Programación Funcional 2

FIUBA - Técnicas de Diseño

Esta función, ¿es funcional?

```
function sumNumbers (nums: number[]) {
    let result = 0;

    for (i = 0; i < nums.length; ++i) {
        result = result + nums[i];
    }

    return result;
}
```

Esta función, ¿es funcional?

Cumple:

- ¿Es predecible?
- ¿Son inmutables sus argumentos?
- ¿Tiene entradas implícitas?
- ¿Tiene efectos colaterales?

No cumple:

- ¿Son inmutables sus variables internas?

Esta función, ¿es funcional?

Respuesta:

- Vista por su implementador: No es funcional
- Vista por su consumidor: Es funcional
    - Suponiendo que hayamos analizado correctamente

Esta función, ¿es funcional?

- Sin valor de retorno

```
void f(int arg)
```

- Sin argumento

```
int g()
```

Estructuras de Datos

Estructuras de Datos Persistentes

"En lugar de actualizar valores, se crean nuevos valores"

¿Qué sucede con los valores viejos?

Estructuras de Datos Persistentes

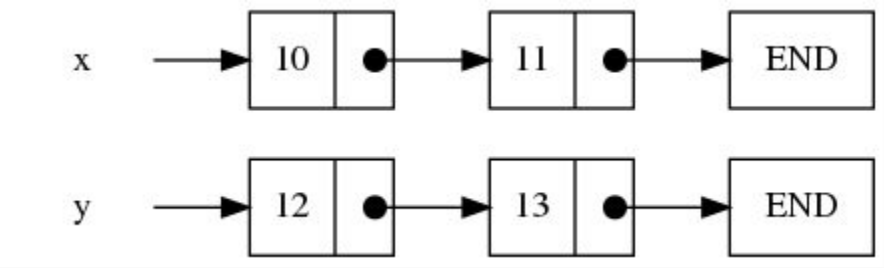
"En lugar de actualizar valores, se crean nuevos valores"

¿Qué sucede con los valores viejos?

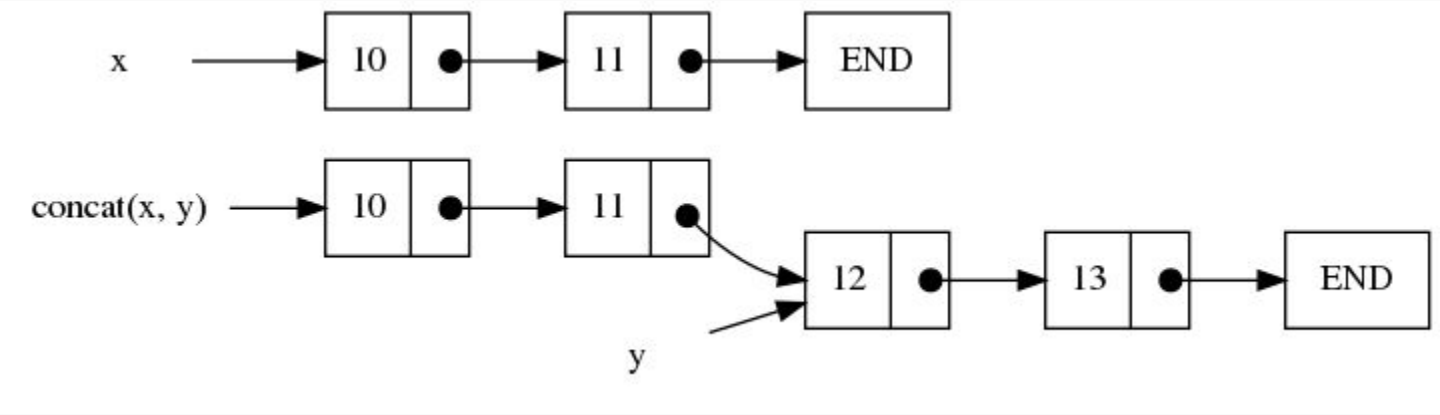
- Seguimos usándolos
- Garbage collector

¿En qué casos puede servir seguir usando los valores?

