Curry

Daniel Delgado, *Estudiante, ITCR,* Wilbert Gonzales, *Estudiante, ITCR,*
Anthony Leandro, *Estudiante, ITCR,* and Bryan Mena, *Estudiante, ITCR*

1. Datos Historicos

Curry es un lenguaje experimental multiparadigma (lógico y funcional), con una sintaxis similar a la del lenguaje de programación Haskell y destinado a proveer una plataforma en común para la investigación, enseñanza y aplicación de lenguajes lógico-funcionales integrados. Curry combina elementos de ambos paradigmas de programación. Por un lado, de la programación funcional (expresiones anidadas, evaluación perezosa, funciones de alto orden) y por otra parte, de la programación lógica (variables lógicas, estructuras de datos parciales, búsqueda incorporada). Es un lenguaje que hasta la fecha su desarrollo sigue en proceso gracias a un grupo de investigadores, en su mayoría de origen alemán, donde se destaca la contribución del Dr. Michael Hanus, quien ha sido el más involucrado en el proyecto. La primera aparición de Curry se dió en los años 90 y el último reporte data de enero de 2016 (dicho reporte es utilizado como referencia para la realización de este documento en casi su totalidad) por lo que el lenguaje cuenta con recientes publicaciones relacionadas al mismo. Por ejemplo, *Programación de bases de datos de alto nivel en Curry*, *Traducción de Curry a Javascript*, *Secuencia de comandos de alto nivel del servidor web en Curry*, *Programando robots autónomos en Curry*, entre otros. Curry provee características adicionales en comparación con lenguajes puros (comparado con programación funcional: búsqueda, computación con información parcial; comparado con programación lógica: evaluación más eficiente debido a la evaluación determinista de funciones.). Además, une los más importantes principios operacionales de lenguajes lógico-funcionales integrados, tales como *residuation* y *narrowing*.

Figura 1. Uno de los logos de Curry.

2. Tipos de Datos

En Curry existen tipos de datos predefinidos, y para cada uno de esos tipos de datos existen operaciones importantes seguidamente detalladas.

2.1. Boolean

Los valores boleanos son predefinidos por la declaración del tipo de dato

Cuadro 1. Código de tipo de dato bool

```
data Bool = True | False
```

La conjunción secuencial está predefinida como el operador infijo asociativo izquierdo (&&)

Cuadro 2. Código de ejemplo de datos bool

```
(&&) :: Bool -> Bool -> Bool
True &&x=x
False && x = False
```

Similarmente, la disyunción secuencial "||z la negación *not* están definidas.

2.2. Functions

El tipo $t_1 -> t_2$ es el tipo de una función que produce un valor de tipo t_2 para cada uno de los argumentos de tipo *t1*. Una función f es aplicada a un argumento x por medio de $"f\,x\,"$.
$fe_1e_2e_3...e_n$ es una abreviación que indica que la función f es aplicada a n argumentos.
También se define la aplicación del operador $ con asociatividad derecha el cual sirve para omitir paréntesis, por lo que la expresión **f $ g $ 3+4** es equivalente a **f (g (3+4))**.

2.3. Integer

Los valores enteros comunes, como **42** o **-15** son considerados como constantes de tipo **Int**. Los operadores usuales de valores enteros, como **+** o *****, son funciones predefinidas que son evaluadas solo si ambos argumentos son valores enteros. Pero también se pueden usar esas funciones como restricciones pasivas. Por ejemplo:

Cuadro 3. Código de ejemplo de restricciones con Integers

```
digit 0 = True
...
digit 9 = True
```

Se define los dígitos del 0-9 como correctos. Por lo tanto, la conjunción **x*x=:=y & x+x=:=y & digit x** se satisface primeramente enlazando la variable **x** a un dígito bajo la restricción **digit x = True** y seguidamente realizando las dos ecuaciones correspondientes. Por lo que, el resultado final sería: **{x=0,y=0}** | **{x=2,y=4}**.

2.4. Floating Point Numbers

Similares a los enteros, valores como **3.14159** o **5.0e-4** son considerados como constantes de tipo **Float**. Debido a que la sobrecarga no está incluida en el kernel de Curry, los nombres de las funciones aritméticas en floats son diferentes de las funciones correspondientes en enteros.

