PROGRAMACIÓN FUNCIONAL

Lambda Cálculo: Programación

Lambda Cálculo

- Programando con λ-cálculo
  - Bottom
  - Booleanos
  - Pares
  - Maybe
  - Listas (y otras estructuras recursivas)
  - Números enteros
  - Recursión

Representando tipos

- ¿Es suficiente el λ-cálculo para programar?
  - Sí. Para mostrarlo, veremos como representar tipos de datos elementales con λ-expresiones.
- ¿Qué significa *representar un tipo* en λ-cálculo?
  - Establecemos qué propiedades deben cumplirse (especificación)
  - Establecemos qué forma tienen:
    - las expresiones que representan elementos del tipo
    - las expresiones que representan operaciones del tipo, de tal forma que respeten la especificación

Definiendo bottom

- Bottom: ¿Cómo definir un término para $\perp$?
  - Con una expresión cuya computación no termine
  - Especificación:

$$bottom \equiv_{def} \ldots$$

tal que

$$(bottom \rightarrow_{\beta}^{*} bottom \rightarrow_{\beta}^{*} \ldots)$$

- Para solucionarlo se puede pensar en la paradoja del Barbero

Definiendo bottom

- Bottom: ¿Cómo definir un término para $\perp$?
  - Con una expresión cuya computación no termine
  - Especificación:

$$bottom \equiv_{def} (\lambda x.xx)(\lambda x.xx)$$

tal que

$$(bottom \rightarrow_{\beta}^{*} bottom \rightarrow_{\beta}^{*} \ldots)$$

- Es fácil ver que la computación de *bottom* no termina

Consideraciones

- Notación
  - introduciremos nombres para representar expresiones
  - usaremos el símbolo $\equiv_{def}$ para ello (similar al #define de C)
  - sólo es una convención sintáctica para simplificar la lectura
- Observaciones
  - Si bien el lenguaje base no tiene tipos, asumiremos que las construcciones que hacemos sí los tienen
  - No nos preocupa el significado de expresiones que no respetan estas reglas de formación 'implícitas'
    - Ej: (*not* $\underline{2}$) será una λ-expresión válida, pero no nos molestaremos por este tipo de expresiones
  - El tratamiento de tipos es tema para otro curso

Definiendo Booleanos

- Booleanos: especificación

  True $\equiv_{\text{def}}$ ...

  False $\equiv_{\text{def}}$ ...

  ifthenelse $\equiv_{\text{def}}$ $(\lambda b.\lambda m.\lambda n.\ \ldots)$

  tal que para todo par de λ-expresiones M y N

  ifthenelse True $M\,N \rightarrow_{\beta}^{*} M$

  ifthenelse False $M\,N \rightarrow_{\beta}^{*} N$

- Observaciones
  - Lo único que deben cumplir es elegir entre 2 alternativas
  - La construcción if es representable como una función

Definiendo Booleanos

- Primer abstraemos *M* y *N*; o sea alcanza pedir que

  ifthenelse True $\rightarrow_\beta^*$ $(\lambda x.\lambda y.x)$

  ifthenelse False $\rightarrow_\beta^*$ $(\lambda x.\lambda y.y)$

- Cualquier grupo de expresiones que cumplan esto sirve
- La solución más simple es decidir que *ifthenelse* sea la identidad ($(\lambda b.b)$ o equivalentemente $(\lambda b.\lambda m.\lambda n.b\ m\ n)$)
  - Entonces *True* y *False* son claramente las expresiones resultantes a la derecha

Definiendo Booleanos

- Booleanos: solución

$$\begin{aligned} \mathit{True} &\equiv_{\text{def}} (\lambda x.\lambda y.x) \\ \mathit{False} &\equiv_{\text{def}} (\lambda x.\lambda y.y) \\ \mathit{ifthenelse} &\equiv_{\text{def}} (\lambda b.\lambda m.\lambda n.b\ m\ n) \end{aligned}$$

cumple que para todo par de λ-expresiones M y N

$\mathit{ifthenelse}\ \mathit{True}\ M\ N \rightarrow_\beta^* M$

$\mathit{ifthenelse}\ \mathit{False}\ M\ N \rightarrow_\beta^* N$

- Observaciones
  - ¡*True* y *False* son funciones! (¿podía ser de otra manera?)

