Capítulo 8

Computação com estados

A computação com estados é um conceito fundamental em programação, especialmente quando lidamos com sistemas que precisam manter informações ao longo do tempo. Em programação funcional, o tratamento de estados é feito de maneira diferente do paradigma imperativo, onde o estado é frequentemente modificado diretamente.

8.1 Conceitos básicos

Estado Em programação, *estado* refere-se ao conjunto de informações do programa em um determinado momento. Em termos práticos, o estado é a coleção de valores de todas as variáveis do programa em um dado instante. Por exemplo, se um programa possui três variáveis, a, b e c, o estado do programa é formado por um conjunto de tuplas (a, b, c) que representam os valores atuais dessas variáveis num dado instante de tempo. Por exemplo, se a = 1, b = 2 e c = 3, o estado do programa é (1, 2, 3). Se num momento posterior, a = 2, b = 3 e c = 4, o estado do programa passa a ser (2, 3, 4).

Esse conceito pode ser generalizado para qualquer tipo de dado, incluindo objetos complexos, listas, dicionários, etc. O estado de um objeto muda quando uma de suas propriedades é alterada. O estado de uma estrutura de dados muda quando um elemento é adicionado, removido ou modificado.

Computação com estados Do ponto de vista da manutenção de estado, podemos distinguir dois modelos computacionais:

- **Computação com estado (*stateful computation*)**: Nesse modelo, o programa depende do histórico de estados para determinar o resultado de uma operação. Por exemplo, em um carrinho de compras online, ao adicionar um novo item, precisamos saber quais itens já estão no carrinho (estado atual) para calcular o total corretamente. Numa conta bancária, o saldo atual depende de todas as transações anteriores (depósitos e retiradas). Nesse modelo, o estado é mantido e atualizado ao longo do tempo, e as operações podem modificar esse estado.

- **Computação sem estado (*stateless computation*)**: Nesse modelo, cada operação é independente do estado anterior. O resultado de uma operação depende apenas dos seus argumentos de entrada, sem considerar qualquer histórico. Por exemplo, uma função matemática pura, como a soma, é um exemplo de computação sem estado, pois o resultado depende apenas dos valores fornecidos como entrada. Outro exemplo é um serviço web RESTful, onde cada requisição é tratada de forma independente, sem manter informações sobre requisições anteriores. Nesse modelo, o estado não é mantido entre as operações, e cada operação é autônoma.

Em suma, existem problemas que exigem a manutenção de estado para serem resolvidos, enquanto outros podem ser resolvidos sem manter estado. Para os problemas que exigem estado, o modo como o estado é mantido e atualizado pode variar dependendo do paradigma de programação adotado.

MANUTENÇÃO IMPERATIVA DE ESTADO Na programação imperativa, o estado é mantido por meio da atualização das variáveis mutáveis do programa. Por exemplo, considere o código 8.1 que é uma abstração mutável para um contador.

Código 8.1: Contador mutável

```
package mutable {
    object Counter {
        private var counter = 0

        def increment: Unit = {
            counter += 1
        }

        def getValue: Int = counter
    }

    @main def testCounter(): Unit = {
        println(Counter.getValue)
        Counter.increment
        println(Counter.getValue)
        Counter.increment
        println(Counter.getValue)
        Counter.increment
        println(Counter.getValue)
    }
}
```

```
0
1
2
3
```

De modo geral, o estado de um objeto é composto pelos valores de todas as suas propriedades em um dado instante de tempo. Nesse programa, o objeto

`mutable.Counter` assume 4 estados, determinados pelos diferentes valores da variável `counter` ao longo da execução do programa: 0, 1, 2 e 3. Cada vez que chamamos o método `increment`, um novo estado do objeto é criado.

MANUTENÇÃO DECLARATIVA DE ESTADO Na programação funcional, o estado é mantido por meio de imutabilidade e persistência. Por exemplo, considere o código 8.2 que é uma abstração imutável para um contador.

Código 8.2: Contador imutável

```
package immutable {
    case class Counter private(value: Int) {
        def increment = Counter(value + 1)
    }

    object Counter {
        def apply(): Counter = new Counter(0)
    }

    @main def testCounter() = {
        val counter1 = Counter()
        val counter2 = counter1.increment
        val counter3 = counter2.increment
        val counter4 = counter3.increment

        println(counter1.value)
        println(counter2.value)
        println(counter3.value)
        println(counter4.value)
    }
}
```

```
0
1
2
3
```

Nesse programa, o objeto `immutable.Counter` também assume 4 estados, determinados pelos diferentes valores da propriedade `value` ao longo da execução do programa: 0, 1, 2 e 3. Cada vez que chamamos o método `increment`, um novo objeto é criado, mas o objeto anterior permanece inalterado. Por isso, precisamos amarrar e encadear os estados em variáveis diferentes (`counter1`, `counter2`, `counter3` e `counter4`).

ENCADEAMENTO Um padrão recorrente que observamos ao manipular estados declarativamente é o encadeamento de estados (*state threading*). O encadeamento se refere ao uso de um estado prévio para gerar um próximo estado, sucessivamente. No programa 8.2, isso ocorre quando precisamos usar o `counter` "i" para gerar a variável `counter` "i+1".

ENCADEAMENTO EXPLÍCITO Como podemos observar, quando usamos variá-

veis explícitas para fazer o encadeamento de estados, o programa tende a ficar mais verboso e difícil de entender. Isso ocorre porque precisamos criar uma sucessão de novas variáveis para armazenar os objetos intermediários. O objetivo desse capítulo é explorar como podemos simplificar a manutenção de estado em programação funcional, tornando-a mais escalável e menos verbosa.

Padrões de projeto Existem vários padrões de projeto que podem ser utilizados para gerenciar estados em programação funcional. Alguns são mais simples (e.g. recursão estrutural, mônadas de estado), enquanto outros são mais complexos (e.g. modelo de ator, programação reativa funcional). Nesse capítulo, vamos explorar alguns padrões mais simples para futuramente evoluirmos para padrões mais complexos.

8.2 Estudo de caso: conta bancária

Uma conta bancária é um exemplo clássico de sistema que precisa manter estados ao longo do tempo. Em um modelo simples de conta bancária, a informação mais importante a ser mantida é o saldo da conta. Com base na manipulação do saldo, podemos realizar operações como depósitos e saques.

Conta bancária mutável Em um modelo imperativo, podemos implementar uma conta bancária mutável como no código 8.3.