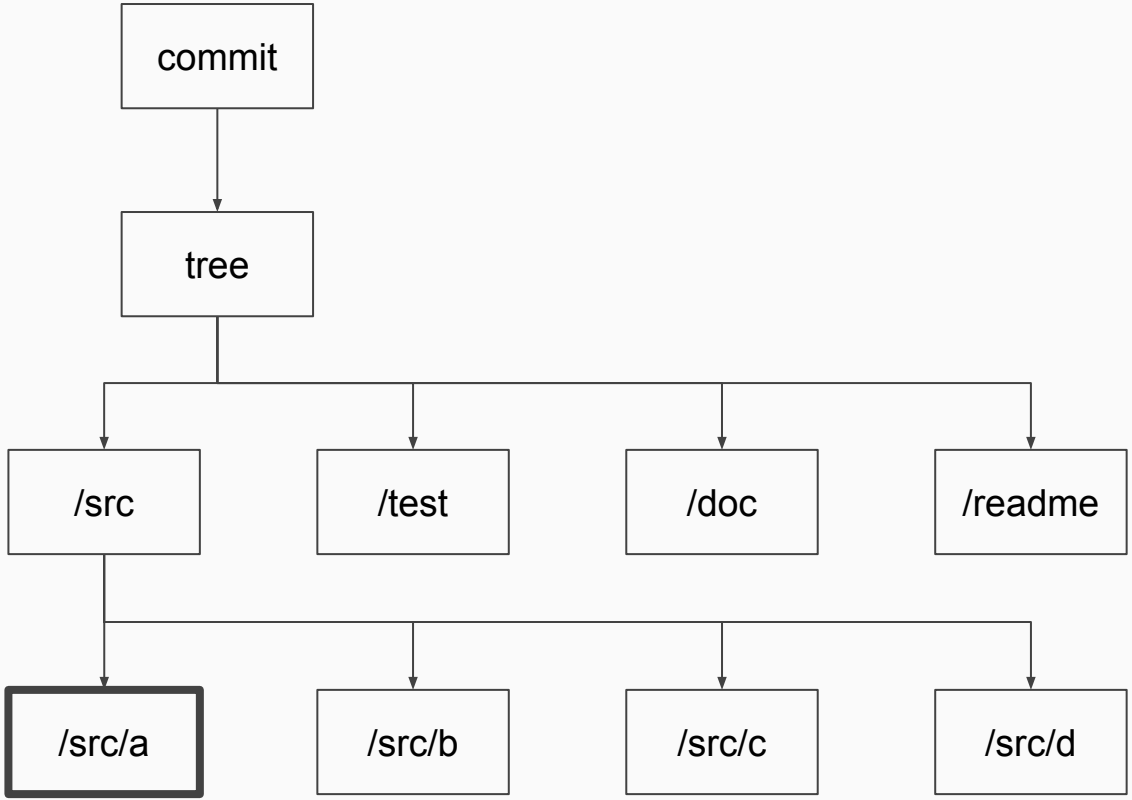
Estructuras de Datos Persistentes



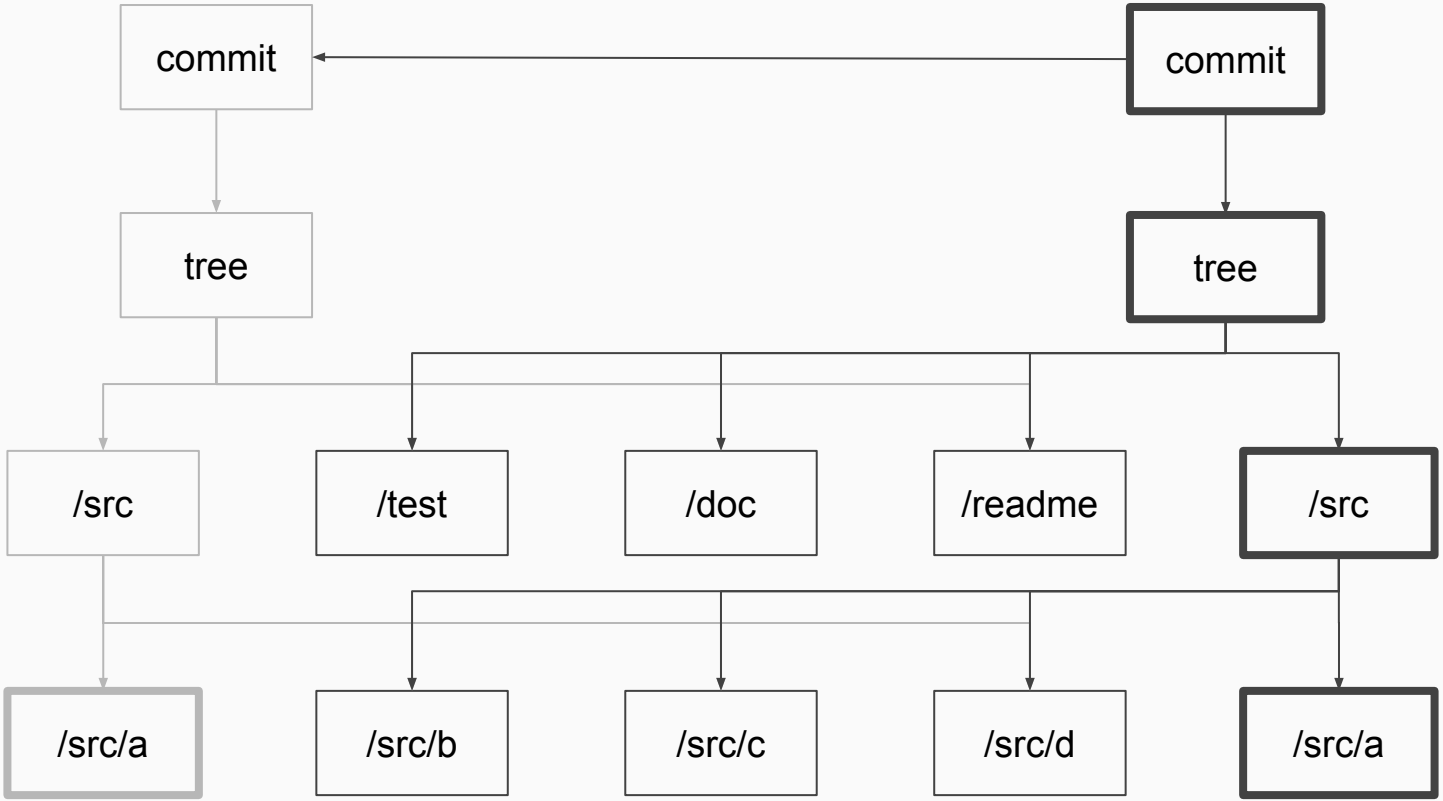
Estructuras de Datos Persistentes



Estructuras de Datos Persistentes



Estructuras de Datos Persistentes



Estructuras de Datos Persistentes

Estructura de datos que preserva sus versiones anteriores al ser modificada

- Uso de almacenamiento depende del patrón de uso
  - Requiere más almacenamiento para cada versión individual
  - Puede compartir almacenamiento entre versiones

- Paralelismo sin locks

Evaluación Perezosa

También llamada **evaluación por necesidad**

Los valores:

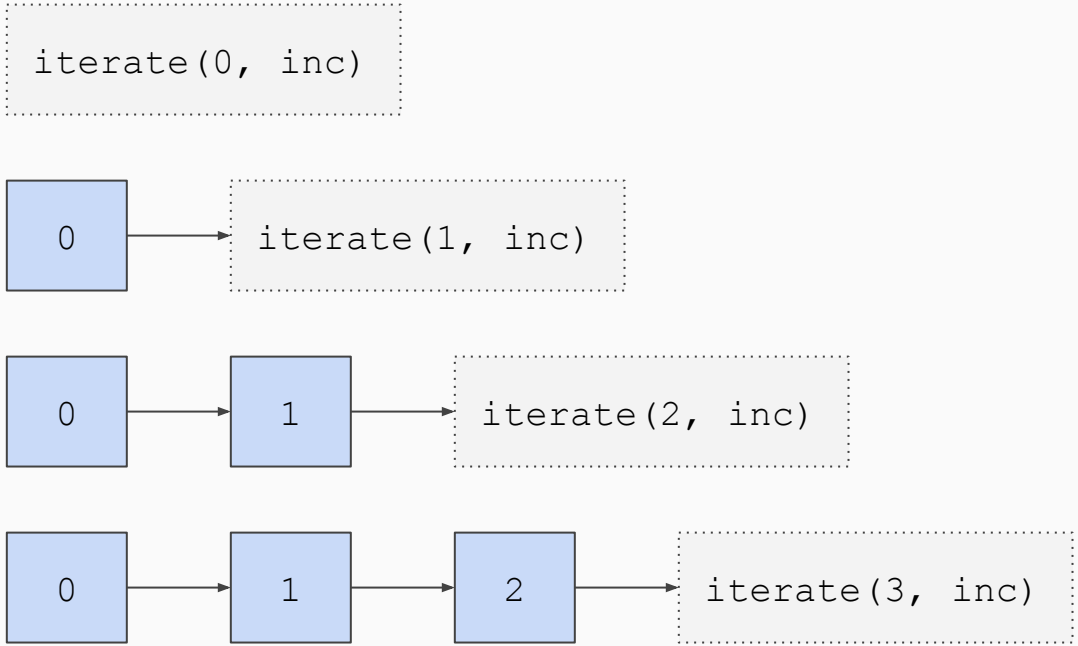
- Se construyen como computaciones suspendidas ("thunks")
- Se computan cuando son necesarios

Usado para:

- Usar una estructura, computando sólo las partes usadas
- Separar definición y uso de una estructura

Evaluación Perezosa

Ejemplo: Tomar los primeros 3 elementos de una lista infinita



No hace falta el resto de la lista infinita

Modelo de Ejecución

Modelo de ejecución

Modelo de ejecución: Especificación del orden y división de tareas al ejecutar un programa.

Dos formas comunes:

- Orden aplicativo
- Orden normal

Modelo de ejecución - Orden normal

Orden Aplicativo

- Para evaluar una función:
  - Evaluar cada argumento
  - En el cuerpo de la función, sustituir cada instancia del argumento por su valor
  - Evaluar el cuerpo de la función
- Repetir hasta llegar a un valor primitivo

Modelo de ejecución - Orden aplicativo

```
sum_of_squares(1 + 1, 2 * 2)
sum_of_squares(  2   ,   4  )

      square(x) + square(y)                [x ← 2, y ← 4]
      square(2) + square(4)

            x * x                           [x ← 2]
            2 * 2
            4

      4 * 16

20
```

Modelo de ejecución - Orden normal

Orden Normal

- Para evaluar una función:
  - Copiar el cuerpo de la función
  - En la copia, sustituir cada instancia de un argumento por su expresión
  - Sustituir el llamado a función por la copia de su cuerpo
- Repetir hasta llegar a un valor primitivo

Modelo de ejecución - Orden normal

```
sum_of_squares(1 + 1, 2 * 2)

square(x) + square(y)                    [x ← 1 + 1, y ← 2 * 2]
square(1 + 1) + square(2 * 2)

x * x + square(2 * 2)                    [x ← 1 + 1]
(1 + 1) * (1 + 1) + square(2 * 2)

2 * (1 + 1) + square(2 * 2)

2 * 2 + square(2 * 2)

4 + square(2 * 2)
```

Modelo de ejecución - Orden normal

```
4 + square(2 * 2)

4 + x * x                                   [x ← 2 * 2]
4 + (2 * 2) * (2 * 2)

4 + 4 * (2 * 2)

4 + 4 * 4

4 + 16

20
```

Modelo de ejecución

Orden Aplicativo

- Evalúa cada argumento exactamente una vez
- Aplicación predecible (orden y cantidad) de efectos colaterales
  - ➢ Lenguajes no-funcionales sólo pueden usar orden aplicativo

Orden Normal

- El programador controla cuántas veces se evalúa cada argumento
  - **Nunca**
  - 1 vez
  - **N veces**
- Puede evaluar funciones que son ciclos infinitos en orden aplicativo

Modelo de ejecución

Ambos modelos de evaluación dan el mismo resultado si:

- Evalúan una función matemática
- Ambos resultan en ciclo infinito o un valor específico
    - Existen casos que son ciclos infinitos solo en orden aplicativo

Modelo de ejecución

Hay funciones que solo son posibles en orden normal:

```
ifThenElse(cond, ifTrue, ifFalse)
```

En lenguajes que solo admiten orden aplicativo, estas funciones necesitan ser casos especiales, parte del lenguaje en sí.

Casos históricos

- Usuarios de common lisp crearon docenas de sistemas de objetos en los 80, mejores rasgos estandarizados (CLOS) en los 90
- try-with-resources de Java

Lenguajes Específicos de Dominio

Lenguajes Específicos de Dominio

Lenguajes especializados para un dominio, sacrificando generalidad por expresividad/usabilidad.