Definiendo Booleanos

- ¿Existen otras representaciones que cumplan?
- Sí. Una alternativa es dar vuelta las representaciones

$$\begin{aligned} \mathit{True}' &\equiv_{\text{def}} (\lambda x.\lambda y.y) \\ \mathit{False}' &\equiv_{\text{def}} (\lambda x.\lambda y.x) \\ \mathit{ifthenelse}' &\equiv_{\text{def}} (\lambda b.\lambda m.\lambda n.b\ n\ m) \end{aligned}$$

- También podemos pensar otras alternativas
  - los booleanos con más parámetros y el if enviando los valores correspondientes a b
  - Preferiremos la primera, por simplicidad

Definiendo Booleanos

- ¿Y otras operaciones sobre booleanos?
  - Se definen usando *ifthenelse*
- Por ejemplo, siendo M un booleano cualquiera:
  - *and True* $M \rightarrow_\beta^* M$
  - *and False* $M \rightarrow_\beta^* False$

en consecuencia

$$and \equiv_{def} \lambda b_1.\lambda b_2.ifthenelse\ b_1\ b_2\ False$$

- Usando una notación infija para *ifthenelse*, queda

$$and \equiv_{def} \lambda b_1.\lambda b_2.if\ b_1\ then\ b_2\ else\ False$$

Definiendo Booleanos

- Expandiendo los sinónimos, *and* queda
  - $\lambda b_1.\lambda b_2.(\lambda b.\lambda m.\lambda n.b\ m\ n)\ b_1\ b_2\ (\lambda x.\lambda y.y)$

  y reduciendo a β-fn (removiendo la abstracción),
  - $\lambda b_1.\lambda b_2.b_1\ b_2\ (\lambda x.\lambda y.y)$

  Así es fácil ver que cumple la especificación
- Ejemplo: *and True (and True False)* $\equiv_{def}$

$$\overbrace{(\lambda b_1.\lambda b_2.b_1\ b_2\ (\lambda x.\lambda y.y))}^{and}\ \overbrace{(\lambda x.\lambda y.x)}^{True}$$

$$((\underbrace{\lambda b_1.\lambda b_2.b_1\ b_2\ (\lambda x.\lambda y.y)}_{and})\ \underbrace{(\lambda x.\lambda y.x)}_{True}\ \underbrace{(\lambda x.\lambda y.y)}_{False})$$

Definiendo Booleanos

- Para reducir una expresión con sinónimos
  - expandirla completamente y llevarla a β-fn
    - *and True False* $\equiv (\lambda b_1\, b_2.b_1\, b_2\, (\lambda xy.y))\, (\lambda xy.x)\, (\lambda xy.y) \rightarrow_\beta^*$ $(\lambda xy.x)\, (\lambda xy.y)\, (\lambda xy.y) \rightarrow_\beta^* (\lambda xy.y) \equiv$ *False*
  - irla expandiendo y β-reduciendo según haga falta
    - *and True False* $\equiv (\lambda b_1\, b_2.b_1\, b_2$ *False*$)$ *True False* $\rightarrow_\beta^*$ *True False False* $\equiv (\lambda xy.x)$ *False False* $\rightarrow_\beta^*$ *False*
  - utilizar las especificaciones de los tipos (luego de haber chequeado que funcionan)
    - *and True False* $\rightarrow_\beta^*$ *False*

Definiendo Booleanos

- Ejercicios
  - dar una λ-expresión *iff* que para todo booleano M cumpla
    - *iff True* $M \rightarrow_\beta^* M$
    - *iff False True* $\rightarrow_\beta^*$ *False*
    - *iff False False* $\rightarrow_\beta^*$ *True*
  - especificar y representar las operaciones *not*, *or* y *xor*
  - reducir las expresiones mediante los tres métodos
    - $(\lambda b_1\ b_2.$ *and* $(or\ b_1\ b_2)\ (not\ (and\ b_1\ b_2)))$ *True False*
    - $(\lambda b.$ *and* $(xor\ b\ (not\ b))\ (iff\ b\ b))$ *False*

Definiendo Pares

- Pares: especificación

$$\begin{aligned} \mathit{pair} &\equiv_{\text{def}} (\lambda xy.\ \ldots) \\ \mathit{fst} &\equiv_{\text{def}} (\lambda p.\ \ldots) \\ \mathit{snd} &\equiv_{\text{def}} (\lambda p.\ \ldots) \end{aligned}$$

tal que para todo par de λ-expresiones M y N

$$\mathit{fst}\ (\mathit{pair}\ M\ N) \rightarrow_{\beta}^{*} M$$

$$\mathit{snd}\ (\mathit{pair}\ M\ N) \rightarrow_{\beta}^{*} N$$

- Observar que las ecuaciones son similares a las de booleanos (¡pero no iguales!)

