Capítulo 1

Lambda cálculo no tipado

Vamos a revisar la definición y propiedades básicas del *lambda cálculo puro* o *no tipado*.

A mediados de los 60s, Peter Landin observó que un lenguaje de programación complejo podía ser formulado a través de un reducido núcleo y un conjunto de *expresiones derivadas* que posteriormente serían traducidas a dicho núcleo. El núcleo al que Peter Landin se refería era el *lambda-cálculo*, un sistema formal inventado en los años 20 por Church en el que todos los cálculos se reducen a dos operaciones básicas, la definición de funciones y su aplicación. Hoy en día, el lambda cálculo se puede considerar como el más pequeño lenguaje universal de programación y se utiliza habitualmente en el diseño e implementación de los lenguajes de programación y en el estudio de los sistemas de tipos.

Existen otros sistemas alternativos al lambda-cálculo que pueden ser utilizados para el mismo objetivo. El *pi-cálculo* se utiliza para definir la semántica de lenguajes concurrentes basados en paso de mensajes. El *object-cálculo* se utiliza a su vez para definir las características básicas de los lenguajes orientados a objetos.

El lambda-cálculo básico es extremadamente limitado, pero puede ser enriquecido de diferentes formas. Por ejemplo, introduciendo una sintaxis concreta para ciertas características comunes como los números, las tuplas o los registros. También es posible introducir otros conceptos más avanzados como las referencias o las excepciones.

1.1. Conceptos básicos

La definición de funciones es una de las características básicas de cualquier lenguaje de programación. En lugar de escribir la misma fórmula una y otra vez, definimos una función o procedimiento realiza el cálculo de forma genérica en función (o no) de un conjunto de parámetros. Por ejemplo, un programador sustituiría la expresión:

```
(5 * 4 * 3 * 2 * 1) + (7 * 6 5 * 4 * 3 * 2 * 1) - (3 * 2 * 1)
```

por `factorial(5) + factorial(7) - factorial(3)` una vez definida la función:

```
factorial(n) = if n = 0 then 1 else n * factorial(n-1)
```

En el lambda-cálculo, la expresión ‘‘λn. ...’’ equivale “la función que, para cada n devuelve ...”. Utilizando esta notación, podemos reescribir la función factorial como:

```
factorial = λn. if n = 0 then 1 else n * factorial(n-1)
```

En el lambda-cálculo puro TODO son funciones, los parámetros de las funciones son funciones, el resultado de las funciones también. La sintaxis es muy sencilla, pues sólo existen tres tipos de términos: las variables, las abstracciones de una variable en un término y la aplicación de un término a otro término.

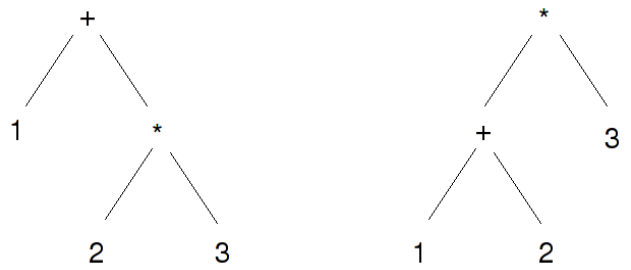
```
t::=
    x
   λx.t
   t t
```

Sintaxis concreta y abstracta

Al hablar de la sintaxis de un lenguaje de programación, debemos distinguir entre la *sintaxis concreta* y la *sintaxis abstracta*. La sintaxis concreta es el conjunto de cadenas de caracteres que los programadores utilizamos para escribir y leer los programas. La sintaxis abstracta es una representación interna muy simplificada en la que los términos se representan como árboles sintácticos.

La transformación entre la sintaxis concreta y la abstracta se lleva a cabo en dos etapas. En primer lugar, un analizador léxico convierte las cadenas de caracteres escritas por el programador en un conjunto de tokens (identificadores, palabras claves, puntuación, etc.). El analizador léxico elimina los comentarios y procesa los espacios en blanco, los formatos de las cadenas de caracteres y números, etc. A continuación, el parseador transforma la secuencia de tokens en un árbol sintáctico. Ciertas convenciones como la precedencia de los operadores reducen la necesidad de incluir paréntesis en la escritura de nuestros programas.