Distintos tipos:

- Embebidos
- Generados
- Interpretados

Lenguajes Específicos de Dominio

Lenguajes Embebidos

- Parte de otro lenguaje de propósito general
- Usan partes su lenguaje anfitrión
  - Simplifica implementación
  - Detalles de implementación no deseados
  - Puede reusar herramientas de desarrollo (Debugger, autocompletar, etc)
- Suele tomar la forma de una librería con modelos conceptuales
  - Especialmente: $PARADIGMA para $LENGUAJE que no lo soporta nativamente

Ejemplos: Java Streams, Spring, React, gradle

Lenguajes Específicos de Dominio

Lenguajes Generados

- Programa externo genera código en su lenguaje anfitrión
- No requieren mucho soporte del lenguaje anfitrión

Ejemplos: ANTLR, bison, babel, go generate, Java Annotations, kapt

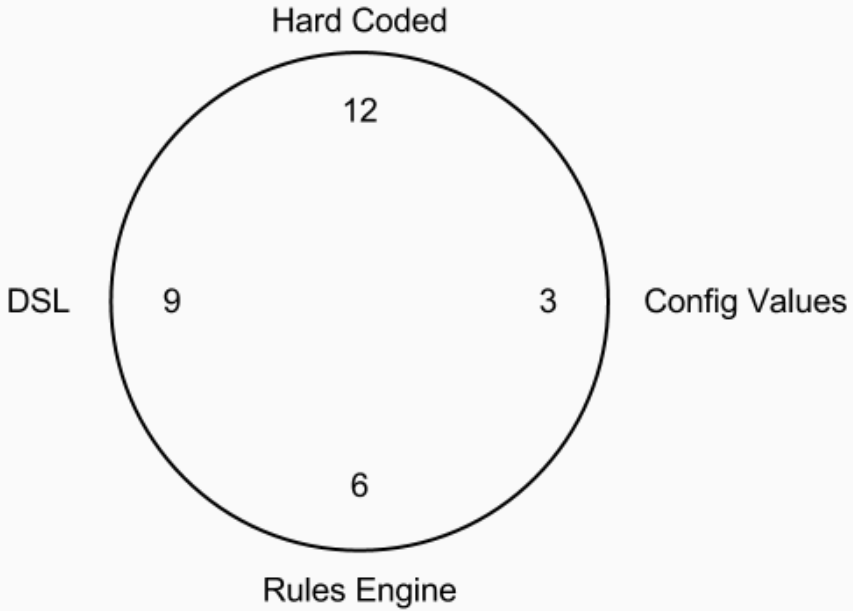
Lenguajes Específicos de Dominio

Lenguajes Interpretados

- El intérprete sólo opera en tiempo de ejecución
- Requiere implementar etapas de parseo y ejecución
- Típico para lenguajes pequeños
  - Mientras más usos tenga, más se agregan características de uso general.
    - Peor caso: Programación en un lenguaje que no fue diseñado para ello

