Ierusalimschy y col. (1996), «Lua – an extensible extension language»

[3]: Luiz Henrique de Figueiredo (1994), «The design and implementation of a language for extending applications»

1: Lenguaje de programación diseñado para ser utilizado en un entorno específico, en donde, normalmente, debe interactuar con otros lenguajes o tecnologías (p. ej., JavaScript para el *front-end* de un sitio web, o Lua como lenguaje de extensión en aplicaciones escritas en C o C++).

[4]: Ierusalimschy y col. (2001), «The evolution of an extension language: a history of Lua»

2: El conjunto intersección de conceptos entre el estándar de C y C++. Un programa escrito en "clean C" puede ser correctamente procesado por un compilador estándar de C y/o de C++.

3: Ver `lua-users.org/wiki/BindingCodeToLua`.

4: Por ejemplo, LÖVE. Ver `love2d.org`

5: Por ejemplo, en el software de edición de fotos Adobe Photoshop Lightroom: `www.adobe.com/devnet/photoshoplightroom.html`, y el sistema de type-setting LuaTex: `www.luatex.org/languages.html`.

6: `blog.cloudflare.com/cloudflares-new-waf-compiling-to-lua`

7: `www.lua.org/uses.html`

8: `www.lua.org/manual/5.2/`

9: `www.lua.org/authors.html`

10: `lua-users.org/wiki/LuaImplementations`

11: `lua-users.org/wiki/ProgramAnalysis`

*: En el encabezado de este capítulo: logo de Lua. Créditos a `www.lua.org`.

La semántica formalizada y su mecanización podrían ayudar a proveer garantías formales de corrección para herramientas que realicen análisis estático de programas Lua, como también en el prototipado de nuevos conceptos de programación y extensiones para Lua. Finalmente, disponer de una descripción formal y exhaustiva de la semántica de Lua puede ser de utilidad para guiar otras implementaciones del lenguaje, sin la necesidad de que estas deban basarse exclusivamente en el estudio de la implementación de la máquina virtual de Lua[12].

12: lua-users.org/lists/lua-l/2001-02/msg00127.html

1.2. Requisitos de formalización de parte de quienes desarrollan aplicaciones con Lua

Inclusive para personas cuyo único objetivo es el de desarrollar aplicaciones con Lua, nuestra formalización puede resultar de utilidad, ya que ayuda a aclarar ambigüedades e imprecisiones de la documentación oficial que, inclusive, son discutidas en la lista oficial de mails de Lua. Un lista no exhaustiva de estos tópicos incluye:

- *Cacheo* de clausuras[13] .
- Semántica de listas de valores retornados por funciones[14].
- Semántica de meta-métodos para llamadas a función[15] y manipulación de tablas[16].
- Interacción entre recolección de basura y finalización[17].
- Semántica de finalizadores[18].

13: lua-users.org/lists/lua-l/2014-03/msg00972.html.
14: lua-users.org/lists/lua-l/2007-09/msg00178.html.
15: lua-users.org/lists/lua-l/2017-02/msg00187.html.
16: lua-users.org/lists/lua-l/2009-07/msg00295.html.
17: lua-users.org/lists/lua-l/2010-10/msg00511.html.
18: lua-users.org/lists/lua-l/2006-01/msg00277.html.

Como ejemplo sencillo, que no requiere de introducir detalles complejos sobre la semántica de Lua, desarrollamos el primer ítem de la lista anterior, que muestra la dificultad para predecir el comportamiento de algunos programas Lua, valiéndose únicamente del manual de referencia. En Lua, las clausuras (es decir, valores de tipo *función*) son manejadas a través de referencias a las mismas. Toda manipulación sobre clausuras (llamadas a función, pasar una clausura como parámetro, retornar una clausura desde una función, asignación a variables, etc) es realizada a través de las referencias a las mismas. Más aún, esa referencia hace a la definición misma de la clausura[19] . Hasta la versión 5.1 de Lua, quienes desarrollaban programas con Lua sabían que cada ejecución de un constructor de funciones creaba una nueva clausura. A partir de la versión 5.2, se contempló la posibilidad de que un intérprete Lua agregue una optimización en la creación de clausuras. Esta optimización consiste en la reutilización de una clausura ya existente como valor resultado de la ejecución de un constructor de funciones, siempre que ambas clausuras (la ya existente y la nueva por crear) no exhiban *diferencias observables*[20] . Remarcamos el hecho de que, para el manual de referencia de Lua 5.2, un intérprete de tal versión del lenguaje es libre de implementar o no la mencionada optimización.

19: De ese modo, por ejemplo, comparación de clausuras se reduce a comparación de referencias a las mismas.

20: www.lua.org/manual/5.2/manual.html#8

El que el manual de referencia no proveyera mayores detalles sobre lo que implica una *diferencia observable* entre clausuras, forzó a que, quienes intentaran comprender las implicaciones de esta optimización, tuvieran que valerse solamente de la experimentación con el intérprete oficial para intuir el significado de la nueva semántica de creación de

clausuras, tal como se observa en los mails de la lista de discusión de Lua, referidos en la página anterior.

La nueva semántica de creación de clausuras, tal como está implementada en el intérprete oficial, es mostrada en la Figura 1.1. En la línea 9 se observan los efectos de la optimización mencionada. La línea 11 exhibe el comportamiento esperado cuando cada ejecución de un constructor de clausuras resulta en un nuevo valor de tipo función, diferente a los previamente creados. Los detalles necesarios para comprender el código serán desarrollados en la sección 4.2.3. Por el momento, simplemente observar que la semántica de creación de clausuras exhibe un comportamiento más complejo que el de la simple construcción de valores nuevos de tipo función.[21]

```
1  function fun_factory()
2    return function() end
3  end
4
5  local f = function () end
6
7  local g = function () end
8
9  print(fun_factory() ==
       fun_factory())
10 -->> outputs: true
11 print(f == g)
12 -->> outputs: false
```

Figura 1.1: Creación de clausuras.

21: Diferencia que llegó a romper retrocompatibilidad con código Lua ya existente, que funcionaba asumiendo la unicidad de cada clausura nueva creada: `lua-users.org/lists/lua-l/2011-06/msg01350.html`.

Es importante aclarar que no pretendemos criticar al manual de referencia de Lua. Dada la complejidad del comportamiento que puede exhibir un lenguaje de programación para aplicaciones reales, es inevitable que haya limitaciones en un documento que se propone como de introducción y descripción informal para la mayoría de las personas interesadas en el lenguaje.

1.3. Criterio de formalización

El lenguaje que vamos a especificar reside en la intersección entre lo que especifica su manual de referencia, lo implementado en sus principales intérpretes y lo que comprenden del mismo las personas que lo utilizan. Desarrollamos a continuación los motivos detrás de este enfoque.

Pretendemos que la formalización propuesta sirva como un adecuado primer paso en el traslado de la especificación y el entendimiento informales de Lua, hacia un terreno más formal. No pretendemos sugerir que no existen otros trabajos sobre formalización de la semántica de Lua[22] , sino que la intención es que la semántica propuesta se ubique adecuadamente como primer paso entre la especificación informal del lenguaje, y el estudio del mismo desde una óptica formal.

22: Más adelante presentaremos otros esfuerzos de formalización de la semántica de Lua.

Dados los objetivos propuestos, solo podemos aspirar a obtener evidencia empírica respecto a la correcta correspondencia entre nuestra formalización y Lua. Nos interesaran 2 tipos de evidencia. Por un lado, la posibilidad de ejecutar código Lua siguiendo nuestra especificación, corroborando que obtenemos lo mismo que con algún intérprete de Lua 5.2. Para nuestro trabajo nos concentraremos en contrastar con el intérprete oficial. Este requisito nos llevará a tomar decisiones de formalización allí en donde el manual de referencia otorgue libertad a quienes implementen un intérprete. Así, por ejemplo, impondremos un orden de evaluación para listas de expresiones (listas de argumentos en una llamada a función, listas de expresiones asignadas a variables, etc), a pesar de que el manual de referencia no especifica tal orden. También incluiremos la optimización en la creación de clausuras descrita en el apartado anterior, ya que está permitida por el manual de referencia, y el intérprete oficial la incluye.

Por otro lado, sería de utilidad que personas familiarizadas con el lenguaje puedan encontrar una conexión lo más directa posible entre Lua y nuestra formalización. Eso puede ayudar a construir confianza sobre la corrección de nuestra formalización. En [5] se observa que, para lenguajes cuya semántica ya está bien comprendida, un enfoque razonable de formalización puede ser el de identificar un conjunto mínimo de conceptos a formalizar, a partir del cual explicar la totalidad del lenguaje. Este enfoque, que podríamos llamarlo de *lenguaje núcleo*, puede rastrearse hacia atrás hasta [6, 7] , en donde se propone una correspondencia entre ALGOL 60 y λ-cálculo; correspondencia que se aprovechó para proveer una formalización de la semántica de ALGOL 60.

[5]: Maffeis y col. (2008), «An Operational Semantics for JavaScript»

[6]: Landin (1965), «Correspondence between ALGOL 60 and Church's Lambda-Notation: Part I»
[7]: Landin (1965), «A Correspondence between ALGOL 60 and Church's Lambda-Notations: Part II»

De no tratarse de un lenguaje para el cual ya se posea una adecuada comprensión de su semántica, podría ser deseable una formalización que se corresponda de manera directa con la principal especificación (informal) que posea el lenguaje, como se sugiere en [5] .

[5]: Maffeis y col. (2008), «An Operational Semantics for JavaScript»

También, en [8] [23], se reconoce que, en esfuerzos de formalización de lenguajes de programación reales, a los fines de ganar confianza respecto a la corrección de la especificación, es importante habilitar la posibilidad de que la labor sea revisada por la mayor cantidad posible de personas con intereses, y entendimientos, distintos sobre la semántica del lenguaje: personas que desarrollan aplicaciones con el lenguaje, personas que desarrollan compiladores/intérpretes para el lenguaje, etc.

[8]: Bodin y col. (2014), «A trusted mechanised JavaScript specification»

23: JSCert: una formalización de la semántica de JavaScript utilizando el asistente de pruebas Coq

El enfoque de *lenguaje núcleo* aplicado a Lua 5.2, sí se ha explorado en [9] , en donde se presenta un λ-cálculo extendido con conceptos de Lua 5.2 que se los entiende como esenciales para explicar la totalidad del lenguaje. Por los motivos expresados previamente, en este trabajo no descartaremos la utilidad de disponer de una formalización que se corresponda de manera casi directa con lo especificado en el manual de referencia. Consideramos que es más adecuado el enfoque sugerido en [5, 8] , si pretendemos ofrecer una formalización que sirva de puente con la especificación informal del lenguaje.

[9]: Lin (2015), «Operational Semantics for Featherweight Lua»

[5]: Maffeis y col. (2008), «An Operational Semantics for JavaScript»
[8]: Bodin y col. (2014), «A trusted mechanised JavaScript specification»

En la práctica, nos quedaremos con el lenguaje resultado de eliminar los azúcares sintácticos y las (pocas) abstracciones lingüísticas que están descritas en el manual de referencia. Notar también que Lua mismo está diseñado con el foco puesto en proveer de un conjunto elemental de conceptos, junto con mecanismos de meta-programación para extender el lenguaje allí donde sea necesario.

1.4. Formalismo

Lua es, en su manual de referencia y en la mente de las personas que trabajan con él, un lenguaje para describir procesos. Y esta noción de proceso es comúnmente definida en términos de estado, es decir, se trata de un proceso cuya finalidad es la de modificar información almacenada en la memoria de la computadora. Como consecuencia, el formalismo que utilizamos para especificar Lua es, principalmente,

el de una semántica operacional *small-step*. A su vez, a los fines de conseguir una descripción modular de la semántica (lo que incluye, también, separar la especificación de los cómputo, de el orden en el que estos ocurren), junto con una definición concisa de aquellas reglas de ejecución con semántica sensible al contexto, haremos uso de la noción de contexto de evaluación, tomada de *reduction semantics* (RS) en el estilo de Felleisen-Hieb [10] [24].

24: RS es introducida en el capítulo 3

En parte, seguimos el camino tomado por [11-14] , donde un formalismo semejante es utilizado para lograr la especificación de lenguajes de programación como Python y JavaScript. Nos diferenciamos de los citados trabajos en la manera en la que presentamos nuestra formalización (poniendo énfasis en distinguir las incumbencias de RS, de las de una semántica operacional tradicional), en la inclusión de características de Lua no presentes en los trabajos citados (principalmente, semántica de recolección de basura) y en la utilización de facilidades de Redex para explorar la validez de propiedades deseables del modelo.

[10]: Felleisen y col. (2009), *Semantics Engineering with PLT Redex*
[11]: Guha y col. (2010), «The Essence of JavaScript»
[12]: Politz y col. (2013), «Python: The Full Monty: A Tested Semantics for the Python Programming Language»
[13]: Politz y col. (2012), «A tested semantics for getters, setters, and eval in JavaScript»
[14]: Matthews y col. (2007), «An operational semantics for Scheme»

1.5. Cobertura y mecanización

A los fines de poder contrastar nuestra semántica con lo implementado en el intérprete oficial de Lua, mecanizamos nuestra formalización con PLT Redex[10] . De este modo, testeamos el modelo con respecto a la suite de test del intérprete oficial, logrando ejecutar exitosamente cada test referido a conceptos que caen dentro del alcance de nuestra formalización. Interpretamos esto como evidencia importante de que nuestro modelo es una correcta representación de las facilidades del lenguaje formalizadas, entre las que se encuentran:

[10]: Felleisen y col. (2009), *Semantics Engineering with PLT Redex*

- Todas las sentencias, excepto **goto**.
- Todos los tipos de Lua, excepto *co-rutinas* y *userdata*.
- Mecanismo de meta-tablas.
- Todos las funciones básicas de la librería estándar, excepto manejo de archivos. Incluimos:
  - La posibilidad de disponer del mismo intérprete en el entorno de ejecución, a través del servicio de librería load.
  - Errores y mecanismos de manejo de errores.
- Recolección de basura, junto con sus interfaces: tablas débiles y finalizadores.
- En la mecanización incluimos también servicios de la librería table y string[25] .

Se excluyeron servicios de librería que sirven de interfaz con el sistema operativo, o que tiene una implementación compleja en C, como los servicios de *pattern matching* para strings.

25: También, a los fines de poder ejecutar la suite de tests del intérprete oficial, incluimos servicios de la librería math, aunque solamente apoyándose en la implementación en Racket (el lenguaje sobre el que se implementa Redex) de servicios análogos.

La mecanización puede ser descargada de

`https://github.com/Mallku2/lua-redex-model`

El resto del presente trabajo está organizado del modo siguiente: §2 presenta una breve descripción de Lua, con énfasis en algunas de las características que incluimos en nuestro modelo; §3 introduce los conceptos básicos relativos al formalismo utilizado, mediante la presentación de un pequeño subconjunto de construcciones de Lua; §4 extiende §3 con la formalización de las partes más importantes e interesantes del modelo, sin incluir recolección de basura; §4.3 introduce *buena formación* de programas, y formaliza las propiedades que se pueden inferir a partir de la corrección de la definición de buena formación; §5 extiende el modelo con la inclusión de recolección de basura; §5.2 introduce el marco teórico dentro del cual se estudia corrección de GC sin interfaces; §6 introduce la mecanización realidad con PLT Redex, la cobertura lograda sobre la suite de tests del intérprete oficial y la utilización de las facilidades de Redex para obtener evidencia sobre la propiedad de progreso de nuestro modelo; §7 discute sobre trabajos relacionados; finalmente, §8 resume las contribuciones del presente trabajo y discute sobre futuras direcciones de investigación.

Lua, un lenguaje de scripting extensible

2

En este capítulo presentaremos una introducción breve a las características más prominentes de Lua. La persona versada en el lenguaje puede omitir esta sección, sin que ello dificulte la compresión de lo presentado posteriormente.

2.1. Clausuras de primera clase y tablas

Iniciamos la presentación de Lua con el estudio de una función[27] sencilla que implementa *memoización*[28], mostrada en la Figura 2.1. La función memoize espera como argumento una función fn y retorna su versión *memoizada*. Los valores de fn ya computados son almacenados en una tabla (t, en la línea 2). Las tablas en Lua son, en esencia, diccionarios indexados por cualquier valor Lua, excepto **nil** (único valor del tipo Nil, utilizado para representar la ausencia de valores, en contextos en donde es esperado un valor). Las tablas son el único tipo de dato estructurado de Lua. Sin embargo, la inclusión de mecanismos de meta-programación permite extender notablemente la funcionalidad de las mismas, más allá de lo ofrecido únicamente por la semántica de diccionarios, como veremos más adelante en el apartado 2.3.

La versión memoizada de fn es retornada a través de una función anónima, en la línea 3. Esta función toma un argumento, x, y, antes de computar fn(x), indexa la tabla t con x como clave (linea 4). Si el resultado de esa operación es **nil**, esto significa que x no es una clave de t. En el contexto de nuestro ejemplo, esto implica que es la primera vez que se intenta computar fn(x). Por lo tanto, se procede a calcularlo y a almacenar el resultado en la tabla, (linea 5–7). El valor resultado, ya sea computado o extraído de la tabla, es retornado, en la linea 8. La función memoize es utilizadas en las líneas 12–16 para construir la versión memoizada de de una función que calcula la sumatoria de los números naturales entre 1 y x.

Es importante notar que todos los procedimientos en Lua son valores de primera clase, que representan clausuras de alcance léxico. Así, las funciones anónimas que retorna memoize efectivamente capturan, en su entorno, a la tabla t.

27: Como es de esperar para un lenguaje imperativo, lo que el manual de referencia describe como *funciones* Lua, son lo que en otros lenguajes se llamarían *procedimientos*. Es decir, no son funciones en el sentido matemático usual. A lo largo del trabajo utilizaremos vocabulario del manual de referencia de Lua, que iremos introduciendo en la medida en la que se requiera.

28: Tomado de `lua-users.org/wiki/FuncTables`.

```
1  local function memoize(fn)
2    local t = {}
3    return function(x)
4      local y = t[x]
5      if y == nil then
6        y = fn(x); t[x] = y
7      end
8      return y
9    end
10 end
11
12 local memsum = memoize(
       function(x)
13   local a = 1
14   for i = 1,x do a = a + i
       end
15   return a
16 end)
```

Figura 2.1: Memoizacion en Lua.

2.2. Definiciones local y el entorno global

En el ejemplo de la función memoize, t y memsum están prefijadas por la palabra reservada **local**. **local** define variables locales, inicializadas con los *rvalues* (es decir, los valores a la derecha del símbolo '='). Si omitimos la palabra **local**, la instrucción se interpreta simplemente

```
x = 10
print (_ENV['x'])   -->> 10
_ENV['x'] = 0
print (x)   -->> 0
_ENV = {}
print ()   -- error, print ya
           -- no se
           -- encuentra en
           -- _ENV
```

Figura 2.2: El entorno global _ENV.

```
1  local MyClass = {}
2  MyClass.__index = MyClass
3
4  function MyClass.new(init)
5    local self = setmetatable
6                   ({}, MyClass)
7    self.value = init
8    return self
9  end
10
11 function MyClass:set_value(
       newval)
12   self.value = newval
13 end
14
15 function MyClass:get_value()
16   return self.value
17 end
18
19 local mc = MyClass.new(5)
20 print(mc:get_value()) -->> 5
21 mc:set_value(6)
22 print(mc:get_value()) -->> 6
```

Figura 2.3: POO implementada con meta-tablas.

[4]: Ierusalimschy y col. (2001), «The evolution of an extension language: a history of Lua»

29: Ver sección "Code Structure / Programming Paradigms" en `lua-users.org/wiki/LuaDirectory`.

30: Tomado de `lua-users.org/wiki/ObjectOrientationTutorial`.

como una *asignación*. Si la variable asignada ha sido previamente declarada como local, la operación actualiza su valor, como es esperado. Lo peculiar de la semántica de Lua es que las variables son *globales por defecto*: toda asignación sobre una variable no expresamente definida como local, define una nueva variable con alcance global. Más aún, las variables globales están definidas como campos de una tabla especial, el *entorno global*, cuyas claves son los identificadores de cada variable global definida. De este modo, una asignación sobre una variable que no está en *scope*, se reduce a una operación de asignación de tabla, el *entorno global*. De un modo semejante, la utilización, como expresión, de un identificador de variable que no está en scope, se interpreta como un indexado al entorno global, con la variable como llave. Los servicios de librería ofrecidos por Lua (como `print`) están asociados a variables globales, y, por lo tanto, definidos en el entorno global.

El entorno global está disponible a través de la variable _ENV, la cual está siempre en scope. Esto es, todo programa Lua se compila en el scope de la definición de la variable _ENV. Esto significa que cada ocurrencia de una variable x que *no* está en scope, es solo azúcar sintáctico para la operación _ENV['x']. Debido a que _ENV es, simplemente, una variable, es posible alterar el entorno global en el cual se ejecuta un programa, simplemente asignando _ENV a otra tabla. Figura 2.2 muestra un ejemplo sencillo de manipulación del entorno global, para ilustrar el concepto.

2.3. Meta-tablas

Una de las características más prominentes de Lua es su mecanismo de metaprogramación, *meta-tablas*, el cual permite adaptar el lenguaje a dominios de aplicación específicos. Gracias a las meta-tablas, Lua se presenta como un lenguaje sencillo y reducido en tamaño (objetivo de diseño de Lua [4]), que, al mismo tiempo, nos permite expresar una variedad notable de conceptos de programación[29].

Como ejemplo de las posibilidades de meta-programación en Lua, implementaremos algunos conceptos básicos de la programación orientada a objetos (POO). El ejemplo[30], en Figura 2.3, modela clases y objetos combinando tablas, funciones como valores de primera clase —ya visto en la sección 2.1— y el mecanismo de meta-tablas. También introduce algunos azúcares sintácticos provistos por Lua para brindar mejor soporte para POO. En Lua, una clase es implementada como una tabla, cuyos nombres de métodos son sus claves, y la implementación de tales métodos son los valores asociados a las claves. Los objetos también serán modelados con tablas, conteniendo como campos los nombres de los atributos como claves, asociadas a sus correspondientes valores.

En el ejemplo, tenemos una clase MyClass con su correspondiente constructor (linea 4), y solo un atributo, value, con sus métodos modificador (linea 11) y accesor (linea 15). Las signaturas de las funciones mencionadas son, en realidad, azúcares sintácticos para operaciones de asignación en campos de tabla, en donde, los lados izquierdos en las

asignaciones son, respectivamente, MyClass['new'], MyClass['set_value'] y MyClass['get_value']. Iremos de a poco introduciendo los conceptos utilizados en el ejemplo, pero, por ahora, notar que en las signaturas de los métodos en las líneas 11 y 15, se utilizó : en lugar de ., como sí se hizo en el constructor de la clase. Se trata de otro azúcar sintáctico que, en la práctica, implica que cuando se llame a estas funciones, se les pasará un parámetro extra, self, no especificado explícitamente en la signatura.

En la última linea de la Figura 2.3 mostramos cómo se crea una instancia de MyClass (linea 19), y cómo invocar sus métodos. En la línea 20 podemos observar que la invocación del método set_value hace uso de un nuevo azúcar sintáctico: mc:set_value(6) es equivalente a mc['set_value'](mc, 6) (recordar que el azúcar sintáctico usado en la signatura de set_value implica que hay un primer parámetro implícito, self).

Si las clases son las que contienen métodos, y los objetos solo contienen los atributos, ¿cómo es que la operación mc['set_value'] retorna el método set_value? La respuesta está relacionada con el mecanismo de meta-tablas, el cual es utilizado en las lineas 2 y 5. En la linea 5, la llamada al servicio de librería setmetatable define MyClass como la meta-tabla de la tabla provista como primer argumento (en este caso, una tabla vacía resultado de la ejecución del constructor de tablas { }). El resultado de esta llamada al servicio setmetatable es la tabla vacía pasada como primer argumento. Esta tabla será nuestro objeto.

El mecanismo de meta-tablas permite extender la semántica de una gran variedad de operaciones con tablas. Para redefinir la semántica de una operación debemos definir un *metamétodo*, el cual consiste en un valor que guardamos en una meta-tabla. Típicamente un meta-método es una función, pero, para ciertas operaciones, es posible utilizar valores de otro tipo. Para poder acceder al meta-método correcto, Lua define una serie de claves con las que debe asociarse el meta-método en una metatabla. En nuestro ejemplo, en la línea 2, se asocia la clave __index a la clase MyClass. Un meta-método asociado a la clave __index es consultado cuando se intenta acceder a una tabla utilizando una clave inexistente. Para este caso en concreto, la semántica de Lua especifica que se debe repetir la operación de indexado, ahora sobre el valor asociado a __index.

Considerando que MyClass es definida como la meta-tabla de cada nuevo objeto, línea 5, el nuevo comportamiento definido es el de indexar MyClass cada vez que intentamos indexar un objeto con una clave inexistente. En base a lo anterior, se entiende que la operación mc['set_value'] resulta en devolver el método set_value de la tabla MyClass.

Lua también define meta-métodos para manejar la asignación en campos de tablas con claves inexistentes, para llamadas a función sobre valores que no son clausuras, para operaciones binarias y unarias sobre operandos de tipo incorrecto, para definir finalizadores, e inclusive para redefinir el comportamiento de algunas de las funciones básicas de la librería.

2.4. Tuplas

Una función Lua puede retornar más de un valor. La lista de valores retornados (la cual denominaremos *tuplas*), tiene una semántica especial, dependiendo del lugar en el que ocurre. Observemos, por ejemplo, lo que ocurre en la Figura 2.4[31] . La función maximum recibe un arreglo (una tabla con claves numéricas), a, y retorna su mayor valor, junto con su correspondiente índice. La función hace uso del *iterador* de tablas ipairs de la librería estándar (linea 4), el cual, en cada iteración del ciclo for retorna una tupla con 2 valores: un índice, en a, y el valor asociado a ese índice. Cuando el bucle for termina, tenemos en la variable m el máximo valor de a, y en su correspondiente índice en mi. Ambos valores son retornados por maximum en la linea 10. La semántica de tuplas indica que, por el lugar en el que aparecen en la linea 10, ninguno de sus valores son descartados; el servicio print recibe todo lo retornado por maximum.

31: Los ejemplos de esta sección están extraídos de www.lua.org/pil/5.1.html .

```
1  function maximum (a)
2    local mi = 1             --
       maximum index
3    local m = a[mi]          --
       maximum value
4    for i,val in ipairs(a) do
5      if val > m then
6        mi = i
7        m = val
8      end
9    end
10   return m, mi
11 end
12
13 print(maximum
       ({8,10,23,12,5})) -->>
       23   3
```

Figura 2.4: Calculando el máximo valor almacenado en un arreglo, junto con su índice.

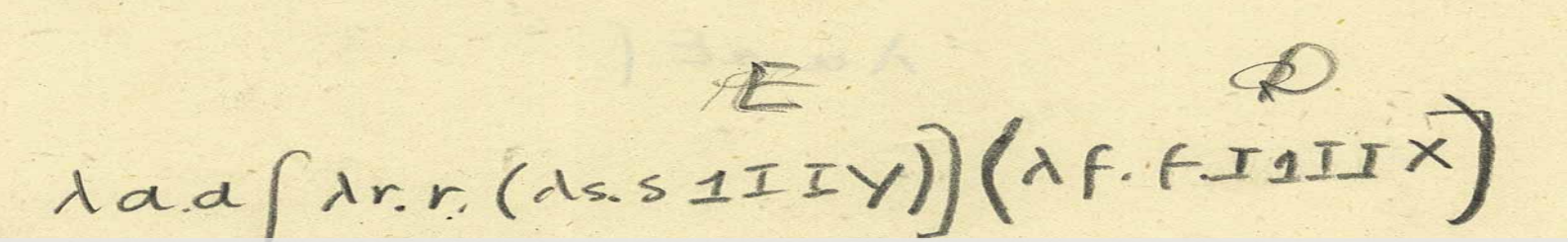
```
1  function unpack (t, i)
2    i = i or 1
3    if t[i] ~= nil then
4      return t[i],
5             unpack(t, i + 1)
6    end
7  end
8
9  a1, a2 = unpack({1,2,3})
10
11 b1, b2, b3, b4 = unpack
12                    ({1,2,3})
```

Figura 2.5: Extrayendo los valores de un arreglo.

La longitud de la tupla no necesariamente tiene que ser determinado estáticamente. Un ejemplo al respecto se observa en la Figura 2.5, donde se define una función unpack, la cual recibe un arreglo, y produce una tupla conteniendo los valores del arreglo. Hay varios detalles a notar en el código de ejemplo. Por comenzar, en Lua las funciones pueden ser llamadas con una cantidad de argumentos diferente a la explicitada en sus signaturas. Cada argumento esperado, pero que no es pasado en una llamada, tendrá el valor **nil**, y cada argumento extra será descartado.

Una situación en la que se puede aprovechar el comportamiento anterior, es en la definición de funciones recursivas, ayudándonos a simplemente ignorar el argumento sobre el que se hace recursión, en la primer llamada a la función. De ese modo, nos evitamos el tener que definir otra función cuyo único rol es el de llamar a la función recursiva, pasándole el correspondiente valor inicial sobre el que debe hacerse recursión. En unpack, el índice i, usado para indexar el arreglo, puede omitirse al llamar a unpack, en cuyo caso, la misma definición de unpack se encarga de que tenga inicialmente el valor 1 (los arreglos en Lua comienzan en 1), usando el operador or en la linea 2. Si i es **nil** (es decir, no se pasó como argumento en la llamada a unpack), entonces or retorna su segundo argumento, 1.

Retornando a la semántica de tuplas, otro aspecto que hace a la misma, consiste en que dependiendo del lugar en el que ocurran, algunos de sus valores podrían ser descartados. En la línea 9, del ejemplo, solo se toman los primeros 2 valores de la tupla retornada por unpack, descartando el tercero. Por otro lado, si un tupla aparece en un contexto en donde se requieren más valores de los que posee, se completa con valores **nil**. En la linea 11 podemos ver esa situación, en donde la cuarta variable asignada, b4, recibe **nil** como valor.



3 Fundamentos de semántica formal

En esta sección presentaremos los conceptos principales del formalismo empleado para precisar la semántica de Lua, a lo largo del presente trabajo. Comenzaremos describiendo los principales recursos utilizados, para luego ilustrarlos mediante la formalización de un subconjunto reducido de construcciones de Lua. *

*: En el encabezado de este capítulo: pieza de papel con nota manuscrita por Alan Turing. De writings.stephenwolfram.com/2019/08/a-book-from-alan-turing-and-a-mysterious-piece-of-paper/.

3.1. Semántica operacional

De acuerdo a lo presentado en §1, nos concentraremos en formalizar Lua como un lenguaje para describir procesos, entendidos estos como modificadores de estado. A su vez, también por lo expuesto en §1, sería deseable que la formalización sea de fácil comprensión para personas con diversos niveles de entendimiento e incumbencias sobre Lua. Sería útil, para esta tarea, el describir las reglas de ejecución haciendo economía de recursos, de modo de facilitar su comprensión.

El formalismo que vamos a utilizar comprende, por un lado, recursos tradicionales de semántica operacional small-step para lenguajes imperativos: ejecución descrita en términos de una relación entre programas, representación de estado utilizando referencias junto con una representación abstracta del almacenamiento, etc. En general, algunos de estos conceptos le resultan familiar a la persona que sabe programa con un lenguaje imperativo.

Por otro lado, es posible utilizar recursos tomados de *reduction semantics* (RS) para reducir la complejidad de las configuraciones de una semántica operacional: en lugar de una representación explícita de continuaciones, utilizar contextos de evaluación (concepto que introduciremos en breve); en lugar de utilizar una representación explícita del *entorno* (mapeo entre identificadores de variables y referencias al almacenamiento), utilizar una función que substituya identificadores por referencias directamente en el scope de una definición de variable.

A continuación formularemos brevemente cuales son las incumbencias de RS y los recursos que tomaremos de ese formalismo. Esto ayudará a comprender las diferencias entre los formalismos que se combinan en este trabajo y entender las particularidades de la herramienta de mecanización utilizada a posterior, PLT Redex[10] , principalmente diseñada para RS.

[10]: Felleisen y col. (2009), *Semantics Engineering with PLT Redex*

3.1.1. Semántica de reducciones

Para un lenguaje de programación dado, proveer una semántica de reducciones[10] para el mismo consiste en definir una relación entre términos del lenguaje, la cual, para un programa t, nos debiera permitir determinar si este retorna algún resultado, y qué resultado, o si no concluye. Nos referiremos a esto como el *comportamiento* de t.

[10]: Felleisen y col. (2009), *Semantics Engineering with PLT Redex*

La esencia de la forma en la que RS explica el comportamiento de un programa t reside en que los recursos a los que apela para esto son puramente sintácticos: solo tenemos programas y relaciones entre los mismos. Con tales herramientas, la tarea de determinar el comportamiento de t se transforma en la tarea de relacionar t con otros programas que sean cada vez más *simples* (precisaremos mejor este concepto a continuación) y que exhiban estos el mismo comportamiento que t [15] , así hasta arribar a programas que sean ellos mismos la representación más directa posible de la noción de resultado que hayamos definido (programas que, en el contexto de RS, se conocen como *formas canónicas*).

[15]: Plotkin (1975), «Call-by-name, call-by-value and the lambda-calculus»

Observemos que lo descrito anteriormente puede verse como un proceso consistente en ir simplificando t, procurando obtener, tras cada simplificación, un programa (o programas) que exhiba el mismo comportamiento que t, hasta llegar a una forma canónica (si existe) que, por causa de la propiedad que este proceso preserva, podremos decir que es el resultado que retorna t. Más aún, podríamos decir que tal forma canónica es tan solo una representación más directa de un resultado que t mismo representa, aunque no de forma tan evidente como lo hace la correspondiente forma canónica. El proceso descrito se conoce como *reducción* y puede verse como análogo al proceso de simplificar los términos a cada lado de una ecuación, hasta llegar a una ecuación que tenga una solución evidente.

Desde este punto de vista sobre la forma en la que determinamos el comportamiento de t, valiéndonos de este proceso que denominamos *reducción*, entonces podemos decir que los programas que vamos obteniendo durante la reducción son cada vez más *simples* porque están cada vez más cerca de una forma canónica, si esta existe.

Sintaxis

$e ::= (\lambda\ x\ .\ e) \mid x \mid e\ \ e$

x ::= identificadores de variables

Semántica

$(\lambda\ x\ .\ e_1)\ e_2\ \beta\ e_1[x \backslash e_2]$
para [__], una función de substitución que evita captura de variables

$(\lambda\ x\ .\ e)\ \alpha\ (\lambda\ x\ .\ e[x \backslash y])$
si y no ocurre libremente en e

Figura 3.1: λ-cálculo

3.1.2. λ-cálculo

Se puede ver que la descripción anterior, aunque informal y abstracta, pareciera coincidir con lo que entendemos por semántica operacional para un lenguaje puramente funcional. De hecho, el λ-cálculo[16] (Figura 3.1) es el ejemplo clásico de semántica de reducciones: las relaciones α y β [33] (denominadas *contracciones* o *nociones de reducción*) vinculan términos del lenguaje del λ-cálculo para los cuales se puede observar el mismo comportamiento.

[16]: Barendregt (1981), *The lambda calculus. Its syntax and semantics.*

33: Dejamos de lado la contracción η ya que no es muy utilizada en semántica de lenguajes de programación.

La relación α expresa que los identificadores de variables son arbitrarios y que, por lo tanto, podemos renombrarlos siempre que lo hagamos de manera consistente. Luego de un renombre de variables,

es de esperar que el comportamiento del programa resultante sea el mismo que el del original.

La contracción β expresa que el comportamiento de la aplicación de una abstracción λ es el mismo que el del cuerpo de la abstracción, habiendo substituido en él el parámetro formal por el actual.

El anterior enunciado tiene una componente operacional que, desde el paradigma funcional, se la utiliza directamente para explicar la ejecución de un programa funcional. Es decir que, para el caso de un lenguaje funcional, es exactamente lo mismo explicar cómo se ejecuta un programa o proveer una semántica de reducciones. Sin embargo, si nos interesara definir RS para el caso de conceptos ajenos al paradigma funcional, primero necesitaríamos disponer de su semántica operacional (la cual no siempre estará descrita en términos de programas y relaciones entre estos), desde donde sea posible entender cual es el comportamiento de los programas. Luego, recién con ese entendimiento, podremos intentar rescatar una semántica de reducciones para tales conceptos.

3.1.3. Semántica de reducciones en la literatura del área

De [17] , traducimos:

> *La ventaja de los lenguajes de programación funcionales es que incluyen automáticamente un sistema poderoso de razonamiento simbólico.* [Los lenguajes funcionales] *son variantes sintácticas y semánticas del λ-cálculo, y esta conexión provee un entendimiento abstracto de los programas, independiente de cualquier maquinaria de implementación. De este modo, a los fines de verificación, transformación, y comparación, quienes programan pueden manipular programas siguiendo un estilo algebraico.*

[17]: Felleisen (1987), «The calculi of Lambda-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages»

Como se puede leer, la propuesta de describir las reducciones, valiéndose únicamente del lenguaje y relaciones entre programas del lenguaje, permite, además de determinar el comportamiento de un programa, razonar sobre el mismo programa sin la necesidad de apelar a otras construcciones más que el código mismo del programa.

Es comprensible, entonces, que resulte interesante el proveer de una semántica de reducciones a lenguajes pertenecientes a paradigmas no funcionales. Al respecto, traducimos de [17] :

> [...] *un lenguaje debe ser expresivo y, simultáneamente, estar asociado a un sistema de razonamiento simbólico*

[17]: Felleisen (1987), «The calculi of Lambda-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages»

Para el caso de lenguajes imperativos, sin embargo, si tenemos en cuenta cómo entendemos la semántica operacional de los mismos, observaremos que, limitados a una descripción de las reducciones usando solo relaciones entre programas, tales reducciones no estarán describiendo la operación de los programas: por ejemplo, no habría mención al almacenamiento que es modificado por el programa (para conocer maneras de capturar una RS para lenguajes imperativos, ver

[10]: Felleisen y col. (2009), *Semantics Engineering with PLT Redex*
[18]: Servetto y col. (2013), «True small-step reduction for imperative object oriented languages»
[19]: Capriccioli y col. (2016), «An imperative pure calculus»

[10, 18, 19]). En tal caso hay que rescatar a RS como manera de relacionar programas con comportamiento equivalente (relación capturada previa definición de la correspondiente semántica operacional de los mismos), y no como descripción de la operación de un programa. La relación entre el paradigma funcional y RS no se debe generalizar.

Por lo expuesto, se entiende que semántica de reducciones posee su propia agenda, con objetivos que pueden ser interesantes, pero la cual no necesariamente coincide con la de una semántica operacional. Nos parece importante destacar y explicar esta diferencia. La literatura relativa a la utilización de RS, semántica operacional y PLT Redex para semántica de lenguajes de programación reales, por ejemplo [11-13] , no tiende a dedicar espacio a explicar el rol de los recursos de RS utilizados, en el contexto de la semántica operacional. Aunque esto es comprensible (se trata de artículos que necesitan explicar resumidamente los resultados de investigaciones sobre semántica de lenguajes reales y complejos, como JavaScript o Python), el estado de la literatura del área no ayuda a que personas recién llegadas puedan comprender la diferencia importante entre las semántica formales en discusión, y rescatar la utilidad que posee RS.

[11]: Guha y col. (2010), «The Essence of JavaScript»
[12]: Politz y col. (2013), «Python: The Full Monty: A Tested Semantics for the Python Programming Language»
[13]: Politz y col. (2012), «A tested semantics for getters, setters, and eval in JavaScript»

Habiendo precisado el objetivo que persigue RS, describiremos a continuación cómo utilizaremos esta para simplificar las configuraciones de nuestra semántica operacional.

3.1.4. Eliminando componentes de las configuraciones

Lo presentado aquí está fuertemente inspirado en lo desarrollado en [17] , en donde se discute cómo lograr una RS para un lenguaje imperativo, con instrucciones de control de flujo. Vamos a exponer brevemente las principales ideas de ese trabajo que son pertinentes para nuestra labor. Para mayores detalles, consultar la cita mencionada.

[17]: Felleisen (1987), «The calculi of Lambda-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages»

Si pretendiéramos describir una semántica operacional para un programa t de Lua 5.2, utilizando recursos tradicionales, arribaríamos a configuraciones de la forma:

$$(\sigma : \epsilon : \kappa : t)$$

donde σ sería una representación abstracta del almacenamiento que modifica el programa t; ϵ sería nuestra representación del entorno (mapeo parcial entre identificadores de variables y referencias al almacenamiento), necesario para explicar variables locales y su definición; κ sería una representación de las continuaciones, necesarias para explicar la semántica de instrucciones que operan sobre lo que resta por computar (como propagación de errores) y para explicar el orden de ejecución de las instrucciones de t.

De RS, con su objetivo de proveer la mencionada relación con dominio únicamente en el conjunto de programas, existen recursos que nos permiten *embeber* en t la información de 2 de las componentes de la configuración previa, de forma de que armoniza con nuestra semántica

operacional. Se trata del entorno ϵ y la representación de las continuaciones κ. En lo que sigue, explicaremos cómo ir embebiendo en t la información de cada una de esas 2 componentes, hasta llegar a una configuración que, esperamos, sea más simple de leer.

Entorno Embeber la información del entorno dentro del programa t se apoya en un recurso conocido en programación funcional: substitución de un identificador de variable por su valor. En la contracción β, presentada en la Figura 3.1,

$$((\lambda\ x\ .\ e_1)\ e_2)\ \beta\ (e_1[x \backslash e_2])$$

podemos ver cómo se utiliza ese recurso: para explicar la ejecución del programa (que involucra, en parte, explicar cómo se define una nueva variable) no es necesario mantener una mapeo explícito entre identificadores y valores[34]; nos basta con reemplazar consistentemente cada ocurrencia de una variable dada por su correspondiente valor.

34: La contracción β descrita no impone llamada a función por valor. Se trata de la contracción original del λ-cálculo, en donde el entorno mapearía identificadores de variables con expresiones arbitrarias. Para un lenguaje como Lua, sí requerimos de imponer llamada por valor.

Al recurso anterior podemos trasladarlo a lenguajes imperativos, también para explicar la definición y el significado de variables locales, con una salvedad: la naturaleza de las variables es diferente. Para explicar su semántica [20] necesitamos de 2 mapeos: uno entre identificadores de variables y referencias al almacenamiento (nuestra particular definición de entorno), y otro entre referencias y valores (el almacenamiento). En este contexto, nuestra función de substitución reemplazará identificadores por referencias frescas. Bajo esta noción de entorno, substitución no necesita evitar captura, pues no existe en el lenguaje la noción de *ocurrencia ligadora* de una referencia. Finalmente, al igual que lo que ocurre en un lenguaje funcional, la representación de clausuras no necesita de tratamiento especial: una definición de función sirve también como representación de una clausura, una vez que hemos reemplazado toda ocurrencia de variable libre por la correspondiente referencia, en el cuerpo de la función. Desarrollaremos más esto en §4.2.3 y §4.3.

[20]: Reynolds (2009), «Theories of Programming Languages»

Tras esta simplificación, podemos eliminar la componente ϵ de nuestras configuraciones, obteniendo $(\sigma : \kappa : t')$; donde ahora t' será un término semánticamente *enriquecido* que puede incluir no solo subtérminos del lenguaje de programación, sino también referencias al almacenamiento.

Correspondencia entre la implementación de un lenguaje y su semántica formal En el contexto de semántica para lenguajes de programación reales se presenta la situación en la que, como en las ciencias naturales, el objeto de estudio se trata de un fenómeno real, el cual solo explicamos aproximadamente mediante una descripción idealizada, que resulte útil a los fines de responder preguntas con rigor matemático. Qué tan compleja es nuestra explicación, qué tantos detalles incluimos en nuestra semántica formal, dependerá de las preguntas que queramos responder sobre los programas.

```
Lua 5.2.4  Copyright (C)
      1994-2015 Lua.org,
      PUC-Rio
>local f = 'foo';
 print(debug.getlocal(1,1));
 f()

f   foo
stdin:1: attempt to call
      local 'f' (a string value
      )
```

Figura 3.2: Definición de variable local: el entorno es preservado.

En nuestro caso, a los fines de explicar el fragmento de Lua 5.2 que hemos escogido, nos basta con suponer que parte de la información del entorno es descartada durante la ejecución de un programa: podemos descartar el identificador de una variable local recién definida y quedarnos solamente con su correspondiente referencia (es decir, podemos usar la substitución ya descripta). Sin embargo, notar que esto no es cierto en general, para la máquina virtual de Lua. Lua mantiene el mapeo completo entre identificadores y referencias, lo que le permite, por ejemplo construir mensajes de error más útiles a los fines de *debugging*. En [21] , Roberto Ierusalimschy y Luiz Henrique de Figueiredo (2 de los principales desarrolladores detrás de Lua), mencionan que una de las primeras facilidades que consideran útiles, por parte de un lenguaje de programación, para guiar en el debugging de programas, consiste en generar mensajes de error explicativos. Por ejemplo para referirse a una operación de llamada a función sobre un valor que no es de tipo clausura, los desarrolladores comentan que hasta Lua 3.2 se generaba un mensaje como "*call expression not a function*". En posteriores versiones se enriqueció con información como el identificador de variable involucrado en la expresión que generó el error obteniendo un mensaje como "*attempt to call a global 'f' (a nil value)*".

[21]: Biancuzzi y col. (2009), *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*

En la sesión con el intérprete oficial mostrada en la Figura 3.2, vemos que, luego de la definición de una variable local, se preserva una descripción completa del entorno. En el programa de ejemplo utilizamos un servicio de la librería `debug`, que ofrece facilidades de reflexión para inspeccionar y modificar el estado de la máquina (facilidades útiles para implementar *debuggers* para programas en Lua) . En particular, consultamos por la primer variable local definida en nuestro programa, obteniendo el identificador de la variable (`a`) y su valor asociado (`'foo'`). Finalmente, intentamos realizar una operación de llamada a función sobre `f` (que solo contiene un string), y obtenemos el correspondiente mensaje de error con los detalles mencionados en el párrafo anterior.

Expresar esto en nuestro modelo no será una incumbencia para nuestros objetivos de formalización, por lo cual, nos basta con utilizar substitución del modo descripto.

Continuaciones Tradicionalmente, antes de cada paso de ejecución de la máquina abstracta sobre la que se define la semántica operacional, se inspecciona el término que representa al programa y se identifica cual es la próxima instrucción a ejecutar. Se extrae esta instrucción, y lo que resta por computar se trasladará a otra componente de la configuración, en donde típicamente se expresará mediante una continuación: una representación de cómputos que deben ser procesados a posterior, comúnmente descriptos de modo de que se permita acceder cómodamente a la porción del resto de cómputo que posiblemente será utilizada luego de la ejecución de la instrucción actual.

A los fines de ilustrar la idea anterior, en la Figura 3.3 mostramos una semántica operacional con continuaciones para el lenguaje de términos del λ-cálculo, con llamada por valor y evaluación de izquierda a derecha. La máquina abstracta propuesta es conocida como máquina

Sintaxis

...

$v ::= \lambda\ x\ .\ e$

Semántica

* *Continuaciones (κ):*

- $\odot \in \kappa$
- $\kappa' \in \kappa,\ \mathbf{arg}\ e \times \kappa' \in \kappa$
- $\kappa' \in \kappa,\ \mathbf{fun}\ e \times \kappa' \in \kappa$

* $\rightarrow^{\kappa} \subseteq (\kappa, e)^2$:

$(\kappa, e_1\ e_2) \rightarrow^{\kappa} ((\mathbf{arg}\ e_2)\ \kappa, e_1)$

$((\mathbf{arg}\ e_2)\ \kappa, \lambda\ x\ .\ e_1) \rightarrow^{\kappa} ((\mathbf{fun}\ \lambda\ x\ .\ e_1)\ \kappa, e_2)$

$((\mathbf{fun}\ \lambda\ x\ .\ e)\ \kappa, v) \rightarrow^{\kappa} (\kappa, e\ [x \leftarrow v])$

Figura 3.3: Semántica operacional con continuaciones, para el lenguaje del λ-cálculo, con llamada por valor y evaluación de izquierda a derecha.

CC [10] (por las componentes de sus configuraciones: las *Continuaciones* y el *string de Control*). En esencia, extendemos la sintaxis del λ-cálculo de la Figura 3.1, agregando una nueva categoría sintáctica, la de los valores v. A su vez, describimos ahora la operación de los programas con una relación $\rightarrow^{\kappa} \subseteq (\kappa, e)^2$. Nuestro conjunto de continuaciones será representado por la letra κ (también utilizamos κ como variable cuantificada sobre el conjunto de continuaciones). Se tratará del menor conjunto que satisface las restricciones mencionadas en la Figura 3.3: las continuaciones serán listas, posiblemente vacías, de términos del lenguaje etiquetados para indicar el rol que cumplen en el programa: argumentos de una llamada a función (etiquetados con **arg**), o abstracción a aplicar (etiquetada con **fun**).

[10]: Felleisen y col. (2009), *Semantics Engineering with PLT Redex*

Las reglas mostradas explican la evaluación de izquierda a derecha de una aplicación de abstracción, mediante pasaje de parámetros por valor. Se puede ver cómo, durante la ejecución, una vez que identificamos cual es el próximo subtérmino a evaluar, ubicamos el resto del cómputo a comienzos de la lista κ, adecuadamente etiquetado para facilitar su posterior consulta una vez que evaluemos la instrucción actual.

En RS, en su búsqueda de reducciones sólo sobre programas, existe un recurso que nos permite desembarazarnos de las continuaciones, de modo de recuperar una semántica en términos de una relación $\rightarrow^{E} \subseteq e^2$. El recurso consiste en la utilización de *contextos de evaluación*, los cuales son términos del lenguaje con una posición especial, denominada *hole*, denotada con $[\![\ ]\!]$. Esto se puede observar en la Figura 3.4, en donde los contextos de evaluación (denotados con E), al ser términos, se definen mediante producciones como las mostradas. La ubicación de E dentro de un contexto indica las posiciones en donde puede ubicarse la próxima instrucción que se puede reducir, denominado *redex*. Las producciones de E provistas indican que, en una aplicación $e_1\ e_2$, primero debemos reducir e_1 (expresado por la producción $E \rightarrow E\ e$), y solo cuando hayamos obtenido un valor, podemos reducir e_2 (expresado por la producción $E \rightarrow v\ E$). Finalmente, para imponer el orden de reducción que expresan nuestros contextos, antes de cada paso debemos partir el programa en un contexto de evaluación de los expresados por E, y un redex. Para nuestro ejemplo, solo existe un redex posible: aplicación de una abstracción sobre un valor. Eso es lo expresado por la única regla que define a $\rightarrow^{E}$. Desarrollaremos más sobre el uso de contextos de evaluación en nuestro modelo, en la sección §3.2.

Sintaxis

...

$E ::= [\![\ ]\!] \mid E\ e \mid v\ E$

Semántica

$^* \rightarrow^{E} \subseteq e^2$:

$E[\![(\lambda\ x\ .\ e)\ v)]\!] \rightarrow^{E}$
$\quad E[\![e\ [x \leftarrow v]]\!]$

Figura 3.4: Semántica operacional con contextos de evaluación, para el lenguaje del λ-cálculo, con llamada por valor y evaluación de izquierda a derecha.

La conexión entre la noción de continuaciones y contextos de evaluación es directa: las continuaciones no son más que representaciones particulares del resto de cómputo, al igual que los contextos de evaluación. Notemos que $\rightarrow^{\kappa}$ es más adecuada como semántica operacional que sirva de guía para la implementación de un intérprete del lenguaje, ya que expresa una optimización por sobre el estilo de semántica descrito por $\rightarrow^{E}$: una vez que hemos particionado al programa en redex y continuación, mantenemos esa información de modo de que sea fácil de consultar, mientras que nos concentramos sólo en evaluar el redex. En $\rightarrow^{E}$ debemos volver a particionar el programa a cada paso.

