PROGRAMACIÓN FUNCIONAL

Lambda Cálculo: Definición - Sustitución

Lambda Cálculo

- Definición de λ-cálculo
- Noción de *binding*
- Sustitución vs. reemplazo

Lambda Cálculo

- ¿Cómo definimos un lenguaje de programación?
  - Sintaxis (qué *forma* tienen los programas)
  - Semántica (qué *significan* los programas)
- ¿Qué es lo mínimo necesario para tener un lenguaje de programación (funcional)?
  - Variables
  - Abstracción funcional
  - Aplicación de funciones

Lambda Cálculo

- ¿Qué sintaxis podemos usar para escribir funciones y su aplicación?
  - Notación λ (lambda) para funciones
    - Ej: usamos (λx.x) para representar una función que retorna su argumento sin alterarlo (identidad)
  - Yuxtaposición para aplicación
    - Ej: (λx.x)(λx.x) representa la aplicación de la función identidad a sí misma
- ¿Y las variables?
  - Cualquier conjunto infinito de identificadores

Lambda Cálculo

- Conjunto de strings para dar sintaxis
  - Sea V un conjunto infinito de identificadores
    - Usaremos las letras x, y, z, ..., x_0, x_1, ... para denotar elementos de V
  - Definimos el conjunto Λ por inducción
    - si $x \in V$ entonces, también se cumple que $x \in \Lambda$
    - si $x \in V$ y $M \in \Lambda$, entonces $(\lambda x.M) \in \Lambda$
    - si $M,N \in \Lambda$, entonces $(MN) \in \Lambda$
  - De manera sintética
    - $\Lambda ::= V \mid (\lambda V.\Lambda) \mid (\Lambda\Lambda)$
      $V ::= x \mid y \mid z \mid ... \mid x_0 \mid x_1 \mid ...$

Lambda Cálculo

- Ejemplos: x (xy) (λx.(xy)) (λx.(λy.((xy)x)))
  - ¡Hay demasiados paréntesis!
- Convenciones de notación
  - La aplicación asocia a izquierda
    - Así (xyz) significa ((xy)z) y no (x(yz))
  - La aplicación tiene más precedencia que la abstracción
    - Así (λx.xy) significa (λx.(xy)) y no ((λx.x)y)
  - Los paréntesis externos pueden omitirse
    - Así (λx.(λy.xyz)) puede escribirse λx.λy.xyz
  - Pueden juntarse varios λs consecutivos
    - Así (λx.λy.λz.xyz) puede escribirse (λxyz.xyz)

Lambda Cálculo

- ¿Es suficiente con esto para programar?
  - Sí. El λ-cálculo tiene el mismo poder computacional que cualquier lenguaje de programación tradicional
- ¿Por qué es interesante tener tan poco?
  - Permite definiciones simples
  - Facilita el estudio de aspectos computacionales
  - Facilita la demostración de propiedades

Lambda Cálculo

- Historia del λ-cálculo y lenguajes funcionales
  - ~1930 – Haskell B. Curry (lógica combinatoria)
    Alonzo Church (λ-cálculo)
    Kurt Gödel (incompletitud)
  - ~1940 – Alan Turing (Máquina de Turing)
    John von Neumann (Arquitectura)
  - ~1950 – John McCarthy (LISP)
  - ~1960 – Peter Landin (λ-cálculo en programación)
  - ~1970 – Robin Milner (ML)
  - ~1980 – John Hughes, Simon Peyton Jones (Haskell)

Lambda Cálculo

- Usos del λ-cálculo
  - Para compilación de lenguajes funcionales
  - Para dar semántica a lenguajes imperativos
    e.g. Algol, LIS
  - Como formalismo para definir otras teorías
    e.g. lógicas, sistemas de reescritura
    - $(\forall x.P(x))$ se representaría como $\forall(\lambda x.P)$
  - Como inspiración para otros cálculos
    e.g. π-cálculo (concurrencia), σ-cálculo (objetos)

Lambda Cálculo

- Estilos de presentación
  - Versión concreta
    - Sintaxis concreta + definiciones con detalles
    - Ejemplo: paréntesis, diferencias de variables
  - Versión abstracta
    - Sintaxis abstracta + definiciones generales
    - Ejemplo: renombre de variables gratis
- Características
  - La visión concreta es más realista (¡pero ineficiente!)
  - La visión abstracta es más intuitiva
  - Intentaremos ir de una a la otra...