Ejemplos: Calculadoras, configuración
Solía ser ejemplo: PHP

Lenguajes Específicos de Dominio



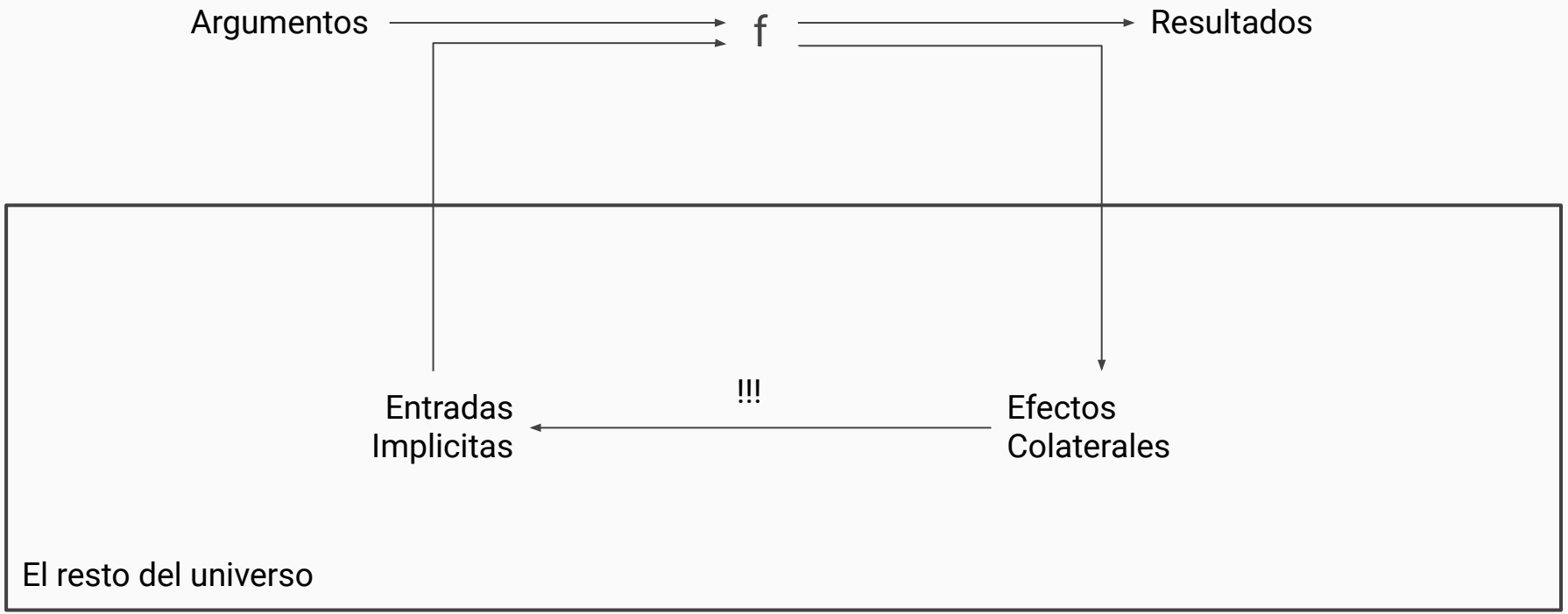
The Configuration Complexity Clock

http://mikehadlow.blogspot.com/2012/05/configuration-complexity-clock.html

Manejo de Efectos Colaterales

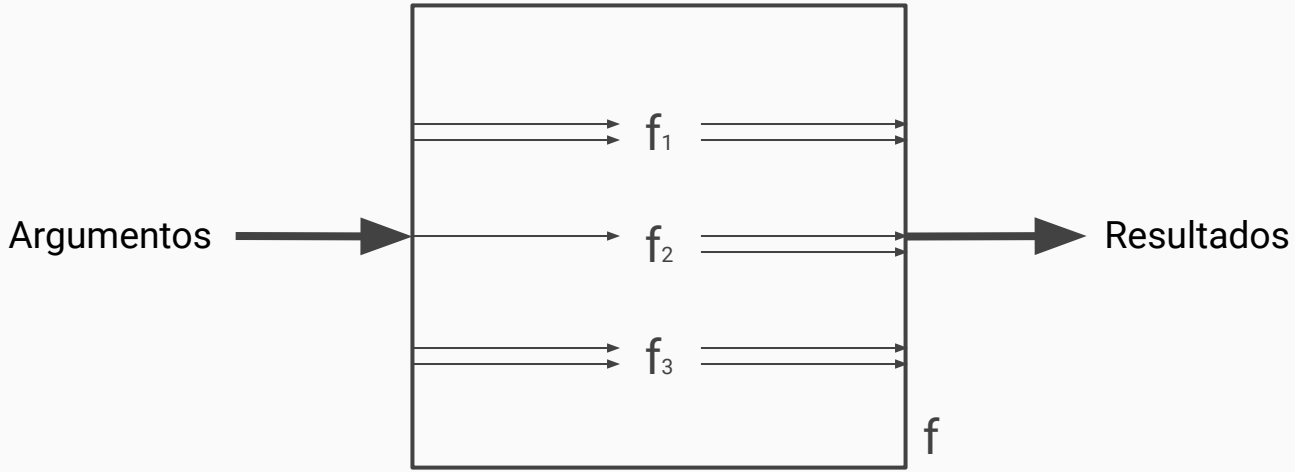
Manejo de Efectos Colaterales

¿Que hay que hacer para obtener una abstracción funcional, si todo lo que tenemos disponible es imperativo?



Manejo de Efectos Colaterales

¿Que hay que hacer para obtener una abstracción funcional, si todo lo que tenemos disponible es imperativo?

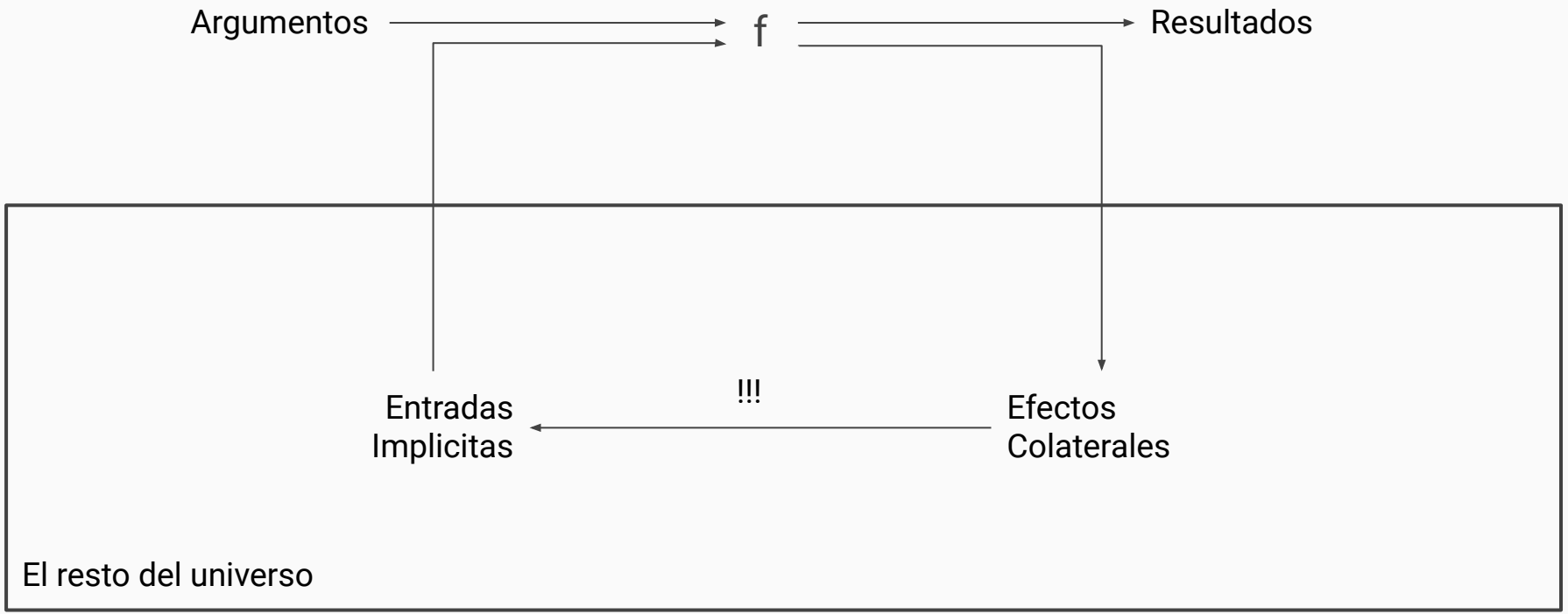


Solución trivial: Componer abstracciones funcionales

➢ ¿Donde obtenemos las funciones primitivas?

Manejo de Efectos Colaterales

¿Que hay que hacer para obtener una abstracción funcional, si todo lo que tenemos disponible es imperativo?



