5 PROGRAMACIÓN FUNCIONAL

En este capítulo introduciremos los fundamentos teóricos de la programación funcional. Para ello, comenzaremos por introducir un lenguaje de programación abstracto que refleja lo que entendemos por lenguaje funcional. **AL** [45] es un ejemplo de lo que se conoce como lenguajes orientados a expresiones, cuyo objetivo es expresar de manera precisa problemas computacionales (ALgoritmos). Por lo tanto, AL debe ser completo, en el sentido de que cualquier computación intuitiva, pueda ser especificada por medios finitos. La sintaxis de AL debe permitir la construcción de algoritmos complejos a partir de partes más simples; y su semántica debe definir el significado de los algoritmos: Qué se va a computar y, en alguna medida, cómo. La primer sección de este capítulo introduce AL, su sintaxis y su semántica. Se revisan algunos ejemplos de algoritmos implementados en AL y se ejemplifica también su evaluación.

AL

Con un lenguaje como éste es posible tender un puente entre el nivel algorítmico y la especificaciones de una máquina capaz de ejecutar tales descripciones algorítmicas. El puente es el **Cálculo-λ** [16], una teoría de las funciones computables que expresa propiedades elementales de operadores y operandos, aplicaciones y el rol de las variables en la computación. Ese será el tema de la segunda sección, donde introduciremos la sintaxis y semántica de una versión mínima del cálculo, conocida como cálculo-λ puro. Se revisaran las propiedades del cómputo que se lleva a cabo y se establecerán los paralelismos correspondientes con AL.

Cálculo-λ

5.1 AL.

Se dice que AL es un lenguaje orientado a **expresiones**, porque los algoritmos en él expresados, son expresiones que al computarse producen **valores**. Las computaciones se definen mediante un conjunto definido de **reglas de transformación**. Estas reglas se aplican sistemáticamente hasta que ya no es posible aplicar regla alguna. El valor encontrado en la última transformación es el valor que deseábamos computar.

Expresiones
Valores
Reglas de transformación

Debe observarse que los valores tienen una interpretación más general en este lenguaje, que en los lenguajes tradicional. Los valores no son necesariamente atómicos, como los números, sino que pueden ser agregados complejos de expresiones que no pueden ser transformados en nada más y son por tanto, en cierto sentido, **constantes**. Tales expresiones constantes pueden incluir funciones, aplicaciones de funciones y variables en contextos específicos.

Valores constantes

Denotemos a las expresiones como e o e_i con $i \in \{0, \ldots, n\}$. Entonces, podemos describir una **computación** como una secuencia:

Computación

$$e_0 \to e_1 \to \cdots \to e_i \to e_{i+1} \to \cdots \to e_n$$

donde e_0 es la expresión inicial y e_n es la expresión terminal ó el valor de e_0. La transformación de e_i a e_{i+1} se da por la aplicación de una única regla de transformación.

Las expresiones en esta secuencia tienen una propiedad muy importante: si e_n es el valor de e_0, entonces podemos decir que ambas expresiones significan lo mismo o que son **semánticamente equivalentes**, aunque sintácticamente difieran. Siendo este el caso, también podemos decir que e_1 es equivalente a e_n y lo mismo para todas las expresiones en la secuencia menores que n. Es decir, las reglas de transformación que definen las computaciones llevadas a cabo **preservan el significado**, pues

Equivalencia semántica
Preservación de significado

obviamente remplazan iguales por iguales. Esta idea puede ilustrarse evaluando en tres transformaciones las siguiente expresión aritmética:

Ejemplo 5.1. *La evaluación de una expresión aritmética preserva significado:*

$$((5+3)\times(8/4)) \rightarrow (8\times(8/4)) \rightarrow (8\times 2) \rightarrow 16$$

Es evidente que todas las sub-expresiones en esta computación son, debido a las reglas de la aritmética, remplazadas sistemáticamente de adentro hacía afuera por números. Las expresiones en la cadena difieren sintácticamente, pero son equivalentes semánticamente, tienen el mismo significado. La equivalencia semántica, por otro lado, no nos dice nada sobre lo que significa una expresión por si misma, otro nivel de interpretación es necesario para ello. En un sentido estricto, podremos hablar de figuras sintácticas y transformaciones puramente sintácticas de expresiones, en otras expresiones que consideramos semánticamente equivalentes; sin que podamos hablar del significado o semántica de las expresiones en si mismas. P. ej., podemos decir que (8×2) es equivalente a 16, pero no que significa la multiplicación.

Aunque el lenguaje que estamos presentando no es un lenguaje estrictamente funcional, si tiene como implementación más cercana a **Lisp** [56] y a su intérprete *EVAL-APPLY*. Esta es una de las razones para comenzar la segunda parte de este curso con esta revisión.

Lisp

5.1.1 Sintaxis

Las expresiones simbólicas en AL incluyen **valores constantes** tales como números (por ahora no distinguiremos entre enteros y reales), cadenas de caracteres, valores booleanos (falso y verdadero), variables y los símbolos primitivos para operadores aritméticos, lógicos y relacionales. Estas expresiones se llaman **átomos** o términos de base (en inglés *grounded*), puesto que no están compuestos por otras expresiones del lenguaje.

Constantes

Atomos

Con los átomos como base, procederemos a definir algunos constructores sintácticos o **formas sintácticas** que están compuestas de expresiones y son a su vez expresiones. Estas formas se usaran para construir expresiones más complejas substituyendo expresiones en posiciones sintácticas reservadas para expresiones. En las definiciones siguientes, asumimos que $e_0, e_1, \ldots, e_n$ y e denotan expresiones válidas en el lenguaje; y que $v_1, v_2, \ldots, v_m$ y $f_1, \ldots, f_k$ denotan variables, también conocidas como identificadores. Informalmente las formas sintácticas son las siguientes:

Formas sintácticas

- $(e_0\ e_1 \ldots e_n)$ denota la **aplicación** de una expresión e_0 a las expresiones $e_1 \ldots e_n$. La expresión e_0 está en la posición del **operador** y las otras expresiones en la posición de los **operandos**. Las aplicaciones son las expresiones más importantes del lenguaje ya que son a ellas que las reglas de transformación se aplican. Observen el uso de la **notación prefija** con fines de uniformidad.

  Aplicación *Operador y operandos* *Prefijo*

- `if` e_0 `then` e_1 `else` e_2 es una forma sintáctica especial, conocida como **condicional**, que denota la aplicación del predicado e_0 a las expresiones consecuente e_1 y alternativa e_2. Su objetivo es elegir e_1 o e_2 para su posterior evaluación, dependiendo del valor del predicado e_0 (`true` o `false`).

  Condicional

- `lambda` $v_1 \ldots v_n$ `in` e_0 denota una **función anónima** de aridad n. Por anónima, entendemos que se trata de una función sin nombre. Se dice también que esta expresión es una **abstracción** de las variables $v_1 \ldots v_n$ de la expresión e_0. El cuerpo de la abstracción (e_0) especifica la computación a realizar. Se dice que `lambda`, conocido como abstractor, **liga** las variables $v_1 \ldots v_n$ en el cuerpo de la abstracción, que a su vez, determina el **alcance** del abstractor. En lo que sigue, abstracción y función se tomarán como sinónimos.

  Función anónima *Abstracción funcional* *Liga* *Alcance*

- $\langle e_1 \dots e_n \rangle$ denota una **lista** n-aria ó una secuencia de expresiones en un orden particular, en el que tiene sentido hablar del primer, último e i-ésimo elemento. La lista vacía se denota por $\langle\rangle$. *Lista*
- `letrec` $f_1 = e_1 \dots f_k = e_k$ `in` e_0 es la expresión más compleja del lenguaje, correspondiente a un conjunto de **definiciones recursivas**. `letrec` precede a un conjunto de ecuaciones que igualan las variables $f_1, \dots, f_k$ con las expresiones $e_1, \dots, e_k$ recursivamente y las ligan en las expresiones $e_1, \dots, e_k$ en e_0. De hecho, las variables f_i pueden verse como **nombres** o identificadores de funciones, por medio de las cuales puede hacerse referencia a la parte derecha de la ecuación más adelante en la expresión `letrec`. El valor de la expresión es el valor de la expresión meta e_0 en la cual la ocurrencia de los identificadores $f_1, \dots, f_k$ es remplazada recursivamente por el lado derecho de sus definiciones. *Definiciones recursivas* *Nombres*
- `let` $v_1 = e_1 \dots v_n = e_n$ `in` e_0 es una versión **no recursiva** de `letrec`. *Definiciones no recursivas*
- Ninguna otra expresión es válida en el lenguaje.

De manera concisa la **sintaxis** de AL puede escribirse como: *Sintaxis*

$$
\begin{aligned}
e =_s\ & const \mid var \mid oper \mid \\
& (e_0\ e_1 \dots e_n) \mid \\
& \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2 \mid \\
& \texttt{let}\ v_1 = e_1 \dots v_n = e_n\ \texttt{in}\ e_0 \mid \\
& \texttt{lambda}\ v_1 \dots v_n\ \texttt{in}\ e_0 \mid \\
& \langle\rangle \mid \langle e_1 \dots e_n \rangle \mid \\
& \texttt{letrec}\ f_1 = e_1 \dots f_n = e_n\ \texttt{in}\ e_0.
\end{aligned}
\tag{5.1}
$$

donde el signo $=_s$ denota equivalencia sintáctica. La expresión *const*, denota valores constantes; *var*, variables; y *oper*, operaciones primitivas de AL.