Código 8.3: Conta bancária mutável

```
1  package mutable {
2    class Account(private var balance: Double = 0.0) {
3
4      def getBalance: Double = balance
5
6      def deposit(amount: Double): Double = {
7        balance += amount
8        balance
9      }
10
11     def withdraw(amount: Double): Double = {
12       balance -= amount
13       balance
14     }
15   }
16 }
```

Nesse modelo, a conta bancária é representada por uma classe `Account` que possui uma variável mutável `balance` para armazenar o saldo. Os métodos `deposit` e `withdraw` modificam diretamente o estado da conta, atualizando o saldo.

Cliente imperativo Um programa cliente imperativo invocaria, a partir do mesmo objeto, uma sucessão de operações de depósito e saque, como no código 8.4.

Código 8.4: Cliente da conta bancária mutável

```
package mutable {
  @main def testAccount = {
    val account = new Account()
    val balanceReport = List(
      account.deposit(100.0),
      account.deposit(50.0),
      account.withdraw(30.0),
      account.deposit(40.0),
      account.withdraw(90.0)
    )
    println(balanceReport)
    println(account.getBalance)
  }
}
```

```
List(100.0, 150.0, 120.0, 160.0, 70.0)
70.0
```

Nesse programa, criamos uma conta bancária e realizamos uma série de depósitos e saques. A cada operação de saque ou depósito, gera-se um novo estado do objeto `account` e, consequentemente, um novo estado do programa. No modelo imperativo, um mesmo objeto pode assumir um número potencialmente infinito de estados, dependendo das operações realizadas e da cardinalidade das suas propriedades.

Por conveniência, criamos também um protótipo de "extrato" da conta, que é uma lista com os saldos após cada operação. Esse tipo de estrutura será útil para ilustrar o conceito de persistência de estado, mantendo o histórico de operações.

Conta bancária imutável Em um modelo funcional, não podemos operar na presença de mutabilidade. Por isso, precisamos criar uma abstração imutável para a conta bancária. Podemos fazer isso criando uma classe `Account` que representa o estado da conta e métodos que retornam novos estados da conta após cada operação, como no código 8.5.

Código 8.5: Conta bancária imutável

```
package immutable {
  case class Account(balance: Double = 0.0) {

    def deposit(amount: Double): (Double, Account) = {
      val newBalance = balance + amount
      (newBalance, Account(newBalance))
    }

    def withdraw(amount: Double): (Double, Account) = {
      val newBalance = balance - amount
      (newBalance, Account(newBalance))
    }
  }
```

```
}
```

Nesse novo modelo, precisamos garantir duas propriedades fundamentais para o nosso objeto: imutabilidade e persistência. A imutabilidade é garantida pela definição da classe `Account` como um `case class`, que define todas as propriedades (no caso, temos apenas uma) como imutáveis. A persistência é garantida pelo retorno de um novo objeto `Account` a cada operação de depósito ou saque, mantendo o estado anterior intacto. Por definição, o resultado de uma operação de depósito ou saque é um saldo atualizado. Como mantemos a persistência, precisamos retornar também o novo objeto `Account` com o saldo atualizado. Assim, cada operação retorna uma tupla com o novo saldo e o novo objeto `Account`.

Cliente funcional ingênuo Um programa cliente funcional invocaria, a partir de diferentes objetos, uma sucessão de operações de depósito e saque, como no código 8.6. Denominamos esse cliente de "ingênuo" porque ele não utiliza nenhum padrão de projeto para simplificar a manutenção de estado.

Código 8.6: Cliente da conta bancária imutável

```
package immutable {
    @main def statefulComputationWithValBindings = {
        val account = Account()

        val (balance1, acc1) = account.deposit(100.0)
        val (balance2, acc2) = acc1.deposit(50.0)
        val (balance3, acc3) = acc2.withdraw(30.0)
        val (balance4, acc4) = acc3.deposit(40.0)
        val (balance5, acc5) = acc4.withdraw(90.0)

        val balanceReport = List(balance1, balance2, balance3, balance4,
        ↪  balance5)
        println(balanceReport)
        println(acc5.balance)
    }
}
```

```
List(100.0, 150.0, 120.0, 160.0, 70.0)
70.0
```

Nesse programa, criamos uma conta bancária e realizamos uma série de depósitos e saques. A cada operação de saque ou depósito, devido à persistência, uma nova instância do objeto `Account` é criado. Como cada operação retorna uma tupla com o novo saldo e o novo objeto `Account`, usamos desestruturação para amarrar o novo saldo e o novo objeto em variáveis diferentes (`balance1`, `acc1`, `balance2`, `acc2`, etc.). Para gerar o extrato da conta (sequência de saldos após cada operação), criamos uma lista com os saldos retornados por cada operação.

Desvantagens do cliente ingênuo O cliente funcional ingênuo é verboso e difícil de manter. Isso se deve às seguintes razões:

- **Encadeamento manual de estados**. Precisamos amarrar manualmente cada

novo estado do objeto `Account` em uma nova variável. Isso é necessário pois o próximo estado depende do estado anterior — por exemplo, `acc2` depende de `acc1`, que depende de `account`. Esse encadeamento manual torna o código verboso e propenso a erros, especialmente em sequências longas de operações.

- **Verbosidade e prolixidade.** Em consequência do encadeamento manual de estados, o código se torna verboso e prolixo. Cada operação requer a criação de novas variáveis para armazenar os novos estados, o que pode levar a um aumento significativo na quantidade de código necessário para realizar operações simples. Cada linha repete o mesmo padrão, como se fosse um "copia e cola".
- **Ausência de componibilidade.** O cliente ingênuo não aproveita a componibilidade das funções. Cada operação é tratada de forma desconectada, em uma linha separada. Isso dificulta a criação de funções compostas que encapsulem sequências comuns de operações, tornando o código menos modular e reutilizável.

Com atenção a essas desvantagens, podemos explorar padrões de projeto que simplificam a manutenção de estado em programação funcional, tornando o código mais legível, modular e fácil de manter. Nas próximas seções, vamos analisar abstrações incrementalmente poderosas para simplificar os códigos que envolvem computações com estados.

8.3 Agregação de estados

Um padrão de projeto bastante útil para lidar com a manutenção de estados é o uso de funções de ordem superior que efetuam agregação de valores. Exemplos de funções clássicas para esse fim são as funções de dobra — `foldLeft`, `foldRight`, `reduceLeft`, `reduceRight`, entre outras. Nessa abordagem, definimos uma estrutura de dados que armazena as operações (por exemplo, uma lista encadeada de operações) e uma função de agregação que aplica essas operações em sequência, cumulativamente, para produzir um resultado final.