Lambda Cálculo

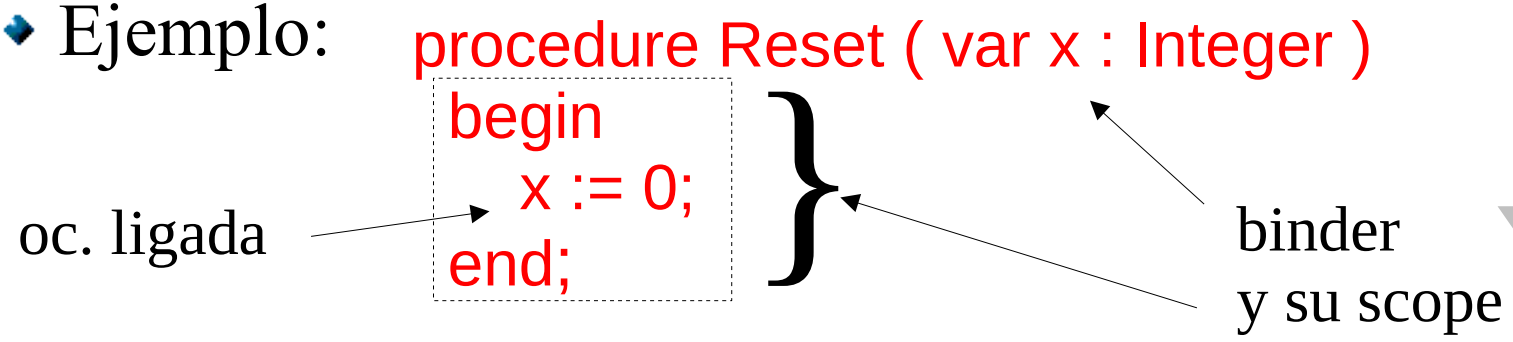
- Versión concreta
  - Λ ::= V | (λV.Λ) | (ΛΛ)
    V ::= x | y | z | ... | x_0 | x_1 | …
    - Los símbolos rojos son terminales
    - La interpretación es como strings
- Versión abstracta
  - $M ::= x \mid \lambda x.M \mid MN$
    - La interpretación es como árboles
    - Sólo se concentra en la estructura
    - Hay otras versiones concretas con la misma estructura
      - Λ ::= V | (\V->Λ) | (ΛΛ)
        V ::= 1 | 2 | 3 | ... | 10 | 11 | …

Lambda Cálculo

- *Binding* (Ligadura de variables)
  - Es un concepto recurrente en programación
  - Las apariciones (ocurrencias) de variables en una expresión son de tres tipos:
    - ocurrencias de ligadura (*binders*)
    - ocurrencias ligadas (*bound occurrences*)
    - ocurrencias libres (*free occurrences*)
  - Cada *binder* tiene un alcance (*scope*), y toda ocurrencia de esa misma variable en el *scope* está ligada (*bounded*) a dicho *binder* (si hay colisión, se liga al de menor *scope*)

Lambda Cálculo

- ¿Para qué sirve la idea de *binding*?
  - Un *binder* identifica y define a una entidad (se lo suele llamar *parámetro formal*)
  - Las ocurrencias ligadas de una variable denotan la entidad asociada al *binder* a la que están ligadas
  - Ejemplo:



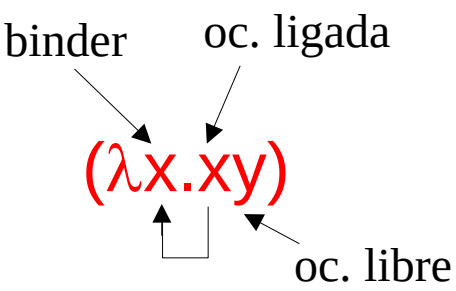
  - Y las ocurrencias libres ¿a qué corresponden?