2.5. Lists

El tipo **[t]** denota todas las listas donde sus elementos son valores de tipo **t**. El tipo de listas puede ser considerado como predefinido por la declaración:

Cuadro 4. Código de ejemplo de declaración de listas

```
data [a] = [] | a : [a]
```

donde **[]** denota la lista vacía y **x:xs** es la lista no vacía que consiste en el primer elemento de **x** y la lista restante **xs**. Como escribir listas entre brackets es soportado Curry, la siguiente expresión es válida: **[e1,e2,...,en]** que también se traduce a la lista **e1:e2:...:en:[]**

2.6. Characters

Valores como **'a'** o **'0'** denotan constantes de tipo **Char**. Los caracteres especiales pueden ser escritos mediante un backslash, por ejemplo: '\n' para el caracter con el valor ASCII 10, o '\228' para el caracter 'ä' con el valor ASCII 228. Existen dos funciones conversoras entre chars y su correspondiente valor ASCII:

Cuadro 5. Código de funciones conversoras de Char-ASCII-Char

```
ord :: Char -> Int
chr :: Int -> Char
```

2.7. Strings

El tipo **String** es una abreviación de **[Char]**. Por ejemplo, los strings pueden ser considerados como una lista de caracteres. Las constantes string están cerradas por doble comilla. Así, la constante string **"hola"** es idéntica a la lista de caracteres **['h','o','l','a']**. Un término puede ser convertido a string mediante la función:

Cuadro 6. Código de conversión de un término a String

```
show :: a -> String
```

Por lo tanto, el resultado de **(show 42)** es la lista de caracteres **['4','2']**

2.8. Tuples

Si $t_1, t_2, ..., t_n$ son tipos y $n >= 2$, entonces $(t_1, t_2, ..., t_n)$ denota el tipo de todas las n-tuplas. Los elementos de tipo $(t_1, t_2, ..., t_n)$ son $(x_1, x_2, ..., x_n)$ donde x_i es un elemento de tipo t_i $(i = 1, ..., n)$. El tipo unidad **()** tiene un único elemento **()** y se considera como definido por: **data () = ()**. Además, el tipo unidad puede ser interpretado como el tipo de 0-tuplas.

3. EXPRESIONES

Las expresiones son una notación fundamental en Curry. Las funciones son definidas por medio de ecuaciones que definen expresiones que son equivalentes a llamadas específicas de funciones. Por ejemplo, la regla

Cuadro 7. Código de una simple expresión.

```
square x = x*x
```

define que la función **square 3** es equivalente a la expresión **3*3**.

3.1. Secuencias aritméticas

Curry maneja dos extensiones sintácticas para definir una lista de elementos de una manera compacta. La primera es una notación para secuencias aritméticas. La secuencia aritmética **[**$e_1, e_2, ..e_3$**]** denota una lista de enteros comenzando por los primeros dos elementos e_1 y e_2 y terminando con el elemento e_3 (donde e_2 y e_3 pueden ser omitidos). El significado exacto para esta notación está definido por las siguientes reglas de traducción:

Secuencia aritmética	Su equivalente
$[e..]$	enumFrom e
$[e_1, e_2..]$	enumFromThen e_1 e_2
$[e_1..e_2]$	enumFromTo e_1 e_2
$[e_1, e_2..e_3]$	enumFromThenTo e_1 e_2 e_3

Las diferentes notaciones tienen el siguiente significado:

- La secuencia $[e..]$ denota la lista infinita $[e, e+1, e+2, ...]$.
- La secuencia $[e_1...e_2]$ denota la lista finita $[e_1, e_1+1, e_1+2, ..., e_2]$. Note que la lista es vacía si $e_1 > e_2$.
- La secuencia $[e_1, e_2..]$ denota la lista infinita $[e_1, e_1+i, e_1+2*i, ...]$ con $i = e_2 - e_1$. Note que i puede ser positivo, negativo o cero.
- La secuencia $[e_1, e_2...e_3]$ denota la lista finita $[e_1, e_1+i, e_1+2*i, ..., e_3]$ con $i = e_2 - e_1$. Note que e_3 no está contenido en esta lista si no existe un entero m talque $e_3 = e_1 + m*i$.

Entonces, $[0, 2.,10]$ denota la lista $[0, 2, 4, 6, 8, 10]$.

3.2. Comprensión de Listas

La segunda notación compacta para listas es *comprensión de listas*. Estas tienen la forma general:
$[e|q_1, ..., q_k]$ con $K \geq 1$ y cada q_i es un calificador que es:

- *un generador* de la forma $p \leftarrow l$, donde p es un patrón local (por ejemplo: una expresión sin símbolos de función definida y sin múltiples ocurrencias de la misma variable) de tipo t y l es una expresión de tipo $[t]$, o sino:
- *un guardia* (por ejemplo: una expresión de tipo **Bool**).

