PROGRAMACIÓN FUNCIONAL

Lambda Cálculo: Programación

Lambda Cálculo

- Programando con λ-cálculo
  - Booleanos
  - Pares
  - Números enteros
  - Listas
  - Recursión
  - Bottom

Lambda Cálculo

- ¿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

Lambda Cálculo

- Notación
  - introduciremos nombres para representar expresiones
  - usaremos el símbolo $\equiv_{def}$ para ello
  - 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

Lambda Cálculo

- Booleanos: sea la siguiente especificación
  - *True*, *False* y *ifthenelse* deben ser λ-expresiones en forma normal, tal que para todo par de λ-expresiones P y Q
    - $(\textit{ifthenelse}\ \textit{True}\ \ P\ Q) \rightarrow_\beta^*\ P$
    - $(\textit{ifthenelse}\ \textit{False}\ P\ Q) \rightarrow_\beta^*\ Q$
- Observaciones
  - Lo único que los booleanos deben cumplir es servir para elegir entre dos alternativas
  - La construcción if es representable como una función

Lambda Cálculo

- Entonces buscamos λ-expresiones tal que
  - (*ifthenelse True*) $\rightarrow_{\beta}^{*}$ (λx.λy.x)
  - (*ifthenelse False*) $\rightarrow_{\beta}^{*}$ (λx.λy.y)
- Cualquier grupo de expresiones que cumplan esto sirve
- La solución más simple es:
  - *True* $\equiv_{def}$ (λx.λy.x)
  - *False* $\equiv_{def}$ (λx.λy.y)
  - *ifthenelse* $\equiv_{def}$ (λb.b)
- ¡Observar que *True* y *False* son funciones!
  (¿podía ser de otra forma?)

Lambda Cálculo

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

Lambda Cálculo

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

  y llevándolo a β-forma normal,
  - $\lambda b_1.\lambda b_2.b_1\ b_2\ (\lambda x.\lambda y.y)$

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

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

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

Lambda Cálculo

- 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*

Lambda Cálculo

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

Lambda Cálculo

- Pares: buscamos λ-expresiones que cumplan
  - *pair*, *fst* y *snd*, λ-expresiones en forma normal
  - *fst* (*pair* P Q) $\rightarrow_\beta^*$ P
  - *snd* (*pair* P Q) $\rightarrow_\beta^*$ Q
- Observaciones
  - las ecuaciones son similares a las de booleanos (¡pero no iguales!)
  - la expresión (*pair* P Q) representa a un par de expresiones

Lambda Cálculo

- ¿Cómo podemos usar la similitud de esta especificación con la de los booleanos?
  - *pair* puede ser un *ifthenelse* con un parámetro
  - *fst* y *snd* instanciarían el parámetro adecuadamente
  - Ello nos lleva a la siguiente definición:
    - $pair \equiv_{def} (\lambda xy.\lambda b.\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y)$
    - $fst \equiv_{def} (\lambda p.(p\ \mathit{True}))$
    - $snd \equiv_{def} (\lambda p.(p\ \mathit{False}))$
  - ¡Observar que el par (*pair P Q*) es una función!

Lambda Cálculo

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

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

  - al β-reducir queda (λb. *if* b *then True else and*)
- Ejercicio:
  - construir funciones para tuplas de 3 y 4 elemenos

Lambda Cálculo

- Notación
  - $F^{(0)}M \equiv_{\text{def}} M$
  - $F^{(n+1)}M \equiv_{\text{def}} F^{(n)}(FM)$
- 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 $F^{(n)}M$ es una constante fuera de Λ

Lambda Cálculo

- Números naturales: sea la especificación
  - para cada natural n, $\underline{n}$ una λ-expresión en β-fn
  - $\underline{n}\ F\ M \rightarrow_\beta^* F^{(n)}M$
- O sea:

$$\underline{0}\ F\ M \rightarrow_\beta^* F^{(0)}M \rightarrow_\beta^* M$$
$$\underline{1}\ F\ M \rightarrow_\beta^* F^{(1)}M \rightarrow_\beta^* FM$$
$$\underline{2}\ F\ M \rightarrow_\beta^* F^{(2)}M \rightarrow_\beta^* F(FM)$$
$$\vdots$$
$$\underline{7}\ F\ M \rightarrow_\beta^* F^{(7)}M \rightarrow_\beta^* F(F(F(F(F(F(FM))))))$$
$$\vdots$$

Lambda Cálculo

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

Lambda Cálculo

- ¿Cómo usamos esto para definir cada $\underline{n}$?
  - $\underline{n}$ tiene que tomar a F y a M, entonces tendrá la forma $(\lambda f.\ \lambda x.E)$ para alguna λ-expresión E
  - $\underline{n}\ F\ M$ tiene que reducir a la expresión $F^{(n)}M$, y eligiendo E como $f^{(n)}x$, se consigue
- Ello nos lleva a $\underline{n} \equiv_{def} (\lambda f.\lambda x.f^{(n)}x)$
  - O sea:

$$\underline{0} \equiv_{def} (\lambda f.\lambda x.f^{(0)}x) \equiv_{def} (\lambda f.\lambda x.x)$$
$$\underline{1} \equiv_{def} (\lambda f.\lambda x.f^{(1)}x) \equiv_{def} (\lambda f.\lambda x.fx)$$
$$\underline{2} \equiv_{def} (\lambda f.\lambda x.f^{(2)}x) \equiv_{def} (\lambda f.\lambda x.f(fx))$$
$$\vdots$$