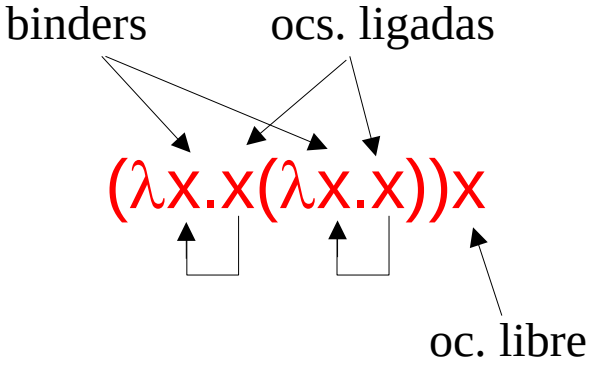
Lambda Cálculo

- ¿Cómo es el *binding* en λ-cálculo?
  - Cada ocurrencia que sigue a un λ es un *binder*
  - Su *scope* es el cuerpo de la abstracción
  - Las demás son ocurrencias libres
- Formalmente:
  - la ocurrencia de x en x es libre
  - toda ocurrencia en M y N permanece igual en (MN)
  - la ocurrencia de x que sigue al λ en $(\lambda x.M)$ es un *binder*
  - toda ocurrencia libre de x en M es una ocurrencia ligada en $(\lambda x.M)$ (y se liga a ese *binder*)
  - toda oc. que no es ligada ni *binder* en $(\lambda x.M)$ es libre en $(\lambda x.M)$

Lambda Cálculo

- Ejemplos

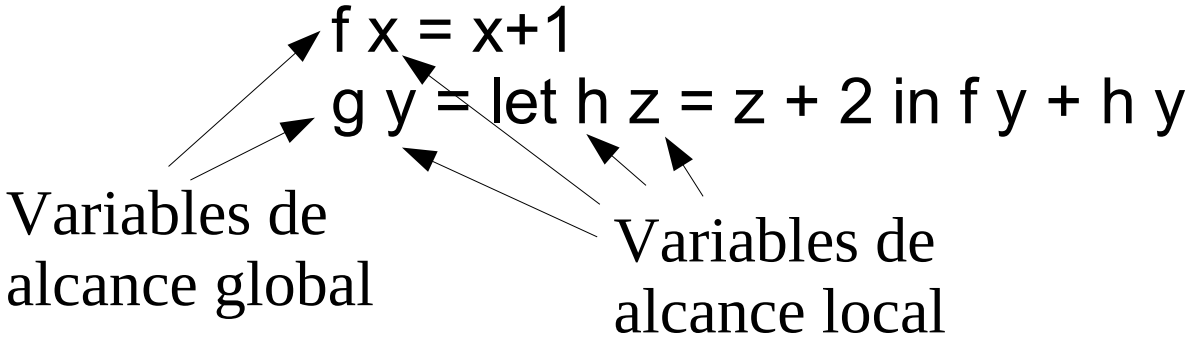




- Observamos que
  - una misma variable puede ocurrir libre y ligada
  - distintas ocurrencias pueden ligarse a distintos binders
  - la ligadura depende de toda la expresión
    (una ocurrencia cambia de "status" de una subexpresión a la expresión final; ej: x vs. (λx.x))

Lambda Cálculo

- Las variables libres
  - Dependen de toda la expresión
    - Una variable libre en una expresión puede ser ligada en otra que la contiene
  - Son variables cuyo significado depende del contexto
  - No deben confundirse con “variables globales”
    - Que sería mejor llamar “variables de alcance global”



Lambda Cálculo

- Variables libres de un término
  - $FV :: \Lambda \rightarrow \wp(V)$
  - $FV(x) = \{ x \}$
  - $FV(MN) = FV(M) \cup FV(N)$
  - $FV(\lambda x.M) = FV(M) / \{ x \}$
- Conceptos
  - Si $FV(M) = \varnothing$, M se dice *cerrado*
  - Sólo llamamos programas a los términos cerrados
    - Significan algo independientemente del contexto

Lambda Cálculo

- ¿Cómo modelamos el cambio de un parámetro formal por uno real en un término?
  - Un parámetro formal corresponde a una variable ligada y sus ocurrencias
  - Por lo tanto, podemos cambiar cada ocurrencia ligada de esa variable por el término que representa al parámetro real
  - Ej: siendo $f(x) = 2*x+1$, $f(3)$ es igual a $2*3+1$
- ¿Y en λ-cálculo?