8.3.1 Acumuladores de estado

O padrão `Accumulator` é uma técnica comum em programação funcional para processar coleções de dados, onde um valor acumulado é atualizado iterativamente com base em cada elemento da coleção. A diferença entre um acumulador imperativo e um acumulador funcional reside no fato de um acumulador funcional ser persistente, isto é, precisamos gerar versões sucessivas do acumulador, mantendo as versões anteriores intactas.

Podemos usar a abstração de um acumulador para modelar as mudanças de estado de uma conta bancária, onde as diferentes versões do acumulador persistente representam os diferentes estados da conta ao longo do tempo.

Antes de explorarmos as funções agregadoras, vamos ilustrar o padrão `Accumulator` com uma abordagem manual, aplicada ao nosso de exemplo de contas

bancárias. Podemos modelar a sequência de operações (depósitos e saques) como processo cumulativo que se inicia com a aplicação no saldo inicial da conta e sucessivamente aos novos saldos. O código 8.7 ilustra uma possível implementação desse acumulador.

Código 8.7: Tipo para operações da conta bancária

```
case class AccountAccumulator(balances: List[Double], account: Account) {
  def apply(operation: Account => (Double, Account)): AccountAccumulator = {
    val (newBalance, newAccount) = operation(account)
    AccountAccumulator(balances :+ newBalance, newAccount)
  }
}
```

`AccountAccumulator` é um tipo produto que representa um acumulador composto por uma lista de saldos (`balances`) e um objeto `Account` que representa o estado atual da conta. A lista de saldos armazena os saldos sucessivos da conta após cada operação, permitindo gerar um extrato da conta facilmente.

Além disso, o acumulador possui um método `apply` que recebe uma operação (uma função que transforma um objeto `Account` em uma tupla com o novo saldo e o novo objeto `Account`) e retorna um novo acumulador com o estado atualizado. O método `apply` aplica a operação ao estado atual do acumulador, obtendo o novo estado da conta, e retorna um novo acumulador com esse novo estado.

Com base nesse tipo produto `AccountAccumulator`, podemos reescrever o cliente funcional ingênuo com um encadeamento mais direto, como no código 8.8.

Código 8.8: Cliente da conta bancária com acumulador

```
val result = AccountAccumulator(List.empty[Double], Account())
    .apply((account) => account.deposit(100.0))
    .apply((account) => account.deposit(50.0))
    .apply((account) => account.withdraw(30.0))
    .apply((account) => account.deposit(40.0))
    .apply((account) => account.withdraw(90.0))

result match {
    case AccountAccumulator(balances, account) =>
    println(s"Balance history: $balances")
    println(s"Final balance: ${account.balance}")
}
```

```
Balance history: List(100.0, 150.0, 120.0, 160.0, 70.0)
Final balance: 70.0
```

Interface fluente Perceba que eliminamos a necessidade de criar variáveis intermediárias para armazenar os estados sucessivos da conta. Em vez disso, encadeamos as operações diretamente no método `apply` do acumulador. Esse padrão é conhecido como *interface fluente* (*fluent interface*), sendo bastante recorrente em APIs orientadas a objetos. A interface fluente permite encadear chamadas de métodos de forma mais legível e expressiva, como se fosse uma linguagem natural. Nesse

modelo, cada chamada a `apply` aplica uma operação ao estado atual do acumulador, retornando um novo acumulador com o estado atualizado. No final, obtemos o saldo final da conta a partir do acumulador resultante.

MELHORIAS NA CONCISÃO Apesar de termos eliminado a criação de variáveis intermediárias, o código ainda está verboso. Vamos propor algumas melhorias:

- **Notação concisa para as operações**. Podemos usar a notação de máscara para tornar a definição da lista de operações mais concisa. Quando temos uma função anônima que recebe um único argumento, podemos omitir o nome do argumento e usar a notação de máscara diretamente. Por exemplo, a seguinte função anônima:

```
(account) => account.deposit(100.0)
```

  Pode ser reescrita como:

```
_.deposit(100.0)
```

- **Omissão de `apply`**. Devemos lembrar que, em Scala, que o método `apply` de um objeto é chamado automaticamente quando usamos a notação de função. Assim, podemos omitir a chamada explícita ao método `apply` e usar a notação de função diretamente. Por exemplo, a seguinte chamada:

```
accountAccumulator.apply((account) => account.deposit(100.0))
```

  Pode ser reescrita como:

```
accountAccumulator((account) => account.deposit(100.0))
```

Aplicando essas duas melhorias, podemos reescrever a lista de operações conforme o código 8.9.

Código 8.9: Lista de operações com notação de máscara

```
val result = AccountAccumulator(List.empty[Double], Account())
    (_.deposit(100.0))
    (_.deposit(50.0))
    (_.withdraw(30.0))
    (_.deposit(40.0))
    (_.withdraw(90.0))

  result match {
    case AccountAccumulator(balances, account) =>
      println(s"Balance history: $balances")
      println(s"Final balance: ${account.balance}")
  }
```

```
Balance history: List(100.0, 150.0, 120.0, 160.0, 70.0)
Final balance: 70.0
```

Com essa notação, o código fica mais enxuto, facilitando a leitura e compreensão da sequência de operações aplicadas ao acumulador.

LIMITAÇÕES Embora tenha melhorado a legibilidade do código, o padrão `Accumulator` ainda apresenta algumas limitações. A principal delas é que a quantidade de operações é fixa e definida no momento da escrita do código. Isso significa que não podemos adicionar ou remover operações dinamicamente em tempo de execução.

Naturalmente, poderíamos escrever uma função agregadora que recebe uma lista de operações e um acumulador, aplicando cada operação ao acumulador em sequência. Essa abordagem permitiria maior flexibilidade na definição das operações a serem realizadas.

No entanto, ao invés de "reinventar a roda", podemos usar funções agregadoras padronizadas, que são mais genéricas e permitem modelar qualquer operação de agregação sobre uma coleção de dados. Por exemplo, poderíamos recuperar uma lista de operações de um banco de dados ou de uma API externa e aplicar essas operações dinamicamente, sem precisar definir cada uma delas no código. Na próxima seção, estudaremos algumas dessas funções.