Las variables insertadas en un patrón local se pueden utilizar en calificadores posteriores y la descripción del elemento e. Como la comprensión de listas denota la lista de elementos que son el resultado de evaluar e en el ambiente producido por la evaluación de primera profundidad y evaluación anidad de los generadores que satisfacen a todos los *guardias*. Por lo tanto, la comprensión de listas

$[x|x \leftarrow [1.,50], xmod7 == 0]$ denota la lista $[7, 14, 21, 28, 35, 42, 49]$ y la comprensión de lista $[(x, y)|x \leftarrow [1, 2, 3], y \leftarrow [4, 5]]$ denota la lista $[(1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5)]$

El significado exacto de las comprensiones de listas está definido por las siguientes reglas de traducción (para el propósito de la definición, también se tomará en cuenta las comprensiones de lista con una lista calificadora vacía):

Comprensión de Lista	Su equivalente
$[e\|]$	$[e]$
$[e\|b, q]$	if b then $[e\|q]$ else []
$[e\|$ let *decls* $q]$	let *decls* in $[e\|q]$

Por ejemplo, la comprensión de lista:

Cuadro 8. Código de una notación de comprensión de lista

```
[x | (2,x) <- [(1,3),(2,4),(3,6)]]
```

(la cuál es evaluada a **[4]**) es traducida a la siguiente expresión:

Cuadro 9. Expresión de una comprensión de lista

```
let ok (y,x) = if y==2 then [x] else []
in concatMap ok [(1,3),(2,4),(3,6)]
```

Cuadro 10. Código de ayuda para comprensión del Cuadro 9.

```
-- Accumulate all list elements by applying
--a binary operator from right to left, i.e.,
-- foldr f z [x1,x2,...,xn] =
-- (x1 'f' (x2 'f' ... (xn 'f' z)...)) :
foldr             :: (a->b->b) -> b -> [a] -> b
foldr _ z []      = z
foldrfz(x:xs)     =fx(foldrfzxs)

-- Concatenate a list of lists into one list
concat            :: [[a]] -> [a]
concat l          = foldr (++) [] l

-- Map a function from elements to list and
--merge the results into one list
concatMap :: (a -> [b]) -> [a] -> [b]
concatMap f = concat . map f
```

3.3. Expresiones Case

Las expresiones Case son una notación conveniente de asociación de patrones secuenciales con valores por defecto. La forma más simple de una expresión case es la siguiente: ($e, e_1, ..., e_n$ son expresiones y los patrones $p_1, ..., p_n$ son términos de datos):

Cuadro 11. Ejemplo de una expresión case

```
case e of
p1 -> e1
...
pn -> en
```

Note que las expresiones case usan la regla de diseño lo que significa que los patrones $p_1, ..., p_n$ deben estar verticalmente alineados. El significado operacional informal de la expresión case es la siguiente:

- Evaluar e para que coincida con un patrón p_i.
- Si esto es posible, reemplace toda la expresión case con la alternativa correspondiente e_i (después de reemplazar las variables de patrón que ocurren en p_1 por su expresión actual).
- Si ninguno de los patrones $p_1, ..., p_n$ indice con una relación con algún e_i, la computación falla.
- La asociación con el patrón se intenta secuencialemente, de arriba hacia abajo y de una manéra rígida, lo cual significa no enlazar variables libres que ocurren en e.
- En particular, la evaluación de una expresión case es suspendida si la expresión discriminadora e es evaluada a una variable libre.

Las expresiones case son una notación convenientes para funciones con casos por defecto, por ejemplo la función

Cuadro 12. Otro ejemplo de una expresión case

```
swap z = case z of
[x,y] -> [y,x]
_ ->z
```

devuelve una lista con elementos intercambiados en caso de que la entrada sea una lista con exactamente dos elementos y en cualquier otro caso, devuelve la identidad. Además si la entrada es una variable libre, se suspende. Si no se tomara en cuenta esta última propiedad, se podría reescribir **swap** por medio de las siguientes reglas:

Cuadro 13. Redefinición del ejemplo anterior

```
swap [] = []
swap [x] = [x]
swap [x,y] = [y,x]
swap (x1:x2:x3:xs) = x1:x2:x3:xs
```

Este último ejemplo muestra la mejora de lectura que se obtiene por medio de expresiones case.
Las expresiones case también podrían tener *guardias* y *declaraciones locales* en las alternativas. Por ejemplo, la siguiente expresión es válida:

Cuadro 14. Expresión case con guardias y declaraciones locales

```
case y of Left z
| z >= 0 -> sqr z
| otherwise -> - sqr z
where sqr x = x * x
_ -> 0
```

Las expresiones *guardias* deben ser del tipo **Bool**. Casos alternativos con *guardias* producen un fallo de semántica: si todos los *guardias* de un caso alternativo evalúan a falso, la asociación continúa con la siguiente alternativa, osea, se maneja como si el patrón no se asociara nunca. Por ejemplo:

Cuadro 15. Expresión case sin asociación

```
case (1,3) of
(x,y) | x < 0 -> (0,y)
z -> z
```

devuelve el par (1,3).