Definiendo Pares

- La expresión (*pair* *M* *N*) representa al par (*M*, *N*)
- ¿Cómo podemos usar la similitud de esta especificación con la de los booleanos?
  - La idea es que un par debe elegir entre *M* o *N* según se use con *fst* o *snd*
  - Entonces, *pair* sería un *ifthenelse* con un parámetro
  - *fst* y *snd* instanciarían el parámetro adecuadamente
  - Ello nos lleva directo a la solución

Definiendo Pares

- Pares: solución

$$\begin{aligned} \mathit{pair} &\equiv_{\text{def}} (\lambda xy.\lambda b.\mathit{if}\ \ b\ \mathit{then}\ x\ \mathit{else}\ y) \\ \mathit{fst} &\equiv_{\text{def}} (\lambda p.(p\ \mathit{True})) \\ \mathit{snd} &\equiv_{\text{def}} (\lambda p.(p\ \mathit{False})) \end{aligned}$$

cumple que para todo par de λ-expresiones M y N

$$\mathit{fst}\ (\mathit{pair}\ M\ N) \rightarrow_{\beta}^{*} M$$

$$\mathit{snd}\ (\mathit{pair}\ M\ N) \rightarrow_{\beta}^{*} N$$

- ¡Observar que el par (*pair* M N) es una función!
  (Ya no debería sorprender...)

Definiendo Pares

- Ejemplo:
  - el par (True, and) se representaría
    - *pair True and* $\equiv_{\text{def}}$

$$\underbrace{(\lambda xy.\lambda b.\ \textit{ifthenelse}\ b\ x\ y)}_{\textit{pair}}\ \underbrace{(\lambda xy.x)}_{\textit{True}}\ \underbrace{(\lambda b_1\ b_2.b_1\ b_2\ (\lambda xy.y))}_{\textit{and}}$$

  - al β-reducir queda $(\lambda b.\ \textit{if}\ \ b\ \textit{then True else and})$
- Ejercicio:
  - construir funciones para tuplas de 3 y 4 elemenos

Definiendo Maybe

- Maybe: especificación

$$\begin{aligned} \textit{Nothing} &\equiv_{\text{def}} \ldots \\ \textit{Just} &\equiv_{\text{def}} (\lambda x.\ \ldots) \\ \textit{caseMaybe} &\equiv_{\text{def}} (\lambda m.\lambda z f.\ \ldots) \end{aligned}$$

tal que para todas F y Z se cumple que

$$\textit{caseMaybe}\ \textit{Nothing}\ Z\ F \rightarrow_{\beta}^{*} Z$$

$$\textit{caseMaybe}\ (\textit{Just}\ X)\ Z\ F \rightarrow_{\beta}^{*} F\ X$$

- El *caseMaybe* es la versión del case, pero en "formato fold"

Definiendo Maybe

- Usamos la misma técnica que con booleanos
  - "Pasar" los argumentos como parámetros

    $\mathit{caseMaybe}\ \mathit{Nothing} \rightarrow_\beta^* (\lambda f.\lambda z.z)$

    $\mathit{caseMaybe}\ (\mathit{Just}\ X) \rightarrow_\beta^* (\lambda f.\lambda z.f\ X)$
  - Expresar el *caseMaybe* como la identidad
  - Entonces los constructores quedan como las expresiones de la derecha

Definiendo Maybe

- Maybe: solución

$$\begin{aligned} \textit{Nothing} &\equiv_{\text{def}} (\lambda zf.z) \\ \textit{Just} &\equiv_{\text{def}} (\lambda x.\lambda zf.f\ x) \\ \textit{caseMaybe} &\equiv_{\text{def}} (\lambda m.\lambda zf.m\ z\ f) \end{aligned}$$

cumple que para todas F y Z se cumple que

$$\textit{caseMaybe}\ \textit{Nothing}\ Z\ F \rightarrow_{\beta}^{*} Z$$

$$\textit{caseMaybe}\ (\textit{Just}\ X)\ Z\ F \rightarrow_{\beta}^{*} F\ X$$

- ¡Son como booleanos con un argumento!

