Programación funcional

Guía de estudio para
Lenguajes de Programación

Universidad Nacional de Colombia

Fundamentos · Composición · Tipos · Evaluación

Equipo de exposición

Nelson Ivan Castellanos Betancourt

Jay Alvarez Baez

Jesús David Giraldo Gomez

Andres Felipe Moreno Pinzón

David Sebastián Hurtado Sanchez

24 de noviembre de 2026

Este material desarrolla el contenido de la exposición de teoría para estudiar, resolver ejemplos y preparar la defensa oral. Sigue un caso de pedidos desde las funciones básicas hasta la composición de resultados, los lotes recursivos y sus costos de ejecución.

Contenido

1 Cómo estudiar esta guía 3
2 Por qué aparece el enfoque funcional 4
3 El caso conductor del reporte 5
4 Funciones como valores y cálculo lambda 7
5 Pureza y razonamiento por ecuaciones 8
6 Inmutabilidad y estructuras persistentes 10
7 Resultados errores y efectos 11
8 De los tipos concretos a Functor Applicative y Monad 13
9 Recorridos acumuladores y monoides 16
10 Tipos algebraicos y lotes anidados 17
11 Terminación y control de la pila 19
12 Evaluación estricta no estricta y perezosa 21
13 Lenguajes y aplicaciones reales 23
14 Ventajas límites y decisiones de ingeniería 25
15 Leyes para consultar y explicar 26
16 Ejercicios para desarrollar dominio 28
17 Soluciones razonadas 30
18 Preparación para la defensa oral 32
19 Glosario y recursos de estudio 34
20 Bibliografía 36

Los números entre corchetes en el texto remiten a la bibliografía. Las ecuaciones están escritas como ecuaciones editables de Word. Los capítulos 16 y 17 contienen ejercicios y soluciones; intenta resolverlos antes de consultar las respuestas.

1 Cómo estudiar esta guía

El objetivo es comprender cómo se diseña, se ejecuta y se justifica un programa funcional. Al terminar, deberías poder explicar los conceptos con tus propias palabras, seguir sus evaluaciones, encontrar contraejemplos a afirmaciones incorrectas y defender decisiones de implementación. La guía desarrolla el contenido de la exposición de teoría de Lenguajes de Programación y usa Scala como puente hacia la exposición práctica. También contrasta OCaml y Haskell para separar las ideas del paradigma de las decisiones de un lenguaje.

El caso conductor es un reporte de pedidos. Primero calcularemos un total sobre una lista; después haremos explícitos los errores; finalmente extenderemos los datos a lotes anidados. Cada extensión plantea una pregunta concreta. ¿De qué depende el resultado? ¿Qué significa fallar? ¿Cómo se combina una computación con otra? ¿Por qué termina el recorrido? ¿Cuánta memoria requiere? Las abstracciones aparecen como respuestas a estas preguntas.

Lee un capítulo, cierra el documento y reconstruye un ejemplo sin mirar. Después cambia una condición: introduce un error, invierte dos pedidos, sustituye una lista por un árbol o modifica la evaluación. Si puedes anticipar qué se conserva y qué cambia, estás usando el concepto y no solamente recordando su definición. Las soluciones al final permiten comprobar el razonamiento.

Una ruta de aprendizaje

Etapa	Lectura y práctica	Resultado comprobable
Fundamentos	Capítulos 2 a 5 y ejercicios 1 a 6	Reducir expresiones y justificar una sustitución
Composición	Capítulos 6 a 9 y ejercicios 7 a 13	Elegir map product o flatMap con una razón
Estructura y ejecución	Capítulos 10 a 12 y ejercicios 14 a 20	Recorrer un lote y explicar sus costos
Criterio de ingeniería	Capítulos 13 y 14 y ejercicios 21 a 24	Comparar alternativas sin prometer ventajas automáticas

Para estudiar en grupo, cada persona puede preparar una etapa o parte de ella, pero todos deben resolver el caso completo. En una segunda reunión, intercambien temas: quien explica recibe una variante que no haya memorizado. La exposición tiene cinco bloques; el dominio del tema no debe depender de que cada integrante conozca únicamente su bloque.

Notación. `A` y `B` representan tipos; `a` y `b`, valores. `A => B` en Scala y `A → B` en las ecuaciones indican funciones. `List[A]` es una lista de elementos de tipo A. `F[A]` representa un tipo construido al aplicar F a A. `Nil` y `[]` representan una lista vacía, según el lenguaje. `x :: xs` es una lista con cabeza x y cola xs. Una igualdad matemática compara significados bajo las hipótesis indicadas; no afirma que dos programas tengan el mismo tiempo de ejecución.

Cómo leer las fuentes. Los números entre corchetes remiten a la bibliografía. Las definiciones y características de los lenguajes se apoyan en documentación oficial y publicaciones académicas. El reporte, sus trazas, los ejercicios y las demostraciones desarrolladas aquí son ejemplos didácticos propios.

Las referencias históricas se presentan con su fecha; un caso industrial de 2015 no describe por sí solo el estado actual del sistema.

2 Por qué aparece el enfoque funcional

El problema de comprender cambios

Imagina un reporte que consulta una variable global de moneda, modifica un acumulador compartido y lee un archivo dentro de su función de suma. Para entender por qué entrega cierto valor necesitas reconstruir qué pasó antes, quién cambió la configuración y cuándo se leyó el archivo. El cálculo está mezclado con una historia de acciones. Esa mezcla puede ser útil en algunas implementaciones, pero aumenta las dependencias que debemos controlar.

Una respuesta funcional consiste en expresar el cálculo mediante valores y funciones, hacer visibles las entradas y componer transformaciones pequeñas. Si la moneda y los pedidos se reciben como argumentos, el reporte puede estudiarse sin conocer una configuración global. Si el error es un resultado explícito, quien llama debe decidir qué hacer con él. Esta organización permite reutilizar y analizar partes del programa. No elimina la necesidad de interactuar con el mundo: da herramientas para organizar esa interacción. [1, 6, 21]

No hubo una sola necesidad ni un único momento de invención. La lógica matemática buscó modelos del cálculo; Lisp atendió necesidades de procesamiento simbólico; la familia ML se relacionó con sistemas de demostración; otros trabajos cuestionaron el acoplamiento entre programas y actualización de estado. Es más preciso hablar de varias tradiciones que confluyeron que de una reacción única contra los bucles o contra la orientación a objetos.

Una historia que explica las ideas

Momento	Aporte	Lectura correcta
Década de 1930	Church desarrolla el cálculo lambda como modelo formal	Una base matemática no es todavía un lenguaje industrial
1958 y 1960	Lisp comienza a implementarse en 1958 y el artículo de McCarthy aparece en 1960	Funciones y recursión conviven con características que permiten efectos
Década de 1970	ML surge en el entorno de LCF y la demostración de teoremas	Tipos e inferencia ayudan a construir programas simbólicos
1977 y 1978	Conferencia de Backus en 1977 y publicación en 1978	Propone formas de combinación para reducir dependencia del estado y la máquina
1987 y 1990	Comité de Haskell en 1987 y versión 1.0 en 1990	Se busca una base común para lenguajes funcionales no estrictos

Las fechas de Lisp distinguen el inicio de implementación de la publicación del artículo. De modo parecido, la conferencia de Backus y su publicación no pertenecen al mismo año. Haskell tampoco es el

primer lenguaje funcional. Estas distinciones evitan convertir la historia en una lista de fechas sin contexto. [9, 10, 34, 35, 36, 37]

La idea central de Hughes es que las funciones de orden superior y la evaluación perezosa pueden ofrecer nuevas formas de modularidad. Para nuestra exposición, esto se traduce en una pregunta: ¿podemos separar una operación que se repite de la función específica que la completa? `map`, `fold` y los recorridos de lotes responderán esa pregunta desde situaciones distintas. [1]

Paradigma y lenguaje son niveles diferentes

Un paradigma organiza la manera de formular soluciones. Un lenguaje ofrece sintaxis, tipos, reglas de evaluación y herramientas que pueden favorecer varios paradigmas. Scala combina programación funcional y orientación a objetos; OCaml también admite objetos y mutabilidad; Common Lisp permite varios estilos. Usar uno de estos lenguajes no convierte automáticamente cualquier programa escrito en él en funcional. [13, 17, 19, 39]

Tampoco hay que identificar programación funcional con tipado estático, evaluación perezosa o ausencia absoluta de asignaciones. Haskell, OCaml y Clojure toman decisiones diferentes. En esta guía nos interesan especialmente las funciones como valores, la composición, el razonamiento sobre expresiones, los datos inmutables y la representación explícita de resultados y efectos. Más adelante distinguiremos cuáles propiedades corresponden a cada lenguaje.

3 El caso conductor del reporte

Contrato antes del código

Un pedido tiene identificador, estado, importe textual y moneda. El reporte debe sumar solamente los pedidos pagados en COP. Los importes representan cantidades enteras de centavos de COP; no se admiten cantidades negativas. Si un pedido seleccionado tiene un importe inválido, el resultado identifica el primer pedido que falla en el orden de entrada. Si no hay pedidos seleccionados, el resultado es cero.

Esta especificación fija decisiones que una palabra como “sumar” no resuelve. Filtrar antes de validar significa que un pedido pendiente con importe inválido queda fuera del reporte. Si se exigiera auditar todos los registros, habría que cambiar el contrato y validar también los excluidos. Ambas políticas son posibles; no son equivalentes.

Pedido	Estado	Importe en centavos	Moneda	Participa
A	Pagado	12000	COP	Sí
B	Pendiente	9000	COP	No
C	Pagado	??	COP	Sí y falla
D	Pagado	8000	COP	Sí

La selección produce A, C y D. A aporta 12000; C no se puede interpretar como entero; el reporte devuelve `Left(EnPedido("C", Formato))`. No debe devolver 20000 ni esconder el error reemplazando

C por cero. Si corregimos C a "5000", obtenemos 12000 + 5000 + 8000 = 25000 centavos, equivalentes a 250 COP. Es esencial mantener la unidad: 25000 centavos no son 25000 pesos.

Usaremos `BigInt` para evitar el desbordamiento de un entero de tamaño fijo en la suma. Eso no hace constante el costo de las operaciones aritméticas: procesar más dígitos requiere trabajo. En un sistema real también habría reglas de formato, escala decimal, monedas y límites de entrada. El ejemplo acepta el formato entero reconocido por `BigInt` después de quitar espacios externos.

El modelo de datos

Scala 3

```
import scala.util.Try

enum Error:
  case Formato, Negativo
  case EnPedido(id: String, causa: Error)

enum Estado:
  case Pagado, Pendiente

case class Pedido(
  id: String,
  estado: Estado,
  importe: String,
  moneda: String
)

def parsear(s: String): Either[Error, BigInt] =
  Try(BigInt(s.trim)).toEither.left.map(_ => Error.Formato)

def noNegativo(n: BigInt): Either[Error, BigInt] =
  if n >= 0 then Right(n) else Left(Error.Negativo)

def importe(s: String): Either[Error, BigInt] =
  parsear(s).flatMap(noNegativo)
```

`Either[Error, BigInt]` tiene dos alternativas: `Left(error)` describe un fallo y `Right(n)` un éxito. La convención de Scala hace que `map` y `flatMap` operen sobre el lado derecho. El código adapta una conversión que puede lanzar una excepción mediante `Try` y transforma el fallo esperado de formato en un valor del dominio. No promete capturar cualquier fallo posible de la máquina ni eliminar todos los errores de programación. [24]

Todavía no necesitas dominar `flatMap`. Léelo provisionalmente así: si `parsear` obtiene un número, úsalo para ejecutar `noNegativo`; si falla, conserva el error. El capítulo 8 explicará su tipo, por qué hace falta y qué leyes satisface. Esta lectura anticipada permite seguir el problema sin introducir una abstracción sin motivación.