3.4. Expresiones case flexibles

Similar al patrón rígido de asociación de expresiones case, también existe una notación para el estándar flexible de patrones de asociación de funciones definidas sin estar explícitamente definiendo una función. Tal caso de expresión case flexible tiene la forma general:

Cuadro 16. Expresión case flexible

```
fcase e of
p1 |g1 ->e1
.
.
.
pn |gn ->en
```

donde $e, e_1, ..., e_n$ son expresiones, los patrones $p_1, .., p_n$ son términos de datos, y los *guardias* (opcionales) $g_1, ..., g_n$ son expresiones de tipo **Bool**. El significado operacional obedece la asociación flexible del patrón con funciones definidas, eso quiere decir, que las alternativas no producen fallos de semántica de expresiones case. Actualmente, esta expresión corresponde a la expresión:

Cuadro 17. Expresión case flexible con el uso de 'let'-'in'

```
let     f p1|g1 = e1
        .
        .
        .
        f pn|gn = en
in      f e
```

donde f es un símbolo funcional auxiliar fresco.Por lo tanto, más de una alternativa puede ser tomada durante la evaluación de una expresión case flexible. Por ejemplo, la expresión:

Cuadro 18. Evaluación múltiple en

```
fcase () of
_ -> False
_ -> True
```

no determinísticamente evalúa a True o False

3.5. Comentarios en el código

Para lograr escribir un comentario en Curry, solo se necesita escribir dos veces <guión>al inicio de una línea de código. Por ejemplo:

Cuadro 19. Ejemplo de comentairo de código en Curry.

```
-- Esto es un comentario en Curry.
```

4. Léxico en Curry

Como parte de un lenguaje multiparadigma, Curry posee muchas reglas que le permite utilizar una gran variedad de caracteres, sin embargo, existen determinadas palabras reservadas para el lenguaje las cuáles son:

case	if	module
data	infix	of
do	infixl	then
else	infixr	type
external	import	where
fcase	in	free
let		

5. Declaración de constructores con campos etiquetados

Un constructor de datos de n argumentos crea un objeto con n componentes. Dichos componentes son normlamente accesados posicionalmente como argumentos al constructor en expresiones o patrones. Para tipos de datos largos es útil asignar campos etiquetados a los componentes de un objeto de datos. Esto permite a un campo específico ser referenciado independientemente de su localización dentro del constructor. Una definición de constructor en una declaración de datos puede asignar etiquetas a los campos del constructor, usando la *sintaxis de grabado* **C {. . .}**. Constructores usando etiquetas de campos pueden ser mezclados libremente con constructores sin ellos. Un constructor con etiquetas de campo asociadas aun puede ser usado como un constructor ordinario. El uso variado de etiquetas es simple abreviatura escrita para operaciones usando un constructor posicionalmente adyacente. Los argumentos del constructor posicional existen en el mismo orden que los campos etiquetados.
Por ejemplo, la definición usando etiquetas de campos:

Cuadro 20. Definición usando etiquetas de campos

```
data Person =
Person { firstName, lastName :: String,
age :: Int }
| Agent { firstName, lastName :: String,
trueIdentity :: Person }
```

es traducido a:

Cuadro 21. Definición usando etiquetas de campos

```
data Person = Person String String Int
| Agent String String Person
```

Una etiqueta no puede ser compartida por más de un tipo en el alcance que tenga.

6. Características

- Composición funcional.
- Asociación de patrones.
- Predicción de restricciones.
- Expresiones anidadas.
- Funciones de alto nivel.
- Concurrencia.
- Evaluación perezosa.
- Residuation.
- Narrowing.

7. Ventajas

- Se puede programar tanto en el paradigma lógico como en el paradigma funcional.
- Es ideal para enseñar a programar en estos paradigmas.

8. Desventajas

- No tiene un compilador propio, el más cercano es Smap que es web e incorpora características de los compiladores existentes pero por lo general, lo que hacen los compiladores actuales es traducir el código de Curry a otros lenguajes, tales como C, Haskell y Prolog.
- Las aplicaciones y documentación fuera del sitio oficial es extremadamente escaza.
- No posee mucho auge debido a que la comunidad de programadores en Curry es muy pequeña.

Referencias

[1] M. Hanus, (2016). Curry: An Integrated Functional Logic Language. Recuperado de http://www.curry-language.org.
[2] W. Jeltsch. A taste of Curry. Consultado de https://jeltsch.wordpress.com.
[3] Y. Xiang, (2010). Functional Logic Programming Language Curry. Recuperado de http://www.cas.mcmaster.ca/cas/