

Programación Funcional en JavaScript

Técnicas, Patrones y Arquitecturas Funcionales

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Octubre 2014



Programación Funcional en JavaScript

Presentación

I. ¿Quién Soy?



Licenciado en informática por la Universidad Politécnica de Madrid (UPM) desde el año 2001 y doctor en informática por la Universidad Nacional de Educación a Distancia (UNED) desde el año 2009, Javier es investigador y está especializado en el diseño y análisis de la colaboración. Esta labor la compagina con actividades de evangelización, consultoría, mentoring y formación especializada para empresas dentro del sector IT. Inquieto, ávido lector y seguidor cercano de las innovaciones en tecnología.



javier.velez.reyes@gmail.com



[@javiervelezreye](https://twitter.com/javiervelezreye)



[linkedin.com/in/javiervelezreyes](https://www.linkedin.com/in/javiervelezreyes)



[gplus.to/javiervelezreyes](https://plus.google.com/+javiervelezreyes)



[jvelez77](#)



[javiervelezreyes](#)



youtube.com/user/javiervelezreyes

II. ¿A Qué Me Dedico?

Desarrollado Front/Back

Evangelización Web

Arquitectura Software

Formación & Consultoría IT

E-learning

Diseño de Sistemas de Colaboración

Learning Analytics

Gamificación Colaborativa

Javier Vélez Reyes
@javiervelezreye
Javier.veler.reyes@gmail.com

1 *Introducción*

- Introducción
- Objetivos de la Programación Funcional
- Principios de Diseño Funcional
- Dimensiones y Planos de Actividad

Programación Funcional en JavaScript

Introducción

I. Introducción

La programación funcional es un viejo conocido dentro del mundo del desarrollo. No obstante, en los últimos años está cogiendo tracción debido, entre otros factores, a la emergencia de arquitecturas reactivas, al uso de esquemas funcionales en el marco de Big Data y al creciente soporte que se le da desde diversas plataformas vigentes de desarrollo. JavaScript ha sido siempre un lenguaje con fuerte tendencia al diseño funcional. En este texto revisaremos sus objetivos y principios y discutiremos los mecanismos, técnicas y patrones empleados actualmente para construir arquitecturas funcionales haciendo uso de JavaScript.

Programación Funcional

*Arquitecturas
centradas en la
transformación*

*Variantes
funcionales*

Inmutabilidad

*Transparencia
Referencial*

Compositividad

*Arquitecturas
dirigidas por flujos de
datos*

Programación Orientada a Objetos

*Puntos de Extensión
Polimórfica*

*Arquitecturas
centradas en la
abstracciones*

*Sustitutividad
Liskoviana*

*Encapsulación de
estado*

*Arquitecturas
dirigidas por flujos de
control*

Programación Funcional en JavaScript

Introducción

II. Objetivos de la Programación Funcional

A. Especificación Declarativa

La programación funcional persigue diseñar especificaciones de comportamiento abstracto que se centren en la descripción de las características de un problema más que en una forma particular de resolverlo. De acuerdo a esto, se entiende que la responsabilidad de encontrar una solución para el problema descansa no tanto en manos del programador sino en la arquitectura funcional subyacente que toma en cuenta dicha especificación.

Funcional *La descripción fluida del estilo funcional permite entender fácilmente la especificación*

```
function total (type) {  
  return basket.filter (function (e) {  
    return e.type === type;  
  }).reduce (function (ac, e) {  
    return ac + e.amount * e.price;  
  }, 0);  
}
```

```
function total (type) {  
  var result = 0;  
  for (var idx = 0; idx < basket.length; idx++) {  
    var item = basket[idx];  
    if (basket[idx].type === type)  
      result += item.amount * item.price;  
  }  
  return result;  
}
```

Aunque operativamente es equivalente, el estilo imperativo es más confuso y resulta más difícil de entender

```
var basket = [  
  { product: 'oranges', type: 'food', amount: 2, price: 15 },  
  { product: 'bleach', type: 'home', amount: 2, price: 15 },  
  { product: 'pears', type: 'food', amount: 3, price: 45 },  
  { product: 'apples', type: 'food', amount: 3, price: 25 },  
  { product: 'gloves', type: 'home', amount: 1, price: 10 }  
];
```

Cómo

Objetos

Programación Funcional en JavaScript

Introducción

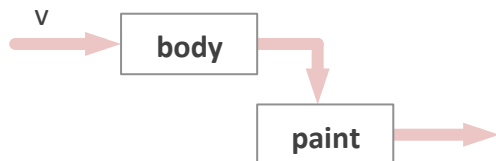
II. Objetivos de la Programación Funcional

B. Abstracción Funcional

La programación funcional persigue alcanzar una especificación de alto nivel que capture esquemas algorítmicos de aplicación transversal. Como veremos a continuación, así se fomenta la reutilización y la adaptación idiomática. Esto se consigue con arquitecturas centradas en la variabilidad funcional y contrasta ortogonalmente con la orientación a objetos.

Funcional

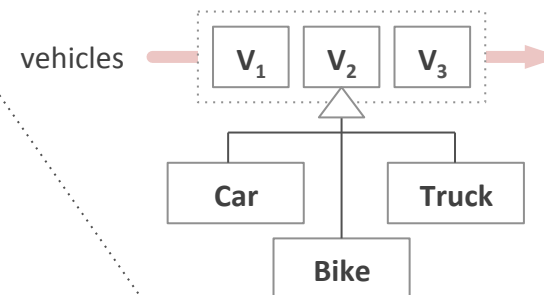
Abstracción Funcional



```
var phases = [
  function body (v) {...},
  function paint (v) {...}
];
var test = function (phases) {
  function (vehicle) {
    return phases.reduce(function (ac, fn) {
      return fn(ac);
    }, vehicle);
  }
};
```

En FP la función es la unidad de abstracción ya que permite definir esquemas algorítmicos que dependen de funciones pasadas como parámetros

```
var vehicles = [new Car(), new Truck(), ...];
var garage = function (vehicles) {
  for (v in vehicles)
    vehicles[v].test();
};
garage(vehicles);
```



En OOP los algoritmos se distribuyen entre objetos con implementaciones variantes

Abstracción de tipos Objetos

Programación Funcional en JavaScript

Introducción


II. Objetivos de la Programación Funcional

C. Reutilización Funcional


Como consecuencia de la capacidad de abstracción en la especificación declarativa, la programación funcional alcanza cotas elevadas en la reutilización de funciones. Este nivel de reutilización algorítmica no se consigue en otros paradigmas como en la orientación a objetos.

```
var get = function (collection) {  
    return function (filter, reducer, base) {  
        return collection  
            .filter(filter)  
            .reduce(reducer, base);  
    };  
};
```

El esquema algorítmico se reutiliza sobre distintas estructuras de datos y con distintas aplicaciones funcionales



```
var users = [  
    { name: 'jvelez', sex: 'M', age: 35 },  
    { name: 'eperez', sex: 'F', age: 15 },  
    { name: 'jlopez', sex: 'M', age: 26 }  
];  
var adult = function (u) { return u.age > 18; };  
var name = function (ac, u) {  
    ac.push(u.name);  
    return ac;  
};  
get(users)(adult, name, []);
```



```
var basket = [  
    { product: 'oranges', type: 'F', price: 15 },  
    { product: 'bleach', type: 'H', price: 15 },  
    { product: 'pears', type: 'F', price: 45 },  
];  
var food = function (p) { return p.type === 'F'; };  
var total = function (ac, p) {  
    return ac + p.price;  
};  
get(basket)(food, total, 0);
```

Programación Funcional en JavaScript

Introducción

II. Objetivos de la Programación Funcional

D. Adaptación Funcional

Una de las virtudes de la programación funcional es la capacidad de poder transformar la morfología de las funciones para adaptarlas a cada escenario de aplicación real. En términos concretos esto significa poder cambiar la signatura de la función y en especial la forma en que se proporcionan los parámetros de entrada y resultados de salida. Veremos que existen mecanismos técnicos y patrones para articular este proceso.

```
function greater (x, y) {  
  return x > y;  
}
```

reverse



Se invierte el orden de aplicación de los parámetros de forma transparente

```
function greater (x, y) {  
  return y > x;  
}
```

curry



Se transforma la evaluación de la función para que pueda ser evaluada por fases

```
(function greater (x) {  
  function (y) {  
    return y > x;  
  }  
})(18)
```

partial



Se configura parcialmente la función para obtener el predicado de adulto

```
function greater (x) {  
  function (y) {  
    return y > x;  
  }  
}
```

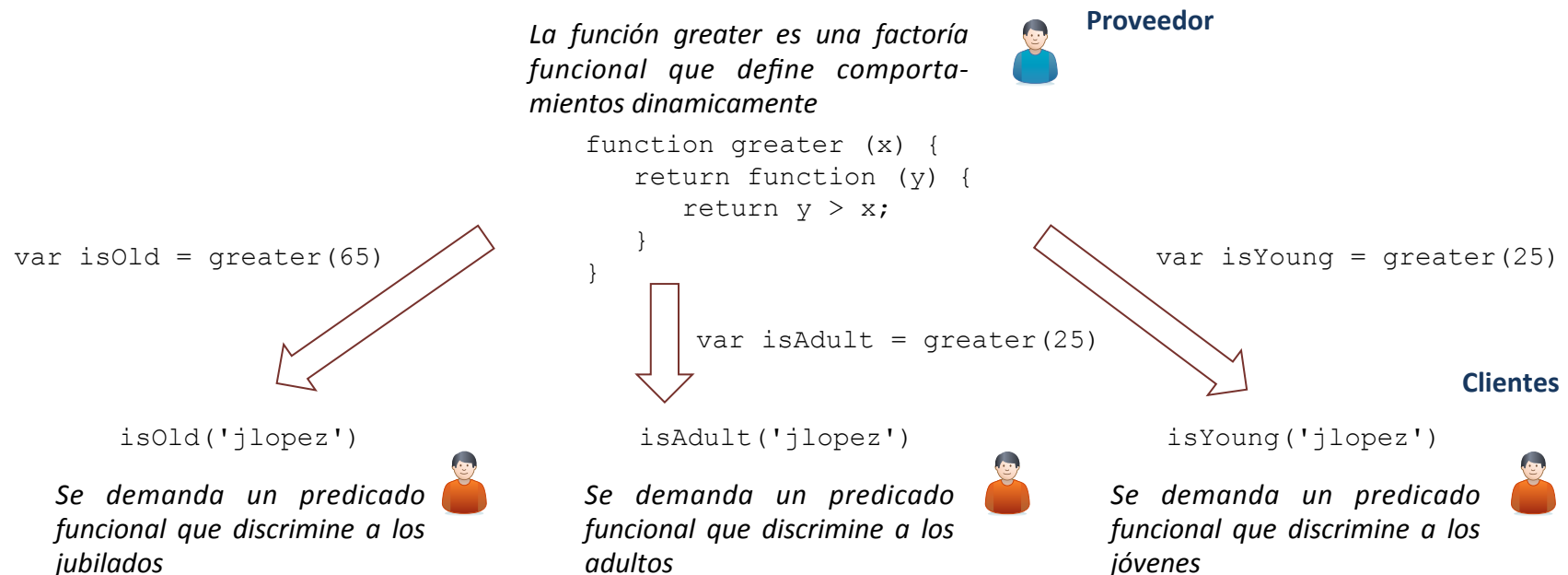

Programación Funcional en JavaScript

Introducción

II. Objetivos de la Programación Funcional

E. Dinamicidad Funcional

La programación funcional permite diseñar funciones que sean capaces de construir abstracciones funcionales de forma dinámica en respuesta a la demanda del contexto de aplicación. Como veremos, este objetivo se apoya en mecanismos de abstracción y también abunda en la adaptación funcional que acabamos de describir.



Programación Funcional en JavaScript

Introducción

III. Principios de Diseño Funcional

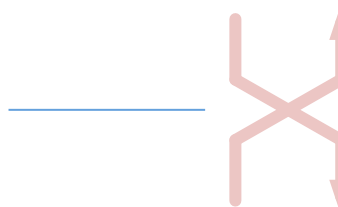
A. Principio de Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. La interpretación de este principio de diseño tiene tres lecturas complementarias.

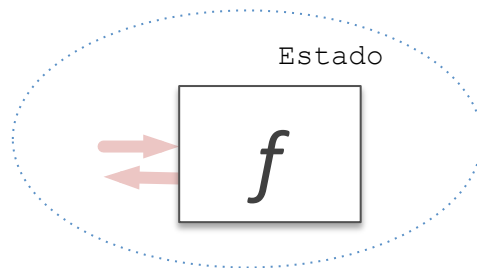
Funciones Puras

Dado que el predicado greater sólo depende de sus parámetros de entrada x e y , se trata de una función pura. Por ello, mantiene la transparencia referencial permitiendo su sustitución sin impacto en la semántica del contexto de aplicación

```
var old = greater(65)
```



```
var old = (function greater (x) {  
  function (y) {  
    return y > x;  
  }  
})(65);
```



Se dice que una función es pura – recuerda al estilo de comportamiento matemático – si su valor de retorno sólo depende de los parámetros de entrada y no del estado ambiental (variables globales, variables retenidas en ámbito léxico, operaciones de E/S, etc.).

Programación Funcional en JavaScript

Introducción

III. Principios de Diseño Funcional

A. Principio de Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. La interpretación de este principio de diseño tiene tres lecturas complementarias.

Comportamiento Idempotente

Estilo Funcional

```
function Stack () {  
  return { stack: [] };  
}  
function push (s, e) {  
  return {  
    stack: s.stack.concat(e),  
    top: e  
  };  
}  
function pop (s) {  
  var stack = [].concat(s.stack);  
  var e = stack.pop();  
  return {  
    stack: stack,  
    top: e  
  };  
}
```

La pila mantiene el estado interno y las operaciones push y pop no resultan idempotentes

El estado se mantiene externamente y se pasa como parámetro a cada operación con lo que el comportamiento es idempotente

Estilo Orientado a Objetos

```
function Stack () {  
  var items = [];  
  return {  
    push: function (e) {  
      items.push(e);  
    },  
    pop: function () {  
      return items.pop();  
    }  
  };  
}
```

Se dice que una función es idempotente si siempre devuelve el mismo resultado para los mismos parámetros de entrada. En nuestro ejemplo debería verificarse esta igualdad:

```
pop + pop === 2 * pop
```

Programación Funcional en JavaScript

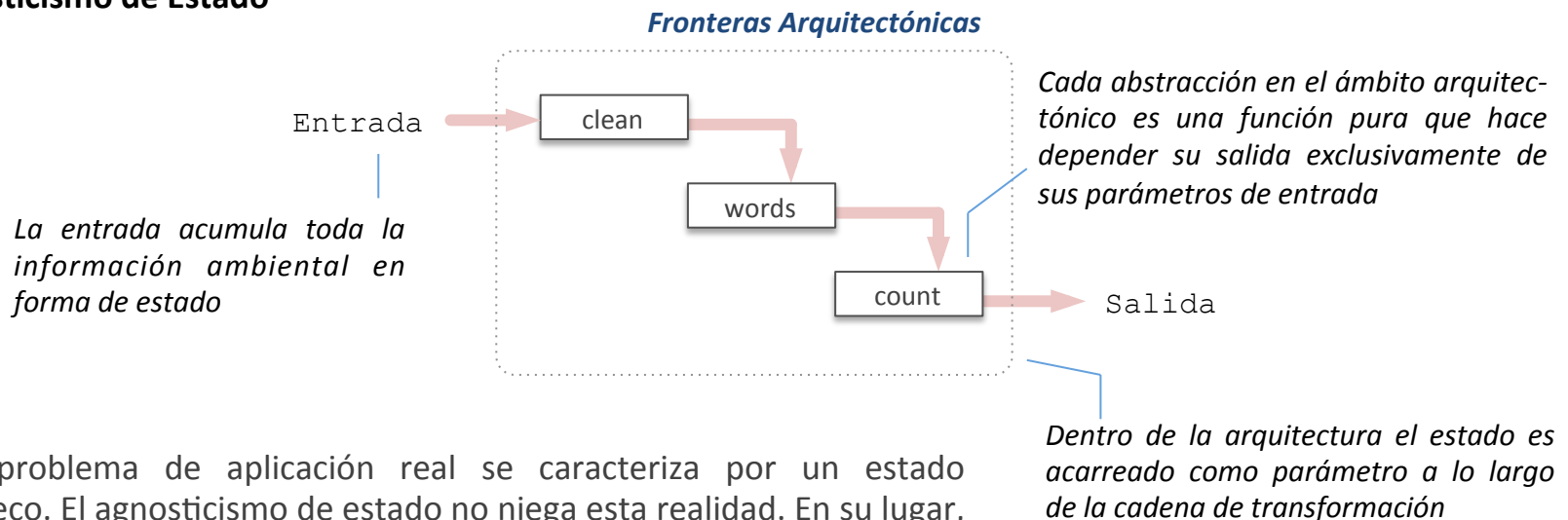
Introducción

III. Principios de Diseño Funcional

A. Principio de Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. La interpretación de este principio de diseño tiene tres lecturas complementarias.

Agnosticismo de Estado



Todo problema de aplicación real se caracteriza por un estado intrínseco. El agnosticismo de estado no niega esta realidad. En su lugar, el principio debe interpretarse como que dentro de las fronteras del sistema el comportamiento funcional sólo puede depender de los parámetros explícitamente definidos en cada función.

Programación Funcional en JavaScript

Introducción

III. Principios de Diseño Funcional

B. Principio de Inmutabilidad de Datos

En el modelo de programación imperativa, la operativa de computo se entiende como un proceso secuencial y paulatino de transformación del estado mantenido por la arquitectura hardware de la máquina. En programación funcional – y en virtud de la transparencia referencial – las funciones no dependen del estado ambiental. De ello se colige que el concepto de variable entendido como depósito actualizable de información no existe.

Imperativo

```
var x = 1
...
x = x + 1
```



P. Funcional

```
x@(0) = 1
...
x@(t+1) = x@(t) + 1
```

En programación imperativa las variables se entienden como depósitos de información en memoria que pueden actualizarse durante el tiempo de vida del programa. La dimensión tiempo queda oculta y esto dificulta el razonamiento

La programación funcional pone estrés en que los cambios en los datos de un programa deben manifestarse como funciones sobre la dimensión tiempo. Esto significa que el concepto de variables como depósito de información y las operaciones de actualización sobre ellas no están permitidas

En términos prácticos la aplicación de este principio se traduce en que las funciones nunca deben actualizar los parámetros de entrada sino generar a partir de ellos resultados de salida

```
function push (s, e) {
  return s.push(e);
}
```

Estilo Imperativo

Se modifica s

```
function push(s, e) {
  return s.concat(e);
}
```

Estilo Funcional

No se modifica s

Programación Funcional en JavaScript

Introducción

III. Principios de Diseño Funcional

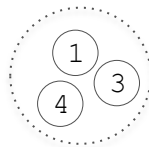
C. Principio de Computación Centrada en Colecciones

A diferencia de lo que ocurre en otros paradigmas como en orientación a objetos donde los datos se organizan en forma de grafos relacionales, en programación funcional éstos deben modelarse como colecciones – arrays o diccionarios en JavaScript – para que se les pueda sacar pleno partido¹. Se dice que el modelo de computación de la programación funcional está centrado en colecciones.

Set

Un conjunto se modela como un array o un diccionario de booleanos

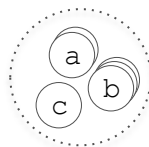
```
[true,
false,
true,
true]
```



Bag

Una bolsa se modela como un diccionario de contadores

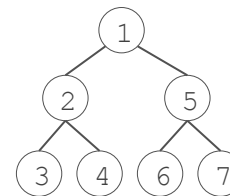
```
{a: 2,
b: 3,
c: 1}
```



Binary Tree

Un array se aplana como una colección anidada de arrays de 3 posiciones, árbol izquierdo, nodo, árbol derecho

```
[
  [
    [], 3, [],
    2,
    [], 4, []],
  1,
  [[], 6, []],
  5,
  [], 7, []]]
```



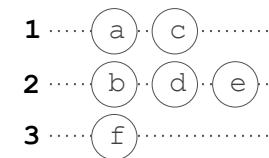
Heap

Una cola con prioridad es un diccionario de arrays o un array de arrays

```
{1: [a, c],
2: [b, c, d],
3: [f]}
```

```
[
  [a, c],
  [b, c, d],
  [f]
]
```

...