Definiendo Listas

- Listas: especificación

$$nil \equiv_{def} \ldots$$
$$cons \equiv_{def} (\lambda x.\lambda xs.\ \ldots)$$
$$foldr \equiv_{def} (\lambda f.\lambda z.\lambda xs.\ \ldots)$$

tal que para todas F y Z se cumple que

$$foldr\ F\ Z\ nil \rightarrow_{\beta}^{*} Z$$
$$foldr\ F\ Z\ (cons\ X\ XS) =_{\beta} F\ X\ (foldr\ F\ Z\ XS)$$

- Observar que en la 2da se usa un signo =

Definiendo Listas

- La recursión queda capturada por el *foldr*
- Cualquier cosa que cumpla esto sirve como listas...
- La idea es proceder como con los booleanos
  - juntar el *foldr* con los constructores
    (*foldr'* = *flip foldr*)
  - “pasar” los argumentos como parámetros
  - expresar el *foldr'* como la identidad

Definiendo Listas

- Juntar el *foldr* con los constructores

  $(foldr'\ nil)\ F\ Z \rightarrow_\beta^*\ Z$

  $(foldr'\ (cons\ X\ XS))\ F\ Z =_\beta F\ X\ ((foldr'\ XS)\ F\ Z)$

- "Pasar" los argumentos como parámetros

  $(foldr'\ nil) \rightarrow_\beta^*\ (\lambda f.\lambda z.z)$

  $(foldr'\ (cons\ X\ XS)) =_\beta (\lambda f.\lambda z.f\ X\ ((foldr'\ XS)\ f\ z))$

- Expresar el *foldr'* como la identidad

  $nil \rightarrow_\beta^*\ (\lambda f.\lambda z.z)$

  $(cons\ X\ XS) =_\beta (\lambda f.\lambda z.f\ X\ (XS\ f\ z))$

Definiendo Listas

- Listas: solución

$$\begin{aligned} \mathit{nil} &\equiv_{\text{def}} (\lambda f.\lambda z.z) \\ \mathit{cons} &\equiv_{\text{def}} (\lambda x.\lambda xs.(\lambda f.\lambda z.f\ x\ (xs\ f\ z))) \\ \mathit{foldr} &\equiv_{\text{def}} (\lambda f.\lambda z.\lambda xs.\ xs\ f\ z) \end{aligned}$$

cumple que para todas F y Z se cumple que

$$\mathit{foldr}\ F\ Z\ \mathit{nil} \rightarrow_{\beta}^{*} Z$$

$$\mathit{foldr}\ F\ Z\ (\mathit{cons}\ X\ XS) =_{\beta} F\ X\ (\mathit{foldr}\ F\ Z\ XS)$$

- ¡Las listas son funciones que esperan para realizar la recursión!

Definiendo Listas

- Observaciones
  - las listas son funciones (obvio)
  - se representan con el patrón de recursión '*diferido*' (o sea, como en los booleanos, haciendo que el *foldr* sólo delegue su trabajo)
  - las operaciones se sintetizan de la especificación
  - el resultado funciona como una especie de “double dispatch”, donde el *foldr* delega el trabajo en los constructores

Definiendo Listas

- Ejemplos:
  - [2,3] se representa como (*cons* *2* (*cons* *3* *nil*)),
    que luego de β-reducir queda (λf.λz.f *2* (f *3* z))
  - [v,w,x,y] se representa como

    (*cons* v (*cons* w (*cons* x (*cons* y *nil*)))),

    que luego de β-reducir queda λf.λz.f v (f w (f x (f y z)))
- En general
  - se representa una lista como el resultado de hacerle *foldr*,
    pero parametrizando las funciones *F* y *Z*

