χ

Modelo funcional de computación
Motivación y sintaxis

0. Introducción. Estudiaremos modelos matemáticos del fenómeno de la computación –es decir, de máquinas o sistemas que computan. Comenzamos por un modelo basado en una idea muy sencilla y ya conocida por nosotros; es la misma que fundamenta al lenguaje Haskell: *Computar es evaluar expresiones.*

En lugar de la palabra "expresión" podríamos usar también "fórmula", lo cual nos remite a una actividad cotidiana de la Matemática y sus aplicaciones –por ejemplo, en ciencia. La noción de máquina que uno asocia a esta idea es la *calculadora* (en nuestro caso, programable). La idea es en realidad tan simple y conocida que el lector podría inclusive estar dudando acerca de la existencia de *otros* modelos. Bueno, considérense los programas en C o Java: ellos no son fórmulas destinadas a producir un *valor* cuando son alimentados con datos de entrada. Más bien, ellos computan actuando sobre una *memoria*, i.e. sobre un dispositivo que contiene datos que pueden ser modificados. Se dice que los datos presentes en la memoria determinan el *estado* de ésta, y que un programa de la especie C/Java computa por la vía de *modificar (paso a paso) el estado de la memoria*.

La cuestión presente es entonces realizar una construcción matemática que represente de manera apropiada la computación "por evaluación de expresiones o fórmulas". Nuestro propósito es luego demostrar propiedades de ese modelo, propiedades que serán interesantes porque valdrán para toda variante o instancia que mantenga sus características esenciales. Para ser un poco más concretos, digamos que nuestro modelo tomará la forma de un lenguaje de programación funcional —y las propiedades en las que estamos interesados serán trasladables, al menos en la mayoría de los casos, a todo otro lenguaje funcional. Ahora bien, el modelo a construir no será un lenguaje destinado al *uso* en la construcción efectiva de software, sino a su estudio como objeto matemático. Esto motiva a que sea lo más simple posible, de hecho, mínimo.

¿Cuál es el sistema mínimo de conceptos que necesitamos para hablar de *evaluación de expresiones*? Bueno, obviamente debe haber *expresiones*. Algunas de éstas deben ser *valores*, los resultados de la evaluación. Y, finalmente, deben existir *reglas* que determinen cómo se realiza la evaluación de cada expresión. Todo esto es muy directo y obvio y, en efecto, nos vamos a ceñir a este simple análisis, armando nuestro modelo en base a esas tres ideas de base. Un tanto más de reflexión es, sin embargo, apropiada: Debemos considerar el concepto de *función*. Primeramente, necesitamos funciones, como ya es sabido, para poder tener una cantidad infinita de valores, cosa necesaria si nuestro modelo va a ser capaz de expresar y realizar computaciones con e.g. números. Los valores se generarán usando *constructores* que en general son funciones, como por ejemplo S (la función sucesor). Por otro lado, ¿cómo es, exactamente, que surge la necesidad o conveniencia de computar (evaluar) expresiones? ¿Por qué no es suficiente con tener valores? La noción de valor ya es conocida: se trata de una expresión cuyo significado debe ser captable por directa lectura. ¿Cuándo es que surge la necesidad de considerar expresiones que no sean valores? La respuesta es: al concebir *operaciones*, es decir, funciones, que permitan transformar unos valores en otros. Por ejemplo, en algún momento nos ha interesado transformar cada número en su factorial. O en un booleano que indica la paridad del número. Cada una de estas transformaciones se expresa como una función que genéricamente toma la forma de una expresión lambda. La *aplicación* de cada una de estas funciones a datos de entrada es lo que constituye la clase más usual de "expresión a ser evaluada", i.e. el origen de la computación. Dadas estas consideraciones, estamos listos para diseñar nuestro modelo, un modelo *funcional* de la computación.

1. Los elementos del modelo.

Funciones. Comenzamos introduciendo las formas de expresión necesarias. Lo esencial resulta tener funciones, en sus dos formas: expresiones lambda

$\lambda \bar{x}.e$

y constructores

c

que serán *nombres* simples (constantes o identificadores).

