FUNDAMENTOS DE COMPUTACIÓN
Repartido: Introducción y Funciones

1. Introducción: presentación del curso

Este es un curso de programación. Vamos a usar un lenguaje muy cercano a la matemática, en el cual básicamente escribimos ecuaciones.
Es muy pequeño (o sea, con pocas construcciones) y es igual de poderoso que cualquier lenguaje que conozcan (javascript, java, C,...): **Haskell**, que es un lenguaje del **paradigma funcional**.
Vamos aprender a programar usando ese paradigma y sobre todo aprender técnicas de programación (principalmente recursión) que sirven para cualquier otro lenguaje.

Pero también es un curso de matemática, porque vamos a enfocarnos en hacer demostraciones de propiedades sobre nuestros programas.
O sea, en vez de tener como objeto de estudio a los reales o las funciones (como en Cálculo), vamos a estudiar a los programas que escribamos.
En este marco vamos a ver técnicas de demostración (principalmente inducción).

2. Objetos, tipos, expresiones

En el mundo real, existen objetos, con los cuales trabajamos naturalmente, y los cuales a veces comparten ciertas características, por ejemplo, existen objetos matemáticos como números, que combinamos con operaciones +,-,*, o funciones *cos, tg,* $f(x)=x^2+3*x+1$.
Como pueden, notar estas son en realidad representaciones escritas de los objetos con símbolos del lenguaje de la aritmética y el cálculo.
Eso mismo pasa con cualquier otro objeto del mundo real que queramos representar, tendremos que tener alguna forma de expresarlo.

Pero para completar estas representaciones es necesario un ingrediente fundamental… los tipos.
Las expresiones sin tipo son ambiguas, por ejemplo: 101+1 = ¿102 o 110?
El resultado es distinto si los consideramos números decimales o binarios.
¡En matemática y en programación no podemos darnos el lujo de tener ambigüedades!
La forma de solucionar esto es asignarle un tipo a cada expresión.
De esta forma 101+1::Binario, se refiere al número 110 (6 en el sistema decimal), mientras que 101+1::Natural, se refiere al número 102.

3. Elementos básicos del lenguaje de programación: Variables, constantes, funciones

Haskell está basado en el Cálculo λ, que es un lenguaje matemático formal, para escribir funciones computables, es Turing Completo (por tanto, cualquier cosa que sea computable, se puede programar en Haskell), y tiene una sintaxis simple compuesta por: variables constantes y funciones .

Por ejemplo: en la expresión $3*x+2^y$, el 3 y el 2 son constantes, x e y son variables, * y + son funciones (operaciones).
En nuestro lenguaje, tenemos que asignarles un tipo, dando como resultado algo del estilo a True :: Bool, π :: R, 3*x+2 :: N o 4x>x :: Bool.

4. Juicios, Juicios hipotéticos

Lo que escribimos arriba son las afirmaciones que podemos hacer respecto a las expresiones que escribimos. Estas afirmaciones se llaman juicios del lenguaje , y pueden ser de 3 formas:

- t es un tipo
- e :: t (la expresión e tiene tipo t)
- e1 = e2 (ambas expresiones son sinónimos)
  {el juicio de igualdad se usa para poder hacer definiciones, como not True = False, o $x^{n+1} = x^n * x$*, y sirve también para tipos, ya que podemos definir strings como listas de caracteres, lo cual se escribe String = [Char]}*

Ahora, en nuestro lenguaje tenemos varios tipos... ¿qué pasaría si sustituimos la x, por True, o por un string? 3*“Hola”+2, claramente no tiene tipo N.
Para esto debemos darle tipo a las variables que aparecen en un juicio.

Entonces definimos lo que es un **contexto de (declaración de) variables**, y en los juicios en los que aparecen variables tenemos la obligación de declararlas.

Por ejemplo: 3*x+2 ::N [x ::N].
Esto se lee, “3*x+2 tiene tipo N, siempre y cuando x tenga tipo N”
Lo mismo pasa con la otra expresión del ejemplo: 4*x > x :: Bool [x :: N], que se lee:
“4* x > x tiene tipo Bool, siempre y cuando x tenga tipo N”.

Estos juicios con declaración de variables se llaman también **juicios hipotéticos**, ya que los contextos son hipótesis que asumimos.

5. Reglas para asignar tipos: variables, funciones y abstracciones

Para todos los juicios del lenguaje tenemos reglas, que nos permiten: definir tipos, definir expresiones (y asignarles tipo) y declarar igualdades entre expresiones o tipos.
Las reglas se presentarán del siguiente modo:

P1 … Pn , donde Pi son juicios que son premisas (lo que suponemos) y C la conclusión
C

Si n=0 la regla se llamará Axioma porque no hay condiciones.

a) Variables

Para **asignar tipo a las variables** existe el axioma de la variable, que básicamente dice que una variable tiene el tipo con que fué definida.

