

Capítulo 9

El Cálculo- λ

Resumen El Cálculo- λ es el modelo más cercano a los **algoritmos** y su **evaluación** tal y como fueron formalizados en el capítulo 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**, **alcance de variables**, y la **substitución** ordenada de variables por expresiones. Se trata de un **lenguaje cerrado** en el sentido de que su **semántica** puede definirse en base a equivalencia entre expresiones (o términos) del cálculo mismo.

9.1. Introducción

El Cálculo- λ , publicado por Alonzo Church en 1932, es uno de los modelos matemáticos que se desarrollaron 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 **algorítmicamente computable**.

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. Aunque no se tenga una idea muy clara de lo que es la computabilidad, una nueva reconfortante es que todos estos modelos son equivalentes. Lo cual nos lleva a la **tesis de Church-Turing**: los problemas intuitivamente o efectivamente computables son exactamente aquellos que pueden computarse en un máquina de Turing (y por lo tanto, en los demás modelos también).

El Cálculo- λ es el modelo más cercano a los **algoritmos** y su **evaluación** tal y como fueron formalizados en el capítulo 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**, **alcance de variables**, y la **substitución** ordenada de variables por expresiones. Se trata de un **lenguaje cerrado** en el sentido de que su **semántica** puede definirse en base a equivalencia entre expresiones (o términos) del cálculo mismo.

9.2. Notación del Cálculo- λ

Una función f de n variables v_1, \dots, v_n se denotan en el Cálculo- λ como:

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

Sustituyendo λ por `lambda` e introduciendo `in` en lugar del punto, obtendríamos la notación empleada en las abstracciones de AL en el capítulo anterior. El símbolo f en el lado izquierdo de la ecuación denota el **nombre** o identificador de la función y puede ser referenciado en cualquier parte. La expresión del lado derecho de la ecuación define una **abstracción** de las variables v_1, \dots, v_n en el cuerpo de la función e_0 . Una definición precisa de lo que **ocurrencia libre de variables** significa, deberemos posponerla para más adelante. Por ahora basta comentar que una variable u es **libre** en la expresión e_0 si ésta no contiene ninguna abstracción de u (la variable no está en la lista $\lambda \dots$).

Una **aplicación** llamada f sobre r expresiones argumento e_1, \dots, e_r tiene la forma:

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

donde r no es necesariamente igual a n .

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

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

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

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

Usando la notación currificada, la aplicación de f a r operandos toma la forma de un r -fold de aplicaciones anidadas:

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

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

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

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 , respectivamente. 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 .

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 válida esté en cualquiera de las dos posiciones y sólo las abstracciones y las operaciones primitivas $+$, $-$, $*$, \dots son funciones verdaderas.

Estamos interesados particularmente en aplicaciones de la forma $(\lambda v. e_0 e_1)$ que tienen una abstracción en la posición del operador y una expresión válida en la posición del operando. Estas son las expresiones relacionadas con la historia completa sobre el papel de las variables, particularmente su alcance y sustitución, en la evaluación de expresiones algorítmicas. De hecho, en ese contexto, basta considerar únicamente el **Cálculo- λ puro**, cuyo conjunto de constantes está vacío. Sin operadores primitivos, 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, listas, entre otras cosas, como λ -abstracciones; aunque estas abstracciones lucen extrañas, sin parecido con su habitual representación.

9.3. β -reducción y α -conversión

La belleza del Cálculo- λ puro reside en que sólo necesitamos preocuparnos por una sola regla de transformación, puesto que sólo contamos con las aplicaciones de abstracciones en tal sistema. Como vimos en el capítulo anterior, segunda sección, esta regla reemplaza tales abstracciones por el cuerpo de la abstracción con las ocurrencias libres de las variables λ -ligadas, substituidas por la expresión correspondiente con el respectivo operando (o argumento). La regla se denota como:

$$\lambda v.e_0 e_1 \rightarrow_{\beta} e_0[v \leftarrow e_1]$$

La regla se conoce como **β -reducción** o **β -contracción** y se dice que **simplifica** o **reduce** el **β -redex** $(\lambda v.e_0 e_1)$ a su **reductum** o **contractum** $e_0[v \leftarrow e_1]$.

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 los mismos nombres, en cuyo caso, las **substituciones** no pueden llevarse a cabo sin una acción correctiva en una de las variables con el mismo nombre. Esto concierne al **estatus de ligadura** de las variables, que debe permanecer invariante contra las β -reducciones para garantizar la **determinación** de los resultados, independiente de la elección del nombre de las variables.