4 Funciones como valores y cálculo lambda

Recibir devolver y componer funciones

Una función de primera clase puede usarse como un valor: pasarse como argumento, devolverse o almacenarse en una estructura. Una función de orden superior recibe funciones o devuelve una función. Así, `filter` separa el recorrido de la decisión particular sobre cada elemento. En el reporte, el recorrido no necesita saber qué significa “pagado en COP”; recibe un predicado que lo decide. [3, 4, 5, 18]

Una clausura combina el comportamiento de una función con el entorno que necesita para interpretar sus variables libres. Por ejemplo, una función que compara un importe con un límite recuerda ese límite. Si captura una referencia mutable y esta cambia, su resultado puede cambiar aunque se invoque con el mismo argumento. Ser una clausura no implica ser pura.

Scala 3

```
def mayorQue(limite: BigInt): BigInt => Boolean =
  n => n > limite

val esGrande = mayorQue(BigInt(10000))
val resultado = esGrande(BigInt(12000)) // true

val seleccionar: Pedido => Boolean =
  p => p.estado == Estado.Pagado && p.moneda == "COP"
```

En `mayorQue`, la primera llamada produce una función; la segunda compara el importe. La currificación representa una función de varios argumentos mediante una sucesión de funciones de un argumento. La aplicación parcial consiste en suministrar solo parte de los argumentos y obtener una función para los restantes. Se relacionan, pero no son la misma operación: una es una forma de representar funciones y la otra, una manera de aplicarlas. [51]

$$f : B \to C \qquad g : A \to B \qquad (f \circ g)(x) = f(g(x))$$

La composición exige que la salida de g sea una entrada válida para f. El orden se lee desde la aplicación interior: primero g, después f. En Scala, `g.andThen(f)` representa esa secuencia. `f.compose(g)` expresa la misma composición. El tipo descarta algunas conexiones inválidas; no demuestra que las funciones respeten todas las reglas del negocio.

La notación lambda sin saltos

El cálculo lambda básico construye expresiones con variables, abstracciones y aplicaciones. `λx. e` expresa una función cuyo parámetro es x y cuyo cuerpo es e. $e_1\ e_2$ aplica la primera expresión a la segunda. La aplicación se asocia a la izquierda: `f x y` significa `(f x) y`. El cuerpo de una lambda se extiende hacia la derecha hasta donde indiquen los paréntesis. Aquí usamos enteros y suma como extensiones didácticas, sin desarrollar su codificación en el cálculo lambda puro. [3, 16]

Una aparición de variable está ligada si queda dentro del alcance de la lambda que la introduce. Si no tiene una ligadura correspondiente, está libre. En `λx. x + limite`, x está ligada y limite está libre. La reducción beta sustituye las apariciones libres del parámetro en el cuerpo por el argumento, evitando capturar variables. [50]

$$(\lambda x.\ e)\ a \ \rightarrow_\beta\ e[x := a]$$

Definamos `dosVeces = λf. λx. f(f x)` y `g = λn. n + 10`. Queremos calcular `dosVeces g 20`. Cada línea elimina una aplicación de una lambda a un argumento o realiza una suma. No se reemplaza una letra por parecido visual: se respeta qué parámetro liga cada aparición.

$$((\lambda f.\ \lambda x.\ f(f\ x))\ g)\ 20$$
$$\rightarrow_\beta\ (\lambda x.\ g(g\ x))\ 20$$
$$\rightarrow_\beta\ g(g\ 20)$$
$$\rightarrow_\beta\ g(20\ +\ 10)\ \rightarrow\ g\ 30$$
$$\rightarrow_\beta\ 30\ +\ 10\ \rightarrow\ 40$$

La función g se usa dos veces, pero eso no significa que su definición cambie. Se construye una expresión con dos aplicaciones. En programación, el mismo patrón permite ejecutar dos transformaciones sucesivas sobre un valor: normalizar dos etapas, aplicar una regla reiterada o componer validaciones cuando sus tipos coinciden.

Renombrar para no capturar variables

Considera `(λx. λy. x) y`, donde la y que aparece como argumento está libre. Sustituir x por y sin cuidado produciría `λy. y`, que devuelve su argumento y cambia el significado. Primero renombramos la variable ligada interior a z. Esta conversión alfa no cambia qué función representa el término. Después realizamos la sustitución.

$$(\lambda x.\ \lambda y.\ x)\ y\ =_\alpha\ (\lambda x.\ \lambda z.\ x)\ y\ \rightarrow_\beta\ \lambda z.\ y$$

La y final sigue libre. El resultado es una función constante respecto de su argumento z, cuyo valor depende del entorno de y. Esta distinción importa cuando se implementan transformaciones de expresiones y cuando se razona sobre alcance de nombres. Para seguir la evaluación correctamente, identifica siempre qué lambda liga cada aparición de variable.

5 Pureza y razonamiento por ecuaciones

Dependencias explícitas y comportamiento observable

Una función pura entrega el resultado correspondiente a sus argumentos y no produce efectos observables externos al cálculo. Si el porcentaje de impuesto viene de una variable global mutable o la función escribe un archivo, la llamada tiene dependencias o efectos que no aparecen como valores ordinarios en su firma. Hacer explícitos los argumentos reduce esa información oculta. [8, 21]

Inmutabilidad y pureza se ayudan, pero son conceptos distintos. `val` impide reasignar una referencia de Scala; no garantiza que el objeto al que apunta sea inmutable. Una función puede recibir un objeto mutable y consultarlo. También puede encapsular una mutación local sin que se observe fuera, si el estado no escapa ni afecta llamadas futuras. El criterio relevante para la pureza es el comportamiento observable, no contar asignaciones en el archivo.

La transparencia referencial permite sustituir una expresión por otra con el mismo significado sin cambiar el comportamiento observable del programa. Para el caso del reporte, sustituir `importe("5000")` por `Right(BigInt(5000))` es razonable bajo el contrato y sin efectos ocultos. Sustituir una lectura de teclado por el texto que devolvió una vez no preserva en general el comportamiento de lecturas futuras.

Una función pura puede no terminar. Tampoco el tipo `Int => Int` de Scala acredita por sí mismo pureza o totalidad: una implementación puede imprimir, lanzar una excepción o entrar en un bucle. Una función total debe producir un resultado definido para todas las entradas de su dominio. Para demostrar leyes que usan sustitución sin condiciones adicionales, aquí trabajaremos con listas finitas y funciones puras y totales.

La ley de composición de map

`map` transforma cada elemento sin decidir cuántos elementos conservar en una lista. Si primero aplicamos g a todos y después f a todos, los valores finales coinciden con aplicar la composición f después de g a cada elemento. Esta ley permite justificar una transformación del programa; no necesita que las funciones sean sumas ni que la lista tenga tres elementos. [5, 15, 28]

$$\mathrm{map}(f,\ \mathrm{map}(g,\ \mathrm{xs})) = \mathrm{map}(f \circ g,\ \mathrm{xs})$$

Tomemos g(n) = n + 10, f(n) = 2n y xs = [1, 2, 3]. El primer recorrido produce [11, 12, 13] y el segundo [22, 24, 26]. La composición transforma cada n directamente en 2(n + 10), con el mismo resultado. El ejemplo ayuda a entender la igualdad; la prueba debe abarcar todas las listas finitas.

Demostración por inducción estructural

Definimos map por los constructores de la lista. Sobre la lista vacía devuelve la lista vacía; sobre una cabeza y una cola transforma la cabeza y continúa con la cola. La estructura del dato determina los casos de la prueba. [45]

$$\mathrm{map}(f,\ [\,]) = [\,]$$
$$\mathrm{map}(f,\ x :: \mathrm{xs}) = f(x) :: \mathrm{map}(f,\ \mathrm{xs})$$

Caso base. Si xs es vacía, el lado izquierdo es `map(f, map(g, [])) = map(f, []) = []`. El lado derecho es `map(f ∘ g, []) = []`. Por tanto, coinciden.

Hipótesis inductiva. Suponemos que para una cola xs se cumple `map(f, map(g, xs)) = map(f ∘ g, xs)`. Esta hipótesis se refiere a una estructura menor; no estamos suponiendo directamente el resultado para la lista completa.

Paso inductivo. Para la lista `x :: xs`, expandimos las definiciones y usamos la hipótesis solo sobre la cola.

$$\begin{aligned}
&\mathrm{map}(f,\ \mathrm{map}(g,\ x :: \mathrm{xs})) \\
&= \mathrm{map}(f,\ g(x) :: \mathrm{map}(g,\ \mathrm{xs})) \\
&= f(g(x)) :: \mathrm{map}(f,\ \mathrm{map}(g,\ \mathrm{xs})) \\
&= (f \circ g)(x) :: \mathrm{map}(f \circ g,\ \mathrm{xs}) \\
&= \mathrm{map}(f \circ g,\ x :: \mathrm{xs})
\end{aligned}$$

Con el caso base y el paso inductivo queda demostrada la igualdad para todas las listas finitas bajo las hipótesis indicadas. La tercera igualdad utiliza la hipótesis inductiva. Esta es la diferencia entre comprobar ejemplos y demostrar una propiedad de una familia completa de estructuras.

Qué no demuestra la ley

Si f y g imprimen mensajes, la versión de dos recorridos ejecuta primero todos los efectos de g y luego todos los de f; la versión compuesta alterna g y f por elemento. Los valores podrían coincidir, pero la secuencia de mensajes no. La ley, aplicada como equivalencia observable, necesita controlar esos efectos.

La igualdad tampoco implica igual costo. En una lista estricta, dos `map` ordinarios pueden construir una lista intermedia y la composición puede evitarla. Eso depende de la implementación y de las optimizaciones disponibles. Una ley autoriza un razonamiento semántico; medir rendimiento requiere además un modelo de costos o un perfil del programa. No se debe afirmar que cualquier compilador fusiona automáticamente los recorridos.

6 Inmutabilidad y estructuras persistentes

Conservar una versión sin duplicarlo todo

Una estructura persistente permite seguir usando versiones anteriores después de producir una nueva. Persistencia no significa guardar en disco. Significa que una actualización no destruye la versión que existía. La compartición estructural hace posible reutilizar las partes que no cambian, en lugar de copiar todos los datos. [19, 26]

En una lista enlazada inmutable, anteponer un elemento puede construir una celda nueva cuya cola apunta a la lista anterior. La versión antigua conserva su primera celda; la nueva agrega otra delante. Las dos versiones comparten la cola. Esto es seguro porque ninguna modifica esas celdas. Si las referencias compartidas permitieran cambios observables, una modificación podría alterar ambas versiones.

La utilidad aparece al mantener historiales, estados de una interfaz o alternativas durante una búsqueda. Podemos comparar versiones sin reconstruirlas a partir de un registro de mutaciones. El costo no desaparece: se asignan nuevos nodos y las versiones todavía accesibles impiden liberar los nodos que alcanzan. Persistencia no equivale a memoria gratuita.

Inserción en un árbol de búsqueda

Considera un árbol de búsqueda con raíz 8, hijo izquierdo 3 y derecho 10. El nodo 3 tiene hijos 1 y 6. Queremos insertar 4. Seguimos el camino 8 → 3 → 6 → vacío izquierdo. Creamos el nodo 4 y reconstruimos los antecesores de ese camino con los enlaces actualizados. Los subárboles de 1 y 10 se comparten con la versión original.

Los nodos nuevos son 4, una nueva versión del nodo 6, una del nodo 3 y una de la raíz 8. No se modifica el nodo 6 anterior ni se copia el árbol completo. Aunque dos nodos tengan el mismo valor, pueden ser objetos diferentes: el nuevo 6 apunta a 4 y el anterior no. Esta diferencia entre valor y referencia explica las líneas de compartición del diagrama de la exposición.