Pero, por otro lado, el ejemplo sugiere que especificar con contextos de evaluación puede implicar una gran economía de recursos: basta con comparar la cantidad de reglas de ejecución necesarias para capturar $\rightarrow^{\kappa}$ y $\rightarrow^{E}$, y la simpleza de $\rightarrow^{E}$[35] por sobre $\rightarrow^{\kappa}$. En este sentido, la semántica con contextos de evaluación es más abstracta, al ocultar detalles que son de incumbencia para una implementación y no para comprender la esencia de los cómputos.

35: Notar que $\rightarrow^{E}$ expresa, en simultaneo, un cómputo y el orden en el que estos ocurren. Esto también podría expresarse por separado, a los fines de lograr reglas de ejecución aún más sucintas. Postergamos esta discusión para la sección §3.2.

Utilizando este último recurso de RS descrito, es que podremos aspirar a lograr una semántica para Lua 5.2 con configuraciones de la forma $(\sigma : t')$, en lugar de ternas $(\sigma : \kappa : t')$.

A continuación, comenzaremos a presentar nuestro modelo explicando la semántica de un fragmento reducido de conceptos de Lua 5.2.

3.2. Modelo de ejemplo

Habiendo discutido respecto al origen y propósito de los recursos que utilizaremos en nuestra formalización, comenzaremos a ilustrar su uso a través de la especificación de un subconjunto de conceptos de Lua.

Introduciremos los conceptos de programación a formalizar, presentando primero la semántica de sentencias y expresiones *puras* (preservando las 2 grandes categorías sintácticas de Lua), a través de una relación específica para estas construcciones; por otro lado, explicaremos sentencias imperativas (es decir, transformadoras de estado) mediante una relación sobre configuraciones que van a incluir una representación abstracta del almacenamiento. Luego, combinaremos ambas relaciones en una tercera, que va a precisar, a su vez, el orden de ejecución de las instrucciones de un programa, permitiéndonos proveer de semántica a programas Lua completos.

s ::= **if** *e* **then** *s* **else** *s* | **;**
e ::= *v* | *e binop e* | *unop e*
v ::= **nil** | *bool_literal*
binop ::= **and** | **or**
unop ::= **not**

Figura 3.5: Sintaxis de sentencias y expresiones del fragmento puro

3.2.1. El fragmento puro

Mostramos la gramática para programas *puros* en la Figura 3.5. Una sentencia ***s*** puede ser un *condicional* o *skip* (denotada con ;). Una expresión ***e*** puede ser un *value* (*v*), la aplicación de un operador[36] binario infijo (*binop*), o la aplicación de un operador unario (*unop*). Los valores que incluiremos son **nil**[37] , literales booleanos (**true** o **false**). Los operadores incluidos en este fragmento son los conectivos lógicos **and**, **or**, y **not**.

36: Utilizaremos la palabra *operador*, no en el sentido algebraico, es decir, una función con signatura $V^n \rightarrow V$, para un conjunto de valores V y n arbitrario; sino que la usaremos en el sentido expresado en el manual de referencia, esto es, para referirnos a una función arbitraria, sobre valores del lenguaje, provista como primitiva de programación: para realizar aritmética, manipular strings, etc.

37: Único valor del tipo Nil. Representa la *ausencia* de un valor resultado de un cómputo; ausencia de parámetros pasados en un llamada; ausencia de valores a asignar, en una operación de asignación, noción que cobra sentido al considerar que, en Lua, existe la asignación múltiple de variables, etc.

Con este fragmento de Lua no estamos en condiciones de escribir programas demasiado interesantes, pero en las siguientes secciones iremos acrecentado el fragmento con nuevas construcciones, hasta llegar a Lua.

La Figura 3.6 presenta la semántica operacional para la sentencia condicional, modelada con la relación $\rightarrow^{s/e}$, con dominio en el conjunto de sentencias y expresiones puras. La primer regla enuncia que, si la guarda de la sentencia **if** se trata de cualquier valor, diferente de **nil**

y **false**, entonces la guarda evalúa a **true**, y, por lo tanto, se ejecuta a continuación el código de la rama **then**.

Notar que, dada la naturaleza small-step de la semántica, no tendremos premisas: solo, a lo sumo, condiciones laterales. Es hábito en el área el reutilizar el espacio de las premisas para mencionar las condiciones laterales. Cuando no haya condiciones laterales que mencionar, omitiremos la linea divisora con la conclusión.

También notar que estas reglas expresan únicamente la evaluación small-step de la guarda de un condicional, sin ofuscar la descripción con información que no es pertinente para comprender la esencia del cómputo: por ejemplo, no hay mención al almacenamiento, ni incluimos información sobre orden de ejecución de las instrucciones.

Volviendo a la Figura 3.6, la segunda regla explica lo esperado: si la guarda evalúa a **false** o **nil**, se ejecutan las instrucciones de la rama **else**.

$$\frac{v \notin \{\textbf{nil}, \textbf{false}\}}{\textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \ \rightarrow^{s/e} \ s_1}$$

$$\frac{v \in \{\textbf{nil}, \textbf{false}\}}{\textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \ \rightarrow^{s/e} \ s_2}$$

Figura 3.6: Semántica de la sentencia condicional.

Figura 3.7 presenta la semántica dada a las expresiones incluidas en nuestro fragmento. En lugar de describir la operación de cada operador primitivo, vamos a proveer una descripción declarativa de la semántica de los mismos, utilizando una función que denotamos con δ. Es decir, pondremos énfasis en el resultado que obtenemos al usar un operador, en lugar de describir cómo el operador produce el resultado, paso a paso. Tomamos este recurso de [11, 12] , aunque lo reinterpretaremos en nuestro contexto, con nuestro particulares objetivos. Los beneficios de utilizar este estilo podrán verse con mayor claridad cuando describamos la semántica de operaciones más complejas (como aritméticas), e inclusive servicios de librería. Para mantener cohesión en nuestra semántica, todo operador primitivo y servicio de librería será descripto de manera declarativa, usando δ. Por este motivo, δ tiene un dominio e imagen complejos, pudiendo incluir, inclusive, el almacenamiento. La función δ puede ser vinculada con la *función de interpretación* utilizada en ISWIM [22] , aunque aquí se la utiliza con motivos ligeramente distintos.

$$\textbf{not } v \rightarrow^{s/e} \ \delta(\textbf{not}, v)$$

$$v \ \mathit{binop} \ e \ \rightarrow^{s/e} \ \delta(\mathit{binop}, v, e)$$

Figura 3.7: Semántica de expresiones.

[11]: Guha y col. (2010), «The Essence of JavaScript»
[12]: Politz y col. (2013), «Python: The Full Monty: A Tested Semantics for the Python Programming Language»

[22]: Landin (1966), «The next 700 programming languages»

Figura 3.8 presenta una descripción simplificada de las ecuaciones usadas para capturar las descripción de los operadores booleanos, mediante δ[38]. Aquí se observa la misma lógica en la interpretación de valores en contextos booleanos, como la empleada para evaluar la guarda de un condicional: por ejemplo, $\delta(\textbf{not}, v)$ mapea a **false** a todo v distinto de **false** y de **nil**.

$$\delta(\textbf{and}, v, e) = \begin{cases} v & \textbf{if } v \in \{\textbf{false}, \textbf{nil}\} \\ e & \textbf{otherwise} \end{cases}$$

$$\delta(\textbf{or}, v, e) = \begin{cases} v & \textbf{if } v \notin \{\textbf{false}, \textbf{nil}\} \\ e & \textbf{otherwise} \end{cases}$$

$$\delta(\textbf{not}, v) = \begin{cases} \textbf{true} & \textbf{if } v \in \{\textbf{false}, \textbf{nil}\} \\ \textbf{false} & \textbf{otherwise} \end{cases}$$

Figura 3.8: Función δ: operadores booleanos.

Una observación final sobre el modelo introducido hasta el momento: mientras que intentamos mantener la distinción sintáctica entre sentencias y expresiones, a los fines de presentar una gramática familiar a la persona que programa con Lua, no dejamos que esta distinción altere la estructura de la semántica. La distinción entre expresiones y sentencias, en Lua, es explicada como resultado de cuestiones de gramática concreta[39], que no son pertinentes para nuestros estudios. La experiencia previa en la formalización de la semántica de Lua[23] , muestra que preservar esa distinción al nivel de la semántica no ayuda a una mejor compresión de los conceptos, mientras que deriva en una proliferación innecesaria de relaciones entre términos.

38: Las ecuaciones originales hacen uso de *expresiones parentizadas*, las cuales introduciremos en §4.1.

[23]: Soldevila y col. (2017), «Decoding Lua: Formal Semantics for the Developer and the Semanticist»

39: Ver `lua-users.org/wiki/ExpressionsAsStatements`.

s ::= ... | **local** x **=** e **in** s
| x **=** e
| r **=** e
e ::= ... | r
r ::= referencias a σ

Figura 3.9: Construcciones imperativas del lenguaje.

40: Sobre sintaxis concreta: esta modificación no entra en conflicto con los identificadores de variable en un programa Lua, pues **in** ya existe como palabra clave; podremos compilar programas Lua reales hacia nuestro lenguaje, sin que esta modificación introduzca conflictos. Cuando no sea así, tendremos en cuenta el léxico de la sintaxis concreta de Lua, para agregar nuevas palabras claves que no pertenezcan al mismo espacio de nombres que las variables.

3.2.2. El fragmento imperativo

Extenderemos el lenguaje adicionando estado. En concreto, añadiremos una construcción para la definición de variables (imperativas) de alcance léxico y asignación de variable. Las nuevas construcciones se muestran en la Figura 3.9. Notar que, como se mencionó en §2.2, la definición de variables locales debe estar precedida por la palabra clave **local**; de otro modo, la variable tendrá alcance global. Por otro lado, por la forma en la que manipularemos entorno (utilizando substitución, como se mencionó en 3.1.4), necesario para explicar la definición de variables locales, agregaremos en la instrucción explícitamente el alcance de la definición precedido por la palabra clave **in**[40]. Si bien la manipulación del entorno mediante substitución nos ayuda a simplificar las configuraciones (y a lograr una representación de clausuras sencilla; ver §4.2.3), nos fuerza a tener que incluir referencias como expresiones del lenguajeinline]Qué hay del reference concept, donde las referencias son incluidas como expresiones del lenguaje?, como se puede ver en la Figura 3.9, denotadas mediante r. A tales construcciones las denominaremos *construcciones de tiempo de ejecución*, para hacer hincapié en que son solo objetos que existen en tiempo de ejecución, agregados al lenguaje solo a los fines de poder formalizar una semántica operacional. A continuación explicaremos referencias y la semántica de instrucciones imperativas.

La semántica operacional comúnmente entendida, de las construcciones que hemos agregado, las presenta como modificadoras de estado. Por lo tanto, para reflejar ello, vamos a comenzar a apartarnos del estilo de *semántica vía sintaxis*, utilizado para el fragmento puro del lenguaje, para agregar a nuestro modelo una representación de estado como una función parcial desde un conjunto de *referencias* hacia los valores, la cual la denotaremos como σ. Nos referiremos a σ como el *almacenamiento de valores* (ya que, a posterior, agregaremos otro almacenamiento para describir la semántica de tablas).

Con respecto a las referencias, no forzaremos ninguna representación en particular, simplemente pediremos que cumplan con ciertas propiedades elementales. Específicamente, pediremos que el dominio de σ (denotado como $\mathsf{dom}(\sigma)$) sea un conjunto finito, con elementos que pueda ser representados sintácticamente, pero distinguibles de cualquier otro objeto del lenguaje. Asumiremos también que es siempre posible obtener referencias frescas al almacenamiento. Escribiremos $(r, v), \sigma$ para representar una extensión del almacenamiento σ con una referencia r, mapeada al valor v, y asumiremos que r es fresca, es decir, $r \notin \mathsf{dom}(\sigma)$.

$$\frac{\sigma' = (r, v), \sigma}{\sigma \ : \ \textbf{local}\ x \ = \ v\ \textbf{in}\ s \ \rightarrow^{\sigma} \ \sigma' \ : \ s[x \backslash r]}$$

$$\frac{\sigma' = \sigma[r := v]}{\sigma : r = v \ \rightarrow^{\sigma} \ \sigma' \ : \ ;}$$

$$\sigma \ : \ r \rightarrow^{\sigma} \ \sigma \ : \ \sigma(r)$$

Figura 3.10: Semántica de variables imperativas.

Figura 3.10 presenta la semántica para las construcciones que hemos agregado al lenguaje. Para esto, utilizaremos una nueva relación $\rightarrow^{\sigma}$, con dominio en los pares ordenados de la forma (σ, t), con t una sentencia o expresión de nuestro lenguaje.

La primer regla formaliza la definición de variables locales. Esta expresa la semántica usual: cuando la expresión inicial de la variable a definir, x, ha sido reducida a un valor v, extendemos el almacenamiento σ

con un nuevo par (r, v), para un referencia r fresca. Finalmente, modificamos el entorno: reemplazamos cada ocurrencia del identificador x por r, en el scope correspondiente, s. Denotamos esta substitución con $s[x \backslash r]$. Como se mencionó en §3.1.4, la función de substitución que utilizamos no necesita evitar captura, pues no existe en el lenguaje el concepto de *ocurrencia ligadora* de una referencia.

La segunda regla de Figure 3.10 explica la semántica operacional usual de asignación de variable. Mediante $\sigma[r := v]$, para una $r \in \mathsf{dom}(\sigma)$, denotaremos al almacenamiento en el cual el valor referido por r ha sido alterado por v. Más precisamente, se trata de un almacenamiento σ' tal que $\mathsf{dom}(\sigma') = \mathsf{dom}(\sigma)$, donde $\sigma'(r) = v$ y $\forall r' \in \mathsf{dom}(\sigma'), r' \neq r \Rightarrow \sigma'(r') = \sigma(r')$. Notar que la asignación reduce a la sentencia *skip* **;**, de modo de indicar que no hay otro cómputo por ejecutar.

Finalmente, recordar que en la Figura 3.9 agregamos referencias también como expresiones. Nos resta explicar qué es una referencia, cuando ocurre como expresión en un programa. Relativo a esto, observemos que, previamente , en trabajos de semántica operacional para lenguajes reales, como λ_{JS} [11] , se experimentó con lo que a veces se conoce como *reference concept*: la introducción, en un lenguaje de programación, de la noción de estado como un concepto de programación opcional, a disposición de la persona que programa para usarlo solamente allí donde resulte necesario definir cómputos imperativos [20] . Una forma de implementar este concepto consiste en agregar referencias como expresiones del lenguaje, junto con operadores para crear y desreferenciar explícitamente a las referencias. De este modo, podemos disponer, por ejemplo, de variables declarativas (o funcionales) que nos permitan mantenernos en el paradigma funcional, y crear explícitamente referencias para tener variables imperativas solamente en aquellos fragmentos del programa en donde resulte conveniente.

[11]: Guha y col. (2010), «The Essence of JavaScript»

[20]: Reynolds (2009), «Theories of Programming Languages»

En experiencias posteriores a la citada, sobre formalización de semántica de JavaScript [13] , se reconoció que esta forma de inclusión de estado no ayudó a lograr una mayor comprensión de los conceptos modelados (en particular, en conjunción con registros funcionales[41], para modelar objetos), mientras que sí incrementó la complejidad del código JavaScript compilado, al momento de contrastar una mecanización del modelo formal contra suites de test de algunas implementaciones de JavaScript.

[13]: Politz y col. (2012), «A tested semantics for getters, setters, and eval in JavaScript»

41: Registros en donde, la operación que altera los valores de sus campos, no está definida en términos de estado, sino que fabrica un nuevo registro que contiene la modificación pretendida.

Basándonos en las anteriores experiencias, nosotros no agregaremos operadores explícitos para crear referencias (como ya se vio en la definición de variables locales), ni tampoco para desreferenciarlas. En su lugar, vamos a incluir una regla que desreferencie implícitamente, cada vez que una referencia ocurra como expresión. La última regla de Figura 3.10 solamente formaliza la operación de desreferenciado. Resta restringir esta operación a las referencias que ocurren como expresión. Esto podremos conseguirlo con los recursos que presentamos a continuación.

3.2.3. Ejecución de programas completos

Disponemos de 2 relaciones diferentes, cada una especializada en explicar componentes distintas de un programa: $\rightarrow^{s/e}$ describe cómputos puros y $\rightarrow^{\sigma}$ describe cómputos imperativos. Lo que resta por definir para describir la ejecución de programas completos, es cómo combinar y ordenar los cómputos descriptos por las mencionadas reglas. Con este fin, definiremos la relación $\mapsto$, la cual especifica cual es el próximo cómputo a ejecutar dentro del programa. Nuestra intención es la describir una semántica operacional determinista, con lo cual, debiera ocurrir que $\mapsto$ identifica a lo sumo un cómputo a ejecutarse, en cada paso.

Capturaremos sintácticamente la noción de *próximo cómputo a ejecutarse*, mediante contextos de evaluación como los descriptos en §3. Nuevamente, se tratarán de términos de nuestro lenguaje con una posición especial, *hole*, indicada mediante $[\![\ ]\!]$. Para cada constructor de términos de nuestro lenguaje, y para cada posición dentro de los mismos en donde puede ubicarse la próxima instrucción a ejecutar, tendremos un constructor de contextos de evaluación, indicando tal posición con un hole. El determinismo pretendido de nuestra semántica surgirá del hecho de que, dado un término t, si usamos los contextos de evaluación para determinar cual es el subtérmino de t que representa al próximo cómputo a ser ejecutado, tal subtérmino, si existe, es único y su posición es la indicada por el hole del correspondiente contexto de evaluación. Esta propiedad se la conoce como de *descomposición única*, en el contexto de semántica de reducciones.

$$
\begin{aligned}
E ::= \ & [\![\ ]\!] \mid \textbf{if } E \textbf{ then } s \textbf{ else } s \\
& \mid \textbf{local } x = E \textbf{ in } s \\
& \mid x = E \mid E \ \mathit{binop}\ e \mid \mathit{unop}\ E
\end{aligned}
$$

Figura 3.11: Contextos de evaluación.

La Figura 3.11 define los contextos de evaluación E necesarios para el subconjunto de construcciones de Lua que hemos introducido hasta el momento. Podemos ver que las producciones imponen el orden de evaluación esperado: en un condicional, primero debemos evaluar la guarda; para la definición de una variable local, debemos evaluar primero la expresión inicial asignada a la variable; en la aplicación de un operador binario, evaluamos los operandos de izquierda a derecha[42]. Notar que, a diferencia de la gramática presentada hasta el momento, estamos utilizando una única categoría de contextos de evaluación que incluye sentencias y expresiones. Mientras que sería posible mantener esa distinción también en los contextos de evaluación, esto nos forzaría a utilizar definiciones complejas para describir aquellos contextos que rodean únicamente a expresiones, mientras que la única razón para la distinción entre las categorías de sentencia y expresión ha sido la presentar una gramática que sea familiar a la persona que programa con Lua.

42: Nuestra definición fuerza un orden de evaluación de las expresiones, de izquierda a derecha. Inclusive a pesar de que el manual de referencia no especifica este comportamiento, se trata del orden de evaluación implementada en los 2 principales intérpretes de Lua 5.2: el oficial y LuaJIT (`luajit.org`).

$$
\frac{t \ \rightarrow^{s/e} \ t'}{\sigma : E[\![t]\!] \ \mapsto \ \sigma : E[\![t']\!]}
\qquad
\frac{\sigma : t \ \rightarrow^{\sigma} \ \sigma' : t'}{\sigma : E[\![t]\!] \ \mapsto \ \sigma' : E[\![t']\!]}
$$

Figura 3.12: Semántica de programas.

Figura 3.12 define la relación $\mapsto$. En su definición hacemos uso del patrón $E[\![t]\!]$, en el lugar de los términos de las configuraciones relacionadas por $\mapsto$. Que un término t' encaje con tal patrón significa que podemos particionarlo en un contexto E y subtérmino t. Considerando la semántica de contextos de evaluación, esto significa que t' contiene un subtérmino t que se trata de la próxima instrucción a ejecutarse, y el resto del cómputo es E. La relación $\mapsto$ simplemente ubica los

cómputos definidos por las relaciones anteriores, en el orden impuesto por nuestros contextos de evaluación.

Ejemplo

Como ejemplo, consideremos la Figura 3.13. Presenta la ejecución, mediante $\mapsto$, de un programa sencillo que implementa la semántica del operador **not**, sobre el valor de una variable *b*. Tras cada paso de ejecución, indicamos cual es el contexto de evaluación E correspondiente, dejando a quien lee la tarea de identificar la instrucción ejecutada en cada paso. Notar que, en el primer paso, la variable *b* es reemplazada por una referencia fresca r1.

Habiendo expuesto los principales conceptos del formalismo usado, estamos en condiciones de presentar los principales aportes del presente trabajo.

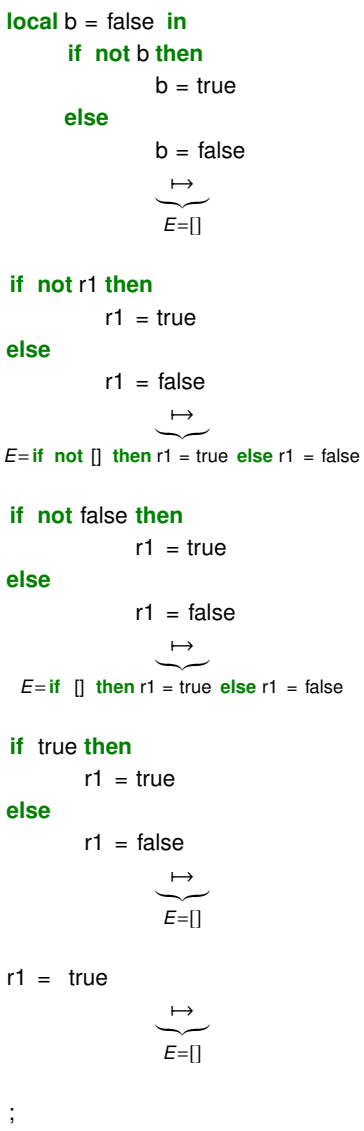


Figura 3.13: Traza de ejecución de un programa sencillo.

Una descripción formal de Lua 4

En este capítulo presentaremos los aspectos más relevantes de nuestra formalización de Lua 5.2. §4.1 cubre el fragmento puro del lenguaje; §4.2 describe el fragmento imperativo; §4.2.4 presenta conceptos añadidos para incluir servicios de librería en nuestro modelo; §4.2.5 describe la semántica de meta-tablas y §4.2.6 extiende el modelo incluyendo con manejo de errores, y combina los fragmentos previos en una única relación que explica la ejecución de programas completos. Presentamos la semántica completa en el apéndice §A.

4.1. Fragmento puro

Extendemos el fragmento puro presentado en §3, con las construcciones presentadas en Figura 4.1. Incluimos bucles *while* y saltos (*breaks*); concatenación de sentencias; números y strings; y operaciones aritméticas, de comparación, operaciones sobre strings (concatenación y longitud) y tablas (longitud; tablas son introducidas en §4.2.1). Notar que los operadores binarios incluidos los definimos bajo una nueva categoría, *strictbinop*, que nos ayudará a definir el comportamiento de operadores que requieren de la evaluación de ambos operandos, antes de poder efectuar el cómputo que representan (a diferencia de lo que ocurría con operadores booleanos).

s ::= ... | **while** e **do** s | **break** | $s\ s$

v ::= ... | *number* | *string*

e ::= ... | e *strictbinop* e

strictbinop ::= **+** | **-** | ***** | **/** | **^** | **%** | **..** | **<** | **≤** | **>** | **≥** | **==**

unop ::= ... | **-** | **#**

Figura 4.1: Sintaxis de instrucciones puras.

4.1.1. Semántica de sentencias

Extendemos acordemente la relación $\rightarrow^{s/e}$ (Figura 4.2) para que explique la semántica de las nuevas sentencias.

Las primeras 4 reglas definen la operación de un bucle **while**. Uno de los objetivos que guían el desarrollo del modelo es el de reducir la complejidad del código Lua compilado hacia nuestro modelo, como ya se discutió en §3.2, en relación al modo en el incorporamos estado en nuestro modelo. En esencia, cuanto más directa sea la relación entre Lua y nuestro lenguaje, más fácil será la tarea de compilación de la suite de test. Las primeras 2 reglas que presentamos en Figura 4.2 están vinculadas con ese objetivo.

La primer regla, por un lado, muestra cómo indicamos en el programa el punto desde el cual debe continuar la ejecución, en caso de encontrarnos con una sentencia **break** dentro de un bucle **while**. Con motivo de que no tenemos representaciones explicitas de continuaciones, en donde podríamos codificar esta información, hacemos uso del programa con estos fines. El costo de configuraciones con menos componentes es el de tener que codificar más información dentro del programa.

while e **do** s $\rightarrow^{s/e}$ $\langle\!|$ **$iter** e **do** s $|\!\rangle_{\text{BREAK}}$

$iter e **do** s $\rightarrow^{s/e}$ **if** e **then** s **; $iter** e **do** s **else ;**

$\langle\!|\ E_{lf}[\![$ **break** $]\!]\ |\!\rangle_{\text{BREAK}} \rightarrow^{s/e}$ **;**

$\langle\!|\ ;\ |\!\rangle_{\text{BREAK}} \rightarrow^{s/e}$ **;**

; $s \rightarrow^{s/e} s$

Figura 4.2: Semántica de sentencias puras.

Para este caso, extendemos la gramática con una construcción nueva, $\langle\!|\ s\ |\!\rangle_{\text{BREAK}}$. Más adelante explicamos su semántica.

Por otro lado, en el intento de simplificar el código compilado, le trasladamos a la semántica la tarea de incrustar esta información dentro del código, cuando corresponda[44]. Finalmente, la traducción de un bucle **while** *e* **do** *s* **end** en **$iter** *e* **do** *s* **end** se hace a los fines de indicar que ya ha sido incrustada la información de salto necesaria para formalizar la instrucción **break**, a los fines de evitar que se repita la operación. En Figura 4.3 mostramos las extensiones hechas a la gramática para incluir las construcciones necesarias para explicar el comportamiento de un bucle **while** y de una sentencia **break**.

44: En λ_{JS}, por ejemplo, se introduce esta información al momento de compilar el código, pues se trata de una propiedad que, evidentemente, puede ser determinada en tiempo de compilación.

$$
\begin{array}{lll}
\mathit{label} & ::= & \textsc{Break} \\
s & ::= & \ldots \mid \textbf{\$iter}\ e\ \textbf{do}\ s \\
 & \mid & \langle\!|\ s\ |\!\rangle_{\mathit{label}} \\
E_{\mathit{lf}} & ::= & \text{contextos de evaluación} \\
 & & \text{sin } \langle\!|\ s\ |\!\rangle_{\mathit{label}}
\end{array}
$$

Figura 4.3: Construcciones de tiempo de ejecución para **while** y **break**.

Se puede observar que, del modo en el que procedimos, simplemente hemos trasladado la complejidad del *parser* hacia la semántica sobre la cual debemos razonar formalmente a posterior. En general, no hay una solución óptima. Siendo que, antes de poder dedicarnos a razonar formalmente sobre las propiedades de la semántica, se encuentra el paso de testeo de conformidad con respecto a alguna suite de test, e inspección del modelo por parte de personas con entendimientos diversos sobre Lua (tarea que nunca concluye), preferimos priorizar el simplificar estas tareas.

Volviendo a las reglas presentadas en la Figura 4.2, la segunda regla describe la ejecución usual de un bucle **while** en términos del uso de una sentencia condicional, para evaluar la guarda del bucle antes de una iteración.

La tercer regla de la Figura 4.2 describe la semántica de una sentencia **break**: cuando se encuentra dentro de un bloque etiquetado con el punto hacia el cual debe saltar, el resto de cómputo contenido dentro del bloque es descartado, efectivamente modelando el efecto de salto por fuera del cuerpo del bucle. Para poder identificar unívocamente el punto hacia el cual debe saltar la sentencia **break**, hacemos uso de una nueva categoría de contexto de evaluación, E_{lf}, que describiremos en §48, pero, en esencia, se trata de contextos de evaluación que no incluyen a sentencias etiquetadas, como $\langle\!|\ s\ |\!\rangle_{\mathit{label}}$. En conjunto, cuando una sentencia encaja con el patrón $\langle\!|\ E_{\mathit{lf}}[\![\ \textbf{break}\ ]\!]\ |\!\rangle_{\text{BREAK}}$, significa que la etiqueta BREAK es la primera que ocurre, conteniendo a la sentencia **break**.

La cuarta regla en la Figura 4.2 explica la finalización de un bucle **while**, cuando no ocurrió un salto mediante **break**. En tal caso, tras la última instrucción simplemente removemos la etiqueta BREAK.

La última regla de la Figura 4.2 formaliza el orden normal de ejecución de una concatenación de sentencias: una vez que concluyó la ejecución de una sentencia (**;**), la ejecución continua con la siguiente sentencia, de izquierda a derecha.

$$
\frac{\mathit{op} \in \{\textbf{+}, \textbf{-}, \textbf{*}, /, \hat{}, \textbf{\%}, <, \leq\} \quad v_1, v_2 \in \mathit{number}}{v_1\ \mathit{op}\ v_2 \ \rightarrow^{s/e}\ \delta(\mathit{op}, v_1, v_2)}
$$

$$
\frac{\begin{array}{c}\mathit{op} \in \{\textbf{+}, \textbf{-}, \textbf{*}, /, \hat{}, \textbf{\%}\} \\ v_1 \notin \mathit{number} \vee v_2 \notin \mathit{number} \\ i \in \{1,2\},\ v_i = \delta(\textbf{tonumber}, v_i, 10) \\ v_i \neq \textbf{nil}\end{array}}{v_1\ \mathit{op}\ v_2 \ \rightarrow^{s/e}\ \delta(\mathit{op}, v_1, v_2)}
$$

$$
\frac{\begin{array}{c}\mathit{op} \in \{\textbf{+}, \textbf{-}, \textbf{*}, /, \hat{}, \textbf{\%}\} \\ v_1 \notin \mathit{number} \vee v_2 \notin \mathit{number} \\ \delta(\textbf{tonumber}, v_1, 10) = \textbf{nil} \\ \vee \\ \delta(\textbf{tonumber}, v_2, 10) = \textbf{nil}\end{array}}{v_1\ \mathit{op}\ v_2 \ \rightarrow^{s/e}\ \langle\!|\ v_1\ \mathit{op}\ v_2\ |\!\rangle_{\text{ARITHWO}}}
$$

Figura 4.4: Semántica de expresiones aritméticas.

4.1.2. Semántica de expresiones

Habiendo definido la semántica de las sentencias, continuamos con las expresiones. En la Figura 4.4 explicamos cómo embebemos la inter-

pretación de los operadores binarios, en la ejecución. La primer regla muestra la evaluación de las operaciones aritméticas y relaciones de orden[45].

La segunda regla muestra la coerción implícita que realiza Lua en operaciones aritméticas, cuando alguno de los operandos es un valor no numérico que puede ser reinterpretado como un número. En la práctica serán strings que solo contengan dígitos numéricos, siendo la excepción la representación de constantes hexadecimales, con parte decimal y exponente opcionales, como se muestra en la Figura 4.5[46].

En este caso, Lua realiza una coerción usando el servicio de librería tonumber, descripto mediante δ(**tonumber**), el cual recibe el valor str a convertir, y un número que indica la base numérica sobre la cual debe interpretarse el número expresado en str. Para coerciones de operandos aritméticos, la base por defecto es 10.

La tercer regla muestra cómo se delega, al mecanismo de meta-tablas, el tratamiento de expresiones aritméticas sobre operandos no contemplados en su semántica original. En tal caso, etiquetamos la expresión con información sobre el error ocurrido (las correspondientes extensiones a la gramática se muestran en la Figura 4.6). En este punto, la ejecución explicada solamente mediante $\rightarrow^{s/e}$, se detendría. El mecanismo de meta-tablas es quien toma estos términos etiquetados con información sobre situaciones inesperadas, y define cómo continua la ejecución (considerando también la información contenida en las meta-tablas).

Separar las incumbencias de ese modo nos permite explicar la semántica de expresiones puras utilizando una relación con un dominio simple (solo términos del lenguaje). A su vez, incrustando información sobre la situación inesperada hallada, nos permite simplificar la definición de la relación que captura la semántica del mecanismo de meta-tablas, como veremos en §4.2.5.

Omitimos las reglas para el único operador aritmético unario (cálculo del negativo de un número), ya que sigue, esencialmente, el mismo patrón que lo expresado para operadores binarios.

Un conjunto de reglas semejantes aplica para operaciones binarias sobre strings (Figura 4.7): concatenación y evaluación de orden lexicográfico. Hay pocas observaciones para hacer al respecto: la primer regla explica las operaciones binarias primitivas sobre strings, en condiciones normales; por otro lado, el operador de concatenación (**..**) puede operar tanto sobre strings como sobre números, haciendo uso del servicio básico tostring en el último caso; lo anterior lo formalizamos en la segunda regla de la Figura 4.7.

Aunque el servicio tostring está definido para cualquier valor Lua, no solo para números, la semántica por defecto de la concatenación no explica qué hacer cuando alguno de los operandos no es un string o número. En tal caso, de ser necesario es posible explicar cual queremos que sea el comportamiento de la concatenación, utilizando el mecanismo de meta-tablas. La tercer regla de la Figura 4.7 muestra cómo

45: Un intérprete Lua solo implementa $<$ y $\leq$; luego, $>$ y $\geq$ se explican en términos de los primeros, mientras que el orden de evaluación de los operandos es consistente en ambos casos, en todas las implementaciones de Lua: de izquierda a derecha. Los formalizamos mediante la evaluación de izquierda a derecha de una expresión que incluye estos últimos operandos, junto con un paso de reescritura a la correspondiente expresión en términos de $<$ y $\leq$

46: Detalles en `www.lua.org/manual/5.2/manual.html#3.1`.

```
> = 1 + "0xA"
11
> = 1 + "0xAp2" --10 * 2 ^ 2
41
> = "0xA" + "0xA"
20
```

Figura 4.5: Coerción de strings a números en base 10.

$$
\begin{aligned}
\mathit{label} &::= \ldots \mid \textsc{OrdCompWO} \mid \textsc{StrConcatWO} \\
&\quad \mid \textsc{ArithWrongOps} \\
e &::= \ldots \mid (\!| e |\!)_{\mathit{label}}
\end{aligned}
$$

Figura 4.6: Extensiones a la gramática, para expresiones .

$$
\frac{\mathit{op} \in \{\mathbf{..}, <, \leq\} \quad v_1, v_2 \in \mathit{string}}{v_1 \ \mathit{op} \ v_2 \ \rightarrow^{s/e} \ \delta(\mathit{op}, v_1, v_2)}
$$

$$
\frac{v_1 \notin \mathit{string} \vee v_2 \notin \mathit{string} \quad i \in \{1,2\},\ v_i \in \mathit{number} \cup \mathit{string} \quad v_i = \delta(\mathbf{tostring}, v_i)}{v_1 \ \mathbf{..} \ v_2 \ \rightarrow^{s/e} \ \delta(\mathbf{..}, v_1, v_2)}
$$

$$
\frac{\neg(v_1, v_2 \in \mathit{number} \cup \mathit{string})}{v_1 \ \mathbf{..} \ v_2 \ \rightarrow^{s/e} \ (\!| v_1 \ \mathbf{..} \ v_2 |\!)_{\textsc{StrConcatWO}}}
$$

$$
\frac{\mathit{op} \in \{<, \leq\} \quad v_1 \in t_1 \quad v_2 \in t_2 \quad t_1 \neq t_2 \vee t_1 \notin \mathit{string} \cup \mathit{number}}{v_1 \ \mathit{op} \ v_2 \ \rightarrow^{s/e} \ (\!| v_1 \ \mathit{op} \ v_2 |\!)_{\textsc{OrdCompWO}}}
$$

Figura 4.7: Semántica de manipulación y comparación de strings.

se deriva el tratamiento de tales expresiones hacia el mecanismo de meta-tablas.

Finalmente, es posible definir relaciones de orden arbitrarias, para valores que no sean de tipo string ni numérico. Reflejamos eso con la última regla de la Figura 4.7.

Abstraemos en $\delta(\texttt{==})$ los correspondientes detalles de la comparación de igualdad. Valores de tipo diferente no están relacionados mediante **==** (con una excepción, explicada a continuación). Números y strings son comparados por valor, mientras que tablas y clausuras (dentro de los tipos incluidos en nuestro modelo) son comparados por referencia. Hay más detalles que son necesarios introducir[47], para definir la comparación de clausuras, los cuales los presentaremos en §4.2.3.

47: Relacionados con la implementación de la optimización en la creación de clausuras, admitida para un intérprete de Lua 5.2, como se explico en §1

Dejamos sin especificar la manera en la que δ interpreta **==**, pues esto va a involucrar detalles concretos de representación interna de valores, los cuales pueden variar de acuerdo a cómo se compiló el intérprete Lua que se esté utilizando. Por ejemplo, en la figura Figura 4.8 se muestra la interacción con un intérprete Lua oficial estándar, es decir, compilado para que utilice números de punto flotante, siguiendo el estándar IEEE 754. Lo que se observa es que, según el estándar, las expresiones 0 y -0 tienen representaciones internas distintas, producen resultados distintos cuando, por ejemplo, intentamos dividir con ellos como dividendos; sin embargo, la relación **==** los considera equivalentes. Todo esto es, de todos modos, comportamiento que respeta el estándar IEEE 754, pero que no tiene sentido incluir en nuestra formalización, salvo a nivel de mecanización, con motivos de comparación con el intérprete oficial.

```
Lua 5.2.4  Copyright (C)
       1994-2015 Lua.org,
       PUC-Rio
> a,b = 0, -0
> print(a,b)
0   -0
> = 1/a
inf
> = 1/b
-inf
> = a == b
true
```

Figura 4.8: Representación IEEE 754 de números de punto flotante.

$$\frac{\delta(\texttt{==}, v_1, v_2) == \textbf{true}}{v_1 \texttt{ == } v_2 \;\rightarrow^{s/e}\; \textbf{true}}$$

$$\frac{\delta(\texttt{==}, v_1, v_2) == \textbf{false} \qquad \delta(\texttt{type}, v_1), \delta(\texttt{type}, v_2) = \text{“}table\text{”}}{v_1 \texttt{ == } v_2 \;\rightarrow^{s/e}\; (\!|\, v_1 \texttt{ == } v_2 \,|\!)_{\text{EQFAIL}}}$$

$$\frac{\delta(\texttt{==}, v_1, v_2) == \textbf{false} \qquad \delta(\texttt{type}, v_1), \delta(\texttt{type}, v_2) \neq \text{“}table\text{”}}{v_1 \texttt{ == } v_2 \;\rightarrow^{s/e}\; \textbf{false}}$$

Figura 4.9: Semántica de comparación de igualdad.

Las reglas para embeber la interpretación del operador **==** dentro de la ejecución, son mostradas en la Figura 4.9. Notar que, en caso de que la interpretación de la igualdad retorne **false**, y los operandos sean tablas (cuyo tipo determinamos con el servicio de librería type), derivamos el tratamiento de la comparación hacia el mecanismo de meta-tablas, mediante el etiquetado de la expresión en cuestión. Como veremos en detalle en §4.2.5, este mecanismo buscará, en lugares predefinidos del programa, funciones que implementen qué significa comparar los valores involucrados en la expresión etiquetada (que, para la semántica de Lua 5.2, solamente pueden ser valores de tipo tabla o *userdata*[48]). De este modo, se ofrece la posibilidad de definir relaciones arbitrarias sobre valores distintos, por sobre la semántica del operador de comparación de igualdad.

Finalmente, la comparación de igualdad de valores distintos, que no sean de tipo tabla, simplemente reduce a **false**.

48: Tipo de dato introducido en Lua para permitir almacenar valores C arbitrarios, en variables Lua; naturalmente, es un concepto útil para el desarrollo de programas Lua que corran embebidos en un programa anfitrión escrito en C, específicamente. Para este primer esfuerzo de formalización dejamos de lado su tratamiento.

Contextos de evaluación

En §4.1.1 habíamos mencionado la categoría de contextos que denotábamos con E_{lf}. La definición de los mismos la presentamos en la Figura 4.10. Esencialmente, son contextos semejantes a los ya presentados en Figura 3.11, con la adición de los operadores binarios desarrollados en esta sección. Notar que estos contextos no se refieren a sentencias

etiquetadas: efectivamente ayudan a formalizar el concepto de punto etiquetado *más interno* del cuerpo del bucle **while**, hacia el cual debe saltar la sentencia **break**. Patrones semejantes utilizaremos para precisar mecanismos de manejo de errores, en §4.2.6.

Con lo que respecta a la ejecución de programas completos, redefinimos a la categoría de contextos de evaluación E, como una extensión simple de E_{lf}, incluyendo términos etiquetados. Con extender a la categoría de contextos de evaluación como E_{lf}, para crear un nuevo contexto de evaluación E, nos referimos a renombrar toda ocurrencia de E_{lf} con E en las producciones del primero, agregando contextos de la forma $\langle\!| E |\!\rangle_{label}$.

$$
\begin{aligned}
E_{lf} ::= \;& [\,] \mid \textbf{if } E_{lf} \textbf{ then } s \textbf{ else } s \\
\mid\; & \textbf{local } x \textbf{ = } E_{lf} \textbf{ in } s \\
\mid\; & x \textbf{ = } E_{lf} \mid E_{lf}\ binop\ e \\
\mid\; & unop\ E_{lf} \mid E_{lf}\ s \\
\mid\; & E_{lf}\ strictbinop\ e \\
\mid\; & v\ strictbinop\ E_{lf}
\end{aligned}
$$

$$E ::= \text{extensión de } E_{lf} \text{ con } \langle\!| E |\!\rangle_{label}$$

Figura 4.10: Contextos de evaluación libres de sentencias etiquetadas.

Notar que ninguna de las categorías de contextos de evaluación presentadas mencionan bucles **while**. Efectivamente, al explicar un bucle en términos de la semántica de los condicionales, mediante la regla expresada en la Figura 4.2, no hay posiciones, dentro del bucle, a señalar como candidatas a ejecución.

4.2. Fragmento Imperativo

Extendemos ahora el modelo presentado en §3, incluyendo definiciones múltiples de variables locales y asignación, e introducimos un nuevos tipos de dato, definidos en términos de la noción de estado: *tablas* (el único tipo de dato estructurado presente en Lua) y funciones.

4.2.1. Tablas

Las *tablas* en Lua son arreglos asociativos (o diccionarios) mutables. Se trata de estructuras de datos en donde podemos almacenar cualquier valor Lua (excepto **nil**), en la forma de pares clave/valor. A su vez, cada tabla se encuentra unívocamente identificada mediante una identidad o referencia a la tabla. En Lua se utiliza esa identidad para referirse a la tabla en operaciones tales como pasar una tabla como parámetro a un función, retornar una tabla desde una función, comparar tablas o asignarlas a una variable. En la Figura 4.11 se muestra la sintaxis para constructores de tablas, e introduce campos de tabla como variables (*var*) sobre las cuales podremos definir la operación de asignación, la cual precisaremos más adelante. Notar que los campos de un constructor de tabla pueden o no indicar la clave. Cuando no está presente, la semántica del constructor indica que se debe computar una clave numérica, considerando números naturales consecutivos, comenzando desde el 1.

$$
\begin{aligned}
var &::= e \textbf{ [} e \textbf{]} \\
e &::= \ldots \mid \textbf{\{ } field \textbf{, } \ldots \textbf{ \}} \\
field &::= e \mid \textbf{[} e \textbf{] = } e
\end{aligned}
$$

Figura 4.11: Constructores de tablas.

A continuación, agregaremos tablas a nuestro modelo y definiremos la semántica de sus operaciones asociadas.

Incluyendo tablas en el modelo

Incluiremos tablas mutables directamente en nuestro modelo. Es decir, la operación de asignación sobre campos de tabla va a estar definida directamente en el lenguaje. Esto implica que, desde el punto de vista del lenguaje, la información en las tablas puede ser alterada.

Un enfoque distinto sería el de tener tablas funcionales, es decir, tablas para las cuales no existe una operación que modifique su contenido, sino que, para poder expresar cambios en el contenido de una tabla, nos tenemos que valer de actualización funcional: simplemente construimos una nueva tabla que contiene la modificación deseada, y mapeamos esta a la identidad original de la tabla. Es importante remarcar que esta discusión se refiere a semántica de tablas desde el punto de vista del lenguaje. La semántica operacional que precisaremos para la operación de asignación en campo de tabla se va a asemejar a lo descripto.

[11]: Guha y col. (2010), «The Essence of JavaScript»

A modo de ejemplo, consideremos cómo se debe realizar, en λ_{JS}[11] , una operación de alteración de un atributo a_i de un objeto referenciado por o, asignando un nuevo valor v. Recordemos que en §3.2 describimos cómo en λ_{JS} se introduce estado como concepto de programación opcional, a utilizarse allí donde se requiera describir cómputos imperativos. A su vez, los objetos son descriptos en términos de registros funcionales, con lo cual, no disponemos de una operación que exprese directamente asignación a un atributo a_i de un objeto o, por ejemplo:

$$o[a_i] = v$$

que tenga una semántica descripta en términos de modificación de estado. De lo que sí disponemos es de una operación de actualización funcional de atributos, que está definida para los registros[49] . De este modo, si el registro asociado a o es $\{[a_1] = v_1, ...\}$, disponemos de una operación de actualización funcional sobre ese registro:

49: No sobre los objetos: entidad compuesta por los atributos, los métodos, más una *identidad*, que, en nuestro ejemplo, sería su referencia o.

$$\{[a_1] = v_1, ...\}[a_i] = v$$

la cual resulta en un nuevo registro $\{[a'_1] = v'_1, ...\}$, que solo puede diferir del anterior en el valor del atributo a_i, que ahora va a ser v. Finalmente, para modelar el efecto completo de alteración de un atributo de objeto, tenemos que hacer uso de una primitiva, **deref**, para desreferenciar explícitamente el identificador o del objeto[50] , para acceder al registro asociado, modificarlo, y guardar la actualización asociándola al mismo identificador o. Obtendríamos:

50: Pues de ese modo está implementada la noción *de estado como concepto de programación*, en λ_{JS}

$$o := ((\textbf{deref}\ o)[a_i] = v)$$

donde utilizamos := para denotar asignación de variable. Comparemos esta manera de expresar asignación de atributos, con la propuesta inicial, utilizando directamente objetos mutables:

$$o[a_i] = v$$

Como ya se mencionó, en trabajos posteriores a λ_{JS}, como [13] , se consideró que la inclusión de registros funcionales, y referencias como mecanismo opcional de programación, no arroja nueva luz sobre los conceptos modelados, mientras que sí resulta en un código compilado más verboso.

[13]: Politz y col. (2012), «A tested semantics for getters, setters, and eval in JavaScript»

En nuestro trabajo, aprovechando experiencias previas, incluiremos tablas mutables directamente. A su vez, necesitaremos incluir un nuevo tipo de referencia que represente a un identificador de tabla, desde la cual manipularla. La mutabilidad de las tablas la modelaremos con un nuevo mapeo, denotado con θ, entre identificadores de tablas (que representaremos con el no-terminal *tid*, en Figura 4.12) y las tablas. Los elementos del dominio de θ serán considerados valores y, acordemente, podrán ser almacenados en σ (es decir, $tid \in v$). Exigiremos que los elementos de *tid* satisfagan las mismas propiedades que para los elementos de $\mathsf{dom}(\sigma)$, con la condición extra de que $tid \cap r = \emptyset$, donde r es el conjunto de referencias a σ. Esta última condición la imponemos a los fines de preservar inalterada la operación de desreferenciado implícito de referencias a σ, explicada en §3.2. La imagen de θ contendrá tablas, junto con cualquier meta-dato relativo a tablas (a introducir en secciones posteriores), necesario para describir la semántica de las mismas. Finalmente, asumiremos que siempre es posible obtener una referencia fresca a θ, denotando con $(tid, t), \theta$ al almacenamiento resultado de extender θ con un nuevo par, (tid, t), para una $tid \notin \mathsf{dom}(\theta)$.

$$v ::= \ldots \mid tid$$

$$label ::= \ldots \mid \textsc{Index} \mid \textsc{WrongKey} \mid \textsc{NonTable}$$

Figura 4.12: Construcciones de tiempo de ejecución relativas a tablas.

Constructores de tablas

La Figura 4.13 define la semántica de los constructores de tablas. Indica que las tablas serán almacenadas únicamente cuando sus campos (ya sea que contengan claves explícitas o no) hayan sido completamente evaluados: $field_i \in v$ se refiere a un campo en donde solo se ha especificado su valor, y ya ha sido completamente evaluado; $field_i = \mathbf{[}\ v_1\ \mathbf{]} = v_2$ se refiere a un campo en donde se han especificado clave y valor, evaluando a ciertos valores v_1 y v_2, respectivamente.

$$\frac{\forall\ 1 \leq i, field_i \in v \vee field_i = \mathbf{[}\ v_1\ \mathbf{]} = v_2 \quad t = (addkeys(\{field_1,\ ...\})\ ,\ \mathbf{nil}\ ) \quad \theta_2 = (tid,\ t),\ \theta_1}{\theta_1\ :\ \{field_1,\ ...\}\ \rightarrow^{\theta}\ \theta_2\ :\ tid}$$

Figura 4.13: Semántica de constructores de tablas.

La semántica completa del constructor de tablas depende de una meta-función, *addkeys*, que agrega las posibles claves ausentes en el constructor (para campos de la forma e, mostrados en la Figura 4.11). La meta-función provee, como claves, números naturales consecutivos, comenzando desde 1.

La razón por la cual las claves comienzan en 1 reside en que Lua heredó cuestiones de sintaxis y diseño de un lenguaje de descripción de datos, con facilidades de programación, denominado SOL [24] , el cual estaba orientado a personas sin una fuerte base en programación.

[24]: Ierusalimschy y col. (2007), «"The evolution of Lua"»

Dejamos *addkeys* sin especificar a nivel de esta formalización, debiendo proveerse alguna especificación a los fines de mecanización. Esto se debe a que los detalles concretos respecto a cómo opera *addkeys* no forman parte del manual de referencia[51], pudiendo observarse diferencias entre las principales implementaciones de Lua 5.2: el intérprete oficial y LuaJIT 2.0[52]. En particular, el manual de referencia no explica

51: Ver `www.lua.org/manual/5.2/manual.html#3.4.8`

52: Si bien LuaJIT 2.0 no es completamente compatible con Lua 5.2, tablas son un concepto ya existente en versiones anteriores de Lua, soportado de manera completa en LuaJIT, salvo por el comportamiento de algunos iteradores de tablas con nueva semántica en Lua 5.2. Ver `luajit.org/extensions.html`.

```
Lua 5.2.4  Copyright (C)
      1994-2015 Lua.org, PUC-
      Rio
> local t = {1, [1] = 2};
  print(t[1])
1

LuaJIT 2.1.0-beta3 Copyright (C)
      2005-2017 Mike Pall. http://
      luajit.org/
> local t = {1, [1] = 2};
  print(t[1])
2
```

Figura 4.14: Diferentes implementaciones de *addkeys*.

$$\frac{\delta(rawget, tid, v_1, \theta_1) \neq \textbf{nil} \quad \theta_2, tid = \delta(rawset, tid, v_1, v_2, \theta_1)}{\theta_1 \ : \ tid\ [v_1] = v_2 \ \rightarrow^{\theta} \ \theta_2 \ : \ ;}$$

$$\frac{s = \ tid\ [v_1] = v_2 \quad \delta(rawget, tid, v_1, \theta) = \textbf{nil}}{\theta \ : \ s \ \rightarrow^{\theta} \ \theta \ : \ (\!| \ s \ |\!)_{\text{WRONGKEY}}}$$

$$\frac{s = \ v_1\ [v_2] = v_3 \quad \delta(type, v_1) \neq \text{“}table\text{”}}{\theta \ : \ s \ \rightarrow^{\theta} \ \theta \ : \ (\!| \ s \ |\!)_{\text{NONTABLE}}}$$

Figura 4.15: Semántica de asignación de campos.