¹ Los diccionarios son una aproximación de representación aceptable en tanto que pueden procesarse como colecciones de pares clave-valor

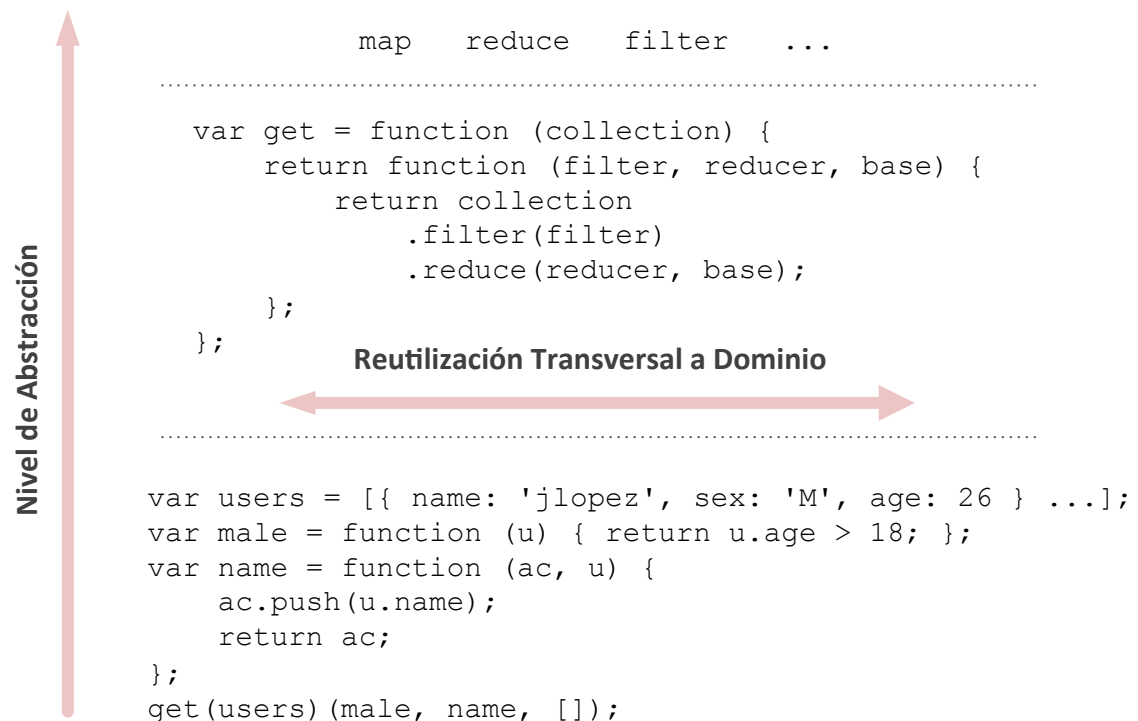
Programación Funcional en JavaScript

Introducción

III. Principios de Diseño Funcional

D. Principio de Diseño Dirigido por Capas

La construcción de arquitecturas funcionales sigue un proceso de diseño dirigido por capas o niveles de especificación. Desde las abstracciones más generales encontradas en librerías funcionales se construye un nivel idiomático que propone un esquema abstracto de solución para el problema propuesto y sólo entonces se concreta en el dominio particular del problema.



Nivel de Librería

Se utilizan funciones generales con un alto nivel de abstracción

Nivel Idiomático

Se crea un esquema funcional que resuelve la familia de problemas al que pertenece el problema planteado

Nivel de Dominio

Se contextualiza el esquema anterior dentro del dominio de aplicación concretando con datos y funciones específicas

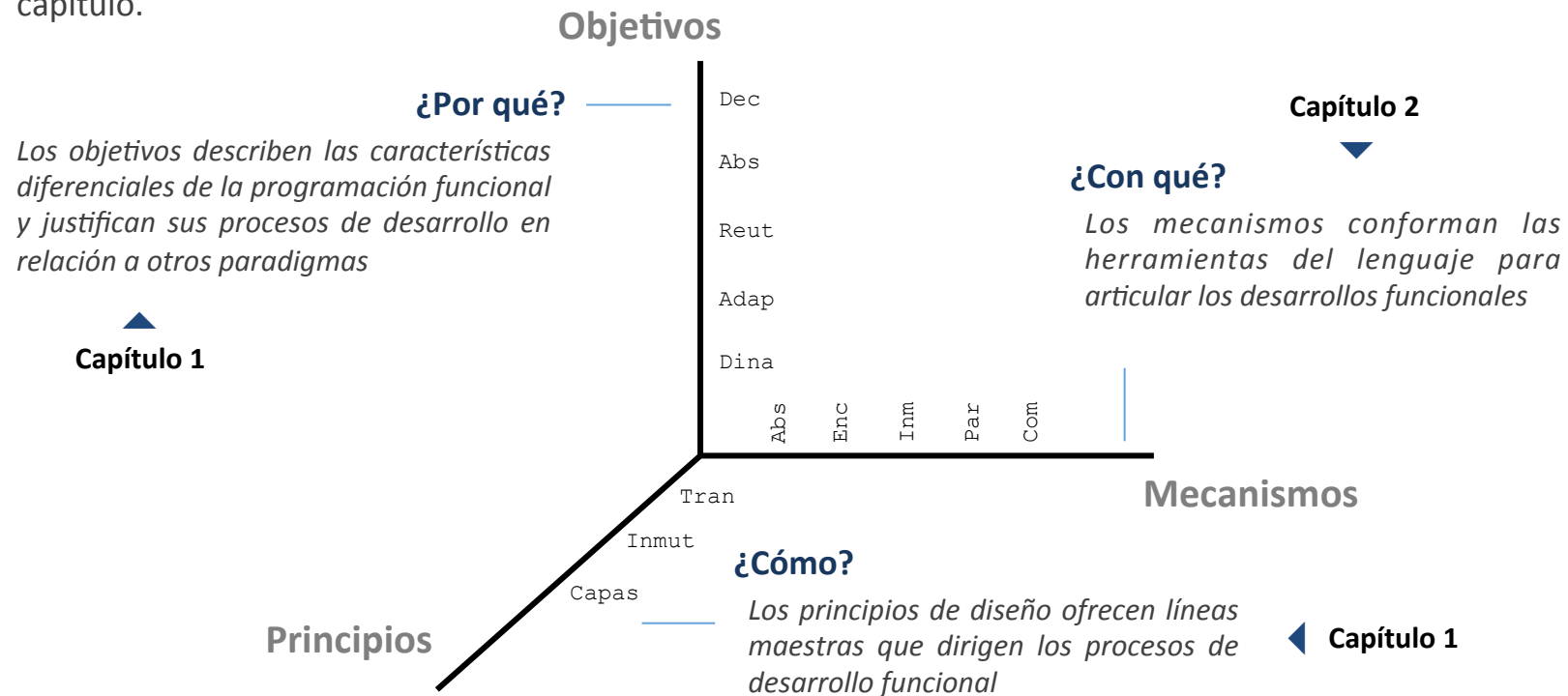
Programación Funcional en JavaScript

Introducción

IV. Caracterización de la Programación Funcional

A. Ejes Dimensionales

A lo largo de esta introducción hemos presentado el paradigma de programación funcional centrándonos en sus objetivos y principios fundacionales. Nos resta por describir los mecanismos esenciales en los que se apoya. No obstante, dado que el soporte a los mismos está fuertemente condicionado por el lenguaje estos aspectos serán cubiertos en el siguiente capítulo.



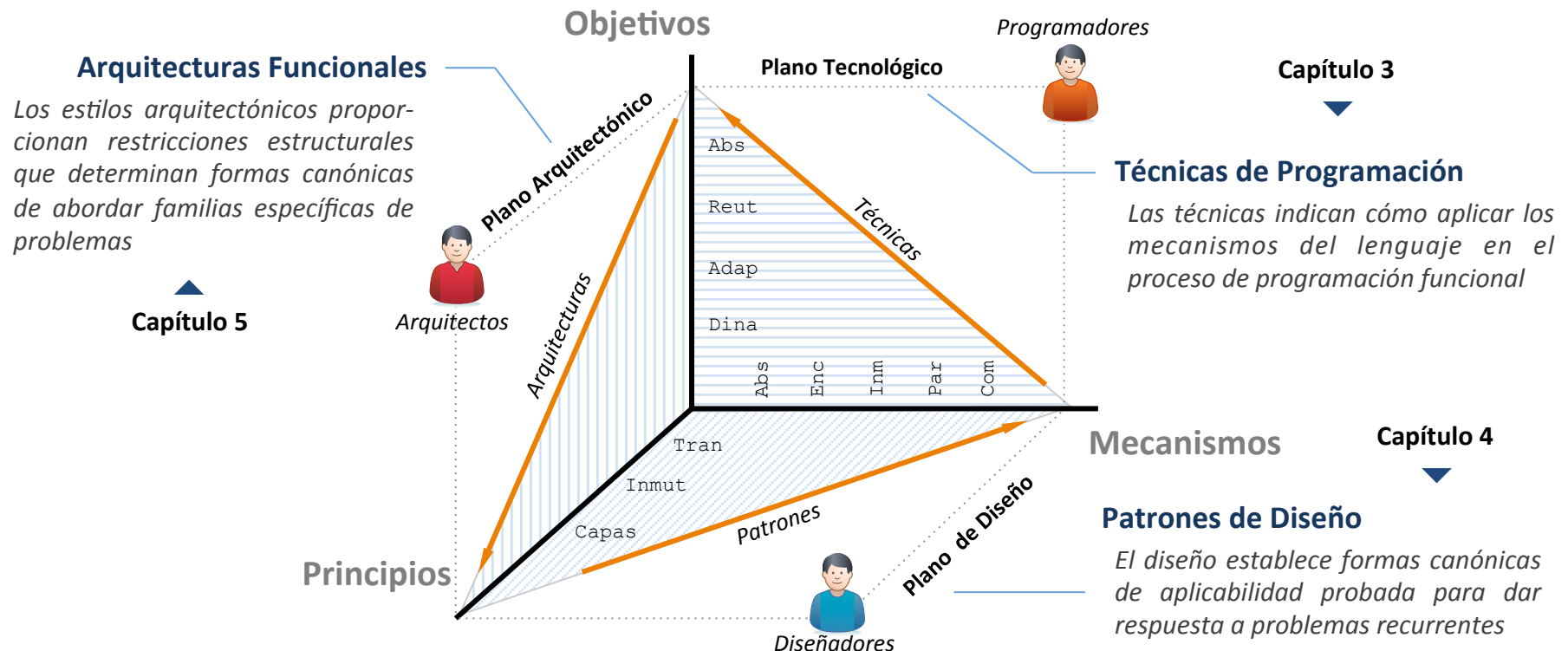
Programación Funcional en JavaScript

Introducción

IV. Caracterización de la Programación Funcional

B. Planos de Actividad

El recorrido de este texto avanza describiendo cada uno de los tres planos de actividad que caracterizan la programación funcional. En el capítulo 3 abordaremos las técnicas de programación esenciales del paradigma. El capítulo 4 profundiza en los patrones de diseño funcionales. Y el capítulo 5 presenta las principales arquitecturas en programación funcional.



Javier Vélez Reyes
@javiervelezreyes
Javier.veler.reyes@gmail.com

2 *JavaScript como Lenguaje Funcional*

- Definición Funcional por Casos
- Definición por Recursión
- Expresiones Funcionales de Invocación Inmediata IIFE
- Definición en Orden Superior
- Clausuras & Retención de Variables

Programación Funcional en JavaScript
JavaScript como Lenguaje Funcional

Programación Funcional en JavaScript

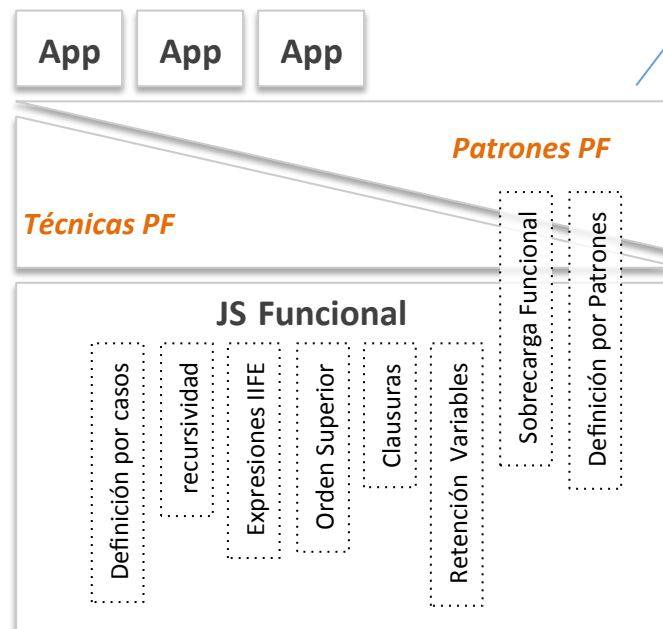
JavaScript como Lenguaje Funcional

I. Introducción

Como ya hemos comentado, los procesos de programación funcional consisten esencialmente en la definición de una colección de declaraciones funcionales que puedan adaptarse a distintos niveles de abstracción y aplicarse conjugadamente para resolver problemas de forma declarativa. Para articular estas tareas de definición el paradigma funcional ofrece una colección de mecanismos esenciales que revisamos en este capítulo.

Nivel de Lenguaje

El nivel de lenguaje está formado por la colección de mecanismos de programación que proporciona el lenguaje de forma nativa



Nivel Idiomático

Las deficiencias que por su concepción presenta el lenguaje se deben suplir a este nivel por aplicación de técnicas y patrones de diseño que revisaremos en próximos capítulos

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

II. Mecanismos de Programación Funcional

A. Definición Funcional por Casos

La forma más sencilla de definir una función es diseñando explícitamente la colección de valores de retorno que debe devolver para cada posible parámetro de entrada. Aunque este mecanismo parece poco flexible aisladamente, cuando se conjuga con otras formas de definición resulta de aplicación frecuente.

Única Expresión

El uso anidado del operador condicional permite expresar los pares caso-resultado como una única expresión de retorno

```
function f (parámetros) {  
  return caso-1 ? resultado-1 :  
         caso-2 ? resultado-2 :  
         resultado-defecto;  
}
```

Enumeración Explícita de casos

La definición por casos se expresa declarando explícitamente el valor que debe devolver la función para cada posible valor de los parámetros de entrada

```
function comparator(x) {  
  return x > 0    ? 1 :  
         x === 0 ? 0 :  
         -1 ;  
}
```

Se utiliza el operador condicional anidado para definir cada caso funcional

```
function even(n) {  
  return n % 2;  
}
```

La definición mediante una única expresión es el escenario más sencillo de definición por casos

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

II. Mecanismos de Programación Funcional

B. Definición por Recursión

La definición de funciones por recursión consiste en la invocación de la propia función como parte del proceso de definición. El esquema de definición recursivo se apoya en el mecanismo anterior para distinguir entre casos base – aquéllos que devuelven un resultado final – de aquéllos casos recursivos – donde el valor de retorno vuelve a invocar a la propia función.

Casos base

Los casos base son el final del proceso recursivo y se suponen soluciones inmediatas a problemas sencillos

```
function f (parámetros) {  
  return caso-base-1 ? resultado-base-1 :  
         caso-base-2 ? resultado-base-2 :  
         caso-recursivo-1 ? resultado-recursivo-1 :  
         caso-recursivo-2 ? resultado-recursivo-2 :  
         resultado-defecto ;  
}
```

Casos recursivos

Los casos recursivos se diseñan invocando a la propia función sobre valores de parámetros que convergen a los casos base

I. Regla de cobertura

Asegúrese de que todos los casos base del problema han sido incluidos explícitamente en la definición de la función

II. Regla de convergencia

Asegúrese de que cada resultado recursivo converge a alguno de los casos base

III. Regla de autodefinition

Para diseñar cada caso recursivo asuma que la función por definir está ya definida para valores de parámetros más próximos a los casos base

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

II. Mecanismos de Programación Funcional

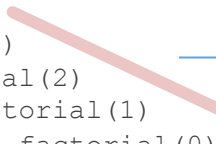
B. Definición por Recursión

En función de cómo se expresan los casos recursivos de una función podemos distinguir entre dos tipos de situaciones. En la recursión directa, los casos recursivos de la función se expresan en términos de llamadas a ella misma. En la recursión indirecta, la función invoca a otra función que recurre sobre la primera.

Recursión Directa

```
function factorial (n) {  
  return n === 0 ? 1 :  
    n * factorial(n - 1);  
}
```


```
factorial(4)  
= 4 * factorial(3)  
= 4 * 3 * factorial(2)  
= 4 * 3 * 2 * factorial(1)  
= 4 * 3 * 2 * 1 * factorial(0)  
= 4 * 3 * 2 * 1 * 1  
= 24
```



Los casos recursivos convergen hacia los casos base reduciendo el tamaño del problema en una unidad a cada paso

La convergencia hacia los casos base va incluyendo un operador de negación que se resuelven al llegar al caso base

```
even(4)  
= !odd(3)  
= !!even(2)  
= !!!odd(1)  
= !!!!even(0)  
= !!!!true  
= true
```



Recursión Indirecta

```
function even (n) {  
  return n === 0 ? true :  
    !odd(n-1);  
}  
function odd (n) {  
  return n === 0 ? false :  
    !even(n-1);  
}
```

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

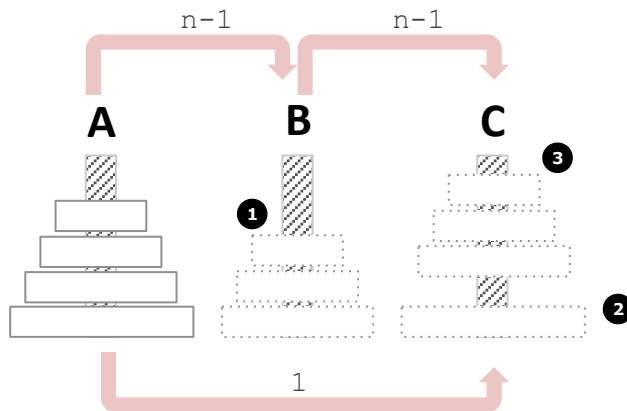
II. Mecanismos de Programación Funcional

B. Definición por Recursión

Dado que la programación funcional no dispone de estructuras de control de flujo, la única forma de resolver problemas que requieren un computo iterativo es expresarlas a partir de una formulación recursiva. Aunque puede resultar en principio más complejo, la expresividad funcional aumenta con este tipo de expresiones. Además hay problemas que tienen una resolución inherentemente recursiva.

Problema de las Torres de Hanoi

Mover discos de uno en uno para dejarlos en la misma posición de A en C usando B como auxiliar. Los discos por encima de uno dado deben ser siempre de menor tamaño que éste.



```
function hanoi (n, origen, aux, destino) {  
  if (n === 1) mover(origen, destino);  
  else {  
    hanoi(n-1, origen, destino, aux);  
    mover(origen, destino);  
    hanoi(n-1, aux, origen, destino);  
  }  
}  
  
function mover (origen, destino) {  
  destino.push (origen.pop());  
}  
  
var A = [4, 3, 2, 1], B = [], C = []  
hanoi(A, B, C);
```

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

II. Mecanismos de Programación Funcional

C. Expresiones Funcionales e Invocación Inmediata

No en pocas ocasiones la necesidad de definir una función se restringe a un solo uso que se realiza a continuación de su definición. Si encerramos una definición de función entre paréntesis y aplicamos a la expresión resultante el operador de invocación – los parámetros actuales a su vez entre parentesis y separados por comas – obtenemos una expresión funcional de invocación inmediata IIFE.

Expresión Funcional

La definición funcional se encierra entre paréntesis para convertirla en una expresión evaluable

```
(function f (parámetros-formales) {  
    <<cuerpo de declaración funcional>>  
}) (parámetros-actuales);
```

Invocación Inmediata

La expresión funcional se invoca directamente por aplicación de los parámetros actuales

```
var cfg = (function config(user) {  
    // getting config from DB  
    return {...}  
}) ('jvelez');
```

Un típico ejemplo de IIFE se da cuando se pretenden realizar cálculos complejos o tareas que se efectuarán una sola vez a lo largo del código

La ventaja de este tipo de construcciones es que la función se libera de la memoria tras su ejecución

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

II. Mecanismos de Programación Funcional

D. Definición en Orden Superior

En JavaScript las funciones son ciudadanos de primer orden. Esto quiere decir que a todos los efectos no existe ningún tipo de diferencia entre una función y cualquier valor de otro tipo primitivo. Esto permite a una función recibir otras funciones como parámetros o devolver funciones como retorno. Este mecanismo se le conoce con el nombre de orden superior.

Funciones como Parámetros a Otras Funciones

El paso de funciones como parámetros a una función permite que el comportamiento de ésta sea adaptada por medio de la inyección de código variante en cada invocación. Esta aproximación contrasta con la programación por tipos variantes propia de la orientación a objetos.

```
function once (fn) {  
  var called = false;  
  return function () {  
    if (!called) {  
      called = true;  
      return fn.apply(this, arguments);  
    }  
  }  
}
```

```
var greater = function (x, y) {...};  
var less    = function (x, y) {...};  
var even    = function (x) {...};  
  
[1, 4, 3, 2].sort(greater);  
[1, 4, 3, 2].sort(less);  
[1, 4, 3, 2].filter(even);
```

Funciones como Retorno de Otras Funciones

Las funciones son capaces de generar dinámicamente funciones que devuelven como resultado. Esto es de especial interés cuando la función devuelta es el resultado de transformar una función recibida como argumento. Este proceso justifica el nombre de orden superior ya que se opera con funciones genéricas.

Programación Funcional en JavaScript

JavaScript como Lenguaje Funcional

II. Mecanismos de Programación Funcional

E. Clausuras & Retención de Variables

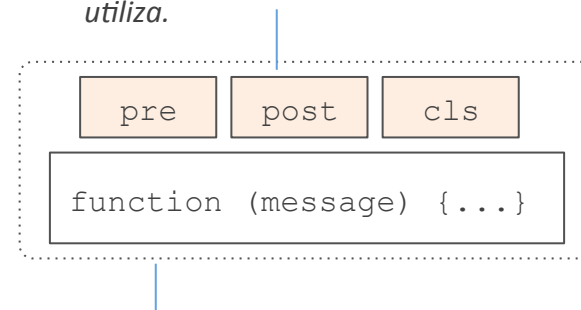
Las funciones que, por medio de los mecanismos de orden superior, devuelven funciones como resultado se pueden expresar en términos de variables locales o parámetros presentes en el entorno funcional donde se definen. La retención de variables es el mecanismo mediante el cual dichas variables y parámetros son mantenidos durante todo el tiempo de vida de la función de retorno. Este tipo de funciones se denominan clausuras.

```
function Logger(cls) {  
  var pre = 'Logger';  
  var post = '...';  
  return function (message) {  
    console.log ('%s[%s] - [%s]%s',  
                pre, cls, message, post);  
  }  
}
```

```
var log = Logger('My Script');  
log('starting');  
log(1234);  
log('end');
```

Variables Retenidas

Al extraer una función fuera de su ámbito léxico de definición se mantiene el contexto de variables y parámetros que dicha función utiliza.