8.3.2 Funções de dobra

As funções de dobra, também conhecidas como funções agregadoras, são uma abstração poderosa para processar coleções de dados, permitindo acumular ou agregar resultados de maneira eficiente e expressiva. Elas operam sobre valores redutíveis (*foldables*), que são, em essência, coleções (por exemplo, listas, arrays, mapas, etc.). O princípio básico consiste em aplicar uma função acumuladora a cada elemento da coleção, produzindo um único resultado final. Uma função de dobra precisa das seguintes informações para operar:

- Uma coleção de elementos a serem processados
- O valor inicial do acumulador
- Uma função que especifica como combinar o acumulador com cada elemento da coleção

Vamos enfatizar duas variantes principais: `foldLeft` e `foldRight`, que são genéricas o suficiente para modelar qualquer operação de agregação sobre uma coleção de dados.

`foldLeft` A função `foldLeft` aplica uma função agregadora a cada elemento da coleção, começando pelo primeiro elemento e acumulando o resultado, associativamente, da esquerda para a direita. A assinatura típica de `foldLeft` é:

```
1 def foldLeft[B](z: B)(op: (B, A) => B):
```

onde z é o valor inicial (ou acumulador) e op é a função que combina o acumulador com cada elemento da coleção. Repare que na operação op, o primeiro argumento é o acumulador (do tipo B) e o segundo argumento é o elemento atual da coleção (do tipo A). O resultado de op torna-se o novo acumulador para a próxima iteração.

O resultado final é o valor acumulado após processar todos os elementos. Por exemplo, suponha que desejamos calcular a soma de uma lista de números inteiros usando `foldLeft`:

```
List(1, 2, 3, 4, 5).foldLeft(0)((acc, n) => acc + n)
```

Como a associatividade é da esquerda para a direita, o traço da execução corresponde à soma associativa da esquerda para a direita:

```
0 + 1 + 2 + 3 + 4 + 5
  1   + 2 + 3 + 4 + 5
      3   + 3 + 4 + 5
          6   + 4 + 5
              10  + 5
                  15
```

`reduceLeft` A função `reduceLeft` é uma variante de `foldLeft` que não requer um valor inicial. Ela assume que a coleção não está vazia e usa o primeiro elemento como o valor inicial do acumulador. A assinatura típica de `reduceLeft` é:

```
def reduceLeft(op: (A, A) => A):
```

Por exemplo, para calcular a soma de uma lista de números inteiros usando `reduceLeft`, podemos fazer:

```
List(1, 2, 3, 4, 5).reduceLeft((acc, n) => acc + n)
```

O resultado é o mesmo que o de `foldLeft`, mas sem a necessidade de fornecer um valor inicial. A execução segue o mesmo traço associativo da esquerda para a direita.

`foldRight` A função `foldRight` é análoga a `foldLeft`, mas aplica a função acumuladora da direita para a esquerda. A assinatura de `foldRight` é:

```
def foldRight[B](z: B)(op: (A, B) => B):
```

onde z é o valor inicial (ou acumulador) e op é a função que combina o elemento atual da coleção com o acumulador. Repare que na operação op, o primeiro argumento é o elemento atual da coleção (do tipo A) e o segundo argumento é o acumulador (do tipo B): essa ordem é invertida em relação à `foldLeft`. O resultado de op torna-se o novo acumulador para a próxima iteração.

O resultado final é o valor acumulado após processar todos os elementos, da direita para a esquerda. Por exemplo, para calcular a soma de uma lista de números inteiros usando `foldRight`, podemos fazer:

```
List(1, 2, 3, 4, 5).foldRight(0)((n, acc) => acc + n)
```

O resultado é o mesmo que o de `foldLeft`, mas a ordem das operações é invertida. O traço da execução corresponde à soma associativa da direita para a esquerda:

```
1 + 2 + 3 + 4 + 5 + 0
1 + 2 + 3 + 4 +   5
1 + 2 + 3 +   9
1 + 2 +   12
1 +   14
15
```

`reduceRight` A função `reduceRight` é uma variante de `foldRight` que não requer um valor inicial. O uso é análogo ao de `reduceLeft`, mas com a função acumuladora aplicada da direita para a esquerda. Devido ao uso análogo, recomenda-se ao leitor consultar a documentação oficial do Scala para mais detalhes.

NOTAÇÃO COM MÁSCARA A notação com máscara é uma forma de representar a aplicação de funções de ordem superior, como `foldLeft` e `foldRight`, de maneira concisa. Em vez de escrever explicitamente a função acumuladora, podemos usar uma notação com máscara para indicar a operação desejada. Por exemplo, podemos escrever:

```
List(1, 2, 3, 4, 5).foldLeft(0)(_ + _)
```

Nesse caso, o _ é um marcador que indica que a função acumuladora será aplicada aos elementos da lista. O primeiro _ representa o acumulador e o segundo _ representa o elemento atual da lista. Essa notação torna o código mais conciso e legível, especialmente quando a função acumuladora é simples.

No caso de `foldRight`, a notação com máscara tem uma semântica invertida, ou seja, o primeiro _ representa o elemento atual da lista e o segundo _ representa o acumulador. Isso tem uma implicação importante para operações que não são associativas.

COMPARAÇÃO ENTRE `foldLeft` E `foldRight` A principal diferença entre `foldLeft` e `foldRight` é a direção em que a função acumuladora é aplicada. Para operações associativas, ambas produzem o mesmo resultado. No entanto, para operações não associativas, o resultado pode ser diferente dependendo da direção da agregação. Por exemplo, para a subtração, `foldLeft` e `foldRight` poderiam produzir resultados diferentes, principalmente se usarmos a notação de máscara, conforme o código 8.10.

Código 8.10: Comparação entre `foldLeft` e `foldRight` para subtração

```
List(1, 2, 3, 4, 5).foldLeft(0)(_ - _)
List(1, 2, 3, 4, 5).foldRight(0)(_ - _)
```

```
-15
3
```

Para entender o resultado 3, basta fazer o traço da execução:

```
1 - 2 - 3 - 4 - (5 - 0)
1 - 2 - 3 - (4 - 5)
1 - 2 - (3 - (-1))
1 - (2 - 4)
(1 - (-2))
3
```

Naturalmente, se evitássemos a máscara poderíamos inverter a ordem dos argumentos e obter o mesmo resultado, mas isso não é possível com a notação de máscara. Por isso, é importante ter cuidado ao usar `foldRight` com operações não associativas, pois o resultado pode ser inesperado.

8.3.3 Dobra de estados

Agora que entendemos como as funções de dobra funcionam, podemos aplicá-las para simplificar a manutenção de estados declarativamente, ao mesmo tempo que melhoramos a legibilidade e concisão do código.