53: Esto significa que el mecanismo de meta-tablas, a ser introducido en §4.2.5, no está involucrado en la semántica de la operación; en el caso de la operación de indexado provista por el lenguaje, su semántica es más compleja, requiriendo de la intervención de este servicio para ser explicada.

cual sería el comportamiento esperado en casos como los mostrados en la Figura 4.14, en donde interpretamos el mismo programa, primero con el intérprete oficial, y luego con LuaJIT 2.0. Notemos que, luego de la operación de *addkeys*, el campo de clave igual a 1 difiere en cada caso.

Retornando a la Figura 4.13, asumimos que siempre es posible obtener una referencia fresca *tid* a θ_1, como se mencionó previamente. La regla explica que θ_2 extiende a θ_1 con un nuevo par, (tid, t), donde $t = (addkeys(\{field_1, \ ...\}) \ ,$ **nil**). Es decir, se trata de la tabla evaluada, habiendo agregado las correspondientes claves faltantes, la cual almacenamos en θ_2 manteniéndola asociada con un nuevo valor, **nil**. Este último valor va a representar a la meta-tabla asociada a la tabla recién creada, tópico que desarrollaremos en §4.2.5. Por el momento, basta con entender que, para una tabla recién creada, su meta-tabla asociada es siempre igual a **nil**.

Operaciones sobre tablas

A los fines de formalizar la semántica de operaciones sobre tablas introducimos una nueva relación, $\rightarrow^{\theta} \subseteq \theta \times t$, con t denotando al conjunto de sentencias y expresiones de nuestro lenguaje.

La Figura 4.15 presenta la semántica de asignación de campo de tabla. Utiliza algunas primitivas descriptas mediante la función δ. Los motivos detrás de la forma en la que δ captura los servicios que describimos a continuación, serán introducidos en §4.2.4: $\delta(type)$ es el servicio que ofrece Lua para determinar tipos en tiempo de ejecución; $\delta(rawget, tid, k, \theta)$ es la primitiva para realizar indexado de tablas[53], la cual retorna el valor asociado con la clave k en la tabla tid, o **nil**, en caso de no existir tal clave; $\delta(rawset, tid, k, v, \theta_1)$ es la primitiva para actualización de valores de campos de tablas, la cual devuelve un nuevo almacenamiento θ_2, que solo difiere de θ_1 en que el valor asociado a k, en la tabla tid, es v.

Retornando a la Figura 4.15, la primer regla describe la operación de asignación bajo condiciones normales (clave existente y operación realizada sobre una tabla), mientras que las 2 últimas reglas delegan el tratamiento de la sentencia al mecanismo de meta-tablas: el primer caso, para cuando la operación de indexado se hace con una clave no existente (en cuyo caso, se etiqueta la sentencia con WRONGKEY), o para cuando la operación se realiza sobre un valor que no es de tipo tabla, operación que puede tener sentido, dependiendo de si hemos utilizado meta-tablas para definir el mismo. En este último caso, etiquetamos con NONTABLE para informar al mecanismo de meta-tablas.

Como puede observarse, la asignación de campos de tabla tiene una semántica más compleja que lo expresado *rawset*, requiriendo de esta última (y de *rawget*) para expresar la semántica de la primera. Respecto a la utilización de *rawget* para consultar por la pertenencia de una clave a una tabla, en las primeras 2 reglas de Figura 4.15, cabe mencionar que el significado de esta noción involucra detalles que ya están contemplados en la semántica de *rawget*. En particular, la comparación de claves

$$\delta(rawset, tid, v_1, v_2, \theta_1) = (\theta_2, tid),$$
$$where \begin{cases} v_1 \notin \{\textbf{nil}, \textbf{nan}\} \\ \theta_1(tid) = (\{..., field_{i-1}, [v'_1] = v_3, field_{i+1}, ...\}, ...) \\ \delta(==, v_1, v'_1) = \textbf{true} \\ \theta_2 = \begin{cases} \theta_1[tid := (\{..., field_{i-1}, field_{i+1}, ...\}, ...)] & if \quad v_2 = \textbf{nil} \\ \theta_1[tid := (\{..., field_{i-1}, [v'_1] = v_2, field_{i+1}, ...\}, ...)] & if \quad v_2 \neq \textbf{nil} \end{cases} \end{cases}$$

$$\delta(rawset, tid, v, v', \theta_1) = (\theta_2, tid),$$
$$where \begin{cases} v \notin \{\textbf{nil}, \textbf{nan}\} \\ \theta_1(tid) = (\{[v_1] = v'_1, ..., [v_n] = v'_n\}, ...) \\ \forall\ k\ \in\ \{v_1, ..., v_n\}, \delta(==, v, k) = \textbf{false} \\ \theta_2 = \begin{cases} \theta_1 & if \quad v' = \textbf{nil} \\ \theta_1[tid := (\{[v] = v', [v_1] = v'_1, [v_n] = v'_n\}, ...)] & if \quad v' \neq \textbf{nil} \end{cases} \end{cases}$$

$$\delta(rawset, v_1, v_2, v_3, \theta) = (\theta, \delta(error, ...))\ ,\ if\ \delta(type, v_1) \neq \text{“}table\text{”} \vee v_2 \in \{\textbf{nil},\textbf{nan}\}$$

$$\delta(rawget, tid, v_i, \theta) = \begin{cases} v_j & if\ \theta(tid) = (\{..., [v'_i] = v_j, ...\}, ...) \\ & and\ \delta(==, v_i, v'_i) = \textbf{true} \\ \textbf{nil} & otherwise \end{cases}$$
$$if\ \ \delta(type, tid) = \text{“}\textbf{table}\text{”}$$

$$\delta(rawget, v_1, v_2, \theta) = \delta(error, ...)\ , if\ \delta(type, v_1) \neq \text{“}\textbf{table}\text{”}$$

Figura 4.16: Función δ: primitivas para asignación e indexado de tablas.

se realiza utilizando lo expresado por δ(**==**), como puede observarse en la Figura 4.17, en donde ejecutamos el código mostrado con un intérprete que use números flotantes IEEE 745. El ejemplo versa sobre parte de los expresado respecto a este estándar, sobre representación y comparación de números flotantes, que ya describimos en §4.1. En la Figura 4.17 vemos que, inclusive a pesar de que indexamos la tabla con claves con una representación interna distinta (exigido por el estándar para 0 y -0), accedemos al mismo campo, pues $\delta(\textbf{==}, 0, -0) = \textbf{true}$ para una función δ(**==**) que respete el estándar.

```
local t,z,nz = {}, 0, -0
t[z] = true
print(t[z]) -- true
print(t[nz]) -- true
```

Figura 4.17: La pertenencia de una clave a una tabla depende de lo expresado por δ(**==**).

La semántica de *rawset* y *rawget* se presenta en la Figura 4.16[54] : **nil** y **nan** no pueden ser claves de una tabla; comparación de claves se delega a la interpretación de **==**; asignación de un campo de tabla puede resultar en la eliminación de un campo de tabla si el nuevo valor que se pretende asignar es **nil** (no así si el valor es **nan**); es posible agregar un nuevo campo mediante la operación de asignación, utilizando una clave no existente; finalmente, indexado con una clave no existente nunca resulta en un error, simplemente es equivalente a **nil**. Las primitivas son equivalentes a un error (concepto a ser introducido en §4.2.6), si intentamos utilizar las mismas sobre un valor que no es de tipo tabla, o si intentamos utilizar $\delta(rawset)$ para asignar a un campo con **nil** o **nan** como clave.

En esencia, $\delta(rawset)$ describe lo fundamental de la operación de asignación de tablas, mientras que la sentencia misma de asignación de tablas, provista por el lenguaje, extiende esa semántica con detalles de manejo de meta-tablas. De ese modo, capturamos lo que comprende respecto de la operación de tal sentencia, la persona que programa con Lua. Veremos que sucede una situación semejante con respecto a la expresión para indexado de tablas, con respecto a $\delta(rawget)$.

54: La notación usada para describir los campos de tablas en la Figura 4.16 no debe ser interpretada considerando la presencia de algún orden entre los campos de tablas. En efecto, la definición de una tabla no incluye una noción de *orden* entre sus campos. Las tablas pueden ser pensadas como funciones parciales con signatura

$$v \setminus \{\textbf{nil}, \textbf{nan}\} \rightarrow v \setminus \{\textbf{nil}\}$$

(donde **nan**, *not a number*, es un valor numérico especial usado para indicar resultados numéricos no definidos o no representables, como $0/0$).

$$\frac{v_2 = \delta(rawget,\ tid,\ v_1,\ \theta) \qquad v_2 \neq \textbf{nil}}{\theta : tid\ [\ v_1\ ] \rightarrow^{\theta} \theta : v_2}$$

$$\frac{e = tid\ [\ v\ ] \qquad \delta(rawget,\ tid,\ v,\ \theta) = \textbf{nil}}{\theta : e \rightarrow^{"} \theta : (\!|\ e\ |\!)_{\text{WRONGKEY}}}$$

$$\frac{e = v_1\ [\ v_2\ ] \qquad \delta(type, v_1) \neq \text{“}table\text{”}}{\theta : e \rightarrow^{\theta} \theta : (\!|\ e\ |\!)_{\text{NONTABLE}}}$$

Figura 4.18: Semántica de indexado de tablas.

```
var ::= ... | x

s   ::= ... | var , ... = e , ...
        | local x , ... = e , ... in s
E   ::= ...
        | local x , ... = v , ... , E , e , ...
          in s
        | evar , ... , E [ e ] , ... = e , ...
        | evar , ... , v [ E ] , ... = e , ...
        | evar , ... = v , ... , E , ...
x   ::= identificadores de variable
```

Figura 4.19: Sintaxis de definición y asignación de múltiples variables.

```
evar ::= r | v [ v ]
```

Figura 4.20: lvalues sobre los que podemos definir asignación.

55: Esto, asumiendo que se trata efectivamente de una operación sobre un campo de tabla. Es posible también definir, para valores de otro tipo, la operación de *asignación en un campo* utilizando metatablas. En tal caso, utilizaríamos la misma sintaxis para referirnos a tal operación. De este modo, en palabras de los autores del lenguaje[2]: “*Lua puede ser extendido para ser utilizado en distintos dominios, creando, de este modo, lenguajes de programación especializados pero que comparten la misma sintaxis*”.

Indexado de tablas se presenta en la Figura 4.18. Su semántica contempla situaciones a análogas a las tratadas en el caso de asignación de campos de tablas: indexado a una tabla utilizando una clave existente; indexado a una tabla con una clave no existente; operación de indexado sobre un valor que no es de tipo tabla. El significado de estas 2 últimas operaciones requiere de la intervención del mecanismo de meta-tablas y, por lo tanto, etiquetamos las expresiones con la información que averiguamos, para que la utilice el citado mecanismo.

4.2.2. Variables locales

En Lua es posible definir y asignar, de manera simultanea, múltiples variables. Para incluir esta facilidad en nuestro modelo realizaremos un pequeño cambio en la sintaxis ya presentada para estas operaciones, junto con los correspondientes cambios en la definición de los contextos de evaluación. Presentamos estos cambios en la Figura 4.19.

Sintácticamente, comenzamos incluyendo variables locales como un nuevo tipo de variable (en la categoría sintáctica *var*), junto con los campos de tablas. Para facilitar la definición de la operación de asignación sobre estas variables, como también definir los correspondientes contextos de evaluación, nos será útil la categoría *evar*, presentada en la Figura 4.20. Representa a las variables locales y a los campos de tablas, respectivamente, cuando ocurren en el lado izquierdo del símbolo **=** (*lvalues*) en una sentencia de asignación. A su vez, se trata de una representación de las variables sobre las que podemos efectivamente definir la operación de asignación: en el primer caso, el identificador de variable local ha sido reemplazado por su referencia a σ (como detallaremos más adelante); en el segundo caso, el campo de tabla ha sido completamente identificado (ya se evaluó la expresión que representa la clave, un valor, y tenemos el identificador de la tabla, otro valor[55]).

Retornando a la Figura 4.19, con respecto a la asignación de múltiples variables (locales o campos de tablas), y a la definición de múltiples variables locales, antes de poder efectuarlas es necesario garantizar que la cantidad de lvalues coincide con la de valores asignados (*rvalues*): primero, los contextos de evaluación definidos imponen evaluación de izquierda a derecha, en las listas de lvalues y rvalues; luego de ello, en caso de haber diferencias en las longitudes de ambas listas, se descartan rvalues o se completa con valores **nil**. Esto se presenta en las primeras 2 reglas de la Figura 4.21 (solo presentamos las reglas correspondientes a la asignación múltiple; para definición de múltiples variables locales se aplican reglas análogas). Notar que para explicar estas reglas no necesitamos mencionar el almacenamiento. Las definimos, por lo tanto, mediante $\rightarrow^{s/e}$.

Finalmente, precisamos distinguir entre las operaciones de asignación sobre una variable local y asignación sobre campo de tabla, por la diferencia entre sus semánticas. A tal fin, una vez que se han igualado las longitudes de los lvalues y rvalues, descomponemos una asignación múltiple en una secuencia de asignaciones simples, como se muestra

$$\frac{k \geq 1 \qquad v_{n-k+1} = \textbf{nil}, ..., v_n = \textbf{nil}}{evar_1,...,evar_n = v_1,...,v_{n-k} \;\rightarrow^{s/e}\; evar_1,...,evar_n = v_1,...,v_{n-k},v_{n-k+1},...,v_n}$$

$$\frac{k \geq 1}{evar_1,...,evar_n = v_1,...,v_n,...,v_{n+k} \;\rightarrow^{s/e}\; evar_1,...,evar_n = v_1,...,v_n}$$

$$\frac{n \geq 2}{evar_1,...,evar_n = v_1,...,v_n \;\rightarrow^{s/e}\; evar_n = evar_n \;;\; evar_1,...,evar_{n-1} = v_1,...,v_{n-1}}$$

Figura 4.21: Reglas para igualar la cantidad de rvalues y lvalues.

en la tercer regla de la Figura 4.21. El orden en el que efectivamente se realizan las asignaciones individuales no forma parte de la especificación de Lua. En nuestro caso simplemente seguimos al intérprete oficial, que asigna de derecha a izquierda[56].

La introducción de variables locales y la operación de asignación sobre las mismas requiere, para su definición, de referirnos al almacenamiento de valores, σ, y al entorno. Precisamos su semántica con $\rightarrow^{\sigma}$, en la Figura 4.23.

La primer regla formaliza la definición de variables locales: una vez que se evaluaron sus valores iniciales, y se igualaron la longitud de la lista de lvalues con la de los rvalues, se procede a obtener variables frescas al almacenamiento de valores, hacerlas corresponder con los valores iniciales de cada variable, y modificar el entorno, agregando los correspondientes mapeos entre los identificadores de variables y sus referencias. Este último paso lo realizamos mediante una función de substitución, como se explicó en §3.1, que introducirá, simultáneamente, las referencias al almacenamiento $r_1,...,r_n$ en el alcance s de la declaración de variables locales $x_1,...,x_n$. Expresamos esto mediante $s[x_1 \backslash r_1, ..., x_n \backslash r_n]$, en la primer regla de la Figura 4.23.

Finalmente, la asignación de una variable local tiene una semántica sencilla, explicada con la segunda regla de Figura 4.23. Denotamos con $\sigma[r := v]$ la correspondiente alteración del valor $\sigma(r)$, operación definida siempre que $r \in \mathsf{dom}(\sigma)$. En particular, dado que las referencias son objetos existentes en tiempo de ejecución, creados por nuestra semántica operacional, esta última condición está siempre garantizada para la ejecución de un programa *bien formado* dado, concepto que precisaremos en §4.3.

56: En este caso se trata de un aspecto que no tiene un efecto observable desde un programa Lua: se trata de una operación atómica. De todos modos, por completitud, mostramos el método utilizado, sugerido en `lua-users.org/lists/lua-l/2001-02/msg00127.html`, para obtener información que ayude a precisar aspectos de Lua que escapan al manual de referencia. La propuesta consiste en observar el *bytecode* [25] correspondiente a un programa Lua dado. En la Figura 4.22 mostramos la secuencia de instrucciones en bytecode de la VM de Lua 5.2 correspondiente al programa: `a,b = 1,2`. Se obtuvo compilando el programa con el compilador oficial `luac`. Simplemente observar cómo, las operaciones de asignación (SETTABUP), son realizadas de derecha a izquierda, primero asignando sobre _ENV ['b'] (es decir, la variable global b), y luego sobre _ENV ['a'].

```
1  [1]  LOADK 0 -3
                ; 1
2  [1]  SETTABUP 0 -2 -4
                ; _ENV "b" 2
3  [1]  SETTABUP 0 -1 0
                ; _ENV "a"
4  [1]  RETURN  0 1
```

Figura 4.22: Bytecode del programa `a,b = 1,2`.

4.2.3. Funciones

La Figura 4.24 muestra la sintaxis para la definición de funciones, llamada a una función y un método[57] (notar que pueden ser tanto expresiones como sentencias), *expresiones parentizadas* (una operación útil para la manipulación de los, posiblemente, múltiples valores que podría retornar o recibir una) y la sentencia **return**, para retornar desde una función o para expresar lo retornado por el programa, a ser recibido por el programa anfitrión en donde se ejecuta el programa Lua.

$$\frac{\sigma' = (r_1, v_1), ..., (r_n, v_n), \sigma}{\sigma \,:\, \textbf{local}\ x_1,...,x_n = v_1,...,v_n\ \textbf{in}\ s \;\rightarrow^{\sigma}\; \sigma' \,:\, s[x_1 \backslash r_1, ..., x_n \backslash r_n]}$$

$$\frac{\sigma' = \sigma[r := v]}{\sigma \,:\, r = v \;\rightarrow^{\sigma}\; \sigma' \,:\, ;}$$

Figura 4.23: Semántica de definición y asignación de variables locales.

57: Forma parte del soporte brindado en Lua para la POO.

La segunda producción para definición de funciones, en Figura 4.24, se refiere a las funciones *vararg*, las cuales pueden recibir una cantidad arbitraria de parámetros actuales. Luego de los parámetros formales *nombrados* (aquellos que especificamos mediante identificadores de *x*), los parámetros extra recibidos son colocados en una *tupla* y serán referidos, en el cuerpo de la función, mediante el identificador especial **...** (no confundir con elipsis matemática, también utilizada para describir de manera sucinta a las producciones).

v ::= ... | **function** l (x , ...) s
| **function** l (x , ... , **...**) s

s ::= ... | **$statFcall** e (e , ...)
| **$statFcall** e **:** x (e , ...)
| **return** e, ...

e ::= ... | **(** e **)** | e(e, ...)
| e**:** x (e , ...)

Figura 4.24: Sintaxis de definición y aplicación de funciones.

Finalmente, la definición de una función requiere de la especificación de una etiqueta *l*, que no debe ser confundida con una variable a la que se asigna la función: una función definida siguiendo la sintaxis presentada es anónima, y la etiqueta *l* cumple un rol en la comparación de igualdad de funciones, que presentaremos más adelante, en esta sección.

Notar que la sintaxis de una llamada que ocurre como sentencia, difiera de la de una que ocurre como expresión. Más adelante precisaremos su significado, por el momento basta con entender que se trata de una distinción necesaria para definir la semántica de las operaciones que se realizan sobre los valores retornados por una función.

Representación de clausuras

La creación de clausuras no requiere de un tratamiento especial. No necesitamos tener *c*lausuras explícitas: una definición de función es ella misma la clausura que se requiere, una vez que el foco de evaluación alcanzó a la definición de función, dado que el entorno está embebido en el cuerpo de la función, y este ya contiene las menciones a las variables libres en el cuerpo de la misma.

Lo anterior ocurre a causa de que las ocurrencias ligadas de los identificadores son reemplazadas por las correspondientes referencias, antes de ejecutar la porción del programa que está en el alcance de las ocurrencias ligadoras de tales identificadores. En nuestro modelo, capturaremos esta propiedad como un corolario evidente de la corrección de nuestra definición de substitución, programas *bien formados* y la correspondiente propiedad de progreso de la semántica, en §4.3 (ver [26] , capítulo 13, para otro modelo operacional en donde esta propiedad es un invariante de la ejecución de un programa).

[26]: Peter Van Roy (2004), *Concepts, Techniques and Models of Computer Programming*

Luego, si una función dada aparece en el alcance de una ocurrencia ligadora de un identificador x, la operación de substitución va a embeber en el cuerpo de tal función, a la correspondiente referencia de x.

Llamadas a funciones y métodos

Las llamadas a función no pueden ser pensadas exactamente como contracciones β. En particular, es válido llamar a una función con una cantidad arbitraria de parámetros actuales, independientemente de lo especificado en la signatura de la función, como puede verse en la sesión con el intérprete mostrada en la figura Figura 4.26.

$$\frac{\begin{array}{c} i \leq min(m,n) \Rightarrow v'_i = v_i \\ i > m \Rightarrow v'_i = \textbf{nil} \\ \sigma' = (r_1, v'_1), ..., (r_n, v'_n), \sigma \end{array}}{\sigma : (\textbf{function } l\ (x_1, ..., x_n)\ s)\ (v_1, ..., v_m) \ \rightarrow^{\sigma}\ \sigma' : (\!|\ s\ [x_1 \backslash r_1, ..., x_n \backslash r_n]\ |\!)_{\text{RETEXP}}}$$

$$\frac{\begin{array}{c} i \leq min(m,n) \Rightarrow v'_i = v_i \\ i > m \Rightarrow v'_i = \textbf{nil} \\ tuple = < v_{n+1}, ..., v_m > \\ \sigma' = (r_1, v'_1), ..., (r_n, v'_n), \sigma \end{array}}{\sigma : (\textbf{function } l\ (x_1, ..., x_n, \textbf{...})\ s)(v_1, ..., v_m) \ \rightarrow^{\sigma}\ \sigma' : (\!|\ s\ [x_1 \backslash r_1, ..., x_n \backslash r_n, \textbf{...} \backslash tuple]\ |\!)_{\text{RETEXP}}}$$

Figura 4.25: Llamadas a funciones.

En la Figura 4.25 capturamos el comportamiento anterior, junto con el manejo de funciones *vararg* (que pueden recibir una cantidad variable de parámetros actuales, y referirse a los mismos mediante el identificador especial **...**), y la semántica de aplicación de métodos. Notar que, en todos los casos tenemos llamadas a función por valor.

Solamente mostramos las reglas para llamadas que ocurren como expresiones, siendo análoga a la semántica de llamadas que ocurren como sentencia, con la diferencia de que etiquetamos de otro modo a los términos resultantes, para distinguir ambos casos. En el siguiente apartado explicaremos los motivos detrás de este diseño.

La primer regla de la Figura 4.25 modela llamadas a funciones no vararg. Los parámetros formales son variables imperativas, por lo cual, creamos referencias frescas al almacenamiento, hacia las cuales mapear cada parámetro. A su vez, contemplamos las situaciones en las que la función es llamada con la misma cantidad de argumentos que parámetros formales declarados en su signatura, así como cuando faltan (en cuyo caso, se asigna **nil** a los parámetros faltantes) o sobran (los cuales son descartados). Como es de esperar, la ejecución continua mediante la evaluación del cuerpo de la función, en un entorno que incluya un mapeo identificar-referencia, para cada parámetro formal de la función. Por último, notar que etiquetamos el cuerpo de la función con RETEXP (presentada en la Figura 4.27), para indicar tanto el punto hacia el cual debe retornar una posible sentencia **return** presente en la función, como también para recordar que esta llamada ocurre como expresión. En el siguiente apartado explicaremos el uso que hacemos de esta información.

La segunda regla de la Figura 4.25 trata el caso de llamadas a funciones vararg: la diferencia radica en que los parámetros actuales sobrantes son colocados en una *tupla* (presentadas en la Figura 4.27), y esta es mapeada a una variable funcional especial, denotada con **...**, denominada *expresión vararg*. Las tuplas son construcciones de tiempo de ejecución, que utilizaremos para describir estos argumentos actuales extras en una llamada a función vararg, como también para modelar los valores retornados de una función. Lo haremos de este modo ya que sobre ambas listas de valores se aplican reglas semejantes, cuando ocurren dentro de otras expresiones del lenguaje.

```
> function f(x1, x2)
    return x1, x2
  end
> return f()
nil    nil
> return f(1)
1   nil
> return f(1,2)
1   2
> return f(1,2,3)
1   2
```

Figura 4.26: Llamadas a función.

$$s ::= \ldots \mid (\!|\ s\ |\!)_{\text{RETSTAT}}$$
$$e ::= \ldots \mid < e, \ldots > \mid (\!|\ s\ |\!)_{\text{RETEXP}}$$

Figura 4.27: Construcciones de tiempo de ejecución, para modelar llamadas a función.

La expresión vararg **...** actúa como variable funcional: no es posible asignarle nuevos valores. Simplemente puede ocurrir como expresión, en el cuerpo de una función vararg. Formalmente, trataremos a la ocurrencia de **...** en la signatura de una función como a una ocurrencia ligadora de una variable, cuyo identificador será **...**, y cuyo alcance léxico tiene una peculiaridad. En particular, **...** sólo puede ocurrir dentro una función vararg, en el sentido mostrado por el ejemplo[58] de la Figura 4.28.

58: Tomado de lua-users.org/wiki/VarargTheSecondClassCitizen

```
> function f (...)
    return function()
              return  ...
           end
  end

stdin :1:  cannot use  '...'
      outside a vararg
      function near  '...'
```

Figura 4.28: Alcance de una definición de identificador vararg.

Es decir que la ocurrencia ligadora de **...** tiene un alcance que incluye el cuerpo s de la correspondiente función vararg, exceptuando el cuerpo de cualquier otra definición de función que ocurra en s.

Incluimos en nuestra función de substitución la anterior regla de alcance léxico, que aplicará exclusivamente a **...** . Nuevamente, como se trata de una variable funcional podemos explicarla con un único mapeo: el entorno. Luego, nuestra función de substitución contendrá el valor que representa **...** en cada llamada, que se reduce a la mencionada tupla.

Por cohesión en la presentación, explicaremos también aquí parte del soporte que incluye Lua para POO: llamadas a *métodos*. En la §2 ilustramos sobre cómo es posible implementar en Lua conceptos de POO, usando tablas, clausuras de primera clase y meta-tablas, junto con un soporte lingüístico para facilitar la definición y llamada a métodos. Por motivos que explicaremos a continuación, incluimos directamente en nuestro lenguaje el soporte lingüístico para llamadas a métodos.

$$v\texttt{:}x\ \texttt{(}e_1\texttt{,...,}e_n\texttt{)} \xrightarrow{s/e} v\texttt{[“}x\texttt{”]}\ \texttt{(}v\texttt{,}e_1\texttt{,...,}e_n\texttt{)}$$

Figura 4.29: Llamadas a métodos.

En la Figura 4.29 se explica la invocación de métodos en términos de una operación de indexado de tabla, en donde buscar el método correspondiente, y una llamada a función pasando como primer parámetro el objeto sobre el que se llamó el método.

Por un lado, incluimos esta construcción para preservar la correspondencia directa entre Lua y nuestro lenguaje. Por otro, notar que la regla presentada en la Figura 4.29 evidencia que el objeto v, cuyo método es llamado, es evaluado una única vez. De este modo, queda claro que no podríamos expresar la llamada $e\texttt{:}x\ \texttt{(}e_1\texttt{,...,}e_n\texttt{)}$ en términos de la expresión $e\texttt{[“}x\texttt{”]}\ \texttt{(}e\texttt{,}e_1\texttt{,...,}e_n\texttt{)}$, sino que sería necesario una traducción más compleja, para asegurarnos de evaluar e sólo una vez.

$$\frac{\delta(type, v) \neq \text{“}function\text{”}}{v\ (v_1, ..., v_n) \xrightarrow{s/e} \langle\!|\ v\ (v_1, ..., v_n)\ |\!\rangle_{\text{WFUNCALL}}}$$

Figura 4.30: Llamada a función sobre valores que no son clausuras.

Finalmente, la Figura 4.30 describe la situación excepcional de llamada a función sobre un valor que no es una clausura. En tal caso, etiquetamos la llamada con información sobre la situación encontrada, delegando al mecanismo de meta-tablas la tarea de continuar con la ejecución. Notar que esta es una circunstancia en la que es posible identificar cual es la situación excepcional encontrada tan solo con mirar la estructura sintáctica del término etiquetado: si solo es posible utilizar meta-tablas para definir la semántica de llamadas a función sobre valores que no son clausuras, entonces debe ser que eso mismo ocurre con un término etiquetado que sintácticamente se corresponda con una llamada a función. Sin embargo esta no es una situación general (por ejemplo, en indexado y asignación de campos de tablas, hemos mostrado que hay 2 situaciones excepcionales a contemplar),

y se trata de un mecanismo que se enriquece en sucesivas versiones de Lua, incluyendo nuevas situaciones que pueden ser manejadas con meta-tablas[59] . Por lo tanto, por cohesión, mantendremos el diseño en donde incluimos una etiqueta particular para cada situación excepcional encontrada.

59: Por ejemplo, en Lua 5.3 hay nuevos operadores binarios, con sus correspondientes situaciones excepcionales y semántica extensible mediante meta-tablas

Llamadas a función como sentencias o expresiones

Las llamadas a función pueden ocurrir como sentencias—como en `f() ; print('hi')`—o como expresiones—como en `print(f())`. Sin embargo, no se reduce a una mera distinción sintáctica: una llamada utilizada como expresión puede retornar 0, 1 o más valores, mientras que una llamada que ocurre como sentencia sólo es útil por sus efectos laterales, descartándose cualquier valor que pudiera retornar.

Para escoger entre los recursos de los que disponemos para capturar el comportamiento anterior, nos guiaremos por la búsqueda de una solución económica. En particular, nos desviaremos ligeramente del objetivo de presentar un lenguaje que se corresponda de manera directa con Lua, y consideraremos utilizar una nueva construcción, ya presentada en la Figura 4.24, que represente a las llamadas a función cuando ocurren como sentencia. Distinguir entre sentencias y expresiones será tarea del parser que traduzca programas Lua a programas en nuestro lenguaje, cuando debamos explicar un programa Lua dado con nuestro modelo.

$$\langle\!| E_{\mathsf{If}}[\![\textbf{return } v, \ldots]\!] |\!\rangle_{\textsc{RetStat}} \rightarrow^{s/e} \textbf{;}$$

$$\langle\!| E_{\mathsf{If}}[\![\textbf{return } v, \ldots]\!] |\!\rangle_{\textsc{RetExp}} \rightarrow^{s/e} \textbf{<}v, \ldots\textbf{>}$$

$$\langle\!| \textbf{;} |\!\rangle_{\textsc{RetStat}} \rightarrow^{s/e} \textbf{;}$$

$$\langle\!| \textbf{;} |\!\rangle_{\textsc{RetExp}} \rightarrow^{s/e} \textbf{< >}$$

$$\langle\!| E_{\mathsf{If}}[\![\textbf{return } v, \ldots]\!] |\!\rangle_{\textsc{Break}} \rightarrow^{s/e} \textbf{return } v, \ldots$$

Figura 4.31: Semántica de finalización de llamada a función.

Como mencionamos previamente, la semántica de llamadas que ocurren como sentencias son semejantes a la de llamadas que ocurren como expresión, con la única diferencia de que las etiquetas que utilizamos ayudan a distinguir entre ambos casos. Esta información contextual será utilizada al momento de definir la semántica de finalización de una llamada a función, como mostramos en la Figura 4.31. Primero, los valores retornados desde una llamada que ocurre como sentencia (es decir, que fue etiquetada con RetStat) son, simplemente, descartados. Notar que, nuevamente, hacemos uso de la categoría *Elf* de contextos de evaluación, para expresar el concepto de *punto de retorno etiquetado más interno*, hacia el cual debe retornar una función. Por otro lado, los valores retornados de una llamada que ocurre como expresión son colocados en una tupla, sobre la cual se aplicarán, a posterior, reglas que explican cómo embeber los valores de la misma dentro de la expresión en donde haya ocurrido la llamada.

La tercer y cuarta regla explican cómo se continua la ejecución, cuando culmina una función que no retorna valor alguno. El caso que merece mención es el de una llamada que ocurre como expresión, la cual evaluará a una tupla vacía.

La última regla muestra la interacción entre la sentencia **return** y un bloque de sentencias etiquetado con el punto hacia el cual la sentencia **break** debe saltar: una sentencia **return** presente en el cuerpo de un bucle **while** termina con la ejecución del mismo.

$$
\begin{array}{lll}
E_{\mathsf{tel}} & ::= & v \textbf{ , } \ldots \textbf{ , } [\![\,]\!] \textbf{ , } e \textbf{ , } \ldots \\
E_{\mathsf{t}} & ::= & \textbf{if } [\![\,]\!] \textbf{ then } s \textbf{ else } s \\
 & | & \textbf{local } x \textbf{ , } \ldots \textbf{ = } E_{\mathsf{tel}} \textbf{ in } s \\
 & | & \textbf{return } E_{\mathsf{tel}} \\
 & | & evar \textbf{ , } \ldots \textbf{ , } [\![\,]\!] \textbf{ [} e \textbf{] , } \ldots \textbf{ = } e \textbf{ , } \ldots \\
 & | & evar \textbf{ , } \ldots \textbf{ , } v \textbf{ [} [\![\,]\!] \textbf{] , } \ldots \textbf{ = } e \textbf{ , } \ldots \\
 & | & evar \textbf{ , } \ldots \textbf{ = } E_{\mathsf{tel}} \\
 & | & \textbf{\$statFcall } [\![\,]\!] \textbf{ (} e \textbf{ , } \ldots \textbf{)} \\
 & | & [\![\,]\!] \textbf{ (} e \textbf{ , } \ldots \textbf{)} \\
 & | & \textbf{\$statFcall } v \textbf{ (} E_{\mathsf{tel}} \textbf{)} \\
 & | & v \textbf{ (} E_{\mathsf{tel}} \textbf{)} \\
 & | & \textbf{\$statFcall } [\![\,]\!] \textbf{ : } x \textbf{ (} e \textbf{ , } \ldots \textbf{)} \\
 & | & [\![\,]\!] \textbf{ : } x \textbf{ (} e \textbf{ , } \ldots \textbf{)} \\
 & | & [\![\,]\!] \; binop \; e \\
 & | & v \; strictbinop \; [\![\,]\!] \\
 & | & unop \; [\![\,]\!] \\
 & | & \textbf{(} [\![\,]\!] \textbf{)} \\
 & | & \textbf{< } E_{\mathsf{tel}} \textbf{ >} \\
 & | & \textbf{\{ } efield \textbf{ , } \ldots \textbf{ , [} [\![\,]\!] \textbf{] = } e \textbf{ , } field \textbf{ , } \ldots \textbf{ \}} \\
 & | & \textbf{\{ } efield \textbf{ , } \ldots \textbf{ , [} v \textbf{] = } [\![\,]\!] \textbf{ , } field \textbf{ , } \ldots \textbf{ \}} \\
 & | & \textbf{\{ } efield \textbf{ , } \ldots \textbf{ , } [\![\,]\!] \textbf{ , } field \textbf{ , } \ldots \textbf{ \}} \\
 & | & [\![\,]\!] \textbf{ [} e \textbf{]} \\
 & | & v \textbf{ [} [\![\,]\!] \textbf{]}
\end{array}
$$

Figura 4.32: Contextos de evaluación en donde las tuplas son truncadas.

$$
\begin{array}{lll}
E_{\text{uel}} & ::= & v\ ,\ \ldots\ ,\ [\![\ ]\!] \\
E_{\text{u}} & ::= & \textbf{local}\ x\ ,\ \ldots\ =\ E_{\text{uel}}\ \textbf{in}\ s \\
 & | & \textbf{return}\ E_{\text{uel}} \\
 & | & evar\ ,\ \ldots\ =\ E_{\text{uel}} \\
 & | & \textbf{\$statFcall}\ v\ (\ E_{\text{uel}}\ ) \\
 & | & v\ (\ E_{\text{uel}}\ ) \\
 & | & <\ E_{\text{uel}}\ > \\
 & | & \{\ efield\ ,\ \ldots\ ,\ [\![\ ]\!]\ \}
\end{array}
$$

Figura 4.33: Contextos de evaluación donde las tuplas son concatenadas.

60: Notar que una lista $v, \ldots$ contiene 0 o más valores.

$$E_t[\![< v_1, v_2, \ldots >]\!] \rightarrow^{s/e} E_t[\![v_1]\!]$$

$$E_t[\![< >]\!] \rightarrow^{s/e} E_t[\![\ \textbf{nil}\ ]\!]$$

$$E_u[\![< v, \ldots >]\!] \rightarrow^{s/e} E_u\ [\![< v, \ldots >]\!]^{un}$$

Figura 4.34: Operaciones sobre tuplas.

[27]: Casey Klein y col. (2011), «A Semantics for Context-Sensitive Reduction Semantics»

$$v_1\ (v_2, \ldots, [\![\ ]\!])\ [\![< v_3, v_4, \ldots >]\!]^{un} = v_1\ (v_2, \ldots, v_3, v_4, \ldots)$$

$$v_1\ (v_2, \ldots, [\![\ ]\!])\ [\![<>]\!]^{un} = v_1\ (v_2, \ldots)$$

Figura 4.35: Definición de $\_[\![\_]\!]^{un}$.

```
> print ( print () ,  print () ,
          print () )


nil  nil
```

Figura 4.36: Ejemplo de semántica de tuplas vacías.

61: Ver `www.lua.org/manual/5.2/manual.html#3.4`.

Semántica de tuplas

Una vez que los valores retornados por una función, o los representados por **...** en una función vararg, son colocados en una tupla, se aplican operaciones sobre la tupla que extraen la primer componente de la tupla o concatenan los valores de la tupla a otra lista de valores. Esto va a depender del contexto en donde ocurra la tupla. Para formalizar esto, definimos nuevas categorías de contextos de evaluación, E_t y E_u, presentados en la Figura 4.32 y la Figura 4.33, respectivamente. Representan el contexto *inmediato* en el que una tupla ocurre: si pensáramos a estos contextos como un árbol t, entonces no podríamos encontrar un sub-árbol propio de t que sea él mismo otro contexto de evaluación válido y tenga la misma raíz que t, con excepción del caso trivial $[\![\ ]\!]$. E_t y E_u representan la información contextual mínima necesaria para poder decidir qué operación debemos realizar sobre una tupla dada.

En la Figura 4.34 presentamos las operaciones sobre tuplas[60] . Los contextos E_t se refieren a aquellos casos en donde se debe extraer el primer valor de una tupla (operación a la que nos referiremos como *truncar* la tupla). Debido a que se trata de situaciones en las que es necesario obtener algún valor, una tupla vacía es transformada en el valor **nil**.

Los contextos E_u se refieren a aquellas situaciones en las que todos los valores de la tupla deben ser concatenados a otra lista de valores. Dado un contexto de E_u y una tupla t, explicaremos qué significa concatenar los valores de t con la lista de valores en la que ocurre dentro de E_u, mediante una función $\_[\![\_]\!]^{un} \in E_u \times tuple \rightarrow s \cup e$ Mostramos en la Figura 4.35, 2 de las ecuaciones simples con las que capturamos $\_[\![\_]\!]^{un}$. Notar que esta operación es semejante a la de *plugging* [27] entre un contexto y un término.

Con la semántica de tuplas propuesta podemos explicar, por ejemplo, por qué obtenemos lo mostrado en la Figura 4.36, en una sesión con el intérprete oficial. Las primeras 3 líneas vacías son producto de las 3 llamadas a print sin argumentos. Luego de cada llamada print (), obtenemos una tupla vacía: no hay resultados devueltos. Las primeras 2 tuplas están en posiciones en donde se requiere un valor, como se indica en los contextos de E_t. La segunda regla de Figura 4.34 explica que las tuplas vacías en tales posiciones deben ser convertidas a **nil**. Finalmente, la tercer tupla vacía obtenida de la llamada print () se encuentra en una posición en donde debemos realizar la operación $\_[\![\_]\!]^{un}$. Las ecuaciones de la Figura 4.35 muestran que tal operación, sobre una tupla vacía, resulta simplemente en el descarte de la tupla. Obtenemos la lista **nil nil** como parámetros para la llamada a print más externa, y eso es lo que se muestra en la última línea. Notar que es difícil explicar esta comportamiento tomando en cuenta la descripción provista por el manual de referencia[61].

Finalmente, en la Figure 4.24 habíamos introducido las *expresiones parentizadas*: para una expresión (e), si e resulta en una tupla, el valor de (e) es el del primer elemento de la tupla, o **nil**, si la tupla es vacía. Esto indica que $([\![\ ]\!])$ (es decir, el contexto de evaluación inmediato

de e en la expresión (e)), puede ser visto como un contexto de E_t. Luego, podemos explicar la semántica de expresiones parentizadas como contextos en donde las tuplas son truncadas. De allí la inclusión de $([\![\ ]\!])$ en E_t.

Equivalencia entre clausuras

La definición de una función incluye una etiqueta única, la cual ayuda a distinguir entre las distintas definiciones, dentro del programa. Cómo son representadas tales etiquetas no es importante, solo necesitamos que sea posible comparar entre las mismas. La unicidad de estas etiquetas, para cada definición de función, está garantizada por el proceso de compilación, en el caso de traducción de programas Lua hacia nuestro lenguaje. De otro modo, la unicidad de estas etiquetas forma parte de la noción programa bien formado, que introducimos en §4.3.

En nuestro modelo, estas etiquetas consisten en una representación abstracta de la información que es tomada del código fuente de un programa Lua, durante su compilación, y que es utilizada como parte del criterio empleado para decidir sobre la reutilización de una clausura ya creada, como resultado del intento de creación de una nueva clausura. Se trata de una optimización permitida por el manual de referencia[62], con claro impacto en la semántica, que será incluida a los fines de poder testear nuestro modelo con respecto al intérprete oficial. Lo que el manual de referencia describe al respecto es que la creación de una clausura puede reutilizar otra ya creada, si no hay *diferencias observables* entre ambas clausuras. El documento no especifica a qué se refiere con *diferencias observables*, lo que deriva en la situación descripta en §1. Precisaremos aquí los detalles que omite el manual.

En esencia, necesitamos observar la definición de función utilizada (identificada unívocamente por su etiqueta) y el entorno de la clausura. Como una definición de función incluye, embebido en su cuerpo, el entorno mismo de la clausura que representa, podemos utilizar directamente las definiciones de funciones como los valores a comparar, y manipular, obteniendo una correcta representación de la semántica de Lua 5.2, cuyas peculiaridades son mostradas en el ejemplo breve de la Figura 4.37, que presenta una sesión con el intérprete oficial de Lua 5.2[63]

Para comprender cómo es que nuestro modelo reproduce el comportamiento anterior, observemos el código compilado que obtendríamos si expresáramos el programa en cuestión en nuestro lenguaje, presentado en la Figura 4.38.

Primero, recordar de §2.2, que las variables globales son, simplemente, campos de la tabla _ENV: es decir, _ENV es la representación en términos de construcciones del lenguaje, del concepto de entorno global. Luego, notar que las clausuras asignadas a las variables globales f y g, mientras que tienen la misma signatura y cuerpo, son creadas a partir de definiciones de función distintas, lo cual es reflejado en sus (distintas) etiquetas (generadas automáticamente durante la compilación).

62: Ver www.lua.org/manual/5.2/manual.html#8.1.

```
> f = function() end
> g = function() end
> print(f == g)
false
> h = function()
      return function() end
    end
> print(h() == h())
true
```

Figura 4.37: Optimización en la creación de clausuras.

63: Por otro lado, LuaJIT 2.0, otra implementación parcialmente compatible con Lua 5.2, no incluye la mencionada optimización: cada clausura creada es nueva. Luego, si interpretamos con LuaJIT las líneas `print(f == g)` y `print(h() == h())`, del ejemplo mostrado, retornarían ambas **false**. Este comportamiento, sigue siendo compatible con el manual de referencia de Lua 5.2, pues el manual especifica

```
_ENV["f"] = function $1 ()
              ;
            end

_ENV["g"] = function $2 ()
              ;
            end

_ENV["print"](_ENV["f"] ==
              _ENV["g"])

_ENV["h"] = function $3 ()
              return
                function $4 ()
                  ;
                end
            end

_ENV["print"](_ENV["h"]()
                ==
              _ENV["h"]()
             )
```

Figura 4.38: Código compilado.

```
> for  i=1,10 do
       print (function()
                 return i
              end)
  end

function: 0x1c4d970
function: 0x1c4e3d0
...
```

Figura 4.39: Diferentes clausuras a partir de la misma definición de función.

```
1  local $var,  $limit ,  $step =
2        1,  10,  1
3  in
4   while ($step > 0.0 and
5           $var <= $limit )
6         or
7         ($step <= 0.0 and
8          $var >= $limit )
9   do
10   local  i  = $var in
11    _ENV[ "print"  ](function $1 ()
12                       return i
13                     end)
14    $var = $var + $step
15   end
16  end
17 end
```

Figura 4.40: for numérico expresado mediante un bucle while.

Por otro lado, las clausuras retornada por cada llamada a h son creadas a partir de la misma definición de función (luego, la misma etiqueta) y tienen el mismo entorno (luego, cada clausura va a tener el mismo cuerpo).

Por otro lado, es posible que la misma definición de función genere clausuras diferentes, debido a diferencias en el entorno de las clausuras (el cual, recordemos, se reduce a las referencias de variables externas embebidas en el cuerpo de las clausuras). Esta situación se ilustra en la Figura 4.39, en donde mostramos el resultado de ejecutar con el intérprete oficial un bucle *for* numérico, en donde, en cada iteración, se crea una nueva clausura y se imprime en consola la referencia a la misma (recordar que en Lua, las clausuras son manejadas por referencias; nosotros no apelamos a ese recurso). Se puede observar cómo cada referencia es distinta, indicando que cada clausura creada difiere de la anterior.

De acuerdo al manual de referencia, para implementar la iteración sobre una variable dada en un bucle *for* numérico, i en nuestro ejemplo, se crea en cada iteración una variable local nueva, a la cual se le asigna el valor correspondiente a la iteración. Esta nueva variable local está capturada por el entorno de cada clausura creada, y de allí que, en cada iteración, la misma definición de función represente a una clausura distinta, cada vez que es evaluada.

En la Figura 4.40 mostramos el código compilado[64] correspondiente al ejemplo de la Figura 4.39. En esencia, prestemos atención a las líneas 10-13, donde la misma definición de función se encuentra en el alcance de la definición de una nueva variable local, en cada iteración. Luego, cada clausura creada va a tener, en su entorno, una referencia distinta mapeada al identificador i.

4.2.4. Servicios de librería

La formalización de los servicios de librería dependerá de varios elementos: primero, la semántica de los mismos, expresada de manera declarativa por la función δ (como ya se presentó en §4.1); segundo, una construcción en el lenguaje que permita invocar δ desde un programa; finalmente, una definición del entorno de ejecución dentro del cual se ejecuta un programa, que provea *bindings* a estos servicios de librería.

Semántica de los servicios

La función δ explica, de un modo declarativo, la semántica de los servicios de librería y los operadores primitivos. Tomada de ISWIM[22] , en donde se la utiliza para abstraer una familia de lenguajes de programación sobre operadores primitivos y constantes[65] , aquí se incorpora por motivos diferentes. También presentamos un lenguaje con construcciones con una semántica operacional, mientras que otras construcciones se explican con especial foco en el resultado que obtenemos al utilizarlas, pero, en nuestro caso, utilizamos este enfoque para reflejar el estilo

64: Tratamos a un bucle for como una abstracción lingüística y lo expresamos mediante un bucle while, del mismo modo en el que se lo explica en el manual de referencia, www.lua.org/manual/5.2/manual.html#3.3.5.

[22]: Landin (1966), «The next 700 programming languages»

65: ISWIM está definido como una familia de lenguajes indexada por los operadores primitivos y las constantes, quienes son definidos de acuerdo al dominio específico para el cual se quiera construir un lenguaje de programación particular.

del manual de referencia. Allí, no hay una explicación operacional desde la cual poder obtener los resultados, sino que sólo se menciona qué resultado esperar al llamar un servicio dado o utilizar un operador en particular. A su vez, mientras que podríamos encontrar servicios u operadores para los cuales sería posible proveer una descripción operacional de su comportamiento, por motivos de cohesión aplicaremos el mismo enfoque para la descripción de la semántica de todos ellos.

Cuando se introdujo la función δ, en §3, se lo hizo a los fines de precisar la semántica de algunos operadores aritméticos y lógicos. En ese caso, se podría haber mencionado que la imagen de δ se encuentra en el conjunto de los valores del lenguaje. Sin embargo, ese conjunto no va a ser suficiente para describir la semántica de los servicios de librería. Vamos a ver que, para poder describir el resultado de varios servicios, vamos a necesitar una función δ con una signatura más compleja.

Invocar δ

En §3, para los términos que representaban la aplicación de un operador primitivo, definimos explícitamente reglas que reducían la semántica de tales términos a la aplicación de δ. Para los servicios de librería proveeremos una nueva construcción que servirá como mecanismo genérico para invocar δ, y así extender el lenguaje con nuevos servicios. Su sintaxis se presenta en la Figura 4.41.

svc ::= **assert** | **error** | ...

e ::= ... | **$builtIn** *svc* (*e*,...)

Figura 4.41: Sintaxis de la construcción **$builtIn**.

svc será un conjunto de etiquetas que identificarán al servicio que se quiere invocar. Agregamos la construcción **$builtIn** como expresión, a los fines de simplificar la definición de su semántica (no extendemos a **$builtIn** la ambivalencia sentencia/expresión de las llamadas a función). La semántica de **$builtIn**, presentada en Figura 4.42, es simple: se reduce a la interpretación de δ. Distinguimos distintos casos, de acuerdo a la semántica de los servicios que se están modelando. Los servicios más sencillo (aquellos que no requieren de acceder al almacenamiento), son explicados con la primer regla de la Figura 4.42. Si requiere de acceder al almacenamiento (que, para el caso de los servicios que incluimos en nuestro modelo, siempre consistirá en el almacenamiento de tablas), pero solo a los fines de lectura, entonces se ejecuta la segunda regla (notar cómo, en la invocación de δ, se incluye ahora no solo los parámetros con los que se invoca el servicio, sino también el almacenamiento θ). Finalmente, si el servicio modifica el almacenamiento de tablas, su semántica se explica con la tercer regla[66]
.

$$\frac{svc \in \{type, assert, error, pcall, select, ...\}}{\textbf{\$builtIn}\ svc\ (v_1, ..., v_n) \xrightarrow{s/e} \delta(svc,\ v_1, ..., v_n)}$$

$$\frac{svc \in \{ipairs, next, pairs, getmetatable, ...\}}{\theta : \textbf{\$builtIn}\ svc\ (v_1, ..., v_n) \xrightarrow{\theta} \theta : \delta(svc, v_1, ..., v_n, \theta)}$$

$$\frac{svc \in \{rawset, setmetatable\} \quad (\theta_2, v) = \delta(svc, v_1, ..., v_n, \theta_1)}{\theta_1 : \textbf{\$builtIn}\ svc\ (v_1, ..., v_n) \rightarrow^{\theta} \theta_2 : v}$$

Figura 4.42: Semántica de **$builtIn**.

Como es de esperar, podemos ver que la signatura de δ incluye lo que bien podría entenderse como el dominio de una relación que explica la semántica operacional de construcciones del lenguaje. En efecto, la diferencia entre δ y las relaciones con las que describimos la operación de los programas radica en el aspecto que capturan de las primitivas de programación, pero no necesariamente en la complejidad de tales primitivas.

En las condiciones laterales de las reglas hemos mencionado algunos de los servicios incluidos en nuestro modelo. La lista completa

66: En la tercer regla, tanto el valor al que evalúa δ, como el dominio de $\rightarrow^{\theta}$, son pares ordenados. Allí expresamos lo que retorna δ, como condición lateral, solo para llamar la atención sobre el hecho de que se trata de un par ordenado, y explicar cuales son sus componentes. A su vez, la notación usada para las configuraciones es solo por economía de escritura y claridad, pero no hay ninguna diferencia entre estas y el par ordenado devuelto por δ.

incluye todos los servicios de las funciones básicas (excepto manejo de archivos), junto con servicios de las librerías math, string y table.