Clausura

Las clausuras son un mecanismo de construcción funcional que captura de forma permanente un estado de configuración que condiciona su comportamiento

Javier Vélez Reyes
@javiervelezreyes
Javier.veler.reyes@gmail.com

3 *Técnicas de Programación Funcional*

- Abstracción
- Encapsulación
- Inmersión por Recursión y Acumulación
- Evaluación Parcial
- Composición Funcional & Composición Monádica

Programación Funcional en JavaScript
Técnicas de Programación Funcional

Programación Funcional en JavaScript

Técnicas de Programación Funcional

I. Introducción

A partir de los mecanismos que ofrece JavaScript como lenguaje de programación se puede escribir software bien centrado en el paradigma funcional. No obstante, para ello es necesario conocer una colección de técnicas básicas que describen formas canónicas de hacer frente a problemas recurrentes. A lo largo de este capítulo haremos una revisión de las técnicas más relevantes.

Abstracción

Las técnicas de abstracción permiten definir especificaciones funcionales con diferentes grados de generalidad y reutilización transversal

Encapsulación

En funcional la encapsulación de estado es un mal a veces necesario y en muchas ocasiones puede reemplazarse por encapsulación de comportamiento

Inmersión

Las técnicas de inmersión son una conjugación de abstracción, encapsulación y recursividad para obtener control de flujo dirigido por datos

Evaluación Parcial

La evaluación parcial permite evaluar una función en varias fases reduciendo paulatinamente el número de variables libres

Composición Funcional

La composición funcional permite articular procesos de secuenciamiento de funciones definidas como expresiones analíticas

Transformación Monádica

La transformación monádica persigue adaptar funciones para convertirlas en transformaciones puras que soporten procesos de composición

Inversión de Control

La inversión de control permite permutar entre las arquitecturas funcionales centradas en los datos y las centradas en las transformaciones

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

A. Técnicas de Abstracción

Una de las técnicas de programación funcional más comúnmente utilizadas, en alineamiento con los principios de diseño presentados anteriormente, es la abstracción funcional. Las tareas de abstracción deben entenderse como un proceso de transformación en el que la definición de una función se reexpresa en términos más generales para dar cobertura a un abanico más amplio de escenarios de aplicación. Podemos distinguir tres dimensiones de abstracción.

add

Suma convencional de dos números pasados como parámetros

```
add(x, y)
```

Abstracción en Anchura



addAll

Suma todos los parámetros de la función independientemente de la aridad

```
addAll(x, y, z, ...)
```

Abstracción en Alcance



reduceFrom

Combina mediante la función c todos los parámetros a partir de uno dado en posición p

```
reduceFrom(p, c)(x, y, ...)
```

Abstracción en Comportamiento



addFrom

Suma todos los parámetros de la función a partir de aquél que ocupa una posición p

```
add(p)(x, y, z, ...)
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

B. Técnicas de Encapsulación

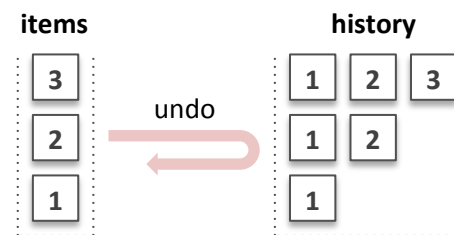
Mediante el uso de clausuras y la retención de variables es posible capturar, durante la fase de diseño o ejecución, cierta información que resulta de utilidad para adaptar el comportamiento de la función a lo largo del tiempo. Es posible distinguir entre encapsulación de estado y de comportamiento.

Encapsulación de Estado. Pila Undo

```
function Stack () {  
  var items = [];  
  var history = [];  
  return {  
    push: function (e) {  
      history.push([].concat(items));  
      items.push(e);  
    },  
    pop: function () {  
      history.push([].concat(items));  
      return items.pop();  
    },  
    undo: function () {  
      if (history.length > 0)  
        items = history.pop();  
    }  
  };  
}
```

Retención de Estado

Las variables que capturan el estado quedan retenidas dentro de la clausura lo que permite articular modelos de programación funcional que evolucionan con el tiempo



Dependencia de Estado

Ojo! Las funciones que dependen del estado son en realidad una mala práctica de programación de acuerdo a los principios de diseño del paradigma

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

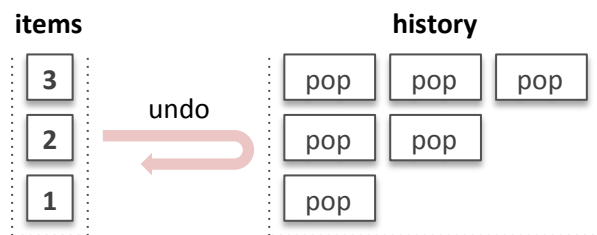
B. Técnicas de Encapsulación

Mediante el uso de clausuras y la retención de variables es posible capturar, durante la fase de diseño o ejecución, cierta información que resulta de utilidad para adaptar el comportamiento de la función a lo largo del tiempo. Es posible distinguir entre encapsulación de estado y de comportamiento.

Encapsulación de Comportamiento. Pila Undo

Retención de Comportamiento

En este caso la variable de histórico captura la colección de operaciones inversas que permiten deshacer las transacciones según han ido sucediendo



Operaciones Inversas

Dentro de cada operación, se registran en el histórico las operaciones inversas para luego invocarlas desde la operación deshacer

```
function Stack () {
  var items  = [];
  var history = [];
  return {
    push: function push (e) {
      history.push(function() {items.pop()});
      items.push(e);
    },
    pop: function pop () {
      var e = items.pop();
      history.push(function() {items.push(e)});
      return e;
    },
    undo: function undo() {
      if (history.length > 0)
        history.pop()(),
    }
  };
}
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

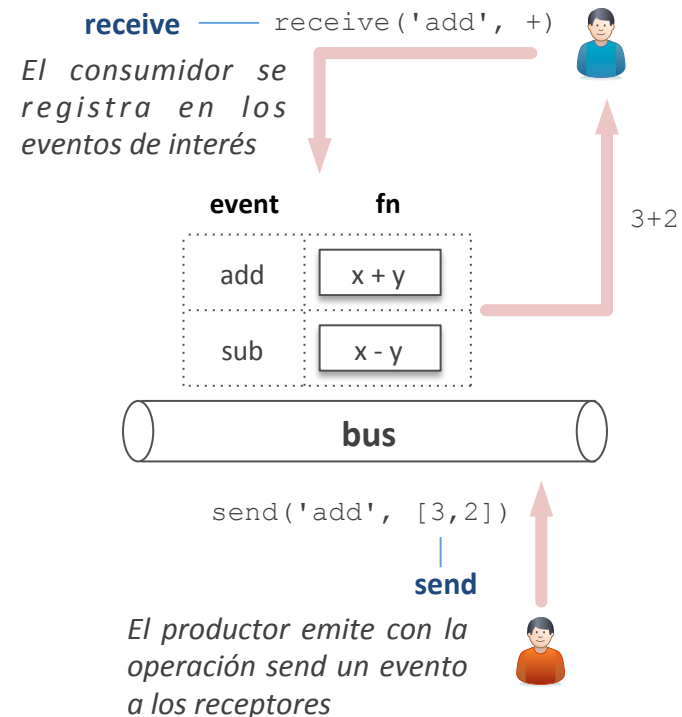
II. Técnicas de Programación Funcional

B. Técnicas de Encapsulación

Mediante el uso de clausuras y la retención de variables es posible capturar, durante la fase de diseño o ejecución, cierta información que resulta de utilidad para adaptar el comportamiento de la función a lo largo del tiempo. Es posible distinguir entre encapsulación de estado y de comportamiento.

Encapsulación de Comportamiento. Bus

```
function Bus () {  
  var fns = {};  
  return {  
    receive: function (e, fn) {  
      fns[e] = fn;  
    },  
    send: function (e, ctx) {  
      return fns[e].apply(null, ctx);  
    }  
  };  
}  
  
var bus = Bus();  
bus.receive('add', function (x,y) {return x+y;});  
bus.receive('sub', function (x,y) {return x-y;});  
bus.send('add', [3,2]);  
bus.send('sub', [7,3]);
```



Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

C. Diseño por Inmersión

Dado que la programación funcional no soporta los esquemas iterativos propios de la programación imperativa, es preciso orientar los cálculos a la recurrencia apoyándose para ello en parámetros auxiliares. Las técnicas de inmersión conjugan encapsulación y abstracción con el uso de parámetros auxiliares. Podemos distinguir diferentes tipos de inmersión.

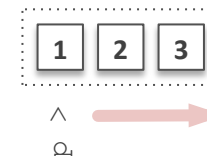
I. Inmersión por Recorrido

```
function find (v, e) {  
  var aux = function (v, e, p) {  
    return (p > v.length) ? -1 :  
           (v[p] === e)    ? p :  
           aux(v, e, p + 1);  
  };  
  return aux(v, e, 0);  
}
```

La inmersión de recorrido extiende la función introduciendo un nuevo parámetro cuyo propósito es llevar la cuenta del punto hasta el que se ha avanzado en el vector.

```
aux([1,2,3], 0) =  
1 + aux([1,2,3], 1) =  
1 + 2 + aux ([1,2,3], 2) =  
1 + 2 + 3 = 6
```

La técnica se apoya es una abstracción que queda sumergida dentro de la más específica por encapsulación



```
function addAll (v) {  
  return (function aux(v, p) {  
    if (p === v.length-1) return v[p];  
    else return v[p] + aux (v, p+1);  
  }) (v, 0);  
}
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

C. Diseño por Inmersión

Dado que la programación funcional no soporta los esquemas iterativos propios de la programación imperativa, es preciso orientar los cálculos a la recurrencia apoyándose para ello en parámetros auxiliares. Las técnicas de inmersión conjugan encapsulación y abstracción con el uso de parámetros auxiliares. Podemos distinguir diferentes tipos de inmersión.

II. Inmersión por Acumulación

```
function even (v) {  
  return (function aux(v, p, ac){  
    if (p === v.length) return ac;  
    else {  
      if (v[p] % 2 === 0) ac.push(v[p]);  
      return aux(v, p+1, ac);  
    }  
  })(v, 0, []);  
}
```

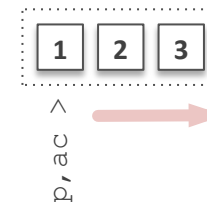
Reducción

Se utiliza la técnica de acumulación para consolidar un vector formado por el subconjunto de los elementos pares del vector

```
aux([1,2,3], 0, 0) =  
  aux([1,2,3], 1, 1) =  
    aux([1,2,3], 2, 3) =  
      aux([1,2,3], 3, 6) =  
        6
```

Tail Recursion

La acumulación se aplica aquí para obtener recursividad final buscando la eficiencia en memoria



```
function addAll (v) {  
  return (function aux(v, p, ac) {  
    if (p === v.length) return ac;  
    else return aux (v, p+1, v[p]+ac);  
  })(v, 0, 0);  
}
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

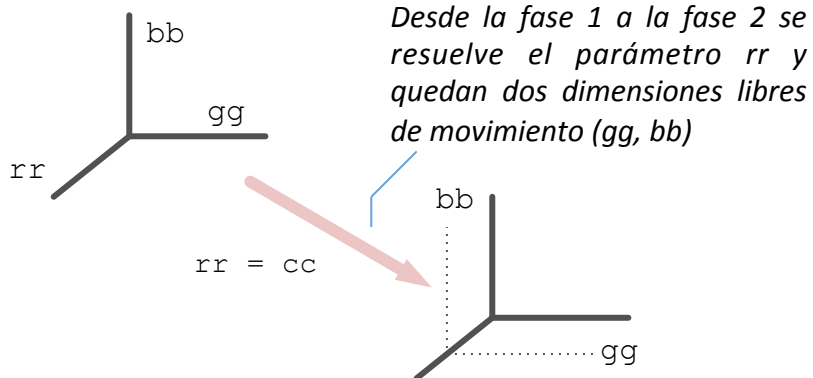
II. Técnicas de Programación Funcional

D. Evaluación Parcial

El uso de clausuras permite transformar una función para que pueda evaluarse de manera parcial resolviendo sólo algunos de sus parámetros. El resultado es una nueva función con menos grados de libertad donde los parámetros ya evaluados se fijan a un valor concreto y el resto quedan como variables libres.

I. Dimensiones Contractuales

A diferencia de lo que ocurre en otros paradigmas, el diseño funcional de abstracciones tiene dos dimensiones de definición, la dimensión espacial y la dimensión temporal.



En la dimensión temporal cada fase conduce a una etapa de resolución parcial de parámetros

La dimensión espacial vincula convencionalmente valores actuales a parámetros

```
function color(rr) {  
  return function (gg) {  
    return function (bb) {  
      return [rr, gg, bb] ;  
    };  
  };  
}
```

II. Reducción Dimensional

Cada fase de aplicación temporal de parámetros conduce a una reducción dimensional en el espacio del problema.

```
color('cc') ('a3') ('45')
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

D. Evaluación Parcial

El uso de la evaluación parcial permite que el programador vaya resolviendo de forma faseada la evaluación de una función a medida que va disponiendo de los argumentos actuales necesarios. En estos casos, el diseño de la función debe cuidarse para garantizar que el orden en el que se demandan los parámetros a lo largo del tiempo corresponde con el esperado en el contexto previsto de uso.

Ejemplo. Evaluación por Fases

```
function schema (scheme) {  
  var uri = { scheme : scheme };  
  return function (host) {  
    uri.host = host;  
    return function (port) {  
      uri.port = port;  
      return function (path) {  
        uri.path = path;  
        return function () {  
          return uri;  
        };  
      };  
    };  
  };  
}
```

Se articula un proceso de construcción por fases que permite inyectar las partes de una Uri. Este esquema recuerda al patrón Builder de OOP pero usando únicamente funciones

```
var host = schema('http');  
var port = host('foo.com');  
var path = port(80);  
var uri = path('index.html');  
uri();
```

Aunque la aplicación de este esquema induce un estricto orden en la resolución paramétrica permite resolver por fases el proceso constructivo a medida que se dispone de los datos

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

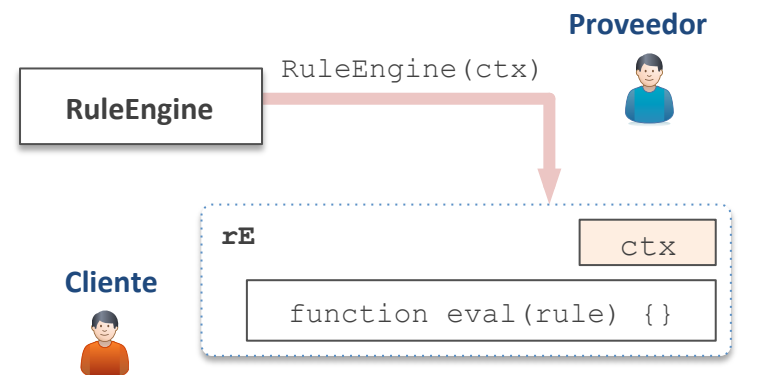
D. Evaluación Parcial

Otro escenario prototípico del uso de la evaluación parcial se da en la comunicación entre funciones proveedoras y clientes. El proveedor resuelve parcialmente una función reduciendo así el número de dimensiones y la entrega en respuesta a la demanda de un cliente que resolverá el resto de dimensiones con argumentos actuales en sucesivas invocaciones. En este caso, el diseño funcional debe fasearse teniendo en cuenta la intervención primera del proveedor y posterior del cliente.

Transferencia Proveedor - Consumidor

```
function RuleEngine (ctx) {  
  return function eval(rule) {  
    return function () {  
      var args = [].slice.call(arguments);  
      if (rule.trigger.apply(ctx, args))  
        return rule.action.apply(ctx, args);  
    };  
  };  
}  
var rE = RuleEngine({age: 18, login: false});
```

El proveedor resuelve parcialmente una función eval y la entrega como resultado a la función cliente



La función obtenida del proveedor permite al cliente evaluar reglas que se apoyan en las variables retenidas

```
rE ({ trigger: function (age) { return age > this.age; },  
      action: function () { return this.login = true; } })(19);  
rE ({ trigger: function () { return this.login; },  
      action: function () { return 'Bienvenido!'; } })();
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

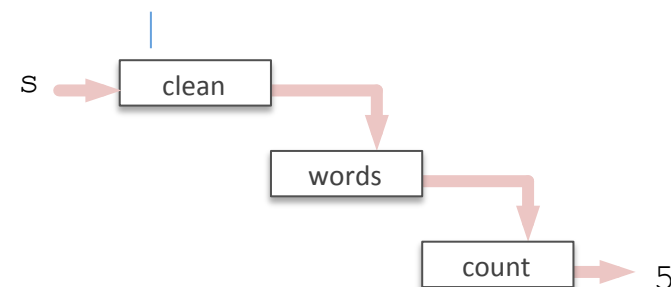
E. Composición Funcional

Dado que en programación funcional las abstracciones son meras transformaciones que no pueden articular un algoritmo secuencial, es frecuente modelar el secuenciamiento en base a la composición. De esta manera, el valor de retorno de una función sirve de entrada para la función subsiguiente.

```
function clean (s){ return s.trim(); }  
function words (s){ return s.split(' '); }  
function count (s){ return s.length; }  
  
count(  
  words(  
    clean('La FP en JS Mola!!!')  
  )  
);
```

La cascada de composición funcional se recorre en sentido inverso al natural. Los datos comienzan por la función más interna (clean) y atraviesan la cadena de composición ascendentemente hasta la cima (count). En el capítulo siguiente estudiaremos patrones que conducen a una lectura descendente más natural

La organización compositiva de funciones conduce a arquitecturas centradas en las transformaciones que los datos atraviesan en cascada



Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

E. Composición Funcional

En ocasiones la forma en la que se definen las abstracciones funcionales no es compatible con los procesos de composición. Para poder articular adecuadamente dichos procesos debemos tener en cuenta dos reglas de transformación funcional que puede ser necesario aplicar para conseguir funciones compositivas.

I. Dominio Simple

Dado que una función f devuelve un único valor de retorno, el número de parámetros que acepta una función g cuando se compone con f debe ser exactamente uno. Para mitigar este problema es posible encapsular los resultados de f en una estructura de datos y/o resolver todos los parámetros de g menos uno por medio de evaluación parcial

```
count(
  words(
    clean(s)
  )
);
```

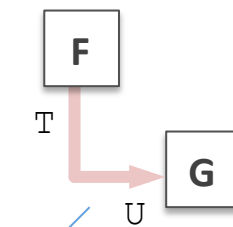
*[String] -> Number
String -> [String]
String -> String*

El tipo de salida T debe ser compatible con el tipo de entrada U . En términos OOP diríamos que T debe ser subtipo de U

```
neg(
  square(
    {stack:[3]}
  )
);
```



```
function square(r) {
  var e = r.stack.pop();
  return {
    value: e*e, stack: r.stack
  };
}
function neg(r) {
  r.stack.push(-r.value);
  return r;
}
```



II. Compatibilidad Rango-Dominio

El tipo del parámetro de entrada de una función debe ser compatible con el tipo del valor de retorno devuelto por la función anterior para que la composición pueda articularse con éxito

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

F. Transformación Monádica

El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

Ejemplo. Monada Escritor

Supongamos que tenemos una librería de funciones matemáticas de un solo parámetro que operan con números y devuelven un número. Por motivos de depuración estas funciones emiten mensajes a la consola

```
function inv (x) { console.log('invertir'); return 1/x; }  
function sqr (x) { console.log('cuadrado'); return x*x; }  
function inc (x) { console.log('incremento'); return x+1; }  
function dec (x) { console.log('decremento'); return x-1; }
```

```
function inv (x) { return { value: 1/x, log: ['invertir'] }; }  
function sqr (x) { return { value: x*x, log: ['cuadrado'] }; }  
function inc (x) { return { value: x+1, log: ['incrementar'] }; }  
function dec (x) { return { value: x-1, log: ['decrementar'] }; }
```

Dado que la traza por consola se considera un efecto colateral indeseable se transforman las funciones en puras parametrizando la traza como valor anexo de salida

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

F. Transformación Monádica

El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

Ejemplo. Monada Escritor

Ahora las funciones son puras pero han perdido su capacidad compositiva puesto que incumplen la regla de compatibilidad Rango-Dominio. Es necesario, en primer lugar, crear una función unit que eleve valores unitarios desde el tipo simple (Number) al tipo monádico (valor + log)

```
function unit (value) {  
  return {  
    value: value,  
    log  : []  
  };  
}
```

```
function bind (m, fn) {  
  var r = fn(m.value);  
  return {  
    value : r.value,  
    log   : m.log.concat(r.log)  
  };  
  return this;  
}
```

Ahora podemos crear una función bind que permita componer nuestras funciones anteriores sin necesidad de alterarlas. Esta función recibe un valor monádico (entrada + log acumulado) y una de nuestras funciones. La función bind primero desenvuelve la monada, después aplica la función pasada como parámetro y anexa la traza de la misma al log acumulado. Finalmente devuelve la estructura monádica con los resultados obtenidos

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

F. Transformación Monádica

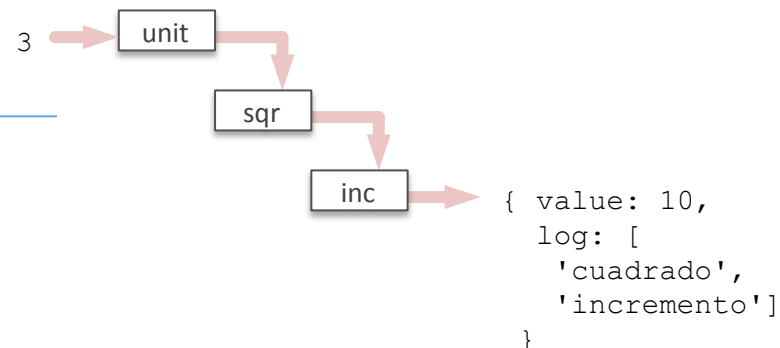
El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