Scala 3

```
enum Arbol:
  case Vacio
  case Nodo(valor: Int, izq: Arbol, der: Arbol)

def insertar(x: Int, t: Arbol): Arbol = t match
  case Arbol.Vacio =>
    Arbol.Nodo(x, Arbol.Vacio, Arbol.Vacio)
  case n @ Arbol.Nodo(v, i, d) =>
    if x < v then Arbol.Nodo(v, insertar(x, i), d)
    else if x > v then Arbol.Nodo(v, i, insertar(x, d))
    else n
```

La variable n permite devolver el nodo existente si el valor ya estaba. En las otras ramas, el constructor recibe un subárbol nuevo y otro reutilizado. El algoritmo conserva la propiedad de búsqueda: todos los valores de la izquierda son menores y los de la derecha mayores, suponiendo que el árbol original ya la cumplía.

Para una altura h, el tiempo de inserción y el número de nodos nuevos son O(h). Solo podemos afirmar O(log n) si la altura permanece logarítmica, por ejemplo mediante balanceo. Este código no rebalancea: una secuencia de inserciones ordenadas puede producir altura O(n). La inmutabilidad no garantiza balance ni complejidad logarítmica.

Cuándo elegir otra estructura

Una lista es adecuada para construir desde el principio y recorrer secuencialmente; no es ideal para acceder muchas veces por índice. Un vector persistente o un árbol balanceado ofrece otros compromisos. En una sección intensiva de cálculo numérico, un arreglo y mutación local pueden ser una elección razonable. La pregunta de ingeniería es qué operaciones dominan y qué garantías necesitamos conservar. [25]

No hace falta copiar un objeto completo cada vez que cambia una parte del programa. Tampoco conviene concluir que cualquier estructura inmutable será eficiente. Debemos identificar qué se comparte, qué se asigna y cuánto tiempo conservamos cada raíz.

7 Resultados errores y efectos

Un error del dominio también es un valor

`Option[A]` indica que puede existir un A o no existir. No informa por sí mismo la causa de la ausencia. `Either[E, A]` distingue un error E de un resultado A. Para el reporte necesitamos identificar si el formato es inválido, el importe es negativo y qué pedido falló. Por eso usamos un tipo de error explícito en lugar de devolver cero o `None`. [24]

Una excepción puede interrumpir el flujo normal sin aparecer en el tipo de retorno ordinario de Scala. Un resultado `Either` hace visible una alternativa que el programa puede inspeccionar y combinar. Esto no vuelve incorrecto todo uso de excepciones: en el ejemplo, `Try` sirve como adaptación de una operación existente. El núcleo trabaja después con resultados explícitos y una política de propagación.

Hay que distinguir un dato inválido de un fallo de infraestructura. `Negativo` describe una regla del negocio; un archivo inaccesible o una conexión perdida pertenecen a otro contexto. Es posible modelar ambos con tipos, pero no conviene borrar la diferencia bajo un mensaje genérico. La precisión del tipo ayuda a decidir si corregir la entrada, reintentar o informar al usuario.

El cálculo completo de la lista

Scala 3

```
def reporte(ps: List[Pedido]): Either[Error, BigInt] =
  val seleccionados = ps.filter(p =>
    p.estado == Estado.Pagado && p.moneda == "COP")
  seleccionados.foldLeft[Either[Error, BigInt]](
    Right(BigInt(0))
  ) { (acc, p) =>
    acc.flatMap { total =>
      importe(p.importe)
        .left.map(e => Error.EnPedido(p.id, e))
        .map(n => total + n)
    }
  }
```

El acumulador no es solamente un número: es un número o un error. Empieza como éxito con cero. Para cada pedido, `acc.flatMap` permite sumar únicamente si todavía existe un total válido. `left.map` agrega el identificador al error sin tocar un éxito; el `map` final agrega el importe al total sin tocar un error.

Con A válido, C inválido y D válido, los estados del acumulador son `Right(0)`, `Right(12000)`, `Left(EnPedido(C, Formato))` y el mismo `Left` al procesar D. Hay una precisión importante: `foldLeft` sigue recorriendo la lista seleccionada después de encontrar un error. Lo que se omite es el cuerpo de la continuación de `flatMap`, por lo que D no se valida. Esta implementación no corta físicamente el recorrido de la lista. [22]

Separar el núcleo de la interacción

Leer un archivo, obtener la hora, consultar una base de datos y publicar un resultado son interacciones con el entorno. Sumar pedidos recibidos como datos puede ser un cálculo puro. Una arquitectura posible lee y adapta los datos en el borde, llama al núcleo y después decide cómo mostrar o guardar el resultado. Así se pueden probar las reglas sin ejecutar la infraestructura.

En Haskell, una expresión de tipo `IO A` describe una acción cuyo resultado tendrá tipo A al ejecutarse. Construir o componer esa descripción no equivale a disponer ya de un A puro. La ejecución de las acciones pertenece al mecanismo de E/S del lenguaje. Las mónadas proporcionan una forma de composición para muchos contextos; no son sinónimo de E/S. [6]

Separar estas partes no significa que todas las dependencias difíciles desaparezcan. La capa de interacción necesita pruebas de integración, manejo de recursos y políticas de fallos. La mejora es poder identificar qué estamos probando: una regla del negocio o su conexión con un sistema externo.

8 De los tipos concretos a Functor Applicative y Monad

Por qué aparece F de A

Ya conocemos tres formas de tener valores: una lista de A, una opción de A y un resultado que puede fallar con E o entregar A. Antes de introducir `F[A]`, observa sus ejemplos concretos: `List[BigInt]`, `Option[BigInt]` y `Either[Error, BigInt]`. En los tres interesa transformar un BigInt, pero el comportamiento alrededor de ese valor cambia.

`List` construye un tipo al recibir el tipo de sus elementos. `Option` hace algo parecido. `Either` recibe dos parámetros; para tratarlo aquí como F fijamos el error y dejamos variar el éxito. En Scala podemos escribir `type Resultado[A] = Either[Error, A]`. Entonces F es Resultado y A puede ser BigInt, String u otro tipo. F no es el número de elementos ni una variable que contiene un pedido.

La palabra “contexto” resume una conducta concreta: posibilidad de ausencia, varios resultados o una alternativa de error. No debe usarse como explicación circular. Para una estructura nueva hay que decir qué representa, cómo se transforma y qué leyes cumple. Tampoco F se limita a cajas: una función o una descripción de acción puede admitir estas abstracciones.

Functor transforma el valor conservando la estructura pertinente

$$\text{map} : F[A] \times (A \to B) \to F[B]$$

Un Functor proporciona una operación map que cambia los valores transformables de A a B y cumple identidad y composición. Para List, aplica la función a cada elemento preservando su posición. Para Option, transforma Some y conserva None. Para Either con E fijo, transforma Right y conserva Left. Estas operaciones son diferentes implementaciones de una misma interfaz con leyes. [28, 49]

Ejemplo propio: si f(n) = n / 100 usando división entera, `Some(25000).map(f)` produce `Some(250)` y `None.map(f)` sigue ausente. Para `Right(25000)` obtenemos `Right(250)`; un error de formato permanece igual. Si necesitamos preservar centavos para mostrar dinero, esa división entera no sirve como formateador general: el tipo no expresa automáticamente todas las exigencias del dominio.

$$\text{map}(\text{id}, \text{fa}) = \text{fa}$$

$$\text{map}(g, \text{map}(f, \text{fa})) = \text{map}(g \circ f, \text{fa})$$

La identidad exige que transformar con una función que devuelve lo mismo no altere la estructura observable. La composición exige que agrupar transformaciones no cambie el significado. En esta segunda ecuación f se aplica primero y g después; en el capítulo 5 usamos otros nombres para la misma ley. Lo que importa es seguir el orden, no memorizar las letras.

Una operación que transforma la lista y además invierte el orden no sería el map habitual: con identidad devolvería la lista invertida y violaría la primera ley salvo casos particulares. Tener la firma correcta no basta para ser una instancia válida. Las pruebas con listas pequeñas detectan errores, pero no sustituyen la demostración o la justificación general de las leyes.

Una clase de tipos describe operaciones parametrizadas por un tipo; una instancia proporciona su implementación para un caso concreto. Una función genérica con una instancia de Functor para F puede

transformar F[A] sin conocer su representación interna. En Haskell se usa esta noción de clase de tipos; en Scala una biblioteca puede expresarla con un trait y un valor que proporcione las operaciones de la instancia. No es necesario que el dato herede de una superclase orientada a objetos para describir la capacidad abstracta. Además, los funtores del sistema de módulos de OCaml son otra noción: módulos parametrizados por módulos. Comparten un nombre histórico, pero no debemos confundirlos con esta interfaz map.

Applicative combina computaciones conocidas de antemano

Supón que recibimos por separado un precio y una cantidad, ambos textuales. Validar la cantidad no necesita conocer el precio validado. Podemos describir las dos validaciones antes de combinarlas. Un Applicative ofrece `pure` y una aplicación dentro de F, habitualmente llamada `ap`; con ellas se pueden construir operaciones como product y map2. [29, 46]

$$\text{pure} : A \to F[A]$$

$$\text{ap} : F[A \to B] \times F[A] \to F[B]$$

$$\text{product} : F[A] \times F[B] \to F[(A, B)]$$

`pure` introduce un valor sin agregar un efecto propio del contexto: en nuestro resultado, `pure(a)` es `Right(a)`. `product` combina dos resultados; `map2` combina y aplica una función de dos argumentos. La palabra “independiente” significa que la descripción de la segunda computación no depende del valor producido por la primera. No significa que se ejecute en paralelo ni que sus efectos puedan reordenarse.

Scala 3

```
def product[A, B](
  fa: Either[Error, A], fb: Either[Error, B]
): Either[Error, (A, B)] =
  fa.flatMap(a => fb.map(b => (a, b)))

val precio = importe("12000")
val cantidad = importe("2")
val subtotal = product(precio, cantidad)
  .map { case (p, q) => p * q }
// Right(24000)
```

Este product se implementa mediante las operaciones de Either, pero se usa para combinar resultados cuya estructura ya conocemos. La cantidad se representa aquí como entero no negativo por simplicidad; el contrato podría exigir que fuera estrictamente positiva. El importe del subtotal conserva la unidad centavos.

La función tiene argumentos ordinarios estrictos. En `product(importe("??"), importe("-1"))` se evalúan ambas llamadas antes de entrar a product. El resultado conserva el primer Left, pero esto no demuestra que la segunda validación no se ejecutó. Para acumular ambos errores habría que elegir otra representación y otra combinación, por ejemplo una validación con colección de errores y una operación asociativa para unirlos. Applicative no impone una política universal de acumulación. [29]

Las leyes de Applicative incluyen identidad, composición, homomorfismo e intercambio. Se desarrollan en el anexo del capítulo 15. Sirven para que introducir funciones puras y agrupar aplicaciones tenga un significado coherente. La capacidad de combinar computaciones precede a cualquier decisión de hilos, tareas o paralelismo.

Monad elige el siguiente paso usando el valor anterior

$$\text{flatMap} : F[A] \times (A \to F[B]) \to F[B]$$

Después de convertir un texto a número, la validación de no negatividad necesita ese número. El siguiente paso tiene tipo `BigInt => Either[Error, BigInt]`. Si usamos map, obtenemos un resultado dentro de otro resultado. `flatMap` combina la transformación y la composición del contexto, evitando esa anidación adicional. [30, 31, 32]

Scala 3

```
val anidado = parsear("5000").map(noNegativo)
// Right(Right(5000))
// Tipo Either[Error, Either[Error, BigInt]]

val directo = parsear("5000").flatMap(noNegativo)
// Right(5000)

val negativo = parsear("-200").flatMap(noNegativo)
// Left(Error.Negativo)

val ilegible = parsear("??").flatMap(noNegativo)
// Left(Error.Formato) y noNegativo no se llama
```

En el primer caso, map no está equivocado: hace exactamente lo que dice su tipo, envolver el resultado de la función. Es nuestra necesidad de composición la que requiere flatMap. La mónada de Either representa una secuencia que propaga el primer error. La mónada de List representa una composición de alternativas; no convierte las listas en operaciones de E/S.