5.1.2 Semántica

Ahora estamos listos para definir la forma en que las expresiones en AL serán evaluadas. Para ello solo debemos definir **reglas** para computar el valor de las formas sintácticas definidas en la ecuación 5.1. Como un primer paso, definiremos estas reglas independientemente de cualquier mecanismo o maquinaria que las vaya a ejecutar, para proveer una idea general de cómo debemos proceder por principios. *Reglas de evaluación*

Con este propósito definiremos un **evaluador abstracto** llamado `EVAL`. Se le debe considerar una meta-función que mapea cada forma sintáctica en otra que representa su valor, o significado. En este sentido, `EVAL` puede considerarse la **función semántica** del lenguaje. Esta meta-función se define mediante aplicaciones recursivas sobre todas, o algunas, de las sub-expresiones de una forma sintáctica, seleccionadas conforme a una **estrategia de evaluación**, que debería computar valores resultantes con un casi mínimo costo computacional. *Evaluador abstracto* *Función semántica* *Estrategia de evaluación*

Definiremos `EVAL` con una estrategia intuitiva: los operandos de una aplicación generalmente deberán evaluarse antes que el operador se aplique a ellos. Si algún caso causa problemas, se aislará y se le procesará de forma diferente. Esta estrategia es directamente implementable y conlleva una alta eficiencia en tiempo de ejecución. De hecho, esta estrategia se implemente en la mayoría de los lenguajes de programación imperativos, y en ese contexto se conoce como estrategia de **llamada por valor**. *Llamada por valor*

La aplicación de `EVAL` a una expresión e se denota `EVAL`$[\![\ e\ ]\!]$ y recorriendo en forma descendente las formas sintácticas, tenemos:

- Las expresiones **atómicas** como constantes, variables y operadores primitivos evalúan a si mismos: *Evaluación de átomos*

$$\texttt{EVAL}[\![\ \textit{átomo}\ ]\!] \ = \ \textit{átomo}$$

- Para **aplicaciones** con expresiones no especificadas en las posiciones de operador y operandos: *Evaluación de aplicación*

$$\texttt{EVAL}[\![\ (e_0\ e_1 \ldots e_n)\ ]\!] \ = \ \texttt{EVAL}[\![\ (\texttt{EVAL}[\![\ e_0\ ]\!]\ \texttt{EVAL}[\![\ e_1\ ]\!] \ldots \texttt{EVAL}[\![\ e_n\ ]\!])\ ]\!]$$

- Para los **condicionales** tenemos: *Evaluación de condicionales*

$$\texttt{EVAL}[\![\ \texttt{if}\ e_0\ \texttt{then}\ e_1\ \texttt{else}\ e_2\ ]\!] = \begin{cases} \texttt{EVAL}[\![\ e_1\ ]\!] & \text{Si } \texttt{EVAL}[\![\ e_0\ ]\!] \ = \ \textit{true} \\ \texttt{EVAL}[\![\ e_2\ ]\!] & \text{Si } \texttt{EVAL}[\![\ e_0\ ]\!] \ = \ \textit{false} \\ \texttt{if EVAL}[\![\ e_0\ ]\!]\ \texttt{then}\ e_1\ \texttt{else}\ e_2 & \text{En cualquier otro caso} \end{cases}$$

  observen que si la forma e_0 no tiene un valor booleano, la expresión de queda casi idéntica, excepto que la expresión e_0 es evaluada.

- Para las **definiciones** tenemos: *Evaluación de definiciones*

$$\texttt{EVAL}[\![\ \texttt{let}\ v_1 = e_1 \ldots v_n = e_n\ \texttt{in}\ e_0\ ]\!] = \texttt{EVAL}[\![\ e_0[\ v_1 \leftarrow \texttt{EVAL}[\![\ e_1\ ]\!] \ldots v_n \leftarrow \texttt{EVAL}[\![\ e_n\ ]\!]\ ]\ ]\!]$$

  esto es, la forma evalúa al valor de e_0 en donde todas las ocurrencias de las variables ligadas por `let` han sido substituidas por sus valores respectivos, mediante $\leftarrow\ \texttt{EVAL}[\![\ e_i\ ]\!] \mid i \in \{1, \ldots, n\}$ denotando las **substituciones**. *Substituciones*

- Para las **abstracciones** funcionales que ocurren en una posición distinta a la del operador, tenemos que: *Evaluación de abstracciones*

$$\texttt{EVAL}[\![\ \texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ ]\!] \ = \ \texttt{lambda}\ v_1 \ldots v_n\ \texttt{in EVAL}[\![\ e_0\ ]\!]$$

  lo que significa que las abstracciones necesitan tener su cuerpo evaluado para determinar sus valores.

- Para las **listas**: *Evaluación de listas*

$$\begin{aligned} \texttt{EVAL}[\![\ \langle\rangle\ ]\!] &= \langle\rangle \quad \text{y} \\ \texttt{EVAL}[\![\ \langle e_1 \ldots e_n \rangle\ ]\!] &= \langle \texttt{EVAL}[\![\ e_1\ ]\!] \ldots \texttt{EVAL}[\![\ e_n\ ]\!] \rangle \end{aligned}$$

  se computan recursivamente los valores de sus sub-expresiones preservando su estructura.

- Para las expresiones **definiciones recursivas**: *Evaluación de definiciones recursivas*

$$\texttt{EVAL}[\![\ \texttt{letrec}\ \ldots f_i = e_i \ldots\ \texttt{in}\ e_0\ ]\!] = \texttt{EVAL}[\![\ e_0[\ \ldots f_i \leftarrow \texttt{EVAL}[\![\ e_i[\ \ldots f_i \leftarrow \texttt{letrec}\ \ldots\ \texttt{in}\ f_i \ldots]\ ]\!] \ldots]\ ]\!]$$

  esta definición, aparentemente complicada, debe leerse como sigue: el valor de la expresión `letrec` completa es el valor de su término meta e_0, computado al expandir todas las ocurrencias de los identificadores de función f_i con los valores e_i en la mano derecha de sus ecuaciones definitorias. Estos valores a su vez deben computarse substituyendo las ocurrencias de los identificadores f_i en las expresiones nuevamente, por copias de las expresiones completas en `letrec`, las cuales sin embargo, tienen su expresión meta remplazada por los mismos f_i. Las formas sintácticas especializada $\texttt{letrec} \ldots f_i = e_i \ldots \texttt{in}\ f_i$ de hecho representa las expresiones e_i en su forma no evaluada. Tan pronto como los f_i desaparecen de la expresión, la expresión `letrec` misma desaparece pues ya no es necesaria para que la computación continúe.

Hemos recorrido todas las formas sintáctica y sin embargo, la definición de `EVAL` está lejos de ser completa. Nos falta definir los casos especiales donde las expresiones en la posición del operador son abstracciones y operaciones primitivas. Estas aplicaciones definen realmente las acciones de transformación de expresiones, por ejemplo, los casos estándar de **aplicaciones completas** (función u operador de aridad n, aplicada a n argumentos) se transforman como sigue:

Evaluación de aplicaciones completas

$$\texttt{EVAL}[\![\ (\texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ e_1 \ldots e_n)\ ]\!] =$$
$$\texttt{EVAL}[\![\ e_0\ [v_1 \leftarrow \texttt{EVAL}[\![\ e_1\ ]\!] \ldots v_n \leftarrow \texttt{EVAL}[\![\ e_n\ ]\!]]\ ]\!]$$

La evaluación de estas aplicaciones resulta en el valor de la expresión e_0, en donde todas las ocurrencias de las variables ligadas (los parámetros formales de las abstracciones) son substituidos de izquierda a derecha por los valores de los operandos en el orden en que ocurren en la aplicación.

Evaluación de aplicaciones bajo desigualdad

Sin embargo, la sintaxis del lenguaje también permite **desigualdades** entre la aridad n de una función y el número m de sus argumentos. En esos casos podemos optar por una aproximación conservadora, definiendo el valor de tal aplicación como una cadena de caracteres comunicando el error:

$$\texttt{EVAL}[\![\ (\texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ e_1 \ldots e_m)\ ]\!] \mid m \neq n =$$
$$\text{“función de aridad } n \text{ recibiendo } m \text{ argumentos”}$$

La computación puede entonces detenerse y regresar esa cadena como el valor de la expresión completa. De manera alternativa se puede intentar mejores soluciones, por ejemplo:

- Si el número de argumentos m es mayor que la aridad de la abstracción n, debemos definir el valor como una **nueva aplicación** cuyo operador es una expresión resultado de la aplicación de la abstracción de aridad n a n argumentos y cuyos operandos son los restantes $m - n$ argumentos evaluados:

  Nueva aplicación

  $$\texttt{EVAL}[\![\ (\texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ e_1 \ldots e_m)\ ]\!] \mid m > n =$$
  $$\texttt{EVAL}[\![\ (\ \texttt{EVAL}[\![\ e_0[\ v_1 \leftarrow \texttt{EVAL}[\![\ e_1\ ]\!] \ldots v_n \leftarrow \texttt{EVAL}[\![\ e_n\ ]\!]\ ]\ ]\!]$$
  $$\texttt{EVAL}[\![\ e_{n+1}\ ]\!] \ldots \texttt{EVAL}[\![\ e_m\ ]\!]\ )\ ]\!]$$

- Si la aridad n de la abstracción excede el número de argumentos m, en cuyo caso tenemos una **aplicación parcial**, el valor es definido como sigue:

  Aplicación parcial

  $$\texttt{EVAL}[\![\ (\texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ e_1 \ldots e_m)\ ]\!] \mid m < n =$$
  $$\texttt{lambda}\ v_{m+1} \ldots v_n\ \texttt{in}\ \texttt{EVAL}[\![\ e_0[\ v_1 \leftarrow \texttt{EVAL}[\![\ e_1\ ]\!] \ldots v_m \leftarrow \texttt{EVAL}[\![\ e_m\ ]\!]\ ]\ ]\!]$$

Conflictos de nombres

Aunque aún no contamos con los elementos para resolver el problema, es necesario señalar que el caso donde $m < n$ puede introducir potencialmente **conflictos entre nombres** de las variables `lambda`-acotadas renombradas, que resultan de la abstracción y las variables libres en las expresiones argumento que están siendo substituidas bajo la abstracción `lambda`. Los casos $n = m$ y $m > n$ y la evaluación de abstracciones aisladas no están completamente libres de este problema, ya que el cuerpo de la abstracción puede incluir recursivamente otras abstracciones que pueden ser penetradas por las substituciones. El renombrado de variables no se considera una solución apropiada a este problema, dada la complejidad inherente al crecer el tamaño y la complejidad de los algoritmos. Por el momento, si queremos seguridad al respecto, podemos adoptar una estrategia conservadora definiendo las evaluaciones parciales como:

$$\texttt{EVAL}[\![\ (\texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ e_1 \ldots e_m)\ ]\!] \mid m < n =$$
$$[\ \texttt{EVAL}[\![\ e_m\ ]\!] \ldots \texttt{EVAL}[\![\ e_1\ ]\!]\ \texttt{lambda}\ v_1 \ldots v_n\ \texttt{in}\ e_0\ ]$$

expresión que tiene los argumentos evaluados y la abstracción empaquetados entre corchetes, en lo que se conoce como una **cerradura** . Este mecanismo representa el valor de una nueva abstracción de aridad $n - m$ sin realmente computarlo., y por tanto, evitando las substituciones bajo las abstracciones y los conflictos entre nombres que las acompañan. Las cerraduras deben tratarse como abstracciones normales, es decir, pueden ser usadas como operadores y operandos de otras aplicaciones y tomar más argumentos hasta que puedan computarse llevando a a cabo las substituciones pospuestas.