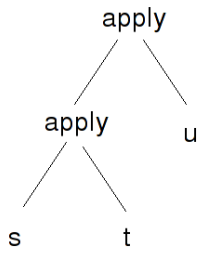
Gracias a las convenciones de asociatividad sabemos que la expresión 1 + 2 * 3 se corresponde con el árbol sintáctico de la izquierda, y no de la derecha.



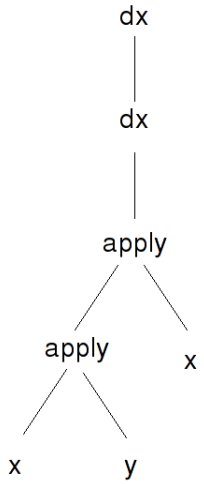
La gramática del lambda cálculo básico debe ser entendida como la estructura de árboles sintácticos y no como un conjunto de cadenas de caracteres utilizados en una sintaxis concreta.

Para evitar el uso excesivo de paréntesis, utilizamos las siguientes convenciones:

1. La aplicación es asociativa por la izquierda `s t u =>(s t) u`.



2. Los cuerpos de las abstracciones son asociativas por la derecha `λx.λy.x y x =>λx.(λy.((x y) x))`



En las expresiones anteriores, las meta-variables `t, s` y `u` son términos, mientras que `x, y` y `z` son variables concretas.

Ámbito

Una variable `x` está ligada cuando aparece en el cuerpo `t` de una abstracción del tipo `λx.t`. Una variable `x` está libre si aparece en un punto no acotado por ninguna abstracción de la forma `λx`. Ejemplos:

1. `x` es una variable libre en `λy.x y`.
2. `x` es una variable ligada en `λx.x` y en `λx.λy. x (y z)`.
3. En `(λx.x) x`, la primera ocurrencia de `x` es libre y la segunda ligada.

Un término sin variables libres se denomina *término cerrado* o *combinador*. El combinador más sencillo es la *función identidad* `λx.x`.

Semántica operacional

En su forma más pura, el lambda-cálculo no tiene constantes, números, operadores aritméticos, condicionales, etc. La semántica operacional consiste únicamente en la aplicación de funciones a argumentos (que a su vez son también funciones). La aplicación de una abstracción se calcula substituyendo la variable acotada por el término correspondiente en el cuerpo de la abstracción.

$(\lambda x.t_{12})\ t_2 \rightarrow [x \mapsto t_2]t_{12}$

La expresión $(\lambda x.t_{12})\ t_2$ se denomina *redex* y la operación de reescribir el redex se denomina beta-reducción.

Ejemplos:

1. $(\lambda x.x)\ y$ se evalúa a y.
2. $(\lambda x.x\ (\lambda x.x))\ (u\ r)$ se evalúa a $(u\ r)\ (\lambda x.x)$.

Existen diferentes estrategias de evaluación en el lambda-cálculo, cada una de las cuales indica qué redex debe ser evaluado en la siguiente evaluación.

- *Beta-reducción completa*: cualquier redex puede evaluarse en cualquier momento. En un determinado paso de la evaluación elegimos en redex que queremos evaluar y lo reducimos. Por ejemplo, el término

  $(\lambda x.x)\ ((\lambda x.x)\ \ (\lambda z.(\lambda x.x)\ z))$

  puede reescribirse como $id\ (id\ (\lambda z.id\ z))$ y contiene tres redexes:

  $\underline{id\ (id\ (\lambda z.id\ z))}$
  $id\ \underline{(id\ (\lambda z.id\ z))}$
  $id\ (id\ (\lambda z.\underline{id\ z}))$

  Si elegimos comenzar por el redex más interno, seguir por el del medio y acabar con el más externo, la secuencia ejecutada será la siguiente:

  $id\ (id\ (\lambda z.\underline{id\ z})) \rightarrow$
  $id\ \underline{(id\ (\lambda z.z))} \rightarrow$
  $\underline{id\ (\lambda z.z)} \rightarrow$
  $\lambda z.z$

- *Orden normal*: los redex que estén más hacia la izquierda (o los más externos) son los que se evalúan primero.