Entorno de ejecución

Los servicios ofrecidos por la librería estándar de Lua son presentados, ante la persona que programa, a través de *bindings* de alcance global. Es decir, los servicios están almacenados como entradas de la tabla global _ENV. Recordemos, de §2.2, que utilizar un identificador de variable que no está en *scope* es equivalente a indexar esta tabla global utilizando como llave el identificador en cuestión. Luego, acceder a servicios de librería se reduce a consultar la tabla global con la llave que se corresponda con el servicio deseado. Por ejemplo, podemos llamar al servicio type, el cual retorna el tipo de un valor dado, del modo que se muestra en la primer línea de la Figura 4.43[67].

67: = type ({}) es equivalente a print (type ({})). Se trata de una facilidad provista por el intérprete oficial. No es código Lua válido.

```
1  > = type ({})
2  table
3  > = _ENV["type "]({})
4  table
5  > type = function ()
6              return 'not a type'
7            end
8  > = type ({})
9  not a type
10 > next(1)
11 stdin :1:  bad argument (table
       expected, got number)
```

Figura 4.43: Accediendo a servicios de librería.

Por otro lado, lo anterior es equivalente a lo mostrado en la línea 3, en donde se accede de manera explícita a través de la tabla global. Naturalmente, podemos alterar el valor al cual está ligada la variable global type, obteniendo lo mostrado en las líneas 5–9.

Sin embargo, el servicio type sigue siendo accesible para otros servicios de librería, como se puede ver en la línea 10, para el servicio next, el cual provee un iterador que nos permite recorrer la tabla que recibe como argumento. La semántica del servicio involucra un chequeo, en tiempo de ejecución, del tipo del argumento recibido, como se ve en el error que obtenemos luego de invocarlo, pasándole un número como argumento.

Para modelar el comportamiento que observamos, previa a la ejecución de un programa, poblaremos la tabla _ENV con los servicios de librería, modelados como simples funciones. Estas funciones serán simplemente *wrappers* de la invocación de δ mediante **$builtIn**. Luego, la definición misma de estos servicios, explicada mediante δ, puede invocar a otros servicios directamente mediante δ o, si fuera necesario, mediante **$builtIn**.

El diseño aquí propuesto, para formalizar e incorporar servicios de librería, nos permite lograr la correspondiente conformidad con respecto al intérprete oficial (como se detalla en §6) y reduce la esencia de la semántica de cada servicio a lo explicado por δ. Esto nos permite poder experimentar con distintas especificaciones de δ (el fragmento más complejo, y con semántica poco especificada en el manual de referencia; como ocurre también en otros lenguajes), sin tener que realizar mayores cambios en otras partes del modelo.

Ejemplos

Cerramos esta sección con la descripción de 2 servicios de librería, pairs y tonumber, que suponen, cada uno, un desafío diferente de formalización.

$$\delta(\text{pairs}, tid, \theta) = \left| \begin{array}{l} (\textbf{function}\ \$\text{getIter}\ () \\ \quad \textbf{local}\ \text{v1},\ \text{v2},\ \text{v3} = \text{h}(\text{tid})\ \textbf{in} \\ \quad\quad \textbf{return}\ \text{v1},\ \text{v2},\ \text{v3} \\ \textbf{end})() \end{array} \right.$$
where $h = indexmeta(tid, \text{“__pairs”}, \theta)$
and $h \neq \textbf{nil}$

$$\delta(\text{pairs}, tid, \theta) = \left| \begin{array}{l} < \textbf{function}\ \$\text{next}\ (\text{table},\ \text{index}) \\ \quad \textbf{return}\ \$\textbf{builtIn}\ \text{next}(\text{table},\ \text{index}) \\ \textbf{end},\ tid\ ,\ \textbf{nil}\ > \end{array} \right.$$
if $indexmeta(tid, \text{“__pairs”}, \theta) = \textbf{nil}$

$$\delta(\text{pairs}, v, \theta) = \delta(\text{error}, \text{“}table\ expected,\ got\ \text{”} \mathbin{..} \delta(\text{type}, v))$$
if $\delta(\text{type}, v) \neq \text{“}table\text{”}$

Figura 4.44: Funciones básicas de la librería estándar: pairs.

pairs En la Figura 4.44, δ es utilizada para definir el resultado de una llamada al servicio **pairs**. Este es utilizado para iterar una tabla, en conjunto, por ejemplo, con un bucle **for**. Tras una invocación del servicio para una tabla *tid*, este retorna 3 valores: una función iteradora, la tabla sobre la que se va a iterar y el primer índice de la tabla. De acuerdo a las ecuaciones de la Figura 4.44, se distinguen 3 escenarios: el primer caso, cuando la tabla *tid* posee un sustituto *h* del servicio **pairs**, almacenado en el campo de llave __pairs de su metatabla; en este caso, se delega la operación a *h*, para obtener la terna de valores necesaria para la iteración. La meta-función indexmeta indexa directamente la metatabla de la tabla recibida como primer argumento, con la clave recibida como segundo argumento. Su definición se presenta en §4.2.5.

Puede resultar extraño el que se retorne una función cuyo cuerpo llama a *h*, en lugar de retornar directamente la expresión que representa a la llamada. La razón detrás de esta definición, por un lado, tiene que ver con lo que especifica el manual: solo se deben retornar los primeros 3 valores de la llamada a la función *h*. Logramos esto mediante la definición de 3 variables, cada una tomando un valor, y retornando las mismas. Si *h* retorna más valores, serán descartados. Segundo, dado que **local** y **return** no son expresiones válidas, y δ debe retornar una expresión[68] , colocamos el código devuelto en el cuerpo de una clausura, y devolvemos una expresión que representa la llamada a esta clausura.

68: Recordemos que, para simplificar la semántica de la construcción **$builtIn**, solamente la hemos incluido en el conjunto de expresiones del lenguaje.

En la segunda ecuación de Figura 4.44, cuando la tabla sobre la que se pretende iterar no posee metatabla, o esta no tiene definido un campo de clave __pairs, el manual de referencia[69] especifica que pairs(t) retorna “*la función next, la tabla t, y* ***nil***". Modelamos esto mediante una clausura que solo invoca al servicio next mediante **$builtIn**. La etiqueta de esta clausura creada, *$next*, será la misma que la que utilizaremos par definir el procedimiento de envoltorio para el servicio next, como así también su cuerpo, generando, de este modo, la semántica esperada.

69: http://www.lua.org/manual/5.2/manual.html#pdf-pairs

Finalmente, la última ecuación de Figura 4.44 explica la generación de

mensajes de error, para los casos en los que el servicio pairs se llamo para un valor que no es una tabla.

tonumber El servicio tonumber permite convertir un string, como también otros números, a un número en base decimal. El número representado por el string, o los números a convertir, pueden estar en cualquier base numérica, entre 2 y 35. Siendo solamente un servicio de librería, el manual solo explica los resultados esperables de una llamada al mismo, excluyendo la especificación de, por ejemplo, cómo se interpretan los strings. Indagar en los detalles de su implementación o experimentar con la misma, para rescatar una especificación, no es fundamental para nuestro modelo. De todos modos, a los fines de poder correr las suite de test del intérprete oficial, necesitamos incluir este servicio en nuestro modelo de algún modo.

Mientras que, para servicios como pairs, podemos proveer fácilmente una especificación mediante ecuaciones que hacen foco en los resultados esperados, para la semántica de servicios como tonumber nos limitaremos a especificar su signatura, abstraída en la definición de δ. Luego, desde la mecanización con Redex, podemos proveer un significado razonable a tonumber mediante el uso de los servicios brindados por Racket (el lenguaje de programación sobre el que está implementado Redex). Esto es, usamos δ como manera de hacer estos servicios disponibles para los programas, solo por el propósito de la conformidad con el intérprete Lua, mientras que, al mismo tiempo, evitamos la formalización de detalles de implementación que no son pertinentes.

Se puede ver que, entonces, nuestra formalización se encontrará en correcta correspondencia con la implementación oficial dependiendo de que se provea una correcta implementación de estos servicios.

4.2.5. Meta-Tablas

Las meta-tablas le permiten a quien programa especificar *manejadores* para ciertas operaciones, ejecutadas sobre operandos de tipo inesperado o, en general, que no satisfacen alguna propiedad esperada: aritmética sobre valores no numéricos, concatenación sobre valores que son strings, comparación de igualdad entre valores distintos, llamadas a función sobre un valor que no es una clausura, indexado o asignación de campo de tabla utilizando una clave no existente, etc.

Las meta-tablas consisten en tablas Lua ordinarias definidas por el programa, y sus manejadores son, típicamente, funciones que deben ser asociadas con claves específicas. Por ejemplo, "__add" es la clave asociada con el manejador para la operación de suma, y "__newindex" está asociada con el manejador de la operación de asignación de campo de tabla, con clave inexistente.

$$indexmeta(\mathit{tid}, v, \theta) \doteq \delta(\text{rawget},\ \theta(\mathit{tid})(2), v)$$
$$protmeta(\mathit{tid}, \theta)\ \doteq indexmeta(\mathit{tid}, \text{“__metatable”}, \theta)$$
$$prot?(\mathit{tid}, \theta)\ \doteq protmeta(\mathit{tid}, \theta) \neq \textbf{nil}$$

$$\delta(\text{setmetatable}, v_1, v_2, \theta) = (\delta(error, \text{“...”}), \theta), \text{ if } \begin{cases} \delta(\text{type}, v_2) \notin \{\text{“}table\text{”}, \text{“}nil\text{”}\} \\ \qquad\qquad \vee \\ prot?(\mathit{tid}, \theta_1) \\ \qquad\qquad \vee \\ \delta(\text{type}, v_1) \neq \text{“}table\text{”} \end{cases}$$

$$\delta(\text{setmetatable}, \mathit{tid}, v, \theta_1) = (\mathit{tid}, \theta_2), \text{ if } \begin{cases} \delta(\text{type}, v) \in \{\text{“}table\text{”}, \text{“}nil\text{”}\} \\ \neg prot?(\mathit{tid}, \theta_1) \\ \theta_2 = \theta_1[\mathit{tid} := (\theta_1(\mathit{tid})(1), v)] \end{cases}$$

$$\delta(\text{getmetatable}, \mathit{tid}, \theta) = \begin{cases} protmeta(\mathit{tid}, \theta) & \text{if} \quad prot?(\mathit{tid}, \theta) \\ \theta(\mathit{tid})(2) & \text{o/w} \end{cases}$$

$$\delta(\text{getmetatable}, v, \theta)\ = \begin{cases} \textbf{nil} & \text{if} \quad \mathit{tid}_{\delta(\text{type},v)} \notin dom(\theta) \\ \mathit{tid}_{\delta(\text{type},v)} & \text{if} \quad \begin{cases} \mathit{tid}_{\delta(\text{type},v)} \in dom(\theta) \\ \neg prot?(\mathit{tid}_{\delta(\text{type},v)}, \theta) \end{cases} \\ protmeta(\mathit{tid}_{\delta(\text{type},v)}, \theta) & \text{o/w} \end{cases}$$

Figura 4.45: Primitivas para instalar y extraer una meta-tabla.

Manipulación de meta-tablas.

Para cada tipo en Lua, excepto tablas y *userdata* (no incluido en nuestro modelo), es posible definir una única meta-tabla, a ser consultada bajo las operaciones especiales cuya semántica puede definirse con este mecanismo de meta-programación. Más adelante explicaremos cómo instalar una meta-tabla para estos tipos.

Para tablas es posible definir una meta-tabla por valor. La librería básica provee 2 primitivas, setmetatable, para definir una meta-tabla para un tabla dada, y getmetatable, para obtener la meta-tabla asociada con un valor dado (de cualquier tipo, no necesariamente una tabla). Siendo servicios de la librería básica, proveemos una descripción declarativa de su comportamiento mediante δ. Estas primitivas operan sobre el almacenamiento de tablas, por lo cual, para especificar su comportamiento necesitaremos poder consultar al almacenamiento θ. Su semántica es presentada en Figura 4.45.

La primera ecuación para capturar setmetatable describe las situaciones erróneas que pueden presentarse utilizando esta primitiva. Esto incluye intentar instalar una meta-tabla para un valor que no es una tabla; intentar definir, como meta-tabla, un valor que no es una tabla o **nil** (en cuyo caso, el efecto sería el de eliminar una meta-tabla previamente definida para la tabla en cuestión, tal como se muestra en la ecuación siguiente); o intentar modificar una meta-tabla *protegida*: una meta-tabla que tiene definida un campo con clave “__metatable”, el cual,

cuando está definido, evita el acceso y la modificación de la meta-tabla [28] [70]. Para poder verificar esta última condición, para una tabla dada *tid* y almacenamiento θ, definimos el predicado $prot?(tid, \theta)$, presentado en Figura 4.45. Al respecto notemos, también, que las ecuaciones para getmetatable muestran que, si el campo de clave "__metatable" está presente, la primitiva simplemente retorna el valor asociado a la clave, en lugar de la meta-tabla.

Retornando a la especificación de setmetatable, su segunda ecuación muestra el comportamiento bajo circunstancias normales: si la meta-tabla de una tabla *tid* dada no está protegida, y si queremos instalar una nueva meta-tabla o eliminar la meta-tabla previamente definida (es decir, asignar **nil** en lugar de una tabla), la primitiva simplemente asocia en el almacenamiento θ_2 el nuevo valor de la meta-tabla, junto con *tid* (a través de un par ordenado, como se mencionó en §4.2.1). En estas condiciones normales, la primitiva retorna la tabla *tid* y el nuevo almacenamiento θ_2.

La primer ecuación que especifica al servicio getmetatable muestra lo que retorna cuando se lo invoca para una tabla *tid*, protegida o no, en cuyo caso simplemente retorna el valor asociado a *tid*, sea otro identificador de tabla o **nil**, significando en este caso que *tid* no tiene una meta-tabla definida. Si la meta-tabla está protegida, se devuelve el valor asociado a la clave "__metatable", como ya se mencionó.

La ecuación restante replica el comportamiento previo de getmetatable, pero para valores que no son tablas. Para definir una meta-tabla de un tipo que no sea tabla o userdata, quien programa debe utilizar la API C, que sirve de interfaz con el programa Lua desde un programa anfitrión. En nuestro modelo asumiremos que, si está definida, la meta-tabla para cada uno de estos tipos estará en correspondencia, mediante un almacenamiento θ, con identificadores específicos definidos de antemano los cuales no serán reutilizables por nuestra semántica. Esto es, para un valor *v* que no es de tipo tabla, asumimos que, si la meta-tabla para el tipo de *v* está instalada, debe ser que está identificada por un cierto $tid_{\delta(\text{type},v)}$. Luego, determinar si tal meta-tabla está definida se reduce a consultar si $tid_{\delta(type,v)} \in \theta$.

[28]: Ierusalimschy (2016), *Programming in Lua*

70: No es posible encontrar mayores detalles sobre su propósito, en el manual de referencia.

$$\frac{v_3 = binhandler(v_1, v_2, bopevkey(op), \theta) \quad v_3 \neq \textbf{nil}}{\theta : \langle\!| v_1\ op\ v_2 |\!\rangle_{\text{ARITHWO}} \rightarrow^{meta} \theta : (v_3\ (v_1, v_2))}$$

$$\frac{binhandler(v_1, v_2, bopevkey(op), \theta) = \textbf{nil} \quad t_1 = \delta(\text{type}, v_1) \quad t_2 = \delta(\text{type}, v_2) \quad msg = errmessage(\text{ARITHWO}, t_1, t_2)}{\theta : \langle\!| v_1\ op\ v_2 |\!\rangle_{\text{ARITHWO}} \rightarrow^{meta} \theta : \delta(\text{error}, msg)}$$

Figura 4.46: Mecanismo de meta-tablas para operadores aritméticos binarios.

Resolviendo términos etiquetados.

En secciones previas hemos visto que la semántica regular de ciertas operaciones etiqueta a las sentencias o expresiones cuando se encuentra una situación en donde no se le puede dar sentido a la operación que está siendo ejecutada (típicamente, por causa de operandos de tipo no esperado). Al etiquetar el término, y no explicar con aquellas relaciones cómo continua la ejecución, se le delega al mecanismo de meta-tablas la tarea de continuar con la ejecución del programa. Este diseño nos ayuda a simplificar las reglas con las que explicamos el comportamiento del mecanismo de meta-tablas (al trasladar a la sintaxis información que no siempre se puede determinar sintácticamente), y ayuda a modularizar la semántica. Al respecto, para explicar meta-tablas utilizaremos una nueva relación, $\rightarrow^{meta} \subseteq \theta \times (\mathbf{s} \cup \mathbf{e})$.

La Figura 4.46 muestra cómo $\rightarrow^{meta}$ resuelve las operaciones aritméticas sobre operandos de tipo no numérico (ni strings convertibles a números), condición que es explicada con la etiqueta ARITHWO que $\rightarrow^{s/e}$ colocó. La meta-función *binhandler* es análoga a *getbinhandler*, del manual de referencia [71]: busca por un manejador, primero en la meta-tabla del operando izquierdo v_1, o, si no lo encontró, busca en la meta-tabla de v_2. Los manejadores deben estar almacenados en las meta-tablas, en campos con claves especificas, que dependen del operador involucrado en la expresión etiquetada. Por ejemplo, la clave consultada en caso de una suma es "__add"; en caso de una resta, "__sub", etc. Este mapeo entre operadores binarios y claves lo abstraemos en la meta-función bopevkey.

La operación de buscar un manejador en una meta-tabla dada siempre está definida, y retornará el manejador o **nil**, debido a 2 invariantes: la meta-tabla de una tabla siempre es una tabla (pues setmetatable realiza el correspondiente chequeo de tipos en tiempo de ejecución), y la meta-tabla de una meta-tabla, si está definida, será ignorada en esta operación. Esto es, abstraer esta operación con *binhandler* no compromete la naturaleza small-step de la semántica.

Retornando a la Figura 4.46, la primer regla de muestra cómo la operación es reescrita como una aplicación del correspondiente manejador sobre los operandos de la operación fallida. Notar que nos quedamos solo con el primer valor que pudiera retornar el manejador, al colocar al llamada entre paréntesis. Si el manejador no es una función, esto disparará nuevamente al mecanismo de meta-tablas.

La segunda regla explica qué ocurre cuando no es posible hallar un manejador: se lanza un error, utilizando el servicio error (explicamos su semántica en §4.2.6). Abstraemos en *errmessage* la construcción del correspondiente mensaje[72].

La Figura 4.47 describe cómo opera el mecanismo de meta-tablas para el caso de operaciones de asignación sobre campos de tabla utilizando claves no existentes o realizadas sobre valores que no son tablas. Las primeras 2 reglas muestran cómo se resuelve la situación, independientemente de la etiqueta de la sentencia, cuando hay manejador efectivamente definido para resolver la situación. En tal caso, se distingue cómo proceder, dependiendo del tipo del manejador (la segunda regla, en donde la operación de asignación es repetida, ahora sobre el manejador, ayudaría a entender cómo opera la implementación de POO presentada en §2.3).

La tercer regla muestra que la asignación, sobre una tabla, de un valor v_2, utilizando una clave no existente v_1 y ante la ausencia de un manejador del evento, resulta en la adición del campo $[v_1] = v_2$, a la tabla. Esta es, en definitiva, la semántica por defecto de la asignación en campos de tabla, con llaves nuevas: recién aquí se puede terminar de explicar qué ocurre en tal situación.

La cuarta regla indica que, por defecto (es decir, ante la ausencia de un manejador apropiado), la asignación sobre una tabla utilizando **nil** or **nan** como llaves no está permitido (ver semántica de rawset, en §4.2.1). Finalmente, la última regla presenta el comportamiento por defecto de

71: Ver www.lua.org/manual/5.2/manual.html#2.4.

72: La información relativa a un error consiste en un valor Lua arbitrario. Es posible, por lo tanto, escribir programas que razonen sobre distintos mensajes de error. Siendo que la medida de correspondencia, entre nuestro modelo y el intérprete Lua, será la posibilidad de ejecutar la suite de test de Lua, no podemos descartar que haya programas que implementen este comportamiento, y que tengamos que ser capaces de explicar también con nuestro modelo. Luego, en lugar de descartar este detalle respecto a la semántica de meta-tablas, lo incluimos.

$$\frac{v_4 = indexmeta(v_1, \text{“__newindex”}, \theta) \quad \delta(type, v_4) = \text{“}function\text{”}}{\theta \,:\, (\!|\, v_1\ \mathbf{[}v_2\mathbf{]} = v_3 \,|\!)_{label} \rightarrow^{meta} \theta \,:\, (\mathbf{\$statFcall}\ v_4\ (v_1, v_2, v_3))}$$

$$\frac{v_4 = indexmeta(v_1, \text{“__newindex”}, \theta) \quad v_4 \neq \mathbf{nil} \quad \delta(\text{type}, v_4) \neq \text{“}function\text{”}}{\theta \,:\, (\!|\, v_1\ \mathbf{[}v_2\mathbf{]} = v_3 \,|\!)_{label} \rightarrow^{meta} \theta \,:\, v_4\ \mathbf{[}v_2\mathbf{]} = v_3}$$

$$\frac{indexmeta(tid, \text{“__newindex”}, \theta_1) = \mathbf{nil} \quad (\theta_2, tid) = \delta(\text{rawset}, tid, v_1, v_2, \theta_1)}{\theta_1 \,:\, (\!|\, tid\ \mathbf{[}v_1\mathbf{]} = v_2 \,|\!)_{\text{WRONGKEY}} \rightarrow^{meta} \theta_2 \,:\,;}$$

$$\frac{indexmeta(tid, \text{“__newindex”}, \theta_1) = \mathbf{nil} \quad (\theta_2, \mathbf{\$err}...) = \delta(\text{rawset}, tid, v_1, v_2, \theta_1)}{\theta_1 \,:\, (\!|\, tid\ \mathbf{[}v_1\mathbf{]} = v_2 \,|\!)_{\text{WRONGKEY}} \rightarrow^{meta} \theta_2 \,:\, \mathbf{\$err}...}$$

$$\frac{indexmeta(v_1, \text{“__newindex”}, \theta) = \mathbf{nil} \quad t = \delta(\text{type}, v_1) \quad msg = errmessage(\text{NONTABLE}, t)}{\theta \,:\, (\!|\, v_1\ \mathbf{[}v_2\mathbf{]} = v_3 \,|\!)_{\text{NONTABLE}} \rightarrow^{meta} \theta \,:\, \delta(error,\ msg)}$$

Figura 4.47: Mecanismo de meta-tablas para asignación de campo de tabla.

una operación de asignación de campo de tabla, sobre un valor que no es una tabla.

4.2.6. Semántica de programas y manejo de errores

En la sección §3 explicamos que RS nos habilita para descomponer la semántica en reglas que sólo explican los cómputos, y reglas que explican el orden entre los mismos, para así poder definir la semántica operacional de programas completos. Debido a que RS es un formalismo cuyos únicos recursos son completamente basados en sintaxis, la forma de explicar el orden de ejecución se trata también de un recurso sintáctico: contextos o, para el caso específico de una semántica determinista, contextos de evaluación. En las secciones previas, junto con la introducción de nuevas construcciones del lenguaje, incluimos los correspondientes contextos de evaluación que explican el orden en el que los subtérminos de un término dado son ejecutados. Aquí los vamos a utilizar para definir la *clausura compatible* (junto con los correspondientes almacenamientos σ y θ) de las relaciones que ya hemos presentado, para obtener una nueva relación, $\mapsto \subseteq \sigma \times \theta \times \mathsf{s}$, que define la semántica operacional de programas completos.

$$\frac{t \rightarrow^{s/e} t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma : \theta : E[\![t']\!]}$$

$$\frac{\sigma : t \rightarrow^{\sigma} \sigma' : t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma' : \theta : E[\![t']\!]}$$

$$\frac{\theta : t \rightarrow^{\theta} \theta' : t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma : \theta' : E[\![t']\!]}$$

$$\frac{\theta : t \rightarrow^{meta} \theta' : t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma : \theta' : E[\![t']\!]}$$

Figura 4.48: Semántica de programas.

Para la semántica de las construcciones ya introducidas, la definición de $\mapsto$ es esencialmente una extensión de la relación más simple presentada en la §3, Figura 3.12, con la diferencia de que el dominio incluye ahora a θ. Presentamos su definición en la figura Figura 4.48.

Finalmente, hay cómputos sensibles al contexto que requieren ser definidos al nivel de $\mapsto$, en concreto, propagación de errores. En lo que resta de esta sección, los incluiremos en nuestro modelo, junto con las primitivas para el manejo de los mismos.

Errores y modo protegido

Lua provee mecanismos para generar explícitamente errores y para atraparlos. Los errores son representados mediante lo que el manual de referencia describe como *error objects* o *error messages*. Pueden ser explícitamente generados por el programa, utilizando la función básica de librería error. Es posible asociar información sobre el error, usando cualquier valor Lua. La semántica de un error consiste, en esencia, en abortar la ejecución del programa en donde son generados.

A los fines de atrapar el error, quien programa dispone de 2 servicios de librería, pcall y xpcall, que permiten la ejecución de una función bajo un *modo protegido*: cualquier error generado por la función será atrapado, evitando la propagación del mismo y permitiendo adicionar código dedicado a manejar la situación excepcional encontrada.

Para llamadas a función que son abortadas por un error, pcall retorna **false** junto con la información asociada al error. De otro modo, y de terminar normalmente la función ejecutada en modo protegido, retorna **true** y los valores que haya devuelto la función llamada. xpcall se comporta como pcall, pero agrega la posibilidad de proveer una función

f que se haga cargo de la situación errónea, esto es, en caso de error, se llama a la función f, pasándole como argumento la información asociada al error generado.

Formalización En la Figure 4.49 presentamos las nuevas construcciones agregadas para formalizar errores y modo protegido. Se trata de términos que son generados en tiempo de ejecución. Un error será representado con la construcción **$err** v, que permite la inclusión de un valor Lua arbitrario como representación de la información asociada al error.

$$
\begin{aligned}
s ::= \;& \ldots \mid \textbf{\$err}\ v \mid (\!| s |\!)_{\text{PROTMD}} \\
& \mid (\!| s |\!)^{v}_{\text{PROTMD}} \\
E ::= \;& \ldots \mid (\!| E |\!)_{\text{PROTMD}} \\
& \mid (\!| E |\!)^{v}_{\text{PROTMD}}
\end{aligned}
$$

Figura 4.49: Construcciones de tiempo de ejecución para introducir errores y modo protegido.

Respecto a la introducción de una representación de la noción de modo protegido, lo que demandamos para la misma es que nos permita modelar la ejecución de una función hasta su (posible) terminación, y que nos permita reconocer que una llamada a función dada está bajo modo protegido, para poder razonar correctamente sobre propagación de errores. A tal fin, utilizaremos la construcción $(\!| s |\!)_{\text{PROTMD}}$, junto con el contexto $(\!| E |\!)_{\text{PROTMD}}$, adicionado a la categoría E de contextos de evaluación, para permitirnos modelar la ejecución de una llamada a función en modo protegido. $(\!| E |\!)^{v}_{\text{PROTMD}}$ representará un modo protegido en donde también se ha definido un manejador v (a través del servicio xpcall). La propagación de un error será decidida en términos de la presencia o no de un modo protegido. Para poder referirnos a la *ausencia de un modo protegido*, utilizaremos una nueva categoría de contextos que, simplemente, no contienen modos protegidos. Denotaremos este conjunto con el no terminal *Enp*.

$$
\begin{aligned}
\delta(\text{xpcall}, v_1, v_2, v_3, \ldots) &= (\!| v_1\ (v_3, \ldots) |\!)^{v_2}_{\text{PROTMD}} \\
\delta(\text{pcall}, v_1, v_2, \ldots) &= (\!| v_1\ (v_2, \ldots) |\!)_{\text{PROTMD}} \\
\delta(\text{error}, v) &= \textbf{\$err}\ v
\end{aligned}
$$

Figura 4.50: Interpretación de servicios de librería asociados a errores y su manejo.

Introduciremos también la correspondiente interpretación, por medio de δ, de los servicios error, pcall and xpcall. Las correspondientes interpretaciones serán directas: simplemente generarán un nuevo error, o colocarán a la función llamada dentro de un modo protegido (Figura 4.50). Notar que, a diferencia de otros servicios como los relacionados a meta-tablas, xpcall y pcall no realizan chequeos de tipos en tiempo de ejecución sobre sus argumentos. En el caso de la función que se debe llamar en modo protegido, sabemos que es posible definir qué significa esta operación (la llamada), sobre valores que no son clausuras. Con respecto al manejador de errores que podemos definir mediante xpcall, el manual de referencia no especifica si, parte de la semántica del servicio, involucra el chequear o no su tipo, antes de llamar a la correspondiente función en modo protegido. La implementación oficial no chequea esto, mientras que LuaJIT sí. Nosotros seguiremos a la implementación oficial. Viendo las definiciones de Figura 4.50, notar que cualquier error por causa de tipos no compatibles con la semántica de la operación que se está realizando, será capturado por el modo protegido.

Al formalizar la propagación de un error tenemos que tener en cuenta que se trata de un comportamiento con semántica sensible al contexto, que va a requerir que contemplemos la totalidad de lo que resta por computar, para decidir sobre la terminación del programa o el manejo del error a través de un modo protegido. Si procediéramos del mismo modo que con las construcciones previas del lenguaje, esto es, proveyendo una regla que explica el cómputo (la propagación del error) y,

luego, permitiendo que ese cómputo ocurra en cierto orden, pero en cualquier lugar de un programa, al embeber la regla en $\mapsto$ mediante contextos de evaluación, terminaríamos obteniendo una relación $\mapsto$ que no se comportaría como función. Esto es, obtendríamos una semántica no determinista.

Para comprender por qué introduciríamos no determinismo procediendo del modo descrito, observemos que un programa de la forma $E[\![\textbf{\$err } v]\!]$, donde E no contiene un modo protegido, podría ser descompuesto de varias maneras, entre contexto y redex, dependiendo del nivel de anidamiento de E. Para cada una de tales descomposiciones, el redex en cuestión siempre reduciría a **$err** v. Esto es, su semántica operacional siempre descartaría lo que resta por computar, propagando de ese modo el error. Luego, con respecto al programa completo, existirían múltiples maneras de explicar la propagación del error. Sin embargo, lo que queremos para el programa $E[\![\textbf{\$err } v]\!]$ es que se detenga completamente en solo un paso, culminando en **$err** v.

La solución estándar para esta situación consiste en definir la propagación de errores directamente desde la relación $\mapsto$. Presentamos esto en la Figura 4.51.

$$\frac{Enp \neq [\![\ ]\!]}{\sigma : \theta : Enp[\![\textbf{\$err } v]\!] \ \mapsto \ \sigma : \theta : \textbf{\$err } v}$$

$$(\!| \ Enp[\![\textbf{\$err } v]\!] \ |\!)_{\text{PROTMD}} \rightarrow^{s/e} \langle \textbf{false}, v \rangle$$

$$\frac{v_2 \in functiondef}{(\!| \ Enp[\![\textbf{\$err } v_1]\!] \ |\!)^{v_2}_{\text{PROTMD}} \rightarrow^{s/e} \ v_2(v_1)}$$

$$\frac{v_2 \notin functiondef}{(\!| \ Enp[\![\textbf{\$err } v_1]\!] \ |\!)^{v_2}_{\text{PROTMD}} \rightarrow^{s/e} (\!| \ Enp \ [\![\textbf{\$err } \text{“}error\text{”}]\!] \ |\!)_{\text{PROTMD}}}$$

$$(\!| \ < v, \ldots > \ |\!)_{\text{PROTMD}} \rightarrow^{s/e} < \textbf{true}, v, \ldots >$$

Figura 4.51: Propagación y manejo de errores.

La primer regla aborta el programa completo, si el error no ocurre dentro de un modo protegido (pedimos por $Enp \neq [\![\ ]\!]$ para que esta regla pueda aplicarse, a lo sumo, una vez). Las reglas restantes solamente descartan lo que resta por computar hasta la ocurrencia del modo protegido más interno. El comportamiento de xpcall se presenta en la tercer y cuarta regla, en donde la información asociada al error es pasada al manejador v_2 o, en caso de no haber recibido una clausura como manejador, la operación descarta la información original asociada al error y lanza un nuevo error informando sobre lo ocurrido con el manejador, también dentro de un modo protegido.

Lo que resta por formalizar es la terminación normal de una función ejecutada en modo protegido. Esto se muestra en la última regla. Notar que estamos intentando explicar cómo culmina una ejecución de función que retornó una cierta lista de valores, representados con una tupla. La semántica del servicio indica que, en este caso, debemos retornar el valor **true**, seguido de la lista de valores retornada por la función.

Las reglas correspondientes a la terminación de modos protegidos instalados mediante xpcall son análogas.

4.3. Propiedades de la semántica

En esta sección vamos a precisar algunos enunciados sobre propiedades deseables de la semántica dinámica presentada hasta el momento. En particular, nos interesará estudiar la propiedad de progreso de nuestra semántica, la cual será capturada formalmente utilizando recursos tradicionales.

A los fines de obtener evidencia de las propiedades deseables del modelo, disponemos de, al menos, 3 enfoques diferentes: la demostración matemática usual *en papel*, con la riqueza conceptual tradicionalmente utilizada en ciencias de la computación, pero con las naturales limitaciones en su alcance (en particular, limitaciones relativas al *tamaño* del objeto matemático que podemos estudiar); la utilización de un asistente de pruebas, que puede permitirnos enfrentar el problema de demostrar rigurosamente propiedades sobre conceptos más complejos y extensos (como nuestra semántica de un lenguaje de programación real, por ejemplo), pero que introduce el problema de las posibles limitaciones conceptuales relativas a la lógica utilizada en el asistente, junto con el problema (en nuestra situación) de la reescritura, en el asistente de pruebas, de un modelo ya mecanizado en otra herramienta, cuando esta no ofrece facilidades para exportar el modelo; finalmente, existe un tercer enfoque, explorado en Redex [29] , que consiste en la obtención de evidencia de los enunciados de interés, utilizando facilidades de *random testing* o de *test aleatorio*, junto con amplias facilidades para definir relaciones semánticas y de tipado. Naturalmente, se trata de un enfoque a emplear solamente como paso previo a métodos más rigurosos, como puede ser la mecanización de la semántica en una asistente de prueba.

[29]: Klein y col. (2012), «Run Your Research: On the Effectiveness of Lightweight Mechanization»

Los enunciados mencionados en este apartado serán explorados utilizando el enfoque Redex descrito. Utilizaremos, parcialmente, el enfoque matemático tradicional para los desarrollos presentados en §5.2. En §8 discutiremos sobre las posibilidades del segundo enfoque descrito, en el contexto del presente trabajo.

En esta sección recogeremos las definiciones y enunciados pertinentes. Los detalles de la experiencia con el test aleatorio de los mismos, en nuestra mecanización, serán presentados en §6.

4.3.1. Chequeo de sanidad de la semántica: propiedad de progreso

$$C ::= [\![\ ]\!] \mid \textbf{while}\ e\ \textbf{then}\ C \mid \textbf{local}\ x\ , \ldots = e\ , \ldots\ \textbf{in}\ C \mid \ldots$$

Figura 4.52: Categoría C de contextos.

Desarrollaremos las correspondientes nociones de buena formación de las configuraciones de nuestra semántica, junto con el enunciado apropiado de *progreso* para nuestra semántica. Adaptaremos ideas tradicionales de semántica estática [30] , para el contexto de nuestro lenguaje dinámico. En nuestro caso, capturaremos buena formación mediante una relación $\vdash_{wfc} \subseteq C \times \sigma \times \theta \times t$, para C, la clase de contextos presentados en la Figura 4.52 (esto es, que contienen la información contextual necesaria para decidir sobre buena formación de términos), y $t = e \cup s$. No solamente descartaremos programas Lua mal formados, sino que también descartaremos términos de tiempo de ejecución mal formados. Capturaremos $\vdash_{wfc}$ mediante un sistema formal, cuya propiedad de corrección implicará la deseada propiedad de progreso de nuestra semántica. Por otro lado, Redex ofrece facilidades para mecanizar sistemas formales, mientras que, a su vez, la propiedad de corrección de los mismos consiste en un enunciado que es posible de someter a un test aleatorio, algo que no sería posible para, por ejemplo,

[30]: Harper (2016), *Practical Foundations for Programming Languages*

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad C \vdash_{wft} \sigma : \theta : v_3 \qquad v_1 \in dom(\theta) \qquad v_2 \notin dom(\theta(v_1)(1))}{C \vdash_{wft} \sigma : \theta : (\!| v_1[v_2] = v_3 |\!)_{\text{WRONGKEY}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad C \vdash_{wft} \sigma : \theta : v_3 \qquad v_1 \neq tid}{C \vdash_{wft} \sigma : \theta : (\!| v_1[v_2] = v_3 |\!)_{\text{NONTABLE}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \ldots \qquad v_1 \notin functiondef}{C \vdash_{wft} \sigma : \theta : (\!| v_1\ (v_2, \ldots) |\!)_{\text{WRONGFUNCALL}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e \ldots \qquad C[\![\textbf{local } x = e, \ldots \textbf{ in } [\![ ]\!]]\!] \vdash_{wft} \sigma : \theta : s}{C \vdash_{wft} \sigma : \theta : \textbf{local } x = e, \ldots \textbf{ in } s}$$

Figura 4.53: Reglas de inferencia para capturar $\vdash_{wft}$.

el enunciado estándar de progreso (a causa del caso de programas que no concluyen).

La noción de buena formación, para una configuración $\sigma : \theta : s$, incluye restricciones sensibles al contexto sobre s, que no pueden ser capturadas por una gramática libre de contexto (o al menos no de manera sencilla), junto con condiciones de buena formación sobre σ y θ. A los fines de simplificar la definición de $\vdash_{wfc}$, definiremos, también con sistemas formales, relaciones $\vdash_{wft}$, $\vdash_{wf\sigma}$ y $\vdash_{wf\theta}$, para validar la buena formación de términos y almacenamientos, respectivamente.

La figura 4.53 muestra alguna de las reglas de inferencia del sistema formal con el que capturamos $\vdash_{wft}$, necesarias para demostrar que un término dado está bien formado, con respecto a un contexto C y la información contenido en almacenamientos bien formados dados. Los primeros 3 casos consisten en reglas de inferencia para demostrar la buena formación de una asignación en campo de tabla que ha sido etiquetada, y una llamada a función, también etiquetada. El último caso muestra la descomposición y la reconstrucción necesarias, de término y contexto, para preservar información contextual útil para determinar la buena formación de una definición de variable local. Por simplicidad, no incluimos una nueva componente exclusivamente para preservar información sobre los identificadores de variables que están en scope. Simplemente reutilizamos la noción de contextos y las operaciones sobre los mismos, para determinar cuándo la ocurrencia de un identificador está ligada, como también para verificar las restantes restricciones sensibles al contexto. Las definiciones completas de $\vdash_{wft}$, y de las demás relaciones, se encuentran en el apéndice §B.

$C \vdash_{wfc} \sigma : \theta : t$ significará que, teniendo en cuenta el contexto C, entonces σ, θ y t están bien formados. Luego, una configuración $\sigma : \theta : s$ estará bien formada sí y sólo si $[\![ ]\!] \vdash_{wfc} \sigma : \theta : s$ (cuando $C = [\![ ]\!]$ simplemente escribiremos $\vdash_{wfc} \sigma : \theta : s$). Dado que nos interesa verificar la coherencia entre $\vdash_{wfc}$ y la semántica dinámica expresada por $\mapsto$, nos interesará verificar qué ocurre para una configuración $\sigma : \theta : s$ para la que podemos demostrar $\vdash_{wft} \sigma : \theta : s$. Esto es,

nos concentraremos en estudiar qué ocurre con las sentencias bien formadas y no con cualquier término, expresiones incluidas.

Como ya se mencionó, el sistema formal con el que capturamos $\vdash_{wfc}$ (y las demás relaciones) ha sido mecanizado con Redex, y su propiedad de corrección se ha estudiado realizando random testing. Detallamos la experiencia en §6. Teniendo en cuenta los resultados obtenidos con la mecanización podemos realizar el siguiente *enunciado* sobre $\vdash_{wfc}$ (en donde, en su enunciado, asumiremos definida la noción de *encaje de un término t con un patrón p*, denotada como $\vdash_{mtch}\ t \approx p$):

Enunciado 4.3.1 (Corrección de $\vdash_{wfc}$) *Para almacenamientos σ, θ, sentencia s, si $\vdash_{wfc}\ \sigma : \theta : s$, entonces vale una de las posibilidades siguientes:*

- *s representa el final de un cómputo. Esto es, s tiene alguna de las siguientes formas:*
  - *$err v*
  - *$E[\![$return $v, ...]\!]$, donde $\nvdash_{mtch}\ E \approx E'[\!(\!\ E_{lf}[\![\ ]\!]\ )\!\!)_{label}]\!]$, con label $\notin$ {RETSTAT, RETEXP, BREAK}*
  - *; (sentencia skip)*
- *Existe una configuración $\sigma' : \theta' : s'$ tal que $\sigma : \theta : s \mapsto \sigma' : \theta' : s'$ y $\vdash_{wfc}\ \sigma' : \theta' : s'$.*

La condición impuesta sobre el contexto de evaluación E, para el caso de una computación convergente que concluye en $E[\![$**return** $v, ...]\!]$, implica que la sentencia **return** ocurre fuera de una definición de función: es el resultado retornado por el programa, el cual, por ejemplo, será recibido en la aplicación *host* dentro de la cual está embebido el programa Lua.

Al realizar el testeo aleatorio del enunciado anterior, aprovechamos también para estudiar si la semántica definida tiene comportamiento determinista. En el contexto de nuestra semántica operacional, la noción de determinismo toma la forma expresada en el siguiente enunciado, que realizamos en base a la mencionada experiencia con la mecanización:

Enunciado 4.3.2 (Descomposición única) *Para almacenamientos σ, θ, sentencia s, tales que $\vdash_{wfc}\ \sigma : \theta : s$, vale una de las posibilidades siguientes:*

- *s representa el final de un cómputo.*
- *Existe un único contexto de evaluación E y término t, tal que s encaja con el patrón $E[\![t]\!]$, de forma tal de que sólo uno de los siguientes es un redex:*
  - *t*
  - *$\sigma : t$*
  - *$\theta : t$*

Esto es, si s no es un término final entonces puede descomponerse entre un único contexto de evaluación y término t, y sólo una de las reglas que definen a $\mapsto$ vale sobre t (junto, quizás, con los correspondientes almacenamientos).

Aceptando la validez de los enunciados anteriores, podemos deducir el siguiente corolario:

Corolario 4.3.3 *Para una configuración* $\sigma : \theta : \mathbf{s}$ *tal que* $\vdash_{wfc} \sigma : \theta : \mathbf{s}$ *solo vale uno de los siguientes enunciados:*

- *La ejecución de* $\sigma : \theta : \mathbf{s}$ *diverge, denotado con* $\sigma : \theta : \mathbf{s} \Uparrow$.
- *La ejecución concluye en una configuración* $\sigma' : \theta' : \mathbf{s}'$*, denotado como* $\sigma : \theta : s \Downarrow \sigma' : \theta' : \mathbf{s}'$*, donde* $\mathbf{s}'$ *solo puede tener una de las siguientes formas:*
  - $\mathbf{\$err}\ v$
  - $E[\![\mathbf{return}\ v, ...]\!]$*, donde* $\nvdash_{mtch} E \approx E'[\![(\!| E_{lf}[\![\ ]\!] |\!)_{label}]\!]$*, con* $label \notin \{\text{RETSTAT}, \text{RETEXP}, \text{BREAK}\}$
  - $\mathbf{;}$

Esto es, la noción de buena formación de configuraciones nos ayuda a descartar programas *stuck*, y solo restan, a lo sumo, programas erróneos que son identificados como tales por la semántica dinámica.

Finalmente, un corolario sencillo del enunciado 4.3.2, el cual nos indica que nuestra representación de clausuras, introducida en §4.2.3, es correcta::

Corolario 4.3.4 *Para un término cerrado* $\mathbf{s}$*, configuración* $\sigma : \theta : \mathbf{s}$ *tal que* $\vdash_{wfc} \sigma : \theta : \mathbf{s}$*, si* $\sigma : \theta : \mathbf{s} \mapsto \sigma' : \theta' : \mathbf{s}'$ *entonces* $\mathbf{s}'$ *también es un término cerrado.*

Lo anterior implica que, una vez que el foco de ejecución alcanzó a la definición de una función (es decir, el próximo redex incluye a una definición de función), toda variable libre en el cuerpo de la función ha sido reemplazada por su correspondiente referencia, conteniendo así, embebido en el cuerpo de la definición de la función, una representación completa del entorno de la clausura.

Recolección de basura 5

Lua 5.2 implementa 2 algoritmos de recolección de basura (GC) basada en alcanzabilidad[74]: un recolector *mark-and-sweep* (que funciona por defecto) y un recolector *generacional*[75]. Quien programa puede cambiar el algoritmo de GC mediante el servicio de librería collectgarbage.

Independientemente del recolector que se esté utilizando, GC es, en principio, transparente para quien programa. Sin embargo, Lua 5.2 incluye interfaces con el recolector de basura, las cuales introducen no-determinismo en la ejecución de los programas, debido a la semántica de estas interfaces. Estas sirven como mecanismo de control refinado de los recursos que utiliza un programa, y, para Lua 5.2, consisten en *finalizadores* y *tablas débiles*. Los primeros consisten en código que se ejecuta antes de que el recolector reclame la memoria de un objeto que ya no es alcanzable; en concreto toman la forma de funciones que son llamadas cuando un elemento (tabla o userdata) se convierte en basura, siendo posible reclamar la memoria en donde está alojado. El rol del finalizador es el de implementar una rutina que ayude con la devolución o eliminación apropiada de recursos utilizados por el programa. Las tablas débiles consisten en tablas ordinarias, pero cuya información (ya sean claves y/o valores) es referida mediante *referencias débiles*, es decir, referencias que no son tenidas en cuenta por el recolector de basura al momento de determinar qué porción del almacenamiento es alcanzable desde el programa. Esto a diferencia de las referencias *fuertes*, o referencias ordinarias.

En las próximas secciones veremos que, por la forma en la que estas interfaces operan, el comportamiento no-determinista del recolector de basura se torna observable desde el programa que las utiliza. Comenzaremos extendiendo nuestro modelo con una especificación del comportamiento de un recolector de basura sintáctico o basado en alcanzabilidad (como el presente en Lua y en la mayoría de los lenguajes de programación con recolección de basura), para luego agregar las interfaces al recolector ofrecidas por Lua. Finalmente, desarrollamos un marco teórico, inspirado en [31] , para razonar sobre propiedades deseables de GC sintáctica. Dentro del mismo verificamos varios lemas que sirven de chequeo de sanidad de nuestro modelo. La formalización aquí presentada también ha sido trasladada a nuestra mecanización en Redex, permitiendo ejecutar programas reales e inspeccionar cómo GC influye en la ejecución de los mismos.

Nuestra formalización describe de manera explícita algunos aspectos de GC en Lua no cubiertos por el manual de referencia, pero que tienen un efecto observable desde los programas: cual es el criterio de alcanzabilidad utilizado en Lua; cual es la semántica de instalación de finalizadores y de finalización, en general. Se trata de aspectos que, aunque tienen un efecto observable en la ejecución de los programas,

74: Consiste en considerar la imposibilidad de acceder a una región de la memoria, a través de los punteros que contiene el programa, como criterio para decidir qué información ya no es requerida por un programa y se puede, por lo tanto, reutilizar la memoria en donde se aloja esta información

75: Ver www.lua.org/manual/5.2/manual.html#2.5

[31]: Morrisett y col. (1995), «Abstract models of memory management»

estos no pueden inferirse directamente desde el manual de referencia, sino que requieren de experimentar con algún intérprete.

El resto del capítulo está organizado del siguiente modo: primero introducimos GC sintáctica en nuestro modelo, mediante una especificación abstracta de un paso de GC genérico, que es satisfecho por cualquier recolector basado en alcanzabilidad. Sobre esto, podremos agregar las correspondientes interfaces con el recolector: primero finalizadores, en §5.1.2, y luego tablas débiles, §5.1.3. Finalmente, en §5.2 presentaremos un desarrollo teórico que captura las propiedades fundamentales de nuestra especificación de GC sintáctica, sin interfaces con el recolector, y algunos enunciados sobre GC con interfaces. En §6 presentaremos los detalles relativos a la mecanización de este modelo, y en §8 discutiremos sobre las posibles aplicaciones de nuestro modelo formal a la resolución de problemas prácticos de programación con Lua.

5.1. Recolección de basura sintáctica

El propósito de GC es el de reclamar memoria en la que está almacenada información que no va a ser utilizada por los cómputos restantes del programa. Uno de los criterios más simples, y comúnmente utilizado, para detectar tales piezas de información, se basa en la noción de *alcanzabilidad*. La idea es sencilla: dado un conjunto de referencias que ocurren literalmente en el programa, el *conjunto raíz* o *root set* de referencias, debe darse el caso en el que cualquier pieza de información en memoria que vaya a utilizar el programa debe ser alcanzable desde ese conjunto raíz. Luego, un *binding* (una referencia a la memoria, junto con su valor referido) que no puede ser alcanzado desde el conjunto raíz, no puede ser alcanzado desde el programa[76] y, por lo tanto, puede ser removido sin que esto tenga un efecto observable en la ejecución del programa.

76: Situación que se modifica con la inclusión de las interfaces al recolector, que estudiaremos en este trabajo.

En el contexto de este trabajo, tanto un binding que no es alcanzable, como la referencia y el valor referido en sí, serán denominados *basura*. Esta idea sencilla, suficiente para especificar el comportamiento de GC en Lua, es puramente sintáctica: consistirá en tener en cuenta la ocurrencia literal de las referencias, dentro del programa, o su alcanzabilidad desde ese conjunto de referencias que ocurre literalmente. En contraste, existen otras maneras de identificar basura, en donde también se tiene en cuenta la semántica del programa [32] .

[32]: Kevin Donnelly (2006), «Formal semantics of weak references»

Durante el desarrollo de las próximas secciones, simplemente asumiremos la corrección del criterio empleado para caracterizar basura: en particular, asumiremos como cierto el que la remoción de los bindings que consideramos basura efectivamente no tiene un efecto observable en la ejecución de un programa, aunque no precisaremos formalmente esta idea intuitiva. En §5.2 daremos los primeros pasos necesarios en el estudio de las propiedades de nuestra formalización, de un modo que debiera ser lo suficientemente abstracto como para permitirnos razonar sobre diferentes estrategias de GC en Lua, en la medida en la que también estén basadas en el concepto de alcanzabilidad. En

este contexto, verificaremos nuestra suposición sobre el efecto de la remoción de basura.

Antes de comenzar a desarrollar las definiciones necesarias para explicar GC en Lua, nos será útil realizar una pequeña modificación al modelo presentado hasta aquí: en Lua, las tablas y las clausuras (dentro de los tipos incluidos en nuestro modelo) son manipuladas a través de referencias. Desde el punto de vista de la semántica dinámica de Lua sin GC, no es necesario incluir la manipulación de clausuras a través de referencias. Si bien es posible mantener esto al momento de explicar GC en Lua, veremos que al hablar de tablas débiles y su semántica, las definiciones a las que se arriba no resultan tan fáciles de comprender, si no se manejan referencias hacia las clausuras. Por tal motivo, trasladaremos clausuras hacia θ, introduciendo un nuevo conjunto de identificadores, *cid*, que usaremos para referir a clausuras (Figura 5.1). La semántica de clausuras es esencialmente la misma, con la única diferencia de que todas las operaciones sobre las mismas (llamadas a función, pasar clausuras como parámetros o devolverlas de una función, asignarlas a variables, etc), serán realizadas a través los identificadores *cid*, del mismo modo en el que ocurre con los identificadores de tables *tid* y las operaciones sobre tablas. Luego, en la sección §5.1.3 sobre tablas débiles, llamaremos la atención sobre esta modificación.

cid ::= identificadores de clausuras
v ::= ... | *cid*

$\theta \subseteq tid \cup cid \rightarrow tableconstructor \cup functiondef$

Figura 5.1: Clausuras manejadas a través de referencias.

5.1.1. GC sin interfaces

Para capturar formalmente la noción de basura, será útil comenzar por una definición formal de la noción de *referencias alcanzables*. Para el caso de Lua, nos basta manejar una definición de alcanzabilidad semejante a las halladas usualmente en la literatura[33, 34] , pero con una pequeña peculiaridad: las meta-tablas de una tabla alcanzable también son consideradas alcanzables, luego, un camino de alcanzabilidad[77] puede pasar a través de tales meta-tablas. Aunque es un detalle menor, no es algo que esté documentado en el manual de referencia y tiene, como es de esperar, un efecto observable.