Las funciones se *aplican* a uno o varios argumentos (dependiendo de la cantidad de parámetros que esta recibe). En materias anteriores como Fundamentos y Lógica, hemos probado que la aplicación de una función que espera 3 argumentos se puede representar como 3 aplicaciones anidadas. Sin embargo, con el fin de simplificar la notación (y posteriormente la ejecución) de estas estructuras diremos que una aplicación se realizará entre una expresión y una lista de expresiones como parámetro separadas por un espacio. Esta última podría contener una o más expresiones.
Con las condiciones mencionadas anteriormente, la aplicación de una función f a 3 argumentos (a_1,a_2,a_3) se escribiría de la siguiente manera:
f [a_1,a_2,a_3]
Otra consecuencia de estas condiciones será que las expresiones *lambda* (para acompañar la construcción de las aplicaciones) tendrán una lista de variables de entrada escribiéndose como $\lambda[x_1,...,x_n]$.e
De esta manera se podrá expresar la aplicación de una abstración funcional con 3 variables y 3 argumentos de la siguiente forma:
($\lambda[x, y, z]$.e) [a_1,a_2,a_3]
Un lector minucioso podrá identificar a simple vista que esta notación podría traer problemas si la cantidad de variables de la primera expresión y la lista de la argumentos no coincide, pero estos problemas serán discutidos posteriormente cuando hablemos de cómo se ejecutará un programa en este lenguaje.

(Ausencia de) Tipos. Un concepto fundamental en Haskell que no hemos mencionado todavía en el presente contexto es el de *tipo*. En Haskell, los valores vienen con tipo —más aún, la introducción del tipo mismo es simultánea a la de sus valores. Y, por el otro lado, las funciones tienen asociado un tipo de dato de entrada y uno de salida, de modo que quedan restringidas sus aplicaciones válidas. Bueno, una característica fundamental de nuestro modelo χ es que es *sin tipos*. La razón es que el concepto de tipo puede obviarse si lo que estamos describiendo es la pura computación: En efecto, en el propio Haskell, la computación consiste en aplicar definiciones y éstas son sólo transformaciones o reescrituras de unas expresiones en otras, sin que en el proceso intervenga de ningún modo la información de tipo de las expresiones en cuestión. También existe esta otra razón: nada va a impedir al programador χ mantener por sí mismo la disciplina de tipo; se sigue que todo lo programable en Haskell lo será también en χ. De modo que no perdemos capacidad expresiva al omitir el control de tipo. Esto es fundamental si queremos investigar cuál es el poder expresivo de nuestro modelo, es decir, qué es lo que se puede programar en él. El resultado recíproco no es, como mínimo, inmediato: no tenemos en principio certeza de que todo algoritmo programable en el sistema sin tipos lo sea también en el que tiene control de tipo.

Todo el asunto precedente deja ver un punto en que *modelo de computación* y *lenguaje de programación* difieren fuertemente. En efecto, la noción de tipo es fundamental como estructura lógica, es decir, del propio pensamiento. Esto quiere decir que la vamos a emplear en el diseño de programas, aún si programamos en χ. En otras palabras, la programación en χ, puede decirse, va a respetar implícitamente la disciplina de tipos —sólo que sin contar con el auxilio del lenguaje, es decir del compilador. El hecho de que uno pueda omitir la información de tipo y seguir produciendo los mismos programas revela que la información de tipo es *redundante*. En efecto, la declaración de tipo de una función es totalmente redundante para la computación, que sólo necesita el código ejecutable. Ahora bien, esa redundancia permite al compilador detectar incongruencias entre lo declarado y lo efectivamente programado —lo cual deja ver errores de programación y aún de concepción o diseño. Atributos como éste, que facilitan la detección temprana de errores, son fundamentales en los lenguajes de programación. Pero no en un modelo de computación cuyo objeto está en representar *todo* lo programable siendo a la vez *mínimo*.

Como consecuencia de la ausencia de control de tipo, resulta que serán válidas expresiones como $S\,True$, o inclusive $S\,S$, $S\,S\,0$ y $True\,0$. En general. lo que va a ocurrir es que no existirá restricción a la aplicación funcional: cualquier función se podrá aplicar a cualquier número de argumentos cualesquiera —y, para reiterarlo, debe ser el programador quien supla con su propia disciplina la ausencia de control del compilador.