Cerradura

Consideraciones similares con respecto a las aridades, aunque no impliquen conflictos entre nombres, aplican en el caso de las operaciones primitivas. Para las **operaciones aritméticas** (binarias) que denotaremos con op_arit, y usando num, num_1 y num_2 para denotar números, tenemos que:

Operaciones aritméticas

$$\texttt{EVAL}[\![\ (\ op\_arit\ e_1\ e_2\ )\ ]\!] =$$
$$\begin{cases} num & \text{Si } num_1 = \texttt{EVAL}[\![\ e_1\ ]\!] \wedge num_2 = \texttt{EVAL}[\![\ e_2\ ]\!] \\ (\ op\_arit\ \texttt{EVAL}[\![\ e_1\ ]\!]\ \texttt{EVAL}[\![\ e_2\ ]\!]\ ) & \text{En cualquier otro caso} \end{cases}$$

esto es, el valor de la aplicación es un número si sus dos argumentos son números o evalúan a números. El valor se obtiene aplicando el operador al valor de los dos argumentos. En cualquier otro caso, la aplicación se mantiene como tal salvo que e_1 y e_2 son remplazados por sus valores.

Para las **operaciones relacionales**, denotadas por op_rel, tenemos que además, str_1 y str_2 denotan cadenas de caracteres, mientras que $bool$ denota un valor booleano:

Operaciones relacionales

$$\texttt{EVAL}[\![\ (\ op\_rel\ e_1\ e_2\ )\ ]\!] =$$
$$\begin{cases} bool & \text{Si } str_1 = \texttt{EVAL}[\![\ e_1\ ]\!] \wedge str_2 = \texttt{EVAL}[\![\ e_2\ ]\!] \\ bool & \text{Si } num_1 = \texttt{EVAL}[\![\ e_1\ ]\!] \wedge num_2 = \texttt{EVAL}[\![\ e_2\ ]\!] \\ (\ op\_rel\ \texttt{EVAL}[\![\ e_1\ ]\!]\ \texttt{EVAL}[\![\ e_2\ ]\!]\ ) & \text{En cualquier otro caso} \end{cases}$$

donde el valor de la aplicación es un booleano si sus argumentos son numéricos o cadenas de caracteres, en cuyo caso, el operador utiliza el orden léxico para establecer su **valor de verdad**. En cualquier otro caso la aplicación permanece intacta, salvo que e_1 y e_2 son substituidos por sus valores.

Valores de verdad

Para completar esta historia, definiremos la semántica de algunas operaciones sobre **listas**. Aprovecharemos el hecho de que las expresiones que componen las listas no necesariamente deben evaluarse en orden para producir listas primitivas funcionalmente aplicables. Esto es, `EVAL` procede sobre los operandos solo cuando se puede decidir que son listas o algo más que aplicaciones:

Operaciones con listas

$$\texttt{EVAL}[\![\ (\ \texttt{empty}\ e\ )\ ]\!] = \begin{cases} \texttt{true} & \text{Si } \texttt{EVAL}[\![\ e\ ]\!] = \langle\rangle \\ \texttt{false} & \text{Si } \texttt{EVAL}[\![\ e\ ]\!] = \langle e_1 \ldots e_n \rangle \\ (\texttt{empty}\ \texttt{EVAL}[\![\ e\ ]\!]\ ) & \text{En cualquier otro caso} \end{cases}$$

$$\texttt{EVAL}[\![\ (\ \texttt{first}\ e\ )\ ]\!] = \begin{cases} e_1 & \text{Si } \texttt{EVAL}[\![\ e\ ]\!] = \langle e_1 \ldots e_n \rangle \\ (\ \texttt{first}\ \texttt{EVAL}[\![\ e\ ]\!]\ ) & \text{En cualquier otro caso} \end{cases}$$

$$\texttt{EVAL}[\![\ (\ \texttt{rest}\ e\ )\ ]\!] = \begin{cases} \langle e_2 \ldots e_n \rangle & \text{Si } \texttt{EVAL}[\![\ e\ ]\!] = \langle e_1\ e_2 \ldots e_n \rangle \\ (\ \texttt{rest}\ \texttt{EVAL}[\![\ e\ ]\!]\ ) & \text{En cualquier otro caso} \end{cases}$$

$$\texttt{EVAL}[\![\ (\ \texttt{append}\ e_1\ e_2\ )\ ]\!] = \begin{cases} \langle e_{11} \ldots e_{1n} \ldots e_{21} \ldots e_{2m} \rangle & \text{Si } \texttt{EVAL}[\![\ e_1\ ]\!] = \langle e_{11} \ldots e_{1n} \rangle \wedge \\ & \texttt{EVAL}[\![\ e_2\ ]\!] = \langle e_{21} \ldots e_{2m} \rangle \\ (\ \texttt{append}\ \texttt{EVAL}[\![\ e_1\ ]\!]\ \texttt{EVAL}[\![\ e_2\ ]\!]\ ) & \text{En cualquier otro caso} \end{cases}$$

Varias observaciones son importantes en este punto. Primero, la definición de EVAL incluye una **verificación dinámica de tipos**. Las aplicaciones de esas funciones son evaluadas solo si sus argumentos son tipo compatibles, en cualquier otro caso quedan intactas (exceptuando que sus argumentos son substituidos por sus valores). Segundo, la meta función EVAL define la **semántica operacional** del lenguaje AL. Nos dice no solo que significa una expresión en el lenguaje (que valor tiene); pero también cómo una persona o una máquina puede computar ese valor.

Verificación dinámica de tipos

Semántica operacional

Además, observen que cuando EVAL aparece en una aplicación, se propaga al frente de sus sub-expresiones, pero el EVAL al frente de la aplicación no desaparece. Esto significa que las sub-expresiones deben ser evaluadas antes que la aplicación completa pueda ser evaluada. Sin embargo, puesto que las sub-expresiones son sintácticamente independientes, pueden ser evaluadas en cualquier orden, incluso simultáneamente; !Y sus valores siempre serán los **mismos**!

Determinismo

Las evaluaciones son propagadas recursivamente, de afuera hacía adentro, hasta que las sub-expresiones encontradas sean, dada su definición, valores por si mismas y permitan que su EVAL desaparezca. Lo mismo sucede con los EVALs al frente de aplicaciones cuyos componentes son todos valores atómicos. Estas formas evaluarán a algo diferente, si aplican operadores legítimos a operandos tipo compatibles; o permanecerán intactos (de la manera definida) en cuyo caso ellos mismos son su propio valor. Estos EVALs pueden verse como peticiones que fuerzan la evaluación de sus sub-expresiones. Tales peticiones se satisfacen de adentro hacía afuera desapareciendo así los EVALs.

Ejemplo 5.2. *Consideremos un ejemplo de evaluación, usando el lenguaje AL:*

$$\begin{array}{c} \texttt{EVAL}[\![\ (\ *\ (-\ 3\ (\ +\ 2\ 3\ )\ )(\ /\ 3\ 6\ )\ )\ ]\!] \\ \downarrow \\ \texttt{EVAL}[\![\ (\ *\ \texttt{EVAL}[\![\ (\ -\ 3\ (\ +\ 2\ 3\ ))\ ]\!]\ \texttt{EVAL}[\![\ (\ /\ 3\ 6\ )\ ]\!]\ )\ ]\!] \\ \downarrow \\ \texttt{EVAL}[\![\ (\ *\ \texttt{EVAL}[\![\ (\ -\ 3\ \texttt{EVAL}[\![\ (\ +\ 2\ 3)\ ]\!]\ )\ ]\!]\ 2\ )\ ]\!] \\ \downarrow \\ \texttt{EVAL}[\![\ (\ *\ \texttt{EVAL}[\![\ (\ -\ 3\ 5\ )\ ]\!]\ 2\ )\ ]\!] \\ \downarrow \\ \texttt{EVAL}[\![\ (\ *\ 2\ 2\ )\ ]\!] \\ \downarrow \\ 4 \end{array}$$