(VAR)_____ *{podría tener como premisa también t tipo}*
x::t [x::t]

b) Funciones: tenemos varios conceptos para definir.

b1. Tipo

Primero definimos el tipo de las funciones, de igual modo que en la matemática (sen, cos):

(FUN) s tipo t tipo
$s \rightarrow t$ tipo

{s es el dominio, t es el codominio}

b2. Aplicación

¿Y para qué sirven las funciones?
Para aplicarlas a sus argumentos, siempre que sean del tipo del dominio de la misma, y obtener un objeto en el codominio de la misma, o sea:

(APL) $f :: s \rightarrow t \quad e :: s$
f e :: t
{la expresión f e denota la aplicación de la función f a la expresión e, o sea, lo que usualmente escribimos f(e), pero en este lenguaje no ponemos paréntesis}

b3. Abstracción

Para ver la forma de construir construir funciones, vamos a ver un ejemplo matemático:

¿Qué podemos decir de la expresión $x^2+3*x+1$?
Según lo que dijimos hasta ahora, lo que tenemos es un real, siempre que x sea un real, o sea: $x^2+3*x+1 :: R$ [x::R].
Pero cuando escribimos $x^2+3*x+1$ quizás queremos referirnos a una parábola…

¿Cómo hacemos que eso sea una función matemáticamente hablando?
En los libros de matemática escriben $f(x) = x^2+3*x+1 :: R \rightarrow R$ y con eso ya hacen referencia a la parábola.
Pero si estamos programando y aparece la expresión $x^2+3*x+1$ no podemos permitir ninguna ambigüedad. O sea, $x^2+3*x+1$ debe ser un real o una función.

En 1936 el matemático Alonzo Church introdujo la notación Lambda (λ), para definir funciones. Escribimos λx→ x^2+3*x+1 para indicar que la expresión es una función de x, y no un real que depende de una x para declarar.
Ahora esta expresión tiene tipo R → R , o sea, es una función que espera que le pasen un real como argumento para devolver un real: λx→x^2+3*x+1 :: R → R.

Esto no es nuevo, cuando en matemática escribimos una integral como $\int x+y,$ debemos aclarar respecto de qué variable estamos integrando, o sea, debemos decir si *x+y* es una función de *x* o de *y*… Eso lo hacemos con el *dx* o el *dy*.
En este caso es lo mismo, podríamos escribir $\int x+y\ dx,$ o $\int \lambda x \rightarrow x+y$.

¿Cómo justificamos ese juicio? Con la regla de la abstracción funcional: si tenemos que x^2+3*x+1 :: R [x::R], para convertir ese *real* que depende de la declaración de la variable x, en una *parábola (función)*, transformamos a esa variable que está declarada en el *parámetro* de la función y armamos una **abstracción funcional** , obteniendo la siguiente expresión: λx→ x^2+3*x+1 :: R → R.
La variable x se llama el **parámetro** de la función, y la expresión x^2+3*x+1 es el **cuerpo**.
La regla entonces será la siguiente:

(ABS) e :: t [x :: s] .
λx→ e :: s → t

{observar que cuando escribimos λx→ e es como escribir f(x) = e, pero sin necesidad de darle un nombre}

Con esta regla pasamos de un juicio hipotético (arriba) a uno que ya no tiene declaraciones de variables (abajo) porque el tipo de x queda declarado en el dominio de la función.

b4. Sustitución, Igualdad beta

Ahora, ¿qué pasa cuando queremos ver el resultado de la función λx→ x^2+3*x+1 en el valor 4? (o sea, f(4) como se escribe en matemática).

Primero: ¿qué tipo tiene f(4)?
Es un real, ya que nuestra función es de tipo R → R y 4 ::R, por lo que usando la regla de la aplicación (APL) nos queda (λx→ x^2+3*x+1) 4 :: R.

Segundo: la regla (APL) nos dice qué tipo tiene la aplicación, pero ahora queremos saber cuál es el resultado de dicha aplicación...
Como ya sabemos de la matemática, con la notación de los libros, si f(x) = x^2+3*x+1 y aplicamos f a 4, el resultado es f(4) = 4^2 +3*4+1. O sea, se sustituye el *parámetro* de la función (x en este caso) en el *cuerpo* de la función (x^2+3*x+1 en este caso) por el *valor* indicado (4 en este caso).

Esta operación se llama **sustitución** , y la vamos a notar del siguiente modo:
x^2+3*x+1[x:=4], o sea **e[x:=a]** es la expresión que se obtiene *reemplazando todas las x que hay en e por la expresión a*.

Entonces definimos la siguiente igualdad: **(λx→ e) a =$_{\beta}$ e[x := a]**, y la llamamos **igualdad β**.