Lista de operações Podemos compreender uma computação com estados como uma sequência de operações que transformam um estado inicial em um estado final. Nesse contexto, podemos modelar esse processo como uma lista de operações que são aplicadas sequencialmente, de modo encadeado, a estados sucessivos. O código 8.11 ilustra como podemos representar essa lista de operações.

Código 8.11: Lista de operações

```
val operations = List[Account => (Double, Account)](
    _.deposit(100.0),
    _.deposit(50.0),
    _.withdraw(30.0),
    _.deposit(40.0),
    _.withdraw(90.0)
)
```

Como podemos ver, a lista `operations` contém funções anônimas que recebem um objeto `Account` e retornam o resultado de uma operação (método) desse objeto. Como as operações do nosso objeto possuem o mesmo tipo de retorno, uma tupla com o novo saldo e o novo objeto `Account`, o tipo da lista é homogêneo: `List[Account => (Double, Account)]`.

Aplicação da dobra Com base nessa lista de operações, podemos usar `foldLeft` para aplicar cada operação sequencialmente a um estado inicial, acumulando o resultado final. O código 8.12 ilustra como podemos fazer isso.

Código 8.12: Agregação de estados com `foldLeft`

```
val finalAccount = operations.foldLeft(initialAccount) {
    (acc, operation) => {
        val (_, nextAcc) = operation(acc)
        nextAcc
    }
}

println(s"Final balance: ${finalAccount.balance}")
```

```
Final balance: 70.0
```

Perceba que especializamos o `foldLeft` para o nosso problema:

- O "acumulador" é o estado atual da conta bancária (`initialAccount`), representado por um objeto `Account`.
- O valor inicial do acumulador é uma nova instância de `Account` com saldo inicial zero.
- A função acumuladora recebe o estado atual da conta (`currentAccount`) e uma operação da lista (`operation`).
- A função acumuladora aplica a operação ao estado atual da conta, obtendo o novo estado da conta (`nextAccount`).
- O resultado de `foldLeft` é o estado final da conta (`finalAccount`), após aplicar todas as operações da lista.

Embora a notação para expressar a sequência de operação da computação com estados seja mais concisa, nosso código ainda carece de uma funcionalidade que estava presente na versão imperativa: a geração de um extrato da conta, ou seja, uma lista com os saldos após cada operação. Ou, em outras palavras, precisamos da persistência dos resultados intermediários.

PERSISTÊNCIA DOS RESULTADOS Para obter os resultados intermediários das operações, podemos modificar a função acumuladora para também coletar os saldos após cada operação. Podemos fazer isso retornando uma tupla com o saldo atual e o estado da conta. O código 8.13 ilustra como podemos fazer isso.

Código 8.13: Agregação de estados com `foldLeft` e persistência

```
val (finalAccount, balanceHistory) =
      operations.foldLeft((initialAccount, List(initialAccount.balance))) {
        (acc, operation) =>
            val (currentAccount, report) = acc
            val (newBalance, nextAccount) = operation(currentAccount)
            (nextAccount, report :+ newBalance)
      }
println(s"Balance history: $balanceHistory")
println(s"Final balance: ${finalAccount.balance}")
```

```
Balance history: List(0.0, 100.0, 150.0, 120.0, 160.0, 70.0)
Final balance: 70.0
```

As modificações mais importantes foram:

- Definimos um acumulador composto, representado por uma tupla com o estado atual da conta (`currentAccount`) e uma lista vazia (`List.empty[Double]`) para armazenar os saldos intermediários.
- Usamos desestruturação para obter o estado atual da conta (`currentAccount`) e o relatório parcial de saldos (`report`) do acumulador.
- Em seguida, definimos o novo valor do acumulador por meio da aplicação da operação atual (`operation`) ao estado atual da conta (`currentAccount`). O resultado é uma tupla com o novo estado da conta (`nextAccount`) e o novo saldo (`newBalance`).
- Por fim, a função acumuladora retorna uma tupla com o novo estado da conta (`nextAccount`) e a lista de saldos atualizada (`report ::+ newBalance`). Aqui usamos a operação de inserção no final da lista que, embora ineficiente, é mais simples de expressar no código.

A principal lição desse exemplo é que podemos usar acumuladores compostos no processo de agregação para manter resultados complementares.

`scanLeft` Como o padrão de manter os resultados intermediarios é tão comum, a biblioteca padrão do Scala oferece uma função chamada `scanLeft` que faz exatamente isso. A assinatura de `scanLeft` é semelhante à de `foldLeft`, mas retorna uma coleção com todos os resultados intermediários, incluindo o valor inicial. A assinatura típica de `scanLeft` é:

```scala
def scanLeft[B](z: B)(op: (B, A) => B): CC[B]
```

Perceba que o resultado de `scanLeft` é uma coleção (`CC[B]`) que contém todos os valores intermediários, incluindo o valor inicial (`z`). Para o nosso exemplo, a coleção retornada será uma sequência de todos os estados da conta corrente, conforme o código 8.14.

Código 8.14: Exemplo de `scanLeft` para conta bancária

```scala
val accounts: List[Account] = operations.scanLeft(initialAccount) {
    (acc, operation) =>
        val (_, nextAcc) = operation(acc)
        nextAcc
}

println(s"Final balance: ${accounts.last.balance}")
println(s"Balance report: ${accounts.map(_.balance)}")
```

```
Final balance: 70.0
Balance report: List(0.0, 100.0, 150.0, 120.0, 160.0, 70.0)
```

Observe que o resultado de `scanLeft` é uma lista de objetos `List[Account]`, onde cada objeto representa um estado da conta após cada operação. O último elemento da lista (`accounts.last`) é o estado final da conta, enquanto a lista mapeada (`accounts.map(_.balance)`) contém todos os saldos intermediários.