Si las substituciones se llevarán a cabo de manera *naif*, es decir, con los operandos literalmente como son, por ejemplo en:

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

obtendríamos como contractum la abstracción $\lambda v.w$ en el primer caso, y $\lambda v.v$ en el segundo caso. Es evidente que la elección de la variable en la posición del operando resulta en dos funciones totalmente diferentes. En el primer caso obtendríamos una **función constante**, que independientemente del operando al que se aplique, siempre regresa w . Sin embargo, en el segundo caso obtenemos la **función identidad** que siempre reproduce la expresión operando:

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

El problema aparece en el segundo caso donde la variable libre v es substituida de manera *naif* bajo el alcance del abstracto λv y por lo tanto deviene ligada a él parasitariamente; mientras que en el primer caso substituímos la variable w que no es afectada por el abstractor λv y por lo tanto preserva su estado de ligadura de variable libre.

Podríamos decidir aceptar estas **ligaduras parásitas**, que de hecho son causadas por **conflicto entre nombres**, como se discutió en el capítulo anterior; y posiblemente sacar ventaja de ellas. Desafortunadamente, tales ligaduras 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 fácilmente.

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 diferentes. 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 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:

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

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

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

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

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 $\lambda v.e$, llamamos al cuerpo e el **alcance** del abstractor λv , lo que significa que todas las ocurrencias libres de v en e están acotadas por λv . Por ejemplo, en la expresión:

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

la variable w está libre, puesto que no existe abstractor para ella. La variable u está acotada en la abstracción $\lambda u.(\dots)$, pero libre en su cuerpo, que es el alcance del abstractor λu . Una misma variable puede estar libre o acotada dependiendo del alcance considerado.

Una abstracción cuyo conjunto de variables libres está vacío, se dice **cerrado** o que es un **combinador**; en otro caso se dice que la abstracción es **abierto**. 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).

Ahora estamos listos para definir de manera precisa, como la **substitución** en la β -reducción

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

se lleva a cabo: el lado derecho de esta regla de transformación debe prescribir la sustitución de todas las ocurrencias libres de la variable v en la expresión e_b por la expresión e_a . Su definición es la siguiente:

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

Los últimos tres casos son muy interesantes, prescriben que debe hacerse cuando la expresión e_b , en la que las sustituciones de las ocurrencias libres de v serán llevadas a cabo, es una abstracción. Si esta abstracción liga a v , entonces no cambia puesto que no hay ocurrencias libres de v en ella. Si liga a otra variable u , entonces v puede sustituirse de manera naif 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 .

El caso complementario donde la abstracción liga a u , con v ocurriendo libremente en el cuerpo de la abstracción e_0 y u ocurriendo libre en e_a , causa conflicto entre nombres: cuando e_a fuera substituida de manera naif en e_0 , las ocurrencias libres de u en e_a sería ligadas parasitariamente por el abstractor λu y cambiaría su estado de ligadura.

Existen dos formas de salir de este dilema. Podemos cambiar la variable libre v en e_a , por decir a w , o cambiar la variable ligada u a 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 se define en el último caso de la definición. Es más conveniente porque renombrar variables solo concierne a las variables que ocurren en el alcance de la aplicación.

La transformación que renombra:

$$\lambda u.e_0 \rightarrow_\alpha \lambda w.e_0[e \leftarrow w]$$

se conoce como α -conversión. Su realización se basa en 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 $\lambda w.e_0[u \leftarrow w]$. Veamos un ejemplo.

Ejemplo 13 Una secuencia de β -reducciones:

$$\begin{array}{c}
(\lambda u.(\lambda v.(\lambda z.(z u) v) u) z) \\
\downarrow \\
(\lambda v.(\lambda z.(z u) v) u)[u \leftarrow z] \\
\downarrow \\
(\lambda v.(\lambda z.(z u) v)[u \leftarrow z] u[u \leftarrow z]) \\
\downarrow \\
(\lambda v.(\lambda z.(z u) v)[u \leftarrow z] z) \\
\downarrow \\
(\lambda v.(\lambda z.(z u)[u \leftarrow z] v[u \leftarrow z]) z) \\
\downarrow \\
(\lambda v.(\lambda w.(z u)[z \leftarrow w][u \leftarrow z] v) z) \\
\downarrow \\
(\lambda v.(\lambda w.(w u)[u \leftarrow z] v) z) \\
\downarrow \\
(\lambda v.(\lambda w.(w z) v) z) \\
\downarrow \\
(\lambda w.(w z) v)[v \leftarrow z] \\
\downarrow \\
(\lambda w.(w z) w) \\
\downarrow \\
(w z)[w \leftarrow z] \\
\downarrow \\
(z z)
\end{array}$$

En el ejemplo reemplazamos la variable libre w por z para evitar un conflicto entre nombres cuando los β -radices son sistemáticamente reducidos de afuera hacia adentro. En esta secuencia, la última regla de sustitución es aplicada en la quinta expresión de abajo hacia arriba para renombrar la 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 adentro hacía afuera, 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 ligadura 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.