[33]: Leal y col. (2005), «A Formal Semantics for Finalizers»
[34]: Gabay y col. (2007), «A Calculus for Java's Reference Objects»

77: Esto es, una sucesión de referencias alcanzables entre sí, comenzando desde el conjunto raíz.

Informalmente, una referencia (referencia a σ o a θ) será alcanzable con respecto a un término *t* (desde donde determinamos el conjunto raíz de referencias) y almacenamientos dados, si se da alguna de las siguientes situaciones:

- La referencia ocurre literalmente en *t*.
- La referencia es alcanzable desde la información asociada a otra referencia alcanzable. Esto es:
  - La referencia es alcanzable desde la clausura asociada a una referencia alcanzable.
  - La referencia es alcanzable desde la tabla asociada a una referencia alcanzable.
  - La referencia es alcanzable desde la meta-tabla asociada a un identificador de tabla alcanzable.

Formalizamos lo anterior en la siguiente definición:

Definición 5.1.1 (Alcanzabilidad para GC sin interfaces) *Decimos que una referencia l (referencia a σ o a θ) es* alcanzable *en una configuración $\sigma : \theta : t$ sí y sólo si:*

$$
\begin{aligned}
reach(l, t, \sigma, \theta) = {} & l \in t \vee \\
& (\exists r \in t, reach(l, \sigma(r), \sigma \setminus r, \theta)) \vee \\
& \exists\, tid \in t, reach(l, \pi_1(\theta(tid)), \sigma, \theta \setminus tid) \vee \\
& \qquad reach(l, \pi_2(\theta(tid)), \sigma, \theta \setminus tid) \vee \\
& \exists\, cid \in t, reach(l, \theta(cid), \sigma, \theta \setminus cid)
\end{aligned}
$$

Escribimos $l \in t$ para expresar que l ocurre literalmente en t (es decir, l está en el conjunto raíz) y denotamos con $\gamma \setminus l$ al resultado de remover del almacenamiento γ solamente el binding de la referencia l.

Para evitar los ciclos generados por definiciones mutuamente recursivas, en el al almacenamiento, que tornarían no definido el predicado anterior, removemos del almacenamiento los bindings ya considerados en la construcción del camino de alcanzabilidad. *reach* está especificando la existencia de un tal camino [78]. Asumimos el predicado como falso si una referencia dada ocurre en t pero no pertenece al dominio del correspondiente almacenamiento.

78: Notar que el predicado *reach* tiene, también, la siguiente interpretación computacional: en cada paso intenta avanzar a través de árboles que tienen como raíz a una referencia del root set; como nodos internos de segundo nivel a aquellas referencias alcanzables desde el nodo raíz mediante una operación de desreferenciado; en el siguiente nivel, las referencias alcanzables en 2 operaciones de desreferenciado; etc. A veces nos referiremos al mismo como el *árbol de alcanzabilidad*, y servirá como base para capturar predicados más complejos de alcanzabilidad, como menores puntos fijos de dominios de funciones que expanden, a lo sumo, n niveles de un árbol de alcanzabilidad, mediante las correspondientes operaciones de desreferenciado.

Notar que para una tabla *tid* chequeamos su contenido ($\pi_1(\theta(tid))$) y su meta-tabla ($\pi_2(\theta(tid))$). Esto es, las meta-tablas de tablas alcanzables también son consideradas al determinar alcanzabilidad. Notar también que, siendo que las meta-tablas son tablas regulares, pueden contener otros identificadores de tablas o clausuras, quienes, a su vez, pueden contener otras referencias. Naturalmente, si no se tuvieran en cuenta también las meta-tablas, al determinar alcanzabilidad, correríamos el riesgo de que se generen errores de punteros colgantes en nuestros programas al intentar acceder a manejadores de eventos contenidos en las meta-tablas. También notar que, en la llamada recursiva $reach(l, \pi_2(\theta(tid)), \sigma, \theta \setminus tid)$ se verificará si l es exactamente $\pi_2(\theta(tid))$ (pues se determinará si $l \in \pi_2(\theta(tid))$, para $\pi_2(\theta(tid))$ siendo **nil** o un identificador de tabla), y, si no, se continuará con la inspección del contenido de la meta-tabla. Para que esto sea posible, no removemos $\pi_2(\theta(tid))$ de θ, en la mencionada llamada recursiva.

El último disyunto verifica alcanzabilidad siguiendo un identificador de clausura *cid* presente en el conjunto raíz de referencias. Lo que necesitamos es expandir el árbol de alcanzabilidad siguiendo el entorno de la clausura (es decir, siguiendo a el mapeo entre los identificadores de variables externas presentes en el cuerpo de la clausura, y sus correspondientes referencias). Como hemos visto, nuestra representación de clausuras consisten, simplemente, en definiciones de funciones en cuyos cuerpos están embebidas las correspondientes referencias de las variables externas. Luego, avanzar a través del árbol de alcanzabilidad, considerando el entorno de una clausura alcanzable, se reduce a buscar por ocurrencias literales de referencias dentro del cuerpo de la clausura.

Finalmente, una observación menor: naturalmente, el manual de referencia no especifica detalles sobre GC que no se debieran tener en cuenta al momento de razonar sobre programas. Aún así, el hecho

de que las meta-tablas son consideradas al determinar alcanzabilidad, consiste en una particularidad menor de Lua pero que tiene, naturalmente, un efecto observable en los programas y es inherente a GC basada en alcanzabilidad en Lua, independientemente de estrategias particulares utilizadas por el recolector implementado. En general, este será el criterio que seguiremos al momento de decidir incluir en nuestro modelo algún aspecto de GC de Lua.

Especificación de un ciclo de GC A continuación, procederemos a incluir en el modelo un paso de GC capturado a fuerza de, simplemente, especificar abstractamente la propiedad elemental que satisface todo recolector basado en alcanzabilidad. Por un lado, es una adición razonable, pues el manual de referencia no especifica ningún algoritmo (más allá de mencionar que están implementados en el intérprete oficial y explicar de qué tipo son). Pero, por otro lado, se trata de una regla de GC que, al emular cualquier paso de un recolector sintáctico concebible, nos sirve para introducir gestión de memoria automática en nuestro modelo y estudiar sus efectos observables, si existen. Esto se desarrollará más en §5.2.

Comenzaremos especificando una meta-función, gc, que modela el efecto de un paso de GC sintáctica sobre una configuración dada:

Definición 5.1.2 $gc(\mathtt{s}, \sigma, \theta) = (\sigma_1, \theta_1)$, *donde:*

- $\sigma = \sigma_1 \uplus \sigma_2$
- $\theta = \theta_1 \uplus \theta_2$
- $\forall l \in dom(\sigma_2) \cup dom(\theta_2), \neg reach(l, \mathtt{s}, \sigma, \theta)$

$\gamma_1 \uplus \gamma_2$ denotará la unión de almacenamientos con dominio disjunto. Esta especificación indica, simplemente, que $gc(\mathtt{s}, \sigma, \theta)$ retorna nuevos almacenamientos σ_1 y θ_1, los cuales son un subconjunto de los almacenamientos σ y θ provistos como argumentos. No especificamos cómo es que se obtuvieron esos subconjuntos, solamente exigimos que los subconjuntos restantes (σ_2 y θ_2) no contengan referencias alcanzables, desde el conjunto raíz, definido por $\mathtt{s}$. Satisfecha esta condición, es seguro correr el programa $\mathtt{s}$ en los nuevos almacenamientos σ_1 y θ_1, ya que no puede ocurrir una operación de desreferenciado sobre un puntero colgante, a causa de GC. Notar que puede darse el caso en el que σ_1 y θ_1 también contengan referencias no alcanzables, de acuerdo al root set definido por $\mathtt{s}$.

Utilizando la meta-función gc, podemos extender nuestro modelo con un paso no-determinista de GC que modelaremos con una nueva relación, $\overset{GC}{\mapsto}$:

$$\frac{(\sigma', \theta') = gc(\mathtt{s}, \sigma, \theta) \qquad \sigma' \neq \sigma \vee \theta' \neq \theta}{\sigma : \theta : s \overset{GC}{\mapsto} \sigma' : \theta' : s}$$

Exigimos para el mismo que, efectivamente, remueva algún binding de alguno de los almacenamientos, para asegurarnos de que este paso no pueda ser ejecutado una cantidad potencialmente infinita de

veces. Esta regla introduce, evidentemente, no-determinismo en la ejecución de nuestros programas: en cualquier momento, siempre que haya bindings no alcanzables, es posible elegir entre recolectar los mismos o continuar con la ejecución del programa. Pero, con respecto a la definición provista hasta el momento, este no-determinismo no debiera generar efectos visibles desde un programa. Estudiaremos formalmente esto en §5.2. Como se mencionó, esta propiedad no seguirá siendo cierta tras la inclusión de las interfaces con el recolector, en las siguientes secciones.

5.1.2. Finalizadores

Lua 5.2 incluye finalizadores, un mecanismo útil para ayudar en la liberación adecuada de recursos externos utilizados por un programa. Los finalizadores son definidos, por quien programa, mediante funciones que serán invocadas por el recolector una vez que un objeto[79] (tabla o userdata), *marcado* para finalización, se torne no accesible. Notar que debido a que los finalizadores son llamados por el recolector no es posible determinar el momento preciso en el que ocurre la finalización. Esto en contraste con, por ejemplo, el concepto de *destructores*, presentes en lenguajes con mecanismos deterministas de administración de memoria (por ejemplo, C++).

79: Aquí también utilizamos vocabulario del glosario de Lua: www.lua.org/manual/5.2/manual.html#2.1.

En la literatura del área [33, 35, 36] se identifican varios problemas que resultan de la mala utilización de los finalizadores, relacionados con el hecho de que son invocados de un modo no-determinista, trasladando el mismo a la ejecución de un programa, con efectos observables. Para mitigar algunos de estos problemas, la implementación en Lua 5.2 de este mecanismo ofrece algunas garantías respecto al orden de ejecución entre los distintos finalizadores instalados, y otorga un tratamiento especial a objetos *resucitados*.

[35]: B. (1992), «Finalization in the collector interface»
[36]: Boehm (2003), «Destructors, Finalizers, and Synchronization»
[33]: Leal y col. (2005), «A Formal Semantics for Finalizers»

Finalización en Lua

Comenzaremos con una presentación informal del mecanismo de finalización implementado en Lua 5.2. Luego, explicaremos cómo extender nuestro modelo de la semántica dinámica de Lua, para incluir esta interfaz con el recolector.

Definición de un finalizador El finalizador de un objeto consiste en una función almacenada en la meta-tabla del objeto, asociada con la clave "__gc". Para que el finalizador sea efectivamente invocado, la clave debe estar presente en la meta-tabla al momento de instalarla (definirla como meta-tabla mediante el servicio setmetatable, presentado en §4.2.5). En tal caso, se dirá que el objeto ha sido *marcado* para finalización. El código presentado en la Figura 5.2 muestra esta semántica: cuando para la tabla a se instala una meta-tabla vacía (b, en la línea 2), inclusive si a posterior se define un campo con clave "__gc" en b (linea 3), la tabla a es recolectada sin efectos observables (línea 7).

Luego, si ensayamos asociar b (que ahora sí posee un campo con clave "__gc") como meta-tabla de una nueva tabla (linea 9), esta tabla será correctamente marcada para finalización (linea 14). También, si el valor asociado a la clave "__gc" no es una función, el recolector ignora el error, sin efectos observables (lineas 15 a 19).

Finalmente, el valor asociado a la clave "__gc" puede ser alterado, entre el momento en el que la meta-tabla es instalada y el momento en el que el objeto es finalizado. El último finalizador instalado es el que será invocado.

Orden de ejecución de los finalizadores Lua ofrece una garantía con respecto al orden de ejecución de los finalizadores: el orden será *cronológico*, es decir, determinado por el momento en el que instalamos un finalizador dado; a su vez, el orden es inverso al del momento en el que definimos los finalizadores.

Explicamos el orden anterior con el código presentado en la Figura 5.3. Este código realiza las siguientes acciones: 1) crea 2 tablas, a y b; 2) les asocia la tabla c como meta-tabla, dentro de la cual define un finalizador que imprime en consola la referencia de la tabla que es finalizada; primero lo hace para a, luego para b; 3) elimina cualquier referencia hacia las tablas (para tornarlas recolectables) e invoca al recolector.

En las líneas 17–20 mostramos la información impresa en consola tras la ejecución del programa anterior, en donde se observa el orden cronológico descripto.

Resurrección Durante la finalización de un objeto, su referencia es pasada al finalizador (por ejemplo, parámetro formal o, del finalizador mostrado en la Figura 5.3), tornando nuevamente alcanzable al objeto que se está finalizando. Este fenómeno se conoce como *resurrección local*, o simplemente *resurrección*, y, normalmente, es transitorio. Sin embargo, existe la posibilidad de que el finalizador torne en permanente esta resurrección, por ejemplo, asociando el objeto a una variable global, haciendo que el objeto sea alcanzable luego de la ejecución del finalizador y evitando, de ese modo, que el objeto sea recolectado. A veces se denomina *resurrección global* a esta última situación.

Esta posibilidad introduce problemas en la implementación de los recolectores de basura, reduciendo su eficacia para reclamar memoria que no está en uso por el programa. A su vez, el fenómeno puede terminar reintroduciendo en el programa objetos que no satisfacen invariantes de representación.

Para mitigar los problemas que podrían ocurrir por causa de un orden no-determinista de ejecución entre finalizadores, Lua garantiza el ya mencionado orden cronológico de ejecución,[80]. Con respecto al problema de resurrección global, Lua 5.2 distingue de modo particular a los objetos ya finalizados: no permite que un objeto finalizado sea marcado para finalización nuevamente. De este modo, el finalizador

```
1  local a, b = {}, {}
2  setmetatable(a, b)
3  b.__gc = function ()
4               print ("bye")
5           end
6  a = nil
7  collectgarbage()
8  -- no hay efectos
9  -- observables
10
11 a = {}
12 setmetatable(a, b)
13 b.__gc = function ()
14              print ("goodbye")
15          end
16 a = nil
17 collectgarbage()  -- 'goodbye'
18 b.__gc = "not a function"
19 a = {}
20 setmetatable(a, b)
21 a = nil
22 collectgarbage()
23 -- no hay efectos
24 -- observables
25
26 a = {}
27 setmetatable(a, b)
28 b.__gc = function ()
29              print ("goodbye")
30          end
31 a = nil
32 collectgarbage()  -- 'goodbye'
```

Figura 5.2: Definición de un finalizador.

```
1  local a, b = {}, {}
2
3  local c = {}
4  c.__gc = function (o)
5               print ("goodbye",o)
6           end
7
8  print (a, b)
9  -- table: 0x5646c8100600
10 -- table: 0x5646c8100640
11
12 setmetatable(a, c)
13 setmetatable(b, c)
14
15 a, b = nil, nil
16 collectgarbage()
17 -- goodbye table:
18 -- 0x5646c8100640 (b)
19 -- goodbye table:
20 -- 0x5646c8100600 (a)
```

Figura 5.3: Orden cronológico de ejecución de los finalizadores.

80: Ver www.lua.org/manual/5.2/manual.html#2.5.1

de un objeto nunca será llamado más de una vez, evitando *objetos indestructibles*, los cuales, entre otras cosas, pueden agotar la memoria de la que dispone un programa sin que el recolector pueda intervenir. En este caso, el objeto resucitado será eliminado una vez que nuevamente se torne no alcanzable.

Semánticamente, esta es la única diferente que distingue a una objeto finalizado. En particular, es posible asociar al mismo una nueva meta-tabla, para configurar su comportamiento de acuerdo a los metamétodos de la misma, excepto __gc.

Manejo de errores Si un programa concluye normalmente, los finalizadores restantes que necesitan ser ejecutados, para liberar adecuadamente los recursos externos del programa, son invocados en *modo protegido* (ya introducido en §4.2.6). De este modo, cualquier error ocurrido durante la ejecución de un finalizador, solamente interrumpe al finalizador, permitiendo la llamada de los finalizadores restantes. A su vez, un finalizador que es interrumpido por una situación errónea no previene la recolección del correspondiente objeto que está siendo finalizado.

Por otro lado, durante la ejecución de un programa, cualquier error en un finalizador es propagado hacia el hilo principal de ejecución. Debido a que las llamadas a finalizadores son intercaladas con código del hilo principal de ejecución, cualquier error arrojado por un finalizador aparecerá en una posición del programa que no puede ser determinada de antemano. Si tal posición resultara ser dentro de una función llamada en modo protegido, el error será atrapado.

Modelo formal

Extenderemos nuestro modelo para incluir finalizadores: primero, asociaremos con las tablas información pertinente sobre finalización; luego, modificaremos el ciclo de GC definido en la sección anterior, para hacer que el recolectar tenga en cuenta este mecanismo, convirtiendo así a los finalizadores en una interfaz con el recolector.

$$\frac{\begin{array}{c}\forall\ 1 \le i, \mathit{field}_i \in v \\ \vee \\ \mathit{field}_i = \mathbf{[}\ v_1\ \mathbf{]} = v_2 \\ t = addkeys(\{\mathit{field}_1, ...\}) \\ \theta_2 = (\ \mathit{tid},\ (t,\ \mathbf{nil},\ \bot)),\ \theta_1\end{array}}{\theta_1\ :\ \{\mathit{field}_1, ...\}\ \rightarrow^{\theta}\ \theta_2\ :\ \mathit{tid}}$$

Figura 5.4: Constructor de tablas, incorporando información sobre finalización (notar $\bot$).

Modelado del orden de finalización Necesitaremos relacionar a las tablas con información sobre su posición respecto a el orden de finalización. Para expresar esta idea, simplemente utilizaremos ternas, en θ, en las que asociaremos a una tabla junto con su meta-tabla y esta pieza de información, su posición en el orden de finalización. Esta posición la representaremos con elementos de un conjunto $\mathcal{P}$, totalmente ordenado por una relación $<^{fin}$, o un elemento $\oslash$: un elemento $\bot \in \mathcal{P}$ indicará que la tabla no ha sido marcada para finalización; necesitaremos que $\bot$ sea mínimo en $\mathcal{P}$, con respecto a $<^{fin}$; $\oslash$ indicará que la tabla no puede ser marcada para finalización (relacionado con la solución de Lua 5.2 para evitar objetos indestructibles, como se explicó al describir resurrección de objetos); finalmente, si una tabla posee un elemento $p \in \mathcal{P} \setminus \{\bot\}$ significará que la misma ha sido marcada para finalización y que tiene prioridad p, de acuerdo al orden $<^{fin}$. Una

tabla recién construida tendrá asociada el valor $\perp$. Modificaremos la semántica del constructor de tablas para reflejar esto (Figura 5.4).

En Lua 5.2, $<^{fin}$ está definida cronológicamente, esto es, teniendo en cuenta el momento en el que la tabla ha sido marcada para finalización (a través del servicio setmetatable). Para nuestra semántica será suficiente con una meta-función $\mathsf{next}_{<^{fin}}$, con signatura $\mathcal{P} \to \mathcal{P}$, que nos debiera proveer de un elemento de $\mathcal{P}$ mayor que el recibido como argumento, de acuerdo a $<^{fin}$. Cuando una meta-tabla es definida con la correspondiente invocación al servicio setmetatable, la posición de la tabla correspondiente, en el orden de finalización, será determinado mediante la meta-función set_fin, que recibirá un id de tabla *tid*, el valor pasado como meta-tabla en la llamada a setmetatable, y el almacenamiento θ. El valor pasado como meta-tabla, naturalmente, podrá ser **nil** u otro identificador de tabla tid_{meta} or **nil**. set_fin describirá la lógica detrás de la operación de marcar una tabla para finalización. Incluirá todos los detalles de esta operación que tienen un efecto observable, en la búsqueda de una semántica en conformidad con el manual de referencia de Lua y su principal implementación. No es necesario comprender todos los detalles de la siguiente definición, para comprender finalización:

$$set\_fin(tid, v, \theta) = \oslash, \text{ if } \pi_3(\theta(tid)) = \oslash \tag{5.1}$$

$$set\_fin(tid, \textbf{nil}, \theta) = \perp, \text{ if } \pi_3(\theta(tid)) \neq \oslash \tag{5.2}$$

$$set\_fin(tid, tid_{meta}, \theta) = \pi_3(\theta(tid)), \text{ if } \begin{cases} \pi_2(\theta(tid)) = tid_{meta} \\ \pi_3(\theta(tid)) \neq \oslash \end{cases} \tag{5.3}$$

$$set\_fin(tid, tid_{meta}, \theta) = \perp, \text{ if } \begin{cases} \text{“\_\_gc”} \notin \pi_1(\theta(tid_{meta})) \\ \pi_2(\theta(tid)) \neq tid_{meta} \\ \pi_3(\theta(tid)) \neq \oslash \end{cases} \tag{5.4}$$

$$set\_fin(tid, tid_{meta}, \theta) = next(p), \text{ if } \begin{cases} \text{“\_\_gc”} \in \pi_1(\theta(tid_{meta})) \\ \pi_2(\theta(tid)) \neq tid_{meta} \\ \pi_3(\theta(tid)) \neq \oslash \end{cases} \tag{5.5}$$

$$\text{where } p = max^{<^{fin}}(filter(map(\pi_3, img(\theta)), \lambda\ pos.pos \neq \oslash))$$

La primer ecuación codifica el significado de $\oslash$: sin importar el contenido de la meta-tabla, si la posición previa de la tabla es $\oslash$, entonces set_fin retorna $\oslash$, evitando que *tid* pueda ser nuevamente marcada para finalización. A partir de las siguientes ecuaciones, se asume que la posición de la tabla no es $\oslash$. La segunda ecuación especifica una de las situaciones en las que una tabla podría ser retirada del orden de finalización: si la meta-tabla es **nil**, y la posición previa de la tabla no es $\oslash$, entonces se retorna $\perp$. A partir de este momento, la tabla *tid* podría ser marcada para finalización. La tercer ecuación considera el caso en el que el valor pasado como meta-tabla, tid_{meta}, es el mismo que el que tenía asociada *tid* como meta-tabla. En tal caso, la posición de la tabla no se modifica. La cuarta ecuación considera el caso en el que meta-tabla que se intenta instalar no contiene definido el campo “__gc”: se quita a la tabla del orden de finalización, aunque se la podría volver a marcar para finalización; se devuelve $\perp$. La última ecuación

explica el caso en el que *tid* es efectivamente marcada para finalización: simplemente se retorna el valor $\mathsf{next}_{<^{fin}}(p)$, para p, la mayor posición de entre todas las tablas marcadas para finalización en θ. Notar que la expresión sobre la que calculamos $max^{<^{fin}}$ siempre contiene al menos un elemento: $\pi_3(\theta(\mathit{tid}))$. A su vez, en ningún momento existen tablas marcadas para finalización, con igual prioridad p.

Naturalmente, mediante el adecuada reemplazo de $<^{fin}$ (y la correspondiente modificación de $\mathsf{next}_{<^{fin}}$), es posible modelar otros ordenes de finalización. Por ejemplo, para implementar orden topológico , dados almacenamientos σ y θ, identificadores de tablas tid_1 y tid_2, es suficiente con considerar una nueva relación $\leq^{fin}$ definida como:

$$\pi_3(\theta(\mathit{tid}_1)) \leq^{fin} \pi_3(\theta(\mathit{tid}_2)) \Leftrightarrow reach(\mathit{tid}_1, \mathit{tid}_2, \sigma, \theta) \quad (6)$$

Se puede verificar que, si consideramos la relación ≡ definida del siguiente modo::

$$\mathit{tid}_1 \equiv \mathit{tid}_2 \Leftrightarrow reach(\mathit{tid}_1, \mathit{tid}_2, \sigma, \theta) \wedge reach(\mathit{tid}_2, \mathit{tid}_1, \sigma, \theta)$$

entonces $\leq^{fin}$ definida como en (6) es reflexiva, transitiva y antisimétrica (con respecto a ≡). Impondrá un orden topológico de finalización, y permitirá la remoción de tablas definidas de manera mutuamente recursiva, aunque de un modo no-determinista.

Retornando a nuestro modelo, solo nos resta definir la nueva semántica del servicio setmetatable. Contemplaremos el caso en el que efectivamente se llama al servicio con argumentos de tipo esperado. Simplemente notar que, ahora, se asocia a el valor $set\_fin(\mathit{tid}, v, \theta_1)$ con la tabla para cual se invoca setmetatable:

$$\delta(\mathsf{setmetatable}, \mathit{tid}, v, \theta_1) = (\mathit{tid}, \theta_2),$$
$$\text{if } \begin{cases} \delta(\mathsf{type}, v) \in \{\text{“}table\text{”}, \text{“}nil\text{”}\} \\ \neg prot?(\mathit{tid}, \theta_1) \\ \theta_2 = \theta_1[\mathit{tid} := (\theta_1(\mathit{tid})(1), v, set\_fin(\mathit{tid}, v, \theta_1))] \end{cases}$$

Especificación de un ciclo de GC con finalización Enriqueceremos la especificación sencilla del algoritmo de GC, propuesta en §5.1.1, para que ahora el recolector tenga en cuenta al mecanismo de finalización. Para realizar esto será útil abstraer en nuevos predicados restricciones que expresan propiedades simples que deben formar parte de la especificación.

Comenzaremos con una condición incluida para evitar generar errores de punteros colgantes, como causa de la ejecución de un finalizador: cualquier referencia alcanzable de una tabla que ha sido marcada para finalización, o alcanzable desde su meta-tabla, no debe ser removida antes que la mencionada tabla, independientemente de la alcanzabilidad de la tabla (recordar que una tabla que está por ser finalizada ya no es alcanzable). Para expresar que una cierta referencia *l* no es alcanzable desde una tabla marcada para finalización, o desde su meta-tabla,

dados almacenamientos σ y θ, usaremos los predicados presentados en Figura 5.5.

Finalmente, para expresar que una tabla, identificada por cierto id de tabla *tid*, está lista para ser finalizada (es decir, ya no es alcanzable y está marcada para finalización) y que debe ser la próxima tabla a ser finalizada, de acuerdo al orden impuesto por $\leq^{fin}$ (la clausura reflexiva de $<^{fin}$), utilizaremos los predicados presentados en la Figura 5.6.

Estamos en condiciones de imponer nuevas condiciones al ciclo de GC, para que también incluya al mecanismo de finalización. Presentamos la especificación en la Figura 5.7.

La nueva especificación del paso de GC retorna almacenamientos σ_1 y θ'_1, junto con un nuevo término t, que consistirá en el finalizador a ejecutar, si corresponde. La primer parte del predicado (identificado como *gc*) es el mismo que usamos para especificar GC sin interfaces, en §5.1.1, e indica que vamos a separar cada almacenamiento en 2 subconjuntos disjuntos, uno conteniendo los bindings que serán descartados (σ_2 y θ_2), y el otro conteniendo los bindings que se preservaran, y sobre los que continuará la ejecución del programa (σ_1 y θ_1). Pero ahora, impondremos condiciones adicionales para especificar los subconjuntos anteriores (condiciones identificadas como *fin*): primero, incluimos la condición not_reach_fin descrita previamente, para evitar errores de punteros colgantes, al ejecutar finalizadores. Notar que preguntamos por alcanzabilidad considerando la porción del almacenamiento que no será descartado. Segundo, cada tabla *tid* en θ_2 no debe estar marcada para finalización ($\neg marked(tid, \theta_2)$): parte de la semántica de finalizadores. De cumplirse ambas condiciones, los bindings de σ_2 y θ_2 pueden ser descartados, sin mayores consideraciones.

Las condiciones anteriores garantizan que sólo tablas que han sido ya finalizadas (o que nunca han sido marcadas para finalización) sean recolectadas, y evita errores de punteros colgantes, al ejecutar finalizadores. Las condiciones que incluimos a continuación caracterizan a la siguiente tabla que va a ser finalizada, si existe: si existe cierto $tid \in \theta_1$ tal que identifica a una tabla finalizable y es la próxima tabla a ser finalizada, de acuerdo a $\leq^{fin}$ (tal como lo expresan los predicados fin y $next\_fin$); y si, a su vez, existe el correspondiente finalizador instalado (una función identificada por *cid*, en el campo "__gc" de la correspondiente meta-tabla), entonces, la próxima instrucción a ejecutar será *cid* aplicado al identificador *tid* (resucitando a *tid*). A su vez, el almacenamiento a retornar será un θ'_1 que simplemente consiste en modificar, en θ_1, la información de finalización asociada a *tid*: ahora, se prohíbe que *tid* pueda ser marcada de nuevo para finalización; es decir, asociados a *tid* con $\oslash$. Notar que *tid* todavía se encuentra en el almacenamiento retornado θ'_1, de otro modo no podría ser accesible al finalizador. La tabla puede ser recolectada en un paso de GC siguiente, o no, dependiendo de lo que haga su finalizador. En caso de que el valor asociado a la clave "__gc" no sea una clausura, *t* es **nil**, pero también retiramos a la tabla del orden de finalización, para que pueda ser recolectada en un paso de GC posterior.

$$marked(\mathsf{tid}, \theta) \doteq \pi_3(\theta(\mathsf{tid})) \notin \{\bot, \oslash\}$$

$$not\_reach\_fin(l, \sigma, \theta) \doteq \nexists\, \mathsf{tid} \in dom(\theta),\ l \neq \mathsf{tid} \wedge marked(\mathsf{tid}, \theta) \wedge reach(l, \mathsf{tid}, \sigma, \theta)$$

Figura 5.5: Condiciones $marked$ y not_reach_fin.

$$fin(\mathsf{tid}, \mathsf{s}, \sigma, \theta) \doteq \neg reach(\mathsf{tid}, \mathsf{s}, \sigma, \theta) \wedge marked(\mathsf{tid}, \theta)$$

$$next\_fin(\mathsf{tid}, \mathsf{s}, \sigma, \theta) \doteq \forall \mathsf{tid}' \in dom(\theta),\ fin(\mathsf{tid}', \mathsf{s}, \sigma, \theta) \Rightarrow \pi_3(\theta(\mathsf{tid}')) \leq^{fin} \pi_3(\theta(\mathsf{tid}))$$

Figura 5.6: Condiciones fin y $next\_fin$.

$$gc_{fin}(\mathsf{s}, \sigma, \theta) = (\sigma_1, \theta'_1, t),\ gc \begin{cases} \sigma = \sigma_1 \uplus \sigma_2 \\ \theta = \theta_1 \uplus \theta_2 \\ \forall l \in dom(\sigma_2) \cup dom(\theta_2), \\ \quad \neg reach(l, \mathsf{s}, \sigma, \theta) \end{cases}$$

$$fin \begin{cases} \forall l \in dom(\sigma_2) \cup dom(\theta_2), \\ \quad not\_reach\_fin(l, \sigma_1, \theta_1) \\ \forall \mathsf{tid} \in dom(\theta_2), \\ \quad \neg marked(\mathsf{tid}, \theta_2) \\ [\exists \mathsf{tid} \in \mathsf{dom}(\theta_1), \\ \quad fin(\mathsf{tid}, \mathsf{s}, \sigma, \theta) \\ \quad next\_fin(\mathsf{tid}, \mathsf{s}, \sigma, \theta) \\ \quad \mathsf{v} = indexmetatable(\mathsf{tid}, \text{“}\_\_gc\text{”}, \theta_1) \\ \quad \mathsf{v} \in \mathsf{cid} \Rightarrow t = \mathsf{v}(\mathsf{tid}) \\ \quad \mathsf{v} \notin \mathsf{cid} \Rightarrow t = \mathbf{nil} \\ \quad \theta'_1 = \theta_1[\mathsf{tid} := (\pi_1(\theta_1(\mathsf{tid})), \pi_2(\theta_1(\mathsf{tid})), \oslash)] \\ \vee \\ \quad t = \mathbf{nil} \\ \quad \theta'_1 = \theta_1] \end{cases}$$

Figura 5.7: GC con finalización.

Finalmente, si ninguna de las condiciones anterior se cumplen, t es **nil** y $\theta'_1 = \theta_1$.

$$\frac{(\sigma', \theta', v(tid)) = gc_{fin}(\sigma, \theta, E[\![s]\!])}{\sigma : \theta : E[\![s]\!] \overset{F}{\mapsto} \sigma' : \theta' : E[\![v(tid); s]\!]}$$

$$\frac{\begin{array}{c}(\sigma', \theta', v(tid)) = gc_{fin}(\sigma, \theta, E[\![e_1]\!]) \\ e_2 = (\textbf{function}\ \$\ ()\ \textbf{return}\ e_1)\ (v(tid))\end{array}}{\sigma : \theta : E[\![e_1]\!] \overset{F}{\mapsto} \sigma' : \theta' : E[\![e_2]\!]}$$

$$\frac{\begin{array}{c}(\sigma', \theta', \textbf{nil}) = gc_{fin}(\sigma, \theta, s) \\ \sigma' \neq \sigma \vee \theta' \neq \theta\end{array}}{\sigma : \theta : s \overset{F}{\mapsto} \sigma' : \theta' : s}$$

Figura 5.8: GC con finalización.

Intercalado de finalización con la ejecución del programa De la definición de gc_{fin} dada, vemos que un paso de GC involucra recolección junto con la posibilidad de invocar un finalizador. Nos resta explicar cómo se intercala la invocación de un finalizador junto con el código del programa en el que ocurre la recolección. A pesar de que solamente especificamos el criterio para seleccionar un único finalizador a invocar, esto no impedirá la ejecución de una cantidad arbitraria de finalizadores, antes de la ejecución de la siguiente instrucción del programa (como puede ocurrir en Lua): especificaremos reglas de ejecución no determinista para GC, que admitirán la ejecución de una cantidad arbitraria de pasos de GC, antes de continuar con las restantes instrucciones del programa. Presentamos las correspondientes reglas en la Figura 5.8, en donde explicamos la semántica de la invocación de una finalizador utilizando una nueva relación $\overset{F}{\mapsto} \subseteq \sigma \times \theta \times s$.

Admitimos la posibilidad de intercalar la invocación del finalizador en cualquier momento de la ejecución, esto es, ya sea que la próxima instrucción a ejecutar es una sentencia o una expresión. En el primer caso, incluir la invocación del finalizador puede ser expresada de manera directa, como una concatenación de sentencias (primer regla). Intercalar la invocación junto con una expresión, regla segunda, requiere de un poco de elaboración, ya que no tenemos manera directa de expresar la idea de *concatenación de expresiones*. En tal caso, reducimos el orden deseado de ejecución de las expresiones al orden de evaluación de una llamada a función. Es en esta circunstancia, observando la semántica de la invocación de finalizadores, que decimos que la tabla que está siendo finalizado es resucitada transitoriamente: ahora hay una referencia a la misma en el programa, convirtiéndola nuevamente en alcanzable.

Finalmente, si no hay finalizador que invocar (tercer regla), solamente pedimos que alguno de los almacenamientos retornados por gc_{fin} haya sido modificado, para imponer, a lo sumo, una cantidad finita de pasos de GC.

5.1.3. Tablas débiles

Una *tabla débil* es una tabla cuyas claves y/o valores son referenciados por *referencias débiles*: esto es, referencias que no son tenidas en cuenta por el recolector, al momento de determinar alcanzabilidad. En Lua, y entre los tipos incluidos en nuestro modelo, sólo tablas y clausuras puede ser recolectadas de una tabla débil: la regla general es (de acuerdo al manual de referencia) que *"sólo objetos que tienen una construcción explícita son removidos de las tablas débiles"*.

A los fines de indicar que una tabla dada será débil, quien programa debe incluir en la meta-tabla de la tabla un campo con llave "__mode", y un valor que deberá ser un string conteniendo los caracteres 'k' (para indicar que las claves deben ser referidas por referencias débiles) y/o

'v' (para indicar que los valores deben ser referidos por referencias débiles).

En el ejemplo de la Figura 5.9, ilustramos tanto la definición de tablas débiles, como la forma en la que alteran la semántica de alcanzabilidad. Notar que el valor retornado dependerá de la persistencia de un valor en una tabla débil. Esto, a su vez, dependerá del comportamiento del recolector de basura. De este modo, no se puede predecir el valor retornado.

El valor del único campo de tabla definido, en la tabla t es referido por una referencia débil, y no hay otra variable asociada de manera directa o indirecta a esta valor. Esto es, no hay un camino de alcanzabilidad hacia el valor utilizando únicamente referencias fuertes (o referencias ordinarias, es decir, las referencias ya introducidas en nuestro modelo, que siempre son tenidas en cuenta por el recolector). Luego, tal valor puede ser recolectado en cualquier momento.

```
1  local t = {}
2  -- t sera una weak table, con
3  -- valores referidos por weak
4  -- references
5  setmetatable(t, {__mode = 'v'})
6  t[1] = {}
7  local i = 0
8  while true do
9      i = i + 1
10     -- generamos basura para
11     -- disparar un ciclo de
12     -- GC
13     local garbage = {}
14     if not t[1] then break end
15 end
16 -- en este punto, el valor de
17 -- i no se puede predecir
18 return i
```

Figura 5.9: No determinismo utilizando tablas débiles.

Representación de tablas débiles

Para modelar tablas débiles no introduciremos referencias débiles de manera explícita. En su lugar, modificaremos el criterio utilizado para determinar alcanzabilidad, para que ahora considere las ocurrencias de objetos en tablas de acuerdo a la *debilidad* de la tabla; esto es, lo que se haya especificado en el campo "__mode" de su meta-tabla.

Ya fue mencionado, en el apartado anterior, que sólo objetos que tienen construcción explícita serán removidos de una tabla débil, de acuerdo a la debilidad de la misma. Este conjunto de *elementos de tabla recolectables* será denotado con cte, y contendrá a los miembros de $tid \cup cid$, correspondientes a los identificadores de tablas y clausuras. En ocasiones utilizaremos cte para denotar a un elemento de cte, apelando al contexto en el que ocurre esto, para desambiguar.

Procedemos ahora a definir formalmente parte de la semántica de tablas débiles mediante la especificación de las *ocurrencias fuertes* de elementos, dentro de una tabla dada: esto es, claves o valores, pertenecientes a cte, que no están referidos por referencias débiles, de acuerdo a la debilidad de la tabla. En la definición presentada en la Figura 5.10 asumimos la existencia de predicados $wk?$ y $wv?$, que permiten consultar si una tabla dada posee claves y/o valores débiles, respectivamente.

Como se muestra, si una tabla dada posee valores débiles, entonces sólo las ocurrencias de sus claves serán consideradas fuertes. El segundo y el tercer caso pueden ser comprendidos de un modo semejante. El tercer caso tiene que ver con lo que es conocido como *ephemerons*[37] : una tabla con claves débiles y valores fuertes que es tratada de un modo especial, por el recolector, a los fines de evitar problemas que podrían presentarse por causa de referencias mutuas entre claves y valores dentro de una tabla débil. Estos ciclos podrían llegar a evitar la correcta recolección de los elementos involucrados en el ciclo, o reducir la eficacia de un ciclo de GC (ver [38] para un análisis del problema

[37]: Hayes (1997), «Ephemerons: A New Finalization Mechanism»

[38]: Barros y col. (2008), «Eliminating Cycles in Weak Tables.»

$$SO(tid,\theta) = \begin{cases} \{k_i | k_i \in (\{k_1,...\} \cap cte)\} & \textbf{if } wv?(tid,\theta) \\ & \quad \wedge \\ & \neg wk?(tid,\theta) \\ \{v | v \in \{k_1, v_1, ...\} \cap cte\} & \textbf{if } \neg(wv?(tid,\theta) \\ & \quad \vee \\ & wk?(tid,\theta)) \\ \{(k_i, v_i) | v_i \in \{v_1,...\} \cap cte\} & \textbf{if } \neg wv?(tid,\theta) \\ & \quad \wedge \\ & wk?(tid,\theta) \\ \emptyset & \textbf{o/w} \end{cases}$$

where $\pi_1(\theta(tid)) = \{[k_1] = v_1, ...\}$

Figura 5.10: Ocurrencias fuertes de elementos de una tabla.

y cómo *ephemerons* se presenta como una solución, desde el punto de vista de la implementación realizada en Lua).

En esencia, en una tabla ephemeron la ocurrencia de un *cte* como valor de un campo, será considerada fuerte si su clave asociada es fuertemente accesible, esto es, accesible solo a través de referencias fuertes. Debido a que esta no se trata de una propiedad que pueda ser determinada localmente, es decir, solamente inspeccionando la tabla ephemeron, retornamos el par clave/valor de la tabla, en el tercer caso de la definición de SO.

Alcanzabilidad de un *cte* Una vez obtenidos los *cte* de una tabla dada, podemos distinguir 2 situaciones, con respecto a la alcanzabilidad del mismo: o bien hay un camino de alcanzabilidad partiendo del conjunto raíz de referencias, y llegando al *cte* utilizando solamente referencias fuertes, o, para todo camino que podemos hallar desde el conjunto raíz hacia el *cte*, nos encontramos con al menos una referencia débil (es decir que, en términos de nuestro modelo, tendríamos que considerar un *cte* que no ocurre fuertemente en una tabla débil). Si se presenta la primer situación, el valor es fuertemente alcanzable y no debe ser recolectado. Por otro lado, el segundo escenario permitiría la recolección del *cte*. Bastará, por lo tanto, con preocuparnos por determinar la existencia de caminos de alcanzabilidad que utilicen solo referencias fuertes.

Definiremos el predicado $reachCte$ para capturar cuándo un *cte* es fuertemente alcanzable, en una configuración dada. En comparación con el criterio capturado por $reach$, solo debemos agregar la consideración de las semántica de tablas débiles, de acuerdo a los especificado por SO.

$$\begin{gathered} eph(id,(k,v),tid,rt,\sigma,\theta) \\ \doteq \\ [k \notin cte \\ \vee \\ reachCte(k,rt,\sigma,\theta|_{tid|_k},rt)] \\ \wedge \\ reachCte(id,v,\sigma,\theta,rt) \end{gathered}$$

Figura 5.11: Alcanzabilidad desde ephemerons.

En el caso particular de ephemerons, dados un identificador *id* (es decir, un *cte*), un campo (k, v) de una ephemeron con identificador *tid*, un término *rt* (desde donde determinar el conjunto raíz de referencias), y almacenamientos σ and θ, consideraremos que *id* es fuertemente alcanzable desde dicho campo de acuerdo a lo especificado por eph, en la Figura 5.11.

Es decir, *id* será alcanzable desde (k, v) si es fuertemente alcanzable desde v, de acuerdo a $reachCte$ (cuya definición desarrollaremos más abajo), y si k no es susceptible de ser recolectado de una tabla débil (es decir, no está en *cte*), o si es fuertemente alcanzable. Al determinar esto último, no debemos tener en cuenta a v. Esta condición se impone para permitir la recolección de un campo de ephemeron hacia cuya clave solamente tenemos una referencia, proveniente de su valor. Utilizamos la notación $\theta|_{tid|_k}$ para referirnos al almacenamiento que se obtiene de remover solamente el campo de clave k, de la tabla identificada por *tid*.

En Lua, la alcanzabilidad de los valores de una ephemeron depende también de la alcanzabilidad de los ephemerons mismos (comparar esto con los punteros débiles clave/valor implementados en GHC [39] ,[81] un concepto semejante a ephemerons). Como ejemplo, observemos el código presentado en la Figura 5.12, donde la tabla c, inclusive a pesar de que está asociada con una clave fuertemente alcanzable, línea 10, es considerada no alcanzable en la línea 14 (y, por lo tanto, su finalizador es invocado). En esa región del programa la variable b no está en scope, haciendo que la ephemeron que era referenciada por la misma ya no sea alcanzable. Colocamos la llamada al servicio collectgarbage en un bucle infinito para asegurarnos de que el finalizador de la tabla c está siendo llamado durante la ejecución del programa y no al finalizar el mismo.

Tener en cuenta la alcanzabilidad de tablas ephemerons no implicará alterar nuestra anterior definición de alcanzabilidad: simplemente tendremos que considerar alcanzabilidad de ephemerons de acuerdo a lo expresado por eph, pero para ephemerons alcanzables desde el conjunto raíz de referencias.

A los fines de lograr definiciones de fácil comprensión, abstraeremos en otro predicado, $reachTable$, la especificación sobre cómo expandir el árbol de alcanzabilidad una vez que encontramos una tabla *tid*, al determinar alcanzabilidad para un identificador *id*, con respecto a un término *rt* (de donde se determinó el conjunto raíz de referencias), y almacenamientos σ y θ. Mostramos el predicado en la Figura 5.13.

Primero chequeamos por alcanzabilidad siguiendo las referencias en *tid*, para el caso en el que se trata de una ephemeron. Esto ya fue especificado en eph.

El siguiente disyunto especifica cómo determinar alcanzabilidad, para el caso en el que *tid* no es una ephemeron: no hay reglas especiales para codificar, simplemente necesitamos considerar cada ocurrencia fuerte de un *cte* presente en la tabla, tal como lo especifica SO.

Finalmente, para cada tabla hallada durante la expansión del árbol de alcanzabilidad, también necesitamos mirar en su correspondiente meta-tabla, del mismo modo en el que lo hicimos al definir $reach$ en §5.1.1.

Ahora estamos en condiciones de precisar $reachCte$, el cual tendrá una signatura semejante a la de $reach$, con la adición del término desde el cual se debe determinar el conjunto raíz de referencias, el cual

81: De [39]: *"La especificación* [del módulo Haskell Weak, que implementa punteros débiles] *no menciona a la alcanzabilidad del objeto clave/valor en sí, por lo cual, la alcanzabilidad del par no afecta la alcanzabilidad de su clave ni de su valor"*.

[39]: Peyton Jones y col. (2000), «Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell»

```
1  a = {}
2  do
3   local b,c,mc =  {},{},{}
4   -- ephemeron
5   setmetatable(b, {__mode="k"})
6   mc.__gc = function ()
7                print (" finalizing  c")
8                end
9   setmetatable(c, mc)
10  b[a] = c
11 end
12
13 while true do
14  collectgarbage()
15  -- " finalizing  c"
16 end
```

Figura 5.12: Alcanzabilidad de valores en ephemerons.

$$
\begin{gathered}
reachTable(id, tid, \sigma, \theta, rt) \\
\doteq \\
[\exists (k, v) \in SO(tid, \theta), \\
eph(id, (k, v), tid, rt, \sigma, \theta)] \\
\vee \\
[\exists v \in SO(tid, \theta), \\
reachCte(id, v, \sigma, \theta, rt)] \\
\vee \\
reachCte(id, \pi_2(\theta(tid)), \sigma, \theta, rt)
\end{gathered}
$$

Figura 5.13: Alcanzabilidad desde una tabla.

$$
\begin{gathered}
reachCte(id, t, \sigma, \theta, rt) \\
= \\
id \in t \\
\vee \\
\exists\, r \in t,\ reachCte(id, \sigma(r), \sigma, \theta, rt) \\
\vee \\
\exists\, tid \in t, reachTable(id, tid, \sigma, \theta, rt) \\
\vee \\
\exists\, cid \in t, \\
reachCte(id, \theta(cid), \sigma, \theta, rt)
\end{gathered}
$$

Figura 5.14: Alcanzabilidad considerando semántica de tablas débiles.

lo mantendremos a través de las llamadas, específicamente para la definición de alcanzabilidad desde ephemerons:

Mientras que sería posible capturar el predicado proveyendo una recursión primitiva, serían necesarias expresiones intrincadas para definir las llamadas recursivas sobre subconjuntos propios de los almacenamientos. En su lugar, seguiremos [33] y definiremos el predicado de interés como el menor punto fijo que satisface la ecuación en Figura 5.14.

El predicado está definido asumiendo que la mera ocurrencia de un *cte* en t implica que tal valor es fuertemente alcanzable. Los casos recursivos son definidos manteniendo esta propiedad, para el término que se pasa como argumento, sobre el cual se determina el conjunto raíz.

El segundo disyunto desreferencia referencias a valores (de $\mathsf{dom}(\sigma)$) encontradas en t. A continuación, se expande el árbol de alcanzabilidad siguiente las tablas encontradas, tal como se expresa en $reachTable$. El último disyunto chequea en el entorno de las clausuras encontradas durante la expansión del árbol.

Respecto al recurso utilizado para capturar $reachCte$, notar que basta con considerar un dominio de predicados ordenados por la cantidad de niveles que exploran, dentro de un árbol de alcanzabilidad: es decir, cuantas operaciones de desreferenciado intentan, a lo sumo, a partir de las referencias del conjunto raíz, para determinar alcanzabilidad de una referencia dada l. El menor punto fijo (de una función continua sobre el dominio descrito) será el predicado que siempre realiza la cantidad necesaria de operaciones de desreferenciado, para determinar si l es alcanzable o no.

Finalmente, como se mencionó en §5.1, para tratar GC basada en alcanzabilidad, hemos considerado necesario incluir el manejo de clausuras a través de referencias. El motivo reside en que, de este modo, podemos modelar de manera tradicional las nociones de *camino de alcanzabilidad* y errores de punteros colgantes (problema que será explicado en el próximo apartado), que surgen durante el tratamiento de la semántica de tablas débiles.

Ciclo de GC con finalización y tablas débiles Nos resta definir en qué consiste un paso de GC que considere las 2 interfaces con el recolector descriptas.

Notar que al proveer una noción de alcanzabilidad que incluye la semántica de referencias débiles (a través de tablas débiles), sería posible que el recolector remueva el binding de un identificador de tabla o clausura *id* que no es fuertemente alcanzable, de acuerdo a la semántica de tablas débiles, pero que aún está presente en una tabla débil alcanzable. Esto, naturalmente, permitiría que ocurran errores de punteros colgantes si el programa intentara desreferenciar tales identificadores, a través de la tabla débil. Notar también que para tener una representación natural de este fenómeno, es adecuado también manejar clausuras mediante referencias, como explicamos

en §5.1: de otro modo, la recolección de un binding (r, *functiondef*) en σ, no provocaría el error esperado en una tabla dada que almacene directamente *functiondef*, en lugar de una referencia a la misma.

Para evitar errores de punteros colgantes, nos aseguraremos de que las tablas débiles sean *limpiadas* (es decir, admitiremos recolección de basura dentro de las tablas) antes de que ocurra la recolección de identificadores que se encuentren en la situación descripta para el identificador *id*.

Para lograr imponer la restricción anterior, especificaremos un paso de GC con finalización y limpieza de tablas débiles, gc_{fin_weak}, procediendo primero de acuerdo a lo especificado en gc_{fin} (salvo por pequeñas modificaciones, que iremos desarrollando). Así, desde el punto de vista de la noción de alcanzabilidad capturada en gc_{fin}, los identificadores alcanzables desde una tabla débil no serán considerados basura y, por lo tanto, no serán removidos hasta que no se proceda a limpiar las correspondientes tablas débiles. Sí habrá interacción entre finalización y tablas débiles: por un lado, finalización ahora debiera determinar cuando una tabla está en condiciones de ser finalizada, también contemplando semántica de tablas débiles; por el otro, existe una forma de interacción entre ambas interfaces que fuerza a tener que redefinir finalización. Contemplaremos esta interacción más adelante, en este mismo apartado. Por el momento, comencemos con la especificación de gc_{fin_weak}, propuesta en la Figura 5.15.

Ahora permitimos que el algoritmo de GC remueva campos de tablas débiles, de acuerdo a la debilidad de las mismas. Esto es lo que expresan el conjunto de condiciones indicadas como wt.

Por comenzar, el almacenamiento retornado por gc_{weak_fin} puede diferir de aquel sobre el cual se realizó recolección con finalización, solamente en el caso de campos de tablas (k, v), con valores *cte*. Esto es lo que expresa la condición $weakness$.

Un campo puede ser removido si no posee clave o valor alcanzable fuertemente. Esto se expresa mediante la condición $reach$. Notar que pasamos s como el último argumento de $reachCte$, para preservarlo como el término desde el cual determinar el conjunto raíz de referencias, desde donde una nueva expansión del árbol de alcanzabilidad debiera comenzar.