Ejemplo. Monada Escritor

Con las funciones de transformación monádica unit & bind hemos creado una solución para garantizar la composición funcional eliminando los efectos colaterales de traza por pantalla

```
bind(  
  bind(  
    unit(3), sqr  
  ), neg  
);
```

El valor primitivo 3, se transforma primero al tipo monádico y luego se compone con la función sqr e inc por medio de la asistencia que proporciona la función bind



Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

F. Transformación Monádica

El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

Ejemplo. Monada Escritor

```
function Writer (value) {  
  this.value = value;  
  this.log    = [];  
}  
Writer.prototype.bind = function (fn) {  
  var m = fn(this.value);  
  var result = new Writer(m.value);  
  result.log = this.log.concat(m.log);  
  return result;  
};
```

Existen muchos tipos de efectos colaterales que pueden evitarse por medio de la aplicación de técnicas de composición monádica. Dado que cada tipo de efecto colateral requiere una lógica unit y bind específica es buena idea expresar estas técnicas como objetos donde unit se codifique como constructor y bind como método miembro del prototipo

Aunque desde un punto de vista pragmático esta operativa resulta cómoda y natural debemos ser conscientes de que se apoya en encapsulación de estado

```
var r = new Writer(3)  
  .bind(sqr)  
  .bind(inc);
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

G. Inversión de Control

La programación funcional permite articular dos tipos de arquitecturas complementarias. Las arquitecturas centradas en los datos fijan una colección de datos de entrada y hacen atravesar en torno a ellos una colección de transformaciones funcionales. Ortogonalmente, las arquitecturas centradas en la transformación establecen una cadena compositiva de funciones que es atravesada por diversas colecciones de datos. Las técnicas de inversión de control son una pasarela para convertir un esquema arquitectónico en el contrario y viceversa.

Arquitecturas Centradas en los Datos

```
function data (value) {  
  return {  
    value: value,  
    do: function (fn) {  
      this.value = fn(this.value);  
      return this;  
    }  
  };  
}
```

Las arquitecturas centradas en los datos fijan un conjunto de datos y permiten realizar transformaciones pasadas como parámetros en orden superior



```
data('La FP en JS Mola!!!')  
  .do(clean)  
  .do(words)  
  .do(count);
```

Programación Funcional en JavaScript

Técnicas de Programación Funcional

II. Técnicas de Programación Funcional

G. Inversión de Control

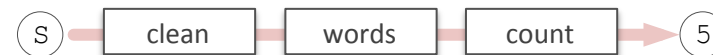
La programación funcional permite articular dos tipos de arquitecturas complementarias. Las arquitecturas centradas en los datos fijan una colección de datos de entrada y hacen atravesar en torno a ellos una colección de transformaciones funcionales. Ortogonalmente, las arquitecturas centradas en la transformación establecen una cadena compositiva de funciones que es atravesada por diversas colecciones de datos. Las técnicas de inversión de control son una pasarela para convertir un esquema arquitectónico en el contrario y viceversa.

Arquitecturas Centradas en la Transformación

```
function clean (s) { return s.trim();      }  
function words (s) { return s.split(' '); }  
function count (s) { return s.length;     }  
  
count(  
  words(  
    clean('La FP en JS Mola!!!')  
  )  
);
```

En el siguiente capítulo estudiaremos patrones dedicados a establecer transformaciones para convertir arquitecturas centradas en los datos a arquitecturas centradas en transformación y viceversa

Las arquitecturas centradas en la transformación establecen una cadena de composición de transformaciones funcionales por las que atraviesan distintas colecciones de datos



Javier Vélez Reyes
@javiervelezreyes
Javier.veler.reyes@gmail.com

4 *Patrones de Diseño Funcional*

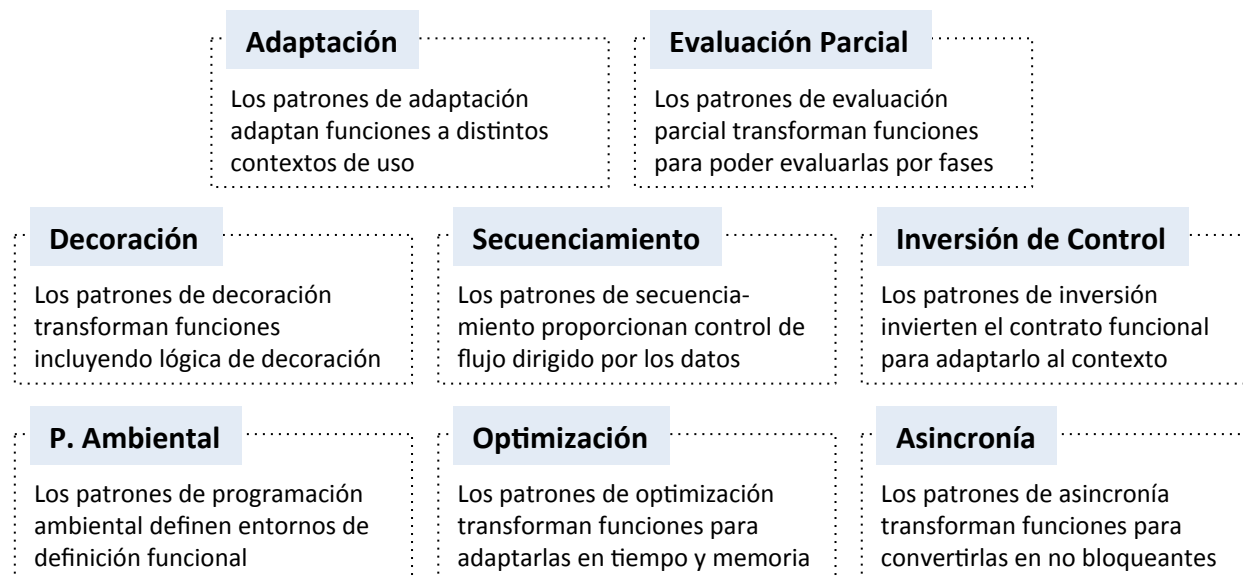
- Adaptación Funcional & Evaluación Parcial
- Decoración & Combinación
- Secuenciamiento & Inversión de Control
- Programación Ambiental & Frameworks Funcionales
- Optimización & Asincronía

Programación Funcional en JavaScript

Patrones de Diseño Funcional

I. Introducción

Las técnicas y mecanismos que se han descrito hasta ahora se presentan con frecuencia combinadas para dar respuesta a problemas que aparecen recurrentemente en los procesos de desarrollo de aplicaciones funcionales con objetivos claros y bien definidos. Esto da pie a identificar este tipo de prácticas como soluciones canónicas de validez probada que juegan el papel de patrones de diseño en el marco de la programación funcional. A lo largo de este capítulo describiremos 9 categorías de patrones funcionales.



Programación Funcional en JavaScript

Patrones de Diseño Funcional

II. Patrones de Adaptación Funcional

La adaptación funcional es una de las capacidades de la programación funcional que persigue transformar el prototipo de una función para adaptarla a las peculiaridades específicas del contexto de explotación. La aplicación de adaptaciones convierte a las arquitecturas funcionales en sistemas con alta plasticidad que permite alcanzar cotas elevadas de reutilización. En esta sección presentamos algunos patrones en este sentido.

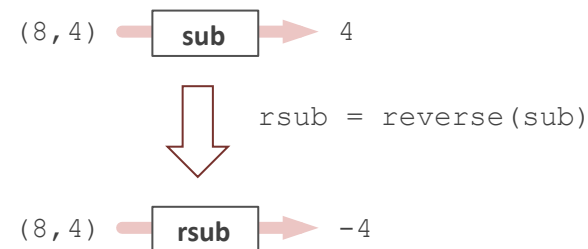
Patrón reverse

```
function reverse (fn) {  
  return function () {  
    var args = [].slice.call(arguments);  
    var iargs = [].concat(args).reverse();  
    return fn.apply(this, iargs);  
  };  
}
```

```
var add = function (x, y) { return x + y; };  
var sub = function (x, y) { return x - y; };  
  
var radd = reverse (add);  
var rsub = reverse (sub);
```

```
radd (2, 8) === add (2, 8); // true  
radd (2, 8) === add (8, 2); // true  
rsub (2, 8) === sub (2, 8); // false  
rsub (2, 8) === sub (8, 2); // true
```

El patrón reverse adapta una función pasada como parámetro para invertir el orden de sus parámetros de entrada



La inversión de parámetros no afecta a las operaciones conmutativas (add) pero sí a las que no cumplen esta propiedad (sub)

Programación Funcional en JavaScript

Patrones de Diseño Funcional

II. Patrones de Adaptación Funcional

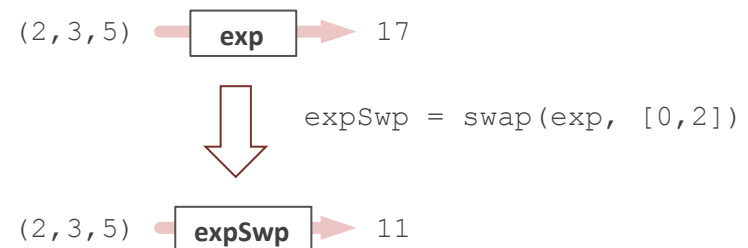
La adaptación funcional es una de las capacidades de la programación funcional que persigue transformar el prototipo de una función para adaptarla a las peculiaridades específicas del contexto de explotación. La aplicación de adaptaciones convierte a las arquitecturas funcionales en sistemas con alta plasticidad que permite alcanzar cotas elevadas de reutilización. En esta sección presentamos algunos patrones en este sentido.

Patrón swap

```
function swap (fn, p) {  
  return function () {  
    var args = [].slice.call (arguments);  
    var temp = args[p[0]];  
    args[p[0]] = args[p[1]];  
    args[p[1]] = temp;  
    return fn.apply(this, args);  
  };  
}
```

```
var exp = function (x, y, z){  
  return x + y * z;  
};  
var expSwp1 = swap (exp, [0, 1]);  
var expSwp2 = swap (exp, [0, 2]);  
exp(2,3,5);      // 17  
expSwp1(2,3,5);  // 13  
expSwp2(2,3,5);  // 11
```

El patrón swap adapta una función pasada como parámetro para invertir el orden de un par de parámetros de entrada



La inversión de parámetros permite reinterpretar expresiones de manera transparente para el usuario externo. Veremos su utilidad más adelante

Programación Funcional en JavaScript

Patrones de Diseño Funcional

II. Patrones de Adaptación Funcional

La adaptación funcional es una de las capacidades de la programación funcional que persigue transformar el prototipo de una función para adaptarla a las peculiaridades específicas del contexto de explotación. La aplicación de adaptaciones convierte a las arquitecturas funcionales en sistemas con alta plasticidad que permite alcanzar cotas elevadas de reutilización. En esta sección presentamos algunos patrones en este sentido.

Patrón arity

```
function arity (n) {  
  return function (fn) {  
    return function () {  
      var args = [].slice.call(arguments, 0, n-1);  
      return fn.apply(this, args);  
    };  
  };  
}
```

```
var unary = arity (1);  
['1', '2', '3'].map(parseInt);           // Error  
['1', '2', '3'].map(unary(parseInt));    // Ok
```

El patrón arity adapta una función de manera que acepte exactamente un número específico de parámetros ignorando el resto

$f(2,3,5)$  $f(2)$

La limitación del número de parámetros que puede aceptar una función puede resultar muy útil a veces. En el ejemplo, la función parseInt acepta un String y opcionalmente la base (radix). Sin embargo queremos que parseInt consuma exactamente un parámetro cada vez que es invocado para recorrer adecuadamente el array

Programación Funcional en JavaScript

Patrones de Diseño Funcional

II. Patrones de Adaptación Funcional

La adaptación funcional es una de las capacidades de la programación funcional que persigue transformar el prototipo de una función para adaptarla a las peculiaridades específicas del contexto de explotación. La aplicación de adaptaciones convierte a las arquitecturas funcionales en sistemas con alta plasticidad que permite alcanzar cotas elevadas de reutilización. En esta sección presentamos algunos patrones en este sentido.

Patrón variadic

```
function variadic (fn) {  
  return function () {  
    var args = [].slice.call(arguments);  
    var other = args.splice(0, fn.length-1);  
    return fn.apply(this, other.concat([args]));  
  };  
}
```

```
var basket = variadic (  
  function (date, user, products){  
    console.log('[%s] - %s:', date, user);  
    console.log(products);  
  });  
basket(  
  'hoy', 'jvelez',  
  'platanos', 'manzanas', 'peras');
```

Pretendemos imitar la capacidad de elipsis (args...) que disponen otros lenguajes como último parámetro. Diseñamos una función incluyendo un último parámetro args:

```
function foo(x, y, args) {...}
```

foo(2, 3, 5, 8) → x = 2
y = 3
args = 5



```
vfoo = variadic(foo)
```

vfoo(2, 3, 5, 8) → x = 2
y = 3
args = [5, 8]

Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Patrones de Evaluación Parcial

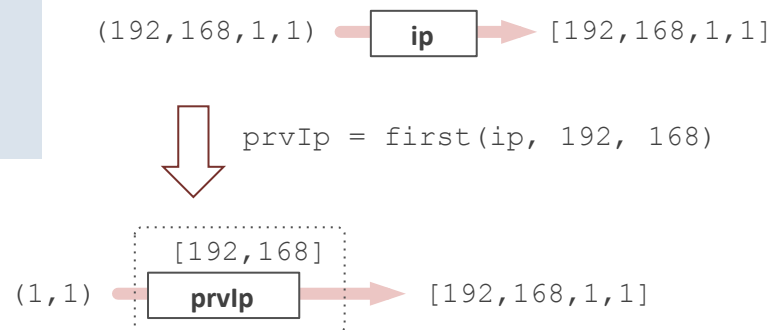
La evaluación parcial ofrecen estrategias de transformación funcional que permiten convertir una función completa en otra que se evalúa por fases. Como ya discutimos en el capítulo 3, esta técnica resulta muy útil para reducir la dimensión de una transformación de forma progresiva y articular procesos de comunicación entre funciones proveedoras y funciones cliente. En esta sección describimos patrones de evaluación parcial.

Patrón first

```
function first () {  
  var fn = arguments [0];  
  var params = [].slice.call(arguments, 1);  
  return function () {  
    var args = [].slice.call(arguments);  
    return fn.apply(this, params.concat(args));  
  };  
}
```

```
var ip = function (a, b, c, d) {  
  return [a, b, c, d];  
};  
var prvIp = first(ip, 192, 168);  
var pubIp = first(ip, 62, 27);  
prvIp(15, 12);    // [192,168,15,12]  
pubIp(23, 31);    // [62,27,23,31]
```

El patrón first adapta una función resolviendo los n primeros parámetros de la misma



Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Patrones de Evaluación Parcial

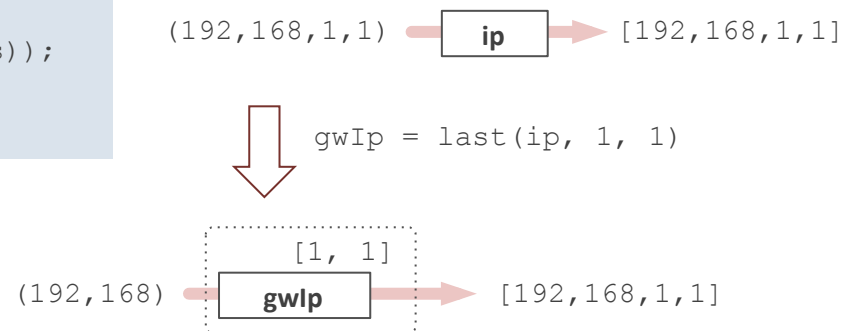
La evaluación parcial ofrecen estrategias de transformación funcional que permiten convertir una función completa en otra que se evalúa por fases. Como ya discutimos en el capítulo 3, esta técnica resulta muy útil para reducir la dimensión de una transformación de forma progresiva y articular procesos de comunicación entre funciones proveedoras y funciones cliente. En esta sección describimos patrones de evaluación parcial.

Patrón last

```
function last () {  
  var fn = arguments[0];  
  var params = [].slice.call(arguments, 1);  
  return function () {  
    var args = [].slice.call(arguments);  
    return fn.apply(this, args.concat(params));  
  };  
}
```

```
var gwIp = last(ip, 1, 1);  
var prvGw = last(prvIp, 1, 1);  
var pubGw = last(pubIp, 1, 1);  
gwIp(192, 168); // [192,168,1,1]  
prvGw();       // [192,168,1,1]  
pubGw();       // [62,27,1,1]
```

De manera complementaria, el patrón last adapta una función resolviendo los *n* últimos parámetros de la misma



Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Patrones de Evaluación Parcial

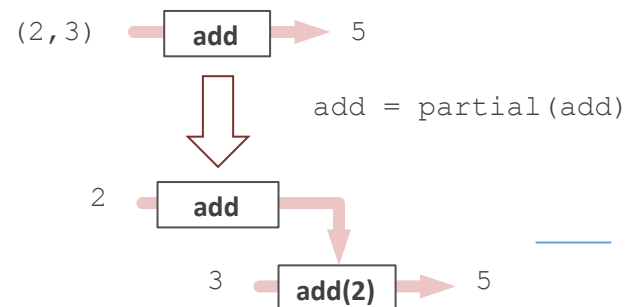
La evaluación parcial ofrecen estrategias de transformación funcional que permiten convertir una función completa en otra que se evalúa por fases. Como ya discutimos en el capítulo 3, esta técnica resulta muy útil para reducir la dimensión de una transformación de forma progresiva y articular procesos de comunicación entre funciones proveedoras y funciones cliente. En esta sección describimos patrones de evaluación parcial.

Patrón partial & rpartial

```
function partial (fn) {  
  return function (x) {  
    return function (y) {  
      return fn.call(this, x, y);  
    };  
  };  
}  
  
function rpartial (fn) {  
  return function (x) {  
    return function (y) {  
      return fn.call(this, y, x);  
    };  
  };  
}
```

El patrón partial puede evaluarse a izquierdas o a derechas

```
var add = partial(function (x, y) { return x + y });  
var sub = rpartial(function (x, y) { return x - y });  
var inc = add(1);  
var dec = sub(1);  
  
[inc(3), dec(4)];
```



El patrón partial proyecta en la dimensión temporal una función binaria en dos fases de evaluación

Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Patrones de Evaluación Parcial

La evaluación parcial ofrecen estrategias de transformación funcional que permiten convertir una función completa en otra que se evalúa por fases. Como ya discutimos en el capítulo 3, esta técnica resulta muy útil para reducir la dimensión de una transformación de forma progresiva y articular procesos de comunicación entre funciones proveedoras y funciones cliente. En esta sección describimos patrones de evaluación parcial.

Patrón curry

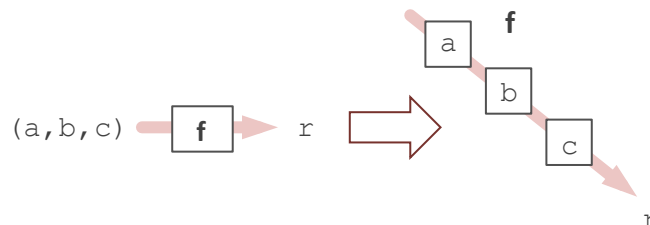
```
function curry (fn) {  
  return (function aux (args) {  
    if (args.length >= fn.length) {  
      return fn.apply(this, args);  
    }  
    else return function () {  
      var nargs = [].slice.call(arguments);  
      return aux(args.concat(nargs));  
    };  
  }) ([]);  
}
```

La aplicación de los parámetros a la función currificada puede aplicarse de forma flexible cuando se dispongan sus valores. Eso si, los parámetros siempre deben aplicarse en orden

```
var Ip = function (a, b, c, d) {  
  return [a, b, c, d];  
};  
var cIp = curry(Ip);
```

```
[ cIp(192, 168, 1, 1),  
  cIp(192, 168, 1)(1),  
  cIp(192, 168)(1, 1),  
  cIp(192, 168)(1)(1),  
  cIp(192)(168, 1, 1),  
  cIp(192)(168, 1)(1)]
```

El patrón curry extiende el patrón parcial para aplicarlo a funciones con cualquier número de parámetros



Programación Funcional en JavaScript

Patrones de Diseño Funcional

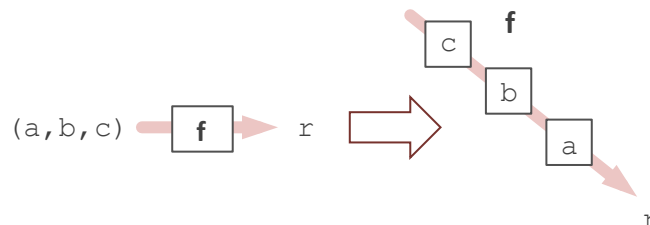
III. Patrones de Evaluación Parcial

La evaluación parcial ofrecen estrategias de transformación funcional que permiten convertir una función completa en otra que se evalúa por fases. Como ya discutimos en el capítulo 3, esta técnica resulta muy útil para reducir la dimensión de una transformación de forma progresiva y articular procesos de comunicación entre funciones proveedoras y funciones cliente. En esta sección describimos patrones de evaluación parcial.

Patrón rcurry

```
function curry (fn) {  
  return (function aux (args) {  
    if (args.length >= fn.length) {  
      return fn.apply(this, args);  
    }  
    else return function () {  
      var nargs = [].slice.call(arguments);  
      return aux(nargs.concat(args));  
    };  
  }) ([]);  
}
```

El patrón rcurry extiende el patrón partial para aplicarlo a funciones con cualquier número de parámetros



El patrón rcurry funciona de manera inversa a curry pero tiene una forma de uso particular cada fase rellena parámetros por el final pero el orden de aplicación de ellos en cada fase es de izquierda a derecha