Ejemplo 5.3. *Ahora definamos factorial:*

$$fac = \textit{lambda}\ n\ \textit{in if}\ (>\ n\ 1)\ \textit{then}\ (*\ n\ (fac(-\ n\ 1)))\ \textit{else}\ 1)$$

y evaluemos la aplicación de factorial a 5:

$$\texttt{EVAL}[\![\ (fac\ 5)\ ]\!]$$
$$\downarrow$$
$$\texttt{EVAL}[\![\ \texttt{EVAL}[\![\ fac\ ]\!]\ \texttt{EVAL}[\![\ 5\ ]\!]\ ]\!]$$
$$\downarrow$$
$$\texttt{EVAL}[\![\ \textit{lambda n in if}\ (>\ n\ 1)\ \textit{then}\ (*\ n\ (fac(-\ n\ 1)))\ \textit{else}\ 1)\ 5\ ]\!]$$
$$\downarrow$$
$$\textit{if}\ \texttt{EVAL}[\![\ (>\ 5\ 1)\ ]\!]\ \textit{then}\ \texttt{EVAL}[\![\ (*\ 5\ (fac(-\ 5\ 1)))\ ]\!]\ \textit{else}\ \texttt{EVAL}[\![\ 1\ ]\!]$$
$$\downarrow$$
$$\textit{if true then}\ \texttt{EVAL}[\![\ (*\ 5\ (fac(-\ 5\ 1)))\ ]\!]\ \textit{else}\ \texttt{EVAL}[\![\ 1\ ]\!]$$
$$\downarrow$$
$$\texttt{EVAL}[\![\ (*\ 5\ (fac(-\ 5\ 1))))\ ]\!]$$
$$\downarrow$$
$$\texttt{EVAL}[\![\ (*\ 5\ \texttt{EVAL}[\![\ (fac(4)))\ ]\!]\ ]\!]$$
$$\vdots$$
$$\texttt{EVAL}[\![\ (*\ 5\ (*\ 4\ (*\ 3\ (*\ 2\ 1))))\ ]\!]$$
$$\vdots$$
$$120$$

5.2 EL CÁLCULO-λ

Algoritmos

El Cálculo-λ es el modelo de cómputo más cercano a los **algoritmos** y su evaluación tal y como fueron formalizados en la sección anterior. Se le puede considerar como el paradigma de todos los lenguajes de programación tal y como los conocemos actualmente. Es primordialmente, una teoría de las **funciones computables** que tiene que ver con propiedades fundamentales de los operadores, sus aplicaciones a operandos y con la construcción sistemática de operadores más complejos (algoritmos) a partir de componentes simples. Su núcleo tiene que ver con poco más que la definición de **variables**, su alcance y la su substitución ordenada por expresiones. Se trata de un **lenguaje cerrado** en el sentido de que su semántica puede definirse con base en la equivalencia entre expresiones del cálculo mismo.

Funciones computables

Lenguaje cerrado

El Cálculo-λ, desarrollado por Church [13], es uno de los modelos matemáticos que se propusieron como respuesta inmediata a los problemas de Hilbert, a principios del siglo XX. El problema en cuestión es si existe un método mecánico general para obtener el valor de verdad de cualquier conjetura lógica, y está íntimamente relacionado con la cuestión de lo que es **computable**.

Computabilidad

Otros modelos al respecto, incluyen los trabajos de Schoenfinkel y los combinadores de Curry (una forma especial de Cálculo-λ), los números de Göedel, el sistema de producción de Post, las funciones recursivas de Kleene, los algoritmos de Markov; y el más prominente con respecto a las computaciones mecánicas que puede llevar a cabo una máquina digital, la **máquina de Turing** [92]. Aunque no se tenga una idea muy clara de lo que es la computabilidad, un resultado reconfortante es que todos estos modelos son equivalentes. Lo cual nos lleva a la **tesis de Church-Turing** [44]: Los problemas intuitivamente o efectivamente computables son exactamente aquellos que pueden computarse en un máquina de Turing (y en los demás modelos también).

Máquina de Turing

Tesis Church-Turing

5.2.1 Sintaxis

Una **función** f de n variables $v_1, \ldots, v_n$ se denota en el Cálculo-λ como:

Función

$$f = \lambda v_1 \ldots v_n.e_0$$

Sustituyendo λ por `lambda` e introduciendo `in` en lugar del punto, obtendríamos la notación empleada en las abstracciones de AL, definidas en la sección anterior. El símbolo f en el lado izquierdo de la ecuación denota el nombre o **identificador** de la función y puede ser usado para hacer referencia a la función en alguna otra expresión. La expresión del lado derecho de la ecuación define una **abstracción** de las variables $v_1, \ldots, v_n$ en la expresión e_0. La expresión $\lambda v_1, \ldots, v_n$ se conoce como **abstractor** y e_0 como el **cuerpo** de la abstracción. El abstractor opera sobre las ocurrencias libres de las variables en el cuerpo de la abstracción. Una definición precisa de lo que significa la **ocurrencia** libre y acotada de las variables, deberá posponerse hasta que contemos con la notación adecuada para ello. Por ahora basta comentar que una variable es libre en el cuerpo de una abstracción, si no está incluida en el abstractor.

Identificador

Abstracción

Abstractor

Cuerpo

Ocurrencia de variables

Una **aplicación** de f sobre r argumentos $e_1, \ldots, e_r$ tiene la forma:

Aplicación

$$(f\ e_1 \ldots e_r) = (\lambda v_1 \ldots v_n.e_0\ e_1 \ldots e_r)$$

donde la aridad de la función (r), no es necesariamente igual al número de argumentos recibidos (n).

El caso especial de la aplicación de una abstracción a las **variables abstraídas**, nos da como resultado el cuerpo de la abstracción:

Aplicación a variables abstraídas

$$(\lambda v_1 \ldots v_n.e_0\ v_1 \ldots v_n) = e_0$$

Para mantener el aparato formal simple y conciso, el Cálculo-λ considera, sin perdida de generalidad, solamente abstracciones de una variable. Esto se debe al descubrimiento de Schoenfinkel y Curry que permite representar abstracciones n-arias, como n-pliegues anidados de abstracciones unarias (¿Recuerdan el extraño nombre de **funciones Curryficadas**? Pudo haber sido peor), es decir:

Funciones curryficadas

$$f = \lambda v_1 \ldots v_n.e_0 \equiv \lambda v_1.\lambda v_2 \ldots .\lambda v_n.e_0$$

Usando la notación currificada, la aplicación de f a r operandos toma la forma de r-pliegues de aplicaciones anidadas de operadores unarios:

$$(f\ e_1 \ldots e_r) \equiv (\ldots((f\ e_1)\ e_2) \ldots e_r)$$

Lo anterior reduce la construcción de expresiones (o términos) del Cálculo-λ a la siguiente **regla sintáctica**:

Sintaxis bajo aplicaciones unarias

$$e =_s v \mid c \mid (e_0\ e_1) \mid \lambda v.e_0$$

Una expresión-λ es una variable, denotada por v; o una constante, denotada por c; o la aplicación de una expresión e_0 a una expresión e_1; o la abstracción de una variable v de una expresión e_0. Las expresiones que se forman a partir de la aplicación sistemática de estas reglas se conocen como **fórmulas bien formadas** (fbf) del Cálculo-λ. Cualquier otra expresión no es válida en el Cálculo-λ.

fbf

Una aplicación $(e_0\ e_1)$ representa el resultado de aplicar e_0 a e_1. Se dice que e_0 es la expresión en la posición del **operador**; y que e_1 es la expresión en la posición del **operando** de la aplicación. Es común que e_0 y e_1 se identifiquen como la función y el argumento de la aplicación, lo cual no es totalmente correcto, puesto que la sintaxis del Cálculo-λ permite que cualquier expresión-λ esté en cualquiera de las dos posiciones y sólo las abstracciones y las operaciones primitivas $+, -, *, \ldots$ son funciones verdaderas.

Operador

Operando

Nuestro principal interés está en las aplicaciones de la forma $(\lambda v.e_0\ e_1)$ que tienen una abstracción como operador y una expresión-λ como operando. Estas expresiones involucran toda la historia acerca del papel de las variables en la evaluación de los algoritmos, el concepto de alcance y el de substitución de variables. De hecho,

nos basta con considerar el Cálculo-λ **puro**, cuyo conjunto de constantes está vacío. Sin operadores primitivos de por medio, las abstracciones son las únicas funciones en juego. A pesar de esta simplificación, tenemos un modelo formal completo que provee los fundamentos necesarios para razonar acerca de la **computabilidad algorítmica**. Si se desea, podemos incluso representar números, valores de verdad y listas, entre otras cosas, como abstracciones-λ; aunque éstas luzcan extrañas, sin parecido con su habitual representación. Pasemos ahora a la semántica del Cálculo-λ.

Cálculo-λ puro

Computabilidad

5.2.2 Semántica

La belleza del Cálculo-λ puro reside en que sólo necesitamos preocuparnos por una sola regla de transformación, puesto que sólo contamos con aplicaciones de abstracciones. La regla, conocida como β-**reducción**, se define como sigue:

β-reducción

$$(\lambda v.e_b\ e_a) \rightarrow_\beta\ e_b[v \leftarrow e_a] \tag{5.2}$$

Se dice que la regla reduce el β-**redex** $(\lambda v.e_b\ e_a)$ a su **reductum**, o contractum, $e_b[v \leftarrow e_a]$.

β-redex

Reductum

Desafortunadamente esta regla no es tan simple como parece a primera vista. Como sabemos, existen problemas con respecto a las variables ligadas en las abstracciones y la existencia de variables libres con el mismo nombre; en cuyo caso, las substituciones no pueden llevarse a cabo sin una acción correctiva en una de las variables. Esto concierne al **estado de ligado** de las variables, que debe permanecer invariante contra las β-reducciones para garantizar el **determinismo** de los resultados, independientemente de la elección del nombre de las variables.