Manejo de Efectos Colaterales

Una forma de obtener abstracciones funcionales si todo lo que tenemos disponible es imperativo:

- Ver que las entradas implícitas y efectos colaterales no son problema **actualmente** e ignorarlas

```
function trace(x) {
    console.log(x);
    return x + CONFIGURACION_QUE_JURO_ES_CONSTANTE;
}
```

Manejo de Efectos Colaterales

Una forma de obtener abstracciones funcionales si todo lo que tenemos disponible es imperativo:

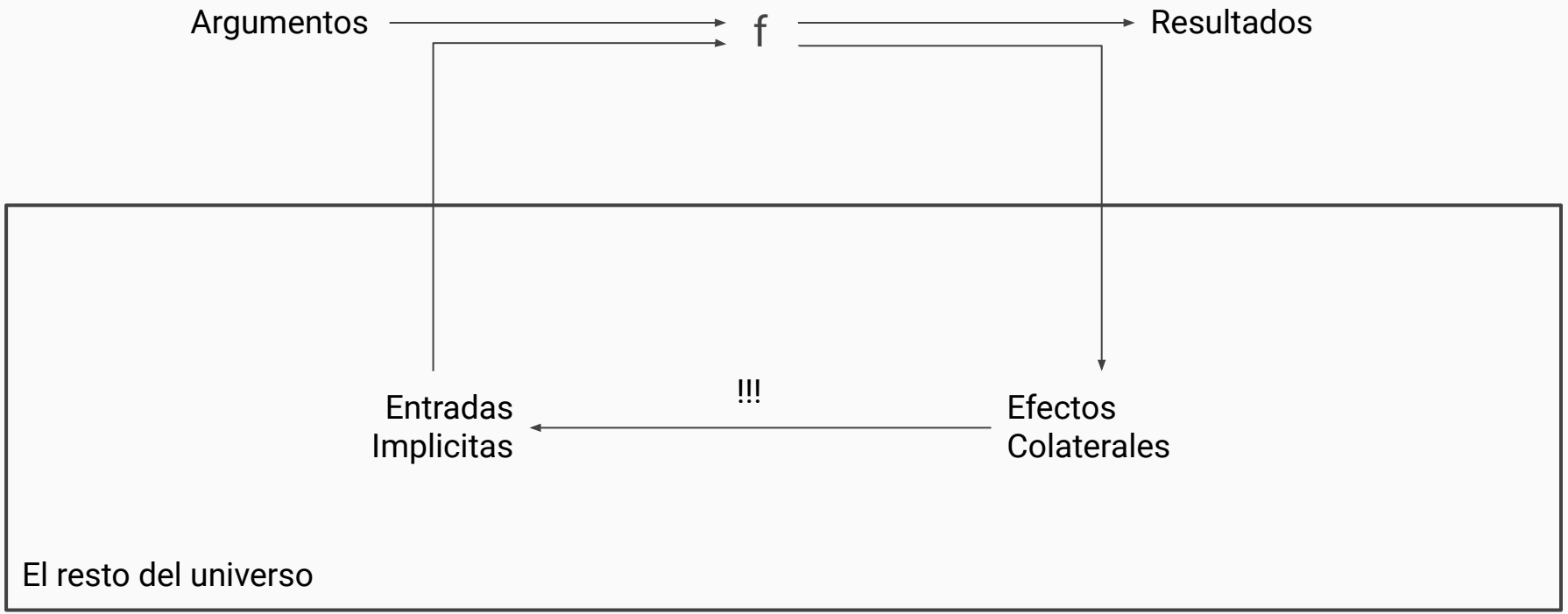
- Ver que las entradas implícitas y efectos colaterales no son problema **actualmente** e ignorarlas

Desventajas:

- Interacción negativa entre abstracciones que funcionan por separado
- Actualmente ≠ actualmente y en el futuro
  - ¿Cómo puede verificarse que esto se mantenga en el tiempo?

Manejo de Efectos Colaterales

¿Que hay que hacer para obtener una abstracción funcional, si todo lo que tenemos disponible es imperativo?



Manejo de Efectos Colaterales

¿Que hay que hacer para obtener una abstracción funcional, si todo lo que tenemos disponible es imperativo?

- Alterar las entradas implícitas (a constante o argumento)

- Invocar la subrutina

- Revertir los efectos colaterales de la subrutina
- Revertir los cambios a entradas implícitas

Manejo de Efectos Colaterales

Estos pasos deberían ser conocidos:

- Set up
- Test
- Tear down

Manejo de Efectos Colaterales

```
tests: /* Código */ -> /* ¿Cumple los requerimientos? */
```

- Buenos tests son funciones matemáticas
  - Predecibles
  - Independientes del orden
  - Independientes de donde se ejecutan

- Testear código imperativo requiere negar efectos colaterales
  - Analizando que no importan
  - Agregando wrappers, mocks, etc

Construir Comandos

Un ejemplo en Haskell:

```
type IO t = RealWorld -> (t, RealWorld)

getLine :: IO String
        -- RealWorld -> (String, RealWorld)

putStrLn :: String -> IO ()
         -- (String, RealWorld) -> RealWorld
```

Construir Comandos

Combinadores:

```
bind :: IO t -> (t -> IO u) -> IO u
```

Constructores:

- Ninguno publico

Casteos:

- Ninguno publico

Construir Comandos

Y un requerimiento de usar este sistema:

```
main :: IO () -- RealWorld -> RealWorld
```

Entonces:

- Necesitamos producir una función
- Nuestras únicas opciones son:
  - Usar las primitivas provistas
  - Usar una combinación de primitivas

➢ Componemos una descripción de las acciones a tomar
➢ El sistema solo invoca el resultado de main

Construir Comandos

- Todas las funciones que interactúan con el mundo lo dicen claramente en su firma

```
evilGetSurfaceArea :: Circle -> IO Float
```

Esta función no tiene ninguna buena razón para interactuar con el mundo

Construir Comandos

Si puede tener razones malas:

```
evilGetSurfaceArea :: Circle -> IO Float

evilGetSurfaceArea c = fireAllMissilesAndReturn (radius c)
```