Las tres leyes clásicas se leen como reglas de composición. Aquí `m` es un valor de F[A] y f y g devuelven valores en F. Debe usarse una noción de igualdad adecuada al contexto, no necesariamente comparar objetos por referencia.

$$\text{pure}(a).\text{flatMap}(f) = f(a)$$

$$m.\text{flatMap}(\text{pure}) = m$$

$$m.\text{flatMap}(f).\text{flatMap}(g) = m.\text{flatMap}(x \Longrightarrow f(x).\text{flatMap}(g))$$

La identidad izquierda dice que introducir a y continuar con f equivale a llamar f(a). La identidad derecha dice que continuar sin cambiar el valor no agrega comportamiento. La asociatividad dice que podemos reagrupar la composición conservando el orden de las dependencias. No permite intercambiar f y g.

Para Either, si m es Left(e), ambos lados de la asociatividad conservan e. Si es Right(a), ambos ejecutan f(a); si esta falla, conservan su error, y si entrega b, continúan con g(b). Esta separación por constructores explica la ley para la implementación usual. Nuestros ensayos de código ilustran estos casos, pero una colección finita de asserts no demuestra una ley para todas las funciones posibles.

Elegir la operación con una pregunta

Necesidad	Forma de la función	Operación	Ejemplo del reporte
Cambiar un valor	A → B	map	Sumar n al total ya validado
Combinar resultados conocidos	Dos valores en F y una combinación	product o map2	Validar precio y cantidad por separado
Continuar con una computación dependiente	A → F[B]	flatMap	Interpretar texto y luego comprobar el número

Un Monad coherente proporciona también capacidades de Applicative y Functor. Esto no obliga a usar la operación más potente en todas partes. Expresar una combinación independiente como tal conserva información sobre la estructura del cálculo. Por otro lado, escribir flatMap por costumbre no hace una solución más funcional. La elección debe describir la dependencia real del problema.

Un producto no es una mónada; una mónada no es un contenedor físico; pure no significa “ejecutar”; map no extrae un valor para abandonar el contexto. Estas correcciones terminológicas evitan analogías que parecen fáciles al principio y después bloquean la comprensión.

9 Recorridos acumuladores y monoides

Qué hace un fold

Un fold resume una estructura usando una regla de combinación y un punto de partida. En una lista, `foldLeft(z)(paso)` procesa elementos de izquierda a derecha y entrega al paso el acumulador actual y el siguiente elemento. Para [a, b, c], la asociación conceptual es `paso(paso(paso(z, a), b), c)`. En el reporte el acumulador tiene tipo Either, no BigInt. [5, 7, 22]

Para A, C y D corregidos, las transiciones son Right(0), Right(12000), Right(17000), Right(25000). Para los datos originales, la segunda transición válida se reemplaza por un Left y el resto de pasos conserva ese Left. El mismo recorrido sirve para otro resumen si cambiamos el tipo del acumulador y su función de paso.

Un fold no exige que su operación sea asociativa. La resta puede usarse en un fold con orden fijo. La asociatividad se vuelve necesaria cuando queremos cambiar la agrupación manteniendo el resultado. Esta diferencia separa una herramienta de recorrido de una propiedad algebraica.

Un monoide permite reagrupar resúmenes

Un monoide consiste en un conjunto o tipo M, una operación cerrada de combinación y un elemento identidad. La operación debe ser asociativa y la identidad funcionar a ambos lados. Escribir “los números forman un monoide” queda incompleto: hay que especificar la operación y su identidad. Los enteros con suma y cero forman uno; con multiplicación y uno, otro. [47]

$$\oplus : M \times M \rightarrow M$$

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$

$$e \oplus a = a = a \oplus e$$

Para un reporte de importes válidos, M puede ser BigInt, la combinación la suma exacta y la identidad cero. Podemos sumar cada partición de pedidos y combinar sus subtotales. La identidad explica por qué una partición vacía no altera el total. La asociatividad explica por qué podemos cambiar los paréntesis.

Reagrupar no es reordenar. La concatenación de textos es asociativa y tiene texto vacío como identidad, pero “ab” no es “ba”. Para permutar arbitrariamente los bloques hace falta además conmutatividad u otra justificación específica. Con errores que preservan el primero, el orden de las hojas también puede formar parte del contrato.

El uso de enteros matemáticos evita el problema de redondeo de punto flotante, pero las representaciones concretas importan. Por ejemplo, en aritmética de doble precisión, `(1e16 + -1e16) + 1.0` puede producir 1, mientras `1e16 + (-1e16 + 1.0)` produce 0. La suma flotante no es asociativa de manera exacta. La resta tampoco lo es: (10 − 3) − 2 = 5, pero 10 − (3 − 2) = 9.

La posibilidad algebraica de combinar particiones no garantiza una aceleración paralela. Hay costos de dividir, transferir y combinar datos; puede haber dependencias de errores y limitaciones de memoria. Primero justificamos la transformación, después elegimos una ejecución y la medimos.

Resumir más de una propiedad

Podemos resumir un grupo de pedidos válidos con un par `(suma, cantidad)`. La combinación suma cada componente y la identidad es `(0, 0)`. Así se calcula el promedio al final dividiendo suma entre cantidad, si la cantidad no es cero. Combinar directamente promedios sin ponderación sería incorrecto cuando los grupos tienen tamaños diferentes.

Ejemplo propio: el grupo [100, 300] tiene promedio 200 y el grupo [1000] tiene promedio 1000. Promediar esos dos promedios da 600; el promedio correcto de los tres datos es 1400/3. El resumen `(suma, cantidad)` conserva la información suficiente para una combinación correcta. Este caso muestra cómo el álgebra orienta el diseño de los datos intermedios.

10 Tipos algebraicos y lotes anidados

Productos para datos que van juntos

Un tipo producto reúne componentes. Un Pedido necesita identificador, estado, importe y moneda al mismo tiempo. Un par `(A, B)` representa un A junto con un B. Cuando A y B tienen conjuntos finitos de valores, el número de combinaciones del producto es el producto de sus cardinalidades. Dos booleanos tienen cuatro pares posibles. [23, 43]

La existencia de campos no garantiza todas las reglas del negocio. Nuestro Pedido permite moneda desconocida o importe inválido porque representa una entrada todavía sin validar. Podríamos introducir un tipo ImporteValido con constructor controlado para impedir ciertos estados después de la validación. El diseño debe distinguir un registro de entrada de un dato al que ya atribuimos invariantes.

Sumas para alternativas identificables

Un tipo suma representa una alternativa entre constructores etiquetados. Estado es Pagado o Pendiente. Either es Left con un error o Right con un resultado. La etiqueta permite distinguir alternativas aunque sus campos tengan el mismo tipo. Para conjuntos finitos, las cardinalidades de alternativas disjuntas se suman.

La palabra “algebraico” se refiere a esta forma de combinar tipos mediante productos y sumas, entre otras construcciones. No significa que los campos deban contener números. El constructor de un producto reúne valores; los constructores de una suma indican qué caso existe. Una definición puede combinar ambos: EnPedido es una alternativa del tipo Error y dentro reúne id y causa.

Un tipo recursivo sigue la forma del dominio

El reporte crece: ahora recibimos lotes que pueden estar vacíos, contener un pedido o unir dos lotes. Una lista plana pierde la agrupación original; una estructura recursiva puede conservarla. La definición vuelve a mencionar Lote dentro de sus constructores, lo que permite anidar sin fijar una profundidad máxima en el tipo.

Scala 3

```
enum Lote:
  case Vacio
  case Hoja(p: Pedido)
  case Unir(izq: Lote, der: Lote)
```

$$\text{Lote} \cong 1 + \text{Pedido} + \text{Lote} \times \text{Lote}$$

Esta ecuación es una lectura estructural, no una ecuación para despejar un número. El 1 representa el único caso vacío sin campos; Pedido representa el contenido de una hoja; el producto representa los dos sublotes de una unión. El signo de suma indica elección entre constructores.

El lote `Unir(Unir(Hoja(A), Hoja(C)), Hoja(D))` contiene los mismos pedidos seleccionables del ejemplo, pero conserva una agrupación. Podemos imaginar la unión izquierda como una sucursal y la unión final como el conjunto de sucursales. El significado empresarial no lo impone el constructor; se añade en el modelo del dominio.

El programa se deriva de los constructores

Scala 3

```
def total(lote: Lote): Either[Error, BigInt] =
  lote match
    case Lote.Vacio => Right(BigInt(0))
    case Lote.Hoja(p) => reporte(List(p))
    case Lote.Unir(i, d) =>
      total(i).flatMap(x => total(d).map(y => x + y))
```

El pattern matching abre una rama por constructor. Vacío entrega la identidad de la suma. Hoja reutiliza el contrato del reporte de un pedido: un pendiente o una moneda distinta aporta cero, incluso si el importe textual no es válido. Unir obtiene los resultados de los sublotes y los combina. Esta correspondencia entre dato y programa es recursión estructural. [23, 43]

Con C corregido, las hojas A y C devuelven Right(12000) y Right(5000); su unión devuelve Right(17000). D devuelve Right(8000); la raíz devuelve Right(25000). El cálculo desciende para resolver subproblemas y asciende al construir sus resultados. Cada resultado responde a una llamada pendiente y se combina en el constructor que la originó.

Con C inválido, su hoja devuelve Left(EnPedido(C, Formato)); la unión izquierda devuelve ese Left. En la raíz, el cuerpo de flatMap no se ejecuta, por lo que `total(d)` no se llama. Aquí sí puede omitirse un subárbol entero. Esta diferencia respecto del fold de la lista depende de dónde está escrita la llamada, no de una propiedad mágica de la palabra Monad.

El patrón `case Lote.Unir(i, d)` introduce nombres para los campos de ese constructor; no vuelve a validar sus tipos en tiempo de ejecución como si no se conocieran. El compilador puede advertir sobre casos no cubiertos en sumas cerradas. Cubrir todos los constructores evita una familia de omisiones, pero no prueba que la regla numérica de cada rama sea correcta.

11 Terminación y control de la pila

Por qué termina la recursión estructural

Decir “el programa es recursivo” no demuestra que termine. Necesitamos identificar una medida que disminuya y no pueda descender indefinidamente. Para un lote finito y acíclico, definimos su tamaño contando constructores. Los sublotes de una unión tienen menos constructores que la unión completa.

$$\text{tamaño}(\text{Vacio}) = 1$$

$$\text{tamaño}(\text{Hoja}(p)) = 1$$

$$\text{tamaño}(\text{Unir}(i, d)) = 1 + \text{tamaño}(i) + \text{tamaño}(d)$$

Cada llamada recursiva de total recibe i o d, por tanto disminuye la medida. Las ramas base no llaman recursivamente a total. Si el reporte de cada hoja termina, el recorrido termina para cualquier lote finito de esta clase. La demostración depende de esas hipótesis: no se extiende automáticamente a una estructura infinita, una referencia cíclica o una validación que nunca devuelve.

Terminar matemáticamente y terminar con los recursos de una máquina son cuestiones diferentes. Un árbol muy profundo puede agotar la pila aunque tenga un número finito de nodos. En `total`, el resultado de la izquierda todavía debe usarse para procesar la derecha y sumar. Queda trabajo pendiente, por lo que no es una recursión de cola. En el recorrido directo, la profundidad de llamadas es O(h), donde h es la altura.

Convertir llamadas pendientes en datos explícitos

Una alternativa mantiene una lista de lotes pendientes y un acumulador. Al encontrar una unión, agrega primero el hijo izquierdo y después el derecho a la lista de trabajo. Así conserva el orden de procesamiento de izquierda a derecha sin dejar una suma pendiente en cada llamada recursiva.

