Sobre Funciones

1. Funciones de Alto Orden

Una función típica tiene parármetros que representan datos. Sin embargo, podemos tener parámetros o resultados que sean tipos de función. Las funciones que toman otras funciones como argumentos y/o producen funciones como resultado las llamamos **funciones de alto orden**.

Vamos a introducir 3 funciones de alto orden:

1. La función `map` que toma una función `F` y una lista `[`a_1`, `a_2`, ..., `a_n`]` y produce como resultado la lista: `[F(`a_1`), F(`a_2`), ..., F(`a_n`)]`
2. La función `reduce` que toma una función `F` de dos argumentos y una lista `[`a_1`, `a_2`, ..., `a_n`]` y produce como resultado el valor: `F(`a_1`,F(`a_2`,( ... , F(`a_{n-1}`,`a_n`) ...)))`
3. La función `filter` que toma un predicado `P` y una lista `[`a_1`, `a_2`, ..., `a_n`]` y devuelve como resultado una lista con los elementos de `[`a_1`, `a_2`, ..., `a_n`]` que satisfacen `P`.

2. Nuestras versiones de map, reduce y filter

Por ahora vamos a trabajar con nuestras propias versiones de `map`, `reduce` y `filter`:

```
-fun simpleMap (F,nil) = nil
   | simpleMap(F,x::xs) = F(x)::simpleMap(F,xs);
val simpleMap = fn : ('a ->'b) * 'a list ->'b list
-exception ListaVacia;
exception ListaVacia
-fun reduce (F,nil) = raise ListaVacia
   | reduce(F,[a]) = a;
   | reduce(F,x::xs) = F(x, reduce(F,xs));
val reduce = fn : ('a * 'a ->'a) * 'a list ->'a
-fun filter (P,nil) = nil
   | filter(P,x::xs) = if P(x) then x::filter(P,xs) else filter(P,xs);
val filter = fn : ('a ->bool) * 'a list ->'a list
```

Veamos algunos ejemplos de aplicación de estas funciones que acabamos de presentar.

Ejemplos:

1. Defina una función que dada una lista de reales devuelva otra lista con los cuadrados de la lista pasada como argumento:

```
-fun cuadrados(x:real) = x*x;
val cuadrados = fn : real -> real

-fun f(l: real list) = simpleMap(cuadrados,l);
val f = fn : real list -> real list
```

2. Defina una función que tome una lista de reales y devuelva el máximo:

```
-fun max(x:real, y) = if(x>y) then x else y;
val max = fn : real * real -> real
```

```
-fun g(l:real list)=reduce(max,l);
val g = fn : real list -> real
```

3. Defina una función que dada una lista de reales devuelva otra con los elementos mayores a 0:

```
-fun mayCero(x:real)= (x>0.0);
val mayCero = fn : real ->bool

-fun h(l:real list) = filter(mayCero,l);
val h = fn : real list -> real list
```

3. Funciones currificadas

Hasta ahora hemos considerado únicamente funciones que tienen un único parámetro, aunque muchas veces ese parámetro es una tupla escrita entre paréntesis y usando comas para separar cada elemento de la misma. Por lo tanto, hemos escrito muchas funciones en ML que parecían tener múltiples parámetros como las funciones de muchos otros lenguajes como C, Pascal, etc. pero que, técnicamente, toman un parámetro único.

ML provee una manera más general de conectar el nombre de una función a sus parámetros o argumentos. Muchas veces es útil expresar funciones con múltiples parámetros en su forma **currificada**[1] donde el nombre de la función es seguido por una lista de sus parámetros, sin paréntesis ni comas. El ejemplo que presentaremos a continuación ilustra la diferencia entre las versiones currificadas y no currificadas de las funciones. Más adelante introduciremos la importancia de las funciones currificadas, cuando presentemos funciones parcialmente instanciadas.

Definiremos dos funciones para computar $\texttt{x}^{\texttt{y}}$.

La función `exponente1` es similar a las que veníamos definiendo desde el inicio de este curso. Esta función toma un parámetro que es un par ordenado compuesto por un `real x` y un `int y` y devuelve el valor $\texttt{x}^{\texttt{y}}$.

```
-fun exponente1 (x,0) = 1.0
   | exponente1(x,y) = x * exponente1(x,y-1);
val exponente1 = fn : real * int -> real
```

La función currificada `exponente2` toma realmente dos parámetros y realiza el mismo cálculo que su versión no currificada `exponente1`.

```
-fun exponente2 x 0 = 1.0
   | exponente2 x y = x * exponente2 x y-1;
val exponente2 = fn : real -> int -> real
```

Los parámetros de `exponente2` no están encerrados entre paréntesis y tampoco separados por comas.

A continuación se muestra un ejemplo de cómo llamar a estas dos funciones para calcular 3^4:

```
-exponente1(3.0,4);
val it = 81.0 : real

-exponente2 3.0 4;
val it = 81.0 : real
```

[1] Curried form: nombrado en homenaje a Haskell Curry, quien investigó esta forma de definir funciones.

4. Funciones Parcialmente Instanciadas

Las funciones currificadas son útiles porque nos permiten definir nuevas funciones al aplicar la función a algunos argumentos, pero no todos los que indican sus parámetros.