Construir Comandos

- Todas las funciones que NO interactúan con el mundo lo dicen claramente en su firma

```
getSurfaceArea :: Circle -> Float
```

- Hay un camino directo de funciones IO desde main hasta cada función que se ejecuta

- En ningún punto necesitamos saber que hay dentro de
  - RealWorld
  - Operaciones IO

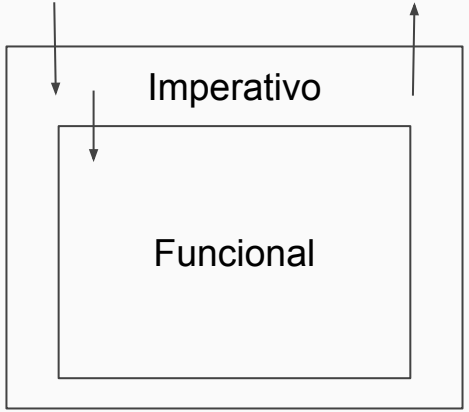
- Este mismo mecanismo puede usarse para explicitar acceso a otras cosas

Functional core, imperative shell

Limita efectos colaterales a las partes que necesitan tenerlos

- Functional core
  - Lógica de dominio
  - Sin acceso al mundo exterior
  - Facil de testear

- Imperative shell
  - Lógica de entrada/salida
  - Dificil de testear



Functional core, imperative shell

```
const mainCycle = () => {
    const world = await getWorldState();
    const input = await getPlayerInputs();

    const nextWorld = nextWorldState(world, input);

    updateWorldState(nextWorld);
    sendPlayerResponse(nextWorld);
}
```

Funcional y Objetos

Funcional y Objetos

¿Que es un objeto?

- Estado
- Comportamiento

Única incompatibilidad: En vez de mutar objetos, creamos objetos nuevos

Funcional y Objetos

En lugar de:

```
setHeight(newHeight: number) {
        this.height = newHeight
}
```

Usamos:

```
withHeight(newHeight: number) {
        return new Rectangle(newHeight, this.width);
}
```

Funcional y Objetos

En java:

```
class A {
    void print(A a) {
     System.out.println("A/A")
    }

    void print(B b) {
     System.out.println("A/B")
    }
}
```

```
class B extends A {
    void print(A a) {
     System.out.println("B/A")
    }

    void print(B b) {
     System.out.println("B/B")
    }
}
```

Funcional y Objetos

```
A a = new A();
A b = new B();

a.print(a) => "A/A"
a.print(b) => "A/A"
b.print(a) => "B/A"
b.print(b) => "B/A"
```

¿Que sucedio?

Funcional y Objetos

Internamente, se tradujo como algo así:

```
A a = new A();
A b = new B();

a.print__A(a) => "A/A"
a.print__A(b) => "A/A"
b.print__A(a) => "B/A"
b.print__A(b) => "B/A"
```

- En Java, Polimorfismo ≅ lookup de un puntero a función en el receptor
  - Hace que solo pueda ser polimórfico en el receptor
  - Hay mejores opciones

Funcional y Objetos

```
f(o, x, y, z)
```

```
o.f(x, y, z)
```

Multimétodos

```
(defmulti my-print (fn [x y] [(class x) (class y)]))

(defmethod my-print [A A] [x y]
  "A/A")
(defmethod my-print [A B] [x y]
  "A/B")
(defmethod my-print [B A] [x y]
  "B/A")
(defmethod my-print [B B] [x y]
  "B/B")
```

- Multimétodo: Extrae una clave de los argumentos
  - Caso comun: Clases
  - Puede ser cualquier valor comparable
- Implementaciones para cada clave

Multimétodos

```
(defmulti crawl-site (fn [url] (extract-url-domain url)))

(defmethod crawl-site "google.com" [url]
  ...)
(defmethod crawl-site "example.com" [url]
  ...)
(defmethod crawl-site "fi.uba.ar" [url]
  ...)
(defmethod crawl-site :default [url]
  ...)
```

Problema de Expresión

Problema de Expresión

```
interface Expr {
    int evaluar();
    String print();
}

record Suma(Expr a, Expr b) {
    int evaluar() {...}
    String print() {...}
}

record Literal(int n) {
    int evaluar() {...}
    String print() {...}
}
```

```
data Expr = Suma Expr Expr
          | Literal Int


evaluar :: Expr -> Int
evaluar (Suma a b) = evaluar a
                   + evaluar b
evaluar (Literal n) = n


print :: Expr -> String
print (Suma a b) = ...
print (Literal n) = show n
```

Problema de Expresión

Deseamos agregar: Expresión multiplicación

```
record Mult(Expr a, Expr b) {
    ...
}

// Otras clases no cambian
```

```
data Expr = Suma Expr Expr
          | Mult Expr Expr
          | Literal Int

-- Todas las funciones cambian

evaluar (Mult a b) = n
print (Mult a b) = ...
```

Problema de Expresión

Deseamos agregar: Operación simplificar

```
interface Expr {
    int evaluar();
    String print();
    Expr simplificar();
}

// Todas las clases cambian

Suma::simplificar
Literal::simplificar
```

```
simplificar :: Expr -> Expr
simplificar (Suma a b) = ...
simplificar (Literal n) = …

-- Otras funciones no cambian
```