Lambda Cálculo

- ¿Cómo definimos funciones sobre naturales?
  - Utilizamos la propiedad especificada
- Ejemplo:
  - definir un término *succ* para la función sucesor
  - debe cumplir $succ\ \underline{n} \rightarrow_\beta^* \underline{n+1}$
  - O sea $succ\ \underline{n}\ F\ M \rightarrow_\beta^* F^{(n+1)}M \equiv_{def} F^{(n)}(F\ M)$
  - Entonces $succ \equiv_{def} (\lambda n.\lambda f.\lambda x.n\ f\ (f\ x))$

Lambda Cálculo

- Definir un término *suma* para la función suma
  - debe cumplir $suma\ \underline{m}\ \underline{n} \rightarrow_\beta^* \underline{m+n}$
  - Podemos usar *succ* de la siguiente manera
    - $m+n$ es igual a sumar m veces 1 a n (o sea, $succ^{(m)}n$)
    - $\underbrace{succ\ (succ\ (succ\ \ldots\ (succ}_{m\ \text{veces}}\ n)\ \ldots\ ))$
  - O sea $\underline{m+n} \equiv_{def} succ^{(m)}\underline{n} \equiv_{def} \underline{m}\ succ\ \underline{n}$
  - Entonces $suma \equiv_{def} (\lambda m.\lambda n.m\ succ\ n)$

Lambda Cálculo

- Ejercicios
  - definir un término para representar la multiplicación
  - definir un término *isNotZero*, que cumpla
    - $\mathit{isNotZero}\ \underline{0} \rightarrow_\beta^* \mathit{False}$
    - $\mathit{isNotZero}\ \underline{n+1} \rightarrow_\beta^* \mathit{True}$
  - 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

Lambda Cálculo

- ¿Cómo representar listas?
  - puede ser mediante pares
    - la primer componente dice si la lista es vacía o no
    - la segunda componente es un par (primer elemento, resto), donde resto es una lista
    - las funciones sobre listas se definen por recursión
  - puede ser mediante funciones que implementen el patrón de recursión asociado
    - cada `lista' toma dos argumentos, correspondientes al foldr
    - las funciones se definen aplicando la `lista'

Lambda Cálculo

- Listas: sea la especificación
  - *nil* y *cons*, λ-términos en forma normal, cumpliendo
    - $\mathit{nil}\ F\ M \rightarrow_\beta^* M$
    - $(\mathit{cons}\ X\ L)\ F\ M \rightarrow_\beta^* F\ X\ (L\ F\ M)$
- Observaciones
  - las listas son funciones
  - representan el patrón de recursión '*diferido*'
  - podría definirse *foldr* $\equiv_{\text{def}}$ (λf.λa.λl. l f a)

Lambda Cálculo

- La solución más sencilla es
  - $nil \equiv_{def} \lambda f.\lambda a.a$
  - $cons \equiv_{def} \lambda x.\lambda l.(\lambda f.\lambda a.f\ x\ (l\ f\ a))$
- Ejemplos:
  - [2,3] se representa como (*cons* $\underline{2}$ (*cons* $\underline{3}$ *nil*)), que luego de β-reducir queda ($\lambda f.\lambda a.f\ \underline{2}\ (f\ \underline{3}\ a)$)
  - [v,w,x,y,z] se representa como (*cons* v (*cons* w (*cons* x (*cons* y (*cons* z *nil*))))), que luego de β-reducir queda $\lambda f.\lambda a.f\ v\ (f\ w\ (f\ x\ (f\ y\ (f\ z\ a))))$

Lambda Cálculo

- ¿Cómo definir funciones sobre listas?
  - Utilizando el patrón de recursión
- Ejemplos
  - dar un λ-término para representar *length*
    - $length \equiv_{def} \lambda l.\ l\ (\lambda x.succ)\ \underline{0}$
  - dar un λ-término para representar *sum*
    - $sum \equiv_{def} \lambda l.\ l\ (\lambda x\ y.suma\ x\ y)\ \underline{0}$
  - dar un λ-término para representar *map*
    - $map \equiv_{def} \lambda l.\ \lambda f.\ l\ (\lambda x\ y.\ cons\ (f\ x)\ y)\ nil$

Lambda Cálculo

- 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*
    - $\mathit{fix} \equiv_{\text{def}} (\lambda y.yy)(\lambda x.\lambda f.f(xxf))$
- Por qué funciona es tema para un curso entero

Lambda Cálculo

- 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
    - $fix\ (\lambda f.\lambda n.(isZero\ n)\ \underline{1}\ (mult\ n\ (f\ (pred\ n))))$

Lambda Cálculo

- Bottom: ¿Cómo definir un término para $\perp$?
  - Mediante alguna expresión cuya computación no termine
- Solución
  - $bottom \equiv_{def} (\lambda x.xx)(\lambda x.xx)$
  - Es fácil ver que la computación de *bottom* no termina ($bottom \rightarrow_\beta^* bottom \rightarrow_\beta^* \ldots$)
  - Sirve para hacer explícita la indefinición
    - Ej: $hd \equiv_{def} (\lambda l.l(\lambda xy.x)bottom)$

Resumen

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