Expresiones `case`. Siguiendo con el diseño de χ, tendremos expresiones `case`, con el mismo sentido que en Haskell. Cada rama de un `case` estará formada por un constructor y una expresión. Igual que en

Haskell, al evaluar un `case` su expresión principal (o discriminante) será evaluada. Si su valor final es de la forma $c\,e_1\,e_2\,\ldots\,e_n$ entonces se buscará la rama rotulada con el constructor c y la expresión será aplicada a $e_1, e_2, \ldots, e_n$. Véase entonces que esta versión de `case` no es idéntica a la de Haskell, pero es similar. La manera de expresar la parte izquierda de una rama será un constructor con una lista de identificadores (no un patrón como se puede realizar en Haskell). De esta manera una rama de χ se representaría así:

$c\ \bar{x}\ \rightarrow\ e$.

Recursión (sin definiciones). En χ *no existirán definiciones.* Si pensáramos en el modo de uso de este lenguaje, él consistiría en escribir expresiones que, en caso de ser sintácticamente correctas, serían directamente evaluadas. Por ejemplo podemos considerar

$\lambda[b].$ `case` b `of` $\{\ \mathit{False}\ \rightarrow\ \mathit{True};\ \mathit{True}\ \rightarrow\ \mathit{False}\}$.

Esta expresión es sintácticamente correcta y corresponde a la negación booleana. Una función (expresión lambda o constructor) es de por sí un valor y por lo tanto esta expresión, al ser evaluada, se tiene a ella misma como resultado. Uno puede además usarla como en:

$(\lambda[b].$ `case` b `of` $\{\ \mathit{False}\ \rightarrow\ \mathit{True};\ \mathit{True}\ \rightarrow\ \mathit{False}\})\ [\mathit{False}]$,

que también es sintácticamente correcta y, evaluada, da como resultado *True*.

Para no obligarnos a reescrituras extensas, nos autorizamos a darle nombre a cualquier expresión. Ese acto, sin embargo, no se realiza en el lenguaje χ, sino en el castellano en el que estamos hablando en este momento, es decir, en el metalenguaje. Nuestros nombres oficiarán de simples "macro" definiciones. Cada nombre podrá ser sustituido (expandido) por la expresión que él representa, textualmente. Por ejemplo, podemos decir ahora mismo

not $= \lambda[b].$ `case` b `of` $\{\ \mathit{False}\ \rightarrow\ \mathit{True};\ \mathit{True}\ \rightarrow\ \mathit{False}\}$.

Hecho esto podemos referirnos al código a la derecha de $=$ mediante el nombre not. En principio esto solamente sería posible en nuestro texto castellano, pero para permitir aún mayor ahorro expresivo autorizamos también a usar estos nombres (llamémoslos "macros") dentro de expresiones χ. Por ejemplo:

not *False*.

En rigor, ésta es una expresión híbrida de metalenguaje y χ. Por supuesto, textualmente no es válida en este último. Más bien debe entenderse que *representa* la expresión que se obtiene al eliminar (expandir) la macro not. En suma, una expresión con macros es una expresión del metalenguaje que representa cierta expresión de χ. Para conocer ésta deben eliminarse (expandirse) todas las macros.

Ahora bien, eliminar toda macro no es posible si ellas se usan recursivamente, porque la expansión no se detiene nunca. Considérese por ejemplo

par $= \lambda[n].$ `case` n `of` $\{\ 0\ \rightarrow\ \mathit{True};\ S\,x\ \rightarrow\ \ \mathrm{not}(\mathrm{par}\ x)\ \}$.

En una aplicación de par, para poder llegar a conocer la expresión sintácticamente correcta en χ que corresponde a ésta, debemos eliminar las macros. Ahora bien, eso es posible con not, pero la eliminación de par es imposible. La solución simple es no autorizar ningún tipo de recursión en el uso de macros, lo cual quiere decir que, al introducir una macro, el código a la derecha de $=$ debe ser ya expandible totalmente a código χ puro. Pero entonces, ¿cómo podremos tener funciones recursivas en χ, como por ejemplo par que determina si un natural dado es o no par?

Bueno, podemos representar la ecuación recursiva que define la función *par* dada arriba por medio de una *expresión* del lenguaje que indique que se está usando el nombre *par* recursivamente. Concretamente, escribiremos:

`rec`.*par* $(\lambda[n].$ `case` n `of` $\{\ 0\ \rightarrow\ \mathit{True};\ S\,x\ \rightarrow\ \ \mathrm{not}(\mathit{par}\ x)\ \})$.