```
var Ip = function (a, b, c, d) {  
  return [a, b, c, d];  
};  
var rcIp = rcurry(Ip);
```

```
[ rcIp(192, 168, 1, 1),  
  rcIp(168, 1, 1)(192),  
  rcIp(1, 1)(192, 168),  
  rcIp(1, 1)(168)(192),  
  rcIp(1)(192, 168, 1),  
  rcIp(1)(168, 1)(192)]
```


Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Patrones de Evaluación Parcial

La evaluación parcial ofrecen estrategias de transformación funcional que permiten convertir una función completa en otra que se evalúa por fases. Como ya discutimos en el capítulo 3, esta técnica resulta muy útil para reducir la dimensión de una transformación de forma progresiva y articular procesos de comunicación entre funciones proveedoras y funciones cliente. En esta sección describimos patrones de evaluación parcial.

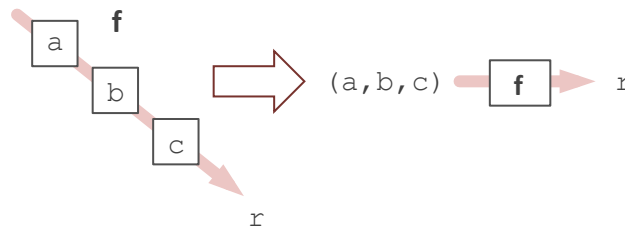
Patrón uncurry

```
function uncurry (fn) {  
  return function () {  
    var args = [].slice.call(arguments);  
    return args.reduce(function (ac, arg){  
      return (ac = ac(arg));  
    }, fn);  
  };  
}
```

Si tomamos una función en aplicación parcial y le aplicamos el algoritmo de descurrifcación obtenemos una función evaluable en una sola fase

```
var cadd = function (x) {  
  return function (y) {  
    return function (z) {  
      return x + y + z;  
    };  
  };  
};  
  
var add = uncurry (cadd);  
cadd(1) (2) (3); // 6  
add(1,2,3);     // 6
```

El patrón uncurry deshace una currifcación para obtener una función con una única fase de evaluación



Programación Funcional en JavaScript

Patrones de Diseño Funcional

IV. Patrones de Decoración & Combinación

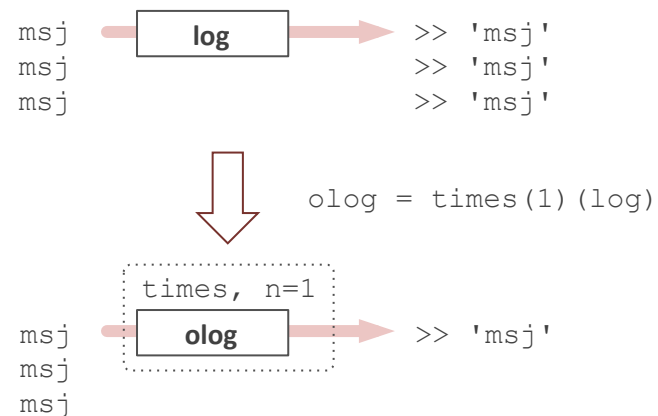
Las capacidades adaptativas de la programación funcional permiten definir nuevas funciones resultantes de un proceso de decoración de funciones pasadas en orden superior o de una combinación estratégica de dos funciones. Esto ofrece posibilidades de intercesión y transformación funcional que flexibilizan el uso de abstracciones funcionales en diferentes contextos de uso. En esta sección presentamos algunos patrones de este tipo.

Patrón times

```
function times (n) {  
  return function (fn) {  
    var times = 0;  
    return function () {  
      if (times < n) {  
        times++;  
        return fn.apply(this, arguments);  
      }  
    };  
  };  
}
```

```
var log = function (msg) {  
  console.log(msg)  
};  
var olog = times(1)(log);  
olog('JS Mola!'); >> JS Mola!  
olog('JS Mola!');
```

El decorador times controla la ejecución de una función limitando el número máximo de invocaciones



Programación Funcional en JavaScript

Patrones de Diseño Funcional

IV. Patrones de Decoración & Combinación

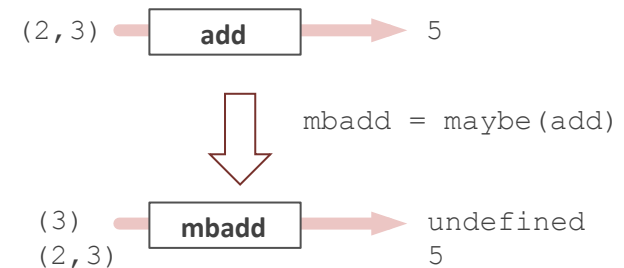
Las capacidades adaptativas de la programación funcional permiten definir nuevas funciones resultantes de un proceso de decoración de funciones pasadas en orden superior o de una combinación estratégica de dos funciones. Esto ofrece posibilidades de intercesión y transformación funcional que flexibilizan el uso de abstracciones funcionales en diferentes contextos de uso. En esta sección presentamos algunos patrones de este tipo.

Patrón maybe

```
function maybe (fn) {  
  return function () {  
    var args = [].slice.call(arguments);  
    var callable = arguments.length >= fn.length &&  
      args.every(function (p) {  
        return (p !== null);  
      });  
    if (callable) return fn.apply(this, args);  
  };  
}
```

```
var add = function (x, y) { return x + y; };  
var mbadd = maybe (add, 2);  
  
console.log(mbadd(3));      // undefined  
console.log(mbadd(2, 3));  // 5
```

— El patrón maybe comprueba si el número de parámetros formales coincide con el de argumentos proporcionados y si ninguno de ellos es null. Sólo en ese caso invoca la función



Programación Funcional en JavaScript

Patrones de Diseño Funcional

IV. Patrones de Decoración & Combinación

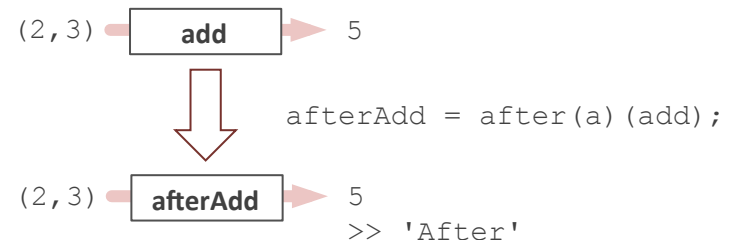
Las capacidades adaptativas de la programación funcional permiten definir nuevas funciones resultantes de un proceso de decoración de funciones pasadas en orden superior o de una combinación estratégica de dos funciones. Esto ofrece posibilidades de intercesión y transformación funcional que flexibilizan el uso de abstracciones funcionales en diferentes contextos de uso. En esta sección presentamos algunos patrones de este tipo.

Patrón before & after

```
function before (dn) {
  return function (fn) {
    return function () {
      dn.apply(this, arguments);
      return fn.apply(this, arguments);
    };
  };
}

function after (dn) {
  return function (fn) {
    return function () {
      var r = fn.apply(this, arguments);
      dn.apply(this, arguments);
      return r;
    };
  };
}
```

Los patrones before y after ejecutan una función decorada antes y después de ejecutar la función que se pasa como parámetro en una segunda fase de evaluación



```
var add = function (x, y) { return x + y; };
var b = function () { console.log ('Before'); };
var a = function () { console.log ('After'); };
var beforeAdd = before(b)(add);
var afterAdd = before(a)(add);
beforeAdd (2,3); // >> 'Before' 5
afterAdd (2,3); // 5 >> 'After'
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

IV. Patrones de Decoración & Combinación

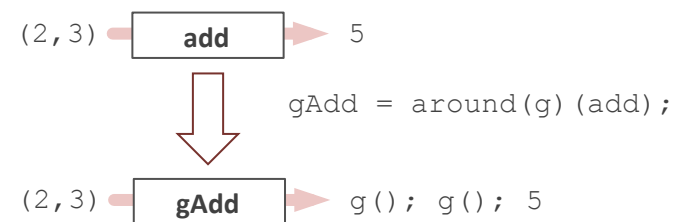
Las capacidades adaptativas de la programación funcional permiten definir nuevas funciones resultantes de un proceso de decoración de funciones pasadas en orden superior o de una combinación estratégica de dos funciones. Esto ofrece posibilidades de intercesión y transformación funcional que flexibilizan el uso de abstracciones funcionales en diferentes contextos de uso. En esta sección presentamos algunos patrones de este tipo.

Patrón around

```
function around (dn) {  
  return function (fn) {  
    return function () {  
      var args = [].slice.call(arguments);  
      args = [dn.bind(this)].concat(args);  
      return fn.apply(this, args);  
    };  
  };  
}
```

```
var add = function (around, x, y) {  
  around();  
  x = x + y;  
  around();  
  return x;  
};  
var gAdd = around(g) (add);  
gAdd(2,3);
```

El patrón around permite diseñar funciones que reciben como primer parámetro una función que representa un punto de intercesión donde se inyecta código desde el exterior



Dentro de la definición de la función add, around () supone una invocación a la función g, que es inyectada por el cliente

Programación Funcional en JavaScript

Patrones de Diseño Funcional

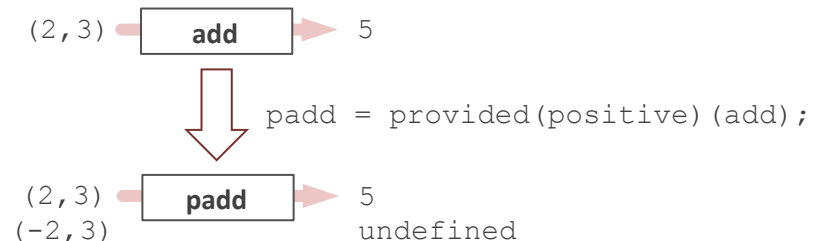
IV. Patrones de Decoración & Combinación

Las capacidades adaptativas de la programación funcional permiten definir nuevas funciones resultantes de un proceso de decoración de funciones pasadas en orden superior o de una combinación estratégica de dos funciones. Esto ofrece posibilidades de intercesión y transformación funcional que flexibilizan el uso de abstracciones funcionales en diferentes contextos de uso. En esta sección presentamos algunos patrones de este tipo.

Patrón provided & except

```
function provided (pn) {  
  return function (fn) {  
    return function () {  
      if (pn.apply(this, arguments))  
        return fn.apply(this, arguments);  
    };  
  };  
}  
  
function except (pn) {  
  return function (fn) {  
    return function () {  
      if (!pn.apply(this, arguments))  
        return fn.apply(this, arguments);  
    };  
  };  
}
```

Los patrones provided y except ejecutan condicionalmente una función cuando se satisface o no, respectivamente un predicado lógico pasado como argumento



```
var add = function (x, y) { return x+y; };  
var positive = function (x, y) { return x*y>0; };  
var padd = provided(positive)(add);  
var eadd = except(positive)(add);  
  
[padd(2, 3), padd(-2, 3)]; // [5, undefined]  
[eadd(2, 3), eadd(-2, 3)]; // [undefined, 1]
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

Patrón `forEach`

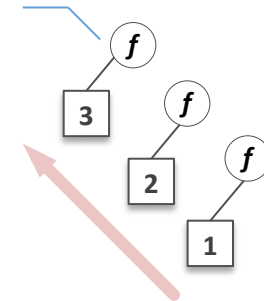
```
function forEach (data, fn, self) {  
  (function aux (data, index, fn) {  
    fn.call(self, data[index], index, data);  
    if (index < data.length - 1)  
      aux(data, index + 1, fn);  
  })(data, 0, fn);  
}
```

```
forEach ([1,2,3], function (item, idx) {  
  console.log(idx, item);  
});
```

La función manejadora sigue un contrato basado en tres parámetros. El primero corresponde con el elemento en curso, el segundo con el índice que ocupa dentro de la colección y el tercero con la propia colección

La función *fn* se aplica secuencialmente a todos los elementos de la colección desde el primero al último

El patrón `forEach` encapsula una inmersión por recorrido de la colección pasada como primer parámetro



Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

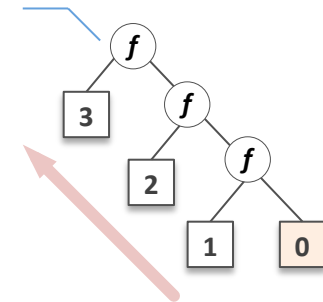
Patrón reduce

```
function reduce (data, fn, base, self) {  
  var ac = base;  
  forEach(data, function(e, i, data) {  
    ac = fn.call(self, ac, data[i], i, data);  
  }, self);  
  return ac;  
}
```

```
reduce ([1,2,3], function (a, e) {  
  return e + a;  
}, 0);
```

La función manejadora sigue un contrato basado en cuatro parámetros. El primero corresponde con el acumulador, el segundo con el elemento en curso, el tercero con el índice que ocupa dentro de la colección y el último con la propia colección

La función *f* combina secuencialmente cada elemento de la colección tomando el 0 como valor inicial



El patrón reduce encapsula una inmersión por acumulación de la colección pasada como primer parámetro. La implementación interna se apoya en *forEach*

Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

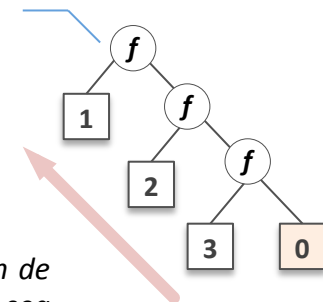
Patrón rReduce

```
function rReduce (data, fn, base, self) {  
  var iData= [].concat(data).reverse();  
  return reduce(iData, fn, base, self);  
}
```

```
rReduce ([1,2,3], function (a, e) {  
  return e + a;  
}, 0);
```

La función manejadora sigue un contrato basado en cuatro parámetros. El primero corresponde con el acumulador, el segundo con el elemento en curso, el tercero con el índice que ocupa dentro de la colección y el último con la propia colección

La función *f* combina secuencialmente en orden inverso cada elemento de la colección tomando el 0 como valor inicial



El carácter asociativo de la operación de suma provoca que el resultado sea equivalente al anterior. Su implementación interna aplica un reduce sobre la inversión del orden de la colección

Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

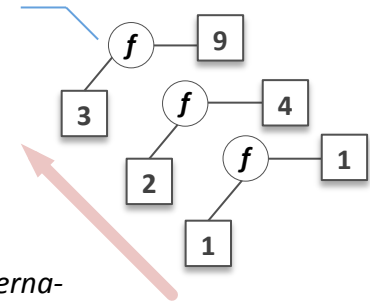
Patrón map

```
function map (data, fn, self) {  
  return reduce(data, function (ac, item) {  
    return ac.concat(fn.call(self, item));  
  }, [], self);  
}
```

```
map([1,2,3], function (e) {  
  return e * e;  
});
```

La función manejadora sigue un contrato basado en tres parámetros. El primero corresponde con el elemento en curso, el segundo con el índice que ocupa dentro de la colección y el tercero con la propia colección

La función *f* se aplica secuencialmente a cada elemento de la colección y genera como resultado una nueva colección transformada



El patrón map se implementa internamente aplicando una reducción. La estrategia de acumulación consiste en partir de un array vacío e ir añadiendo a cada paso cada ítem transformado de la colección al acumulador

Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

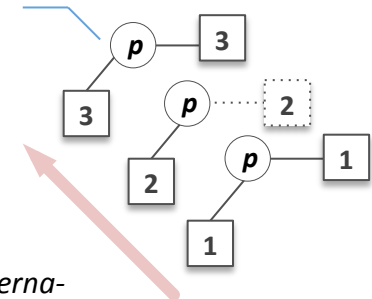
Patrón filter

```
function filter (data, fn, self) {  
  return reduce(data,  
    function (ac, e, i, data) {  
      if (fn.call(self, e, i, data))  
        ac.push(e);  
      return ac;  
    }, [], self);  
}
```

```
filter([1,2,3], function (e){  
  return e % 2 !== 0;  
});
```

La función manejadora es un predicado lógico que sigue un contrato basado en tres parámetros. El primero corresponde con el elemento en curso, el segundo con el índice que ocupa dentro de la colección y el tercero con la propia colección

El predicado *p* se aplica secuencialmente a cada elemento de la colección y, si se evalúa a cierto, lo incluye en la colección de salida



El patrón filter se implementa internamente aplicando una reducción. La estrategia de acumulación consiste en partir de un array vacío e ir añadiendo a cada paso cada ítem si se supera el predicado lógico

Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

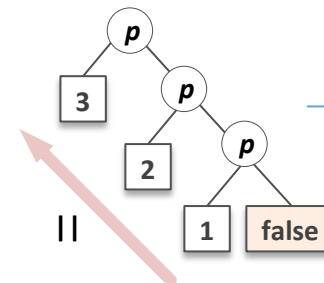
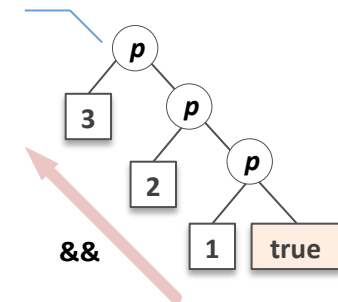
La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

Patrón every & some

```
function every (data, fn, self) {  
  return reduce (data,  
    function (ac, e, i, data) {  
      return ac && fn.call(self, e, i, data);  
    }, true, self);  
}  
  
function some(data, fn, self) {  
  return reduce (data,  
    function (ac, e, i, data) {  
      return ac || fn.call(self, e, i, data);  
    }, false, self);  
}
```

```
var p = function (e){return e < 4; };  
every([1,2,3], p);  
some ([1,2,3], p);
```

El predicado p se aplica secuencialmente a cada elemento de la colección y combina cada resultado a través de la conjunción lógica



El predicado p se aplica secuencialmente a cada elemento de la colección y combina cada resultado a través de la disyunción lógica

Programación Funcional en JavaScript

Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

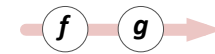
La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

Patrón compose & sequence

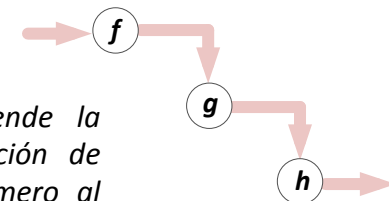
```
function compose (fn, gn) {  
  return function (x) {  
    return fn(gn(x));  
  };  
}  
  
function sequence (fns) {  
  return function (x) {  
    return fns.reduce (function (ac, fn) {  
      return fn(ac);  
    }, x);  
  };  
}
```

```
function clean (s){ return s.trim();    }  
function words (s){ return s.split(' '); }  
function count (s){ return s.length;    }  
compose(count,  
  compose(words, clean)) ('NodeJS Mola');
```

El patrón compose es una función de orden superior que das dos funciones devuelve la función compuesta



El patrón sequence extiende la composición a una colección de funciones tomada del primero al último



```
sequence([  
  clean,  
  words,  
  count  
) ('La FP en JS Mola');
```

Programación Funcional en JavaScript

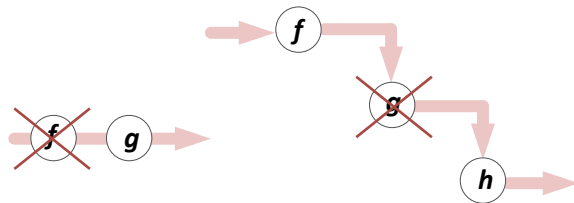
Patrones de Diseño Funcional

V. Patrones de Composición & Secuenciamiento

La programación funcional está basada en la construcción de abstracciones descritas como expresiones funcionales. El concepto de secuenciamiento algorítmico y de control de flujo, tal y como se entiende en la programación imperativa, no es de aplicación en este paradigma. Los patrones de secuenciamiento desvelan formas con las que es posible emular hasta cierto punto un estilo de programación secuencial con directivas de control de flujo.

Patrón composeBreak & sequenceBreak

Este par de patrones extiende los anteriores de manera que si alguna función de la cadena devuelve null desiste de terminar la composición



```
function sqr(x){ return x * x ; }
function grt(x){ if (x > 3) return x; }

var c = composeBreak(sqr, grt);
[c(3), c(4)]; // [undefined, 16]
```

```
function composeBreak (fn, gn) {
  return function (x) {
    var r = gn(x);
    return (r !== void 0) ? fn(r) : void 0;
  };
}

function sequenceBreak (fns) {
  return function (x) {
    return fns.reduce (function (ac, fn) {
      return (ac !== void 0) ? fn(ac) : void 0;
    }, x);
  };
}
```

```
var s = sequenceBreak ([
  function sqr (x) { return x * x ; },
  function grt (x) { if (x > 10) return x; },
  function inc (x) { return x + 1 ; },
]);

[s(3), s(4)]; // [undefined, 17]
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

VI. Patrones de Inversión de Control

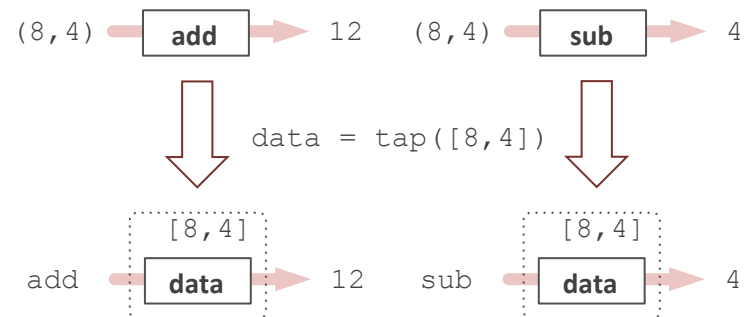
Como describimos en la introducción existen dos modelos arquitectónicos complementarios dentro de la programación funcional. Las arquitecturas centradas en los datos fijan un conjunto de datos y sobre él aplican una cadena de transformaciones. Complementariamente es posible invertir este modelo de control y pasar a un modelo arquitectónico donde una cadena funcional queda fija y son los datos los que la atraviesan.