8.3.4 Limitações da agregação de estados

A abordagem de agregação de estados oferece uma notação concisa e expressiva de manter estados em programação funcional. Ela permite que tratemos sequências de operações como uma lista de transformações, aplicando-as sequencialmente a um estado inicial. Além disso, a persistência dos resultados intermediários é facilmente alcançada, tornando o código mais legível e modular. No entanto, essa abordagem oferece limitações:

- **Homogeneidade de estados**. A abordagem de agregação de estados assume que todas as operações são homogêneas, ou seja, que elas operam sobre o mesmo tipo de estado e produzem resultados do mesmo tipo. No entanto, é possível que precisemos executar operações heterogêneas, envolvendo diferentes tipos de estado. Uma transferência entre contas bancárias, por exemplo, envolve duas operações distintas: uma para debitar de uma conta e outra para creditar em outra conta:

```
def transfer(from: Account, to: Account, amount: Double): (Account,
↪  Account) = {
    val (balance1, newFrom) = from.withdraw(amount)
    val (balance2, newTo) = to.deposit(amount)
    (newFrom, newTo)
}
```

  Esse tipo de operação seria muito difícil de expressar usando a abordagem de agregação de estados, pois teríamos que lidar com dois estados diferentes (as duas contas) e suas respectivas operações de forma independente.

- **Sem resiliência a erros**. Algumas operações podem ser funções parciais, que são adaptadas para retornar tipos opcionais. Por exemplo, a operação de saque poderia negar o saque na ausência de fundos:

```
def withdraw2(amount: Double): (Either[AccountError, Double], Account)
↪  = {
    if (amount > balance) {
        (Left(InsufficientFunds(balance, amount)), this)
    } else {
        val newBalance = balance - amount
        (Right(newBalance), Account(newBalance))
    }
}
```

Na presença de um erro de insuficiência de fundos, a agregação com `foldLeft` simplesmente ignoraria o erro, continuando a computação com o estado anterior:

```
@main def statefulComputationWithFoldLeftError = {
    // Define the sequence of operations
    val initialAccount = Account()

    val operations = List(
    (account: Account) => account.deposit(100.0),
    (account: Account) => account.deposit(50.0),
    (account: Account) => account.withdraw(30.0),
    (account: Account) => account.withdraw2(300.0),
    (account: Account) => account.deposit(40.0),
    (account: Account) => account.withdraw(90.0),
    )

    // Use foldLeft to chain operations and collect all balances
    val (finalAccount, balanceHistory) =
    operations.foldLeft((initialAccount, List[Double |
    ↪  Either[AccountError, Double]](initialAccount.balance))) {
        (acc, operation) =>
            val (currentAccount, report) = acc
            val (newBalance, nextAccount) = operation(currentAccount)
            (nextAccount, report :+ newBalance)
    }

    println(s"Balance history: $balanceHistory")
    println(s"Final balance: ${finalAccount.balance}")
}
```

```
Balance history: List(0.0, 100.0, 150.0, 120.0,
    Left(InsufficientFunds(120.0,300.0)), 160.0, 70.0)
Final balance: 70.0
```

Perceba que no histórico de saldos, a operação de saque que falhou (devido a fundos insuficientes) resultou em um valor `Left(InsufficientFunds(120.0,300.0))`, que indica que a operação não pôde ser concluída. Porém, essa operação foi simplesmente ignorada na computação do saldo final.

- Entre outras limitações importantes.

Devido a essas limitações, a abordagem de agregação de estados pode não ser a mais adequada para todos os cenários. Em particular, ela pode se tornar complexa e difícil de gerenciar quando lidamos com operações que envolvem múltiplos estados, operações heterogêneas ou que requerem lógica condicional. Em resumo, para cenários mais complexos de computação com estados, precisamos de abordagens mais poderosas.

8.4 Mônadas

Antes de avançarmos para novas abordagens de computação com estados, é necessário introduzir um conceito geral de programação funcional que fundamenta muitas abordagens para lidar com efeitos colaterais: as mônadas. Antes de provermos uma definição formal do conceito de mônadas, vamos explorar uma definição mais intuitiva. Por fim, vamos ver exemplos de mônadas comumente usadas em Scala.

8.4.1 Definição intuitiva

Uma mônada é um padrão de projeto abstrato utilizado para sequenciar computações com um contexto. De modo mais intuitivo, podemos dizer que uma mônada é um container que encapsula um valor. Além de um valor, uma mônada também possui um contexto e um conjunto de regras que ditam como o seu valor encapsulado deve ser manipulado.

ENCAPSULAMENTO Em estudos anteriores, já nos deparamos com alguns tipos monâdicos:

- `Option`: encapsula um valor que pode estar presente ou ausente.

```scala
val v1: Option[Int] = Some(42)
val v2: Option[Int] = None
```

- `Either`: encapsula um valor de sucesso ou um valor de erro.

```scala
val v1: Either[String, Int] = Right(42)
val v2: Either[String, Int] = Left("Erro")
```

- `Future`: encapsula um valor que será computado assíncronamente.

```scala
val v1: Future[Int] = Future.successful(42)
val v2: Future[Int] = Future.failed(new Exception("Erro"))
```

- `List`: encapsula uma coleção de valores.

```scala
val v1: List[Int] = List(1, 2, 3)
val v2: List[Int] = List.empty
```

CONTEXTO A noção de contexto de uma mônada é fundamental para entender como ela funciona. O contexto pode ser visto como um ambiente que envolve o valor encapsulado, fornecendo informações adicionais sobre como esse valor deve ser tratado. Em termos concretos, o contexto envolve:

- A estrutura mantida pela mônada para manter o valor encapsulado.

- A lógica de encadeamento da mônada, representada pelo método `flatMap`.

Ainda não analisamos o que é o método `flatMap`, mas podemos pensar nele como uma transformação no valor encapsulado, resultando em um novo valor encapsulado no mesmo contexto. Por isso, podemos dizer que a implementação da função `flatMap` é dependente de contexto.

Diferentes tipos de mônadas representam diferentes tipos de contexto.

- `Option`: o contexto é a **opcionalidade**. O valor encapsulado pode estar presente (`Some`) ou ausente (`None`).
- `Either`: o contexto é a **possibilidade** de ocorrer um erro. O valor encapsulado pode ser um valor de sucesso (`Right`) ou um valor de erro (`Left`).
- `Future`: o contexto é **assincronicidade**. O valor não está imediatamente disponível — é uma promessa que será cumprida no futuro.
- `List`: o contexto é a **coleção** de valores. O valor encapsulado é uma lista de elementos, que podem ser processados em sequência.

Encadeamento O poder das mônadas reside principalmente na capacidade de encadear operações que manipulam valores encapsulados em um contexto. Isso é feito principalmente por meio da operação `flatMap`, que executa o seguinte processo:

- **Extração**: o valor encapsulado é extraído do contexto da mônada.
- **Aplicação**: uma função é aplicada ao valor extraído, resultando em um novo valor.
- **Encapsulamento**: o novo valor é encapsulado de volta no contexto da mônada.