Debe notarse, en primer lugar, que *par* ya no es un nombre de macro del metalenguaje sino un identificador (una variable, de hecho) del lenguaje χ.

Pero, obviamente, debemos además dar a este operador `rec` que acabamos de introducir, una semántica acorde con nuestras intenciones. Veamos esto informalmente:

La forma general de este tipo de expresiones es

`rec` $f.\,e$

donde f es el nombre usado recursivamente en el cuerpo e. La idea es que esto represente la función f que, en Haskell hubiéramos definido recursivamente mediante

$f = e$.

Entonces, para evaluar

$\texttt{rec}\, f.\, e$

debemos pensar que esta expresión representa a f definida como $f = e$. Por lo tanto, por un lado, es equivalente a e. Pero además, ocurre que el identificador f puede aparecer en esta e y allí debe volver a ser reinterpretado como $\texttt{rec} f.\, e$. Esto se lograría si nuestras reglas semánticas forzaran a que el valor de $\texttt{rec} f.\, e$ sea idéntico al de e donde el nombre f se sustituye por $\texttt{rec}\, f.\, e$, lo cual notaremos mediante

$e[f := (\texttt{rec}\, f.\, e)]$.

2. Sintaxis.

Sintaxis concreta. La siguiente definición BNF establece una sintaxis concreta de las *expresiones* e de χ

$$e ::= x \mid C \mid \lambda \bar{x}.e \mid e_1\, \bar{e} \mid \texttt{case}\ e\ \texttt{of}\ \{\ \overline{b}\ \} \mid \texttt{rec}\ x.\ e$$

La única notación que necesita explicación adicional es la correspondiente a las ramas de las expresiones `case`. El suprarrayado en general significará *lista de*. Por lo tanto, $\overline{b}$ significa: 0 o más elementos de la categoría b. Ésta, a su vez, se define como sigue:

$b ::= c\, \bar{x} \ \rightarrow\ e$.

Es decir, una rama de `case` es una pareja formada por un constructor y una lista de variables, y una expresión, separadas por flecha.

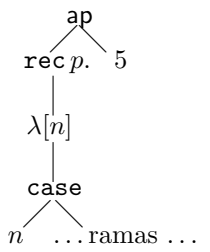
Quedan detalles por especificar:

1. Las ramas dentro de un `case` se separan entre sí por `;`.
2. Se usan paréntesis para marcar el orden de precedencia en aplicaciones sucesivas.
3. Pero, como es habitual en los lenguajes funcionales, se pueden ahorrar paréntesis aplicando la siguiente convención:
   (a) La aplicación tiene prioridad sobre la abstracción funcional (o, equivalentemente, el alcance de cada abstracción sólo está limitado por paréntesis que la encierren en forma completa).
   (b) En una sucesión de aplicaciones no parentizada, es decir, $e_1\, e_2\, e_3\, \ldots\, e_n$, las aplicaciones se toman de izquierda a derecha.

Sintaxis abstracta. Las explicaciones precedentes determinan cuáles son las *secuencias* de símbolos (strings) válidas en χ. Esta especificación es, por supuesto, indispensable, teniendo en cuenta que, ya sea usando un teclado o lápiz y papel, nuestra escritura normal procede en forma secuencial, y debemos conocer las reglas para producir las secuencias correctas en el lenguaje que estamos estudiando. Sin embargo, la sintaxis concreta no es el dispositivo más conveniente para pensar, es decir, *razonar* sobre las expresiones, lo cual incluye diseñar algoritmos útiles tales como los de traducción (compilación) e interpretación. Por el contrario, es sabido que las expresiones (de todo lenguaje, en forma general) se entienden naturalmente en forma de *árbol*. Disponer una expresión en forma de árbol expone su estructura en términos de los elementos con los que ha sido construida, e ignora un número de detalles irrelevantes a estos efectos. Por ejemplo, al leer