Patrón tap

```
function tap (args) {  
  return function (fn) {  
    return fn.apply(this, args);  
  };  
}
```

```
var add = function (x, y) { return x + y; };  
var sub = function (x, y) { return x - y; };  
var mul = function (x, y) { return x * y; };  
var div = function (x, y) { return x / y; };  
  
var data = tap([8,4]);  
[data(add), data(sub), data(mul), data(div)]
```

El patrón *tap* invierte el control sobre la invocación funcional de manera que primero se captura un conjunto de datos y luego se pueden aplicar diferentes funciones sobre ellos



Programación Funcional en JavaScript

Patrones de Diseño Funcional

VI. Patrones de Inversión de Control

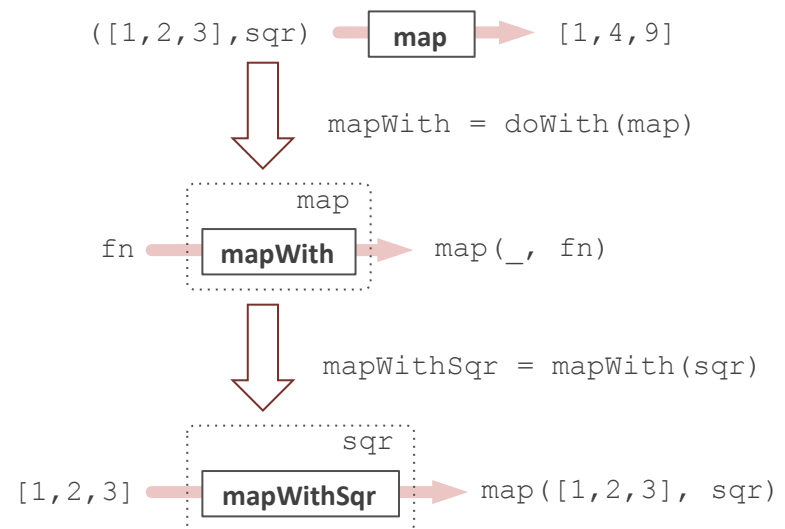
Como describimos en la introducción existen dos modelos arquitectónicos complementarios dentro de la programación funcional. Las arquitecturas centradas en los datos fijan un conjunto de datos y sobre él aplican una cadena de transformaciones. Complementariamente es posible invertir este modelo de control y pasar a un modelo arquitectónico donde una cadena funcional queda fija y son los datos los que la atraviesan.

Patrón doWith & {map, reduce, filter, every, some}With

```
function doWith (fn) {  
  return function (hn) {  
    return function () {  
      var args = [].slice.call(arguments);  
      args.splice(1, 0, hn);  
      return fn.apply(this, args);  
    };  
  };  
}  
  
var mapWith    = doWith(map);  
var reduceWith = doWith(reduce);  
var filterWith = doWith(filter);  
var everyWith  = doWith(every);  
var someWith   = doWith(some);
```

```
var sqr = function (x) { return x*x; };  
var mapWithSqr = mapWith(sqr);  
mapWithSqr([1,2,3]) // [1,4,9]
```

El patrón doWith cambia el orden de aplicación de los parámetros colección y manejador de las funciones de secuenciamiento (map, reduce, filter...) aplicando evaluación parcial para obtener inversión de control



Programación Funcional en JavaScript

Patrones de Diseño Funcional

VI. Patrones de Inversión de Control

Como describimos en la introducción existen dos modelos arquitectónicos complementarios dentro de la programación funcional. Las arquitecturas centradas en los datos fijan un conjunto de datos y sobre él aplican una cadena de transformaciones. Complementariamente es posible invertir este modelo de control y pasar a un modelo arquitectónico donde una cadena funcional queda fija y son los datos los que la atraviesan.

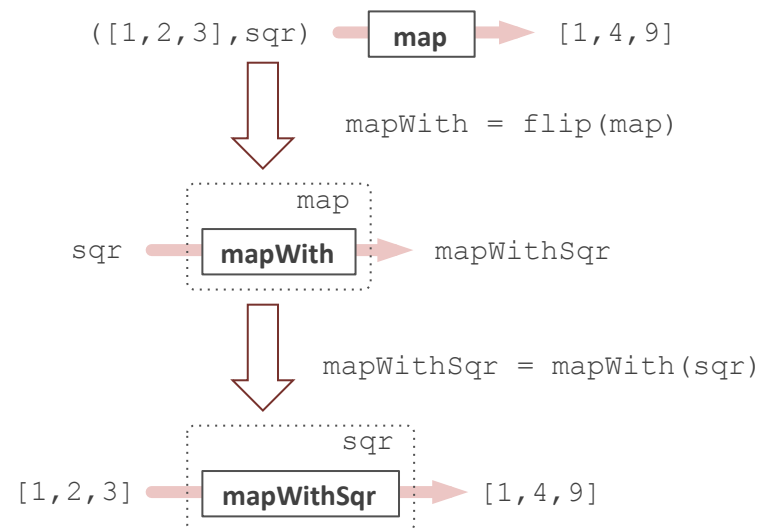
Patrón flip

```
function flip (fn) {  
  return function () {  
    var args = [].slice.call (arguments);  
    return function (second) {  
      return fn.apply(second, args);  
    };  
  };  
}
```

```
var mapWith    = flip([].map);  
var reduceWith = flip([].reduce);  
var filterWith = flip([].filter);  
var everyWith  = flip([].every);  
var someWith   = flip([].some);
```

```
var sqr = function (x) { return x*x; };  
var mapWithSqr = mapWith(sqr);  
mapWithSqr([1,2,3]) // [1,4,9]
```

El patrón flip es equivalente en propósito a doWith con la diferencia que éste se aplica a la versión de los patrones secuenciales de JS que se encuentran en el prototipo Array



Programación Funcional en JavaScript

Patrones de Diseño Funcional

VI. Patrones de Inversión de Control

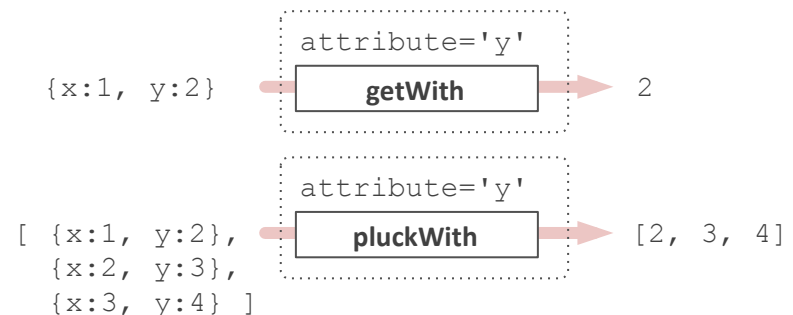
Como describimos en la introducción existen dos modelos arquitectónicos complementarios dentro de la programación funcional. Las arquitecturas centradas en los datos fijan un conjunto de datos y sobre él aplican una cadena de transformaciones. Complementariamente es posible invertir este modelo de control y pasar a un modelo arquitectónico donde una cadena funcional queda fija y son los datos los que la atraviesan.

Patrón `getWith` & `pluckWith`

```
function getWith (attribute) {  
  return function (object) {  
    return object[attribute];  
  };  
};  
  
function pluckWith (attribute) {  
  return mapWith(getWith(attribute));  
}
```

```
var accounts = [  
  { name: 'jvelez', free: 123 },  
  { name: 'eperez', free: 315 },  
  { name: 'jlopez', free: 23 },  
  { name: 'jruiz', free: 65 }  
];  
  
var free = pluckWith('free');  
free(accounts); // [123,315,23,65]
```

El patrón `getWith` define una función que recupera un atributo específico sobre cualquier objeto. `PluckWith` se apoya en `mapWith` para obtener los atributos de un tipo de una colección de objetos



Programación Funcional en JavaScript

Patrones de Diseño Funcional

VI. Patrones de Inversión de Control

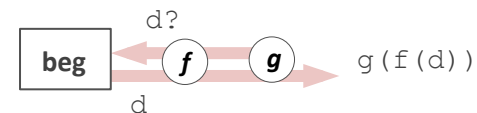
Como describimos en la introducción existen dos modelos arquitectónicos complementarios dentro de la programación funcional. Las arquitecturas centradas en los datos fijan un conjunto de datos y sobre él aplican una cadena de transformaciones. Complementariamente es posible invertir este modelo de control y pasar a un modelo arquitectónico donde una cadena funcional queda fija y son los datos los que la atraviesan.

Patrón `composeWith` & `sequenceWith`

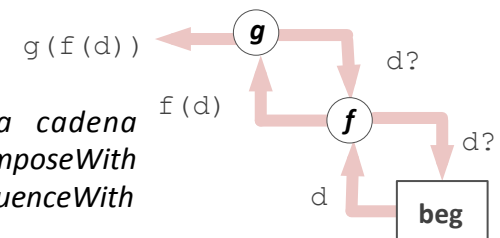
```
function composeWith (fn, gn, beg) {  
  return function () {  
    return fn(gn(beg()));  
  };  
}  
  
function sequenceWith (fns, beg) {  
  return function () {  
    return fns.reduce (function (ac, fn) {  
      return fn(ac);  
    }, beg());  
  };  
}
```

```
var data() = // 3, 4, ...  
var s = sequenceBreakWith ([  
  neg, sqr  
], data);  
[s(), s()]; // [undefined, 17]
```

El patrón `composeWith` solicita un dato a la fuente `beg` y lo hace atravesar por la composición a la inversa



La extensión a cadena funcional de `composeWith` es la función `sequenceWith`



```
var data = function () { return 3; };  
var neg = function neg (x) { return -x; };  
var sqr = function sqr (x) { return x*x; };  
var sqrNeg = composeWith (neg, sqr, data);  
sqrNeg (); // -9
```

Programación Funcional en JavaScript

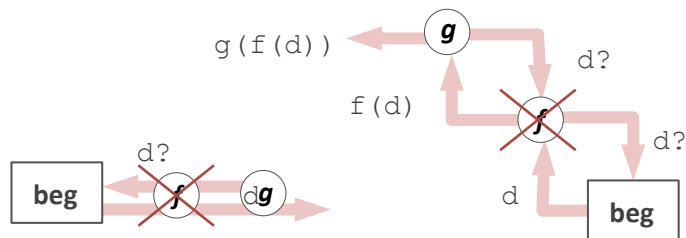
Patrones de Diseño Funcional

VI. Patrones de Inversión de Control

Como describimos en la introducción existen dos modelos arquitectónicos complementarios dentro de la programación funcional. Las arquitecturas centradas en los datos fijan un conjunto de datos y sobre él aplican una cadena de transformaciones. Complementariamente es posible invertir este modelo de control y pasar a un modelo arquitectónico donde una cadena funcional queda fija y son los datos los que la atraviesan.

Patrón {compose & sequence}BreakWith

Este par de patrones extiende los anteriores de manera que si alguna función de la cadena devuelve null desiste de terminar la composición



```
function composeBreakWith (fn, gn, beg) {  
  return function () {  
    var r = gn(beg());  
    return (r !== void 0) ? fn(r) : void 0;  
  };  
}  
  
function sequenceBreakWith (fns, beg) {  
  return function () {  
    return fns.reduce(function(ac, fn) {  
      return (ac !== void 0) ? fn(ac) : void 0;  
    }, beg());  
  };  
}
```

```
var data = function (x) {  
  return function () { return x; };  
};  
function sqr (x) { return x * x; }  
function grt (x) { if (x > 10) return x; }  
composeBreakWith(grt,sqr,data(3))();  
composeBreakWith(grt,sqr,data(4))(); // 16
```

```
var data() = // 3, 4, ...  
var s = sequenceBreakWith ([  
  function sqr (x) { return x * x; },  
  function grt (x) { if (x > 10) return x; },  
  function inc (x) { return x + 1; },  
], data);  
[s(), s()]; // [undefined, 17]
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

VII. Patrones de Programación Ambiental

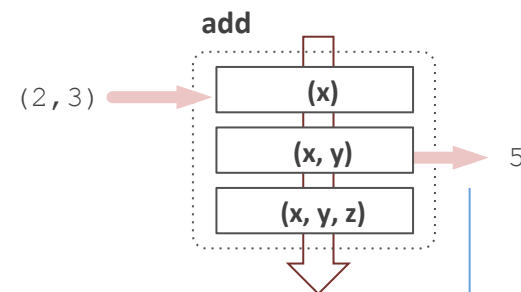
La programación ambiental es un modelo arquitectónico en el que el desarrollo funcional se apoya en un framework que realiza de forma transparente transformaciones funcionales sobre el código proporcionado. En esta sección describimos algunos de los patrones de diseño prototípicos de la programación ambiental.

Patrón overload

```
function overload () {  
  var fns = [].slice.call(arguments);  
  return function () {  
    var args = [].slice.call(arguments);  
    var fn = fns.filter(function (fn) {  
      return fn.length === args.length;  
    })[0];  
    if (fn) return fn.apply(this, args);  
  };  
}
```

```
var add = overload(  
  function (x)      { return x; },  
  function (x, y)   { return x + y; },  
  function (x, y, z){ return x + y + z; }  
);  
add(2,3)  
add(2,3,5)
```

En patrón overload proporciona un entorno de programación que permite definir variantes funcionales diferenciadas por el número de parámetros formales que las definen



Se revisa secuencialmente la aridad de cada variante funcional incluida en el entorno. La primera función que encaja exactamente se evalúa y devuelve el resultado

Programación Funcional en JavaScript

Patrones de Diseño Funcional

VII. Patrones de Programación Ambiental

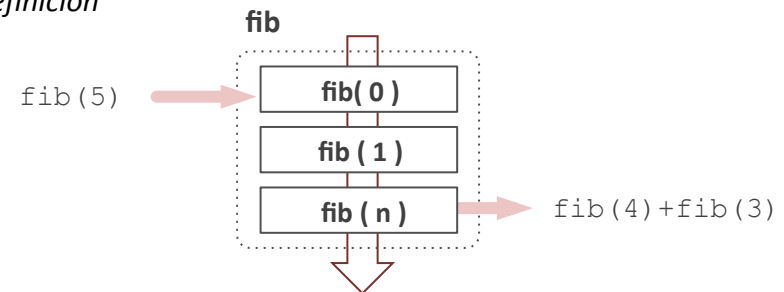
La programación ambiental es un modelo arquitectónico en el que el desarrollo funcional se apoya en un framework que realiza de forma transparente transformaciones funcionales sobre el código proporcionado. En esta sección describimos algunos de los patrones de diseño prototípicos de la programación ambiental.

Patrón cases-when

```
function when (guard, fn) {
  return function () {
    if (guard.apply(this, arguments))
      return fn.apply(this, arguments);
  };
}

function cases () {
  var fns = [].slice.call(arguments);
  return function () {
    var result;
    var args = [].slice.call(arguments);
    fns.forEach(function (fn) {
      if (result !== void 0)
        result = fn.apply(this, args);
    }, this);
    return result;
  };
}
```

En patrón cases-when proporciona un entorno para definir funciones por casos. Cada caso tiene un predicado y una función asociada. Los casos se evalúan en el orden de definición



```
var equals = curry(function (x, y) {
  return x === y;
});
...
var fib = cases(
  when (equals(0), function (n) {return 1; }),
  when (equals(1), function (n) {return 1; }),
  function (n) { return fib (n-1) + fib (n-2); }
);
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

VII. Patrones de Programación Ambiental

La programación ambiental es un modelo arquitectónico en el que el desarrollo funcional se apoya en un framework que realiza de forma transparente transformaciones funcionales sobre el código proporcionado. En esta sección describimos algunos de los patrones de diseño prototípicos de la programación ambiental.

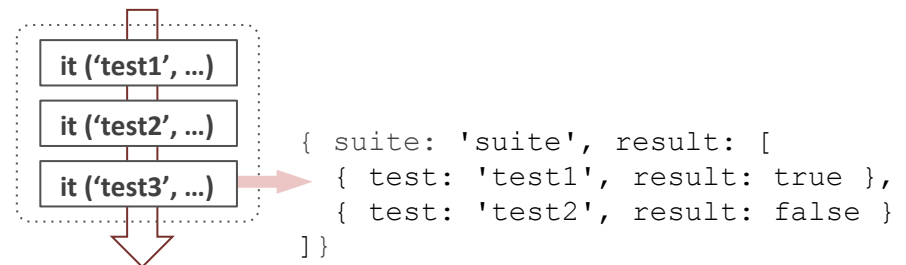
Patrón describe-it

```
function it (test, fn) {
  return function () {
    return {
      test : test,
      result : fn.apply(this, arguments)
    };
  };
}

function describe (suite) {
  return function () {
    var its = [].slice.call(arguments);
    var results = its.map(function (it) {
      return it.apply(this, arguments);
    }, this);
    return {
      suite : suite,
      result : results
    };
  };
}
```

En patrón describe-it proporciona un entorno para definir test unitarios. La función recibe un título y una colección de its en orden superior que ejecuta en secuencia

describe ('suite')



```
describe('Test Suite') {
  it('test 1', function () {
    return 2 + 3 === 5;
  }),
  it('test 2', function () {
    return 2 - 3 === 5;
  })
};
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

VIII. Patrones de Optimización

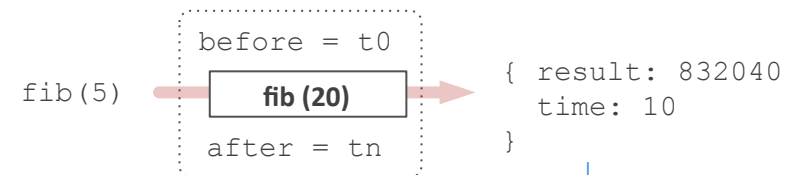
La optimización de código tanto en tiempo como en memoria es una de las preocupaciones esenciales de la programación cuando se abordan problemas que requieren un alto grado de rendimiento. Los patrones de optimización presentan estrategias esenciales acerca de cómo realizar transformaciones funcionales abstractas dirigidas a tal fin.

Patrón time

```
function time (fn) {  
  return function () {  
    var before = Date.now();  
    var result = fn.apply(this, arguments);  
    var after = Date.now();  
    return {  
      value : result,  
      time : (after - before)  
    };  
  };  
}
```

```
var fib = function (n) {  
  if (n < 2) return n;  
  else return fib(n-1) + fib(n-2);  
};  
var tfib = time(fib);  
tfib(20);
```

El patrón time permite medir el tiempo que tarda una función pasada en orden superior en ejecutar



Se pide una marca de tiempo al sistema antes y después de ejecutar la función y se devuelve un objeto con el resultado y la diferencia de tiempos en ms

Programación Funcional en JavaScript

Patrones de Diseño Funcional

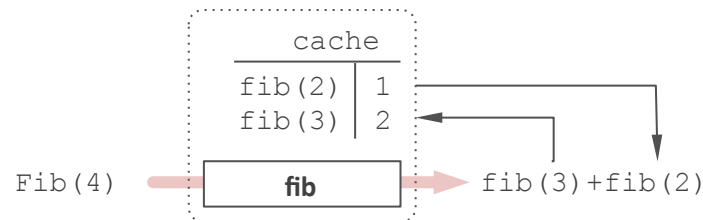
VIII. Patrones de Optimización

La optimización de código tanto en tiempo como en memoria es una de las preocupaciones esenciales de la programación cuando se abordan problemas que requieren un alto grado de rendimiento. Los patrones de optimización presentan estrategias esenciales acerca de cómo realizar transformaciones funcionales abstractas dirigidas a tal fin.

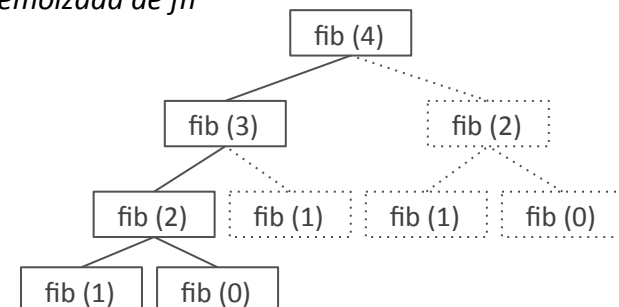
Patrón memoize

```
function memoize (fn) {  
  var cache = {};  
  var mfn = function () {  
    var args = [].slice.call (arguments);  
    var key = JSON.stringify(args);  
    return key in cache ?  
      cache[key] :  
      cache[key] = fn.apply(this, args);  
  };  
  return mfn;  
}
```