Scala 3

```
def totalSeguro(lote: Lote): Either[Error, BigInt] =
  @annotation.tailrec
  def loop(
    pendientes: List[Lote], acumulado: BigInt
  ): Either[Error, BigInt] = pendientes match
    case Nil => Right(acumulado)
    case Lote.Vacio :: resto =>
      loop(resto, acumulado)
    case Lote.Hoja(p) :: resto =>
      reporte(List(p)) match
        case Left(e) => Left(e)
        case Right(n) => loop(resto, acumulado + n)
    case Lote.Unir(i, d) :: resto =>
      loop(i :: d :: resto, acumulado)
  loop(List(lote), BigInt(0))
```

La anotación `@tailrec` exige que el compilador pueda optimizar las llamadas recursivas de esa función como recursión de cola; si la condición no se cumple, informa un error. No convierte por sí sola cualquier algoritmo recursivo en uno de cola. [44]

La pila de llamadas del bucle puede permanecer constante, pero la lista explícita de pendientes puede ocupar O(h) espacio. Cambiamos dónde se representa el trabajo pendiente. No debemos presentar esta versión como de memoria total constante. Además del espacio de recorrido, existen el árbol original, el acumulador BigInt y los objetos temporales.

Para razonar sobre su corrección, usamos un invariante: el acumulador contiene la suma válida de las hojas ya procesadas, y la lista pendiente conserva el orden de las hojas todavía no procesadas. Vacío no cambia el total; una hoja válida agrega su aporte; una unión se reemplaza por sus hijos en el mismo orden. Si aparece un error, se devuelve antes de procesar los pendientes restantes. Si la lista queda vacía, el acumulador es el total completo.

La versión directa y la versión con pendientes coinciden en valor y primer error para lotes finitos bajo el contrato. Pueden diferir en uso de pila y asignaciones. En el material ejecutable del equipo se comprobó además un lote profundo de 20000 hojas con la versión de cola. Esa prueba es evidencia de un caso grande, no una demostración para todos los tamaños ni un estudio de rendimiento.

Un argumento de terminación para el bucle

La cantidad de elementos de la lista pendiente no disminuye en cada paso: al abrir una unión, un elemento se convierte en dos. Por eso no sirve como medida simple de terminación. Sí sirve la suma de los tamaños de todos los lotes pendientes. Al reemplazar Unir(i, d) por i y d, esa suma disminuye en uno; al retirar Vacio o Hoja, también disminuye. Es un número natural que se reduce en cada iteración mientras no haya terminado con un error.

Esta observación es útil para defender el algoritmo: una medida correcta debe seguir disminuyendo en todas las ramas, no solo en los ejemplos pequeños.

12 Evaluación estricta no estricta y perezosa

Cuándo se necesita un argumento

En evaluación estricta, se evalúan los argumentos antes de entrar al cuerpo ordinario de la función. Scala y OCaml siguen este comportamiento de manera habitual, con mecanismos adicionales para diferir trabajo. En una semántica no estricta puede obtenerse un resultado sin evaluar un argumento que no es necesario. Haskell tiene semántica no estricta. [2, 4, 16, 20]

Considera una función que ignora su segundo argumento y devuelve 1. Si el segundo argumento es una computación que no termina, una evaluación estricta no alcanza el cuerpo; una evaluación no estricta puede devolver 1. Esta diferencia se refiere a la necesidad del argumento, no a que la función sea “más rápida” en todos los casos.

$$\text{constante} = \lambda x.\ \lambda y.\ x$$

$$\text{constante}\ 1\ \Omega \rightarrow 1 \qquad \text{si el argumento } \Omega \text{ no se demanda}$$

Ω representa aquí una expresión divergente, es decir, una que no termina. La notación resume la diferencia entre estrategias; no es una llamada que debamos ejecutar durante la exposición. Si la función necesitara efectivamente el segundo argumento, diferirlo no evitaría la divergencia.

La evaluación por nombre difiere una expresión y puede repetirla cada vez que se usa. La evaluación por necesidad comparte el resultado de la evaluación cuando este se exige, evitando repetir el mismo trabajo diferido. “No estricto” describe una propiedad semántica; “por necesidad” describe una estrategia habitual para implementarla. Las palabras están relacionadas, pero no son intercambiables sin explicación.

Diferir no implica memorizar

Scala 3

```
def dosUsos(x: => Int): Int = x + x

// Si se fuerza x dos veces, su expresión se evalúa dos veces.
def dosUsosCompartidos(x: => Int): Int =
  lazy val guardado = x
  guardado + guardado
```

El parámetro por nombre de Scala no memoriza por sí solo: su expresión se evalúa en cada uso. El `lazy val` introduce una inicialización diferida compartida en este ejemplo. Con expresiones puras puede cambiar el costo sin cambiar el resultado; con efectos también puede cambiar cuántas veces ocurre una acción. [20]

Diferir trabajo genera suspensiones, a veces llamadas thunks, que conservan la expresión pendiente y lo necesario de su entorno. Mientras sean alcanzables, pueden retener memoria. Una cadena de sumas sin evaluar o una referencia al comienzo de una secuencia memoizada puede mantener más datos de lo esperado. La pereza cambia qué se evalúa y cuándo; también cambia qué permanece vivo.

Una secuencia infinita con demanda finita

Scala 3

```
val resultado = LazyList.from(1)
  .filter(_ % 2 == 0)
  .map(_ * 2)
  .take(3)
  .toList
// List(4, 8, 12)
```

La demanda final pide tres resultados. Se inspecciona 1 y se rechaza; 2 pasa el filtro y se transforma en 4. Se rechaza 3; 4 se transforma en 8. Se rechaza 5; 6 se transforma en 12. Ya hay tres resultados, de modo que esta cadena no necesita inspeccionar 7 para construir esa lista final. La fuente es potencialmente infinita; el prefijo demandado es finito. [48]

Entrada inspeccionada	Decisión del filtro	Salida transformada	Resultados obtenidos
1	Rechazar	Ninguna	0
2	Conservar	4	1
3	Rechazar	Ninguna	1
4	Conservar	8	2
5	Rechazar	Ninguna	2
6	Conservar	12	3 y detener la demanda

Este mecanismo permite separar productores y consumidores sin materializar una secuencia infinita. No implica memoria constante en cualquier uso: conservar referencias a partes anteriores de una LazyList puede mantener valores ya calculados. Tampoco todo pipeline perezoso es productivo.

Productividad y acumuladores diferidos

Considera `LazyList.from(1).filter(_ => false).take(1).toList`. Se pide un solo resultado, pero nunca hay un elemento que supere el filtro. El cálculo sigue buscando. Una demanda finita no garantiza un resultado. La productividad requiere poder obtener cada porción necesaria en tiempo finito bajo las condiciones del productor y sus transformaciones.

En Haskell, `foldl` asocia las combinaciones por la izquierda, pero no fuerza por ello cada acumulador durante el recorrido. Puede construir una cadena de trabajo diferido. `foldl'` fuerza el acumulador a forma normal débil de cabeza en cada paso; eso no equivale a evaluar profundamente cualquier estructura contenida en él. La asociación de un fold y la estrictitud de sus evaluaciones son propiedades distintas. [27]

Para un acumulador numérico simple, esta distinción suele ser importante al evitar cadenas de sumas pendientes. Para un acumulador compuesto, puede hacer falta analizar también la evaluación de sus campos. La elección correcta depende del dato y del consumo, no de una regla universal que haga a la pereza mejor o peor.

13 Lenguajes y aplicaciones reales

La familia funcional admite decisiones diferentes

La siguiente comparación sirve para reconocer una familia de ideas, no para asignar una etiqueta excluyente a cada lenguaje. Los sistemas de tipos, la evaluación y los modelos de concurrencia pertenecen a dimensiones diferentes. Un lenguaje puede tener tipos estáticos y efectos, o funciones de orden superior y tipado dinámico.

Lenguaje	Rasgos pertinentes	Qué no hay que confundir
Haskell	Tipado estático y semántica no estricta con separación explícita de E/S	No estricto no significa que nada se evalúe
OCaml	Tipado estático con inferencia y evaluación estricta habitual	Admite mutabilidad y objetos
Scheme	Tradición Lisp con alcance léxico y funciones de primera clase	Alcance léxico no significa tipado estático
Common Lisp	Lenguaje multiparadigma con funciones y macros	Su estilo funcional no exige pureza global
Scala	Integra objetos funciones y tipos algebraicos en Scala 3	Su nombre no garantiza pureza de cada método
F#	Enfoque funcional con tipado estático en el ecosistema .NET	Puede integrar código orientado a objetos e imperativo
Clojure	Tradición Lisp y estructuras persistentes en su diseño	Persistencia de datos no es almacenamiento en disco
Erlang	Funciones y procesos con paso de mensajes en BEAM	Concurrencia y tolerancia a fallos dependen también del runtime y OTP
Elixir	Funciones datos inmutables y procesos sobre BEAM	Las capacidades de tipos evolucionan y no deben reducirse a una etiqueta fija

Las características de esta tabla se apoyan en los reportes y documentos de cada proyecto. Para Scheme y Common Lisp importan los estándares; para las capacidades actuales de herramientas conviene consultar sus documentos de versión. En Elixir, la evolución del sistema de tipos hace especialmente inadecuado afirmar sin fecha que “no tiene ninguna comprobación de tipos”. [2, 13, 17, 23, 37, 38, 39, 40, 41, 42]

El enfoque es útil en transformaciones de datos, reglas del negocio, procesamiento simbólico y descripción de flujos. Estas son posibilidades de diseño, no pruebas de que todo sistema de esos dominios deba usar el mismo lenguaje. Las bibliotecas, el equipo, las restricciones operativas y la interoperabilidad también cuentan.

El mismo recorrido expresado en OCaml

OCaml

```
type lote =
  | Vacio
  | Hoja of pedido
  | Unir of lote * lote

let rec total = function
  | Vacio -> Ok 0
  | Hoja p -> reporte_hoja p
  | Unir (i, d) ->
      Result.bind (total i) (fun x ->
        Result.map (fun y -> x + y) (total d))
```

Este fragmento usa las definiciones de pedido y reporte_hoja del archivo complementario Lotes.ml. La función reporte_hoja filtra el estado y la moneda, convierte el texto y comprueba la no negatividad antes de devolver Ok o Error. `Ok` y `Error` cumplen un papel semejante a Right y Left; `fun x -> ...` introduce una función anónima. La llamada al sublote derecho queda dentro de la continuación del resultado izquierdo. Cambia la sintaxis, pero se conserva la estructura del razonamiento. [4, 5, 43]

Hay una diferencia material: el ejemplo OCaml usa `int`, cuyo rango es limitado, y el de Scala usa BigInt. La equivalencia numérica se entiende para valores que caben y no desbordan en la versión OCaml. Para exigir enteros arbitrarios también allí habría que elegir una biblioteca adecuada. Este fragmento ilustra un recorrido de datos anidados y errores; no es un “hola mundo” ni demuestra por sí solo igualdad de rendimiento entre lenguajes.

Sigma y Haxl como caso documentado

En una publicación del 26 de junio de 2015, el equipo de Facebook describió la migración de reglas de Sigma a Haskell y su uso de Haxl para organizar consultas de datos. El ejemplo es relevante porque une reglas, tipos y acceso a datos dentro de un sistema de producción. Debe presentarse como un caso histórico documentado de esa fecha. [11]

La separación conceptual es útil para nuestro reporte: una regla necesita datos, pero no debería duplicar en cada lugar la lógica de agrupar consultas o administrar cachés. Cuando dos consultas pueden describirse antes de conocer sus resultados, una estructura aplicativa permite exponer esa independencia a una biblioteca. Cuando la siguiente consulta se decide usando un valor recibido, hay una dependencia monádica.

La biblioteca y su runtime son quienes implementan las políticas de ejecución. La pureza de una función no crea por sí sola una red de consultas eficiente. Esta distinción permite explicar el valor de las abstracciones sin atribuirles propiedades operativas que dependen del sistema concreto.

