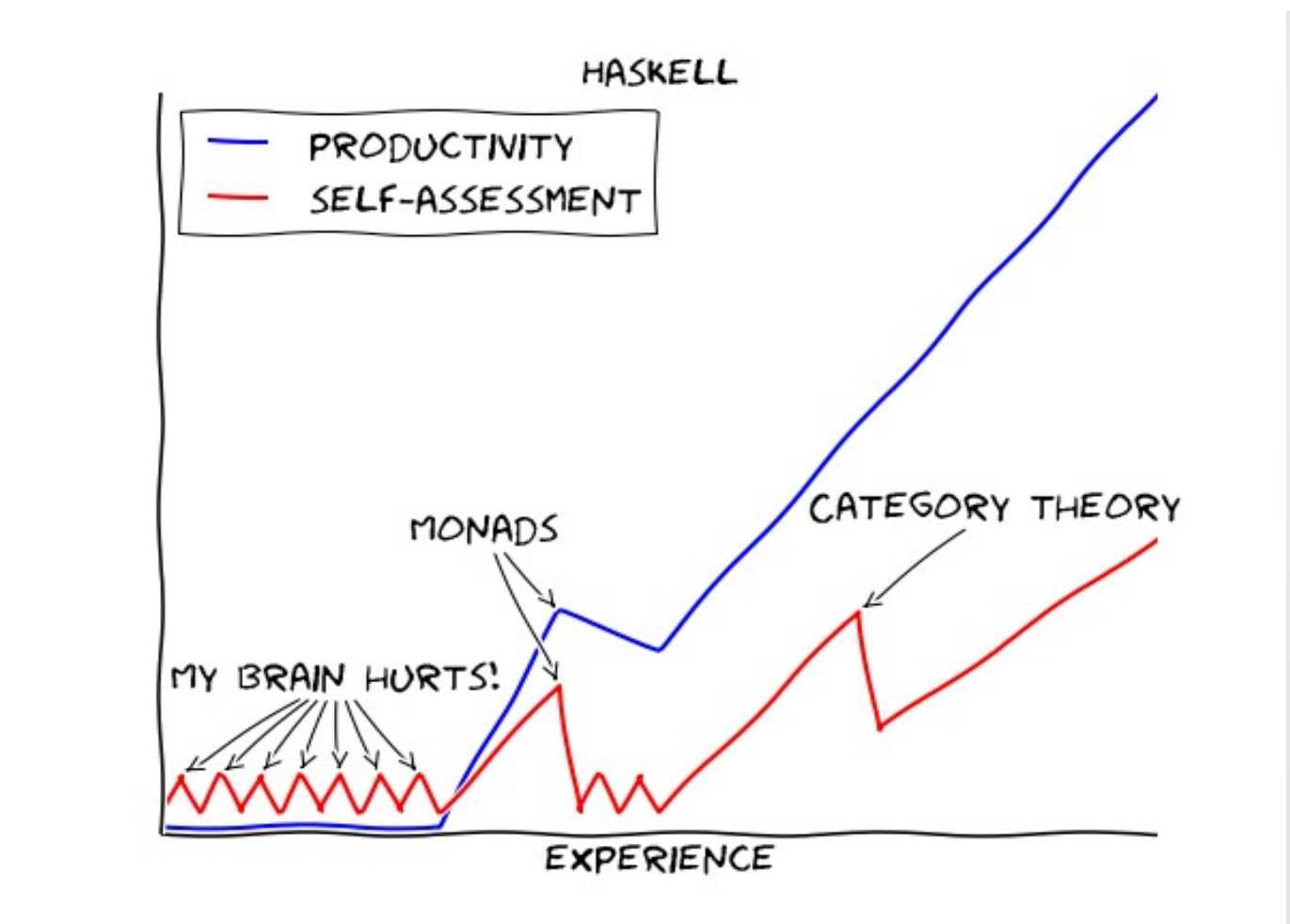
Introducción a la Programación Funcional

Universidad Autónoma de Entre Ríos

- Mónadas o computation builder



Empecemos por un ejemplo

- Las mónadas tienen mayor presencia en Haskell porque solo se permiten funciones puras, o sea, funciones que no tienen efectos colaterales. Las funciones puras aceptan entrada de datos mediante argumentos y producen salida de datos como valores de retorno, y nada más.
- La introducción típica a las mónadas te contará que todo va de poder introducir estos efectos colaterales en este modelo para que puedas hacer I/O, pero ese es solo uno de sus posibles usos. Las mónadas tratan realmente sobre composición de funciones, como veremos más adelante.