```
var fib = function (n) {  
  if (n < 2) return n;  
  else return fib(n-1) + fib(n-2);  
};  
var fib = memoize(fib);  
tfib(43);
```



Memoize retiene una cache interna sobre evaluaciones anteriormente calculadas dentro del proceso recursivo. Ahora en el árbol de activación todas las llamadas en líneas discontinuas no se realizan. Nótese que la memoización el código se hace sobre la misma variable fib ya que es la única forma de garantizar que en memoize se recurre sobre la versión memoizada de fn



Programación Funcional en JavaScript

Patrones de Diseño Funcional

VIII. Patrones de Optimización

La optimización de código tanto en tiempo como en memoria es una de las preocupaciones esenciales de la programación cuando se abordan problemas que requieren un alto grado de rendimiento. Los patrones de optimización presentan estrategias esenciales acerca de cómo realizar transformaciones funcionales abstractas dirigidas a tal fin.

Patrón trampoline

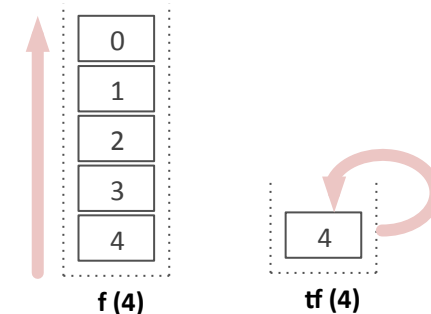
```
var f = function (n) {  
  if (n===0) return 1;  
  return n*f(n-1);  
};  
var tf = function (n, ac) {  
  if (n === 0) return ac;  
  return tf(n-1, n*ac);  
};
```

El problema

La función *f* tiene recursividad no final puesto que en el caso recursivo, después de invocar a *f*, es necesario multiplicar por *n*. Esto tiene impacto en memoria ya que el compilador tiene que recordar cada valor de *n* para cada una de las llamadas recursivas a *f*. Los lenguajes con soporte a la recursividad utilizan una pila para gestionar esto. En el ejemplo vemos como *f*(4) consume 4*k espacios de memoria donde *k* es el tamaño del registro de activación que necesita el compilador para gestionar cada invocación

Tail Recursion. Una solución inútil

En la versión *tf*, se rediseña la función *f* de manera que el caso recursivo no tiene operaciones de combinación tras su invocación. A esto se le llama diseño en recursividad final o tail recursión y para ello se emplean técnicas de inmersión por acumulador. Algunos lenguajes saben sacar partido de este tipo de diseños haciendo que las llamadas no se gestionen por pila sino en una zona de memoria estática. Aunque esto sobrescribe los valores en cada invocación recursiva no tiene, por diseño en este caso, ningún efecto neto y sí un considerable ahorro de espacio en memoria. Lamentablemente este no es el caso de JavaScript



Programación Funcional en JavaScript

Patrones de Diseño Funcional

VIII. Patrones de Optimización

La optimización de código tanto en tiempo como en memoria es una de las preocupaciones esenciales de la programación cuando se abordan problemas que requieren un alto grado de rendimiento. Los patrones de optimización presentan estrategias esenciales acerca de cómo realizar transformaciones funcionales abstractas dirigidas a tal fin.

Patrón trampoline

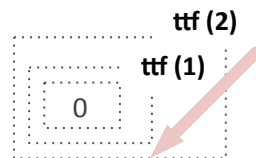
```
var ttf = function f(n, ac) {  
  if (n === 0) return ac;  
  return function () {  
    return f(n-1, n*ac);  
  };  
};
```

Diseño por Thunks

Partiendo de la versión de en recursividad final es posible crear un nuevo diseño en el que el caso recursivo no se invoca directamente sino que se envuelve en una función. Este tipo de envolturas recibe el nombre de Thunk y se utiliza como función vehicular para comunicar datos a un framework. El efecto neto en este caso es que la resolución se rompe a cada paso introduciendo una fase de evaluación

El framework trampoline

Con este diseño la función trampoline juega el papel del framework encargado de desenvolver dentro de una iteración cada fase de envoltura con thunk y acumularla en una variable de resultados. De esta forma se consigue desarrollar la recursividad sin hacer uso de pila



```
function trampoline (fn) {  
  return function () {  
    var result = fn.apply(this, arguments);  
    while (result instanceof Function)  
      result = result();  
    return result;  
  };  
}  
  
var ttftf = trampoline(ttf);  
ttftf(5); // 120
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

¹ Programación Asíncrona en NodeJS. En Slideshare@ jvelez77

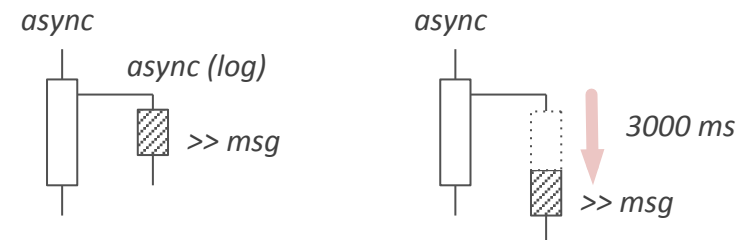
IX. Patrones de Programación Asíncrona

La programación asíncrona es un modelo de programación en el que algunas funciones son no bloqueantes y devuelven el control al programa llamante de manera instantánea. Dado que estas funciones se independizan del flujo de ejecución central, es necesario proporcionarles de alguna forma la lógica de continuación que prescribe cómo procesar la respuesta cuando la función acabe. Dado que ya dedicamos un documento entero a la programación asíncrona¹ nos centraremos aquí en dar patrones para hacer invocaciones asíncronas.

Patrón delay & async

```
function delay (fn, ms) {  
  return function () {  
    var args = [].slice.call(arguments);  
    setTimeout (function () {  
      fn.apply(this, args);  
    }.bind(this), ms);  
  };  
}  
  
function async (fn) {  
  return delay(fn, 0);  
}
```

Los patrones delay y async transforman una función en no bloqueante instantáneamente y tras un retardo determinado respectivamente



```
var log = function(msg){ console.log (msg); };  
var dlog = delay(log, 3000);  
var alog = async(log);  
  
dlog('NodeJS Madrid Mola!!!');  
alog('NodeJS Madrid Mola!!!');
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

¹ Programación Asíncrona en NodeJS. En Slideshare@ jvelez77

IX. Patrones de Programación Asíncrona

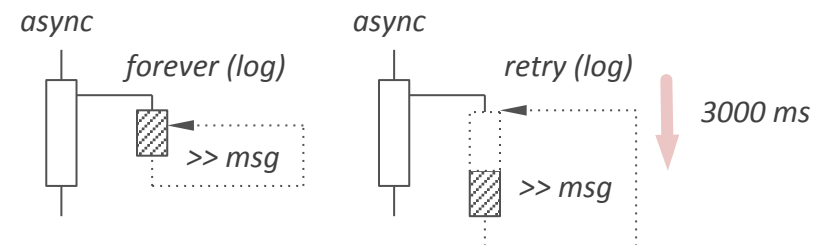
La programación asíncrona es un modelo de programación en el que algunas funciones son no bloqueantes y devuelven el control al programa llamante de manera instantánea. Dado que estas funciones se independizan del flujo de ejecución central, es necesario proporcionarles de alguna forma la lógica de continuación que prescribe cómo procesar la respuesta cuando la función acabe. Dado que ya dedicamos un documento entero a la programación asíncrona¹ nos centraremos aquí en dar patrones para hacer invocaciones asíncronas.

Patrón forever & retry

```
function forever (fn, ms) {
  return function () {
    var args = [].slice.call(arguments);
    setInterval (function () {
      fn.apply(this, args);
    }.bind(this), ms);
  };
}

function retry (fn, n, ms) {
  var idx = 0;
  return function aux () {
    var r = fn.apply(this, arguments);
    if (idx < n && !r) {
      r = delay(fn, ms)
        .apply(this, arguments); idx++;
    }
    return r;
  };
}
```

El patrón forever itera eternamente una función a intervalos de tiempo marcados mientras que retry reintenta la invocación de una función un número de veces mientras se obtenga un resultado no definido



```
var log = function(msg){ console.log (msg); };
var rlog = retry (log, 3, 1000);
var flog = forever (log, 3000);

rlog ('NodeJS Madrid Mola!!!');
flog ('NodeJS Madrid Mola!!!');
```

Javier Vélez Reyes
@javiervelezreyes
Javier.veler.reyes@gmail.com

5 *Arquitecturas de Programación Funcional*

- Arquitecturas Map-Reduce
- Arquitecturas Reactivas
- Arquitecturas Asíncronas

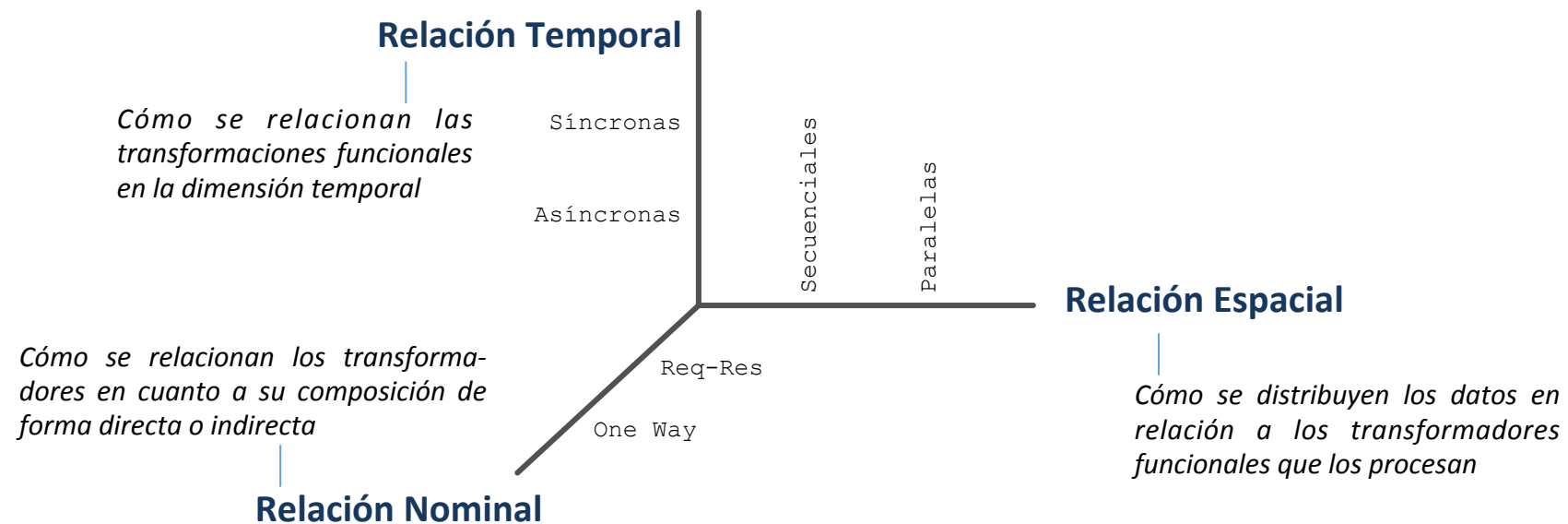
Programación Funcional en Node JS
Arquitecturas de Programación Funcional

Programación Funcional en Node JS

Arquitecturas de Programación Funcional

I. Introducción

En los capítulos anteriores hemos presentado los mecanismos, técnicas y patrones de diseño que se emplean en el marco de la programación funcional. No obstante, sobre esta base, es posible definir estilos arquitectónicos propios por aplicación de una serie de restricciones en relación a los objetivos y principios de diseño que se persiguen para un determinado tipo de problemas. Sin ánimo de completitud, a lo largo de este capítulo presentaremos una colección de arquitecturas funcionales que mantienen actual vigencia en problemas canónicos en la práctica empresarial. El siguiente marco clasifica la familia de arquitecturas existentes.



Programación Funcional en Node JS

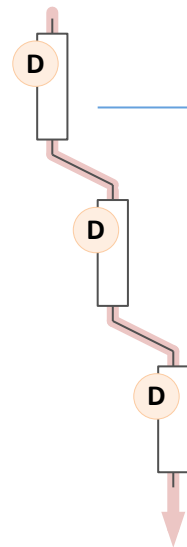
Arquitecturas de Programación Funcional

I. Introducción

A. Relación Espacial

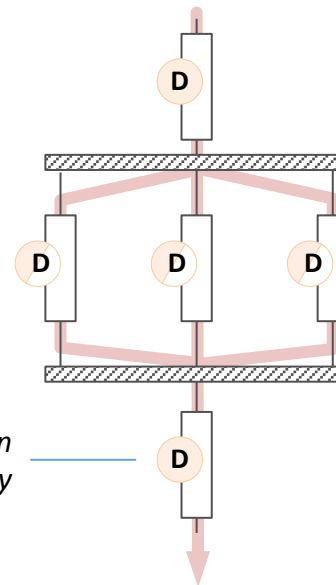
La relación espacial discrimina cómo se distribuyen los datos entre la colección de transformadores funcionales que los procesan. En las arquitecturas secuenciales, las funciones se disponen compositivamente de manera que los datos atraviesan una cadena de transformaciones en cascada. Las arquitecturas paralelas parten los datos en conjuntos homogéneos y operan sobre ellos de manera simultánea.

Arquitecturas Secuenciales



En las arquitecturas secuenciales los datos atraviesan en cascada una cadena compositiva

Arquitecturas Paralelas



Las arquitecturas paralela parten los datos de forma homogénea y los procesan de forma simultanea

Programación Funcional en Node JS

Arquitecturas de Programación Funcional

¹ Programación Asíncrona en NodeJS. En Slideshare@ jvelez77

I. Introducción

B. Relación Temporal

La relación temporal clasifica las soluciones en función del carácter bloqueante o no bloqueante de las transformaciones funcionales. Las arquitecturas síncronas encadenan operaciones bloqueantes mientras que las asíncronas invocan operaciones no bloqueantes y continúan con el flujo de procesamiento normal. Esto complica considerablemente el modelo de programación requiriendo introducir continuaciones o mónadas de continuidad¹.

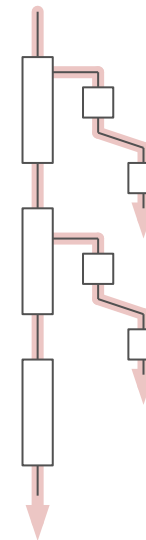
Arquitecturas Síncronas



En las arquitecturas síncronas todas las transformaciones funcionales son bloqueantes lo que implica que una operación no comienza hasta que termina la anterior

Las arquitecturas asíncronas entrelazan operaciones bloqueantes y no bloqueantes de manera que aparecen varios flujos de transformación

Arquitecturas Asíncronas



Programación Funcional en Node JS

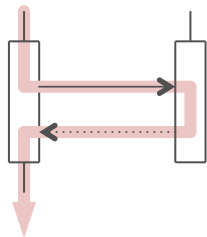
Arquitecturas de Programación Funcional

I. Introducción

C. Relación Nominal

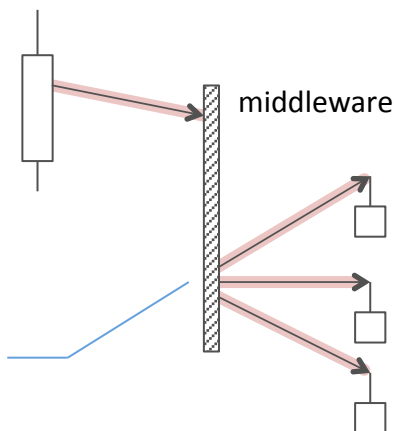
La relación nominal se centra en describir las arquitecturas atendiendo al tipo de composición que mantienen las abstracciones funcionales entre sí. Se distingue entre arquitecturas acopladas nominalmente, donde la función llamante conoce la identidad del destinatario de la llamada y las arquitecturas desacopladas donde la identidad es desconocida y el número de los receptores que atienden una llamada puede ser múltiple.

Arquitecturas Solicitud Respuesta



En las arquitecturas Solicitud Respuesta el llamante conoce la identidad del llamado y cada llamada es atendida exactamente por un único receptor

Arquitecturas One Way



En las arquitecturas One Way, el emisor no conoce la identidad de sus destinatarios ni tampoco la cantidad de aquéllos que atenderán su solicitud

Programación Funcional en Node JS

Arquitecturas de Programación Funcional

II. Arquitecturas Map-Reduce

Consideremos un dataset con información sobre los usuarios de twitter. Cada registro incluye la edad y el número de contactos. Se pretende hacer una consulta map-reduce sobre estos datos para obtener un histograma de frecuencias agrupando contactos por rango de edad.

Fase I. Split

Split recibe el dataset y el número de segmentos en los que se fragmentarán los datos. La política de fragmentación en este ejemplo no es importante. Sólo se requiere un tamaño homogéneo entre los mismos

```
function split (dataset, n) {  
  var results = [];  
  dataset.forEach(function (item, index) {  
    var batch = index % n;  
    if (results[batch]) results[batch].push(item);  
    else results[batch] = [item];  
  });  
  return results;  
}
```

```
function map (batch) {  
  return batch.map(function (item) {  
    return {  
      age      : item.age,  
      contacts : item.contacts,  
      count    : 1  
    };  
  });  
}
```

Fase II. Map

La operación map recibe un segmento de datos y lo enriquece incluyendo un contador a 1. Este dato será transformado en fases posteriores del algoritmo y se empleará en la etapa de consolidación final

Programación Funcional en Node JS

Arquitecturas de Programación Funcional

II. Arquitecturas Map-Reduce

Consideremos un dataset con información sobre los usuarios de twitter. Cada registro incluye la edad y el número de contactos. Se pretende hacer una consulta map-reduce sobre estos datos para obtener un histograma de frecuencias agrupando contactos por rango de edad.

Fase III. Shuffle

Esta operación recibe toda la base de datos y crea unos nuevos segmentos para cada rango de edad

En el ejemplo se ha decidido, por simplicidad, dividir la edad por 10 para identificar el rango de edad al que pertenece un usuario pero otros criterios más elaborados son posibles

Nótese que el propósito de esta operación es redistribuir los datos de manera que cada nuevo segmento incluya todos los datos con una clave común (en nuestro ejemplo rango de edad) pero aún no se aplican estrategias de reducción que son responsabilidad de la siguiente fase

```
function shuffle (dataset) {
  return dataset.reduce(function (ac, item) {
    var range = Math.floor(item.age / 10);
    if (!ac[range]) {
      ac[range] = [];
    }
    ac[range].push({
      range    : range,
      age      : item.age,
      contacts : item.contacts,
      count    : item.count
    });
    return ac;
  }, []);
}
```

Programación Funcional en Node JS

Arquitecturas de Programación Funcional

II. Arquitecturas Map-Reduce

Consideremos un dataset con información sobre los usuarios de twitter. Cada registro incluye la edad y el número de contactos. Se pretende hacer una consulta map-reduce sobre estos datos para obtener un histograma de frecuencias agrupando contactos por rango de edad.

Fase VI. Shuffle

La operación reduce recibe un segmento asociado a un rango de edad y acumula el número de contactos y el número de usuarios dentro de ese rango

```
function reduce (batch) {  
  return batch.reduce(function (ac, item) {  
    ac.range      = item.range;  
    ac.contacts += item.contacts;  
    ac.count      += item.count;  
    return ac;  
  }, { contacts: 0, count: 0 });  
}
```

```
function join (dataset) {  
  return dataset.reduce(function (ac, batch) {  
    ac.push({  
      range      : batch.range,  
      contacts    : batch.contacts,  
      count      : batch.count,  
      freq       : batch.contacts / batch.count  
    });  
    return ac;  
  }, []);  
}
```

Fase V. Join

Finalmente, la operación Join acumula todos los segmentos que agrupan los rangos de edad y para cada uno de ellos calcula sus frecuencias relativas

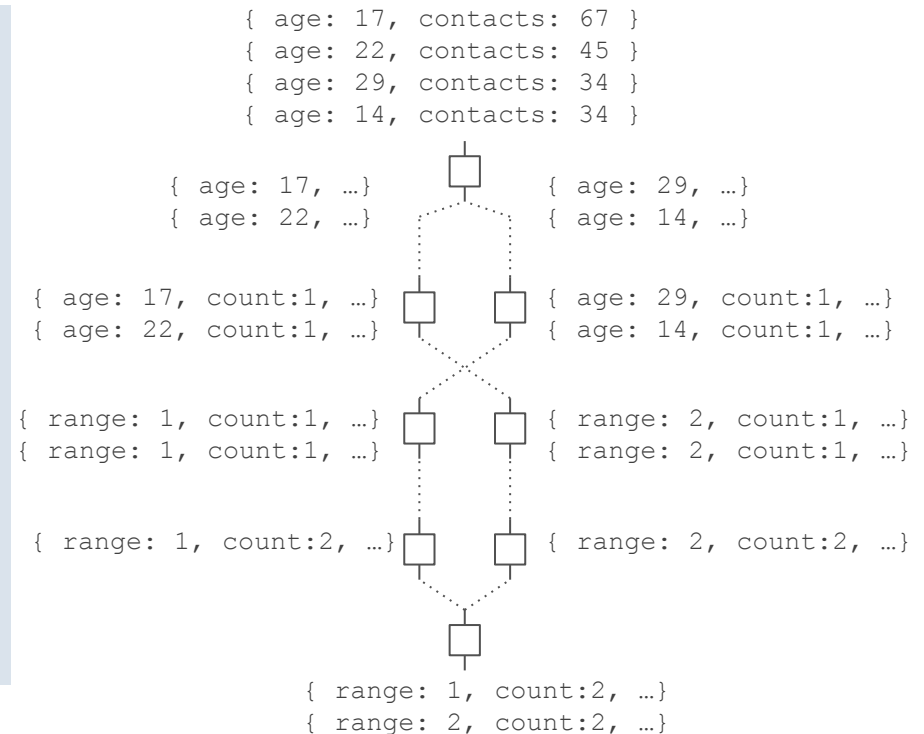
Programación Funcional en Node JS

Arquitecturas de Programación Funcional

II. Arquitecturas Map-Reduce

Consideremos un dataset con información sobre los usuarios de twitter. Cada registro incluye la edad y el número de contactos. Se pretende hacer una consulta map-reduce sobre estos datos para obtener un histograma de frecuencias agrupando contactos por rango de edad.