  $\underline{id\ (id\ (\lambda z.id\ z))} \rightarrow$
  $\underline{id\ (\lambda z.id\ z)} \rightarrow$
  $\lambda z.\underline{id\ z} \rightarrow$
  $\lambda z.z$

- *Llamada por nombre*: similar a la estrategia anterior pero más restrictiva, pues no permite llevar a cabo reducciones dentro de abstracciones.

```
id (id (λz.id z)) →
id (λz.id z) →
λz.id z
```

Esta estrategia es la utilizada por lenguajes de programación como Algol-60 o Haskell. Haskel utiliza una variante más eficiente de la llamada por nombre, la *llamada por necesidad* en la que, una vez evaluado un argumento, se sustituyen todas las ocurrencias de dicho argumento con su valor, en lugar de re-evaluar el argumento cada vez que se utiliza.

- *Llamada por valor*: en cada paso de la evaluación se reduce el redex más externo. Además, un redex únicamente puede reducirse cuando el término a su derecha ya ha sido reducido a un valor (un valor es una abstracción).

```
id (id (λz.id z)) →
id (λz.id z) →
λz.id z
```

En llamada por valor los argumentos de la función siempre se evalúan, aunque luego no se usen en el cuerpo de la función. En otras estrategias como llamada por nombre los argumentos no usados no llegan a evaluarse.

Nosotros vamos a utilizar la estrategia de llamada por nombre, porque ser una de las más utilizada en los lenguajes de programación más populares (ML, Lisp, C, Java) y por resultar muy sencillo su enriquecimiento con elementos como excepciones o referencias.

1.2. Programación en el lambda-cálculo

El lambda-cálculo es mucho más potente de lo que su reducida definición podría hacernos pensar. De hecho, vamos a comprobar cómo a través de términos del lambda-cálculo podemos programar diferentes elementos habituales en los lenguajes de programación de alto nivel, como los booleanos, los números o los pares.

Múltiples argumentos

El lambda-cálculo puro no permite la definición de funciones con varios argumentos. Lo que sí permite es definir funciones que a su vez devuelvan funciones como resultado. Si `s` es un término que utiliza dos variables libres `x` e `y`, y queremos escribir una función `f` que para cada par de argumentos `(v,w)` devuelva el resultado de sustituir `x` por `v` e `y` por `w` en `s`, lo que tenemos que hacer, en lugar de definir una función `f = λ(x,y).s` es definir la función de *alto orden* `f = λx.λy.s`. Al aplicar la función `f` a los argumentos `v` y `w`, lo que estamos haciendo es:

`(f v) w` $\rightarrow$ `(`δ `y.[x` $\mapsto$ `v]s) w` $\rightarrow$ `[y` $\mapsto$ `w][x` $\mapsto$ `v]s`

La transformación de funciones con varios argumentos en funciones de alto nivel se denomina *decurrificar* las funciones, en honor de Haskell Curry.

Booleanos de Church

Otra de las características de un lenguaje de programación que puede ser fácilmente codificada con el lambda-cálculo son los booleanos. Los términos `tru` y `fls`, que representan las constantes "true" y "false", respectivamente, se definen como:

```
tru = λt.λf.t
fls = λt.λf.f
```

A partir de estos términos, podemos definir los condicionales. En la expresión:

```
test = λl.λm.λn.l m n
```

el booleano `b` es un condicional: toma dos argumentos y elige entre el primero (cuando `b` es `tru`) o el segundo (cuando `b` es `fls`):

```
test tru v w
= (λl.λm.λn.l m n) tru v w
→ (λm.λn.tru m n) v w
→ (λn.tru v n) w
→ tru v w
= (λt.λf.t) v w
→ (λf.v) w
→ v
```

También podemos definir los operadores lógicos. El operador `and` se definiría como:

```
and = λb.λc.b c fls
```

`and` recibe como entrada dos booleanos `b` y `c` y devuelve `c` si `b` es `tru` o `fls` si `b` es `fls`. Ejemplos:

```
and tru tru
> λt.λf.t
and tru fls
> λt.λf.f
```

Ejercicio: Definir el operador lógico `not`.