Lambda Cálculo

- Reemplazo (*search&replace*)
  - Cambiar una variable por un término
    - Ej: reemplazar x por (λy.y) en xz da (λy.y)z
  - ¿Qué pasa con los *bindings*?
    - Ej: reemplazar x por (λy.yz) en (λz.xz) da (λz.(λy.yz)z)
  - ¿Es el resultado esperado? ¿Por qué?
    - ¡¡El *binding* de z en (λy.yz) cambió!!
  - ¿Qué significa que un *binding* cambie?
    - ¡La entidad denotada por la variable es otra!
  - ¿Qué debemos hacer para no *capturar variables*?

Lambda Cálculo

- Sustitución
  - Cambiar una variable por un término, teniendo en cuenta los *bindings*
  - Dado que el nombre de una variable ligada no es importante, podemos renombrarla
    - Ej: sustituir x por (λy.yz) en (λz.xz) da (λw.(λy.yz)w)

      (observar que la z del término (λz.xz) cambió a w para evitar la captura de la z de (λy.yz))
  - Las entidades denotadas, ¿son las mismas?
    O sea, ¿cambió algún *binding*?

Lambda Cálculo

- Sustitución (definición)
  - Dados $M,N \in \Lambda$, y $x \in \mathrm{V}$, se define $M\{x \leftarrow N\}$ (el término resultante de sustituir x por N en M) por inducción *en el tamaño* de M

) $x\{x \leftarrow N\} = N$

a) $y\{x \leftarrow N\} = y$, si $y \neq x$

b) $(PQ)\{x \leftarrow N\} = (P\{x \leftarrow N\}Q\{x \leftarrow N\})$

c) $(\lambda x.P)\{x \leftarrow N\} = (\lambda x.P)$

d) $(\lambda y.P)\{x \leftarrow N\} =$

1) $(\lambda y.P\{x \leftarrow N\})$, si $y \neq x$ e $y \notin \mathrm{FV}(N)$

2) $(\lambda z.P\{y \leftarrow z\}\{x \leftarrow N\})$, si $y \neq x$ e $y \in \mathrm{FV}(N)$

(donde $z \notin \mathrm{FV}(N) \cup \mathrm{FV}(P)$)

Lambda Cálculo

- Explicación
  - Los casos a), b) y c) son simples
  - En d), la *x* ligada en *M* es distinta de la que se sustituye y por ello *M* no cambia
  - En e1), no hay peligro de captura, y se procede inductivamente
  - En e2), para evitar la captura de *y* en *N* se renombra *y* a una nueva variable *z*, antes de proseguir inductivamente
- Ej: $(\lambda z.xz)\{x \leftarrow (\lambda y.yz)\}$ es igual a $(\lambda w.(\lambda y.yz)w)$
  - Se aplica e2), obteniendo $(\lambda w.(xz)\{z \leftarrow w\}\{x \leftarrow (\lambda y.yz)\})$
  - Aplicando c), luego b) y a) obtenemos $(\lambda w.(xw)\{x \leftarrow (\lambda y.yz)\})$
  - Finalmente, c) y luego b) y a) dan el resultado final

Lambda Cálculo

- ¿Qué propiedades tiene la sustitución?
- Lema de sustitución
  - si $x \notin \mathrm{FV}(Q)$, entonces
    $M\{x \leftarrow P\}\{y \leftarrow Q\} = M\{y \leftarrow Q\}\{x \leftarrow P\{y \leftarrow Q\}\}$
- Propiedad (a veces llamada *garbage collection*)
  - si $x \notin \mathrm{FV}(M)$,
    entonces $M\{x \leftarrow N\} = M$

¡Estos iguales no quieren decir idéntico!
¿Por qué?

Resumen

- Hacen falta muy pocos conceptos bien ensamblados para tener un lenguaje de programación
- El λ-cálculo es importante para el estudio de lenguajes de programación
- Las nociones de *binding*, sustitución y renombre de variables son útiles en toda teoría de lenguajes que considere abstracción