14 Ventajas límites y decisiones de ingeniería

Qué ganamos y qué debemos resolver

Decisión	Ventaja posible	Costo o condición
Dependencias explícitas	Pruebas reproducibles del núcleo y funciones reutilizables	Hay que diseñar los límites y probar también la integración
Datos inmutables	Versiones compartidas sin cambios accidentales sobre esos datos	Nuevos nodos y versiones retenidas consumen memoria
Tipos suma para errores	Casos esperados visibles y combinables	El tipo debe modelar causas y políticas adecuadas
Composición con leyes	Transformaciones justificables y abstracciones reutilizables	Requiere conocer hipótesis y respetar las instancias
Evaluación diferida	Consumir solo lo necesario en ciertos recorridos	Puede retener entornos o buscar un resultado que nunca llega
Estructuras recursivas	Modelo cercano a la forma del dominio	Profundidad y estrategia de recorrido afectan la pila

Las ventajas son concretas cuando podemos relacionarlas con un problema. En el reporte, recibir una lista evita leer archivos para cada prueba; Either conserva la causa del fallo; la estructura Lote permite tratar cada sucursal como un subproblema. No basta enumerar “modularidad, seguridad y paralelismo” sin explicar qué mecanismo proporciona cada una.

La programación funcional puede exigir aprender abstracciones, representaciones y modelos de evaluación poco familiares. Una solución demasiado genérica también puede ser difícil de leer. Conviene introducir la abstracción después de reconocer una repetición útil y conservar nombres del dominio. En la exposición, la pregunta “¿qué necesidad resuelve?” debe poder responderse para Functor igual que para la persistencia.

Costos que sí podemos justificar

Operación o recorrido	Modelo de costo	Límite de la afirmación
Anteponer en lista enlazada inmutable	O(1) nodos y pasos estructurales	No cuenta el costo de construir el elemento
Acceso por índice en esa lista	O(n) en el peor caso	Otras estructuras pueden ofrecer otro acceso
Agregar al final de una lista simple	O(n) con reconstrucción del prefijo	No confundir con anteponer
Insertar en el árbol mostrado	O(h) tiempo y nodos nuevos	O(log n) requiere altura logarítmica

Operación o recorrido	Modelo de costo	Límite de la afirmación
Recorrer lotes sin errores	O(N) visitas a constructores	Se agregan costos de interpretar textos y sumar enteros
Recursión directa sobre lotes	O(h) profundidad de llamadas	Puede agotar la pila en árboles profundos
Bucle de cola con pendientes	Pila constante para el bucle y O(h) pendientes	No es memoria total constante

N es el número de constructores y h la altura. El reporte de lista hace primero una selección que visita todos los pedidos y después un fold sobre los seleccionados. Aun cuando aparezca un error temprano, esta versión ya filtró toda la entrada y el fold recorre la lista seleccionada. En el árbol, el error izquierdo puede evitar visitas al subárbol derecho.

La aritmética BigInt y la conversión desde texto tienen costos que dependen del número de dígitos. Por eso O(N) visitas no es una afirmación completa de tiempo si los importes pueden crecer sin límite. El modelo debe indicar qué toma como operación básica. [22, 25]

Las pruebas unitarias verifican comportamientos seleccionados; las pruebas basadas en propiedades exploran familias de entradas; una demostración establece una afirmación bajo hipótesis; un benchmark mide un programa en condiciones determinadas. Son herramientas complementarias. Una medición rápida no demuestra corrección y una prueba de corrección no establece latencia real.

Qué defender sobre el caso completo

Podemos justificar qué registros participan, qué error se conserva, por qué la suma de importes válidos admite reagrupación, cómo se conserva una versión de un árbol y por qué termina un recorrido de un lote finito. Podemos comparar la pila de dos implementaciones y explicar qué se conserva al pasar de Scala a OCaml.

Quedan decisiones de producto si el sistema crece: acumular todos los errores o conservar uno, limitar profundidad, validar antes de filtrar, usar moneda como tipo cerrado o integrar persistencia real. El paradigma ayuda a expresar y analizar estas decisiones; el contrato del problema determina cuál es correcta.

15 Leyes para consultar y explicar

Recordatorio de los tipos

En este capítulo F representa una construcción con una instancia válida de las operaciones correspondientes. Las funciones se interpretan en un marco donde las igualdades son válidas y los efectos se modelan por F. Las ecuaciones se usan para razonar; no son órdenes de cambiar libremente el orden de ejecución.

Para Functor tenemos map. Applicative agrega pure y ap; escribiremos u ⊛ v para aplicar una función en F a un argumento en F. Monad agrega flatMap de manera coherente con las capacidades anteriores. El uso de ⊛ evita confundir una aplicación contextual con la aplicación ordinaria de funciones.

Las cuatro leyes de Applicative

$$\text{pure}(\text{id}) \circledast v = v$$

Identidad. Introducir la función identidad y aplicarla dentro del contexto no debe cambiar v. En Either, si v es Right(a), se obtiene Right(a); si es Left(e), se conserva e.

$$\text{pure}(\circ) \circledast u \circledast v \circledast w = u \circledast (v \circledast w)$$

Composición. Supón u de tipo F[B → C], v de tipo F[A → B] y w de tipo F[A]. Componer las funciones dentro de F y aplicarlas a w debe coincidir con aplicar primero v a w y luego u. En el lado izquierdo la aplicación ⊛ se asocia a la izquierda. Esta ley conserva el orden funcional v antes de u; no declara que dos efectos sean intercambiables.

$$\text{pure}(f) \circledast \text{pure}(x) = \text{pure}(f(x))$$

Homomorfismo. Si tanto la función como el argumento son valores introducidos con pure, aplicar dentro del contexto equivale a calcular ordinariamente e introducir el resultado. La estructura no debe inventar un comportamiento extra en esta situación.

$$u \circledast \text{pure}(y) = \text{pure}(\lambda f.\ f(y)) \circledast u$$

Intercambio. Aplicar la función contextual u a un argumento puro y puede expresarse también introduciendo la función que aplica su entrada a y. Los tipos ayudan a leerla: u tiene tipo F[A → B], y tiene tipo A y ambos lados producen F[B]. [29, 33, 46]

Coherencia entre las operaciones

$$\text{map}(f, \text{fa}) = \text{pure}(f) \circledast \text{fa}$$

$$\text{map}(f, \text{fa}) = \text{fa}.\text{flatMap}(a \Longrightarrow \text{pure}(f(a)))$$

La primera ecuación relaciona Functor con Applicative y la segunda con Monad. En una implementación coherente, cambiar la forma de expresar la transformación no debe cambiar su significado. No son tres permisos para implementar map con políticas distintas dentro de una misma instancia.

Las tres leyes de Monad del capítulo 8 corresponden a identidades izquierda y derecha y asociatividad. Una buena defensa oral debe explicar al menos una de ellas con un éxito y un fallo concretos. Memorizar los símbolos sin identificar los tipos impide reconocer cuándo una expresión ni siquiera está bien formada.

16 Ejercicios para desarrollar dominio

Resuelve primero sin consultar las respuestas. En cada ejercicio escribe el resultado y la razón. Si detectas que falta una hipótesis, indícala; reconocer una condición necesaria es parte de la solución. Los ejercicios 1 a 8 comprueban fundamentos, 9 a 16 conectan abstracciones con el caso y 17 a 24 exigen analizar variantes.

Fundamentos y sustitución

1. Dependencias. Una función recibe una lista de pedidos, consulta una variable global mutable de moneda y devuelve un total sin escribir nada. ¿Es suficiente que no escriba para considerarla pura? Propón una firma que haga visible esa dependencia.

2. Alcance. En `λx. λz. x + y + z`, identifica variables ligadas y libres. ¿Qué debe guardar una clausura que se construye en un entorno donde y vale 7?

3. Reducción. Usa `dosVeces = λf. λx. f(f x)` para reducir `dosVeces (λn. 3n) 2`. Escribe cada sustitución y no solo el resultado.

4. Captura. Reduce `(λx. λy. x y) y` sin capturar la variable libre. Explica por qué la sustitución literal da un resultado distinto.

5. Composición. Sean g(n) = n + 4 y f(n) = 3n. Calcula `xs.map(g).map(f)` para xs = [1, 2]. Expresa la versión de un solo map y verifica el orden de composición.

6. Prueba. Reconstruye la prueba de composición de map sin mirar el capítulo 5. Señala en qué línea se usa la hipótesis inductiva. ¿Por qué ensayar cien listas no es la misma prueba?

Estructuras y resultados

7. Persistencia. En el árbol con raíz 8, hijos 3 y 10, y con hijos 1 y 6 bajo 3, inserta 5. Enumera los nodos nuevos y los compartidos. ¿Puede conservarse la raíz original?

8. Complejidad. ¿Qué ocurre con la altura al insertar 1, 2, 3, 4 y 5 en el árbol sin balanceo? ¿Qué afirmación de costo sería incorrecta?

9. Tipos. Fija E = Error en Either[E, A]. Escribe F[A] para A = BigInt y A = String. ¿Es F un pedido o una construcción de tipos?

10. Operación. Elige map, product o flatMap para cada caso: convertir un total válido a texto; combinar precio y cantidad validados independientemente; leer un identificador y elegir la consulta que depende de ese identificador. Justifica cada elección por su tipo.

11. Anidación. Compara los tipos y resultados de `parsear("-3").map(noNegativo)` y `parsear("-3").flatMap(noNegativo)`.

12. Errores independientes. Se llama `product(importe("??"), importe("-1"))` con la definición del capítulo 8. ¿Qué resultado devuelve? ¿Se evaluó la segunda llamada? ¿Qué habría que cambiar para conservar ambos errores?

Acumulación y datos recursivos

13. Contrato. Agrega al reporte un pedido pendiente con importe "??" y otro pagado en USD con importe "??". ¿Cambia el resultado? ¿Cambiaría si el contrato exigiera auditar todos los registros?

14. Álgebra. ¿Forman un monoide los textos con concatenación y texto vacío? ¿Se pueden reordenar libremente? Da un ejemplo que distinga asociatividad de conmutatividad.

15. Resúmenes. Un lote contiene [200, 400] y otro [1200]. Diseña un resumen combinable para obtener el promedio de los tres valores. Compara con promediar directamente los dos promedios.

16. Lotes. Evalúa `Unir(Hoja(A), Unir(Hoja(C), Hoja(D)))` con los pedidos originales. Explica qué llamadas se ejecutan y cuáles se omiten. Después corrige C y vuelve a evaluar.

17. Terminación. ¿Por qué “la lista de pendientes tiene cada vez menos elementos” no es un argumento válido para totalSeguro? Propón una medida que sí disminuya en todas las ramas.

18. Memoria. Un compañero afirma que totalSeguro usa memoria constante porque tiene @tailrec. Corrige la afirmación separando pila, lista pendiente y almacenamiento de los datos.

Evaluación y criterio técnico

19. Demanda. Traza `LazyList.from(1).filter(_ % 3 == 0).map(_ + 1).take(2).toList`. ¿Cuál es la última entrada que necesita inspeccionar para obtener la salida?

20. Productividad. ¿Por qué `LazyList.iterate(BigInt(1))(_ + 1).filter(_ < 0).take(1).toList` no termina aunque se pida un elemento?

21. Repetición. En `dosUsos(x: => Int) = x + x`, x se recibe como una expresión que imprime y devuelve 10. ¿Cuántas veces imprime? ¿Qué cambia si se usa el lazy val del capítulo 12?

22. Transformación. Se quiere paralelizar un reporte que conserva el primer error de izquierda a derecha. ¿Basta con decir que la suma es asociativa? Indica qué otro comportamiento debe preservarse.

23. Lenguajes. Refuta con una explicación precisa: “Scala no sirve para la parte práctica porque no es exclusivamente funcional” y “todos los lenguajes funcionales evalúan de forma perezosa”.

24. Diseño abierto. Propón una variante del reporte que acumule todos los errores y entregue también el total de los pedidos válidos. Define el tipo de resultado, la combinación de dos resúmenes y qué significa un lote vacío. Decide si el orden de errores forma parte del contrato.