Una expresión de la forma (λx→ e) a se llama **expresión reducible** o **redex** (que viene de *reducible expresión,* en inglés), y escribimos (λx→ e) a →$_{\beta}$ e [x := a] (o sea, usamos una flecha) cuando queremos indicar la dirección en que se computa.

Computar es entonces tratar de sacar todos los redexes de una expresión. Una expresión sin redexes se llama **forma normal**. En Haskell la regla de cómputo que se utiliza es justamente la reducción β (de izquierda a derecha) hasta que no haya más redexes, o sea, hasta llegar a una *forma normal*.

Ejemplos de reducciones: (subrayamos los redexes que reducimos)

- (λz→ z+y) 5 →$_{\beta}$ 5+y
- (λw→ w^2 +2*w - x + y) ((λz→ z) 2) →$_{\beta}$ (λw→ w^2 +2*w - x + y) 2 →$_{\beta}$ 2^2 +2*2 - x + y
- ((λf→ (λx→ f x)) (λy→ y+y)) 2 →$_{\beta}$ ((λx→ (λy→ y+y) x)) 2 →$_{\beta}$ (λx→ x+x) 2 →$_{\beta}$ 2+2

b5. Variables libres y ligadas, igualdad alfa (cambio de variable)

Dijimos que λx→ x+1 es una función de tipo R → R, que representa una recta.
¿Qué pasa con las siguientes funciones: λy→ y+1 , λz→ z+1 , λw→ w+1 ?

Claramente son la misma función, la variable que se abstrae (parámetro) no importa. Observar que es lo mismo que un parámetro de un método de cualquier lenguaje de programación.

Esto justifica la siguiente igualdad: λx→ e = **α** λy→ (e[x := y]), o sea, se puede hacer un **cambio de variable** , sin cambiar el significado de una expresión. Esta igualdad se llama α.

Ahora, ¿podemos aplicar la igualdad α en cualquier lado?
Si tenemos λx→ x+z :: R → R [z::R],ya sabemos que λx→ x+z = **α** λy→ y+z, pero:
¿se cumple λx→ x+z = **α** λx→ x+w?

Claramente no son la misma expresión, ya que z y w no son parámetros de la función, sino que son variables que deben ser declaradas y pueden tomar distintos valores.

Entonces tenemos que diferenciar si una variable aparece como parámetro en una expresión (eso es, dentro de un λ) o no.
Llamamos a una ocurrencia de una variable **ligada** , si ésta aparece dentro del alcance de un λ. Si una ocurrencia no está ligada, entonces decimos que está **libre**.

Con este nuevo concepto podemos revisar la sustitución , y decir que e[x:=a] es la expresión que se obtiene *reemplazando todas las ocurrencias libres de x que hay en e por la expresión a*.

Este concepto sirve también para evitar situaciones de este tipo: λx→ x+z[z:=x] = λx→ x+x, lo cual no es correcto porque la x que está ahora en lugar de la z queda ligada cuando antes la z no lo estaba…
A este fenómeno se lo llama **captura de variable** y no es deseable, por lo que si queremos hacer esa sustitución, lo que hacemos primero es un cambio de variable (α), y así tenemos, por ejemplo: λx→ x+z[z:=x] = α λw→ w+z[z:=x], y ahora podemos sustituir la z por x sin peligro, obteniendo λw→ w+z[z:=x] = λw.w+x.

b6. Currificación

Consideremos el siguiente juicio: z - x :: R [x::R , z::R].

Podemos aplicar la regla ABS y obtener: λx→ z - x :: R → R [z::R].
Si aplicamos otra vez ABS obtenemos f = λz→ (λx→ z - x) :: R → (R → R).

¿Qué hace esta función?
Espera un z para devolver una función que espera un x y se lo resta.
O sea: f 5 = (λz→ (λx→ z - x)) 5 =$_\beta$ λx→ 5 - x :: R → R.
Y a su vez, (f 5) 2 = (λx→ 5 - x) 2 =$_\beta$ 5 - 2::R.

¿Qué diferencia hay entre esta función de Haskell y la función g(z,x) = z - x de matemática?

- f nos provee de dos funciones: una es f = λz→ (λx→ z - x) :: R → (R → R), y si le aplicamos f a cualquier real r , tenemos otra función f r = (λx→ r - x) :: R → R. Por ejemplo, f aplicada a 0, nos dará como resultado la función que calcula el opuesto aditivo de un número real, ya que f 0 = λx→ 0 - x
- g, en cambio, siempre tiene que aplicarse a dos reales juntos, r y s , pero es una sola función.

Claramente con f podemos hacer lo mismo que con g, pero con g no podemos hacer lo mismo que con f, ya que no se la podemos aplicar a un solo argumento.
El poder de Haskell y los lenguajes funcionales es justamente que una función pueda servir para muchas cosas, dependiendo de a cuántos argumentos se aplica.