Pares

Una vez definidos los booleanos, podemos definir también los pares:

```
pair = λf.λs.λb.b f s
fst = λp. p tru
snd = λp. p fls
```

`pair v w` es una función que, aplicada a un valor booleano `b`, devuelve `v` cuando `b` es `tru` y devuelve `w` cuando `b` es `fls`. Las funciones `fst` y `snd` devuelven la primera y la segunda componente de un par, respectivamente:

```
fst (pair v w)
= fst ((λf.λs.λb.b f s) v w)
→ fst ((λs.λb.b v s) w)
→ fst (λb.b v w)
= (λp. p tru) (λb.b v w)
→ (λb.b v w) tru
→ tru v w
→ ^* v
```

Numerales de Church

Para representar los números en el lambda-cálculo puro utilizamos la siguiente sintaxis:

```
c0 = λs.λ z.z
c1 = λs.λ z.s z
c2 = λs.λ z.s (s z)
c3 = λs.λ z.s (s (s z))
```

Cada número `n` se representa a través de un combinador que toma dos argumentos `s` y `z` (representando el sucesor y el cero) y aplica `s` n veces a `z`. Esto significa que cada número es en realidad una función de dos argumentos.

La definición del c_0 coincide con la definición de `fls`. Este hecho es muy común en los lenguajes de programación de alto nivel, donde un determinado patrón de bits puede representar diferentes valores (entero, flotante, cuatro caracteres, etc.) dependiendo de cómo se interprete. De hecho, en lenguajes como el C, la representación de `0` y `false` es idéntica.

Podemos definir la función sucesor como:

```
scc = λn.λs.λ z.s (n s z)
```

El término `scc` es un combinador que toma un numeral `n` y devuelve otro numeral, es decir, una función que depende de dos parámetros `s` y `z`. Para ello, primero pasa `s` y `z` como argumentos a `n` y aplicando `s` una vez más al resultado obtenido.

Ejercicio: Calcular el sucesor de c_2.

La suma de numerales se define como un combinador que toma como argumentos dos numerales `m` y `n` y devuelve otro numeral, es decir, una función de `s` y `z`. Para ello aplica `s` a `z` un total de `n` veces y a continuación aplica `s` al resultado un total de `m` veces.

```
plus = λm.λn.λs.λz.m s (n s z)
```

Ejercicio: Calcular la suma de c_1 y c_2.

Para definir la multiplicación podemos hacer uso de la función `plus`. La función `times` aplicará `m` veces la función que suma `n` al numeral c_0.

```
times = λm.λn.m (plus n) c0
```

Para comprobar si un numeral es igual a cero, debemos encontrar un par de argumentos `ss` y `zz` tales que, aplicar `ss` a `zz` una o varias veces debe devolver `fls` mientras que no aplicárselo nunca debe devolver `tru`.

```
iszro = λm.m (λx.fls) tru
```

Ejemplos: Calcular `iszro` $\mathtt{c}_0$ y `iszro` $\mathtt{c}_1$.

Vamos a definir ahora la función predecesor que, cuando recibe $\mathtt{c}_0$ como argumento, devuelve $\mathtt{c}_0$ y cuando recibe $\mathtt{c}_{i+1}$ devuelve $\mathtt{c}_i$.

```
zz = pair  c0  c0
ss = λp. pair (snd p) (plus c1 (snd p))
prd = λm.fst (m ss zz)
```

La definición resulta válida porque utiliza el numeral `m` para aplicar la función `ss` a `zz` un total de `m` veces. En esta definición, la función `ss` recibe como argumento un par de numerales `pair` $\mathtt{c}_i$ $\mathtt{c}_j$ y devuelve como resultado un par de numerales `pair` $\mathtt{c}_j$ $\mathtt{c}_{j+1}$. Por tanto, la función `ss` devolverá el predecesor de `m` en la primera componente del par.

Enriquecimiento del lambda cálculo

A pesar de que los elementos básicos de cualquier lenguaje e programación se pueden codificar a través del lambda cálculo puro, a veces resulta conveniente enriquecer dicho cálculo con elementos que no pertenecen al sistema básico, como por ejemplo los números y los booleanos. En general, denotaremos como λ al lambda-cálculo puro y $\lambda\mathbf{NB}$ al enriquecido con booleanos y expresiones aritméticas.

Por tanto, en $\lambda\mathbf{NB}$ tenemos dos posibles implementaciones de los booleanos y de los números:

- Los booleanos y numerales de Church, definidos a partir del lambda-cálculo puro.
- Los booleanos y números añadidos al sistema básico, denominados primitivos.