Estudo de caso: composição de funções parciais Suponha que temos que temos duas funções, uma para calcular o fatorial e outra calcular o recíproco de um número, conforme o código 8.15.

Código 8.15: Funções parciais

```
def factorial(v: Int): Option[Int] =
  if (v < 0) None else Some((1 to v).product)

def reciprocal(x: Int): Option[Double] =
  if (x != 0) Some(1.0 / x) else None
```

Ambas as funções são parciais: fatorial só aceita números não-negativos e recíproco só aceita números diferentes de zero. Indicamos esse fato pelo valor de retorno ser `None` em casos de entrada inválida.

Suponha que queiramos avaliar a seguinte expressão abstrata:

```
reciprocal(factorial(x)) + 10
```

Não conseguimos avaliar essa expressão diretamente pois as funções não são componíveis: `factorial` retorna um `Option[Int]` e `reciprocal` espera um `Int`. Para resolver isso, vamos criar uma função auxiliar que extrai os valores necessários e trata a ausência dos valores quando houver, conforme o código 8.16.

Código 8.16: Composição de funções parciais

```
def calculateMatch(n: Int): Option[Double] =
  factorial(n) match {
    case Some(f) => reciprocal(f) match {
      case Some(r) => Some(r + 10)
      case None    => None
    }
    case None    => None
  }
```

A função 8.16 precisa fazer uma série de verificações para garantir que os valores intermediários estejam presentes antes de prosseguir com a computação. Isso resulta em um código verboso e difícil de ler, especialmente quando temos várias funções parciais encadeadas.

Perceba que temos um padrão de processamento:

- **Extração**. Precisamos extrair o valor encapsulado de `Option`: isso é feito pela expressão `match`.
- **Transformação**. Precisamos aplicar a próxima função ao valor extraído: isso é feito pela mera aplicação da função ao valor extraído.
- **Encapsulamento**. Precisamos encapsular o resultado da aplicação em um novo `Option`. Isso é feito pela própria função, que retorna um `Option`.

É justamente esse padrão de processamento (extração -> transformação -> encapsulamento) que a operação `flatMap` abstrai. Agora precisamos lembrar que `Option` é um tipo monádico e, como tal, opera sobre as regras monádicas que permitem o uso da operação `flatMap`. Por exemplo, o código 8.17 mostra como podemos reescrever a função 8.16 usando `flatMap`.

Código 8.17: Composição de funções parciais com `flatMap`

```
def calculateFlatmap(n: Int): Option[Double] =
  factorial(n).flatMap(f => reciprocal(f).map(_ + 10))
```

Perceba que o código ficou muito mais conciso e aumentamos a legibilidade. Por outro lado, quando usamos um encadeamento muito denso de `flatMaps`, é comum que a legibilidade diminua. Para tanto, temos uma sintaxe especial que ajuda a tornar o encadeamento mais "linear", a compreensão `for`. O código 8.18 ilustra como podemos usar a compreensão for para tornar o encadeamento mais legível.

Código 8.18: Composição de funções parciais com `for`

```
def calculateFor(n: Int): Option[Double] = for {
  f <- factorial(n)
  r <- reciprocal(f)
} yield r + 10
```

Em suma, as mônadas abstraem o padrão de processamento (extração -> transformação -> encapsulamento) e fornecem uma maneira de sequenciar computações de forma mais legível e concisa.

8.4.2 Definição formal

Uma mônada é um construtor de tipos equipado com duas operações:

- `pure` ou `apply`. Encapsula um valor puro `A` no contexto de uma mônada `M`, resultando em `M[A]`. Atua como um "construtor" de mônadas a partir de valores puros.
- `flatMap` ou `bind`. Permite encadear operações que produzem uma nova mônada, facilitando a composição de computações. Partindo de uma mônada `M[A]`, extrai o valor A, aplica uma função `f: A => M[B]`, resultando na mônada `M[B]`.

Além das duas operações, uma mônada deve satisfazer as seguintes leis:

- `left identity`. Para qualquer valor `a: A` e função `f: A => M[B]`, a seguinte equivalência deve ser verdadeira:

  ```
  pure(a).flatMap(f) == f(a)
  ```

- `right identity`. Para qualquer mônada `m: M[A]`, a seguinte equivalência deve ser verdadeira:

  ```
  m.flatMap(pure) == m
  ```

- `associativity`. Para qualquer mônada `m: M[A]`, e funções `f: A => M[B]` e `g: B => M[C]`, a seguinte equivalência deve ser verdadeira:

  ```
  m.flatMap(f).flatMap(g) == m.flatMap(a => f(a).flatMap(g))
  ```

8.4.3 Estudo de caso: `Option` como mônada

Já mencionamos que `Option` é um tipo monádico. Vamos demonstrar como ele satisfaz as propriedades de uma mônada.

Operações de mônada O tipo `Option` possui as seguintes operações que satisfazem as propriedades de uma mônada:

- `pure` (ou `apply`): Para um `Option`, essa propriedade é satisfeita pela possibilidade de encapsular valores em um `Option`. Por exemplo:

```
scala> val option: Option[Int] = Some(42)
val option: Option[Int] = Some(42)

scala> val option2: Option[Int] = None
val option2: Option[Int] = None
```

O valor `None`, embora não encapsule um valor, também é considerado uma instância de `Option`, representando a ausência de um valor.

- `flatMap`: `Option` possui uma implementação bastante simples de `flatMap`:

```
def flatMap[B](f: A => Option[B]): Option[B] =
    if (isEmpty) None else f(this.get)
```

Note que, caso o `Option` seja `Some`, a função `f` é aplicada ao valor encapsulado, retornando um novo `Option[B]`. Caso contrário, se o `Option` for `None`, a função não é aplicada e o resultado é `None`. Por exemplo:

```
scala> Some(42).flatMap((x: Int) => Some(x * 2))
val res3: Option[Int] = Some(84)

scala> None.flatMap((x: Int) => Some(x * 2))
val res4: Option[Int] = None
```

Em resumo, a principal característica de `flatMap` é aplicar uma transformação no valor encapsulado, mas delegar para a função de transformação a responsabilidade de encapsular o resultado em um novo `Option`. Isso permite que a computação continue de forma encadeada, mesmo quando o valor não está presente.

LEIS DA MÔNADA O programa 8.19 ilustra a verificação das leis da mônada para `Option`.