En caso de claves débiles, la semántica de Lua impone un comportamiento particular, en relación con la invocación de sus finalizadores: claves débiles que consisten en objetos marcados para finalización son removidos sólo luego de que hayan sido finalizados. Esto es expresado por condición $fin\ key$. Esta restricción permite que un finalizador de una clave débil acceda a la información (*propiedad*) asociada con la clave que está siendo finalizada, en caso de que la propiedad esté definida mediante la tabla débil a la que pertenece la clave.

Con respecto a los valores débiles, el manual de referencia especifica que estos son removidos de tablas débiles antes de ser finalizados. Para incorporar esta restricción, tendremos que fortalecer las condiciones

$$
\begin{array}{l}
gc_{fin\_weak}(s,\sigma,\theta) \quad = \quad (\sigma_1,\theta_1''',e), \\
\text{donde } (\sigma_1,\theta_1',e) = gc_{fin}(s,\sigma,\theta), \text{ y:} \\
wt \left\{ \begin{array}{l}
sea\ \theta_1''/dom(\theta_1'') = dom(\theta_1') \\
\\
\forall tid \in dom(\theta_1''), \theta_1''(tid) = \theta_1'(tid) \vee \\
[\pi_1(\theta_1''(tid)) \subset \pi_1(\theta_1'(tid)) \wedge \\
\exists (k,v) \in \pi_1(\theta_1'(tid)), (k,v) \notin \pi_1(\theta_1''(tid))/ \\
\quad weakness \left\{ \begin{array}{c} wk?(tid,\theta) \ \wedge \ k \in cte \\ \vee \\ wv?(tid,\theta) \ \wedge \ v \in cte \end{array} \right. \\
\quad \wedge \\
\quad reach \left\{ \begin{array}{c} \neg reachCte(k,s,\sigma,\theta,s) \\ \vee \\ \neg reachCte(v,s,\sigma,\theta,s) \end{array} \right. \\
\quad \wedge \\
\quad fin\ key \left\{ \begin{array}{c} k \in \mathsf{dom}(\theta) \wedge wk?(tid,\theta) \\ \Rightarrow \\ \neg marked(k,\theta) \end{array} \right. \\
\quad \wedge \\
\quad rem \left\{ \begin{array}{c} \pi_2(\theta_1''(tid)) = \pi_2(\theta_1'(tid)) \\ \wedge \\ \pi_3(\theta_1''(tid)) = \pi_3(\theta_1'(tid))] \end{array} \right.
\end{array} \right. \\
\\
fin \left\{ \begin{array}{l}
\theta_1''' = \theta_1''[tid := (\pi_1(\theta_1''(tid)), \pi_2(\theta_1''(tid)), \oslash)], \\
\quad si\ gc_{fin}(s,\sigma,\theta) = (\sigma_1,\theta_1',v(tid)) \\
\\
\theta_1''' = \theta_1'', \\
\quad si\ gc_{fin}(s,\sigma,\theta) = (\sigma_1,\theta_1',\mathbf{nil})
\end{array} \right.
\end{array}
$$

Figura 5.15: GC con finalización y tablas débiles.

que caracterizan a una tabla que puede ser finalizada: ahora, finalización necesita evaluar la alcanzabilidad de una tabla utilizando el criterio expresado por $reachCte$, de modo de que finalización también ocurra de acuerdo a la semántica de tablas débiles; a su vez, la tabla escogida para finalización no puede ser un valor de una tabla débil.

Realizaremos pequeños cambios en la definición de gc_{fin}, para incorporar estas formas de interacción entre finalización y tablas débiles (comparar la presencia de esta interacción con el modelo presentado en [40] , en donde no hay otra forma de interacción entre ambas interfaces). Primero, definiremos un nuevo predicado, fin_{weak}, que caracterizará a las tablas que están en condiciones de ser finalizadas (Figura 5.16). Notar que difiere del predicado utilizado originalmente, fin, en el reemplazo de $reach$ por $reachCte$. Finalmente, debemos agregar una condición que impida que se invoque finalización para una tabla que ocurre como valor de una tabla débil. Capturamos esta condición con el predicado not_fin_val, de la Figura 5.16.

[40]: Leal (2005), «Finalizadores e referências fracas: interagindo com o colector de lixo.»

$$
\begin{array}{c}
fin_{weak}(\mathit{tid}, \mathsf{s}, \sigma, \theta) \\
\doteq \\
\neg reachCte(\mathit{tid}, \mathsf{s}, \sigma, \theta, \mathsf{s}) \\
\wedge \\
marked(\mathit{tid}, \theta)
\end{array}
$$

$$
\begin{array}{c}
not\_fin\_val(\mathit{tid}, \theta) \\
\doteq \\
\nexists \mathit{tid}' \in \mathsf{dom}(\theta), k \in v / \\
(wk?(\mathit{tid}', \theta) \vee wv?(\mathit{tid}', \theta)) \\
\wedge \\
(k, \mathit{tid}) \in \pi_1(\theta(\mathit{tid}'))
\end{array}
$$

Figura 5.16: Criterio de finalización, utilizando semántica de tablas débiles.

Solo resta por redefinir gc_{fin} de forma tal de que emplee el criterio fin_{weak} en lugar de fin, para determinar qué tablas están en condición de ser finalizadas, considerando ahora semántica de tablas débiles; y que fortalezca el criterio anterior con lo expresado con not_fin_val, para que se demore la finalización de un valor de tabla débil, hasta después de que haya sido removido de su tabla. Finalmente, trasladamos a gc_{fin_weak} la responsabilidad de evitar que una tabla *tid* finalizada pueda ser nuevamente marcada para finalización; expresábamos esto en nuestro modelo asociando $\oslash$ con *tid*. Realizaremos esto en gc_{fin_weak}, condición *fin* de la Figura 5.15. El motivo detrás de esto radica en que, una vez escogido el siguiente finalizador, debemos esperar a que se ejecute el mismo, antes de realizar limpieza de claves de tablas débiles, como ya hemos explicado. No logramos imponer esta restricción si imponemos la especificación de limpieza de tablas débiles ya habiendo asociado $\oslash$ con una tabla *tid* seleccionada para finalización. Con motivo de que esos son los únicos cambios necesarios, omitimos la nueva definición de gc_{fin}.

Retornando a Figura 5.15, la última condición, rem, simplemente expresa que la información restante, asociada a la tabla débil dentro de la cual se realiza la limpieza, persiste inalterable.

Finalmente, no hay necesidad de redefinir el paso de GC, más allá de reemplazar gc_{fin} por gc_{weak_fin}: los detalles de GC de tablas débiles están abstraídos en la meta-función gc_{weak_fin}, y su interfaz con la ejecución de un programa no difiere de lo realizado con gc_{fin}.

5.2. Propiedades de GC

En esta sección capturaremos un conjunto de conceptos relacionado a GC sintáctica, y demostraremos ciertas propiedades del modelo presentado, las cuales serán de utilidad para razonar sobre corrección de cualquier algoritmo de GC basado en basura sintáctica.

Comenzaremos por el tratamiento formal de la noción de basura, capturada en términos del efecto de recolectarla, en el comportamiento de un programa.

5.2.1. Basura sintáctica

Informalmente, basura consistirá en un binding (un par referencia/valor o identificador/clausura, tabla) el cual, si es eliminado, no producirá un efecto *observable* en la ejecución de un programa. Para nuestro trabajo, observaremos los valores retornados por la ejecución de un programa (a definir en la próxima sección) o su no terminación. Una estrategia de GC correcta, entonces, no debiera alterar ese comportamiento observado.

Queda claro, entonces, que para poder atacar formalmente la definición de basura y de estrategias de GC correctas, es necesario disponer de una semántica dinámica sobre la cual estudiar la mencionada noción de comportamiento.

Comenzaremos definiendo una noción adecuada de *resultado de un programa*, para poder, a continuación, capturar formalmente la mencionada noción de basura.

Resultado de un programa Lua Para un lenguaje imperativo como Lua, el resultado de un programa dado podría consistir en el término de la última configuración de su computación (convergente), junto con la información de los almacenamientos necesaria para explicar el significado de las referencias dentro del término (las variables libres, que han sido substituidas por sus referencias).

Naturalmente, solo tipos no-primitivos van a requerir de información adicional de los almacenamientos, para explicar completamente un valor retornado, de tales tipos. De entre los tipos Lua, las tablas son el único tipo de dato estructurado. Consistente en entidades definidas por su estructura y su identificador, que es único para un programa dado. A su vez, estas tablas podrían contener referencias a otras tablas, potencialmente creando ciclos en el *heap*. A su vez, programas Lua bien formados puede crear e inspeccionar tablas que están definidas de manera mutuamente recursiva, por lo cual, permitiremos estas construcciones en nuestra noción de *resultado de un programa Lua*.

Por ejemplo, consideremos los siguientes almacenamientos θ:

$$\{tid_1 \to (\{[tid_2] = 1\}, ...), tid_2 \to (\{[tid_1] = 2\}, ...)\}$$

$$\{tid_1 \to (\{[tid_2] = 1\}, ...), tid_2 \to (\{[tid_2] = 2\}, ...)\}$$

estos contienen listas estructuralmente diferentes, ya que $tid_1 \neq tid_2$; diferencia reconocible por un programa Lua, y pueden, a su vez, ser generadas por un programa Lua bien formado.

Con respecto a los tipos de datos restantes, solamente clausuras requieren una consideración especial. En nuestro modelo, y como se mencionó en §4.2.3, el entorno de una clausura está parcialmente embebido en su cuerpo: cualquier identificar de variable externa, ha sido reemplazada por su correspondiente referencia al almacenamiento de valores. Luego, para poder describir de manera completa a la clausura que una definición de función dada está representando, necesitaremos incluir los bindings del almacenamiento de valores, referidos en el entorno de la clausura.

$$result(\sigma : \theta : E\ [\![\textbf{return } v_1, \ldots, v_n]\!]) = \sigma|_E : \theta|_T : \textbf{return } v_1, \ldots, v_n,$$

donde
$$\begin{cases} \not\vdash_{mtch} E \approx E' \ [\![(\!| E_{lf} [\![\ ]\!] |\!)_{label}]\!], label \notin \{\text{RETSTAT}, \text{RETEXP}, \text{BREAK}\} \\ E = \bigcup_{r \in \text{dom}(\sigma),\ reach(r, \textbf{return } v_1, \ldots, v_n, \sigma, \theta)} r \\ T = \bigcup_{id \in \text{dom}(\theta),\ reach(id, \textbf{return } v_1, \ldots, v_n, \sigma, \theta)} id \end{cases}$$

$$result(\sigma : \theta : \textbf{\$err } v) = \sigma|_E : \theta|_T : \textbf{\$err } v,$$

donde E y T son análogos al caso anterior

$$result(\sigma : \theta : \textbf{;}) = \emptyset : \emptyset : \textbf{;}$$

Figura 5.17: Resultado de un programa.

Finalmente del corolario 4.3.3, presentado en §4.3, podemos saber los posibles resultados de una computación convergente, bajo la semántica dinámica indicada por $\mapsto$. Sintetizaremos la discusión previa, junto con lo explicado por el corolario 4.3.3, para capturar la noción de *resultado* utilizando la función $result$ en la Figure 5.17. Simplemente toma la configuración final de una computación convergente y extrae los bindings necesarios, de cada almacenamiento, para describir completamente al resultado. Se trata de una definición sencilla, de la que simplemente cabe destacar que, en el contexto del estudio de recolectores sintácticos, esa noción de resultado no será sensible a estrategias de GC diferentes, o, inclusive, a la total ausencia de GC.

Como se mencionó previamente, las tablas están definidas tanto por la información que contienen, como por su identidad, la cual es representada en nuestro modelos modelo por los elementos del conjunto *tid*. Luego, si un identificador *tid* es devuelto por un programa, no podemos utilizarlo para extraer su tabla asociada en θ y considerar esta como el resultado. Necesitamos de ambos elementos, la identidad y su información asociada. A su vez, como una tabla podría contener otras tablas como llaves o valores, necesitamos recorrer cada tabla retornada, buscando por la presencia de otros identificadores de tablas o clausuras. Lo que estamos describiendo ya está capturado por el predicado $reach$, esto es, alcanzabilidad sin contemplar la semántica de tablas débiles.

inline]Quizás acá falta un lema que mencione que, bajo gc con tablas débiles, en la configuración final ya se han eliminado todos los campos de tablas débiles?

Finalmente, necesitamos expresar la idea de que la representación particular de una referencia (a un valor, o identidad de tabla o clausura) no tiene importancia, cuando se trata de comprar entre resultados de un programa. Si un programa dado siempre retorna un identificador de tabla, y la tabla misma es, estructuralmente siempre la misma, quisiéramos poder considerar como equivalentes a los resultados devueltos por el programa.

Para formalizar la observación precia, por un lado, seguiremos ideas tradicionales y asumiremos la existencia de la correspondiente α-conversión entre referencias de σ y θ. Por otro lado, notar que estamos tratando con la semántica de un lenguaje de programación real, que provee varios servicios de librería que podrían invalidar nuestras definiciones y resultados. Por ejemplo, en Lua 5.2 es posible, a través del servicio tostring, convertir a un string la representación de un identificador de tabla o clausura. Naturalmente, si permitiéramos esto estaríamos en condiciones de escribir programas cuyos valores retornados dependerán de detalles no contemplados de manejo de memoria, y que estarán más allá del alcance del tratamiento formal que aquí estamos dando a la noción de resultado (por ejemplo, en casos en el que valor resultado es de un tipo primitivo, como números o strings). Por lo tanto, en lo que resta de esta sección asumiremos que los programas Lua no están en condiciones de convertir las representaciones de identificadores de tablas y clausuras hacia valores primitivos.

```
1  local t = {}
2
3  for i=1,10 do
4    t [{}] = i
5  end
6
7  for k,v in pairs(t) do
8    -- Campos accedidos en
9    -- orden no
10   -- determinista
11   print(v)
12 end
13
```

Figura 5.18: Comportamiento no determinista en iteración de una tabla con claves no numéricas.

Observaciones Lo que resta es definir las observaciones sobre programas, en términos de las cuales mediremos el efecto de la adición de GC y sus interfaces, a nuestra semántica dinámica. Para programas de un lenguaje real, como Lua, inclusive bajo GC simple (es decir, sin interfaces con el recolector) podemos observar no-determinismo. Además de la posibilidad de generar valores pseudo-aleatorios a partir de convertir referencias a strings, como ya se discutió, existen otras fuentes de no-determinismo: lectura de archivos, networking, lectura del reloj de la máquina, etc. A su vez, iterar sobre una tabla con claves no numéricas, utilizando iteradores de la librería estándar, puede presentar un comportamiento sensible a detalles de manejo de memoria (en particular, a la forma en la que Lua almacena tablas) y ser, por lo tanto, no determinista. Figura 5.18 presenta un ejemplo: el orden en el que los campos son accedidos en el último bucle, líneas 7-9, está dependiendo de detalles no especificados de manejo de memoria, variando entre distintas ejecuciones.

Nuestro modelo no incluye servicios para manipulación de archivos, networking o, en general, llamadas a servicios del sistema operativo. Nuestro modelo sí incluye iteradores para tablas de librería estándar, aunque bajo GC sin interfaces no observaremos iteraciones no deterministas: nuestro modelo no incluye detalles respecto a cómo se ubican los campos de una tabla en la memoria de la computadora. Sí observaremos no determinismo al incluir tablas débiles, e intentar iterar una tabla débil, pero se trata de un fenómeno propio de la semántica de estas.

Revisar las fuentes de no determinismo en $\mapsto$ nos permitirá realizar el chequeo de sanidad estándar encontrado en la literatura[31, 33, 41] para GC sin interfaces: esto es, verificar que la adición de un paso de GC sin interfaces no altera las observaciones sobre un programa dado. Naturalmente, al agregar las interfaces sí observaremos no determinismo. Será solo por causa de estas únicas fuentes de no determinismo que proveeremos una noción de observaciones sobre programas lo suficientemente flexible como para permitirnos también hablar de comportamiento de programas con GC con interfaces.

La definición estará parametrizada sobre una relación $\rightarrow$ que formalizara la semántica dinámica en términos de la cual estamos definiendo las observaciones. Para nuestros estudios, $\rightarrow$ serán $\mapsto$ y $\overset{GC}{\mapsto}$. Utilizaremos la notación introducida en el corolario 4.3.3, para hablar de convergencia o divergencia de un cómputo, pero ahora le agregaremos un súper-índice para especificar la semántica $\rightarrow$ en términos de la cual observamos convergencia o divergencia. Utilizaremos C como variable cuantificada sobre el conjunto de configuraciones:

inline]Should we separate errors from results?, in Aweak it is done like that.

Definición 5.2.1 (Observaciones) *Para una configuración C tal que $\vdash_{wfc} C$, y semántica $\rightarrow$, definiremos $obs(C, \rightarrow)$, las observaciones sobre C dada $\rightarrow$, como:*

$$obs(C, \rightarrow) = \{\bot \mid C \Uparrow_{\rightarrow}\} \cup \{result(C') \mid C \Downarrow_{\rightarrow} C'\}$$

La anterior definición se apoya en la idea de que una cierta propiedad de progreso vale para $\rightarrow$: si $result$ está definida sobre la configuración final de una computación convergente, esta configuración debe estar bien formada y tener alguna de las formas esperadas para una configuración final. Mientras que esto sería cierto para $\rightarrow{=}\mapsto$, no hemos provisto evidencia de que esto también vale tras la adición de $\overset{GC}{\mapsto}$, y sus interfaces. Durante el estudio de propiedades de nuestro modelo, veremos que sí podemos extender la propiedad de corrección de $\vdash_{wfc}$ ahora sobre la semántica con GC (sin interfaces), deduciendo de ello que no hay estados *stuck* en esta nueva semántica. inline]Is it necessary to include the stuck states for $\rightarrow$?

Finalmente, la siguiente relación de equivalencia tiene como fin el permitirnos enunciar de manera sencilla propiedades respecto a observaciones sobre programas:

Definición 5.2.2 (Equivalencia de observaciones) *Sea $\equiv$ la relación definida cómo:*

$$(C, \rightarrow) \equiv (C', \rightarrow') \Leftrightarrow obs(C, \rightarrow) = obs(C', \rightarrow')$$

Basura Con las definiciones desarrolladas hasta el momento, podemos formalizar la noción de basura, discutida hasta ahora de manera informal:

Definición 5.2.3 (Basura) *Para una configuración $\sigma \uplus \{(r, v)\} : \theta : s$, tal que $\vdash_{wfc}\ \sigma \uplus \{(r, v)\} : \theta : s$, semántica operacional $\rightarrow$, decimos que el binding (r, v) es basura, con respecto a $\rightarrow$, sí y sólo si:*

$$(\sigma \uplus \{(r, v)\} : \theta : s, \rightarrow) \equiv (\sigma : \theta : s, \rightarrow)$$

Un binding en θ es definido como basura de manera análoga.

Naturalmente, para $\rightarrow{=}\mapsto$ el conjunto de observaciones sobre una configuración C dada será un singleton. Un binding en C será basura si, luego de remover tal binding, las observaciones sobre la configuración resultante, y bajo la misma semántica, siguen siendo el mismo singleton. Luego de la adición de tablas débiles y finalización, el conjunto de observaciones podría no ser un singleton.

La serie de conceptos hasta acá introducidos nos permitirán definir y verificar una noción de corrección para GC en ausencia de sus interfaces, pero también podrían ser de utilidad para futuros estudios de propiedades y aplicaciones del modelo presentado de tablas débiles y finalización.

5.2.2. Propiedades

Con las nociones que hemos definido podemos comenzar con el estudio de las propiedades del modelo de GC que hemos presentado. Para GC sin interfaces, en particular, podremos realiza el chequeo de sanidad estándar, esto es, definir y verificar la propiedad de corrección del paso de GC. Informalmente, consiste en verificar que la adición a $\mapsto$ del paso de GC no altera las observaciones de un programa dado, bajo la semántica sin pasos de GC, es decir, solo $\mapsto$. inline]Alguna mención a las propiedades que demostraremos sobre GC con interfaces?

Corrección de $\overset{GC}{\mapsto}$

inline]Mejor descripción? Para llegar a una prueba de corrección de $\overset{GC}{\mapsto}$ necesitaremos, primero, verificar varios lemas que hablan sobre propiedades sencillas sobre $\mapsto$ y $\overset{GC}{\mapsto}$. Comenzaremos estudiando $\mapsto$.

Propiedades preservadas por un paso de $\mapsto$ El primer lema que trabajaremos enuncia que, una vez que un binding se torna recolectable (es decir, ya no es alcanzable), entonces se mantendrá en ese estado luego de cualquier paso de cómputo de $\mapsto^*$.

Una demostración completa requeriría realizar análisis por casos sobre todas las reglas de nuestra semántica. Por razones de brevedad nos concentraremos en argumentar sobre la corrección del lema únicamente sobre los casos interesantes:

Lema 5.2.1 *Para configuraciones bien formadas* $(\sigma_1 : \theta_1 : s_1)$, $(\sigma_2 : \theta_2 : s_2)$, *si* $(\sigma_1 : \theta_1 : s_1) \mapsto (\sigma_2 : \theta_2 : s_2)$, *entonces* $\forall l \in dom(\sigma_1) \cup dom(\theta_1)$, $\neg reach(l, s_1, \sigma_1, \theta_1) \Rightarrow \neg reach(l, s_2, \sigma_2, \theta_2)$.

Demostración. Nos guiaremos por la estructura de $\mapsto$ para razonar sobre cada regla de cómputo que transforma $(\sigma_1 : \theta_1 : s_1)$ en $(\sigma_2 : \theta_2 : s_2)$. Distinguimos las siguientes situaciones, para el paso de $\mapsto$:

- *El cómputo no depende del contenido de los almacenamientos (es decir, no altera bindings o desreferencia)*: se puede ver, realizando análisis por casos, que una tal regla no introduce nuevas referencias en el término s_2 (tales cómputos simplemente se explican mediante $\rightarrow^{s/e} \subseteq s \cup e$). Lo que sí podría ocurrir es que el conjunto raíz de referencias, desde el cual determinar alcanzabilidad, se reduzca, al perder alguna de las referencias presentes en s_1. En cualquier caso, debe ser que para una $l \in dom(\sigma_1) \cup dom(\theta_1)$, si $\neg reach(l, s_1, \sigma_1, \theta_1)$ también ocurre que $\neg$reach$(l, s_2, \sigma_2, \theta_2)$.
- *El cómputo altera bindings o desreferencia referencias de* σ_1: asumamos que para una cierta $l \in dom(\sigma_1) \cup dom(\theta_1)$ tal que $\neg reach(l, s_1, \sigma_1, \theta_1)$, se da que $reach(l, s_2, \sigma_2, \theta_2)$. Para razonar sobre este enunciado consideremos el caso de la operación de desreferenciado implícito de referencias a σ_1, presentado en §4.2. Debe ser, entonces, que s_1 encaja con el patrón $E[\![\ r\ ]\!]$, para un contexto de evaluación E y una referencia r, y el cómputo es:

$$\sigma_1 : \theta_1 : \overset{s_1}{\overbrace{E[\![\ r\ ]\!]}} \overset{L}{\mapsto} \sigma_1 : \theta_1 : \overset{s_2}{\overbrace{E[\![\sigma_1(r)]\!]}}$$

  donde vemos que los almacenamientos no son alterados, luego del cómputo. Luego, el conjunto raíz en s_1 solamente cambió como resultado del reemplazo de r por las referencias en $\sigma_1(r)$. Si $\neg reach(l, s_1, \sigma_1, \theta_1)$ pero $reach(l, s_2, \sigma_2, \theta_2)$, esto significaría que l es alcanzable desde las referencias en $\sigma_1(r)$. Pero en s_1, las referencias de $\sigma_1(r)$ también eran alcanzables, descubriendo que l era alcanzable en s_1. Esto contradice nuestra hipótesis original. Debe ser que si $\neg reach(l, s_1, \sigma_1, \theta_1)$, entonces l se mantiene no alcanzable en $(\sigma_2 : \theta_2 : s_2)$.
- *El cómputo cambia o desreferencia referencias de* θ_1: para una $l \in dom(\sigma_1) \cup dom(\theta_1)$ dada, tal que $\neg reach(l, s_1, \sigma_1, \theta_1)$, asumamos que $reach(l, s_2, \sigma_2, \theta_2)$. Analizaremos el caso de la la semántica de constructores de tablas. Entonces, debe ser que s_1 encaja con el patrón $E[\![\ t\ ]\!]$, para un contexto de evaluación E y un constructor de tabla t, en donde cada campo ya ha sido evaluado, dejando a la tabla en condiciones de ser ubicada en el almacenamiento θ_1. Luego, el cómputo (simplificado) es:

$$\frac{\theta_2 = (tid,\ (t, ...)),\ \theta_1}{(\sigma_1\ :\ \theta_1\ :\ E[\![\ t\ ]\!])\ \mapsto\ (\sigma_1\ :\ \theta_2\ :\ E[\![\ tid\ ]\!])}$$

  en donde vemos que el almacenamiento de valores no se ha alterado, esto es, $\sigma_2 = \sigma_1$. Luego, el conjunto raíz de referencias solamente cambió en el caso del reemplazo de las referencias en t por el identificador fresco de tabla tid. Si $\neg reach(l, \sigma_1, \theta_1, s_1)$ pero $reach(l, \sigma_2, \theta_2, s_2)$, esto significaría que $l = tid$, lo cual no puede ser el caso, pues $l \in dom(\sigma_1) \cup dom(\theta_1)$ y $tid \notin dom(\sigma_1) \cup dom(\theta_1)$ (pues siempre tid es un identificador fresco). Luego, debe ser que si $\neg reach(l, \sigma_1, \theta_1, s_1)$, entonces l sigue siendo no alcanzable en $(\sigma_2 : \theta_2 : s_2)$.

□

[*] Notar que esta propiedad sencilla no vale para el caso de GC con finalización o tablas débiles; pensemos, por ejemplo, en el fenómeno de resurrección de tablas que están por ser finalizadas.

La definición y lema siguientes capturan un concepto estándar en semántica operacional para lenguajes imperativos: dado un programa, el resultado de su ejecución bajo ciertos almacenamientos dados dependerá solamente de la porción alcanzable de dichos almacenamientos, desde el programa.

Definición 5.2.4 *Para configuraciones bien formadas* $(\sigma_1 : \theta_1 : \mathtt{s})$ *y* $(\sigma_2 : \theta_2 : \mathtt{s})$*, diremos que ambas configuraciones* coinciden en la porción alcanzable de sus almacenamientos*, denotado como:*

$$(\sigma_1 : \theta_1 : \mathtt{s}) \overset{rch}{\sim} (\sigma_2 : \theta_2 : \mathtt{s})$$

sí y sólo si $\forall l \in dom(\sigma_1) \cup dom(\theta_1)/reach(l, \mathtt{s}, \sigma_1, \theta_1)$*, vale que:*

- $reach(l, \mathtt{s}, \sigma_2, \theta_2)$
- $l \in dom(\sigma_1) \Rightarrow \sigma_1(l) = \sigma_2(l)$
- $l \in dom(\theta_1) \Rightarrow \theta_1(l) = \theta_2(l)$

y lo mismo ocurre $\forall l \in dom(\sigma_2) \cup dom(\theta_2)$*.*

En la definición anterior asumimos que, si es necesario, es posible realizar α-conversión de las referencias de ambas configuraciones, de modo tal de hacerlas comparables en el sentido expresado por $\overset{rch}{\sim}$. Finalmente, es fácil demostrar que $\overset{rch}{\sim}$ es una relación de equivalencia.

La propiedad importante, satisface por configuraciones que coinciden en la porción alcanzable de sus almacenamientos, es enunciada por el siguiente lema:

Lema 5.2.2 *Para configuraciones bien formadas* $(\sigma_1 : \theta_1 : \mathtt{s}_1)$, $(\sigma_2 : \theta_2 : \mathtt{s}_1)$*, tales que* $(\sigma_1 : \theta_1 : \mathtt{s}_1) \overset{rch}{\sim} (\sigma_2 : \theta_2 : \mathtt{s}_1)$*, si* $\exists\, (\sigma_3 : \theta_3 : \mathtt{s}_2)/(\sigma_1 : \theta_1 : \mathtt{s}_1) \mapsto (\sigma_3 : \theta_3 : \mathtt{s}_2)$*, entonces* $(\sigma_2 : \theta_2 : \mathtt{s}_1) \mapsto (\sigma_4 : \theta_4 : \mathtt{s}_2)$*, para el mismo cómputo de* $\mapsto$ *y para una configuración* $(\sigma_4 : \theta_4 : \mathtt{s}_2)$ *tal que* $(\sigma_3 : \theta_3 : \mathtt{s}_2) \overset{rch}{\sim} (\sigma_4 : \theta_4 : \mathtt{s}_2)$*.*

Demostración. Seguiremos la estructura de $\mapsto$ para razonar sobre la aplicación de un cómputo de $\mapsto$ sobre $(\sigma_2 : \theta_2 : s_1)$:

- *El cómputo se explica mediante* $\rightarrow^{s/e}$: entonces, debe ser el caso de que toda la información de los almacenamientos, necesaria para permitir el cómputo, ya está contenida en el término s_1. Es decir que el paso a realizar de $\mapsto$, y el resultado a obtener, depende exclusivamente de $\mathtt{s}_1$. También, podemos inferir que luego del cómputo, los almacenamientos no han sido modificados. Esto implica que :

$$(\sigma_1 : \theta_1 : \mathtt{s}_1) \mapsto (\overset{\sigma_1}{\overset{=}{\sigma_3}} : \overset{\theta_1}{\overset{=}{\theta_3}} : \mathtt{s}_2) \ \wedge\ (\sigma_2 : \theta_2 : \mathtt{s}_1) \mapsto (\overset{\sigma_2}{\overset{=}{\sigma_4}} : \overset{\theta_2}{\overset{=}{\theta_4}} : \mathtt{s}_2)$$

  Luego, el conjunto raíz de referencias de ambas configuraciones, $(\sigma_3 : \theta_3 : \mathtt{s}_2)$ y $(\sigma_4 : \theta_4 : \mathtt{s}_2)$, coincide. A su vez, dado que $(\sigma_1 : \theta_1 : \mathtt{s}_1) \overset{rch}{\sim} (\sigma_2 : \theta_2 : \mathtt{s}_1)$ y que los almacenamientos de ambas configuraciones no son modificados luego del paso de $\mapsto$, podemos concluir en que las porciones alcanzable a partir de $\mathtt{s}_2$, de estos últimos almacenamientos, deben coincidir en el sentido expresado por $\overset{rch}{\sim}$. Entonces:

$$(\sigma_3 : \theta_3 : \mathtt{s}_2) \overset{rch}{\sim} (\sigma_4 : \theta_4 : \mathtt{s}_2)$$

- *El cómputo se explica mediante* $\rightarrow^{\sigma}$: consideremos la operación de desreferenciado implícito de referencias a σ. En este caso, la hipótesis puede ser reescrita como:

$$(\sigma_1 : \theta_1 : \mathtt{s}_1) \overset{L}{\mapsto} (\overset{\sigma_1}{\overset{=}{\sigma_3}} : \overset{\theta_1}{\overset{=}{\theta_3}} : \mathtt{s}_2)$$

  donde $\mathtt{s}_2$ contiene el valor asociado con la referencia desreferenciada por el paso de $\mapsto$. Luego, dado que:

$$(\sigma_1 : \theta_1 : s_1) \overset{rch}{\sim} (\sigma_2 : \theta_2 : s_1)$$

la operación de desreferenciado retornará el mismo resultado, si la realizamos sobre el almacenamiento σ_2. Luego:

$$(\sigma_2 : \theta_2 : s_1) \mapsto (\overset{\sigma_2}{\overline{\sigma_4}} : \overset{\theta_2}{\overline{\theta_4}} : s_2)$$

Finalmente, dado que los almacenamientos no son modificados, luego del paso de $\mapsto$, y dado que las porciones alcanzables de los almacenamientos en las configuraciones originales, coinciden de acuerdo a $\overset{rch}{\sim}$, entonces debe ser el caso de que la porción alcanzable de los almacenamientos obtenidos luego del paso de $\mapsto$ también deben coincidir, considerando el mismo conjunto raíz de referencias. Luego:

$$(\sigma_3 : \theta_3 : s_2) \overset{rch}{\sim} (\sigma_4 : \theta_4 : s_2)$$

- *El cómputo se explica mediante* $\rightarrow^{\theta}$ *o* $\rightarrow^{meta}$: consideremos el caso de la semántica de constructores de tablas. Podemos, entonces, reescribir la hipótesis como:

$$(\sigma_1 : \theta_1 : E[\![t]\!]) \overset{L}{\mapsto} (\overset{\sigma_3}{\overline{\sigma_1}} : \theta_1 \uplus \{(\overset{\theta_3}{\overline{tid}}, t, \ldots)\} : E[\![tid]\!])$$

Entonces podemos inferir que vale lo siguiente:

$$(\sigma_2 : \theta_2 : E[\![t]\!]) \overset{L}{\mapsto} (\overset{\sigma_4}{\overline{\sigma_2}} : \theta_2 \uplus \{(\overset{\theta_4}{\overline{tid}}, t, \ldots)\} : E[\![tid]\!])$$

donde, si fuese necesario, podríamos aplicar un renombre consistente de identificadores de tablas en $(\sigma_2 : \theta_2 : E[\![t]\!])$, de forma de que preserve su equivalencia con $(\sigma_1 : \theta_1 : E[\![t]\!])$ y, a la vez, *tid* sea un identificador de tabla fresco. Entonces, debe ser que ocurre lo siguiente:

$$(\sigma_3 : \theta_3 : E[\![tid]\!]) \overset{rch}{\sim} (\sigma_4 : \theta_4 : E[\![tid]\!])$$

□

Finalmente, el siguiente lema expresa una propiedad intuitiva que vale sobre configuraciones finales que resultan equivalentes según $\overset{rch}{\sim}$:

Lema 5.2.3 *Para configuraciones finales* $(\sigma_1 : \theta_1 : s)$ *y* $(\sigma_2 : \theta_2 : s)$*, tales que* $(\sigma_1 : \theta_1 : s) \overset{rch}{\sim} (\sigma_2 : \theta_2 : s)$*, entonces:*

$$result(\sigma_1 : \theta_1 : s) = result(\sigma_2 : \theta_2 : s)$$

Demostración. El resultado será una consecuencia inmediata de las definiciones de $result$, en Figura 5.17, y $\overset{rch}{\sim}$. Realizaremos análisis por caso en la estructura de s, para la configuración $(\sigma_1 : \theta_1 : s)$, considerando que se trata de una configuración final:

- $s = \textbf{return}\, v_1, \ldots, v_n$: por simplicidad, consideremos $n = 1$, y descartamos un posible contexto E en donde puede ocurrir **return**, pues no es tenido en cuenta por la noción de resultado definida por $result$. Para valores mayores de n el razonamiento es análogo:
  - $v_1 \in number \cup string$: entonces, $result(\sigma_1 : \theta_1 : s)$ y $result(\sigma_2 : \theta_2 : s)$ no dependen del contenido de los almacenamientos. Luego, $result(\sigma_1 : \theta_1 : s) = result(\sigma_2 : \theta_2 : s)$.
  - $v_1 \in tid \cup cid$: consideremos el caso en el que $v_1 = tid$ para cierto $tid \in \mathsf{dom}(\theta_1)$ (el razonamiento para el caso $v_1 \in cid$ será análogo). Luego, por definición de $result$:

$result(\sigma_1 : \theta_1 : \textbf{return } tid) = \sigma_1|_E : \theta_1|_T : \textbf{return } tid$

donde $\begin{cases} E = \bigcup_{r \in \text{dom}(\sigma_1),\ reach(r, \textbf{return } tid, \sigma_1, \theta_1)} r \\ T = \bigcup_{id \in \text{dom}(\theta_1),\ reach(id, \textbf{return } tid, \sigma_1, \theta_1)} id \end{cases}$

Luego, claramente $result(\sigma_1 : \theta_1 : \textbf{return } tid)$ está dependiendo de las porciones alcanzables de σ_1 y θ_1, a partir del conjunto raíz de referencias definido por *tid*. Dado que

$$(\sigma_1 : \theta_1 : tid) \overset{rch}{\sim} (\sigma_2 : \theta_2 : tid)$$

las porciones alcanzables de ambas configuraciones coinciden. Luego, $result(\sigma_1 : \theta_1 : s) = result(\sigma_2 : \theta_2 : s)$.

- $s = \textbf{error} v$: este caso es idéntico al caso anterior.
- $s = \textbf{;}$: luego, $result(\sigma_1 : \theta_1 : s)$ no está dependiendo del contenido de los almacenamientos, y lo mismo ocurre con $result(\sigma_2 : \theta_2 : s)$. Luego,

$$result(\sigma_1 : \theta_1 : s) = result(\sigma_2 : \theta_2 : s)$$

□

Propiedades preservadas por $\overset{GC}{\mapsto}$ Comenzaremos verificando una propiedad sencilla, que enuncia que los bindings alcanzables son preservados por un paso de $\overset{GC}{\mapsto}$, en el sentido de que continúen siendo alcanzables y no modificados: el valor asociado a la referencia es el mismo. Expresaremos esta propiedad con lo 2 siguientes lemas:

Lema 5.2.4 *Para una configuración bien formada $(\sigma_1 : \theta_1 : s)$, si $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$, para cierta configuración $(\sigma_2 : \theta_2 : s)$, entonces $\forall r \in dom(\sigma_1), reach(r, s, \sigma_1, \theta_1) \Rightarrow \sigma_1(r) = \sigma_2(r)$. La propiedad análoga vale para cualquier $id \in dom(\theta_1)$.*

Demostración. Sea $r \in dom(\sigma_1)$, $reach(r, s, \sigma_1, \theta_1)$. Luego, por definición de gc, en 5.1.2, y $\overset{GC}{\mapsto}$, debe ser el caso de que $gc(s, \sigma_1, \theta_1) = (\sigma_2, \theta_2)$ y $\sigma_1(r) = \sigma_2(r)$, pues $r \in dom(\sigma_1)$ tal que $reach(r, s, \sigma_1, \theta_1)$ no puede ser removida ni alterado su binding, según gc.

Para elementos de $dom(\theta_1)$ el razonamiento es análogo. □

Lema 5.2.5 *Para una configuración bien formada $(\sigma_1 : \theta_1 : s)$, si $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$, para cierta configuración $(\sigma_2 : \theta_2 : s)$, entonces $\forall l \in dom(\sigma_1) \cup dom(\theta_1), reach(l, s, \sigma_1, \theta_1) \Rightarrow reach(l, s, \sigma_2, \theta_2)$.*

Demostración. Demostraremos este enunciado realizando inducción en la cantidad mínima de operaciones de desreferenciado que son necesarias de realizar, de acuerdo a *reach* en la definición 5.1.1, para alcanzar una cierta referencia l para la cual vale $reach(l, s, \sigma_1, \theta_1)$ inline]Si hay tiempo agregar esto:Minimum path that can be shown to exist since we could define reach as the minimum fixed point **TODO**:Complete.. De la definición de *reach* vemos que, necesariamente, vale uno de los siguientes casos:

- $l \in s$: entonces debe darse el caso en el que también valga que $reach(l, s, \sigma_2, \theta_2)$.
- $\exists r \in dom(\sigma_1)$, $l \in \sigma_1(r)$, que se encuentra en un camino de alcanzabilidad de distancia mínima, partiendo del conjunto raíz de referencias: luego $reach(r, s, \sigma_1, \theta_1)$, y por hipótesis inductiva $reach(r, s, \sigma_2, \theta_2)$. A su vez, por lema 5.2.4, $\sigma_1(r) = \sigma_2(r)$. Luego, $l \in \sigma_2(r)$ y $reach(l, s, \sigma_2, \theta_2)$, por definición de *reach*.

- $\exists \mathit{tid} \in dom(\theta_1), l \in \pi_1(\theta_1(\mathit{tid}))$, que se encuentra en un camino de alcanzabilidad de distancia mínima, partiendo del conjunto raíz de referencias: luego $reach(\mathit{tid}, \mathsf{s}, \sigma_1, \theta_1)$, y, por hipótesis inductiva, $reach(\mathit{tid}, \mathsf{s}, \sigma_2, \theta_2)$. A su vez, por lema 5.2.4, $\theta_1(\mathit{tid}) = \theta_2(\mathit{tid})$. Luego $l \in \pi_1(\theta_2(\mathit{tid}))$ y $reach(l, \mathsf{s}, \sigma_2, \theta_2)$, por definición 5.1.1 de $reach$.
- $\exists \mathit{cid} \in dom(\theta_1), l \in \theta_1(\mathit{cid})$, que se encuentra en un camino de alcanzabilidad de distancia mínima, partiendo del conjunto raíz de referencias: el razonamiento es análogo al caso anterior. Concluimos en que $reach(l, \mathsf{s}, \sigma_2, \theta_2)$.
- $\exists \mathit{tid} \in dom(\theta_1), l \in \pi_2(\theta_2(\mathit{tid}))$, que se encuentra en un camino de alcanzabilidad de distancia mínima, partiendo del conjunto raíz de referencias: el razonamiento es análogo al caso anterior. Concluimos en que $reach(l, \mathsf{s}, \sigma_2, \theta_2)$.

□

Una consecuencia inmediata de los lemas previos es que lo que está preservando $\overset{GC}{\mapsto}$ es, en esencia, la propiedad capturada por $\overset{rch}{\sim}$:

Corolario 5.2.6 *Para configuraciones bien formadas* $(\sigma_1 : \theta_1 : \mathsf{s})$ *y* $(\sigma_2 : \theta_2 : \mathsf{s})$, *si* $(\sigma_1 : \theta_1 : \mathsf{s}) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : \mathsf{s})$, *entonces* $(\sigma_1 : \theta_1 : \mathsf{s}) \overset{rch}{\sim} (\sigma_2 : \theta_2 : \mathsf{s})$.

Demostración. Es un corolario directo de los lemas 5.2.4, 5.2.5 y la definición de $\overset{rch}{\sim}$. □

El siguiente lema nos va a permitir, entre otras cosas, el extender la propiedad de progreso de nuestra semántica dinámica $\mapsto$, en relación con $\vdash_{wfc}$, hacia la semántica que obtenemos al agregar GC, esto es, $\mapsto \cup \overset{GC}{\mapsto}$. En particular, esto nos permitirá garantizar que la noción de observaciones sobre programas, que hemos introducido en el comienzo de este capítulo, también está definida para esta última semántica, permitiéndonos, más adelante, enunciar la propiedad de corrección deseada para $\overset{GC}{\mapsto}$. inline]Agregar al comienzo mención a este lema, para justificar que tiene sentido hablar de observaciones sobre programas con gc

Lema 5.2.7 *Para una configuración* $(\sigma_1 : \theta_1 : s)$, *tal que* $\vdash_{wfc} (\sigma_1 : \theta_1 : s)$, *si* $(\sigma_1 : \theta_1 : s) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s)$, *para cierta configuración* $(\sigma_2 : \theta_2 : s)$, *entonces* $\vdash_{wfc} (\sigma_2 : \theta_2 : s)$.

Demostración. De la definición de $\overset{GC}{\mapsto}$ podemos inferir que el programa s no es alterado, luego de un paso de GC sin interfaces. Por otro lado, los lemas previos indican que un paso de $\overset{GC}{\mapsto}$ no introduce punteros colgantes en el programa. También indican que gc no realiza ninguna otra alteración de la información de los almacenamientos, además de remover bindings no alcanzables. Por lo tanto, debe ser el caso de que $\vdash_{wfc} (\sigma_2 : \theta_2 : s)$. □

Otra consecuencia evidente de la definición de $\overset{GC}{\mapsto}$, pero que es necesario capturar en un enunciado, es que, para todo programa, GC requiere sólo una cantidad finita de pasos para eliminar todo binding no alcanzable:

Lema 5.2.8 *Sobre una configuración bien formada* $(\sigma : \theta : s)$, *solo pueden aplicarse una cantidad finita de pasos de* $\overset{GC}{\mapsto}$.

Demostración. De la definición de gc en 5.1.2 y $\overset{GC}{\mapsto}$, si

$$(\sigma : \theta : s) \overset{GC}{\mapsto} (\sigma' : \theta' : s)$$

debe ser que σ' o θ' es un subconjunto propio de σ o θ, respectivamente. Luego, siendo que los almacenamientos son funciones parciales finitas, es claro que GC puede ser realizada solo una cantidad finita de veces. □

El marco teórico desarrollado hasta aquí ya nos permite verificar los últimos 2 enunciados fundamentales. El primero de ellos, el lema de aplazamiento, nos servirá de herramienta para resolver la demostración del enunciado de corrección de $\overset{GC}{\mapsto}$. El lema de aplazamiento, tomado de [31], codifica una intuición sencilla sobre GC sin interfaces: sobre una computación convergente, debería ser posible obtener otra que resulte de aplazar, o posponer, cualquier paso de GC hacia el final del cómputo, sin cambiar las observaciones.

En la formulación del lema de aplazamiento, usaremos el hecho de que $\overset{GC}{\mapsto}$ no altera el programa. Por otro lado, durante la demostración aprovecharemos la naturaleza no determinista de $\overset{GC}{\mapsto}$, para buscar pasos de GC que sean adecuados para nuestros razonamientos. Si pensamos en una traza de ejecución del modelo $\mapsto \cup \overset{GC}{\mapsto}$, para cada configuración no habrá, necesariamente, una única configuración resultado de un cómputo dado, sino que, en caso que sea posible GC, tendremos tantas configuraciones como pasos posibles de $\overset{GC}{\mapsto}$ existan, cada uno representando lo que haría un algoritmo de GC en particular. Con lo cual, si para configuraciones C_1 y C_2, enunciamos que se cumple $C_1 \overset{GC}{\mapsto} C_2$, es importante recordar que solamente significa que C_2 es solamente una de entre, posiblemente, varias configuraciones hacia las que llega desde C_1 mediante $\overset{GC}{\mapsto}$.

Lema 5.2.9 (Aplazamiento) *Para una configuración bien formada* $(\sigma_1 : \theta_1 : s_1)$*, si*

$$(\sigma_1 : \theta_1 : s_1) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_1) \mapsto (\sigma_3 : \theta_3 : s_2)$$

entonces $\exists(\sigma_4 : \theta_4 : s_2)$ *tal que:*

$$(\sigma_1 : \theta_1 : s_1) \mapsto (\sigma_4 : \theta_4 : s_2) \overset{GC}{\mapsto} (\sigma_3' : \theta_3' : s_2)$$

donde $(\sigma_3 : \theta_3 : s_2) \overset{rch}{\sim} (\sigma_3' : \theta_3' : s_2)$.

Demostración. Seguiremos la estructura de $\mapsto$ para razonar sobre el paso que transforma $(\sigma_2 : \theta_2 : s_1)$ en $(\sigma_3 : \theta_3 : s_2)$:

- *El cómputo se explica mediante* $\rightarrow^{s/e}$: entonces debe ser que, de ser requerida, toda la información de los almacenamientos, necesaria para hacer posible y determinar el cómputo, ya se encuentra en s_1. Luego, la hipótesis podría ser reescrita como:

  $$(\sigma_1 : \theta_1 : s_1) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_1) \mapsto (\sigma_2 : \theta_2 : s_2)$$

  Si tomamos $(\sigma_4 : \theta_4 : s_2) = (\sigma_1 : \theta_1 : s_2)$, entonces podemos aseverar que:

  $$(\sigma_1 : \theta_1 : s_1) \mapsto (\sigma_1 : \theta_1 : s_2) = (\sigma_4 : \theta_4 : s_2)$$

  en donde aprovechamos el hecho de que, para que pueda efectuarse el paso de $\mapsto$ anterior, el contenido de los almacenamientos no altera la posibilidad de efectivamente realizar el cómputo, ni determinan el resultado del mismo. Luego, por lema 5.2.1, si un binding ya estaba en condiciones de ser recolectado en $(\sigma_1 : \theta_1 : s_1)$ permanecerá en ese estado en $(\sigma_1 : \theta_1 : s_2)$. Luego, debido a la naturaleza no determinista de $\overset{GC}{\mapsto}$, podemos pedir por un paso de GC que remueva los mismos bindings en $(\sigma_1 : \theta_1 : s_1)$, que nos llevaron a obtener $(\sigma_2 : \theta_2 : s_1)$. Luego, debe ser que vale $(\sigma_1 : \theta_1 : s_2) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_2)$. Lo que hemos obtenido es que:

  $$(\sigma_1 : \theta_1 : s_1) \overset{L}{\mapsto} (\sigma_1 : \theta_1 : s_2) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_2)$$

Finalmente, $(\sigma_3 : \theta_3 : s_2) \overset{rch}{\sim} (\sigma'_3 : \theta'_3 : s_2)$ pues

$$(\sigma_3 : \theta_3 : s_2) = (\sigma_2 : \theta_2 : s_2) = (\sigma'_3 : \theta'_3 : s_2)$$

- *El cómputo se explica mediante* $\rightarrow^{\sigma}$: consideremos la operación de desreferenciado implícito de referencias a σ_1. Esto es, el paso de $\mapsto$ debe ser:

$$(\sigma_2 : \theta_2 : \overset{s_1}{\equiv} E[\![r]\!]) \mapsto (\overset{\sigma_3}{\equiv} \sigma_2 : \overset{\theta_3}{\equiv} \theta_2 : \overset{s_2}{\equiv} E[\![\sigma_2(r)]\!])$$

Luego, la hipótesis puede ser reescrita como:

$$(\sigma_1 : \theta_1 : \overset{s_1}{\equiv} E[\![r]\!]) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : \overset{s_1}{\equiv} E[\![r]\!]) \mapsto (\overset{\sigma_3}{\equiv} \sigma_2 : \overset{\theta_3}{\equiv} \theta_2 : \overset{s_2}{\equiv} E[\![\sigma_2(r)]\!])$$

Si tomamos $(\sigma_4 : \theta_4 : s_3) = (\sigma_1 : \theta_1 : s_2)$, podemos aseverar que:

$$(\sigma_1 : \theta_1 : \overset{s_1}{\equiv} E[\![r]\!]) \mapsto (\sigma_1 : \theta_1 : \overset{s_2}{\equiv} E[\![\sigma_2(r)]\!])$$

debido a que r es alcanzable en $(\sigma_1 : \theta_1 : s_1)$, y que el paso de $\overset{GC}{\mapsto}$ de la hipótesis preserva su binding, en el sentido expresado por el lema 5.2.4: por lo tanto, si fue posible realizar el desreferenciado en $(\sigma_2 : \theta_2 : s_1)$ (por hipótesis), debe ser posible realizarlo en $(\sigma_1 : \theta_1 : s_1)$, obteniendo el mismo resultado. Finalmente, por preservación de bindings listos para recolección luego de un paso de $\mapsto$, lema 5.2.1, y el comportamiento no determinista de $\overset{GC}{\mapsto}$, podemos pedir por un paso de GC que remueva exactamente los bindings necesarios para que valga:

$$(\sigma_1 : \theta_1 : s_2) \overset{GC}{\mapsto} (\overset{\sigma'_3}{\equiv} \sigma_2 : \overset{\theta'_3}{\equiv} \theta_2 : s_2)$$

Lo que obtuvimos es que:

$$(\sigma_1 : \theta_1 : s_1) \overset{L}{\mapsto} (\sigma_1 : \theta_1 : s_2) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_2)$$

Por otro lado, podemos aseverar que $(\sigma_3 : \theta_3 : s_2) \overset{rch}{\sim} (\sigma'_3 : \theta'_3 : s_2)$ pues

$$(\sigma_3 : \theta_3 : s_2) = (\sigma_2 : \theta_2 : s_2) = (\sigma'_3 : \theta'_3 : s_2)$$

- *El cómputo se explica mediante* $\rightarrow^{\theta}$: consideremos el caso de un constructor de tabla. Entonces, la hipótesis puede ser reescrita como:

$$(\sigma_1 : \theta_1 : s_1) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 : s_1) \mapsto (\overset{\sigma_3}{\equiv} \sigma_2 : \theta_2 \uplus \{(\mathit{tid}, t, \ldots)\} : \overset{s_2}{\equiv} E[\![\mathit{tid}]\!])$$

para un constructor de tabla t, e identificador de tabla tid, que, para nuestros propósitos, nos será útil si $\mathit{tid} \notin dom(\theta_1)$. Si no fuera el caso, podemos continuar nuestro razonamiento sobre la configuración resultante de α-convertir las referencias en $(\sigma_1 : \theta_1 : s_1)$ de forma tal de que $\mathit{tid} \notin dom(\theta_1)$. Es por casos como estos que no podemos enunciar una versión de aplazamiento tan fuerte como la de [31]: aquí no hablamos sobre la posibilidad de convergencia hacia una misma configuración, sino que necesitamos generalizar al caso de configuraciones equivalentes de acuerdo a $\overset{rch}{\sim}$.
Continuando nuestro razonamiento, si consideramos:

$$(\sigma_4 : \theta_4 : s_3) = (\sigma_1 : \theta_1 \uplus \{(\mathit{tid}, t, \ldots)\} : s_2)$$

podemos asegurar que vale:

$$(\sigma_1 : \theta_1 : s_1) \mapsto (\sigma_1 : \theta_1 \uplus \{(\mathit{tid}, t, \ldots)\} : s_2)$$

donde pedimos que el programa resultante sea exactamente $s_2 = E[\![\mathit{tid}]\!]$, nuevamente, por el no determinismo de la selección de identificadores frescos de tablas, en el momento de almacenarlas en memoria. Por lema 5.2.1 sabemos que todo binding que está en condiciones de ser recolectado en $(\sigma_1 : \theta_1 : s_1)$ permanece en ese estado en $(\sigma_1 : \theta_1 \uplus \{(\mathit{tid}, t, \ldots)\} : s_2)$. Más aún, tales bindings sólo pertenecen a σ_1 o a θ_1 (es decir, no ocurre esto con tid). Luego, por la naturaleza no determinista de $\overset{GC}{\mapsto}$ podemos pedir que se remueven solamente los bindings necesarios para que sea cierto que

$$(\sigma_1 : \theta_1 \uplus \{(tid, t, \ldots)\} : s_2) \overset{GC}{\mapsto} (\sigma_2 : \theta_2 \uplus \{(tid, t, \ldots)\} : s_2).$$

Luego, podemos aseverar de que vale lo siguiente:

$$(\sigma_1 : \theta_1 : s_1) \overset{L}{\mapsto} \ldots \overset{GC}{\mapsto} (\sigma_2 : \theta_2 \uplus \{(tid, t, \ldots)\} : s_2)$$

Finalmente, $(\sigma_3 : \theta_3 : s_2) \overset{rch}{\sim} (\sigma_3' : \theta_3' : s_2)$ dado que

$$(\sigma_3 : \theta_3 : s_2) = (\sigma_2 : \theta_2 \uplus \{(tid, t, \ldots)\} : s_2) = (\sigma_3' : \theta_3' : s_2)$$

□

El enunciado deseado de corrección de GC debiera mencionar que, para una configuración dada, las observaciones que podemos realizar bajo $\mapsto$ debieran ser las mismas que las que podemos realizar bajo $\overset{L+GC}{\mapsto} = \overset{L}{\mapsto} \cup \overset{GC}{\mapsto}$. De todos modos, bajo $\mapsto$ y $\overset{L+GC}{\mapsto}$ esperamos que se pueda demostrar que las observaciones son un singleton: los programas sólo divergen o sólo concluyen retornando un cierto resultado o error. Dada esta observación, podemos cambiar el enunciado de corrección para llegar a una propiedad que puede ser demostrada con menor esfuerzo: dada una configuración, bajo $\mapsto$ su ejecución concluye *sí y sólo si* concluye también bajo $\overset{L+GC}{\mapsto}$, y, en ambos casos lo retornado (sean valores o un error) es lo mismo. Es decir, en lugar de contemplar cada posible observación para un programa dado, nos concentraremos en el caso de convergencia. Preservación de cómputos divergentes será una consecuencia de la estructura de doble implicación del enunciado.