Definiendo Listas

- ¿Cómo definir funciones sobre listas?
  - Utilizando el patrón de recursión
    - *length* $\equiv_{def}$ λxs. *foldr* (λx.*succ*) *0* xs
- Ejemplos (reduciendo el *foldr*)
  - *length* $\equiv_{def}$ λxs. xs (λx n.*succ* n) *0*
  - *sum* $\equiv_{def}$ λxs. xs (λx y.*suma* x y) *0*
  - *map* $\equiv_{def}$ λf. λxs. xs (λx zs. *cons* (f x) zs) *nil*

Definiendo otros tipos

- ¿Y otros tipos recursivos?
  - se representarían en base a su fold
  - el fold sólo debe delegar el trabajo
  - las operaciones constructoras serían consecuencia de esta decisión y la especificación
- Ejercicios
  - Definir árboles binarios (tipo Tree)
  - Definir expresiones aritméticas (tipo ExpA)

Definiendo Números

- Números: especificación

$$\underline{0} \equiv_{\text{def}} \ldots$$
$$succ \equiv_{\text{def}} (\lambda n.\ \ldots)$$
$$foldNat \equiv_{\text{def}} (\lambda s.\lambda z.\lambda n.\ \ldots)$$

  cumple que para todos S y Z

$$foldNat\ S\ Z\ \underline{0} \rightarrow_{\beta}^{*} Z$$
$$foldNat\ S\ Z\ (succ\ N) =_{\beta} S\ (foldNat\ S\ Z\ N)$$

- Seguimos la idea para otros tipos recursivos (y así obtenemos los “numerales de Church”)

Definiendo Números

- La recursión queda capturada por el *foldNat*
- Cualquier cosa que cumpla esto sirve como números...
- La idea es proceder como con otros tipos recursivos
  - juntar el *foldNat* con los constructores (*foldNat'* = *flip foldNat*)
  - “pasar” los argumentos como parámetros
  - expresar el *foldNat'* como la identidad

Definiendo Números

- Números: solución

$$\underline{0} \equiv_{\text{def}} (\lambda s.\lambda z.z)$$
$$succ \equiv_{\text{def}} (\lambda n.(\lambda s.\lambda z.s\ (n\ s\ z)))$$
$$foldNat \equiv_{\text{def}} (\lambda s.\lambda z.\lambda n.\ n\ s\ z)$$

cumple que para todos S y Z

$$foldNat\ S\ Z\ \underline{0} \rightarrow_{\beta}^{*} Z$$
$$foldNat\ S\ Z\ (succ\ N) =_{\beta} S\ (foldNat\ S\ Z\ N)$$

- Nuevamente observamos el “double dispatch” de representar los números como su fold *diferido*

Definiendo Números

- Ejemplos:
  - 2 se representa como *2* dado por *succ* (*succ* *0*)), que luego de β-reducir queda (λs.λz.s (s z))
  - *2* ≡ *succ* (*succ* *0*))

    succ *succ* *0*

    $\equiv_{def}$ (λn s z.s (n s z))((λn' s' z'.s' (n' s' z'))(λs'' z''.z''))

    $\rightarrow_{\beta}^{*}$ (λs z.s ((λn' s' z'.s' (n' s' z'))(λs' 'z''.z'')) s z)

    $\rightarrow_{\beta}^{*}$ (λs z.s (λs' z'.s' ((λs'' z''.z'') s' z')) s z)

    $\rightarrow_{\beta}^{*}$ (λs z.s ((λs' z'.s' z') s z)) $\rightarrow_{\beta}^{*}$ (λs z.s (s z))
  - 3 se representa como *3* dado por *succ* (*succ* (*succ* *0*))), que luego de β-reducir queda (λs.λz.s (s (s z)))

Definiendo Números

- Más ejemplos
  - 17 se representa como *17* dado por (*succ* ... (*succ* *0*)...)),

    17 veces

    que luego de β-reducir queda (λs.λz.s (... (s z)...))
  - un número n se representa como *n* dado por

    (*succ* ... (*succ* *0*)...)),

    n veces

    que luego de β-reducir queda (λs.λz.s (... (s z)...))