Estado de ligado

Determinismo

Si las substituciones se llevarán a cabo de manera *naïf*, es decir, con los operandos interpretados literalmente como son, sucederían cosas como las que se ejemplifican a continuación:

$$(\lambda u.\lambda v.u\ w) \rightarrow \lambda v.w \quad \text{y} \quad (\lambda u.\lambda v.u\ v) \rightarrow \lambda v.v$$

Observen que, a pesar de que se trata de la misma función aplicada a dos funciones de nombre diferente, la reducción nos arroja como resultado dos funciones completamente diferentes. En el primer caso obtendríamos una función constante, que independientemente del operando al que se aplique, siempre regresa w; y en el segundo caso, obtendríamos la función identidad, que siempre reproduce la expresión operando. Observen la aplicación de las funciones así obtenidas a la variable a.

$$(\lambda v.w\ a) \rightarrow w \quad \text{y} \quad (\lambda v.v\ a) \rightarrow a$$

El problema aparece en la aplicación de la derecha, donde el operando v es una variable libre en la abstracción $\lambda u.\lambda v.u$, que al ser substituida de manera *naïf*, queda ligada bajo el alcance del abstractor λv **parasitariamente**; mientras que en la primer aplicación substituimos la variable w que no es afectada por el abstractor λv y por lo tanto preserva su estado de ligado de variable libre.

Liga parasitaria

Podríamos decidir aceptar estas ligas parásitas, que de hecho son causadas por **conflicto entre nombres** e incluso sacar ventaja de ellas; desafortunadamente, tales ligas destruyen dos propiedades muy útiles e importantes del Cálculo-λ que, como veremos más adelante, garantizan un comportamiento ordenado con respecto a las estrategias de evaluación, y por tanto no deben abandonarse.

Conflicto entre nombres

Para prevenir los conflictos entre nombres, podemos optar por una estrategia segura y demandar que todas las variables dentro de una λ-expresión tengan **nombres únicos**. Sin embargo, tal estrategia no parece realista debido a razones prácticas. Al incrementarse el número de variables en algoritmos complejos, será incrementalmente más complicado inventar nombres de variables que reflejen su uso o

Nombres únicos

convención. Construir nombres únicos automáticamente, por ejemplo, por enumeración, puede ser una opción siempre y cuando los nombres nuevos recuerden a los originales de alguna manera.

Aquí exploraremos una solución que requiere de una definición precisa de lo que significa una variable libre y una acotada, así como el alcance de una variable. Con V denotando el conjunto de variables, definimos el **conjunto de variables libres** FV de una expresión e recorriendo las tres formas sintácticas del Cálculo-λ puro y especificando lo que son las variables libres por casos:

Variables libres

$$FV(e) = \begin{cases} \{v\} & \text{Si } e =_s v \in V \\ FV(e_0) \cup FV(e_1) & \text{Si } e =_s (e_0\ e_1) \\ FV(e_0) \setminus \{v\} & \text{Si } e =_s \lambda v.e_0 \end{cases} \tag{5.3}$$

Esta definición recursiva nos dice que el conjunto contiene solo la variable v si v es la expresión e; o que es la unión de las variables libres de un operador y su operando, si e es una aplicación; o, si e es una abstracción, las variables que aparecen en el cuerpo de ésta, pero no en el abstractor.

Es posible ofrecer una definición complementaria del **conjunto de variables acotadas** en la expresión e:

Variables acotadas

$$BV(e) = \begin{cases} \emptyset & \text{Si } e =_s v \in V \\ BV(e_0) \cup BV(e_1) & \text{Si } e =_s (e_0\ e_1) \\ BV(e_0) \cup \{v\} & \text{Si } e =_s \lambda v.e_0 \end{cases} \tag{5.4}$$

Con la ayuda de estas definiciones podemos expresar que una variable v está libre en una expresión e, si y sólo si $v \in FV(e)$; y que una variable v está acotada en una expresión e, si y sólo si $v \in BV(e)$. En una abstracción $\lambda v.e$, se dice que el cuerpo e es el **alcance** del abstractor λv, lo que significa que todas las ocurrencias libres de v en e están acotadas por λv. Una misma variable puede estar libre o acotada de pendiendo del alcance considerado.

Alcance

Ejemplo 5.4. *Por ejemplo, en la expresión:*

$$\underbrace{(\lambda u.\,(\lambda v.\,(\lambda z.\,\underbrace{\underbrace{(z\ (v\ u))}_{\text{alcance de } \lambda z}\ v)}_{\text{alcance de } \lambda v}\ u)}_{\text{alcance de } \lambda u}\ w)$$

la variable w está libre, puesto que no existe abstractor para ella. La variable u está acotada en la abstracción $\lambda u.(\ldots)$, pero libre en su cuerpo, que es el alcance del abstractor λu.

Una abstracción sin variables libres, se dice **cerrada**; en otro caso se dice que la abstracción es **abierta**. Las abstracciones cerradas también se conocen como **combinadores**. De gran relevancia para la implementación de lenguajes de programación basados en el Cálculo-λ, son los llamados **super combinadores**, que son abstracciones cerradas cuyos cuerpos pueden contener recursivamente sólo expresiones cerradas (o super combinadores).

Abstracciones cerradas y abiertas

Combinador

Ahora estamos listos para definir de manera precisa, como la **substitución** de variables en la β-reducción (Ecuación 5.2) , se lleva a cabo: El reductum debe prohibir la substitución de todas las ocurrencias libres de la variable v en la expresión e_b por la expresión e_a. Su definición es la siguiente:

Substitución

$$e_b[v \leftarrow e_a] = \begin{cases} e_a & \text{Si } e_b =_s v \in V \\ u & \text{Si } e_b =_s u \in V \wedge v \neq_s u \\ (e_0[v \leftarrow e_a]\ e_1[v \leftarrow e_a]) & \text{Si } e_b =_s (e_0\ e_1) \\ \lambda v.e_0 & \text{Si } e_b =_s \lambda v.e_0 \\ \lambda u.e_0[v \leftarrow e_a] & \text{Si } e_b =_s \lambda u.e_0 \quad \wedge \\ & u \notin FV(e_a) \vee v \notin FV(e_b) \\ \lambda w.e_0[u \leftarrow w][v \leftarrow e_a] & \text{Si } e_b =_s \lambda u.e_0 \quad \wedge \\ & u \in FV(e_a) \wedge v \in FV(e_b) \quad \wedge \\ & w \in V \wedge w \notin FV(e_a) \cup FV(e_b) \end{cases} \tag{5.5}$$

Los últimos tres casos son muy interesantes, establecen que debe hacerse cuando e_b es una abstracción:

- Si la abstracción e_b liga a v, entonces no hay ocurrencias libres de v en ella; y la substitución regresa e_b sin cambios.
- Si la abstracción e_b liga a otra variable, p. ej., u, entonces v puede sustituirse de manera naïf por e_a, si no hay ocurrencias libres de u en e_a; o si tenemos el caso trivial de que v no ocurre en e_b.
- Si la abstracción e_b liga a u y v ocurre libre en su cuerpo e_0, mientras que u ocurre libre en e_a, tenemos un conflicto entre nombres: Si e_a fuera substituida de manera naïf en e_0, las ocurrencias libres de u en e_a sería ligadas parasitariamente por el abstractor λu y cambiaría su estado de ligadura.

Estrategias de cambio de nombre

Existen dos **soluciones** para un conflicto entre nombres. Podemos cambiar la variable libre v en e_a, p. ej., por w; o cambiar la variable ligada u, p. ej., por w en la abstracción. En ambos casos w debe ser una variable nueva que no haya sido usada en el β-redex original. La solución tradicional es la última, que es más conveniente, porque el cambio de nombre solo concierne a las variables que ocurren en el alcance de la aplicación.

α-conversión

La transformación que **renombra** se conoce como α-conversión y se define como sigue:

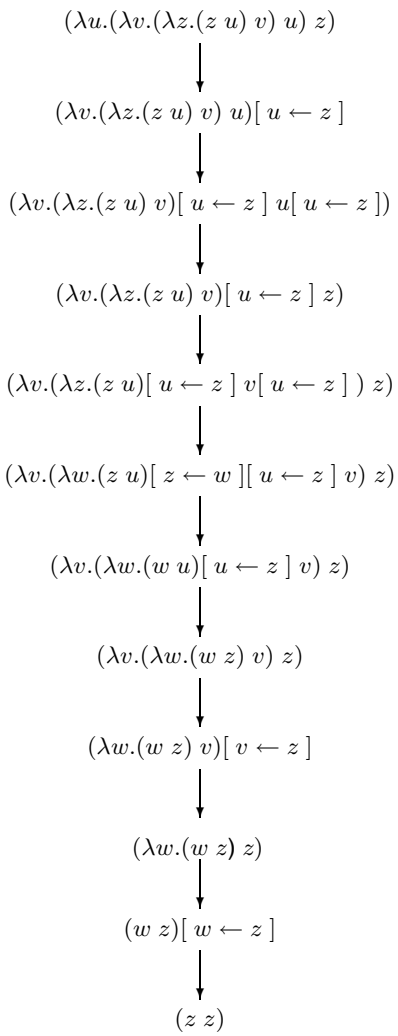
$$\lambda u.e_0 \rightarrow_\alpha \lambda w.e_0[u \leftarrow w] \tag{5.6}$$

Se lleva a cabo mediante la aplicación de una función de α-conversión a la abstracción cuyas variables ligadas necesitamos cambiar:

$$(\lambda v.\lambda w.(v\ w)\ \lambda u.e_0)$$

Esta transformación procede con la aplicación de las reglas definidas anteriormente en dos pasos: primero a $\lambda w.(\lambda u.e_0\ w)$ y después a $\lambda w.e_0[u \leftarrow w]$.

Ejemplo 5.5. *Una secuencia de β-reducciones y α-conversiones:*



En el ejemplo remplazamos la variable libre w por z para evitar un conflicto entre nombres cuando los β-radices son sistemáticamente reducidos de afuera hacía adentro. En esta secuencia, la última regla de substitución es aplicada en la sexta expresión de arriba hacía abajo para renombrar las variable ligada z en la abstracción $\lambda z.(z\ u))$ como w para evitar el conflicto entre nombres con la variable z que debe ser substituida por u. Pero esto es justo lo que queríamos evitar, que una variable nueva que no estaba en la expresión original apareciera., cambiando la apariencia pero no el significado de la abstracción $\lambda z.(z\ u)$ a $\lambda w.(w\ u)$. En este caso tuvimos suerte, si sólo miramos las expresiones inicial y final, ignorando los pasos intermedios, el renombrado de variables pasa desapercibido puesto que la forma normal es una aplicación de la variable original z a sí misma, tal que preserva su estado de ligadura como variable libre en toda la secuencia de pasos de reducción.