Por lo tanto, podemos definir funciones de conversión entre unos y otros, como por ejemplo las siguientes:

1. Transformaciones entre booleanos de Church y booleanos primitivo:

   ```
   realbool = λb. b true false

   churchbool = λb. if b then tru else fls
   ```

2. Función que determina si dos numerales de Church son iguales y devuelve un booleano primitivo.

   ```
   realeq = λm. λn.(equal m n) true false
   ```

3. Conversión de un numeral de Church en un número:

   ```
   realnat = λm. m (λx.succ x) 0
   ```

Recursividad

Anteriormente hemos comentado que un término al que no se le puede aplicar ninguna regla de evaluación está en *forma normal*. En el lambda-cálculo puro, algunos términos nunca llegan a estar en forma normal, como por ejemplo el combinador *divergente*.

```
omega = (λx.x x) (λx.x x)
```

Reducir el redex presente en `omega` vuelve a devolver `omega`. De los términos que no se evalúan nunca a una forma normal se dice que *divergen*.

El *combinador de punto fijo*, utilizado para definir funciones recursivas es también divergente.

```
fix = λf.(λx.f (λy.x x y)) (λx.f (λy.x x y))
```

Al igual que `omega`, `fix` tiene una estructura repetitiva difícil de entender leyendo su definición. Vamos a ver cómo funciona en un ejemplo en el que se utiliza para escribir una función recursiva.

La función factorial se define, como:

```
if n = 0 then 1 else n * factorial (n-1)
```

Si "desenrollamos" la definición de factorial cada vez que esta aparece, obtendremos la siguiente definición intuitiva:

```
if n = 0 then 1
else n * (if (n-1) = 0 then 1
          else (n-1) * (if (n-2) = 0 then 1
                        else (n-2) * ... ))
```

Utilizando numerales de Church, la función puede escribirse como:

```
if realeq n c0 then c1
else times n (if realeq (prd n) c0 then c1
              else times (pred n)
                         (if realeq (prd (prd n)) c0 then c1
                          else times (prd (prd n)) ... ))
```

El mismo efecto puede conseguirse utilizando el combinador `fix` definiendo `g =` `λf.<cuerpo conteniendo f>` y a continuación `h = fix g`. En este caso particular, definimos:

```
g = λf2. λn. if realeq n c0 then c1 else (times n (f2 (prd n)))
factorial = fix g
```

Vamos a aplicar la función `factorial` a c_3 y ver qué pasa:

```
factorial c3 =
fix g c3 →
h h c3                  (h = λx. g (λy.x x y)) →
```

```
g fct c3                  (fct = λy. h h y)  →
(λn. if realeq n c0
               then c1
               else times n (fct (prd n))) c3  →
if realeq c3 c0
then c1
else times c3 (fct (prd c3))  →^*
times c3 (fct (prd c3))  →^*
times c3 (fct c2)  →^*
times c3 (g fct c2)  →^*
times c3 (times c2 (g fct c1))  →^*
times c3 (times c2 (times c1 (g fct c0)))  →^*
times c3 (times c2 (times c1 c1))  →^*
c6
```

1.3. Sintaxis del lambda cálculo

La definición inductiva de la sintaxis del lambda-cálculo es la siguiente. Si V es el conjunto de variables, el conjunto de términos será el menor conjunto τ tal que:

1. $\mathtt{x} \in \tau$, para todo $\mathtt{x} \in V$.
2. Si $\mathtt{t}_1 \in \tau$, entonces $\lambda\mathtt{x}.\mathtt{t}_1 \in \tau$.
3. Si $\mathtt{t}_1 \in \tau$ y $\mathtt{t}_2 \in \tau$, entonces $\mathtt{t}_1\ \mathtt{t}_2 \in \tau$.

Vamos a definir inductivamente el conjunto de *variables libres* de un término `t`:

```
FV(x) = {x}
FV(λx.t1) = FV(t1)  {x}
FV(t1 t2) = FV(t1) ∪ FV(2)
```

1.4. Sustitución

La operación de sustitución de una variable por un término dentro de una abstracción es más compleja de lo que pueda parecer a primera vista. Imaginaros que definimos la sustitución `[x ↦ s]` inductivamente de la siguiente manera:

```
[x ↦ s]x          = s
[x ↦ s]y          = y                        (x ≠ y)
[x ↦ s](λy.t1)    = λy.[x ↦ s]t1
[x ↦ s](t1 t2)    = ([x ↦ s]t1) ([x ↦ s]t2)
```

Esta definición funciona bien en la mayoría de los casos, como por ejemplo:

$[x \mapsto (\lambda z.\, z\ w)](\lambda y.\, x) = (\lambda y.\, \lambda z.\, z\ w)$

Pero qué pasa si llevamos a cabo la siguiente sustitución:

$[x \mapsto y](\lambda x.\, x) = \lambda x.\, y$

Hemos transformado la función identidad en una función diferente, lo cual no es correcto. El problema viene de que no hemos establecido una distinción entre las variables libres de las variables ligadas. Si x es una variable libre, podemos sustituirla, si es una variable ligada no:

- **Dos términos que difieran únicamente en el nombre de las variables ligadas son totalmente equivalentes e intercambiables** ($\lambda x.\, x \equiv \lambda z.\, z$).

Por este motivo, no se debe aplicar una sustitución a una variable ligada. La solución al problema consistiría en detener la sustitución $[x \mapsto s]$ en el momento que encontremos una abstracción de la forma $\lambda x.\ \ldots$.

$$
\begin{array}{lll}
[x \mapsto s]x & = s & \\
[x \mapsto s]y & = y & (x \neq y) \\
[x \mapsto s](\lambda y.\, t_1) & = \lambda y.\, t_1 & (y = x) \\
 & = \lambda y.\, [x \mapsto s]t_1 & (x \neq y) \\
[x \mapsto s](t_1\ t_2) & = ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

Hemos solucionado un problema, pero no todos. Imaginemos la siguiente sustitución:

$[x \mapsto z](\lambda z.\, x) = \lambda z.\, z$

Una función que no es la identidad se transforma en la función identidad, lo cual tampoco es correcto. Hemos convertido una variable libre en una variable ligada. Cuando una variable libre se convierte en ligada a causa de una sustitución, se dice que de ella que es una *variable capturada*. Para evitar este tipo de problemas, en $[x \mapsto s]t$, las variables libres del término s deben ser diferentes de las variables ligadas del término t. Para conseguir este efecto, debemos modificar las reglas que rigen la sustitución:

$$
\begin{array}{lll}
[x \mapsto s]x & = s & \\
[x \mapsto s]y & = y & (x \neq y) \\
[x \mapsto s](\lambda y.\, t_1) & = \lambda y.\, t_1 & (x = y) \\
 & = \lambda y.\, [x \mapsto s]t_1 & (x \neq y \ \&\&\ y \notin \mathrm{FV}(s)) \\
[x \mapsto s](t_1\ t_2) & = ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}
$$

Ahora nuestra definición formal de sustitución ya no actúa de forma incorrecta en ningún caso, el único problema es que a veces no hace nada en absoluto! Por ejemplo, en nuestra definición inductiva de la sustitución, no existe ninguna regla válida para el término `[x ↦ y z](λy.x y)`.

Una posible solución sería renombrar la variable ligada para evitar así que una variable libre sea *capturada*. La operación de renombrar una variable ligada se denomina *alfa-reducción*.

Por ejemplo, si queremos sustituir: `[x ↦ y z](λy.x y)`, lo primero que debemos hacer es renombrar `λy.x y` a `λw.x w`. A continuación ya podremos llevar a cabo la sustitución `[x ↦ y z](λw.x w) = λw.y z w`

1.5. Semántica operacional del lambda-cálculo puro

A continuación definimos la semántica operacional del lambda cálculo puro. En primer lugar definimos la gramática como:

```
t::=
    x
   λx.t
   t t

v::=
   λx.t
```

Como la estrategia de evaluación *llamada por valor* se detiene cuando encuentra una abstracción no aplicada a un término, los valores de nuestra gramática son las abstracciones.

Las reglas de evaluación de nuestra semántica son las siguientes:

$$\frac{\mathtt{t}_1 \rightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \rightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \mathtt{E-App1}$$

$$\frac{\mathtt{t}_2 \rightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \rightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \mathtt{E-App2}$$

$$(\lambda \mathtt{x}.\mathtt{t}_{12})\ \mathtt{v}_2 \rightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \mathtt{E-AppAbs}$$

Estas reglas nos dicen que para evaluar la aplicación $\mathtt{t}_1\ \mathtt{t}_2$, primero debemos reducir $\mathtt{t}_1$ a un valor, luego reducir $\mathtt{t}_2$ a un valor, y por último utilizar `E-AppAbs` para realizar la aplicación de la función.