17 Soluciones razonadas

Soluciones de los ejercicios 1 a 6

1. No basta con no escribir. Si la moneda global cambia entre llamadas, los mismos pedidos pueden producir resultados diferentes. Una firma más explícita es `reporte(ps: List[Pedido], moneda: String): Either[Error, BigInt]`. Aun con esa firma debemos revisar la implementación para descartar otras dependencias o efectos ocultos. Hacer visibles las entradas es una condición de diseño, no una prueba automática de pureza.

2. Las apariciones de x y z están ligadas por sus lambdas; y está libre en el término completo. Una clausura construida en un entorno con y = 7 debe conservar acceso al valor o a la ligadura correspondiente. Si esa ligadura permitiera mutación, habría que analizar su comportamiento. No hay una variable libre x simplemente porque aparezca lejos de su lambda: el alcance incluye el cuerpo de la función interior.

3. `((λf. λx. f(f x)) (λn. 3n)) 2` se reduce a `(λx. (λn. 3n)((λn. 3n) x)) 2`; luego a `(λn. 3n) ((λn. 3n) 2)`; la aplicación interior produce 6 y la exterior produce 18. Se aplica la transformación dos veces. El resultado no es 12, que correspondería a sumar dos copias de f(2), en lugar de componer las aplicaciones.

4. Renombramos primero la y ligada interior: `(λx. λz. x z) y`. Al sustituir x por la y libre queda `λz. y z`. Si escribiéramos `λy. y y`, capturaríamos la aparición del argumento y produciríamos una función diferente. Cambiar nombres ligados es válido cuando se conservan las relaciones de alcance.

5. El primer map da [5, 6] y el segundo [15, 18]. La forma de un recorrido es `xs.map(n => 3 * (n + 4))` o `xs.map(g.andThen(f))`. Invertir la composición produciría [7, 10], porque se aplicaría primero 3n y después se sumaría 4. Los tipos pueden coincidir y aun así el orden ser semánticamente incorrecto.

6. El caso base reduce ambos lados a []. El paso para `x :: xs` produce `f(g(x)) :: map(f, map(g, xs))`. La hipótesis inductiva permite reemplazar solo la cola por `map(f ∘ g, xs)`. La definición de composición y de map reconstruye el lado derecho. Cien listas son un conjunto finito de casos; la inducción cubre cualquier lista finita construida con los dos constructores.

Soluciones de los ejercicios 7 a 12

7. El camino es 8 → 3 → 6 → vacío izquierdo. Se crea 5 y nuevas versiones de 6, 3 y 8. Se reutilizan los subárboles de 1 y 10. La raíz antigua sigue apuntando a sus nodos originales y no contiene 5; la nueva raíz sí lo alcanza. No se mutan los antecesores antiguos.

8. Cada nuevo valor queda como hijo derecho del anterior; la altura crece linealmente. Sería incorrecto atribuir O(log n) garantizado a insertar en esta implementación. El costo correcto es O(h) y puede ser O(n). Para garantizar altura logarítmica hace falta mantener una condición de balance apropiada.

9. `F[BigInt]` es `Either[Error, BigInt]` y `F[String]` es `Either[Error, String]`. F representa el constructor de tipos con el parámetro Error fijado. No es un pedido, un error concreto ni la cantidad de elementos de una colección.

10. Convertir un éxito a texto usa map con una función `BigInt => String`. Combinar precio y cantidad usa product o map2 porque ambas computaciones están descritas antes de obtener sus resultados. Elegir una consulta a partir del identificador usa flatMap porque la continuación tiene tipo `Id => F[Datos]`. Usar una operación más general puede ser posible, pero ocultaría la independencia que sí conocemos en el segundo caso.

11. El map devuelve `Right(Left(Negativo))` de tipo `Either[Error, Either[Error, BigInt]]`. La conversión textual fue exitosa y su valor se transformó en un resultado fallido anidado. El flatMap devuelve `Left(Negativo)` de tipo `Either[Error, BigInt]`, porque compone las dos etapas dentro del mismo contexto.

12. Devuelve Left(Formato), según la política del product definido. Ambas llamadas a importe se evalúan antes de entrar a la función porque sus parámetros son estrictos. Para conservar ambos errores se necesita un resultado capaz de contener una colección de errores y una combinación que los reúna. No basta con renombrar product ni con agregar una segunda llamada a map.

Soluciones de los ejercicios 13 a 18

13. Ninguno de los dos pedidos nuevos pasa la selección, por lo que no cambia el resultado bajo el contrato actual. Si se exige auditar toda la entrada, la validación debe hacerse también para ellos y el comportamiento cambia. La ubicación del filtro es una decisión semántica, no solo una optimización.

14. Sí: concatenar textos produce otro texto, la concatenación es asociativa y el texto vacío es identidad a ambos lados. No es conmutativa: “ab” y “ba” difieren. Podemos pasar de (“a” ++ “b”) ++ “c” a “a” ++ (“b” ++ “c”), pero no permutar las piezas sin otra razón.

15. Los resúmenes son (600, 2) y (1200, 1). Su combinación es (1800, 3), así que el promedio es 600. Promediar 300 y 1200 da 750 y es incorrecto porque los grupos tienen tamaños distintos. La combinación correcta suma numeradores y cantidades, no promedios sin pesos.

16. Se obtiene el resultado de A y luego se entra en la unión derecha. C produce el error de formato; D se omite porque su llamada está dentro de una continuación que no se ejecuta. El resultado final es Left(EnPedido(C, Formato)). Con C corregido, la unión derecha produce 13000 y la raíz 25000. Reagrupar manteniendo el orden de hojas conserva el primer error; permutarlas puede cambiarlo.

17. Abrir una unión aumenta la cantidad de pendientes de uno a dos para ese elemento. La suma de los tamaños de los lotes pendientes sí disminuye: reemplazar un constructor Unir por sus hijos elimina exactamente un constructor del trabajo restante. Retirar Vacio o Hoja también la reduce. El argumento supone una estructura finita y que la validación de hojas termina.

18. El bucle optimizado mantiene constante su pila de llamadas, pero representa pendientes en una lista que puede ocupar O(h). El árbol de entrada, el acumulador y las asignaciones temporales requieren memoria adicional. En un árbol muy profundo no desaparece todo el trabajo pendiente: cambia de representación.

Soluciones de los ejercicios 19 a 24

19. Se rechazan 1 y 2; 3 produce 4. Se rechazan 4 y 5; 6 produce 7. La salida es [4, 7] y la última entrada necesaria es 6. La cadena no necesita inspeccionar 7 para satisfacer esta demanda. Cada etapa debe interpretarse en el orden en que está escrita.

20. La fuente usa BigInt para producir enteros positivos crecientes sin desbordamiento de un Int, y ninguno satisface el filtro. El consumidor no recibe ni siquiera su primer elemento y la búsqueda continúa indefinidamente. take limita la cantidad de salidas, no el número de entradas que el filtro debe inspeccionar para encontrarlas.

21. En la función con parámetro por nombre, la expresión imprime dos veces al usarse dos veces. Con `lazy val guardado = x`, se fuerza una vez durante la primera suma y se reutiliza el valor. El cambio afecta un efecto observable; por eso no podemos justificarlo solo por una ley pensada para expresiones puras.

22. No basta. Hay que preservar qué registros se validan, qué error se observa, qué orden de hojas define el primer error y qué efectos existirían en las validaciones. Se puede diseñar una combinación de resultados que conserve el primer error por posición, pero la ejecución paralela podría hacer trabajo que antes se omitía. La equivalencia de resultados y la equivalencia de efectos o recursos deben analizarse por separado.

23. Scala es multiparadigma y ofrece funciones como valores, tipos algebraicos, colecciones inmutables y operaciones de composición; permite practicar el enfoque aunque también tenga objetos y mutación. La evaluación perezosa tampoco define a toda la familia: OCaml es habitualmente estricto y Haskell tiene semántica no estricta. Hay varias dimensiones independientes.

24. Una respuesta posible es `Resumen(totalValido: BigInt, errores: List[Error])`. Una hoja válida seleccionada aporta su importe y lista vacía; una inválida seleccionada aporta cero y su error; una excluida aporta cero y ningún error. Combinar suma los totales y concatena los errores de izquierda a derecha. El lote vacío es (0, []). Esta combinación es asociativa bajo las operaciones elegidas y conserva el orden de errores. Cambia el contrato: ahora se entrega un total parcial acompañado de errores y se visitan ambas ramas para recopilarlos. La interfaz debe dejar claro que ese total puede ser incompleto, para que no se interprete como un reporte plenamente válido.

18 Preparación para la defensa oral

Una explicación sólida contiene la idea, un ejemplo y las condiciones bajo las cuales vale. Practica respuestas de un minuto y luego acepta una variante. Estas preguntas no buscan repetir una frase exacta; comprueban que puedes relacionar el significado con la implementación.

Preguntas que conviene poder responder

Por qué no basta con usar map para afirmar que un programa es funcional. map también puede recibir funciones que modifican estado o imprimen. La organización funcional debe analizar dependencias, datos y composición, no solamente nombres de métodos.

Qué aporta el tipo Either al reporte. Hace visible la posibilidad de fallo, preserva la causa y permite decidir cómo propagarla. No demuestra por sí mismo que toda entrada sea válida ni que el programa termine.

Dónde se ve realmente Functor. En una transformación que modifica los éxitos de Either y conserva los errores, de manera coherente con identidad y composición. F[A] se introduce a partir de ese comportamiento concreto, después de conocer List y Either.

Por qué Applicative no significa paralelo. Expresa que la estructura de las computaciones combinadas se conoce sin depender de sus resultados. La biblioteca y su instancia deciden cómo ejecutar y combinar; el tipo abstracto no crea hilos.

Cuándo hace falta flatMap. Cuando la continuación necesita el valor anterior y devuelve otro resultado contextual. En el ejemplo, noNegativo necesita el número que parsear produce. El map simple conserva un nivel adicional de Either.

Qué permite una ley algebraica. Justificar una igualdad o una reagrupación bajo hipótesis. No garantiza menor tiempo, ejecución paralela ni preservación de efectos que no se modelaron.

Por qué la persistencia no copia todo el árbol. Solo se reconstruye el camino que cambia y se comparten los subárboles intactos. El costo depende de la altura y del balance, y las versiones retenidas ocupan memoria.

Por qué una estructura finita puede provocar desbordamiento de pila. Terminar en un modelo sin límites no elimina las restricciones de la máquina. Un recorrido directo de profundidad grande deja muchas llamadas pendientes.

Qué cambia al pasar a recursión de cola. El trabajo pendiente se representa explícitamente o se reorganiza. El bucle puede usar pila constante, pero la lista de pendientes y los datos siguen consumiendo espacio.

Por qué la pereza no garantiza resultados. Un consumidor puede pedir un elemento que nunca aparece tras un filtro. La demanda finita necesita productividad del conjunto de transformaciones.

Por qué Scala es una elección válida. Permite implementar y observar las ideas funcionales en un lenguaje multiparadigma, lo que además muestra que el paradigma no pertenece exclusivamente a un lenguaje.

Qué evidencia aporta Sigma. Un caso industrial fechado de reglas y acceso a datos con Haskell y Haxl. No prueba que cualquier sistema de reglas deba usar Haskell ni certifica la arquitectura actual de esa plataforma.

Comprobar el propio dominio

Puedes considerar consolidado un concepto cuando eres capaz de definirlo sin metáforas vagas, calcular un caso nuevo, dar un contraejemplo a una generalización incorrecta y mencionar una condición de validez. Si solo recuerdas que “Monad encadena” pero no puedes escribir el tipo de la continuación, vuelve a los ejemplos de Either.

Como ensayo final, explica el reporte desde el contrato hasta los lotes en diez minutos. Otra persona debe interrumpir con tres cambios: conservar todos los errores, procesar un lote muy profundo y sustituir BigInt por Double. La respuesta debe distinguir cambios de significado, de recursos y de propiedades algebraicas. Ese ejercicio integra los temas mejor que recitar las diapositivas en su orden.

19 Glosario y recursos de estudio

Glosario de consulta

Abstracción. Descripción que conserva propiedades relevantes y omite detalles para reutilizar un razonamiento o una operación.