Ejemplo en javascript

- Supón que tienes una función para obtener el seno de un número, cuya forma en JavaScript consistiría en un sencillo envoltorio de la librería Math:
- var seno = function(x) { return Math.sin(x) };
- Y que también tienes una función para obtener el cubo de un número:
- var cubo = function(x) { return x * x * x };

Ejemplo en javascript

- Estas funciones usan un número como entrada y salida, haciéndolas componibles: puedes usar la salida de una como la entrada de la siguiente:

  var senoAlCubo = cubo(seno(x));

- Creemos una función para encapsular la función de composición. Tomará como entrada dos funciones f y g y devolverá una nueva función que devuelve f(g(x)).

```
var componer = function(f, g) {
  return function(x) {
    return f(g(x));
  };
};

var senoDelCubo = componer(seno, cubo);
var y = senoDelCubo(x);
```

Ejemplo en javascript

- Después decidimos que necesitamos depurar estas funciones por lo que añadimos un registro de que se las ha llamado. Podríamos hacerlo así:

```
var cubo = function(x) {
  console.log('cubo ha sido llamado.');
  return x * x * x;
};
```

- Pero no se nos permite hacer esto en un sistema que solo permite funciones puras: console.log() no es ni un argumento ni un valor de retorno de la función – es un efecto colateral. Si queremos capturar esta información de registro, debe formar parte del valor de retorno. Vamos a modificar nuestras funciones para que devuelvan una pareja de valores: el resultado y algo de información de depuración:

```
var seno = function(x) {
  return [Mathx.sin(x), 'seno ha sido llamado.'];
};

var cubo = function(x) {
  return [x * x * x, 'cubo ha sido llamado.'];
};
```

Ejemplo en javascript

- Pero ahora descubrimos con horror que estas funciones no son componibles:

```
cubo(3) // -> [27, 'cubo ha sido llamado.']

componer(seno, cubo)(3) // -> [NaN, 'seno ha sido llamado.']
```

- Esto falla por dos motivos: seno está intentando calcular el seno de un array, cuyo resultado es NaN y estamos perdiendo la información de depuración de la llamada a cubo. Sería de esperar que la composición de ambas funciones devolviera el seno del cubo de x y una cadena que nos dijera que cubo y seno han sido llamados:

```
componer(seno, cubo)(3)

// -> [0.956, 'cubo ha sido llamado.seno ha sido llamado.']
```

Ejemplo en javascript

- Una composición sencilla no funcionará por que el valor de retorno de cubo (un array) no es del mismo tipo que el argumento de seno (un número). Se necesita un poco de pegamento. Podríamos escribir una función para componer estas funciones ‘depurables': separaría los valores de retorno de cada función y los volvería a juntar de modo que tuviera sentido:

```
var componerDepurables = function(f, g) {
  return function(x) {
    var gx = g(x),  // por ejemplo: cubo(3) -> [27, 'cubo ha sido llamado.']
        y  = gx[0], // 27
        s  = gx[1], // 'cubo ha sido llamado.'
        fy = f(y),  // seno(27) -> [0.956, 'seno ha sido llamado.']
        z  = fy[0], // 0.956
        t  = fy[1]; // 'seno ha sido llamado.'
    return [z, s + t];
  };
};

componerDepurable(seno, cubo)(3)
// -> [0.956, 'cubo ha sido llamado.seno ha sido llamado.']
```

Ejemplo en javascript

- Hemos compuesto dos funciones que toman un número y devuelven una pareja de número + cadena y hemos creado otra función con la misma firma, significando que puede ser compuesta reiteradamente con otras funciones depurables.

- Para simplificar las cosas, voy a tomar prestada algo de la notación de Haskell. Las siguientes firmas de tipos dicen que la función cubo acepta un número y devuelve una tupla con un número y una cadena:

- cube :: Number -> (Number,String)

Ejemplo en javascript

- Esta es la firma de todas nuestras funciones depurables y sus composiciones. Nuestra función original tenía una firma más sencilla (Number -> Number); la simetría de los tipos del argumento y el valor de retorno es lo que hace las funciones componibles. En vez de escribir lógica específica para componer nuestras funciones depurables, qué tal si sencillamente las convertimos para que su firma diga así:

```
cubo :: (Number,String) -> (Number,String)

seno :: (Number,String) -> (Number,String)
```

Ejemplo en javascript

- Después podríamos usar nuestra función componer original para pegarlas juntas. Podríamos hacer esta conversión a mano y reescribir el código de cubo y seno para aceptar (Number,String) en vez de solo Number, pero esto no escala y terminarás escribiendo el mismo código repetitivo en todas tus funciones. Sería mucho mejor que cada función cumpliera sencillamente su cometido y proveer de una herramienta para forzar las funciones al formato deseado. Llamaremos a esto bind y su trabajo es el de tomar una función Number -> (Number,String) y devolver otra función (Number,String) -> (Number,String).

Ejemplo en javascript

```
var bind = function(f) {
  return function(tupla) {
    var x  = tupla[0],
        s  = tupla[1],
        fx = f(x),
        y  = fx[0],
        t  = fx[1];

    return [y, s + t];
  };
};
```

- Podemos usarla para que nuestras funciones tengan firmas componibles y, después, componer sus resultados.