Teorema 5.2.10 (Corrección de GC sin interfaces) *Para una configuración bien formada* $\sigma : \theta : s$,

$$(\sigma : \theta : s) \Downarrow_{\mapsto} (\sigma' : \theta' : s') \Leftrightarrow (\sigma : \theta : s) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma'' : \theta'' : s'')$$

y $result(\sigma' : \theta' : s') = result(\sigma'' : \theta'' : s'')$.

Demostración. Asumamos que $(\sigma : \theta : s) \Downarrow_{\mapsto} (\sigma' : \theta' : s')$. Luego $(\sigma' : \theta' : s')$ es una configuración final sobre la cual $result$ está definido. Dado que $\mapsto \subseteq \overset{L+GC}{\mapsto}$, siempre es posible emular la traza de ejecución previa, por ejemplo, no usando pasos $\overset{GC}{\mapsto}$. Luego, $(\sigma : \theta : s) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma' : \theta' : s')$, de donde se sigue que el cómputo retorna lo mismo bajo $\overset{L+GC}{\mapsto}$ y $\overset{L}{\mapsto}$.

Por otro lado, asumamos que

$$(\sigma : \theta : s) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma' : \theta' : s')$$

Luego, debe ser que existe una traza finita de pasos de cómputo, como la siguiente:

$$(\sigma : \theta : s) \overset{L+GC}{\mapsto} (\sigma_1 : \theta_1 : s_1) \overset{L+GC}{\mapsto} \ldots \overset{L+GC}{\mapsto} (\sigma_n : \theta_n : s_n)$$

donde $(\sigma_n : \theta_n : s_n) = (\sigma' : \theta' : s')$ es la configuración final, sobre la que $result$ está definido.

Mediante un razonamiento inductivo en la cantidad de pasos de cómputo y el uso del lema de aplazamiento 5.2.9, se puede demostrar que la traza anterior puede ser reescrita como:

$$(\sigma : \theta : s) \mapsto \ldots \overset{L}{\mapsto} (\sigma_{i'} : \theta_{i'} : s_{i'}) \overset{GC}{\mapsto} \ldots \overset{GC}{\mapsto} (\sigma_{n'} : \theta_{n'} : s_{i'})$$

donde todo cómputo que no involucra GC se realiza al comienzo. De este modo, obtuvimos una traza convergente consistente solamente en pasos de $\mapsto$. Es decir:

$$(\sigma : \theta : s) \Downarrow_{\mapsto} (\sigma_{i'} : \theta_{i'} : s_{i'})$$

Lo que resta por verificar es que el resultado también es preservado. Con tal objetivo, notar que el lema de aplazamiento también nos dice que:

$$(\sigma_{n'} : \theta_{n'} : s_{i'}) \overset{rch}{\sim} (\sigma_n : \theta_n : s_n)$$

Luego, debido a que configuraciones finales que son $\overset{rch}{\sim}$ representan el mismo resultado, de acuerdo al lema 5.2.3, se sigue que:

$$result(\sigma_{n'} : \theta_{n'} : s_{i'}) = result(\sigma_n : \theta_n : s_n)$$

Finalmente, debido a que $\overset{rch}{\sim}$ está cerrada bajo pasos de $\overset{GC}{\mapsto}$, lema 5.2.6, debe ser el caso de que:

$$(\sigma_{i'} : \theta_{i'} : s_{i'}) \overset{rch}{\sim} (\sigma_{n'} : \theta_{n'} : s_{i'})$$

Entonces,

$$result(\sigma_{i'} : \theta_{i'} : s_{i'}) = result(\sigma_{n'} : \theta_{n'} : s_{i'}) = result(\sigma_n : \theta_n : s_n)$$

□

Una conclusión inmediata del teorema anterior es que, bajo $\overset{L+GC}{\mapsto}$, el conjunto de observaciones es un singleton, a pesar del no determinismo de $\overset{GC}{\mapsto}$:

Corolario 5.2.11 (Unicidad de observaciones bajo GC) *Para una configuración bien formada* $\sigma : \theta : \mathtt{s}$, $|obs(\sigma : \theta : \mathtt{s}, \overset{L+GC}{\mapsto})| = 1$

Demostración. Se sigue inmediatamente del teorema anterior y el determinismo de programas bajo $\mapsto$, enunciado en el corolario 4.3.3 del capítulo anterior. □

Finalmente, podemos enunciar corrección de GC de un modo más natural, en término de la preservación de observaciones sobre un programa dado:

Corolario 5.2.12 (Preservación de observaciones bajo GC) *Para una configuración bien formada* $\sigma : \theta : \mathtt{s}$,

$$(\sigma : \theta : \mathtt{s}, \mapsto) \equiv (\sigma : \theta : \mathtt{s}, \overset{L+GC}{\mapsto})$$

Demostración. Se sigue de manera directa del teorema 5.2.10 y el corolario anterior: si $result(\sigma', \theta', \mathtt{s}') \in obs(\sigma : \theta : \mathtt{s}, \overset{L}{\mapsto})$, para $(\sigma : \theta : \mathtt{s}) \Downarrow_{\mapsto} (\sigma' : \theta' : \mathtt{s}')$, por teorema 5.2.10 lo anterior ocurre sí y sólo si $(\sigma : \theta : \mathtt{s}) \Downarrow_{\overset{L+GC}{\mapsto}} (\sigma'' : \theta'' : \mathtt{s}'')$, donde

$$result(\sigma', \theta', \mathtt{s}') = result(\sigma'', \theta'', \mathtt{s}'')$$

Luego, $result(\sigma', \theta', \mathtt{s}') \in obs(\sigma : \theta : \mathtt{s}, \overset{L+GC}{\mapsto})$. Luego, por la unicidad de observaciones bajo $\mapsto$ y $\overset{L+GC}{\mapsto}$, entonces $obs(\sigma : \theta : \mathtt{s}, \overset{L}{\mapsto}) = obs(\sigma : \theta : \mathtt{s}, \overset{L+GC}{\mapsto})$.

Si $\bot \in obs(\sigma : \theta : \mathtt{s}, \overset{L}{\mapsto})$, por corrección de GC debe ser que lo anterior ocurre sí y sólo si $\bot \in obs(\sigma : \theta : \mathtt{s}, \overset{L+GC}{\mapsto})$. Nuevamente, por unicidad de observaciones, corolario 5.2.11, debe ser que:

$$obs(\sigma : \theta : \mathtt{s}, \overset{L}{\mapsto}) = obs(\sigma : \theta : \mathtt{s}, \overset{L+GC}{\mapsto}$$

□

6

Mecanización

La formalización de la semántica fue realizada en paralelo con su mecanización en PLT Redex[10]. Esta herramienta nos ayudó a detectar problemas en nuestros primeros intentos de formalización, y nos permitió experimentar con nuevas ideas, antes de incorporarlas en la formalización. Finalmente, nos permitió ejecutar parte de los tests de la suite de test del intérprete oficial,[*] obteniendo así evidencia de que nuestra semántica se corresponde con la implementación oficial. Finalmente, el soporte de PLT Redex para la mecanización de sistemas formales y testeo aleatorio, nos permitió mecanizar nuestra definición de programas bien formados, como un sistema formal, y realizar testeo aleatorio de la propiedad de corrección asociada al sistema.

En lo que sigue discutiremos detalles técnicos sobre la mecanización incluida en el presente trabajo.

Correspondencia con el modelo formal

Mientras que PLT Redex es presentado como una herramienta para mecanizar semánticas operaciones, lo que provee es soporte para los conceptos de semántica de reducciones con contextos de evaluación, tal como fue introducida en §3.1: es decir, soporte para definición de lenguajes, contextos de evaluación y reglas que expresan contracciones, esto es, que captura relaciones sobre términos del lenguaje; junto con las necesarias meta-funciones. Esto es todo lo que forma parte de un modelo en PLT Redex, a lo que debemos agregarle la posibilidad de definir sistemas de inferencia, que permiten, por ejemplo, definir algoritmos de *type checking*.

En la práctica, lo anterior implica que ciertos elementos semánticos de nuestro modelo tendrán que ser codificados como frases de un lenguaje, inclusive a pesar de que tales conceptos no son entendidos como tales. Un ejemplo consistiría en la necesidad de definir, a los fines de poder mecanizar con PLT Redex, almacenamientos como frases de un lenguaje (junto con las consecuentes manipulaciones de los mismos con operaciones propias de términos del lenguaje), a pesar de que son entendidos como funciones parciales.

Sin embargo, debido al soporte para conceptos de semántica de reducciones que son ampliamente empleados en nuestro modelo; sistemas formales; facilidades de testeo aleatorio; su implementación como un DSL sobre Racket (lo que permite incluir código Racket aleatorio; por ejemplo, para implementar servicios de librería); y su facilidad de uso, PLT Redex resultó adecuado como herramienta de prototipado y experimentación con nuestro modelo.

Cobertura de la suite de tests

No nos fue posible utilizar la suite de tests completa, por las siguientes razones:

[*] Disponible en `www.lua.org/tests/`.

Archivo	Facilidades testeadas	Cobertura
vararg.lua	vararg	100 %
events.lua	metatablas	90.4 %
calls.lua	funciones y llamadas	82.81 %
math.lua	nros. y lib. math	82.2 %
constructs.lua	sintaxis y op. short-circuit.	63.18 %
locals.lua	variables locales y entorno	62.3 %
nextvar.lua	tables, next, y for	53.24 %
closure.lua	clausuras	48.5 %
sort.lua	(parte de) librería table	24.1 %

Figura 6.1: Cobertura de la suite de test de Lua 5.2.

- Facilidades del lenguaje no cubiertas por nuestra formalización: co-rutinas, sentencia **goto**, algunas función de la librería estándar (aquellas relacionadas con manejos de archivos) y otras librerías implementadas en C (bit32, coroutine, debug, io, *etc.*).
- Presencia de tests diseñado para testear detalles de implementación del intérprete, que no son propias del lenguaje: implementación de *tail call*, manipulación de tablas de gran tamaño, generación de bytecode y optimizaciones. De acuerdo a los autores de Lua, el objetivo de la suite de test que proveen consiste solamente en testear la implementación oficial de Lua, y no como manera de verificar implementaciones alternativas,[*].

inline]Alguna respuesta parcial a cómo mejorar esta cobertura?, comentar sobre la experiencia con LuaRocks?

inline]Mencionamos sobre los problemas para mecanizar GC?, la necesidad de mantener una representación explícita del environment?

En la práctica, de los 25 archivos .lua presentes en la suite de test, dedicados a verificar facilidades del lenguaje, estamos en condiciones de traducir, ejecutar y comparar con respecto nuestra mecanización, 9 archivos. Figura 6.1 presenta la cobertura en porcentajes de LOCs testeada, completando 1256 LOCs que fueron verificada exitosamente. Cada archivo de la suite consiste en una secuencia de aserciones a verificar, sobre resultados esperados de programas válidos, como también de programas que resultan en errores.

Es importante remarcar que cada archivo y línea que no pudo ser incluida en esta verificación, se debió a las razones explicadas anteriormente, mientras que todo archivo y linea que cae dentro del alcance de este trabajo han sido exitosamente verificados por nuestro modelo.

Testeo aleatorio de progreso

En la sección §4.3 se mencionó que seguimos el enfoque tradicional para formular la propiedad de progreso de nuestra semántica operacional, como corolario de la correspondiente propiedad de un sistema formal que captura la noción de programas bien formados (más exactamente, configuraciones bien formadas). Las facilidad de PLT Redex para implementar sistemas formales y realizar testeo aleatorio (llamada redex-check ([42])) nos permitieron mecanizar nuestra noción de programas bien formados y testear la propiedad de corrección asociada.

Notablemente, los términos generados pseudo-aleatoriamente por redex-check mostraron errores inesperados en la mecanización de nuestra semántica dinámica, que no fueron descubiertos durante el testeo

[*] Ver www.lua.org/wshop15/Ierusalimschy.pdf .

con respecto a la suite de test de Lua. Los términos generados, intrincados y difíciles de descifrar resultaron ser una herramienta útil para descubrir casos mal definidos en nuestras relaciones semánticas y meta-funciones. Como se menciona en [29], el generador procede de manera completamente desinhibida, sin contemplaciones o conocimiento de la semántica de los términos que combina.

En nuestra experiencia, pudimos pedirle al generador que produzca $5 * 10^6$ términos, sobre los cuales verificamos su buena formación. Para cada término bien formado, verificamos el predicado de corrección del sistema formal que captura la noción de configuraciones bien formadas.

Incluimos todos los archivos necesarios para realizar estos tests, junto con la mecanización de nuestro modelo, en `github.com/Mallku2/lua-redex-model`.

7 Trabajos relacionados

Con respecto a la semántica del fragmento de Lua sin GC, los trabajos fundamentales para nuestra labor consisten en [11-14]. De estos trabajos incorporamos el modelo semántica, junto con la idea de mecanizar el mismo en PLT Redex. Las diferencias radican en:

- Facilidades particulares provistas por Lua, como su mecanismo de metaprogramación (el cual, aunque semejante en posibilidades a lo ofrecido por Python, a través de la construcción metaclass, estas facilidades no son cubiertas en la formalización presentada en [12]).
- El enfoque empleado, que no consiste en la identificación de un conjunto núcleo de facilidades (enfoque habilitado por el tamaño reducido de Lua, y otras razones ya desarrolladas en §1)
- En nuestra presentación de la semántica, haciendo foco en identificar el rol de los recursos de semántica de reducciones utilizados, y en distinguir nuestra semántica operacional de una semántica de reducciones tradicional
- Mayor foco en especificación y testeo aleatorio de las propiedades de nuestro modelo, utilizando las facilidades de PLT Redex.
- Para la semántica de Lua sin GC, realizamos los testeos directamente sobre nuestra mecanización, en lugar de implementar otro intérprete, como normalmente se hace evitar los problemas de performance que surgen al ejecutar la mecanización [11, 12]. Al proceder de este modo, incrementamos la confianza en nuestra mecanización, a expensas de mayor inversión de tiempo en testing.

Al no seguir el enfoque de lenguaje núcleo, evitamos las conocidas complejidades que surgen en el modelo resultante[5, 8]): código compilado más verboso, junto con una confianza reducida con respecto a la correspondencia entre la semántica formal provista y la especificación original del lenguaje (típicamente en la forma de un manual de referencia que explica la semántica en lenguaje natural). Adicionalmente, al mantener la proximidad con el lenguaje original, allanamos el camino hacia una formalización y mecanización que también podrían ser utilizadas por personas que desarrollan programas con Lua, con fines de verificación o para desarrollar herramientas de análisis; ya que nuestra semántica va a reflejar de mejor manera la intuición que estas personas tienen sobre la semántica de Lua.

Con respecto a otros hitos en semántica formal de lenguajes reales, podemos mencionar JSCert ([8]): una formalización del documento ES5 en el asistente e pruebas Coq, junto con un intérprete extraído de esta formalización (JSRef). La formalización consiste en una semántica operacional *big-step*. A los fines de incrementar la confianza respecto a la correspondencia entre los formalizado y lo especificado en ES5, las personas detrás de JSCert reconocen la importancia de la revisión de su modelo en Coq por personas de diferentes áreas, desde personas que desarrollan herramientas de análisis para JavaScript, quienes desarrollan VMs e inclusive quienes especifican los documentos ECMA. En nuestro proyecto, además de pesonas dedicadas al área de semántica formal y a quienes implementan VMs para Lua, apuntamos a incluir personas que desarrollan software con Lua. Esto podría ser difícil de conseguir sobre un modelo en un asistente de pruebas como Coq, debido a, como reconocen las personas detrás de [8], utilizar asistentes de pruebas requiere un conocimiento considerablemente mayor que el necesario para utilizar, por ejemplo, herramientas diseñadas especificamente para mecanizar especificaciones de lenguajes. Mientras que a futuro sería interesante obtener una mecanización de las pruebas de las propiedades deseables de nuestro modelo, quizás utilizando Coq, también queremos aprovechar la facilidad de uso de PLT Redex, como primer paso entre el entendimiento informal de Lua, y uno formal, pero mecanizado en un asistente de pruebas.

[5] introduce una semántica operacional small-step para la especificación ECMA-262, 3 edición, incluyendo pruebas de varias propiedades del modelo. Las personas detrás de este trabajo reconocen que, al

definir la semántica de cada construcción del modo especificado en el documento ECMA, se maximiza la posibilidad de que el modelo sea correcto. Mientras que este enfoque resultó en un modelo que heredó el tamaño y las complejidades del lenguaje formalizado, la experiencia es interesante para nuestras investigaciones con Lua, ya que estamos tratando con un lenguaje de menor tamaño y más sencillo en su semántica.

Específicamente sobre semántica de Lua, solamente conocemos otra experiencia: [9] presenta unas semántica operacional para Lua 5.2, consistente en un λ-cálculo extendido con un conjunto reducido de conceptos, en términos de los cuales poder explicar Lua 5.2, siguiendo el enfoque de Featherweight Java ([43]). Contempla un subconjunto de las facilidades que presentamos en este trabajo y provee una implementación de referencia en Haskell. Se trata de un enfoque interesante, pero que padece de las limitaciones que ya hemos descrito, que surgen al considerar el estudio de un lenguaje núcleo, en lugar de intentar la formalización del lenguaje completo.

GC En [33], Marcus Amorim Leal y col. presentan una semántica formal para una λ-cálculo extendido con referencias (fuertes y débiles), y finalizadores. De la literatura consultada, este es el único trabajo en donde ambas interfaces con el recolector son incluidas en un mismo modelo. La semántica presentada para finalizadores no impone un orden de ejecución entre finalizadores, como ocurre en Lua 5.2, y la semántica de objetos resucitados no difiere de la de los demás. A su vez, no hay interacción entre referencias débiles y finalización, del modo en el que ocurre en nuestro modelo.

En [31], Morrisett y col. presentan una semántica de reducciones para modelar GC (denominada λ_{GC}), que no incluye interfaces con el recolector. La teoría desarrollada para demostrar corrección de GC sirvió de inspiración para nuestro desarrollo, aunque el desarrollo aquí presentado abunda en mayores detalles y, por las características de ambos modelos, en nuestro trabajo necesitamos pensar en términos de configuraciones equivalentes, según $\overset{rch}{\sim}$, mientras que en λ_{GC} es posible demostrar teoremas de convergencia hacia un mismo término, reflejado en una propiedad de *aplazamiento* diferente, y la posibilidad de demostrar una propiedad diamante. A su vez, la definición de basura no se explica en términos de alcanzabilidad, sino que se captura en términos de la aparición de variables libres, al remover un binding dado, del heap. En [34] se muestra que definir GC en términos de alcanzabilidad resulta en un modelo más expresivo, reflejado en la posibilidad de emular mayor cantidad de algoritmos de GC basados en alcanzabilidad. En nuestro trabajo seguimos este camino.

En [32], Donnelly y col. extienden λ_{GC} incluyendo referencias débiles. Utilizan su modelo (denominado λ_{weak}) para explicar la semántica de las referencias débiles clave/valor (concepto semejante a las ephemerons de Lua) presentadas en la implementación GHC de Haskell. A su vez, presentando un sistema de tipos para su modelo, y muestro cómo utilizarlo para la recolección de basura alcanzable (es decir, *basura semántica*). Finalmente, tocan el problema de la introducción de no determinismo en la evaluación de un programa que hace uso de referencias débiles. Proveen un criterio sintáctico (decidible) para reconocer programas con buen comportamiento con respecto a GC (es decir, con comportamiento determinista a pesar de utilizar referencias débiles, y proponer un criterio semántica para caracterizar una clase mayor de programas con este mismo comportamiento determinista. Debido a que λ_{weak} está directamente derivado de λ_{GC}, presenta la misma falta de expresividad de no un modelo que no está basado en alcanzabilidad. Por otro lado, la teoría desarrollada para su modelo está basada en observaciones de programas que contemplan la posibilidad de comportamiento no determinista. Siendo que el no determinismo es un fenómeno también presente en nuestro modelo, su teoría nos resulta útil para el desarrollo de la nuestra.

El concepto de ephemerons, tal como es implementado en Lua, es descrito en [38]. Aunque no se presenta un modelo formal en términos de los cuales entender la semántica del concepto. Uno de los trabajos clásicos en donde se abunda sobre el concepto de ephemerons es [37], aunque, nuevamente, la discusión no se realiza sobre un modelo formal que incluya el concepto.

8 Conclusiones y trabajos a futuro

Hemos presentado una semántica operacional small-step para un conjunto amplio de conceptos del lenguaje de programación Lua 5.2, tal como es especificado en su manual de referencia e implementación oficial.

La semántica resuelve la mayoría de las facilidades complejas de Lua, tales como metatablas, manejo de errores, servicios de librería y GC con sus 2 interfaces.

Proveemos una mecanización en PLT Redex, de nuestra semántica formal. La porción de la suite de test del intérprete oficial, que corresponde a conceptos incluidos en nuestro nuestro modelo, sin incluir GC, ha sido verificada exitosamente, proveyendo evidencia de que nuestro modelo se corresponde con lo especificado en el manual de referencia y en el intérprete oficial.

Utilizando las facilidades de PLT Redex para la mecanización de sistemas formales y testeo aleatorio, definimos la noción de buena formación de programas y testeamos su propiedad de corrección. De ese modo, a la vez que obtuvimos evidencia de la deseable propiedad de progreso de nuestra semántica, también pudimos detectar errores en nuestra semántica que no habían sido puestos en evidencia durante la verificación con la suite de test de Lua.

Para GC, desarrollamos un marco teórico dentro del cual pudimos verificar varios lemas y teoremas fundamentales (para el caso de GC sin interfaces) que sirven como chequeo de sanidad de nuestra semántica. A su vez, proveemos un marco teórico dentro del cual estudiar propiedades de GC con interfaces, y algoritmos de GC basados en alcanzabilidad.

La semántica formal desarrollada, su mecanización y su suite de test constituyen una herramienta que podría ser de utilidad tanto para personas del área de semántica formal como para quienes desarrollan aplicaciones con Lua, tanto para comprender como para experimentar con nuevas facilidades para el lenguaje.

Trabajos a futuro Existen varias vías de desarrollo del trabajo aquí presentado; entre ellas:

- Extender el modelo con facilidades no incluidas: corutinas; sentencia **goto**; facilidades introducidas en Lua 5.2, como nuevos operadores y tipos numéricos. De entre estos conceptos, el tratamiento formal de la sentencia **goto** podría requerir un uso distinto de un concepto ya presente en el modelo:: contextos de evaluación, a los fines de mantener la porción del programa ya ejecutada, como se realiza en [44]. El estilo small-step de nuestra semántica parece ser adecuada para modelar la interacción de la sentencia **goto** con variables con alcance de bloque, tal como se describe en el mencionado trabajo.
- La semántica formal y su mecanización podrían servir de base para la especificación, implementación y verificación formal de herramientas de análisis estático de programas Lua. Ya existen herramientas y extensiones para el lenguaje que realizan esto (como Luacheck (`github.com/mpeterv/luacheck`), Ravi (`ravilang.github.io`), y Typed Lua [45]), pero carecen de garantías formales de corrección, al no disponer de una semántica dinámica formalizada.
- La experiencia con PLT Redex sugiere que, a pesar de ser una adecuada herramienta para el prototipado de modelos de semántica operacional, y la experimentación con los mismos, no puede consistir en el último paso en la tarea de formalización. Sería interesante poder extraer código PLT Redex hacia algún asistente de pruebas, dentro del cual poder demostrar las propiedades deseadas. De este modo, se puede ofrecer una mecanización lista para ser utilizada por una audiencia más amplia, mientras que, a su vez, se puede disponer de un modo práctico de trasladar la

mecanización a un asistente de pruebas, para verificar propiedades de cada iteración del modelo en PLT Redex; a su vez, nos permitiría extraer un intérprete verificado. Esta última posibilidad también podría ofrecer una solución a la pobre performance de las mecanizaciones en PLT Redex, que a veces [11, 12] impone la necesidad de implementar otro intérprete, siguiendo lo que especifica la mecanización, y solamente realizar testing sobre este intérprete en lugar de la mecanización en PLT Redex, reduciendo la confianza sobre la corrección de esta última.

Apéndices

A

Semántica operacional

Comenzamos introduciendo la sintaxis del lenguaje, junto con las extensiones realizadas al mismo a los fines de formalizar la semántica dinámica, y los contextos de evaluación:

s ::= **;**
| **break**
| **return** *e*
| **$statFunCall** *e* **(** *e* **,** ... **)**
| **$statFunCall** *e* **:** *x* **(** *e* **,** ... **)**
| *var* **,** ... **=** *e* **,** ...
| **if** *e* **then** *s* **else** *s*
| **while** *e* **do** *s*
| **local** *x* **,** ... **=** *e* **,** ... **in** *s*
| *s s*

v ::= **nil** | **false** | **true** | *number*
| *string*

e ::= *v* | **...** | *var* | **(** *e* **)** | *e binop e*
| *unop e* | **{** *field* **,** ... **}**
| *e* **(** *e* **,** ... **)** | *e* **:** *x* **(** *e* **,** ... **)**
| **$builtIn** *svc* **(** *e* **,** ... **)**
| **function** / **(** *x* **,** ... **)** *s*
| **function** / **(** *x* **,** ... **,** **...)** *s*

var ::= *x* | *e* **[** *e* **]**

svc ::= nombres de servicios de librería

field ::= *e* | **[** *e* **]** **=** *e*

efield ::= *v* | **[** *v* **]** **=** *v*

strictbinop ::= **+** | **-** | ***** | / | ^ | **%**
| **..** | **<** | ≤ | > | ≥
| **==**

binop ::= *strictbinop* | **and** | **or**

unop ::= **-** | **not** | **#**

(a) Lenguaje.

s ::= ... | **$nextItWhile** *e* **do** *s*
| ⦇ *s* ⦈*label*

v ::= ... | *tid*

e ::= ... | *r*
| **$err** *v*
| **<** *e* **,** ... **>**
| ⦇ *e* ⦈*label*

evar ::= *r* | *v* **[** *v* **]**

efield ::= *v* | **[** *v* **]** **=** *v*

label ::= ARITHWO
| STRCONCATWO
| ORDCOMPWO
| EQFAIL
| NEGWO
| STRLENWO
| WRONGKEY
| NONTABLE
| WFUNCALL
| RETEXP
| PROTMD
| RETSTAT
| BREAK

(b) Extensiones de tiempo de ejecución.

$$
\begin{array}{rcl}
E_{lf} & ::= & [\,] \\
& | & \textbf{return}\ v\ ,\ \ldots\ ,\ E_{\text{lf}}\ ,\ e\ ,\ \ldots \\
& | & \textbf{\$statFuncall}\ E_{\text{lf}}\ (\ e\ ,\ \ldots\ ) \\
& | & \textbf{\$statFuncall}\ v\ (\ v\ ,\ \ldots\ ,\ E_{\text{lf}}\ ,\ e\ ,\ \ldots\ ) \\
& | & \textbf{\$statFuncall}\ E_{\text{lf}}\ \textbf{:}\ x\ (\ e\ ,\ \ldots\ ) \\
& | & evar\ ,\ \ldots\ ,\ E_{\text{lf}}\ [\ e\ ]\ ,\ \ldots\ =\ e\ ,\ \ldots \\
& | & evar\ ,\ \ldots\ ,\ v\ [\ E_{\text{lf}}\ ]\ ,\ \ldots\ =\ e\ ,\ \ldots \\
& | & evar\ ,\ \ldots\ =\ v\ ,\ \ldots\ ,\ E_{\text{lf}}\ ,\ \ldots \\
& | & \textbf{if}\ E_{\text{lf}}\ \textbf{then}\ s\ \textbf{else}\ s \\
& | & \textbf{local}\ x\ ,\ \ldots\ =\ v\ ,\ \ldots\ ,\ E_{\text{lf}}\ ,\ e\ ,\ \ldots\ \textbf{in}\ s \\
& | & E_{\text{lf}}\ s \\
& | & (\ E_{lf}\ ) \\
& | & E_{lf}\ binop\ e \\
& | & v\ strictbinop\ E_{lf} \\
& | & unop\ E_{lf} \\
& | & \{\ efield\ ,\ \ldots\ ,\ [\ v\ ]\ =\ E_{lf}\ ,\ \ldots\ \} \\
& | & \{\ efield\ ,\ \ldots\ ,\ [\ E_{lf}\ ]\ =\ e\ ,\ \ldots\ \} \\
& | & \{\ efield\ ,\ \ldots\ ,\ E_{lf}\ ,\ \ldots\ \} \\
& | & E_{lf}\ (\ e\ ,\ \ldots\ ) \\
& | & v\ (\ v\ ,\ \ldots\ ,\ E_{lf}\ ,\ e\ ,\ \ldots\ ) \\
& | & E_{lf}\ \textbf{:}\ x\ (\ e\ ,\ \ldots\ ) \\
& | & \textbf{\$builtIn}\ x\ (\ v\ ,\ \ldots\ ,\ E_{lf}\ ,\ e\ ,\ \ldots\ ) \\
& | & E_{lf}\ [\ e\ ] \\
& | & v\ [\ E_{lf}\ ] \\
& | & <\ v\ ,\ \ldots\ ,\ E_{lf}\ ,\ e\ ,\ \ldots\ >
\end{array}
$$

$$
\begin{array}{rcl}
E_{np} & ::= & E_{lf} \\
& | & (\!|\ E_{np}\ |\!)_{\text{RETSTAT}} \\
& | & (\!|\ E_{np}\ |\!)_{\text{RETEXP}} \\
& | & (\!|\ E_{np}\ |\!)_{\text{BREAK}}
\end{array}
$$

$$
\begin{array}{rcl}
E & ::= & E_{np} \\
& | & (\!|\ E\ |\!)_{\text{PROTMD}} \\
& | & (\!|\ E\ |\!)_{\text{PROTMD}\ v}
\end{array}
$$

$$
\begin{array}{rcl}
E_{\text{tel}} & ::= & v\ ,\ \ldots\ ,\ [\![\ ]\!]\ ,\ e\ ,\ \ldots \\
E_{\text{t}} & ::= & \textbf{return}\ E_{\text{tel}} \\
& | & \textbf{\$statFunCall}\ [\![\ ]\!]\ (\ e\ ,\ \ldots\ ) \\
& | & \textbf{\$statFunCall}\ v\ (\ E_{\text{tel}}\ ) \\
& | & \textbf{\$statFunCall}\ [\![\ ]\!]\ \textbf{:}\ x\ (\ e\ ,\ \ldots\ ) \\
& | & evar\ ,\ \ldots\ ,\ [\![\ ]\!]\ [\ e\ ]\ ,\ \ldots\ =\ e\ ,\ \ldots \\
& | & evar\ ,\ \ldots\ ,\ v\ [\ [\![\ ]\!]\ ]\ ,\ \ldots\ =\ e\ ,\ \ldots \\
& | & evar\ ,\ \ldots\ =\ E_{\text{tel}} \\
& | & \textbf{if}\ [\![\ ]\!]\ \textbf{then}\ s\ \textbf{else}\ s \\
& | & \textbf{local}\ x\ ,\ \ldots\ =\ E_{\text{tel}}\ \textbf{in}\ s \\
& | & (\ [\![\ ]\!]\ ) \\
& | & [\![\ ]\!]\ binop\ e \\
& | & v\ strictbinop\ [\![\ ]\!] \\
& | & unop\ [\![\ ]\!] \\
& | & \{\ efield\ ,\ \ldots\ ,\ [\ [\![\ ]\!]\ ]\ =\ e\ ,\ field\ ,\ \ldots\ \} \\
& | & \{\ efield\ ,\ \ldots\ ,\ [\ v\ ]\ =\ [\![\ ]\!]\ ,\ field\ ,\ \ldots\ \} \\
& | & \{\ efield\ ,\ \ldots\ ,\ [\![\ ]\!]\ ,\ field\ ,\ \ldots\ \} \\
& | & [\![\ ]\!]\ (\ e\ ,\ \ldots\ ) \\
& | & v\ (\ E_{\text{tel}}\ ) \\
& | & [\![\ ]\!]\ \textbf{:}\ x\ (\ e\ ,\ \ldots\ ) \\
& | & \textbf{\$builtIn}\ x\ (\ E_{\text{tel}}\ ) \\
& | & [\![\ ]\!]\ [\ e\ ] \\
& | & v\ [\ [\![\ ]\!]\ ] \\
& | & <\ E_{\text{tel}}\ >
\end{array}
$$

$$
\begin{array}{rcl}
E_{\text{uel}} & ::= & v\ ,\ \ldots\ ,\ [\![\ ]\!] \\
E_{\text{u}} & ::= & \textbf{return}\ E_{\text{uel}} \\
& | & \textbf{\$statFunCall}\ v\ (\ E_{\text{uel}}\ ) \\
& | & evar\ ,\ \ldots\ =\ E_{\text{uel}} \\
& | & \textbf{local}\ x\ ,\ \ldots\ =\ E_{\text{uel}}\ \textbf{in}\ s \\
& | & \{\ efield\ ,\ \ldots\ ,\ [\![\ ]\!]\ \} \\
& | & v\ (\ E_{\text{uel}}\ ) \\
& | & \textbf{\$builtIn}\ x\ (\ E_{\text{uel}}\ ) \\
& | & <\ E_{\text{uel}}\ >
\end{array}
$$

Figura A.2: Contextos de evaluación.

A.1. Semántica mediante $\rightarrow^{s/e}$

Definición A.1.1 (Semántica de sentencias)

$$\frac{v \notin \{\textbf{nil}, \textbf{false}\}}{\textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \rightarrow^{s/e} s_1} \qquad \frac{v \in \{\textbf{nil}, \textbf{false}\}}{\textbf{if } v \textbf{ then } s_1 \textbf{ else } s_2 \rightarrow^{s/e} s_2} \qquad \textbf{while } e \textbf{ do } s \rightarrow^{s/e} (\!| \textbf{\$iter } e \textbf{ do } s |\!)_{\text{BREAK}}$$

$$\textbf{\$iter } e \textbf{ do } s \rightarrow^{s/e} \textbf{if } e \textbf{ then } (s \textbf{ ; \$iter } e \textbf{ do } s) \textbf{ else ;} \qquad (\!| E_{lf}[\![\textbf{break}]\!] |\!)_{\text{BREAK}} \rightarrow^{s/e} \textbf{;} \qquad (\!| \textbf{;} |\!)_{\text{BREAK}} \rightarrow^{s/e} \textbf{;}$$

$$\textbf{; } s \rightarrow^{s/e} s \qquad \frac{k \geq 1 \qquad v_{n-k+1} = \textbf{nil}, \ldots, v_n = \textbf{nil}}{evar_1,\ldots,evar_n = v_1,\ldots,v_{n-k} \rightarrow^{s/e} evar_1,\ldots,evar_n = v_1,\ldots,v_{n-k},v_{n-k+1},\ldots,v_n}$$

$$\frac{k \geq 1}{evar_1,\ldots,evar_n = v_1,\ldots,v_n,\ldots,v_{n+k} \rightarrow^{s/e} evar_1,\ldots,evar_n = v_1,\ldots,v_n}$$

$$\frac{n \geq 2}{evar_1,\ldots,evar_n = v_1,\ldots,v_n \rightarrow^{s/e} evar_n = evar_n \textbf{ ; } evar_1,\ldots,evar_{n-1} = v_1,\ldots,v_{n-1}}$$

$$\frac{k \geq 1}{\textbf{local } x_1, \ldots, x_n = v_1, \ldots, v_n, \ldots, v_{n+k} \textbf{ in } s \rightarrow^{s} \textbf{local } x_1, \ldots, x_n = v_1, \ldots, v_n \textbf{ in } s}$$

$$\frac{k \geq 1 \qquad v_{n-k+1} = \textbf{nil}, \ldots, v_n = \textbf{nil}}{\textbf{local } x_1, \ldots, x_n = v_1,\ldots,v_n,\ldots,v_{n-k} \textbf{ in } s \rightarrow^{s} \textbf{local } x_1, \ldots, x_n = v_1,\ldots,v_{n-k},v_{n-k+1},\ldots,v_n \textbf{ in } s}$$

$$\textbf{v:x } (e_1,\ldots,e_n) \rightarrow^{s/e} \textbf{v[“x”] } (v,e_1,\ldots,e_n)$$

$$\textbf{\$statFunCall v:x } (e_1,\ldots,e_n) \rightarrow^{s/e} \textbf{\$statFunCall v[“x”] } (v,e_1,\ldots,e_n)$$

$$\frac{\delta(type, v) \neq \text{“}function\text{”}}{v\,(v_1, \ldots, v_n) \rightarrow^{s/e} (\!| v\,(v_1, \ldots, v_n) |\!)_{\text{WFUNCALL}}}$$

$$\frac{\delta(type, v) \neq \text{“}function\text{”}}{\textbf{\$statFunCall } v\,(v_1, \ldots, v_n) \rightarrow^{s/e} (\!| \textbf{\$statFunCall } v\,(v_1, \ldots, v_n) |\!)_{\text{WFUNCALL}}}$$

$$(\!| E_{lf}[\![\textbf{return } v, \ldots]\!] |\!)_{\text{RETSTAT}} \rightarrow^{s/e} \textbf{;}$$

$$(\!| E_{lf}[\![\textbf{return } v, \ldots]\!] |\!)_{\text{RETEXP}} \rightarrow^{s/e} \textbf{<}v, \ldots\textbf{>} \qquad (\!| \textbf{;} |\!)_{\text{RETSTAT}} \rightarrow^{s/e} \textbf{;} \qquad (\!| \textbf{;} |\!)_{\text{RETEXP}} \rightarrow^{s/e} \textbf{< >}$$

$$(\!| E_{lf}[\![\textbf{return } v, \ldots]\!] |\!)_{\text{BREAK}} \rightarrow^{s/e} \textbf{return } v, \ldots$$

$$\frac{Enp \neq [\![\ ]\!]}{\sigma : \theta : Enp[\![\textbf{\$err } v]\!] \mapsto \sigma : \theta : \textbf{\$err } v}$$

$$(\!| Enp[\![\textbf{\$err } v]\!] |\!)_{\text{PROTMD}} \rightarrow^{s/e} \textbf{<false, } v\textbf{>}$$

$$\frac{v_2 \in functiondef}{(\!| Enp[\![\textbf{\$err } v_1]\!] |\!)^{v_2}_{\text{PROTMD}} \rightarrow^{s/e} v_2(v_1)} \qquad \frac{v_2 \notin functiondef}{(\!| Enp[\![\textbf{\$err } v_1]\!] |\!)^{v_2}_{\text{PROTMD}} \rightarrow^{s/e} (\!| Enp\,[\![\textbf{\$err } \text{“}error\text{”}]\!] |\!)_{\text{PROTMD}}}$$

$$(\!| \textbf{< } v, \ldots \textbf{ >} |\!)_{\text{PROTMD}} \rightarrow^{s/e} \textbf{< true, } v, \ldots \textbf{ >}$$

Definición A.1.2 (Semántica de expresiones)

$$E_t[\![< v_1, v_2, \ldots >]\!] \rightarrow^{s/e} E_t[\![v_1]\!] \qquad E_t[\![< >]\!] \rightarrow^{s/e} E_t[\![\ \textbf{nil}\ ]\!] \qquad \begin{array}{c} E_u[\![< v, \ldots >]\!] \\ \rightarrow^{s/e} \\ E_u\ [\![< v, \ldots >]\!]^{un} \end{array}$$

$$\frac{op \in \{\textbf{+}, \textbf{-}, \textbf{*}, /, \hat{}, \textbf{\%}, \textbf{<}, \leq\} \qquad v_1, v_2 \in number}{v_1\ op\ v_2 \rightarrow^{s/e} \delta(op, v_1, v_2)} \qquad \frac{op \in \{<, \leq, \textbf{..}\} \qquad v_1, v_2 \in string}{v_1\ op\ v_2 \rightarrow^{s/e} \delta(op, v_1, v_2)}$$

$$v \in number \wedge unop = \textbf{-} \quad \vee \quad v \in string \wedge unop = \textbf{\#} \quad \vee$$

$$\frac{op \in \{\textbf{and}, \textbf{or}\}}{v_1\ op\ v_2 \rightarrow^{s/e} \delta(op, v_1, v_2)} \qquad \frac{unop = \textbf{not}}{unop\ v \rightarrow^{s/e} \delta(op, v_1, v_2)} \qquad \frac{\delta(\textbf{==}, v_1, v_2) == \textbf{true}}{v_1\ \textbf{==}\ v_2 \rightarrow^{e} \textbf{true}}$$

$$\frac{\delta(\textbf{==}, v_1, v_2) = \textbf{false} \qquad v_1 \notin tid \vee v_2 \notin tid}{v_1\ \textbf{==}\ v_2 \rightarrow^{s/e} v_1\ \textbf{==}\ v_2} \qquad \frac{\textbf{op} \in \{>, \geq\}}{v_1\ \textbf{op}\ v_2 \rightarrow^{s/e} v_1\ \textbf{op}^{-1}\ v_2}$$

$$\frac{op \in \{\textbf{+}, \textbf{-}, \textbf{*}, /, \hat{}, \textbf{\%}\} \qquad v_1 \notin number \vee v_2 \notin number \qquad i \in \{1,2\},\ v_i = \delta(\textbf{tonumber}, v_i, 10) \qquad v_i \neq \textbf{nil}}{v_1\ op\ v_2 \rightarrow^{s/e} \delta(op, v_1, v_2)}$$

$$\frac{v \notin number \qquad \delta(\textbf{tonumber}, v, 10) \neq \textbf{nil}}{\textbf{-}v \rightarrow^{s/e} \delta(\textbf{tonumber}, v, 10)}$$

$$\frac{v_1 \notin string \vee v_2 \notin string \qquad i \in \{1,2\},\ v_i \in number \cup string \qquad v_i = \delta(\textbf{tostring}, v_i)}{v_1\ \textbf{..}\ v_2 \rightarrow^{s/e} \delta(\textbf{..}, v_1, v_2)}$$

$$\frac{\begin{array}{c} op \in \{\textbf{+}, \textbf{-}, \textbf{*}, /, \hat{}, \textbf{\%}\} \qquad v_1 \notin number \vee v_2 \notin number \\ \delta(\textbf{tonumber}, v_1, 10) = \textbf{nil} \vee \delta(\textbf{tonumber}, v_2, 10) = \textbf{nil} \end{array}}{v_1\ op\ v_2 \rightarrow^{s/e} (\!|\ v_1\ op\ v_2\ |\!)_{\text{ARITHWO}}} \qquad \frac{v \notin number \qquad \delta(tonumber,\ v) = \textbf{nil}}{-\ v \rightarrow^{s/e} (\!|\ -\ v\ |\!)_{\text{NEGWO}}}$$

$$\frac{\neg(v_1, v_2 \in number \cup string)}{v_1\ \textbf{..}\ v_2 \rightarrow^{s/e} (\!|\ v_1\ \textbf{..}\ v_2\ |\!)_{\text{STRCONCATWO}}} \qquad \frac{op \in \{\textbf{<}, \leq\} \qquad v_1 \in t_1 \qquad v_2 \in t_2 \qquad t_1 \neq t_2 \vee t_1 \notin string \cup number}{v_1\ op\ v_2 \rightarrow^{s/e} (\!|\ v_1\ op\ v_2\ |\!)_{\text{ORDCOMPWO}}}$$

$$\frac{v \notin string}{\#\ v \rightarrow^{s/e} (\!|\ \#\ v\ |\!)_{\text{STRLENWO}}} \qquad \frac{\delta(\textbf{==}, tid_1, tid_2) = \textbf{false}}{v_1\ \textbf{==}\ v_2 \rightarrow^{s/e} (\!|\ tid_1\ \textbf{==}\ tid_2\ |\!)_{\text{EQFAIL}}}$$

$$\frac{\begin{array}{c} l \in \{assert, error, load, pcall, rawequal, select, tonumber, type, xpcall, math.abs, \\ math.acos, math.asin, math.atan, math.ceil, math.cos, math.cosh, math.deg, \\ math.exp, math.floor, math.fmod, math.log, math.max, math.modf, math.rad, \\ math.sin, math.sinh, math.sqrt, math.tan, math.tanh, string.len, string.rep, \\ string.reverse, string.sub, table.pack\} \end{array}}{\textbf{\$builtln}\ l\ (v_1, \ldots, v_n) \rightarrow^{s/e} \delta(l,\ v_1, \ldots, v_n)}$$

A.2. Semántica mediante $\rightarrow^{\sigma}$

Definición A.2.1 (Semántica de sentencias)

$$\frac{\sigma' = \sigma[r := v]}{\sigma : r = v \rightarrow^{\sigma} \sigma' : ;} \qquad \frac{\sigma' = (r_1, v_1), ..., (r_n, v_n), \sigma}{\sigma : \textbf{local } x_1, \ldots, x_n = v_1, \ldots, v_n \textbf{ in } s \rightarrow^{\sigma} \sigma' : s[x_1 \backslash r_1, ..., x_n \backslash r_n]}$$

$$\frac{\begin{array}{c} i \leq min(m,n) \Rightarrow v'_i = v_i \\ i > m \Rightarrow v'_i = \textbf{nil} \\ \sigma' = (r_1, v'_1), ..., (r_n, v'_n), \sigma \end{array}}{\sigma : (\textbf{\$statFunCall function } l\ (x_1, ..., x_n)\ s)\ (v_1, ..., v_m) \rightarrow^{\sigma} \sigma' : (\!|\ s\ [x_1 \backslash r_1, ..., x_n \backslash r_n]\ |\!)_{\text{RETSTAT}}}$$

$$\frac{\begin{array}{c} i \leq min(m,n) \Rightarrow v'_i = v_i \\ i > m \Rightarrow v'_i = \textbf{nil} \\ tuple = \langle v_{n+1}, ..., v_m \rangle \\ \sigma' = (r_1, v'_1), ..., (r_n, v'_n), \sigma \end{array}}{\sigma : (\textbf{\$statFunCallfunction } l\ (x_1, ..., x_n, ...)\ s)(v_1, ..., v_m) \rightarrow^{\sigma} \sigma' : (\!|\ s\ [x_1 \backslash r_1, ..., x_n \backslash r_n, ... \backslash tuple]\ |\!)_{\text{RETSTAT}}}$$

Definición A.2.2 (Semántica de expresiones)

$$\sigma : r \rightarrow^{\sigma} \sigma : \sigma(r) \qquad \frac{\begin{array}{c} i \leq min(m,n) \Rightarrow v'_i = v_i \\ i > m \Rightarrow v'_i = \textbf{nil} \\ \sigma' = (r_1, v'_1), ..., (r_n, v'_n), \sigma \end{array}}{\sigma : (\textbf{function } l\ (x_1, ..., x_n)\ s)\ (v_1, ..., v_m) \rightarrow^{\sigma} \sigma' : (\!|\ s\ [x_1 \backslash r_1, ..., x_n \backslash r_n]\ |\!)_{\text{RETEXP}}}$$

$$\frac{\begin{array}{c} i \leq min(m,n) \Rightarrow v'_i = v_i \\ i > m \Rightarrow v'_i = \textbf{nil} \\ tuple = \langle v_{n+1}, ..., v_m \rangle \\ \sigma' = (r_1, v'_1), ..., (r_n, v'_n), \sigma \end{array}}{\sigma : (\textbf{function } l\ (x_1, ..., x_n, ...)\ s)(v_1, ..., v_m) \rightarrow^{\sigma} \sigma' : (\!|\ s\ [x_1 \backslash r_1, ..., x_n \backslash r_n, ... \backslash tuple]\ |\!)_{\text{RETEXP}}}$$

A.3. Semántica mediante $\rightarrow^{\theta}$

Definición A.3.1 (Semántica de sentencias)

$$\frac{\delta(rawget, tid, v_1, \theta_1) \neq \textbf{nil} \qquad \theta_2, tid = \delta(rawset, tid, v_1, v_2, \theta_1)}{\theta_1 : tid\ [v_1] = v_2 \rightarrow^{\theta} \theta_2 : ;} \qquad \frac{s = tid\ [v_1] = v_2 \qquad \delta(rawget, tid, v_1, \theta) = \textbf{nil}}{\theta : s \rightarrow^{\theta} \theta : (\!|\ s\ |\!)_{\text{WRONGKEY}}}$$

$$\frac{s = v_1\ [v_2] = v_3 \qquad \delta(type, v_1) \neq \text{“}table\text{”}}{\theta : s \rightarrow^{\theta} \theta : (\!|\ s\ |\!)_{\text{NONTABLE}}}$$