```
join (split (dataset, 3)
  .map (function (batch) {
    return map(batch);
  })
  .reduce (function (ac, batch, index, ds) {
    return (index < ds.length-1) ?
      ac.concat (batch) :
      [ac.concat (batch)];
  }, [])
  .reduce (function (ac, ds) {
    ac = shuffle (ds);
    return ac;
  }, [])
  .reduce (function (ac, batch) {
    ac.push (reduce (batch));
    return ac;
  }, [])
);
```

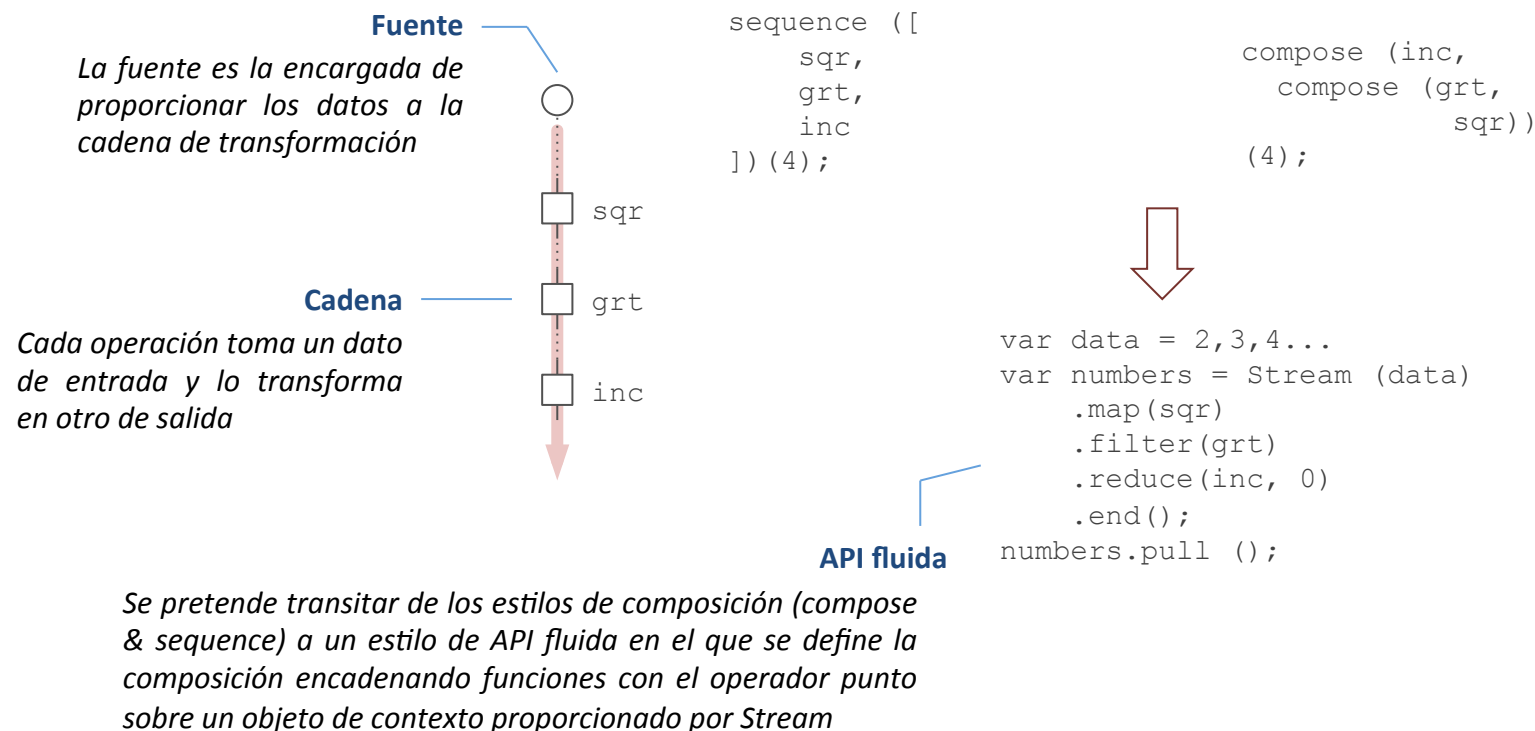


Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.



Programación Funcional en JavaScript

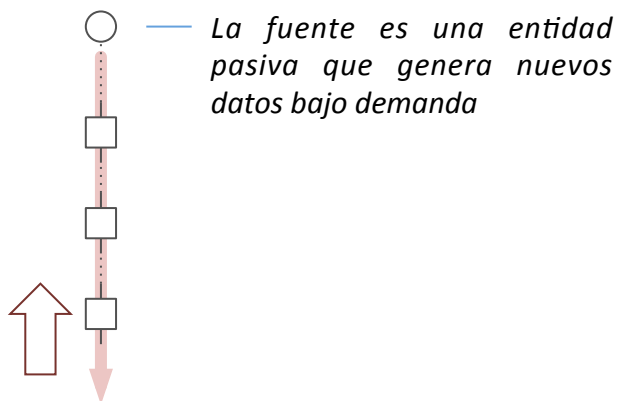
Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

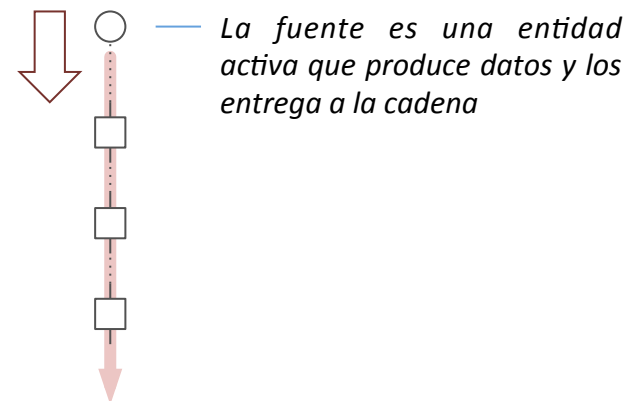
I. Modelo Pull

En el modelo pull, el último nodo solicita nuevos datos y la solicitud recorre ascendentemente la cadena hasta llegar a la fuente



I. Modelo Push

En el modelo push, a medida que se disponen de nuevos datos, la fuente los empuja por la cadena de transformación



Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

pull & push

```
function pull(fn, base) {  
  var idx = base || 0;  
  return function () {  
    return fn.call(this, idx++);  
  };  
}
```

```
var inc = function (x) { return x + 1; };  
var dbl = function (x) { return 2 * x; };  
var s1 = pull(inc);  
var s2 = pull(dbl);  
[s1(), s1(), s1()] // [0, 1, 2]  
[s2(), s2(), s2()] // [0, 2, 4]
```

El patrón pull transforma un función en un iterador con memoria que devuelve en cada invocación el siguiente elemento de la serie

Las fuentes activas corresponden generalmente con escuchadores de eventos. Para emular datos asíncronos emitidos a intervalos regulares usamos el patrón push

```
function push (fn, ms) {  
  return function (cb) {  
    var source = pull(fn);  
    setInterval (function () {  
      cb(source());  
    }, ms || 1000);  
  };  
}
```

```
var log = console.log  
var delay = 3000;  
push(inc, log, delay); >> 0, 1, 2, ...  
push(dbl, log, delay); >> 0, 2, 4, ...
```

Programación Funcional en JavaScript

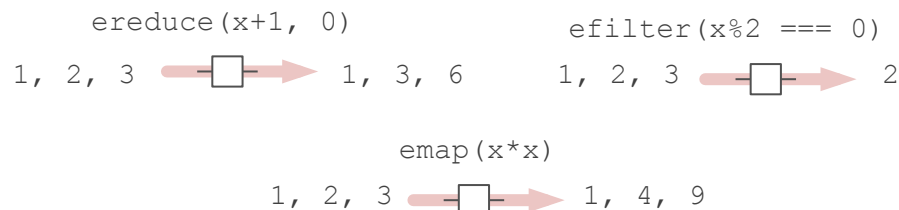
Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

emap, efilter & ereducer

Para inyectar las transformaciones sobre la cadena de composición, se utilizan los patrones de secuenciamiento pero en versión de un solo dato en vez de una colección



```
function ereducer (fn, b) {  
  var ac = b;  
  return function () {  
    var args = [].slice.call(arguments);  
    ac = fn.apply(this, [ac].concat(args));  
    return ac;  
  };  
}
```

```
function efilter (fn) {  
  return function () {  
    var out = fn.apply(this, arguments);  
    if (out) return arguments[0];  
  };  
}
```

```
function emap (fn) {  
  return function () {  
    return fn.apply(this, arguments);  
  };  
}
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

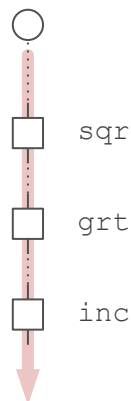
III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

fluent

Dado que cada operador (clean, words, count) devuelve un dato transformado, no podemos encadenar sus llamadas tal cual.

```
var data = 2,3,4...
var numbers = Stream (data)
    .map(sqr)
    .filter(grt)
    .reduce(inc, 0)
    .end();
numbers ();
```



```
function fluent (hn) {
  var cb = hn || function () {};
  return function (fn) {
    return function () {
      cb (fn.apply(this, arguments));
      return this;
    };
  };
}
```

Necesitamos transformar los operadores a través del patrón fluid que genera una función para ejecutar el operador, entregar el resultado a un manejador y devolver una referencia al objeto contexto devuelto por Stream para poder seguir encadenando operadores

Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

Stream Pull

La función define aplica fluent para registrar cada función manejadora que introduce el cliente en map filter y reduce en fns

La función next aplica sequenceBreakWith sobre cada función registrada en fns para obtener una cadena de composición

Se devuelve el objeto de contexto con los métodos map, filter y reduce decorados con define. El método end transforma el contexto en un objeto con un método pull para explotar la cadena

```
function pullStream (source) {
  var fns = [];
  var define = fluent(function (fn) {
    fns.push(fn);
  });
  var next = function (fns, beg) {
    var result = sequenceBreakWith(fns, beg)();
    return result ? result : next(fns, beg);
  };
  return {
    map    : define(emap),
    filter: define(efilter),
    reduce: define(ereduce),
    end    : function () {
      return {
        pull: function () {
          return next(fns, source);
        }
      };
    };
  };
}
```

Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

Stream Pull

Se define una fuente de datos de números naturales y se aplica la cadena de transformación [sqr, even, add]

```
var source = pull (function (x) { return x + 1; }, 0);
var stream = Stream(source)
    .map      (function (e)      { return e * e;      })
    .filter   (function (e)      { return e % 2 === 0})
    .reduce   (function (a, e) { return a + e;      }, 0)
    .end();
stream.pull(); // 0, 4, 20, 50, ...
```

```
var source = function s() { return 'NodeJS Mola!' };
var stream = Stream (source)
    .map (function (s) { return s.trim ();      })
    .map (function (s) { return s.split (' '); })
    .map (function (s) { return s.length;      })
    .end ();
source.pull(); // 2
```

Se define una fuente de datos constante con la cadena 'NodeJS Mola' y se aplica la cadena [clean, split, count] para contar las palabras

Programación Funcional en JavaScript

Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

Stream Push

La colección lns acumula los escuchadores que se registran al stream para ser notificados de cada nuevo resultado procesado

La función end aplica sequenceBreak una vez para obtener la cadena de composición y después arranca la fuente con un manejador que obtiene el siguiente resultado y si es definido lo publica a cada escuchador

Se devuelve un objeto con un método listen que permite registrar nuevos manejadores asociados a escuchadores interesados

```
function pushStream (source) {
  var fns = [];
  var lns = [];
  var define = fluent(function (fn) {...});
  return {
    map    : define(emap),
    filter: define(efilter),
    reduce: define(ereduce),
    end: function () {
      var sequence = sequenceBreak(fns);
      source(function (data) {
        var result = sequence(data);
        lns.forEach(function (ln) {
          if (result) ln(result);
        });
      });
    },
    listen: function (ln) {
      lns.push(ln);
    }
  };
}
```

Programación Funcional en JavaScript

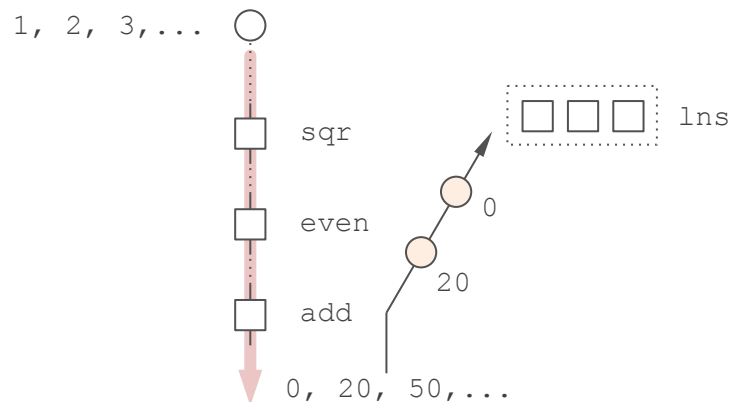
Patrones de Diseño Funcional

III. Arquitecturas Reactivas

Las arquitecturas de programación reactiva establecen cadenas de composición que son atravesadas por streams de datos. Cada operación dentro de la cadena se encarga de recibir un dato cada vez y transformarlo de acuerdo a determinada lógica. En esta sección describiremos cómo funciona un framework de programación reactiva internamente y discutiremos los patrones que utiliza.

Stream Push

La fuente es ahora un emisor de datos activo que funciona según el patrón push. Cuando hay datos nuevos, ésta los empuja por la cadena de composición y los publica encada uno de los escuchadores registrados



```
var source = push(function (x) { return x + 1; });
var stream = Stream(source)
  .map      (function (e)      { return e * e;      })
  .filter   (function (e)      { return e % 2 === 0})
  .reduce   (function (a, e)   { return a + e;      }, 0)
  .end();

stream.listen(console.log);
```

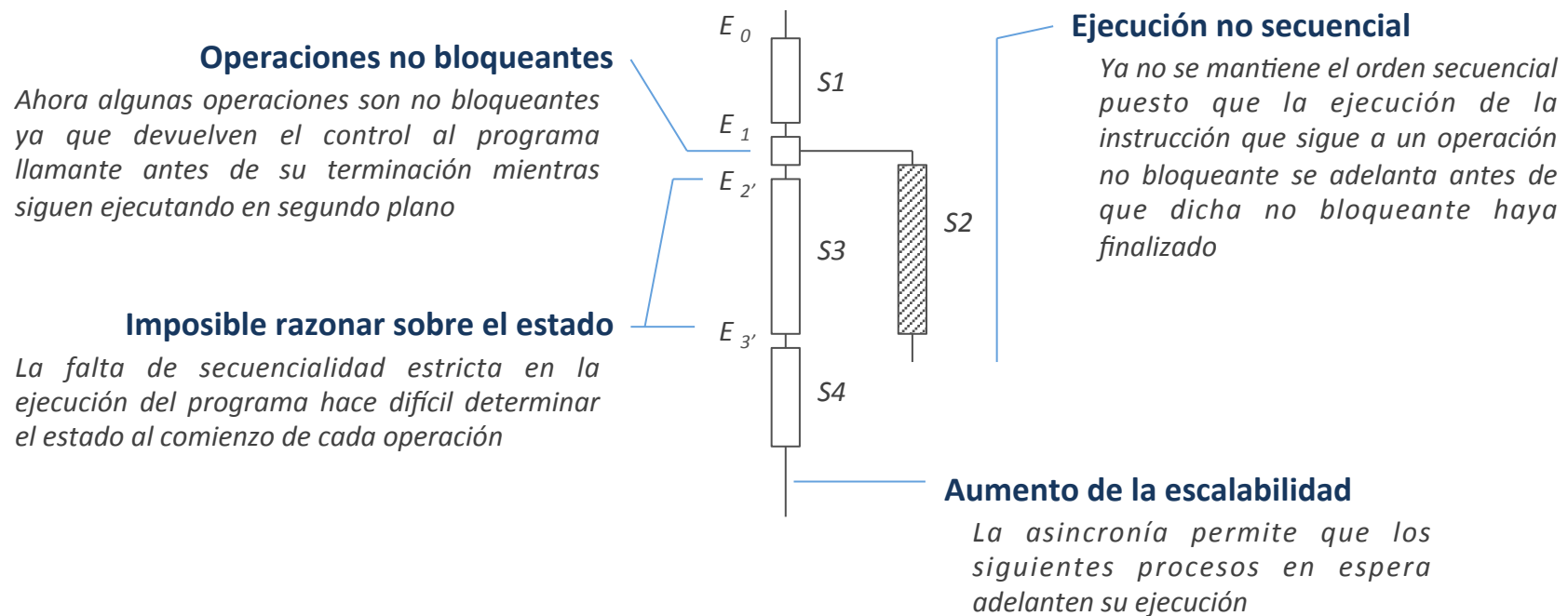
Cada escuchador interesado en ser notificado de nuevos datos emitidos por el stream debe registrar una función manejadora a través del método push

Programación Funcional en Node JS

Arquitecturas de Programación Funcional

IV. Arquitecturas Asíncronas

Las arquitecturas asíncronas establecen la posibilidad de hacer que algunas operaciones devuelvan el control al programa llamante antes de que hayan terminado mientras siguen operando en segundo plano. Esto agiliza el proceso de ejecución y en general permite aumentar la escalabilidad pero complica el razonamiento sobre el programa.



Programación Funcional en Node JS

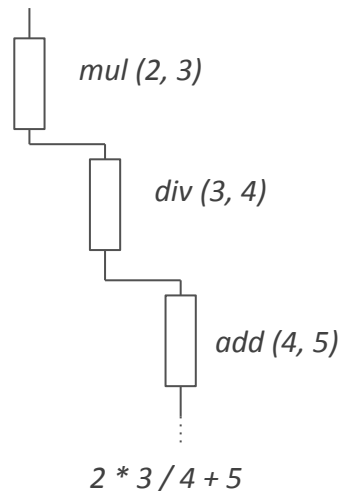
Arquitecturas de Programación Funcional

IV. Arquitecturas Asíncronas

Las arquitecturas asíncronas establecen la posibilidad de hacer que algunas operaciones devuelvan el control al programa llamante antes de que hayan terminado mientras siguen operando en segundo plano. Esto agiliza el proceso de ejecución y en general permite aumentar la escalabilidad pero complica el razonamiento sobre el programa.

Modelo de Paso de Continuaciones

La función de continuación permite indicar a la operación bloqueante como debe proceder después de finalizada la operación



```
function div(a, b, callback) {  
  if (b === 0) callback (Error (...))  
  else {  
    var result = a / b;  
    callback (null, result);  
  }  
}
```

```
div (8, 2, function (error, data) {  
  if (error) console.error (error);  
  else console.log (data);  
});
```

La manera de proceder dentro de este modelo para establecer flujos de ejecución secuenciales exige ir encadenando cada función subsiguiente como continuación de la anterior donde se procesarán los resultados. Esto conduce a una diagonalización del código incomoda de manejar

Programación Funcional en Node JS

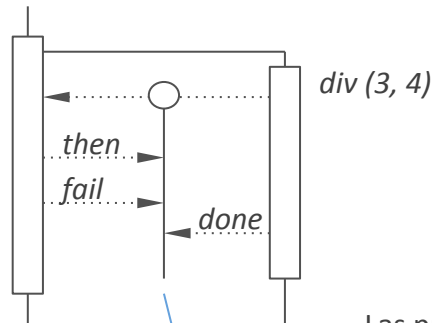
Arquitecturas de Programación Funcional

IV. Arquitecturas Asíncronas

Las arquitecturas asíncronas establecen la posibilidad de hacer que algunas operaciones devuelvan el control al programa llamante antes de que hayan terminado mientras siguen operando en segundo plano. Esto agiliza el proceso de ejecución y en general permite aumentar la escalabilidad pero complica el razonamiento sobre el programa.

Modelo de Promesas

La promesa es una monada de continuidad que incuye los inyectores de continuidad `then` y `fail` para incluir los manejadores de éxito y fallo



```
var promise = div (a, b);
    .then (function (data) {
        console.log (data)
    })
    .fail (function (error) {
        console.error (error)
    });
```

```
function div (x, y) {
    var result = ...
    return <<toPromise (result)>>;
}
```

Las promesas se convierten en un protocolo de comunicación entre el flujo llamador y llamante. El llamante invoca la operación asíncrona. Ésta le devuelve un objeto promesa. El llamante inyecta los manejadores de éxito y fracaso en la promesa. Y tras finalizar, la operación asíncrona busca en la promesa la función de continuación

Programación Asíncrona en Node JS

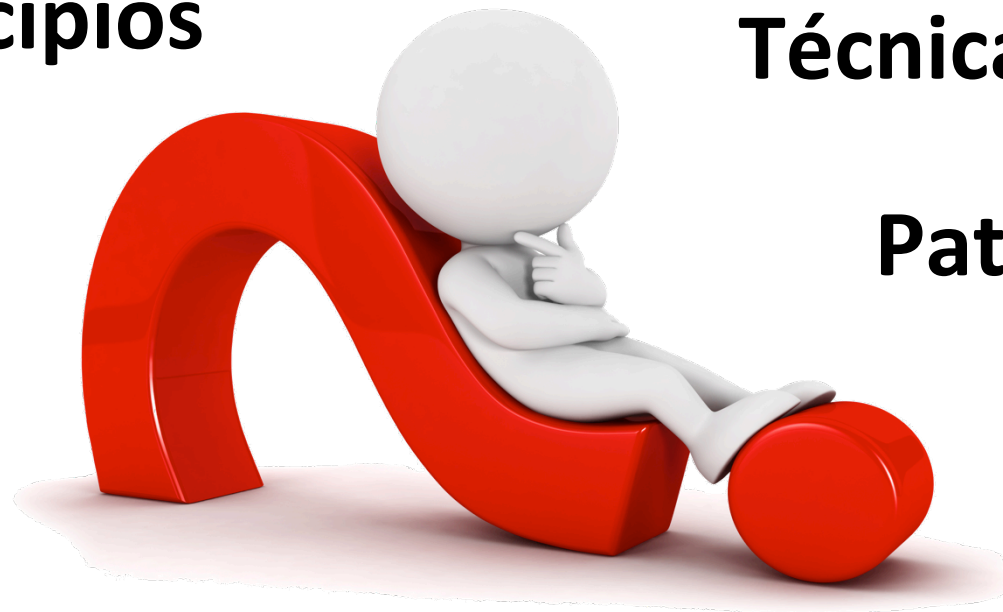
Preguntas

Mecanismos

Principios

Técnicas

Patrones



Javier Vélez Reyes

@javiervelezreye

Javier.velez.reyes@gmail.com

Programación Funcional en NodeJS

Técnicas, Patrones y Arquitecturas Funcionales

Javier Vélez Reyes

@javiervelezreye
Javier.velez.reyes@gmail.com

Octubre 2014