Aplicación parcial. Suministrar parte de los argumentos y obtener una función para los restantes. **Currificación.** Representar una función de varios argumentos mediante funciones que reciben uno cada vez.

Clausura. Función junto con el entorno necesario para interpretar sus variables libres. **Variable ligada.** Aparición cuyo alcance está determinado por una ligadura; **libre**, aquella que no queda ligada en la expresión considerada.

Composición. Construcción de una función aplicando una después de otra, respetando los tipos intermedios. **Orden superior.** Propiedad de una función que recibe funciones o devuelve una función.

Pureza. Ausencia de efectos observables y dependencias de resultado que no correspondan a los argumentos. **Transparencia referencial.** Posibilidad de sustituir expresiones por otras de igual significado preservando el comportamiento observable. **Totalidad.** Producción de un resultado definido para todas las entradas del dominio.

Inmutabilidad. Imposibilidad de cambiar un valor o estructura después de construidos, según la garantía concreta del modelo. **Persistencia.** Disponibilidad de versiones anteriores tras una actualización. **Compartición estructural.** Reutilización de nodos por varias versiones.

Constructor de tipos. Construcción que recibe tipos para formar otros, como List. **Constructor de datos.** Forma de construir valores de un tipo, como Hoja o Right. **Tipo producto.** Reunión de componentes. **Tipo suma.** Elección etiquetada entre alternativas.

Functor. Abstracción con map coherente con identidad y composición. **Applicative.** Abstracción con pure y aplicación contextual, útil para combinar computaciones cuya estructura no depende de sus resultados. **Monad.** Abstracción que compone continuaciones dependientes mediante flatMap y cumple sus leyes.

Fold. Recorrido que resume una estructura con operaciones dadas. **Monoide.** Tipo o conjunto con operación asociativa y elemento identidad. **Conmutatividad.** Propiedad adicional de una operación que permite intercambiar sus operandos.

Recursión estructural. Resolución de un dato mediante sus constructores y subestructuras. **Recursión de cola.** Recursión cuya llamada recursiva constituye el resultado final de la rama, sin trabajo pendiente en esa función.

No estrictitud. Posibilidad de obtener un resultado sin evaluar un argumento innecesario. **Evaluación por necesidad.** Evaluación diferida que comparte el resultado demandado. **Thunk.** Representación de una computación pendiente y su entorno. **Productividad.** Capacidad de producir las porciones demandadas en tiempo finito bajo las condiciones pertinentes.

Qué leer según la dificultad que encuentres

Necesidad	Recursos	Cómo aprovecharlos
Comprender la motivación	Hughes [1] y la historia de Haskell [36]	Identifica el problema de modularidad y distingue historia de semántica
Aprender funciones y recorridos	Tutorial de Haskell [3] y documentación OCaml [4, 5, 7]	Traduce un recorrido pequeño entre notaciones
Conectar con Scala	Scala 3 Book [21, 23, 24] y Tour [18, 20]	Reconstruye el reporte y cambia un caso de prueba
Entender leyes y tipos	Cats [28 a 33], Monoid [47] y artículo [46]	Escribe los tipos antes de interpretar cada igualdad
Practicar demostraciones	Cornell [45]	Repite la prueba de map y una prueba sobre Lote
Analizar memoria y evaluación	Okasaki [26], Data List [27] y LazyList [48]	Explica qué se comparte y qué queda retenido
Relacionar con una aplicación	Publicación sobre Sigma y Haxl [11]	Separa las decisiones del lenguaje de las de la biblioteca

El libro de Cornell permite estudiar con ejercicios y explicaciones más extensas. El artículo de McBride y Paterson requiere más comodidad con tipos y notación de Haskell; conviene abordarlo después de los ejemplos concretos de Applicative. El capítulo de Okasaki puede requerir acceso bibliográfico, mientras que la documentación oficial y los tutoriales ofrecen rutas de consulta en la web.

Para practicar con el código del equipo, utiliza Ejemplos.scala y Lotes.ml junto a esta guía. El primero incluye comprobaciones del reporte, errores, lotes, recursión de cola, leyes ilustradas y persistencia. El segundo permite comparar la sintaxis y el recorrido en OCaml. Los ejemplos Scala del material se comprobaron con Scala 3.3.6 y JDK 21; el archivo OCaml es material de contraste y no se presenta como una ejecución verificada en ese entorno.

La bibliografía siguiente conserva la numeración de las fuentes de la presentación y agrega recursos sobre Applicative, monoides y evaluación. Consulta documental de esta guía: 24 de noviembre de 2026. Las páginas vivas pueden cambiar; para estudiar una API concreta, contrasta la versión que tengas instalada con la versión documentada.

20 Bibliografía

Fuentes de la guía y documentación complementaria de la presentación. Los números conservan la referencia usada en el texto. Fecha de consulta de la guía 24 de noviembre de 2026.

[1] John Hughes (1990 [versión original: 1989]). Why Functional Programming Matters. Introducción y secciones 2–4. Consultar la fuente

[2] Simon Marlow (ed.) (2010). Haskell 2010 Language Report · Introduction. Capítulo 1. Consultar la fuente

[3] Paul Hudak, John Peterson y Joseph Fasel (2000, revisión de junio). A Gentle Introduction to Haskell · Functions. §3, especialmente §3.1 y §3.2. Consultar la fuente

[4] OCaml Documentation (s. f.). Values and Functions. Introduction, Function as Values y Function Application. Consultar la fuente

[5] OCaml Documentation (s. f.). Higher Order Functions. Common Higher-Order Functions; Mapping, Filtering y Folding. Consultar la fuente

[6] Paul Hudak, John Peterson y Joseph Fasel (2000, revisión de junio). A Gentle Introduction to Haskell · Input/Output. §7 y §7.1. Consultar la fuente

[7] OCaml Documentation (s. f.). Lists. Functions on Lists y Higher Order Functions on Lists. Consultar la fuente

[8] Cornell University · CS312 (2002). Lecture 21: Side Effects and References. Side Effects. Consultar la fuente

[9] Gang Tan · Penn State (2004). A Brief History of Functional Programming. Their common ancestor: Lambda Calculus. Consultar la fuente

[10] John McCarthy (1979). History of Lisp · The implementation of LISP. The implementation of LISP. Consultar la fuente

[11] Facebook Engineering (2015, 26 de junio). Fighting spam with Haskell. Why Haskell?, Automatic batching and concurrency y How Haskell fits in. Consultar la fuente

[12] Erlang/OTP (s. f.). What is Erlang?. §1.1, §1.3 y §1.5. Consultar la fuente

[13] OCaml Documentation (s. f.). Mutability and Imperative Control Flow. Referencias, mutabilidad y control imperativo. Consultar la fuente

[14] Erlang/OTP (s. f.). Expressions. Expression Evaluation y Variables. Consultar la fuente

[15] Simon Marlow (ed.) (2010). Haskell 2010 Language Report · Standard Prelude. PreludeList: map, filter, foldl, foldr y take; composición (.). Consultar la fuente

[16] Simon Marlow (ed.) (2010). Haskell 2010 Language Report · Expressions. §3.1, §3.3, §3.10 y §3.17. Consultar la fuente

[17] Scala Documentation (s. f.). Tour of Scala · Introduction. What is Scala?, Scala is object-oriented y Scala is functional. Consultar la fuente

[18] Scala Documentation (s. f.). Tour of Scala · Higher-order Functions. Higher-order Functions y Functions that accept functions. Consultar la fuente

[19] Scala Documentation (s. f.). Mutable and Immutable Collections. Mutable and Immutable Collections. Consultar la fuente

[20] Scala Documentation (s. f.). Tour of Scala · By-name Parameters. By-name Parameters. Consultar la fuente

[21] Scala Documentation (s. f.). Scala 3 Book · Pure Functions. Pure Functions y But impure functions are needed. Consultar la fuente

[22] Scala Standard Library (s. f.). List · foldLeft. foldLeft[B](z: B)(op: (B, A) => B). Consultar la fuente

[23] Scala Documentation (s. f.). Algebraic Data Types. Algebraic Datatypes y Recursive Enumerations. Consultar la fuente

[24] Scala Documentation (s. f.). Functional Error Handling. Option, Try y Either. Consultar la fuente

[25] Scala Documentation (s. f.). Performance Characteristics. Tablas de secuencias, mapas y explicación de C, eC, Log y L. Consultar la fuente

[26] Chris Okasaki (1998). Purely Functional Data Structures · §2 Persistence. Capítulo 2, pp. 7–16, extracto público del autor/editorial. Consultar la fuente

[27] Haskell base library (s. f.). Data.List · foldl and foldl prime. foldl y foldl': strictness, weak head normal form. Consultar la fuente

[28] Typelevel · Cats (s. f.). Functor. Functor: definición, Identity y Composition. Consultar la fuente

[29] Typelevel · Cats (s. f.). Applicative. pure, ap/product y relación con Functor. Consultar la fuente

[30] Typelevel · Cats (s. f.). Monad. pure, flatMap y flatten. Consultar la fuente

[31] Typelevel · Cats (s. f.). MonadLaws · identidades. monadLeftIdentity y monadRightIdentity. Consultar la fuente

[32] Typelevel · Cats (s. f.). FlatMapLaws · asociatividad. flatMapAssociativity. Consultar la fuente

[33] Typelevel · Cats (s. f.). ApplicativeLaws. applicativeIdentity, Homomorphism, Interchange y Composition. Consultar la fuente

[34] John McCarthy (1960). Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. Introduction; CACM 3(4), 184–195. Consultar la fuente

[35] John Backus (1978). Can Programming Be Liberated from the von Neumann Style?. Conferencia Turing 1977; CACM 21(8), 613–641, introducción y §11. Consultar la fuente

[36] Paul Hudak, John Hughes, Simon Peyton Jones y Philip Wadler (2007). A History of Haskell: Being Lazy With Class. §1–2: comité de 1987 y Haskell 1.0 de 1990. Consultar la fuente

[37] OCaml (s. f.). Why OCaml?. A Brief History; Features. Consultar la fuente

[38] John Cowan y comité R7RS (2013; edición HTML 2021). Revised⁷ Report on the Algorithmic Language Scheme. Introduction; §1; §4.1.4; §6.10. Consultar la fuente

[39] LispWorks / ANSI Common Lisp (s. f.). Common Lisp HyperSpec. §3 Evaluation and Compilation; §5 Data and Control Flow; §7 Objects. Consultar la fuente

[40] Microsoft Learn (s. f.). What is F#. Functional programming; data types; interoperability. Consultar la fuente

[41] Clojure (s. f.). Rationale. Functional programming; concurrency; hosted on the JVM. Consultar la fuente

[42] Elixir (s. f.). The Elixir programming language. Maintenance; Scalability; Language features. Consultar la fuente

[43] OCaml Documentation (s. f.). Basic Data Types and Pattern Matching. Tuples; Records; Variants; Recursive Variants. Consultar la fuente

[44] Scala Standard Library (s. f.). scala.annotation.tailrec. Descripción de la anotación tailrec. Consultar la fuente

[45] Cornell University · CS3110 (s. f.). Structural Induction. Induction on variants, lists and trees. Consultar la fuente

[46] Conor McBride y Ross Paterson (2008). Applicative Programming with Effects. Journal of Functional Programming 18 1 páginas 1 a 13. Consultar la fuente

[47] Typelevel Cats (s. f.). Monoid. Definición y leyes de identidad. Consultar la fuente

[48] Scala Standard Library (Versión 2.13.17). LazyList. Evaluación diferida memorización y retención de memoria. Consultar la fuente

[49] Simon Marlow editor (2010). Haskell 2010 Language Report Predefined Types and Classes. Capítulo 6 especialmente Functor y Monad. Consultar la fuente

[50] Cornell University CS3110 (2021). Capture Avoiding Substitution. Sección 10 2 3. Consultar la fuente

[51] Cornell University CS3110 (2018). Lab Higher Order Programming. Currying. Consultar la fuente