Definición A.3.2 (Semántica de expresiones)

$$\frac{\forall\, 1 \le i, \mathit{field}_i \in v \vee \mathit{field}_i = \mathbf{[}\ v_1\ \mathbf{]} = v_2 \qquad t = (\mathit{addkeys}(\{\mathit{field}_1,\ ...\}),\ \mathbf{nil}) \qquad \theta_2 = (\mathit{tid},\ t),\ \theta_1}{\theta_1\ :\ \{\mathit{field}_1,\ ...\} \rightarrow^{\theta}\ \theta_2\ :\ \mathit{tid}}$$

$$\frac{v_2 = \delta(\mathit{rawget},\ \mathit{tid},\ v_1,\ \theta) \qquad v_2 \neq \mathbf{nil}}{\theta : \mathit{tid}\ [\ v_1\ ]\ \rightarrow^{\theta}\ \theta : v_2} \qquad \frac{e = \mathit{tid}\ \mathbf{[}\ v\ \mathbf{]} \qquad \delta(\mathit{rawget},\ \mathit{tid},\ v,\ \theta) = \mathbf{nil}}{\theta : e\ \rightarrow^{"}\ \theta : (\!|\ e\ |\!)_{\text{WRONGKEY}}}$$

$$\frac{e = v_1\ \mathbf{[}\ v_2\ \mathbf{]} \qquad \delta(\mathit{type}, v_1) \neq \text{“}\mathit{table}\text{”}}{\theta : e\ \rightarrow^{\theta}\ \theta : (\!|\ e\ |\!)_{\text{NONTABLE}}}$$

$$\frac{\mathit{svc} \in \{\mathit{ipairs}, \mathit{next}, \mathit{pairs}, \mathit{loadfile}, \mathit{getmetatable}, \mathit{tostring}, \mathit{rawget}, \mathit{rawlen}, \mathit{require}, \mathit{table.concat}, \mathit{table.unpack}, \mathit{string.dump}\}}{\theta : \mathbf{\$builtIn}\ \mathit{svc}\ (v_1, ..., v_n) \rightarrow^{\theta} \theta : \delta(\mathit{svc}, v_1, ..., v_n, \theta)}$$

$$\frac{\mathit{svc} \in \{\mathit{rawset}, \mathit{setmetatable}\} \qquad (\theta_2, v) = \delta(\mathit{svc}, v_1, ..., v_n, \theta_1)}{\theta_1 : \mathbf{\$builtIn}\ \mathit{svc}\ (v_1, ..., v_n) \rightarrow^{\theta}\ \theta_2 : v}$$

A.4. Semántica mediante $\rightarrow^{meta}$

Definición A.4.1 (Semántica de sentencias)

$$\frac{v_{n+1} = \mathit{indexmetatable}(v_1,\ \text{“}\_\_\mathit{call}\text{”},\ \theta) \qquad v_{n+1} \notin \{\mathbf{nil},\ \mathbf{false}\}}{\theta : (\!|\ \mathbf{\$statFunCall}\ v_1\ (v_2, ..., v_n)\ |\!)_{\text{WFUNCALL}} \rightarrow^{meta}\ \theta : \mathbf{\$statFunCall}\ v_{n+1}(v_1, v_2, ..., v_n)}$$

$$\frac{\mathit{indexmetatable}(v_1,\ \text{“}\_\_\mathit{call}\text{”},\ \theta) \in \{\mathbf{nil}, \mathbf{false}\} \qquad \mathit{type} = \delta(\mathit{type}, v_1, \theta)}{\theta : (\!|\ \mathbf{\$statFunCall}\ v_1\ (v_2, ..., v_n)\ |\!)_{\text{WFUNCALL}} \rightarrow^{meta}\ \theta : \mathbf{\$builtIn}\ \mathit{error}\ (\text{“}\mathit{error}\text{”})}$$

$$\frac{v_4 = \mathit{indexmeta}(v_1, \text{“}\_\_\text{newindex}\text{”}, \theta) \qquad \delta(\mathit{type}, v_4) = \text{“}\mathit{function}\text{”}}{\theta : (\!|\ v_1\ \mathbf{[}v_2\mathbf{]} = v_3\ |\!)_{\mathit{label}} \rightarrow^{meta}\ \theta : \mathbf{(\$statFcall}\ v_4\ \mathbf{(}v_1, v_2, v_3\mathbf{))}}$$

$$\frac{v_4 = \mathit{indexmeta}(v_1, \text{“}\_\_\text{newindex}\text{”}, \theta) \qquad v_4 \neq \mathbf{nil} \qquad \delta(\mathit{type}, v_4) \neq \text{“}\mathit{function}\text{”}}{\theta : (\!|\ v_1\ \mathbf{[}v_2\mathbf{]} = v_3\ |\!)_{\mathit{label}} \rightarrow^{meta}\ \theta : v_4\ \mathbf{[}v_2\mathbf{]} = v_3}$$

$$\frac{\mathit{indexmeta}(\mathit{tid}, \text{“}\_\_\text{newindex}\text{”}, \theta_1) = \mathbf{nil} \qquad (\theta_2, \mathit{tid}) = \delta(\mathit{rawset}, \mathit{tid}, v_1, v_2, \theta_1)}{\theta_1 : (\!|\ \mathit{tid}\ \mathbf{[}v_1\mathbf{]} = v_2\ |\!)_{\text{WRONGKEY}} \rightarrow^{meta}\ \theta_2 : ;}$$

$$\frac{\mathit{indexmeta}(\mathit{tid}, \text{“}\_\_\text{newindex}\text{”}, \theta_1) = \mathbf{nil} \qquad (\theta_2, \mathbf{\$err}...) = \delta(\mathit{rawset}, \mathit{tid}, v_1, v_2, \theta_1)}{\theta_1 : (\!|\ \mathit{tid}\ \mathbf{[}v_1\mathbf{]} = v_2\ |\!)_{\text{WRONGKEY}} \rightarrow^{meta}\ \theta_2 : \mathbf{\$err}...}$$

$$\frac{\mathit{indexmeta}(v_1, \text{“}\_\_\text{newindex}\text{”}, \theta) = \mathbf{nil} \qquad t = \delta(\mathit{type}, v_1) \qquad \mathit{msg} = \mathit{errmessage}(\text{NONTABLE}, t)}{\theta : (\!|\ v_1\ \mathbf{[}v_2\mathbf{]} = v_3\ |\!)_{\text{NONTABLE}} \rightarrow^{meta}\ \theta : \delta(\mathit{error},\ \mathit{msg})}$$

$$
\begin{array}{lll}
getbinhandler(v_1, v_2, v_3, \theta) \doteq \begin{cases} indexmeta(v_1, v_3, \theta) & \textbf{if} & indexmeta(v_1, v_3, \theta) \notin \{\textbf{nil}, \textbf{false}\} \\ indexmeta(v_2, v_3, \theta) & \textbf{if} & indexmeta(v_1, v_3, \theta) \in \{\textbf{nil}, \textbf{false}\} \\ & & \wedge \\ & & indexmeta(v_2, v_3, \theta) \notin \{\textbf{nil}, \textbf{false}\} \\ \textbf{nil} & \textbf{c.c.} & \end{cases} \\
getunaryhandler(v_1, v_2, \theta) \doteq \begin{cases} indexmeta(v_1, v_2, \theta) & \textbf{if} & indexmeta(v_1, v_2, \theta) \notin \{\textbf{nil}, \textbf{false}\} \\ \textbf{nil} & \textbf{c.c.} & \end{cases} \\
getequalhandler(v_1, v_2, \theta) \doteq \begin{cases} indexmeta(v_1, \text{“}\_\_eq\text{”}, \theta) & \textbf{if} & \{\delta(type, v_1, \theta), \delta(type, v_2', \theta)\} = \{\text{“}table\text{”}\} \\ & & \wedge \\ & & indexmeta(v_1, \text{“}\_\_eq\text{”}, \theta) \\ & & = \\ & & indexmeta(v_2, \text{“}\_\_eq\text{”}, \theta) \\ \textbf{nil} & \textbf{c.c.} & \end{cases}
\end{array}
$$

Figura A.3: Funciones auxiliares para meta-tablas sobre expresiones.

Para la definir la semántica del mecanismo de meta-tablas sobre expresiones, precisaremos de las funciones auxiliares de la Figura A.3, definidas considerando lo presentado en `www.lua.org/manual/5.2/manual.html#2.4`.

Definición A.4.2 (Semántica de expresiones)

$$\frac{v_{n+1} = indexmetatable(v_1, \text{“__call”}, \theta) \qquad v_{n+1} \notin \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ (v_2, ..., v_n) |\!)_{\text{WFUNCALL}} \rightarrow^{meta} \theta : v_{n+1}(v_1, v_2, ..., v_n)}$$

$$\frac{indexmetatable(v_1, \text{“__call”}, \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad type = \delta(type, v_1, \theta)}{\theta : (\!| v_1\ (v_2, ..., v_n) |\!)_{\text{WFUNCALL}} \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ (\text{“error”})}$$

$$\frac{v_3 = indexmetatable(v_1, \text{“__index”}, \theta) \qquad \delta(type, v_3, \theta) = \text{”function”}}{\theta : (\!| v_1\ \textbf{[}v_2\textbf{]} |\!)_{label} \rightarrow^{meta} \theta : v_3\ \textbf{(}v_1, v_2\textbf{)}}$$

$$\frac{v_3 = indexmetatable(v_1, \text{“__index”}, \theta) \qquad v_3 \neq \textbf{nil} \quad \delta(type, v_3, \theta) \neq \text{“function”}}{\theta : (\!| v_1\ \textbf{[}v_2\textbf{]} |\!)_{label} \rightarrow^{meta} \theta : v_3\ \textbf{[}v_2\textbf{]}}$$

$$\frac{indexmetatable(tid, \text{“__index”}, \theta) = \textbf{nil}}{\theta : (\!| tid\ \textbf{[}v\textbf{]} |\!)_{\text{WRONGKEY}} \rightarrow^{meta} \theta : \textbf{nil}} \qquad \frac{indexmetatable(v_1, \text{“__index”}, \theta) = \textbf{nil}}{\theta : (\!| v_1\ \textbf{[}v_2\textbf{]} |\!)_{\text{NONTABLE}} \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ \textbf{(}\text{“error”}\ \textbf{)}}$$

$$\frac{v_3 = getbinhandler(v_1, v_2, binopeventkey(op), \theta) \qquad v_3 \notin \{\textbf{nil}, \textbf{false}\} \qquad op \in \{\textbf{+}, \textbf{-}, \textbf{*}, \textbf{/}, \textbf{\^{}}, \textbf{\%}, \textbf{..}\}}{\theta : (\!| v_1\ op\ v_2 |\!)_{label} \rightarrow^{meta} \theta : v_3\ \textbf{(}v_1, v_2\textbf{)}}$$

$$\frac{getbinhandler(v_1, v_2, binopeventkey(op), \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad op \in \{\textbf{+}, \textbf{-}, \textbf{*}, \textbf{/}, \textbf{\^{}}, \textbf{\%}, \textbf{..}\}}{\theta : (\!| v_1\ op\ v_2 |\!)_{label} \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ (\text{“error”})}$$

$$\frac{v_2 = getunaryhandler(v_1, \text{“__unm”}, \theta) \qquad v_2 \notin \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| \textbf{-}v_1 |\!) \rightarrow^{meta} \theta : v_2\ (v_1)}$$

$$\frac{getunaryhandler(v_1, \text{“__unm”}, \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad v_2 = \delta(type, v, \theta)}{\theta : (\!| \textbf{-}v_1 |\!) \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ (\text{“error”})}$$

$$\frac{v_2 = getunaryhandler(v_1, \text{“__len”}, \theta) \qquad v_2 \notin \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| \textbf{\#}\ v_1 |\!)_{\text{STRLENWO}} \rightarrow^{meta} \theta\ ;\ v_2\ (v_1)}$$

$$\frac{getunaryhandler(v, \text{“__len”}, \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad \delta(type, v, \theta) = \text{”table”}}{\theta : (\!| \textbf{\#}\ v |\!)_{\text{STRLENWO}} \rightarrow^{meta} \theta : \delta\ (\#, \pi_1(\theta(v)))}$$

$$\frac{getunaryhandler(v, \text{“__len”}, \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad \delta(type, v, \theta) = \text{”table”}}{\theta : (\!| \textbf{\#}\ v |\!)_{\text{STRLENWO}} \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ (\text{“error”})}$$

$$\frac{v_3 = getequalhandler(v_1, v_2, \theta) \qquad v_3 \notin \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ \textbf{==}\ v_2 |\!)_{\text{EQFAIL}} \rightarrow^{meta} \theta : \textbf{not not}\ v_3\ (v_1, v_2)} \qquad \frac{getequalhandler(v_1, v_2, \theta) \in \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ \textbf{==}\ v_2 |\!)_{\text{EQFAIL}} \rightarrow^{meta} \theta : \textbf{false}}$$

$$\frac{op \in \{<, \leq\} \qquad v_3 = getbinhandler(v_1, v_2, binopeventkey(op), \theta) \qquad v_3 \notin \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ op\ v_2 |\!)_{\text{ORDCOMPWO}} \rightarrow^{meta} \theta : v_3\ (v_1, v_2)}$$

$$\frac{getbinhandler(v_1, v_2, \text{“__lt”}, \theta) \in \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ \textbf{<}\ v_2 |\!)_{\text{ORDCOMPWO}} \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ (\text{“error”})}$$

$$\frac{getbinhandler(v_1, v_2, \text{“__le”}, \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad v_3 = getbinhandler(v_1, v_2, \text{“__lt”}, \theta) \qquad v_3 \notin \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ \leq\ v_2 |\!)_{\text{ORDCOMPWO}} \rightarrow^{meta} \theta : \textbf{not}\ v_3\ (v_2, v_1)}$$

$$\frac{getbinhandler(v_1, v_2, \text{“__le”}, \theta) \in \{\textbf{nil}, \textbf{false}\} \qquad getbinhandler(v_1, v_2, \text{“__lt”}, \theta) \in \{\textbf{nil}, \textbf{false}\}}{\theta : (\!| v_1\ \leq\ v_2 |\!)_{\text{ORDCOMPWRONGOPS}} \rightarrow^{meta} \theta : \textbf{\$builtIn}\ error\ (\text{“error”})}$$

A.5. Semántica mediante ↦

Definición A.5.1 (Semántica de sentencias)

$$\frac{t \rightarrow^{s/e} t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma : \theta : E[\![t']\!]} \qquad \frac{\sigma : t \rightarrow^{\sigma} \sigma' : t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma' : \theta : E[\![t']\!]} \qquad \frac{\theta : t \rightarrow^{\theta} \theta' : t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma : \theta' : E[\![t']\!]}$$

$$\frac{\theta : t \rightarrow^{meta} \theta' : t'}{\sigma : \theta : E[\![t]\!] \mapsto \sigma : \theta' : E[\![t']\!]} \qquad \frac{Enp \neq [\![\ ]\!]}{\sigma : \theta : Enp[\![\textbf{\$err}\ v]\!] \mapsto \sigma : \theta : \textbf{\$err}\ v}$$

A.6. Meta-función δ

A continuación, proveeremos de las ecuaciones que sirven de especificación, mediante la función δ, de los operadores primitivos del lenguaje y las funciones básicas de la librería estándar incluidas en el modelo. Incluiremos también, por completitud, aquellas especificaciones que resultan triviales. La mecanización del modelo incluye servicios de las librerías math, string y table, implementados sobre servicios de la librería Racket, por lo que no los incluiremos en este apartado.

A.6.1. Operadores primitivos

Mencionaremos los recursos que utilizaremos para distinguir entre elementos del lenguaje objeto y elementos del meta-lenguaje utilizado para la especificación. Para referirnos a literales numéricos (del lenguaje objeto) utilizaremos la variable n (posiblemente con subíndices), mientras que con $\hat{n}$ denotaremos al número que se corresponde con el literal n; s será una variable cuantificada sobre los literales string. A su vez, para el caso de un símbolo **op** del lenguaje objeto que represente una operación aritmética o relación de orden, denotaremos con el mismo símbolo a la correspondiente función o relación en el meta-lenguaje, apelando al contexto en el que ocurre cada símbolo para desambiguar.

Las primeras ecuaciones especifican de manera usual el comportamiento esperado para los operadores aritméticos, lógicos y relaciones de orden de Lua. Notar que las constantes **true** y **false** que se deben retornar son los literales booleanos de Lua; de allí la forma en la que están definidas estas ecuaciones. También notar cómo especificamos operaciones aritméticas y sobre strings exclusivamente en términos de las correspondientes operaciones en el meta-lenguaje, asumiendo ya resueltas cuestiones de coerción y meta-programación, las cuales hacen parte de la semántica de las expresiones en cuestión, en Lua.

Definición A.6.1 (Interpretación de operadores primitivos: operadores aritméticos y lógicos)
$\delta(\textbf{op}, n_1, n_2) = \hat{n_1}\ op\ \hat{n_2}, \textbf{op} \in \{\textbf{+}, \textbf{-}, \textbf{*}, \textbf{/}, \textbf{\^{}}, \textbf{\%}\}$

$$\delta(\textbf{and}, v, e) = \begin{cases} v & \textbf{si } v \in \{\textbf{false}, \textbf{nil}\} \\ (e) & \textbf{c.c.} \end{cases}$$

$$\delta(\textbf{or}, v, e) = \begin{cases} v & \textbf{si } v \notin \{\textbf{false}, \textbf{nil}\} \\ (e) & \textbf{c.c.} \end{cases}$$

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

Definición A.6.2 (Interpretación de operadores primitivos: relaciones de orden)

$$\delta(\textbf{==}, v_1, v_2) = \begin{cases} \textbf{true} & \textbf{si } v_1 = v_2 \\ \textbf{false} & \textbf{c.c.} \end{cases}$$

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

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

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

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

A.6.2. Funciones básicas de librería

Definición A.6.3 (Desde assert a getmetatable)

$$\delta(\textit{assert}, v_1, v_2) = \begin{cases} < v_1, v_2 > & \textbf{si} \quad v_1 \neq \textbf{nil} \\ & \qquad \wedge \\ & \quad v_1 \neq \textbf{false} \\ \delta(\textit{error}, v_3) & \textbf{c.c.} \quad \textit{donde } v_3 = \begin{cases} v_2 & \textbf{si} \quad v_2 \neq \textbf{nil} \\ \text{“}assertion\ failed!\text{”} & \textbf{c.c.} \end{cases} \end{cases}$$

$\delta(\textit{collectgarbage}, v_1, v_2, s, \sigma, \theta) = gc_{fin\_weak}(s, \sigma, \theta)$ *(ver §5.1.3)*

$\delta(\textit{error}, v) = \textbf{\$err } v$

$$\delta(\textit{getmetatable}, \textit{tid}, \theta) = \begin{cases} protmeta(\textit{tid}, \theta) & \textbf{si} \quad prot?(\textit{tid}, \theta) \\ & \qquad\qquad (ver\ §4.2.5) \\ \theta(\textit{tid})(2) & \textbf{c.c.} \end{cases}$$

$$\delta(\textit{getmetatable}, v, \theta) = \begin{cases} \textbf{nil} & \textbf{si} \quad \textit{tid}_{\delta(\textit{type},v)} \notin dom(\theta) \\ \textit{tid}_{\delta(\textit{type},v)} & \textbf{si} \quad \begin{cases} \textit{tid}_{\delta(\textit{type},v)} \in dom(\theta) \\ \neg prot?(\textit{tid}_{\delta(\textit{type},v)}, \theta) \end{cases} \\ protmeta(\textit{tid}_{\delta(\textit{type},v)}, \theta) & \textbf{c.c.} \end{cases}$$

Definición A.6.4 (Desde ipairs a next)

$$\delta(\mathit{ipairs}, v, \theta) = \left| \begin{array}{l} (\textbf{function}\ \$\mathit{IpairsCustomIter}\ (\ ) \\ \quad \textbf{local}\ v1,\ v2,\ v3 = \mathit{any}\ (v) \\ \quad \textbf{in} \\ \quad\quad \textbf{return} < v1,\ v2,\ v3 >)\ (\ ) \end{array} \right. , \quad \textbf{donde} \left\{ \begin{array}{l} \delta(\mathit{type}, v) = \text{“}table\text{”} \\ \mathit{any} = indexmetatable(v, \text{“}\_\_ipairs\text{”}, \theta) \\ \mathit{any} \neq \textbf{nil} \end{array} \right.$$

$$\delta(\mathit{ipairs}, v, \theta) = \left| \begin{array}{l} < \textbf{function}\ \$\mathit{iPairsIter}\ (t,\ var) \\ \quad \textbf{local}\ result,\ ttype = \textbf{nil},\ \$\mathit{builtIn}\ type\ (t) \\ \quad \textbf{in} \\ \quad\quad \textbf{if}\ ttype == \text{“}table\text{"}\ \textbf{then}\ ; \\ \quad\quad \textbf{else} \\ \quad\quad\quad \$\mathit{builtIn}\ error\ (\text{“}bad\ argument\ \#1\ (table\ expected,\ got\text{"}\ ..\ ttype\ ..\ \text{") ")} \\ \quad\quad var = var + 1 \\ \quad\quad result = \$\mathit{builtIn}\ rawget\ (t,\ var) \\ \quad\quad \textbf{if}\ result == \textbf{nil}\ \textbf{then return} < \textbf{nil} > \\ \quad\quad \textbf{else return} < var,\ result >, \\ \quad v, \\ \quad 0 > \end{array} \right.$$

$$\textbf{donde} \left\{ \begin{array}{l} \delta(\mathit{type}, v, \theta) = \text{“}table\text{”} \\ indexmetatable(v, \text{“}\_\_ipairs\text{”}, \theta) = \textbf{nil} \end{array} \right.$$

$$\delta(\mathit{ipairs}, v, \theta) = \delta(\mathit{error}, \text{“}bad\ argument\ \#1\ (table\ expected,\ got\ \text{”}\ ..\ \delta(\mathit{type}, v_1)\ ..\ \text{”})\text{”}) ,$$
$$\textbf{si}\ \ \delta(\mathit{type}, v) \neq \text{“}table\text{”}$$

$\delta(\mathit{load}) \in v^4 \rightarrow (\mathit{functiondef} \cup \textbf{error})$ *(ver mecanización)*

$$\delta(\mathit{next}, v, v_i, \theta) = \left\{ \begin{array}{lll} \textbf{<}v_1, v_2\textbf{>} & \textbf{si} & \theta(v) = \textbf{<\{ [}v_1\textbf{] =} v_2\textbf{,...\}, ...>} \\ & & v_i = \textbf{nil} \\ \textbf{<}v_{i+2}\textbf{,}\ v_{i+3}\textbf{>} & \textbf{si} & \theta(v) = \textbf{<\{...,\ [}v'_i\textbf{] =}\ v_{i+1}\textbf{,\ [}v_{i+2}\textbf{] =}\ v_{i+3}\textbf{,\ ...\},} \\ & & \quad \textbf{...>} \\ & & \delta(==, v_i, v'_i) = \textbf{true} \\ \textbf{<nil>} & \textbf{si} & \theta(v) = \textbf{<\{...,\ [}v'_i\textbf{] =}\ v_{i+1}\textbf{\}, ...>} \\ & & \delta(==, v_i, v'_i) = \textbf{true} \\ \textbf{<nil>} & \textbf{si} & \theta(v) = \textbf{<\{ \}, ...>} \\ \delta(\mathit{error}, \text{“}invalid\ key\ to\ 'next'\text{”}) & \textbf{c.c.} & \end{array} \right.$$

si $\delta(\mathit{type}, v) = \text{“}table\text{”}$, *para un orden entre claves de tablas dado (no forma parte de la especificación)*

$$\delta(\mathit{next}, v_1, v_2, \theta) = \delta(\mathit{error}, \text{”}bad\ argument\ \#1\ (table\ expected,\ got\ \text{”}\ ..\ \delta(\mathit{type}, v_1)\ ..\ \text{“})\text{”}) ,$$
$$\textbf{si}\ \ \delta(\mathit{type}, v_1) \neq \text{“}table\text{”}$$

Definición A.6.5 (Desde pairs a rawset)

$$\delta(pairs, v, \theta) = \left| \begin{array}{l} (\textbf{function}\ \$pairsCustomIter\ () \\ \quad \textbf{local}\ v1,\ v2,\ v3 = any\ (v) \\ \quad \textbf{in} \\ \quad\quad \textbf{return} < v1,\ v2,\ v3 >)\ (\ ) \end{array} \right. , donde \left\{ \begin{array}{l} \delta(type, v) = \text{“}table\text{”} \\ any = indexmetatable(v, \text{“}\_\_pairs\text{”}, \theta) \\ any \neq \textbf{nil} \end{array} \right.$$

$$\delta(pairs, v, \theta) = \left| \begin{array}{l} < \textbf{function}\ \$next\ (table,\ index) \\ \quad \textbf{return}\ \$builtIn\ next\ (table,\ index), \\ \quad v, \\ \quad \textbf{nil}\ > \end{array} \right. , donde \left\{ \begin{array}{l} \delta(type, v) = \text{“}table\text{”} \\ indexmetatable(v, \text{“}\_\_pairs\text{”}) = \textbf{nil} \end{array} \right.$$

$$\delta(pairs, v, \theta) = \delta(error, \text{“}bad\ argument\ \#1\ (table\ expected,\ got\ \text{”} .. \delta(type, v_1) .. \text{“})\text{”}) ,$$
$$\textbf{si}\ \ \delta(type, v) \neq \text{“}table\text{”}$$

$$\delta(pcall, v_1, v_2, ...) = (\!|\ v_1(v_2, ...)\ |\!)_{\text{PROTMD}}$$

$$\delta(rawequal, v_1, v_2) = \delta(==, v_1, v_2)$$

$$\delta(rawget, tid, v_i, \theta) = \left\{ \begin{array}{ll} v_j & \textbf{si}\ \theta(tid) = (\{..., [v'_i] = v_j, ...\}, ...) \\ & \textbf{y}\ \delta(==, v_i, v'_i) = \textbf{true} \\ \textbf{nil} & \textbf{c.c.} \end{array} \right.$$
$$\textbf{si}\ \ \delta(type, tid) = \text{“}\textbf{table}\text{”}$$

$$\delta(rawget, v_1, v_2, \theta) = \delta(error, ...)\ , \textbf{si}\ \delta(type, v_1) \neq \text{“}\textbf{table}\text{”}$$

$$\delta(rawlen) \in v \times \theta \rightarrow (number \cup \textbf{error})\ (ver\ mecanización)$$

$$\delta(rawset, tid, v_1, v_2, \theta_1) = (\theta_2, tid),$$
$$donde \left\{ \begin{array}{l} v_1 \notin \{\textbf{nil, nan}\} \\ \theta_1(tid) = (\{..., field_{i-1}, [v'_1] = v_3, field_{i+1}, ...\}, ...) \\ \delta(==, v_1, v'_1) = \textbf{true} \\ \theta_2 = \left\{ \begin{array}{lll} \theta_1[tid := (\{..., field_{i-1}, field_{i+1}, ...\}, ...)] & \textbf{si} & v_2 = \textbf{nil} \\ \theta_1[tid := (\{..., field_{i-1}, [v'_1] = v_2, field_{i+1}, ...\}, ...)] & \textbf{si} & v_2 \neq \textbf{nil} \end{array} \right. \end{array} \right.$$

$$\delta(rawset, tid, v, v', \theta_1) = (\theta_2, tid),$$
$$donde \left\{ \begin{array}{l} v \notin \{\textbf{nil, nan}\} \\ \theta_1(tid) = (\{[v_1] = v'_1, ..., [v_n] = v'_n\}, ...) \\ \forall\ k\ \in\ \{v_1, ..., v_n\}, \delta(==, v, k) = \textbf{false} \\ \theta_2 = \left\{ \begin{array}{lll} \theta_1 & \textbf{si} & v' = \textbf{nil} \\ \theta_1[tid := (\{[v] = v', [v_1] = v'_1, [v_n] = v'_n\}, ...)] & \textbf{si} & v' \neq \textbf{nil} \end{array} \right. \end{array} \right.$$

$$\delta(rawset, v_1, v_2, v_3, \theta) = (\theta, \delta(error, ...))\ ,\ si\ \delta(type, v_1) \neq \text{“}table\text{”} \vee v_2 \in \{\textbf{nil,nan}\}$$

Definición A.6.6 (Desde select a xpcall)

$$\delta(\mathit{select}, v_1, v_2, ..., v_n) = \begin{cases} < v_{\hat{v}_1+1}, ..., v_n > & \mathit{if} \quad 1 \leq \hat{v}_1 \leq n-1 \\ < > & \mathit{if} \quad n-1 < \hat{v}_1 \\ < v_{n-1+|\hat{v}_1|}, ..., v_n > & \mathit{if} \quad -(n-1) \leq \hat{v}_1 \leq -1 \\ < \ulcorner n-1 \urcorner > & \mathit{if} \quad v_1 = "\#" \end{cases}$$

$$\delta(\mathit{setmetatable}, v_1, v_2, \theta) = (\delta(error, "..."), \theta), \mathit{if} \begin{cases} \delta(\mathit{type}, v_2) \notin \{"table", "nil"\} \\ \vee \\ prot?(\mathit{tid}, \theta_1) \\ \vee \\ \delta(\mathit{type}, v_1) \neq "table" \end{cases}$$

$$\delta(\mathit{setmetatable}, \mathit{tid}, v, \theta_1) = (\mathit{tid}, \theta_2), \mathit{if} \begin{cases} \delta(\mathit{type}, v) \in \{"table", "nil"\} \\ \neg prot?(\mathit{tid}, \theta_1) \\ \theta_2 = \theta_1[\mathit{tid} := (\theta_1(\mathit{tid})(1), v)] \end{cases}$$

$\delta(\mathit{tonumber}) \in v^2 \rightarrow (\mathit{number} \cup \{\mathbf{nil}\} \cup \mathbf{error})$ *(ver mecanización)*

$\delta(\mathit{tostring}) \in v \times \theta \rightarrow \mathbf{e}$ *(ver mecanización)*

$$\delta(\mathit{type}, v) = \begin{cases} "nil" & \mathit{si} \quad v = \mathbf{nil} \\ "number" & \mathit{si} \quad v \in \mathit{number} \\ "string" & \mathit{si} \quad v \in \mathit{string} \\ "bool" & \mathit{si} \quad v \in \{\mathbf{false}, \mathbf{true}\} \\ "table" & \mathit{si} \quad v \in \mathit{tid} \\ "function" & \mathit{c.c.} \end{cases}$$

$$\delta(\mathit{xpcall}, v_1, v_2, v_3, ...) = (\!| \, v_1(v_3, ...) \, |\!)^{v_2}_{\text{PROTMD}}$$

B

Buena formación de configuraciones

Capturaremos la noción de configuraciones bien formadas mediante un sistema formal o de inferencia. Definiremos buena formación de configuraciones en términos de sistemas formales que capturaran, por separado, la noción de términos y almacenamientos bien formados.

B.1. Buena formación de términos

Definición B.1.1 (Buena formación de sentencias)

$$C \vdash_{wft} \sigma : \theta : \mathbf{;} \qquad \frac{\vdash_{mtch} C \approx C'[\![\ \mathbf{while}\ e\ \mathbf{do}\ C''\ \mathbf{end}\ ]\!] \vee \vdash_{mtch} C \approx C'[\![\ (\!|\ C''\ |\!)_{\text{BREAK}}\ ]\!]}{C \vdash_{wft} \sigma : \theta : \mathbf{break}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e\ \ldots}{C \vdash_{wft} \sigma : \theta : \mathbf{return}\ e\ \ldots} \qquad \frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2\ \ldots}{C \vdash_{wft} \sigma : \theta : \mathbf{\$statFunCall}\ e_1\ (e_2, \ldots)}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2\ \ldots}{C \vdash_{wft} \sigma : \theta : \mathbf{\$statFunCall}\ e_1\ \mathbf{:}\ x\ (e_2, \ldots)} \qquad \frac{C \vdash_{wft} \sigma : \theta : var\ \ldots \qquad C \vdash_{wft} \sigma : \theta : e\ \ldots}{C \vdash_{wft} \sigma : \theta : var, \ldots = e, \ldots}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e \qquad C \vdash_{wft} \sigma : \theta : s_1 \qquad C \vdash_{wft} \sigma : \theta : s_2}{C \vdash_{wft} \sigma : \theta : \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e \qquad C[\![\ \mathbf{while}\ e\ \mathbf{do}\ [\![\ ]\!]\ ]\!] \vdash_{wft} \sigma : \theta : s}{C \vdash_{wft} \sigma : \theta : \mathbf{while}\ e\ \mathbf{do}\ s}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e\ \ldots \qquad C[\![\ \mathbf{local}\ x, \ldots = e, \ldots\ \mathbf{in}\ [\![\ ]\!]\ ]\!] \vdash_{wft} \sigma : \theta : s}{C \vdash_{wft} \sigma : \theta : \mathbf{local}\ x, \ldots = e, \ldots\ \mathbf{in}\ s}$$

$$\frac{C \vdash_{wft} \sigma : \theta : s_1 \qquad C \vdash_{wft} \sigma : \theta : s_2\ \ldots}{C \vdash_{wft} \sigma : \theta : s_1\ s_2\ \ldots}$$

Definición B.1.2 (Buena formación de construcciones de tiempo de ejecución: sentencias)

$$\frac{\begin{array}{c} C[\![\ (\!|\ [\![\ ]\!]\ |\!)_{\text{BREAK}}\ ]\!] \vdash \sigma : \theta : s \\ \vdash_{mtch} s \approx \textbf{\$iter}\ e\ \textbf{do}\ s \quad \vee \\ \vdash_{mtch} s \approx \textbf{if}\ e\ \textbf{do}\ s'\ \textbf{; \$iter}\ e\ \textbf{do}\ s'\ \textbf{else} \quad \vee \\ \vdash_{mtch} s \approx s'\ \textbf{; \$iter}\ e\ \textbf{do}\ s' \quad \vee \\ \vdash_{mtch} s \approx \textbf{;} \end{array}}{C \vdash \sigma : \theta : (\!|\ s\ |\!)_{\text{BREAK}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e \qquad C[\![\ \textbf{\$iter}\ e\ \textbf{do}\ [\![\ ]\!]\ ]\!] \vdash_{wft} \sigma : \theta : s \qquad \vdash_{mtch} C \approx C'[\![\ (\!|\ C''\ |\!)_{\text{BREAK}}\ ]\!]}{C \vdash_{wft} \sigma : \theta : \textbf{\$iter}\ e\ \textbf{do}\ s}$$

$$\frac{C \vdash_{wft} \sigma : \theta : tid \qquad C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad v_1 \notin dom(\theta(tid)(1))}{C \vdash_{wft} \sigma : \theta : (\!|\ tid[v_1] = v_2\ |\!)_{\text{WRONGKEY}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad C \vdash_{wft} \sigma : \theta : v_3 \qquad v_1 \neq tid}{C \vdash_{wft} \sigma : \theta : (\!|\ v_1[v_2] = v_3\ |\!)_{\text{NONTABLE}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \ldots \qquad v_1 \notin functiondef}{C \vdash_{wft} \sigma : \theta : (\!|\ \textbf{\$statFunCall}\ v_1\ (v_2, \ldots)\ |\!)_{\text{WRONGFUNCALL}}} \qquad \frac{C[\![\ (\!|\ [\![\ ]\!]\ |\!)_{\text{RETSTAT}}\ ]\!] \vdash_{wft} \sigma : \theta : s}{C \vdash_{wft} \sigma : \theta : (\!|\ s\ |\!)_{\text{RETSTAT}}}$$

Definición B.1.3 (Buena formación de expresiones)

$$C \vdash_{wft} \sigma : \theta : \textbf{nil} \qquad C \vdash_{wft} \sigma : \theta : \textbf{true} \qquad C \vdash_{wft} \sigma : \theta : \textbf{false} \qquad C \vdash_{wft} \sigma : \theta : number$$

$$C \vdash_{wft} \sigma : \theta : string \qquad \frac{tid \in dom(\theta)}{C \vdash_{wft} \sigma : \theta : tid} \qquad \frac{C[\![\ \textbf{function}\ x_1\ (x_2, \ldots)\ [\![\ ]\!]\ ]\!] \vdash_{wft} \sigma : \theta : s}{C \vdash_{wft} \sigma : \theta : \textbf{function}\ x_1\ (x_2, \ldots)\ s}$$

$$\frac{\vdash_{mtch} C \approx C'[\![\ \textbf{function}\ x_1\ (x_2, \ldots, \textbf{...})\ C''\ ]\!]}{C \vdash_{wft} \sigma : \theta : \textbf{...}}$$

$$\frac{\vdash_{mtch} C \approx C'[\![\ \textbf{function}\ x_1\ (x_2, \ldots, x, x_3, \ldots)\ C''\ ]\!] \quad \vee \quad \vdash_{mtch} C \approx C'[\![\ \textbf{local}\ x_1, \ldots, x, x_2, \ldots \textbf{ = } e, \ldots \textbf{ in } C''\ ]\!]}{C \vdash_{wft} \sigma : \theta : x}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2}{C \vdash_{wft} \sigma : \theta : e_1[e_2]} \qquad \frac{C \vdash_{wft} \sigma : \theta : e, \ldots}{C \vdash_{wft} \sigma : \theta : \textbf{\$builtIn}\ serv\ (e, \ldots)} \qquad \frac{C \vdash_{wft} \sigma : \theta : e, \ldots}{C \vdash_{wft} \sigma : \theta : \textbf{(}\ e\ \textbf{)}}$$

$$\frac{C \vdash_{wft\_field} \sigma : \theta : field, \ldots}{C \vdash_{wft} \sigma : \theta : \textbf{\{}\ field, \ldots\ \textbf{\}}} \qquad \frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2}{C \vdash_{wft} \sigma : \theta : e_1\ binop\ e_2} \qquad \frac{C \vdash_{wft} \sigma : \theta : e}{C \vdash_{wft} \sigma : \theta : unop\ e}$$

$$\frac{r \in dom(\sigma)}{C \vdash_{wft} \sigma : \theta : r} \qquad \frac{C \vdash_{wft} \sigma : \theta : e, \ldots}{C \vdash_{wft} \sigma : \theta : \textbf{<}\ e, \ldots\ \textbf{>}} \qquad \frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2 \ldots}{C \vdash_{wft} \sigma : \theta : e_1\ (e_2, \ldots)}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2 \ldots}{C \vdash_{wft} \sigma : \theta : e_1\ \textbf{:}\ x\ (e_2, \ldots)}$$

Definición B.1.4 (Buena formación de campos de tabla)

$$\frac{C \vdash_{wft} \sigma : \theta : e_1 \qquad C \vdash_{wft} \sigma : \theta : e_2}{C \vdash_{wft\_field} \sigma : \theta : \mathbf{[}e_1\mathbf{]} = e_2}$$

$$\frac{C \vdash_{wft} \sigma : \theta : e}{C \vdash_{wft\_field} \sigma : \theta : e}$$

Definición B.1.5 (Buena formación de construcciones de tiempo de ejecución: expresiones)

$$\frac{\begin{array}{c} C \vdash_{wft} \sigma : \theta : e \\ \vdash_{mtch} e \approx (\!| s |\!)_{\text{RETEXP}} \quad \vee \\ \vdash_{mtch} e \approx \mathbf{<} v, \ldots \mathbf{>} \quad \vee \\ \vdash_{mtch} e \approx v_1\ (v_2, \ldots) \quad \vee \\ \vdash_{mtch} e \approx (\!| v_1\ (v_2, \ldots) |\!)_{\text{WRONGFUNCALL}} \end{array}}{C \vdash_{wft} \sigma : \theta : (\!| e |\!)_{\text{PROTMD}}}$$

$$\frac{\begin{array}{c} C \vdash_{wft} \sigma : \theta : e \\ C \vdash_{wft} \sigma : \theta : v \\ \vdash_{mtch} e \approx (\!| s |\!)_{\text{RETEXP}} \quad \vee \\ \vdash_{mtch} e \approx \mathbf{<} v, \ldots \mathbf{>} \quad \vee \\ \vdash_{mtch} e \approx v_1\ (v_2, \ldots) \quad \vee \\ \vdash_{mtch} e \approx (\!| v_1\ (v_2, \ldots) |\!)_{\text{WRONGFUNCALL}} \end{array}}{C \vdash_{wft} \sigma : \theta : (\!| e |\!)_{\text{PROTECTEDMODE}\ v}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad v_1 \notin tid}{C \vdash_{wft} \sigma : \theta : (\!| v_1[v_2] |\!)_{\text{NONTABLE}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v \qquad C \vdash_{wft} \sigma : \theta : tid \qquad v \notin dom(\theta)(1)}{C \vdash_{wft} \sigma : \theta : (\!| tid\,[v] |\!)_{\text{WRONGKEY}}}$$

$$\frac{\begin{array}{c} C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \\ (v_1 \notin num \wedge \delta(\mathbf{tonumber}, v_1) \notin num) \quad \vee \quad (v_2 \notin num \wedge \delta(\mathbf{tonumber}, v_2) \notin num) \end{array}}{C \vdash_{wft} \sigma : \theta : (\!| v_1\ binop\ v_2 |\!)_{\text{ARITHWO}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad v_1 \notin num \cup string \vee v_2 \notin num \cup string}{C \vdash_{wft} \sigma : \theta : (\!| v_1 \mathbf{..} v_2 |\!)_{\text{STRCONCATWO}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad \neg(\delta(\mathbf{type}, v_1) = \delta(\mathbf{type}, v_2) \wedge v_1 \in num \cup string)}{C \vdash_{wft} \sigma : \theta : (\!| v_1\ \mathbf{relop}\ v_2 |\!)_{\text{ORDCOMPWO}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v \qquad v_1 \notin string}{C \vdash_{wft} \sigma : \theta : (\!| \#v |\!)_{\text{STRLENWO}}} \qquad \frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \qquad \delta(\mathbf{==}, v_1, v_2) \neq \mathbf{true}}{C \vdash_{wft} \sigma : \theta : (\!| v_1 \mathbf{==} v_2 |\!)_{\text{EQFAIL}}}$$

$$\frac{C \vdash_{wft} \sigma : \theta : v_1 \qquad C \vdash_{wft} \sigma : \theta : v_2 \ldots \qquad v_1 \notin functiondef}{C \vdash_{wft} \sigma : \theta : (\!| v_1\ (v_2, \ldots) |\!)_{\text{WRONGFUNCALL}}} \qquad \frac{C[\![\ (\!|\ [\![\ ]\!]\ |\!)_{\text{RETEXP}}\ ]\!] \vdash_{wft} \sigma : \theta : s}{C \vdash_{wft} \sigma : \theta : (\!| s |\!)_{\text{RETEXP}}}$$

B.2. Buena formación de configuraciones

Finalmente, buena formación de configuraciones se reduce a la siguiente regla:

Definición B.2.1 (Buena formación de configuraciones)

$$\frac{C \vdash_{wft} \sigma : \theta : s \qquad \forall r \in \text{dom}(\sigma), C \vdash_{wft} \sigma : \theta : \sigma(r) \qquad \forall l \in \text{dom}(\theta), C \vdash_{wft} \sigma : \theta : \theta(l)}{C \vdash_{wfc} \sigma : \theta : s}$$

Bibliography

Here are the references in citation order.

[1] L. H. de Figueiredo R. Ierusalimschy y W. Celes. «Passing a language through the eye of a needle». En: *ACM Queue* 9.5 (2011), págs. 20-29 (vid. pág. 1).

[2] L. H. de Figueiredo R. Ierusalimschy y W. Celes. «Lua – an extensible extension language». En: *Software: Practice and Experience* 26.6 (1996), págs. 635-652 (vid. págs. 1, 34).

[3] Waldemar Celes Filho Luiz Henrique de Figueiredo Roberto Ierusalimschy. «The design and implementation of a language for extending applications». En: *Proceedings of XXI Brazilian Seminar on Software and Hardware*. 1994, págs. 273-283 (vid. pág. 1).

[4] R. Ierusalimschy, L. H. de Figueiredo y W. Celes. «The evolution of an extension language: a history of Lua». En: *Brazilian Symposium on Programming Languages*. 2001 (vid. págs. 1, 8).

[5] S. Maffeis, J. C. Mitchell y A. Taly. «An Operational Semantics for JavaScript». En: *APLAS '08*. 2008 (vid. págs. 4, 93).

[6] P. J. Landin. «Correspondence between ALGOL 60 and Church's Lambda-Notation: Part I». En: *Commun. ACM* 8.2 (feb. de 1965), págs. 89-101. DOI: 10.1145/363744.363749 (vid. pág. 4).

[7] P. J. Landin. «A Correspondence between ALGOL 60 and Church's Lambda-Notations: Part II». En: *Commun. ACM* 8.3 (mar. de 1965), págs. 158-167. DOI: 10.1145/363791.363804 (vid. pág. 4).

[8] M. Bodin y col. «A trusted mechanised JavaScript specification». En: *POPL '14*. 2014 (vid. págs. 4, 93).

[9] Hanshu Lin. «Operational Semantics for Featherweight Lua». Tesis de mtría. San José State University, mar. de 2015 (vid. págs. 4, 94).

[10] M. Felleisen, R. B. Finlder y M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 2009 (vid. págs. 5, 11, 12, 14, 17, 89).

[11] A. Guha, C. Saftoiu y S. Krishnamurthi. «The Essence of JavaScript». En: *ECOOP '10*. 2010 (vid. págs. 5, 14, 19, 21, 30, 93, 96).

[12] J. G. Politz y col. «Python: The Full Monty: A Tested Semantics for the Python Programming Language». En: *OOPSLA '13*. 2013 (vid. págs. 5, 14, 19, 93, 96).

[13] J. G. Politz y col. «A tested semantics for getters, setters, and eval in JavaScript». En: *DLS '12*. 2012 (vid. págs. 5, 14, 21, 31, 93).

[14] Jacob Matthews y Robert Bruce Findler. «An operational semantics for Scheme». En: *Journal of Functional Programming* (2007) (vid. págs. 5, 93).

[15] G.D. Plotkin. «Call-by-name, call-by-value and the lambda-calculus». En: *Theoretical Computer Science* 1.2 (1975), págs. 125-159. DOI: https://doi.org/10.1016/0304-3975(75)90017-1 (vid. pág. 12).

[16] H. P. Barendregt. *The lambda calculus. Its syntax and semantics.* North-Holland Publishing Company, 1981 (vid. pág. 12).

[17] Matthias Felleisen. «The calculi of Lambda-v-CS conversion: a syntactic theory of control and state in imperative higher-order programming languages». Tesis doct. Indiana University, 1987 (vid. págs. 13, 14).

[18] Marco Servetto y Lindsay Groves. «True small-step reduction for imperative object oriented languages». En: *FTfJP '13*. 2013 (vid. pág. 14).

[19] A. Capriccioli, M. Servetto y E. Zucca. «An imperative pure calculus». En: *ENTCS* 322 (2016). DOI: 10.1016/j.entcs.2016.03.007 (vid. pág. 14).

[20] John C. Reynolds. «Theories of Programming Languages». En: Cambridge University Press, 2009 (vid. págs. 15, 21).

[21] Federico Biancuzzi y Shane Warden. *Masterminds of Programming: Conversations with the Creators of Major Programming Languages*. 1st. O'Reilly Media, Inc., 2009 (vid. pág. 16).

[22] P. J. Landin. «The next 700 programming languages». En: *Communications of the ACM* 9 (1966), págs. 157-166 (vid. págs. 19, 42).

[23] Mallku Soldevila y col. «Decoding Lua: Formal Semantics for the Developer and the Semanticist». En: *Proceedings of the 13th ACM SIGPLAN Dynamic Languages Symposium*. DLS 2017. 2017 (vid. pág. 19).

[24] Roberto Ierusalimschy y Waldemar Celes Luiz Henrique de Figueiredo. «"The evolution of Lua"». En: *HOPL III*. ACM New York, NY, USA, 2007 (vid. pág. 31).

[25] Kein-Hong Man. *A No-Frills Introduction to Lua 5.1 VM Instructions*. Available at luaforge.net/docman/83/98/ANoFrillsIntroToLua51VMInstructions.pdf (vid. pág. 35).

[26] Seif Haridi Peter Van Roy. *Concepts, Techniques and Models of Computer Programming*. The MIT Press, 2004 (vid. pág. 36).

[27] Steven Jaconette Casey Klein Jay McCarthy y Robert Bruce Findler. «A Semantics for Context-Sensitive Reduction Semantics». En: *APLAS'11*. 2011 (vid. pág. 40).

[28] Roberto Ierusalimschy. *Programming in Lua*. 4th. Lua.org, 2016 (vid. pág. 48).

[29] Casey Klein y col. «Run Your Research: On the Effectiveness of Lightweight Mechanization». En: *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '12. Philadelphia, PA, USA: ACM, 2012, págs. 285-296. DOI: 10.1145/2103656.2103691 (vid. págs. 53, 91).

[30] Robert Harper. *Practical Foundations for Programming Languages*. 2nd. USA: Cambridge University Press, 2016 (vid. pág. 53).

[31] G. Morrisett, M. Felleisen y R. Harper. «Abstract models of memory management». En: *FPCA '95*. 1995 (vid. págs. 57, 77, 84, 85, 94).

[32] Assaf Kfoury Kevin Donnelly J. J. Hallett. «Formal semantics of weak references». En: *ISMM '06 Proceedings of the 5th international symposium on Memory management*. 2006, págs. 126-137 (vid. págs. 58, 94).

[33] Marcus Amorim Leal y Roberto Ierusalimschy. «A Formal Semantics for Finalizers». En: *J. UCS* 11.7 (2005), págs. 1198-1214. DOI: 10.3217/jucs-011-07-1198 (vid. págs. 59, 62, 72, 77, 94).

[34] Yarom Gabay y Assaf J. Kfoury. «A Calculus for Java's Reference Objects». En: *SIGPLAN Not.* 42.8 (ago. de 2007), págs. 9-17. DOI: 10.1145/1294297.1294299 (vid. págs. 59, 94).

[35] Hayes B. «Finalization in the collector interface». En: Springer, Berlin, Heidelberg, 1992 (vid. pág. 62).

[36] Hans-J. Boehm. «Destructors, Finalizers, and Synchronization». En: *SIGPLAN Not.* 38.1 (ene. de 2003), págs. 262-272. DOI: 10.1145/640128.604153 (vid. pág. 62).

[37] Barry Hayes. «Ephemerons: A New Finalization Mechanism». En: *SIGPLAN Not.* 32.10 (oct. de 1997), págs. 176-183. DOI: 10.1145/263700.263733 (vid. págs. 69, 94).

[38] Alexandra Barros y Roberto Ierusalimschy. «Eliminating Cycles in Weak Tables.» En: 14 (ene. de 2008), págs. 3481-3497 (vid. págs. 69, 94).

[39] Simon Peyton Jones, Simon Marlow y Conal Elliott. «Stretching the Storage Manager: Weak Pointers and Stable Names in Haskell». En: mayo de 2000. DOI: 10.1007/10722298_3 (vid. pág. 71).

[40] Marcus Amorim Leal. «Finalizadores e referências fracas: interagindo com o colector de lixo.» Tesis doct. Pontificia Universidade Católica do Rio de Janeiro, 2005 (vid. pág. 74).

[41] Rob Hunter y Shriram Krishnamurthi. «A Model of Garbage Collection for OO Languages». En: *Tenth International Workshop on Foundations of Object-Oriented Lang*. FOOL10, 2003 (vid. pág. 77).

[42] Casey Klein. «Randomized Testing in PLT Redex». En: *Proc. Scheme and Functional Programming*. 2009, págs. 26-36 (vid. pág. 90).

[43] A. Igarashi, B. C. Pierce y P. Wadler. «Featherweight Java: a minimal core calculus for Java and GJ». En: *TOPLAS* 23 (2001), págs. 396-450 (vid. pág. 94).

[44] R. Krebbers y F. Wiedijk. «Separation logic for non-local control flow and block scope variables». En: *FOSSACS'13*. 2013. DOI: 10.1007/978-3-642-37075-5_17 (vid. pág. 95).

[45] A. M. Maidl, F. Mascarenhas y R. Ierusalimschy. «A Formalization of Typed Lua». En: *DLS '15*. Pittsburgh, PA, USA, 2015. DOI: 10.1145/2816707.2816709 (vid. pág. 95).