9.4. 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**.

Ejemplo 14 *Las dos abstracciones que se muestran a continuación son sintácticamente equivalentes módulo α -conversión de los nombres de las variables:*

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

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 9.1.

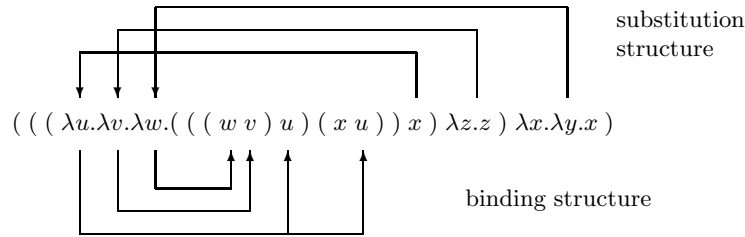


Figura 9.1 Estructuras de sustitución y ligado.

La Figura 9.2 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 hacia 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 sí misma.

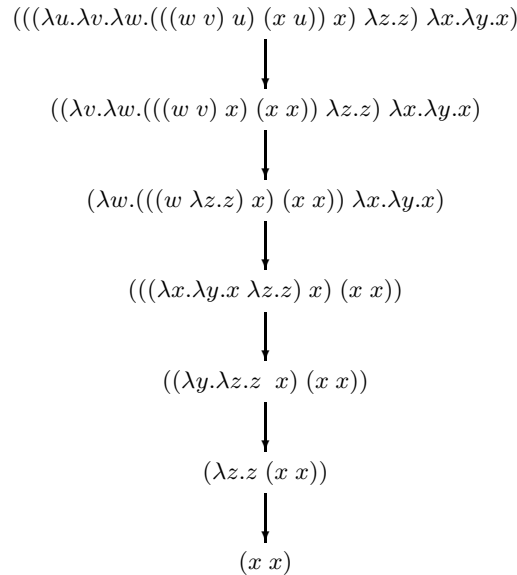


Figura 9.2 Reducción de la expresión ejemplo

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:

$$\underbrace{// \dots //}_n z$$

entonces la ocurrencia está protegida contra las n imbricaciones más internas del abstractor λz que liga a la variable. Usando esta 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, por ejemplo, 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 9.3. 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 9.3 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 desaparecido, de forma que se le debe 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.

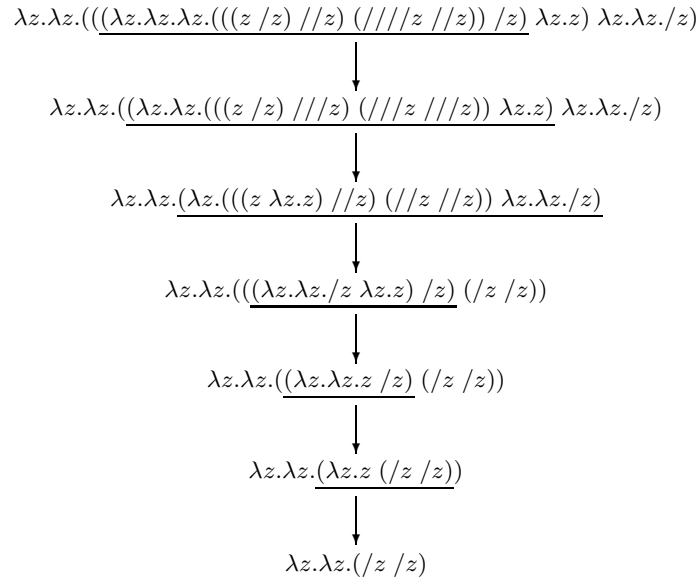


Figura 9.3 Reducción con variables de nombre único y llaves protectoras.

Sea

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

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

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

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:

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.