$(\texttt{rec}\, p.\ (\lambda[n].\ \texttt{case}\ n\ \texttt{of}\ \{\ 0\ \rightarrow\ \mathit{True};\ S\, x\ \rightarrow\ \ \mathrm{not}(p\ x)\ \}))\ 5$

lo que nos interesa es entender que se trata de una aplicación entre el operador `rec` y el número 5. A su vez la expresión `rec` *liga* la variable p que se encuentra dentro de su cuerpo (siendo este la expresión: $(\lambda[n].\ \texttt{case}\ n\ \texttt{of}\ \{\ 0\ \rightarrow\ \mathit{True};\ S\, x\ \rightarrow\ \ \mathrm{not}(p\ x)\ \})$). Este cuerpo, a su vez es una *abstracción funcional*, donde se abstrae la variable n y cuyo cuerpo es una estructura `case`, etc. Todo esto es mucho más claro visualizando la estructura



Nótese que, además de que la estructura de cada expresión es expuesta con claridad, desaparece la necesidad de tener en cuenta detalles tales como e.g. los símbolos concretos usados como delimitadores o separadores o, enteramente, el uso de paréntesis y sus reglas.

Es entonces también necesario, para realizar la matemática y la programación de las expresiones, definir su estructura conceptual como árboles. Esa definición se denomina la *sintaxis abstracta* del lenguaje, y los árboles definidos se denominan *árboles sintácticos* o *de sintaxis abstracta*. La sintaxis abstracta es mucho más sencilla y fácil de recordar que la concreta. La razón para usar la segunda es que resulta más amigable a nuestras costumbres de escritura y lectura. El proceso por el cual una expresión lineal correcta se traduce a un árbol sintáctico se denomina *análisis gramatical* (en inglés: *parsing*) y constituye la primera fase de cualquier compilador o intérprete.

Definir un tipo de árboles no es otra cosa que definir un tipo inductivo, es decir, dar los *constructores* de esos árboles. La notación BNF también puede ser usada con este propósito:

$$exp ::= V\,id \;\mid\; C\,id \;\mid\; Lam\,[id]\,exp \;\mid\; Ap\,exp\,[exp] \;\mid\; Case\,exp\,bs \;\mid\; Rec\,id\,exp.$$

En esta especificación se usa una categoría (tipo) *id* que es el de los nombres simples o *identificadores*. Los dos primeros constructores de expresiones forman respectivamente una variable y un constructor de χ a partir de un identificador dado. Debemos mantener separados dos usos de la palabra "constructor": Desde el comienzo estamos diciendo que χ cuenta con ciertos nombres llamados constructores, designados globalmente con la letra C, y que sirven para formar valores del lenguaje. Ahora estamos hablando de constructores del tipo de las expresiones de χ. Los primeros están *dentro* del lenguaje y los segundos aparecen cuando desde fuera describimos la sintaxis de χ. Para ayudar a evitar confusiones distinguiremos *constructores de valores* (los primeros, que son en general funciones de χ) y *constructores sintácticos* (los usados en la sintaxis abstracta del lenguaje).

Siguiendo con los constructores sintácticos, viene enseguida el de abstracciones funcionales, donde se usa una lista de identificadores como nombres a ser ligados. Luego viene la aplicación de una expresión a una lista. El constructor sintáctico correspondiente al "case" utiliza un argumento de tipo *ramas* que no es otra cosa que una *lista* de ramas, es decir una lista de tuplas formadas por un identificador (que representa el constructor), una lista de identificadores (que representan las variables del constructor) y una expresión. Finalmente, existe un operador de ligadura correspondiente al operador de recursión. Por completitud incluimos la definición de la categoría de las ramas:

$bs ::= Empty \;\mid\; Cons\ b\ bs.$

$b ::= (id\,,\,[id], exp).$

En la última línea usamos la notación tradicional de tuplas.

Naturalmente, otra manera de definir la sintaxis abstracta es hacerlo en Haskell, o en Haskell. Lo dejamos como ejercicio para el lector. Este ejercicio sería el comienzo de un proceso de imbibición de χ en el lenguaje funcional elegido, conducente a programar en ese lenguaje un intérprete del modelo.

Como último comentario, el lector debe recordar los conceptos de variable *libre* y *ligada* y de expresión *cerrada* y *abierta* que es necesario considerar dada la existencia del constructor *Lam* de la abstracción funcional y el nuevo operador de ligadura *Rec*.