Problema de Expresión

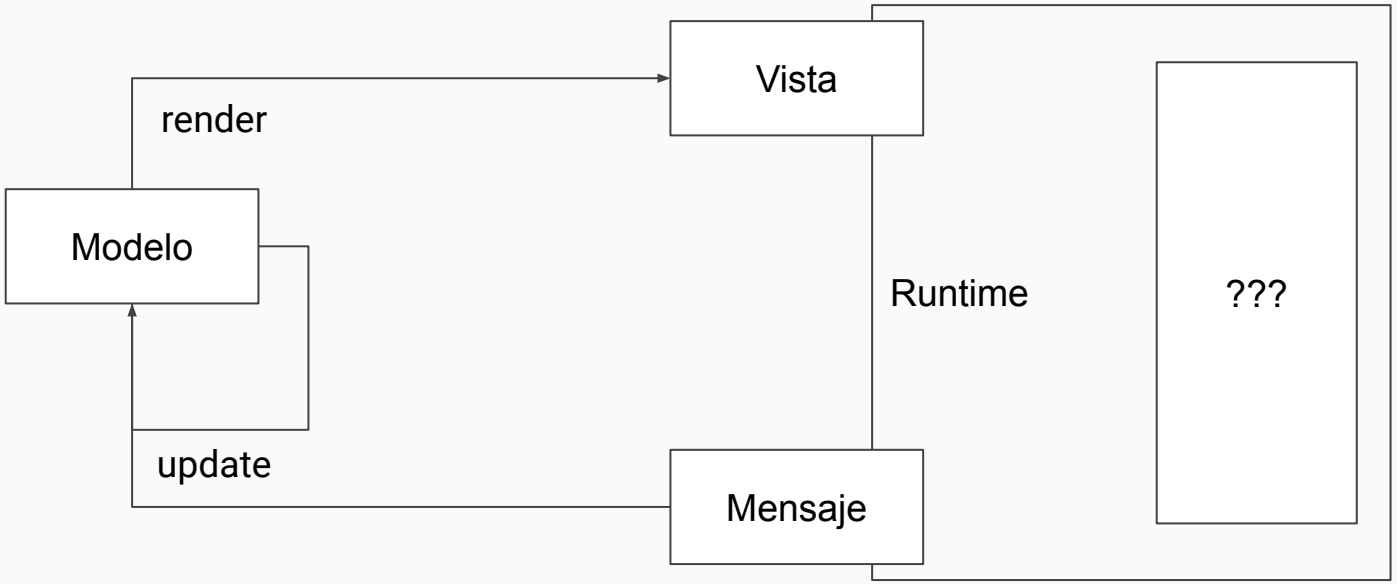
		Casos		
		Suma	Multiplicacion	Literal
Operaciones	Evaluar			
	Imprimir			
	Simplificar			

	Agrega código usando	Casos	Operaciones	Modular
Objetos	Clase = columna	✅		✅
Funcional	Función = fila		✅	✅
Multimétodos	Cuadros arbitrarios	✅	✅	

Aplicaciones Interactivas

Arquitectura Elm

Arquitectura usada para aplicaciones interactivas en el lenguaje Elm



Arquitectura Elm

- **Modelo**: Estado de la aplicación
- **render**: Genera la vista correspondiente al modelo.
- **Vista**: Descripción de la interfaz de usuario deseada, incluyendo mensajes a generar.
- **Mensaje**: Evento indicando que algo sucedió
- **update**: Dado un modelo y un mensaje, genera un nuevo modelo

Arquitectura Elm

El sistema controla todas las entradas y salidas al modelo:

- Agrupar usos de update/render
- Omitir render si el modelo no cambio

En ningún punto necesitamos conocer:

- Cómo se usan las vistas
- Cómo se generan los mensajes

React

Framework para aplicaciones interactivas

```
const Componente = () => {
    return (
        <div>
            <h1>JSX</h1>
            <h2>No es HTML</h2>
        </div>
    )
}
```

React

JSX: Lenguaje específico de dominio similar a HTML, para expresar interfases de usuario, internamente se traduce a algo así:

```
const Componente = () => {
    return React.createElement(
        'div',
        {},
        React.createElement('h1', {}, 'JSX'),
        React.createElement('h1', {}, 'No es HTML')
    )
}
```

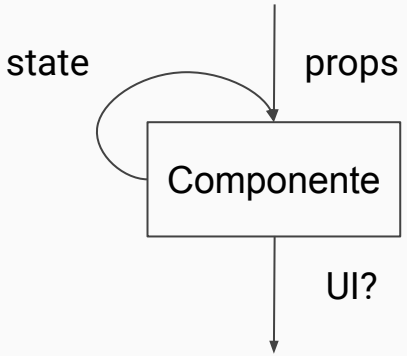
React

Cada componente se trata como una función de:

- Parámetros provistos por un ancestro ("props")
- Estado (mantenido explícitamente por React)

Hooks

- Forma de mantener estado mutable
- El estado es constante durante cada renderizado
    - Mutación controlada por el framework



React

```
const Componente = (props) => {
    const [isConfirmed, setConfirmed] = useState(false)

    const onClick = () => {
        if (isConfirmed) {
            props.onClick();
        } else {
            setConfirmed(true);
        }
    }

    return (
        <button onClick={onClick}>
            {isConfirmed ? "Really delete this" : "Delete this"}
        </button>
    )
}
```

Redux

Usualmente, aplicamos reduce a una **secuencia espacial** (por ejemplo, arrays)

Apliquemos reduce a una **secuencia temporal**

$$estado_N = reduce(eventos, combine, estado_0)$$

$$estado_{N+1} = combine(estado_N , evento_N)$$

Redux

Efectos clave de funcional:

- Los eventos son inmutables
- El estado es persistente
- La función de avance de estado es predecible

Todo puede volverse a usar, sin distinción si es la primer o enésima vez:

➢ Transiciones reproducibles
➢ Time-travel debugging

Redux

Usemos los nombres de redux

state = reducer(action, state)

- Cada acción tiene parámetros "type" y "payload"
- Reducers "hoja":
  - Verifican si es aplicable con type
  - Calculan un nuevo estado con payload

Redux

Reducers compuestos:

```
const userStateReducer = combineReducer({
    token: loginTokenReducer,
    profile: profileReducer
})
```

- Pasa la acción recibida a cada descendiente en paralelo
  - Más de un descendiente puede responder a una acción
- Cada descendiente gestiona su parte (un "slice") del estado compuesto