La historia es diferente para λ-expresiones que reducen abstracciones, como:

$$(\lambda u.(\lambda v.\lambda z.((z\ u)\ v)\ z)\ v))$$

Al ejecutar las β-reducciones, de afuera hacía adentro, encontramos dos conflictos entre nombres, el primero al substituir el argumento externo v bajo λv; y el segundo al substituir el argumento interno z bajo λz. Si renombramos v por x y luego z por y, obtenemos la abstracción $\lambda y.((y\ v)\ z)$ como resultado. Sin embargo, si invertimos el orden en el que usamos x e y, obtendríamos la abstracción $\lambda x.((x\ v)\ z)$ que es igual a la anterior, excepto que el nombre de la variable ligada ha cambiado.

Aunque la elección del nombre para las variables ligadas es totalmente irrelevante, ejecutar muchas β-reducciones que requieren cambio de nombre en un contexto

mayor debe ser confuso y alienante con respecto a la expresión original, más allá de la expresión resultante.

Para evitar este problema de renombrar variables debemos recurrir a una idea más inteligente para representar el estado de liga de las variables, así como para lograr que la aplicación de las β-reducciones preserve el nombre de las variables introducidas en la expresión inicial, bajo toda circunstancia.

5.2.3 Un esquema de indexación para variables ligadas

Al especificar una abstracción, la elección de los nombres de las variables ligadas no es importante. Su única función es relacionar a los abstractores con posiciones sintácticas en el cuerpo de la abstracción. Funcionan como receptáculos donde los argumentos necesitan ser substituidos. A esta relación se le conoce como **estructura de ligado**.

Estructura de ligado

Ejemplo 5.6. *Las dos abstracciones que se muestran a continuación son sintácticamente equivalentes módulo α-conversión de los nombres de las variables:*

$$\lambda u.\lambda v.\lambda w.(((w\ v)\ u)(x\ u)) =_s \lambda x_1.\lambda x_2.\lambda x_3.(((x_3\ x_2)\ x_1)(x\ x_1))$$

En casos como el anterior, cuando aplicamos las abstracciones a los mismos argumentos, obtenemos en ambos casos el mismo resultado, puesto que dos expresiones que son sintácticamente equivalentes, también lo son semánticamente.

La posición de un abstractor en una secuencia de abstractores también identifica, en el caso de aplicaciones anidadas, el nivel de anidamiento en que el argumento será tomado. A esto se le llama **estructura de substitución**. La estructura resultante, por una parte asocia abstractores con la ocurrencia de variables en el cuerpo de la abstracción; y por la otra con las posiciones de los operandos en aplicaciones anidadas. Esto se ilustra en la figura 5.1.

Estructura de substitución

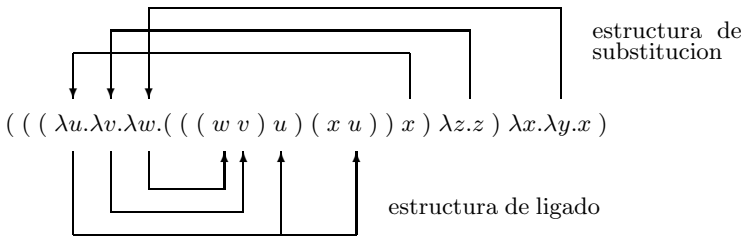


Figura 5.1: Estructuras de substitución y ligado.

La figura 5.2 muestra la secuencia de β-reducciones que evalúan esas aplicaciones anidadas. Necesitaremos esa secuencia para efectos de comparación más adelante. La secuencia se obtiene aplicando las β-reducciones de adentro hacía afuera: x por u, $\lambda z.z$ por v y $\lambda x.\lambda y.x$ por w; y entonces se procede a evaluar el cuerpo instanciado, regresando el resultado de la aplicación $(x\ x)$ de la variable x libre a si misma.

Ahora procederemos a llevar el renombrado de variables un paso más adelante, llamándolas a todas x y distinguiéndolas por sus subíndices. A todas las variables se les dará el mismo nombre, por ejemplo z, y la estructura de ligado se definirá por medio de los llamados operadores de desligue ó **llaves protectoras** que se colocan frente a la ocurrencia de las variables en el cuerpo de la abstracción. Estas llaves protectoras complementan a los abstractores, en un sentido amplio, deshacen ligaduras: Si una ocurrencia de la variable z está precedida por n de estos operadores, denotados como:

Llaves protectoras

$$\underbrace{//\ldots/}_{n}\ z$$

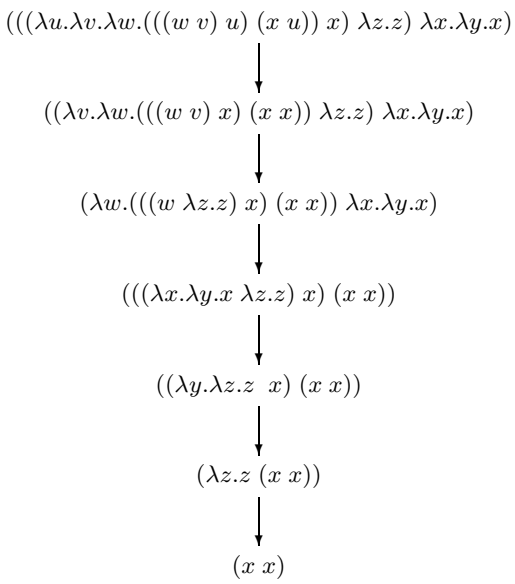


Figura 5.2: Reducción de la expresión ejemplo

entonces la ocurrencia está protegida contra las n imbricaciones más internas del abstractor λz que liga a la variable. Usando está notación, la abstracción del ejemplo puede representarse como sigue:

$$\lambda u.\lambda v.\lambda w.(((w\ v)\ u)(x\ u)) =_s \lambda z.\lambda z.\lambda z.(((z\ /z)\ //z)\ (x\ //\ z))$$

Todas las variables ligadas se llaman ahora z y, p. ej., las ocurrencias de z que remplazan a la variable original u, están precedidas por dos llaves de protección $//$, de forma que los dos abstractores λz más internos, no le afecten. La variable x no se modifica porque aparece libre en la expresión.

El problema con las llaves de protección es que necesitan ser modificadas dinámicamente, conforme se aplican las β-reducciones. Cada que un abstractor desaparece de la abstracción original, una llave de protección debe desaparecer también. Pero también, si una variable libre entra en el alcance de un abstractor, y esta variable tiene el mismo nombre que la variable abstraída, entonces en concordancia, debemos introducir una llave de protección más. Ejemplificaremos esto aplicando nuestra abstracción ejemplo a tres abstracciones cuyas variables también se llaman z. Para hacer el ejemplo más interesante agregaremos dos abstractores al frente de la abstracción, de forma que el más externo liga ahora a lo que era la variable libre x (que ahora aparece como z con cuatro llaves protectoras, que se corresponden con los tres abstractores que ya existían y el abstractor interno de los dos que agregamos). La β-reducción de esta aplicación se muestra en la figura 5.3. Esta reducción se lleva a cabo en el mismo número de pasos que la anterior, y produce las mismas expresiones intermedias, solo que con diferente representación.

La primera de las β-reducciones en la secuencia, incluye todo lo que debemos saber acerca del procesamiento de las llaves protectoras durante la ejecución de las reducciones. El β-redex bajo consideración está subrayado en la figura 5.3 y su argumento es $/z$. Al hacer esto, elimina el primer λz y substituye $/z$ por todas las ocurrencias de variables ligadas en el cuerpo de la abstracción (las dos ocurrencias de $//z$). Ahora, al substituir $/z$ bajo el alcance de los dos abstractores restantes, debemos añadirle dos llaves protectoras, esto es, las dos ocurrencias de $//z$ deben ser substituidas por $///z$. para mantener la distancia correcta con el λz más externo. Pero aún hay más, la ocurrencia de la variable $////z$ que está dentro de la abstracción se verá afectada por el λz que ha desparecido, de forma que se le debe

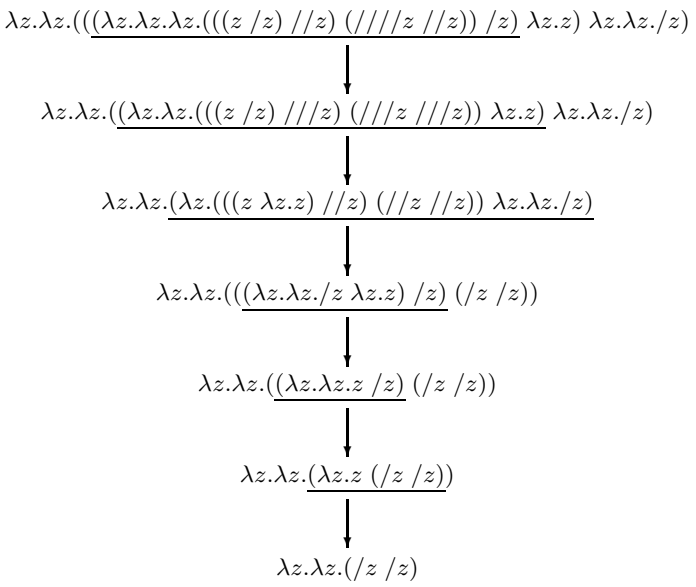


Figura 5.3: Reducción con variables de nombre único y llaves protectoras.

quitar una llave protectora, lo que da lugar a tres ocurrencias de $///z$ ligadas al abstractor más externo.