Analicemos más en detalle las funciones presentadas en la sección anterior. ML interpreta el tipo de `exponente1` como una función que toma un par ordenado del tipo `real * int` y devuelve un `real`. Sin embargo, el tipo de `exponente2` es `real -> int -> real`. Sabiendo que el operador `->` asocia a derecha, se interpreta al tipo como `real -> (int -> real)`, o sea, `exponente2` es función que toma un `real` y devuelve como resultado otra función de tipo `int -> real`.

En ML, las funciones currificadas pueden ser instanciadas parcialmente y esto permite definir nuevas funciones.

Ejemplo:

Usando la definición de `exponente2` podemos crear una nueva función instanciando únicamente su primer argumento:

```
-val g = exponente2 3.0;
val g = fn: int ->real
```

Luego podemos usar la función `g` como cualquier otra función:

```
-g 4;
val it=81.0
```

5. Composición de Funciones

Podemos definir una función composición de esta forma:

```
- fun comp(F,G,x) = G(F(x));
val comp = fn : ('a ->'b) * ('b ->'c) * 'a ->'c
```

O podemos definirla como una función currificada de la siguiente forma:

```
- fun comp2 F G =
        let
           fun C x = G (F (x))
        in
           C
        end;
val comp2 = fn : ('a ->'b) ->('b ->'c) ->'a ->'c
```

O dada dos funciones, por ejemplo:

```
- fun F x = x+3;
val F = fn : int ->int
- fun G y = y*y + 2*y;
val G = fn : int ->int
```

podemos definir la composición usando el operador composición de ML ('o' minúscula):

```
- val H = G o F;
val H = fn : int ->int
```

6. La verdadera versión de map

Si escribimos en la línea de comandos del intérprete:

```
- map; (* si ML no conoce la definición de map, ejecute antes open List;*)
```

veremos:

```
val it = fn : ('a ->'b) ->'a list ->'b list
```

La definición de `map` de ML es la siguiente:

```
fun map F =
        let
           fun M nil = nil
              | M (x::xs) = F x :: M xs
        in
           M
        end;
```

Ejemplo:

```
- fun square (x:real) = x*x;
val square = fn : real ->real

- val squarelist = map square;
val squarelist = fn : real list ->real list

- squarelist[1.0,2.0,3.0];
val it = [1.0,4.0,9.0] : real list
```

7. Las versiones de ML de reduce

7.1. foldr

Si escribimos en la línea de comandos del intérprete:

```
- foldr; (* si ML no conoce la definición de foldr, ejecute antes open List;*)
```

veremos:

```
val it = fn : ('a * 'b ->'b) ->'b ->'a list ->'b
```

La definición de `foldr` de ML es la siguiente:

```
- fun foldr F y nil = y
    | foldr F y (x::xs) = F(x, foldr F y xs);
```

donde:

- `F` es una función que toma que toma dos argumentos: el primero de tipo `'a` y el segundo de tipo `'b`, y devuelve un valor de tipo `'b`
- `y` es el valor resultado asociado a la lista vacía.

Ejemplo:

Usando `foldr` definimos una función que calcula el producto de los elementos de una lista de enteros.

```
- fun F(a,x) = a * x;
val F = fn : int * int ->int
```

```
- fun prod list = foldr F 1 list;
val prod = fn : int list ->int

- prod [1,2,3];
val it = 6 : int
```

7.2. foldl

Si escribimos en la línea de comandos del intérprete:

```
- foldl; (* si ML no conoce la definición de foldl, ejecute antes open List;*)
```

veremos:

```
val it = fn : ('a * 'b ->'b) ->'b ->'a list ->'b
```

La definición de `foldl` de ML es la siguiente:

```
- fun foldl F y nil = y
    | foldl F y (x::xs) = foldl F (F(y,x)) xs;
```

donde:

- `F` es una función que toma que toma dos argumentos: el primero de tipo `'a` y el segundo de tipo `'b`, y devuelve un valor de tipo `'b`
- `y` es el valor resultado asociado a la lista vacía.

Ejemplo:

Usando `foldl` definimos una función que calcula el producto de los elementos de una lista de enteros.

```
- fun F(x,y) = x*y;
val F = fn : int * int ->int

- fun prod list=foldl F 1 list;
val prod = fn : int list ->int

- prod[1,2,3];
val it = 6 : int
```

8. La verdadera versión de filter

Si escribimos en la línea de comandos del intérprete:

```
- filter; (* si ML no conoce la definición de filter, ejecute antes open List;*)
```

veremos:

```
val it = fn : ('a ->bool) ->'a list ->'a list
```

La definición de `filter` de ML es la siguiente:

```
-fun filter P nil = nil
   | filter P (x::xs) = if P(x) then (x::(filter P xs)) else (filter P xs);
```

Ejemplo:

Definimos otra versión de la función que dada una lista de reales devuelve otra con los elementos mayores a 0:

```
-fun mayCero(x:real)= (x>0.0);
val mayCero = fn : real ->bool

-fun h l:real list = filter mayCero l;
val h = fn : real list ->real list

- h [1.0, ~1.0, 0.0, 3.0, ~7.5];
val it = [1.0,3.0] : real list
```