Definiendo Números

- Notación
  - $F^{(0)}Z \equiv_{\text{def}} Z$
  - $F^{(n+1)}Z \equiv_{\text{def}} F^{(n)}(FZ)$
- Ejemplo:
  - $(\lambda x.x)^{(2)}y$
    - $\equiv (\lambda x.x)^{(1)}((\lambda x.x)y)$
    - $\equiv (\lambda x.x)^{(0)}((\lambda x.x)((\lambda x.x)y))$
    - $\equiv (\lambda x.x)((\lambda x.x)y)$
- Observar:
  - el n en la expresión $F^{(n)}Z$ es una constante fuera de Λ

Definiendo Números

- Observaciones
  - ¡¡los números son funciones!!
  - la cantidad que un 'número' representa se usa para aplicar una función *S* esa cantidad de veces
  - el n utilizado en *n* es una constante fuera de Λ
  - la representación del *0* y la de *False* coinciden
    - (pero no hay problemas, pues no consideramos expresiones en las que no coincidan los 'tipos')

Definiendo Números

- ¿Cómo usamos esta notación para definir cada *n*?
- Usamos el esquema $\underline{n} \equiv_{def} (\lambda s.\lambda z.s^{(n)}z)$
  - O sea:

$$\underline{0} \equiv_{def} (\lambda s.\lambda z.s^{(0)}z) \equiv_{def} (\lambda s.\lambda z.z)$$
$$\underline{1} \equiv_{def} (\lambda s.\lambda z.s^{(1)}z) \equiv_{def} (\lambda s.\lambda z.s\ z)$$
$$\underline{2} \equiv_{def} (\lambda s.\lambda z.s^{(2)}z) \equiv_{def} (\lambda s.\lambda z.s\ (s\ z))$$
$$\vdots$$

Definiendo Números

- ¿Cómo definimos funciones sobre naturales?
  - Con *foldNat*
- Ejemplo:
  - definir un término *suma* para la función suma
  - debe cumplir $\mathit{suma}\ \underline{n}\ \underline{m} \rightarrow_\beta^* \underline{n+m}$
  - podemos usar *foldNat*
    - $\mathit{suma} \equiv_{\text{def}} (\lambda n.\lambda m.\mathit{foldNat}\ \mathit{succ}\ m\ n)$

Definiendo Números

- Vemos que *succ* se usa de la siguiente manera
  - $m+n$ es igual a sumar n veces 1 a m (o sea, $succ^{(n)}m$)
  - *succ* (*succ* (*succ* ... (*succ m*) ...))
- Después de reducir queda

$$suma \equiv_{def} (\lambda n.\lambda m.n\ succ\ m)$$

Definiendo Números

- Ejercicios
  - definir un término para representar la multiplicación
  - definir un término *isNotZero*, que cumpla
    - *isNotZero* $\underline{0} \rightarrow_\beta^* False$
    - *isNotZero* $\underline{n+1} \rightarrow_\beta^* True$
  - definir términos
    - *isZero*, para la función que dice si un número es 0
    - *exp*, para representar la exponenciación
    - *pred*, para representar la función que resta uno (difícil)
    - *resta*, para representar la resta de dos naturales

Definiendo recursión

- Recursión: se utiliza el siguiente 'truco'
  - dada una ecuación recursiva $f = \ldots f \ldots$, definir
    - $f \equiv_{\text{def}} \mathit{fix}\ (\lambda f.\ \ldots f \ldots)$

    siendo *fix* cualaquier λ-término que cumpla

    $$\mathit{fix}\ F \rightarrow_{\beta}^{*} F\ (\mathit{fix}\ F)$$

  - Ejemplo de un término para representar *fix*

    $$\mathit{fix} \equiv_{\text{def}} (\lambda x.\lambda f.f(xxf))(\lambda x.\lambda f.f(xxf))$$

- Por qué funciona es tema para un curso entero

Definiendo recursión

- Ejemplo: sea *fact* un término que cumple que
  - *fact* es equivalente a

    λn. *if* (*isZero* n) *then* *1*
    else mult n (*fact* (*pred* n))
  - Entonces
    - $fact \equiv_{def} fix\ (\lambda f.\lambda n.if\ (isZero\ n)\ then\ \underline{1}\ else\ mult\ n\ (f\ (pred\ n)))$

    y luego de β-reducir
    - *fix* (λf.λn.(*isZero* n) *1* (*mult* n (f (*pred* n))))

Resumen

- Se mostró cómo representar tipos de datos básicos en el λ-cálculo puro
  - bottom
  - booleanos
  - tuplas
  - Maybe
  - listas
  - números naturales
  - recursión