Ahora podemos ser más específicos sobre la manipulación de las llaves protectoras y para ello debemos definir el **estado de ligadura** de las variables que preceden. Sea *Estado de ligadura*

$$\underbrace{/\ldots//}_{i}\ v =_s /(i) \mid i \in \{0, 1, \ldots\}$$

la ocurrencia de una variable v con i llaves de protección dentro de la λ-expresión e, y sea que $j \in \mathbb{N}_0$ enumere, de la ocurrencia de la variable hacía afuera y comenzando en $j = 0$, los abstractores $\lambda v.$ que la rodean. Entonces, en relación con el abstractor j-ésimo, esta ocurrencia de la variable se dice:

$$\rightarrow \textit{libre} \text{ Si } i > j, \quad \rightarrow \textit{acotada} \text{ Si } i = j, \quad \rightarrow \textit{sombreada} \text{ Si } i < j$$

El índice de protección i de la ocurrencia de una variable, denota lo que se conoce como **índice de ligado** ó distancia de ligado (con respecto al abstractor). Con estos atributos de las ligaduras, podemos definir informalmente la regla de β-reducción en estos términos: *Indice de ligado*

Dado un redex $(\lambda v.e_b\ e_a)$, su β-reducción regresa una expresión e'_b que se obtiene de e_b y una ocurrencia de $/^{(i)}v$ en:

- El cuerpo de la abstracción e_b:
  - Decrementar el número i de llaves de protección, si aún quedan disponibles,
  - Se substituye por e_a, si la variable está acotada,
  - No se cambia nada, si la variable está sombreada.
- El operando e_a:
  - Incrementar el número i de llaves de protección en un valor k si la variable es libre o acotada con respecto al λv más interno que rodea la aplicación $(\lambda v.e_b\ e_a)$; y si la variable penetra el alcance de k abstractores anidados λv cuando es substituida en e_b,
  - Permanece intacta si la variable está sombreada.

Conversión a variables únicas

Se puede **transformar** una expresión con variables acotadas de distinto nombre, en una que tenga solo variables, digamos z, aplicando la función de α-conversión $\lambda v.\lambda z.(v\ z)$ a todas las abstracciones de una variable. En el caso del ejemplo, la función se inserta en la expresión original:

$$\lambda v.\lambda z.(v\ z)\lambda u.\ldots$$

Aplicando estas α-conversiones de adentro hacía afuera y usando la regla de β-reducción definida informalmente antes, se obtiene la secuencia de reducciones que se muestra en la figura 5.4. Esta secuencia termina con una abstracción en la que todas las ocurrencias de las variables ligadas se sustituyeron por z y el número adecuado de llaves de protección ha sido introducido. La variable x no cambia por que no hay abstractor asociado a ella en toda la expresión.

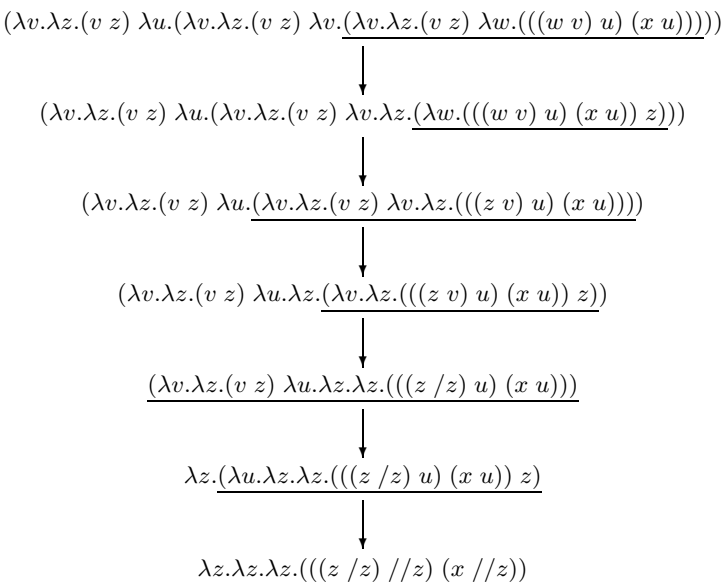


Figura 5.4: Reducción a nombres únicos y llaves de protección adecuadas.

Variables sin nombres

Evidentemente, si seguimos en esta línea de procesamiento de variables acotadas y su estructura de ligadura, el siguiente paso es un Cálculo-λ **sin nombres de variables** ([45], pp. 63–68) donde los abstractores están asociados a índices y la semántica de la β-reducción se basa en la manipulación de esos índices, tal y como lo definimos aquí informalmente. Por cuestiones de tiempo, se deja al lector revisar el material asociado al Cálculo-λ sin nombres. La ventaja de este enfoque sobre el Cálculo-λ con variables nombradas es que, aunque para los humanos la reducción de las expresiones sea menos clara, su reducción automática es mucho más sencilla de implementar.

5.2.4 Propiedades de las secuencias de reducción

β-reducible

Revisemos ahora, algunas de las propiedades generales de las secuencias de reducción que hemos computado. Dadas dos expresiones-λ, e y e', se dice que e es **β-reducible** a e', denotado por $e \longmapsto_\beta e'$, si y sólo si e puede ser transformada en e' por una secuencia finita (posiblemente vacía) de reducciones-β y conversiones-α. Esto produce una secuencia $e_0, \ldots, e_n$ de expresiones-λ con $e_0 =_s e$ y $e_n =_s e'$ tal que para todos los índices $i \in \{0, \ldots, n-1\}$ tenemos que $e_i \to_\beta e_{i+1}$ ó $e_i \to_\alpha e_{i+1}$.

Equivalencia semántica

Con base en tales secuencias podemos definir que dos λ-expresiones son **semánticamente equivalentes**, denotado por $e = e'$, si y sólo si e puede ser transformada en e' por una secuencia finita (posiblemente vacía) de reducciones-β, reducciones-β

reversas y conversiones-α. Esto es, para todos los índices $i \in \{0, \ldots, n-1\}$, debemos tener $e_i \to_\beta e_{i+1}$ ó $e_{i+1} \to_\beta e_i$ ó $e_i \to_\alpha e_{i+1}$.

El objetivo de reducir una λ-expresión e es transformarla en alguna expresión e^{NF} que no contiene más redices, ó a la que no se le puedan aplicar más reglas de β-reducción. Esta expresión se conoce como la **forma normal** ó el valor de la expresión e. Si para llegar de e a e^{NF} necesitamos una secuencia finita de reducciones-β, entonces e^{NF} es también la forma normal de las expresiones-λ intermedias.

Forma normal

Una λ-expresión compleja que contiene diversos redices, generalmente lleva a una elección entre diferentes secuencias alternativas de reducciones-β. El problema con estas elecciones es que, iniciando de una expresión inicial, debemos alcanzar una forma normal, bajo las siguientes condiciones posibles:

- eventualmente para todas las posibles secuencias, si tenemos suerte;
- para ninguna de las secuencias posibles, porque no existe una forma normal. Por ejemplo, ninguna de las secuencias termina en un número finito de pasos;
- para algunas de las secuencias posibles, pero no para todas. Esto es, algunas secuencias terminan de manera finita, pero otras no.

El último caso es muy interesante porque demanda una **estrategia** que asegure que las reducciones serán aplicadas en un orden que lleve a una forma normal, si es que ésta existe.

Estrategias de reducción

Ejemplo 5.7. *Consideremos un ejemplo de la primer clase de expresiones:*

$$(\lambda u.(\lambda w.(\lambda w.u)\ u)\ w))$$

donde:

- *procediendo de afuera hacía adentro:*

$$(\lambda u.(\lambda w.(\lambda w.u)\ u)\ w)) \to_\beta (\lambda w.(\lambda(w.//w\ /w)w) \to_\beta (\lambda w./w\ w) \to_\beta w$$

- *procediendo de adentro hacía afuera:*

$$(\lambda u.(\lambda w.(\lambda w.u)\ u)\ w)) \to_\beta (\lambda u.(\lambda w.u\ u)\ w) \to_\beta (\lambda u.u\ w) \to_\beta w$$

- *y aún si comenzásemos por el radice de en enmedio, llegaríamos a la forma normal w.*

Ejemplo 5.8. *Ahora, una expresión simple que no tiene forma normal es la **auto-aplicación**:*

Auto-aplicación

$$(\lambda u.(u\ u)\ \lambda u.(u\ u)) \to_\beta (\lambda u.(u\ u)\ \lambda u.(u\ u)) \to_\beta \ldots$$

que incesantemente se reproduce a si misma.

Ejemplo 5.9. *La auto-aplicación servirá para definir una expresión simple cuya reducción, dependiendo del orden en que los redices son contraídos, puede terminar o no. Observen la siguiente aplicación:*

$$((\lambda w.\lambda v.u\ \lambda w.w) \quad (\lambda u.(u\ u)\ \lambda u.(u\ u)))$$

*identificamos de manera inmediata que la abstracción $\lambda w.\lambda v.u$ es una **función selector** que reproduce su primer argumento, es decir λww, pero elimina el segundo, que en este caso particular es la misma aplicación. De forma que una reducción de afuera hacía adentro lleva a w, pero una de afuera hacía adentro no termina.*

Función selector

El problema con este tipo de secuencias que no terminan aunque una forma normal exista, es que tratan de reducir sub-expresiones cuya forma normal no contribuye a llegar a la forma normal de la expresión completa. En el mejor de los casos, esto atenta contra la eficiencia al ejecutar computaciones superfluas; en el peor de

los casos puede llevarnos a casos de **no terminación**. Es por ello que necesitamos garantizar la terminación con formas normales si estas existen.

No terminación

La estrategia que garantiza esto se llama **reducción en orden normal**. La idea es aplicar las abstracciones a operandos no evaluados y forzar su reducción si y sólo si hay formas normales diferentes a abstracciones, p. ej., variables o aplicaciones que no pueden β-reducirse, en la posición del operador de las aplicaciones.