A esta forma de escribir funciones se la llama **Currificación** en honor al matemático *Haskell B. Curry*, que hizo aportes fundamentales al Cálculo λ.

b7. Parentización (convenciones sintácticas para evitar escribir paréntesis)

Entonces, con las funciones currificadas, no vamos a tener necesidad de estar construyendo n-uplas (pares, ternas, etc.), ya que en el caso general partiremos de expresiones de la forma:

$$e :: t_e\ [z::t_z\ ,y::t_y\ ,x::t_x\ ,w::t_w\ ]$$

y aplicando la regla de abstracción, iremos obteniendo las siguientes expresiones:

$$\text{(ABS) } \underline{e :: t_e\ [z::t_z\ ,y::t_y\ ,x::t_x\ ,w::t_w\ ]}$$
$$\text{(ABS) } \underline{(\lambda z\rightarrow e) ::t_z \rightarrow t_e\ [y::t_y\ ,x::t_x\ ,w::t_w\ ]}$$
$$\text{(ABS) } \underline{\lambda y\rightarrow (\lambda z\rightarrow e) :: t_y \rightarrow (t_z \rightarrow t_e\ )\ [x::t_x\ ,w::t_w\ ]}$$
$$\text{(ABS) } \underline{\lambda x\rightarrow (\lambda y\rightarrow (\lambda z\rightarrow e)) :: t_x \rightarrow (t_y \rightarrow (t_z \rightarrow t_e\ ))\ [w::t_w\ ]}$$
$$\text{(ABS) } \lambda w \rightarrow (\lambda x\rightarrow (\lambda y\rightarrow (\lambda z\rightarrow e))) :: t_w \rightarrow (t_x \rightarrow (t_y \rightarrow (t_z \rightarrow t_e\ )))$$

Para evitar escribir tantos paréntesis, imponemos las siguientes convenciones sintácticas:

1) Las abstracciones asocian a derecha.
O sea, los λ se expanden lo máximo posible hacia la derecha, abarcando todo lo que pueden. De esa forma, podemos evitar los paréntesis en la expresión de arriba y escribir λw→ λx→ λy→ λz→ e.

{Método para recuperar los paréntesis que sacamos: cada vez que vemos un λ, ponemos un "(" a su izquierda y lo cerramos lo más lejos posible (sin cruzar paréntesis)}.

2) La flecha de los tipos asocia a derecha.
De esta forma el tipo $t_w \rightarrow (t_x \rightarrow (t_y \rightarrow (t_z \rightarrow t_e\ )))$ se escribe $t_w \rightarrow t_x \rightarrow t_y \rightarrow t_z \rightarrow t_e$, sin paréntesis.

Finalmente, si tenemos la función

$$f = \lambda w\rightarrow \lambda x\rightarrow \lambda y\rightarrow \lambda z\rightarrow e :: t_w\rightarrow\ t_x \rightarrow t_y \rightarrow t_z \rightarrow t_e$$

se la podemos aplicar primero a un w_0 de tipo t_w y obtendremos:

$$f\ w_0 =_\beta \lambda x\rightarrow \lambda y\rightarrow \lambda z\rightarrow e[w:= w_0] ::\ t_x \rightarrow t_y \rightarrow t_z \rightarrow t_e.$$

Luego podemos aplicar esta nueva función a un x_0 de tipo t_x, y obtener:

$$(f\ w_0)\ x_0 = \lambda y\rightarrow \lambda z\rightarrow e[w:= w_0][x:= x_0]$$

Finalmente aplicamos esta función a un $y_0 ::t_y$ y a un $z_0 ::t_z$ y obtenemos:

$$(((f\ w_0)\ x_0)\ y_0)\ z_0 = e[w:= w_0][x:= x_0][y:= y_0][z:= z_0]$$

Esto justifica la última convención sintáctica: **3) La aplicación asocia a izquierda.**

O sea, escribimos $\mathbf{f\ w_0\ x_0\ y_0\ z_0}$ en vez de $(((f\ w_0)\ x_0)\ y_0)\ z_0$.

Ejemplo: consideremos la siguiente secuencia de reducciones
$(\lambda x\rightarrow \lambda y\rightarrow x^2 - y + x\ .\ y\ )\ 3\ 5 = ((\lambda x\rightarrow (\lambda y\rightarrow x^2 - y + x\ .\ y\ ))\ 3)\ 5 \rightarrow_\beta (\lambda y\rightarrow 3^2 - y + 3\ .\ y\ )\ 5 \rightarrow_\beta 3^2 - 5 + 3\ .\ 5$

Esto se puede hacer directamente, considerando que las aplicaciones se hacen de izquierda a derecha en el mismo orden en que aparecen los λ, y no de afuera hacia adentro. O sea, el 3 se corresponde con el x y el 5 con el y.