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:

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

Aplicando estas α -conversiones de adentro hacia afuera y usando la regla de β -reducción definida informalmente antes, se obtiene la secuencia de reducciones que se muestra en la Figura 9.4. Esta secuencia termina con una abstracción en la que todas las ocurrencias de las variables ligadas se substituyeron 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.

$$\begin{array}{c}
 (\lambda v.\lambda z.(v z) \lambda u.(\lambda v.\lambda z.(v z) \lambda v.(\lambda v.\lambda z.(v z) \lambda w.(((w v) u) (x u)))) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.(\lambda v.\lambda z.(v z) \lambda v.\lambda z.(\lambda w.(((w v) u) (x u)) z))) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.(\lambda v.\lambda z.(v z) \lambda v.\lambda z.(((z v) u) (x u)))) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.\lambda z.(\lambda v.\lambda z.(((z v) u) (x u)) z)) \\
 \downarrow \\
 (\lambda v.\lambda z.(v z) \lambda u.\lambda z.\lambda z.(((z/z) u) (x u))) \\
 \downarrow \\
 \lambda z.(\lambda u.\lambda z.\lambda z.(((z/z) u) (x u)) z) \\
 \downarrow \\
 \lambda z.\lambda z.\lambda z.(((z/z) //z) (x //z))
 \end{array}$$

Figura 9.4 Reducción a nombres únicos y llaves de protección adecuadas.

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 ([12], 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.

9.5. Secuencias de reducción

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

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

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 β -reducciones, entonces e^{NF} es también la forma normal de las λ -expresiones intermedias.

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

- para eventualmente todas las posibles secuencias, es decir, tras muchas β -reducciones finitas ejecutadas en cualquier orden; 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.

Consideremos un ejemplo de la primer clase de expresión: $(\lambda u.(\lambda w.(\lambda w.u) u) w)$ donde:

- procediendo de afuera hacía adentro:

$$(\lambda u.(\lambda w.(\lambda w.u) u) w) \rightarrow_{\beta} (\lambda w.(\lambda (w./w /w)w) \rightarrow_{\beta} (\lambda w./w w) \rightarrow_{\beta} w$$

- procediendo de adentro hacía afuera:

$$(\lambda u.(\lambda w.(\lambda w.u) u) w) \rightarrow_{\beta} (\lambda u.(\lambda w.u u) w) \rightarrow_{\beta} (\lambda u.u w) \rightarrow_{\beta} w$$

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

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

$$(\lambda u.(u u) \lambda u.(u u)) \rightarrow_{\beta} (\lambda u.(u u) \lambda u.(u u)) \rightarrow_{\beta} \dots$$

que incesantemente se reproduce a si misma.

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

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

identificamos de manera inmediata que la abstracción $\lambda w.\lambda v.u$ es una **función selector** que reproduce su primer argumento, es decir $\lambda w w$, 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.

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 imperativamente garantizar la terminación con formas normales si estas existen.

La estrategia que garantiza ésto 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, por ejemplo, variables o aplicaciones que no pueden β -reducirse, en la posición del operador de las aplicaciones.

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

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

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 del capítulo 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).

La reducción en orden normal también se conoce como **de afuera a adentro y de izquierda a derecha**. Esta estrategia se conoce también como **llamada por nombre** y es usada por lenguajes de programación como Algol y Simula. Es posible definir una estrategia 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.

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 β -reducciones son llevadas a cabo sobre una λ -expresión, existe siempre una λ -expresión en la cual dos diferentes secuencias de β -reducciones pueden volverse a encontrar. Esto se puede formalizar como sigue:

Sean e_0, e_1, e_2, e_3 λ -expresiones. Entonces $e_0 \mapsto_{\beta} e_1$ y $e_0 \mapsto_{\beta} e_2$ implican que existe una expresión e_3 tal que $e_1 \mapsto_{\beta} e_3$ y $e_2 \mapsto_{\beta} e_3$. 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.

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

- **forma normal completa**, si no contiene β -redices. El cálculo- λ que computa formas normales completas, se conoce como normalizador completo, o simplemente normalizador;
- **forma normal débil (cabeza)**, si es una abstracción de alto nivel (contiene 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

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

- donde $i \in \{0, \dots, n-1\}$, cuya forma no puede cambiar más hacia la izquierda de la variable u_i ya que solo es posible llevar a cabo β -reducciones en las expresiones operando e_1, \dots, 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 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 naif, son suficientes para la normalización débil puesto que solo se permiten reducciones de alto nivel que evitan los conflictos entre nombres.