Reducción en orden normal

Esta estrategia debe ser definida por una **función de transformación** τ_N que mapea λ-expresiones a λ-expresiones como sigue:

Función de transformación

$$\tau_N(e) = \begin{cases} v & \text{Si } e =_s v \in V \\ \lambda v.\tau_N(e_b) & \text{Si } e =_s \lambda v.e_b \\ \tau'_N(e) & \text{Si } e =_s (\lambda v.e_b\ e_2) \\ \tau'_N(\tau_N(e_1)e_2) & \text{Si } e =_s (e_1\ e_2) \text{ y } e_1 \neq_s \lambda v.e_b \end{cases}$$

donde:

$$\tau'_N(e) = \begin{cases} \tau_N(e_b[v \leftarrow e_2]) & \text{Si } e =_s (\lambda v.e_b\ e_2) \\ (e_1\ \tau_N(e_2)) & \text{Si } e =_s (e_1\ e_2) \text{ y } e_1 \neq_s \lambda v.e_b \end{cases}$$

La función τ_N define un **evaluador abstracto** para expresiones del Cálculo-λ puro, similar al evaluador `EVAL` de la sección anterior. A diferencia de `EVAL`, τ_N solo se propaga recursivamente a través de las expresiones operador, pero no toca los operandos hasta que el operador es procesado y no es una abstracción (el último caso en τ'_N. Si es una abstracción, entonces el operando es substituido por ocurrencias libres de la variable ligada (el primer caso de τ'_N).

Evaluador abstracto

La reducción en orden normal también se conoce como de afuera a adentro y de izquierda a derecha, o como **llamada por nombre**, y es usada por lenguajes de programación como Algol y Simula. Es posible definir una estrategias como la usada por `EVAL`, conocida como primero operandos, o **llamada por valor**. Intenten definir una función τ_A que implemente la llamada por valor.

Llamadas por nombre

Llamada por valor

La propiedad más importante de las secuencias de β-reducciones es capturada por el conocido **Teorema de Church-Rosser**. Este teorema expresa esencialmente que independientemente del orden en que las reducciones-β son llevadas a cabo sobre una expresión-λ, existe siempre una expresión-λ en la cual dos diferentes secuencias de reducciones-β pueden volverse a encontrar. Esto se puede formalizar como sigue:

Teorema de Church-Rosser

Sean e_0, e_1, e_2, e_3 λ-expresiones. Entonces $e_0 \longmapsto_\beta e_1$ y $e_0 \longmapsto_\beta e_2$ implican que existe una expresión e_3 tal que $e_1 \longmapsto_\beta e_3$ y $e_2 \longmapsto_\beta e_3$. La prueba está más allá del contenido de este curso, pero observen que si e_3 es una forma normal, entonces esta forma es única. Dicho de otra manera, la forma normal de una expresión-λ es **invariante** con respecto a las diversas secuencias de reducción que se pueden seguir para alcanzarla.

Determinismo

Dejando de lado el problema de conflicto entre nombres, la reducción-β es una operación muy simple: lleva a cabo substituciones libres de contexto entre iguales; es decir, remplaza una aplicación por otra, sin causar ningún efecto colateral en la expresión donde se encuentra (su contexto). A esta propiedad se le llama **transparencia referencial**. Además, la sintaxis del Cálculo-λ asegura que los redices-β no se traslapen: Pueden estar completamente anidados unos en otros, o ser completamente independientes, pero no comparten componentes. De ahí el determinismo de las formas normales, dejando de lado los problemas de terminación. Lo mismo aplica para las variaciones introducidas, p. ej., variables únicas, indices de ligado, variables sin nombre.

Transparencia referencial

Además de las formas normales, que son la meta última del proceso de reducción-β en el cálculo-λ puro, hay dos variantes **intermedias** de forma normal. Para distinguirlas diremos que una λ-expresión es una:

Formas normales

- **forma normal completa**, si no contiene redices-β. El cálculo-λ que computa formas normales completas, se conoce como normalizador completo, o simplemente normalizador;
- **forma normal débil**, si es una abstracción de alto nivel (puede contener redices en su cuerpo) ó una aplicación de alto nivel de una abstracción n-aria a un conjunto de operandos con aridad menor a n, que están en forma débil. El cálculo-λ que computa solo formas débiles se conoce como normalizador débil; y
- **forma normal de cabeza**, si es una forma especial de abstracción de alto nivel

  $$\lambda u_1 \ldots . \lambda u_n . (\ldots (u_i\ e_1) \ldots e_m)$$

- donde $i \in \{0, \ldots, n-1\}$, cuya forma no puede cambiar más hacía la izquierda de la variable u_i ya que solo es posible llevar a cabo β-reducciones en las expresiones operando $e_1, \ldots, e_m$. El cálculo-λ que computa formas normales de cabeza se dice normalizador de cabezas.

Las tres formas están relacionadas de la siguiente manera: toda forma normal completa, es también una forma normal de cabeza, y toda forma normal de cabeza es también una forma normal débil, pero no a la inversa; es decir, forman una jerarquía. La normalización completa y la de cabeza, requieren de normalizadores completos puesto que necesitarán substituciones y reducciones bajo los abstractores, lo cual puede ocasionar conflictos entre nombres. Las substituciones naïf, son suficientes para la normalización débil puesto que solo se permiten reducciones de alto nivel que evitan los conflictos entre nombres.

5.2.5 Recursividad

Antes de terminar, una nota final sobre la recursividad. Puesto que el cálculo-λ no tiene forma de definir ecuaciones, debemos encontrar otra manera de representar el concepto de reproducir expresiones en ellas mismas. Podríamos intentar lograr esto vía la auto-aplicación, de la que conocemos el caso especial $(\lambda u.(u\ u)\ \lambda u.(u\ u))$. Mejor aún, deberíamos buscar un **Operador de recursividad** universal, digamos s, tal que: $(s\ f) = (f(s\ f))$. Observen que si invertimos las expresiones a ambos lados de la igualdad, es fácil ver que $(s\ f)$ es un punto fijo de f: es una expresión que f mapea a si misma. Por lo que necesitamos un **combinador de punto fijo**, p. ej.:

Operador de recursividad

Combinador-Y

$$Y =_s (p\ p) \tag{5.7}$$

donde $p =_s \lambda u.\lambda v.(v((u\ u)\ v))$.

Ejemplo 5.10. *Probemos un ejemplo para observar el comportamiento de Y:*

$$\begin{gathered}(Y\ f) =_s (\lambda u.\lambda v.(v((u\ u)\ v))p)\ f) \rightarrow_\beta \\ (\lambda v.(v\ (p\ p)\ v)\ f) \rightarrow_\beta \\ (f\ ((p\ p)\ f) =_s (f\ Y(f))\end{gathered}$$

El combinador-Y extingue la auto-aplicación de funciones individuales, y efectua llamadas recursivas uniformemente sobre todas las abstracciones.

Ejemplo 5.11. *Consideren el siguiente ejemplo de una ecuación para una función recursiva:*

$$f = \lambda u.\lambda v.(((f\ u)\ v)((f\ v)\ u))$$

Podemos escribir la parte derecha de la ecuación como una aplicación semanticamente equivalente:

$$f = (\lambda z.\underbrace{\lambda u.\lambda v.(((z\ u)\ v)((z\ v)\ u))}_{e_b}\ f)$$

Y usando e_b como una abreviatura del cuerpo de la abstracción, como:

$$f = (\lambda z.e_b f)$$

Para darle a esta ecuación la forma de $(Y\ f) = (f(Yf))$, la solución más obvia es:

$$f = (Y\ \lambda z.e_B$$

de forma que:

$$(Y\ \lambda z.e_b) = (\lambda z.e_b\ (Y\ \lambda z.e_b))$$

De manera que, podemos tomar una ecuación que define una función recursiva y convertirla en una expresión-λ si 1) abstraemos las ocurrencias del identificador de la función en el lado derecho de la ecuación; y 2) le aplicamos a esta nueva abstracción el combinador-Y. Desafortunadamente, si seguimos la estrategia de llamada por nombre, el combinador-Y se propaga innecesariamente en la posición del operador. Si cambiamos a una estrategia de orden normal, el asunto queda solucionado. También podemos diseñar un operador dedicado, como una extensión del Cálculo-λ, que ligue recursivamente las ocurrencias de z en e_b por copias de la expresión entera tal cual. Ambas aproximaciones pueden usarse para definir funciones mutuamente recursivas, como en la forma sintáctica `letrec` del lenguaje `AL`.

5.3 LECTURAS Y EJERCICIOS SUGERIDOS

Church [13] introduce por vez primera el Cálculo-λ al presentar un conjunto de postulados para fundamentar la lógica. Posteriormente [15], lo vuelve a utilizar en su artículo sobre un problema irresoluble de la teoría de números elemental. La presentación más completa está en su libro "El Cálculo de Lambda-Conversión" [16]. La tesis de Church-Turing, en realidad fue introducida por Kleene [44].

La presentación del Cálculo-λ en este capítulo está basada en el libro de Kluge [45]. Por ello, hemos comenzado por introducir un lenguaje abstracto (AL) que se asemeja a Lisp, de forma que sea más fácil ver la relación entre el Cálculo-λ y un lenguaje aplicativo. Otra buena introducción es la propuesta por Barendregt y Barendsen [3]. Hudak [38] introduce el Cálculo-λ en el contexto de la Programación Funcional y revisa la evolución de los lenguajes de programación que se han desarrollado bajo este paradigma.

Ejercicios

Ejercicio 5.1. *Implemente la función longitud de una lista en `AL`.*

Ejercicio 5.2. *Defina una función τ_A que implemente la llamada por valor como estrategia de reducción en el Cálculo-λ puro.*

Ejercicio 5.3. *Defina el concepto de combinador, proponga un ejemplo y explique su relevancia en el Cálculo-λ.*

Ejercicio 5.4. *Explique brevemente en qué consiste la tesis Church-Turing.*