- var f = componer(bind(seno), bind(cubo));
- f([3, '']) // -> [0.956, 'cubo ha sido llamado.seno ha sido llamado.']

Ejemplo en javascript

- Pero ahora todas las funciones con las que trabajemos tomarán (Number,String) como argumento y lo que nos gustaría en realidad sería pasarles solo un Number. Además de poder convertir funciones, necesitamos poder convertir valores a tipos válidos. O sea, que necesitamos la siguiente función:

unit :: Number -> (Number,String)

- El rol de unit es el de tomar un valor y el de envolverlo en un contenedor básico que las funciones en las que estamos trabajando puedan consumir. Para nuestras funciones depurables esto solo implica emparejar el valor con una cadena vacía:

```
// unit :: Number -> (Number,String)
var unit = function(x) { return [x, ''] };

var f = componer(bind(seno), bind(cubo));
f(unit(3)) // -> [0.956, 'cubo ha sido llamado.seno ha sido llamado.']

// o ...
componer(f, unit)(3) // -> [0.956, 'cubo ha sido llamado.seno ha sido llamado.']
```

Ejemplo en javascript

- Esta función unit también nos permite convertir una función en una función depurable, convirtiendo su valor de retorno en una entrada válida para funciones depurables:

```
// redondear :: Number -> Number

var redondear = function(x) { return Math.round(x) };


// redondearDepurable :: Number -> (Number,String)

var redondearDepurable = function(x) { return unit(round(x)) };
```

- De nuevo, esta tipo de conversión de una función ‘sencilla’ a una depurable puede abstraerse en una función que llamaremos lift. La firma dice que lift toma una función con firma Number -> Number y devuelve una función con firma Number -> (Number,String).

Ejemplo en javascript

```
// lift :: (Number -> Number) -> (Number -> (Number,String))
var lift = function(f) {
  return function(x) {
    return unit(f(x));
  };
};

// o, más sencillo:
var lift = function(f) { return componer(unit, f) };
```

Probemos esto con nuestras funciones para ver si funciona:

```
var redondear = function(x) { return Math.round(x) };

var redondearDepurable = lift(round);

var f = componer(bind(redondearDepurable), bind(seno));
f(unit(27)) // -> [1, 'seno ha sido llamado.']
```

Ejemplo en javascript

- Hemos descubierto tres abstracciones importantes para poder mezclar funciones depurables:
  - lift, que convierte una función ‘sencilla’ en una depurable
  - bind, que convierte una función depurable en una componible
  - unit, que convierte un valor simple en uno válido para depurar, metiéndolo en un contenedor
- Estas abstracciones (bueno, en realidad solo bind y unit), definen una mónada. En la librería estándar de Haskell se llama la mónada **Writer**.

Entonces ???

- En la programación funcional , una mónada (monad en inglés), es una estructura que representa cálculos definidos como una secuencia de pasos. Un tipo de dato con una estructura mónada define lo que simboliza en un bloque de código, o anida funciones del mismo tipo. Esto le permite al programador crear tuberías informáticas que procesen datos en pasos, en los cuales cada acción es ajustada a un Decorator (patrón de diseño) junto con reglas de proceso adicionales provistas por la mónada. Por lo tanto, las mónadas han sido descritas como un “punto y coma programable”; un punto y coma, siendo el operador usado para unir varias declaraciones en la programación imperativa,1 en consecuencia la expresión implica que código extra será ejecutado entre las declaraciones de una tubería. Las mónadas también han sido explicadas con un metáfora física, donde se comportan como una línea de ensamblaje, en las que un objeto transporta datos entre unidades funcionales que la van transformando un paso a la vez.2 También se las puede ver como un patrón de diseño funcional para construir tipos genéricos.

- Los programas puramente funcionales pueden usar las mónadas para estructurar procedimientos que incluyan operaciones en secuencia como aquellos encontrados en la programación estructurada.4 5 Muchos conceptos de programación comunes pueden ser descritos en términos de estructuras de mónadas, incluyendo los efectos secundarios, la Entrada/salida, asignación de variables, el manejo de excepciones, el analizador sintáctico, la programación no determinística, concurrencia y continuaciones. Esto permite que tales conceptos sean definidos en maneras puramente funcionales sin el uso de extensiones a la semántica del lenguaje. Los lenguajes como Haskell proveen mónadas en el núcleo estándar, permitiendo a los programadores rehusar largas partes de su definición formal y aplicar en diversas librerías las mismas interfaces para combinar funciones.

Usos

- Listas por compresión
- Input/output
- Asynchronous programming

```
let AsyncHttp(url:string) =
    async {  let req = WebRequest.Create(url)
             let! rsp = req.GetResponseAsync()
             use stream = rsp.GetResponseStream()
             use reader = new System.IO.StreamReader(stream)
             return reader.ReadToEnd() }
```

En scala

- En scala existe el tipo Monad y la una diferencia con la bibliografía en que el método bind se llama flatMap