Código 8.19: Verificação das leis da mônada para `Option`

```
@main def LawsForSome = {
  def a = 42
  def m = Some(a)
  def f(a: Int) = Some(a * 2)
  def g(a: Int) = Some(a + 3)

  // pure(a).flatMap(f) == f(a)
  def leftIdentity = Some(a).flatMap(f) == f(a)

  // m.flatMap(pure) == m
  def rightIdentity = m.flatMap(Some(_)) == m

  // m.flatMap(f).flatMap(g) ==
```

```
  def associativity = m.flatMap(f).flatMap(g) == m.flatMap(a =>
  ↪  f(a).flatMap(g))

  println(s"Left Identity: $leftIdentity")
  println(s"Right Identity: $rightIdentity")
  println(s"Associativity: $associativity")
}
```

```
Left Identity: true
Right Identity: true
Associativity: true
```

Como podemos observar, caso o `Option` seja do tipo `Some`, as leis da mônada são satisfeitas. No entanto, para que as leis da mônada sejam válidas, precisamos garantir que o `Option` também trate corretamente o caso em que é `None`.

Exercício. Escreva um programa semelhante ao 8.19, mas que trate o caso em que o `Option` é `None`. Verifique se as leis da mônada são satisfeitas.

8.4.4 Composição de mônadas

Uma característica importante das mônadas é que elas permitem a composição de computações que operam em valores encapsulados. Isso é feito por meio da função `flatMap`. Considere como base o código 8.20, que define uma função parcial para somar inteiros positivos.

Código 8.20: Exemplo de função parcial para somar inteiros positivos

```
def addPositive(x: Int, y: Int): Option[Int] = {
    if (x > 0 && y > 0) Some(x + y)
    else None
}
```

A função `addPositive` recebe dois inteiros e retorna um `Option[Int]` que contém a soma se ambos os inteiros forem positivos, ou `None` caso contrário. Essa função é parcial, pois não está definida para os casos em que um dos argumentos é negativo ou zero.

A composição de funções parciais pode ser realizada utilizando `flatMap`. Por exemplo, podemos somar os números de 1 a 5 utilizando composição de chamadas `flatMap`, conforme o código 8.21.

Código 8.21: Composição de funções parciais com `flatMap` com resultado de sucesso

```
@main def testCompositionSuccess = {
    val result: Option[Int] = addPositive(1, 2)
        .flatMap(sum1 => addPositive(sum1, 3))
        .flatMap(sum2 => addPositive(sum2, 4))
        .flatMap(sum3 => addPositive(sum3, 5))

    result match {
```

```
        case Some(value) => println(s"Final composed value: $value")
        case None => println("Composed value is not positive")
    }
}
```

```
Final composed value: 15
```

Nesse exemplo, usamos `flatMap` para encadear as chamadas a `addPositive`. Cada chamada a `flatMap` aplica a função de soma ao resultado da chamada anterior, permitindo que a computação continue de forma encadeada. Se em algum ponto a soma não for positiva, o resultado final será `None`. Podemos observar esse resultado no código 8.22.

Código 8.22: Composição de funções parciais com `flatMap` com resultado de falha

```
@main def testCompositionFailure = {
    val result: Option[Int] = addPositive(1, 2)
        .flatMap(sum1 => addPositive(sum1, -3))
        .flatMap(sum2 => addPositive(sum2, 4))
        .flatMap(sum3 => addPositive(sum3, 5))

    result match {
        case Some(value) => println(s"Final composed value: $value")
        case None => println("Composed value is not positive")
    }
}
```

```
Composed value is not positive
```

Nesse exemplo, a segunda composição falha porque há uma tentiva de efetuar soma com argumento negativo (-3). Perceba que, mesmo havendo outras composições adiante, o resultado final será `None`.

`Either` Embora tenhamos focado nossa discussão na mônada `Option`, é importante mencionar que `Either` também é uma mônada, portanto obedece às mesmas leis. Por exemplo, considere o código 8.23, que define uma função parcial para somar inteiros positivos, mas agora usando `Either` para representar o erro.

Código 8.23: Exemplo de função parcial para somar inteiros positivos com `Either`

```
sealed trait AddError
case class LeftOperandNegative(x: Int) extends AddError
case class RightOperandNegative(y: Int) extends AddError

def addPositive2(x: Int, y: Int): Either[AddError, Int] = {
    if (x > 0 && y > 0) Right(x + y)
    else if (x <= 0) Left(LeftOperandNegative(x))
    else Left(RightOperandNegative(y))
}
```

Optamos por representar os erros numa granularidade mais fina: agora pode-

mos distinguir entre diferentes tipos de erros de adição, como operandos negativos à esquerda ou à direita. O código 8.24 ilustra como podemos compor essas funções parciais usando `flatMap`, em uma situação de erro.

Código 8.24: Composição de funções parciais com `flatMap` com resultado de falha

```
@main def testCompositionEitherFailure = {
    val result: Either[AddError, Int] = addPositive2(1, 2)
        .flatMap(addPositive2(_, -3))
        .flatMap(addPositive2(_, 4)) // This will cause an error
        .flatMap(addPositive2(_, 5))

    result match {
        case Right(value) => println(s"Final composed value: $value")
        case Left(error) => println(s"Composed value is not positive:
        ↪  $error")
    }
}
```

```
Composed value is not positive: RightOperandNegative(-3)
```

Perceba que temos um erro já na segunda composição. As composições com `flatMap` garantem que, a partir desse ponto, nenhuma composição adicional será realizada se um erro for encontrado. A expressão é avaliada em sua totalidade, no entanto o erro "borbulha" para o resultado da composição. Outra observação importante é que, nesse exemplo, usamos uma notação concisa: `addPositive2(_, -3)` é o mesmo que `x => addPositive2(x, -3)`.

COMPOSIÇÃO MONÁDICA COM FUNÇÕES DE DOBRA No exemplo do código 8.24 as chamadas de `flatMap` recebem sempre o mesmo tipo função, alterando apenas o valor dos argumentos. Isso é um padrão comum em composições monádicas, que pode ser expresso de forma mais concisa usando funções de dobra, como `foldLeft`. O código 8.25 ilustra como podemos usar `foldLeft` para compor uma lista de operações parciais.

Código 8.25: Composição de funções parciais com `foldLeft`

```
@main def eitherWithFold = {
    def addPositives(start: Int, values: Int*): Either[AddError, Int] = {
        values.foldLeft(Right(start): Either[AddError, Int]) { (acc, v) =>
            acc.flatMap(addPositive2(_, v))
        }
    }

    addPositives(1, 2, -3, 4, 5) match {
        case Right(value) => println(s"Final composed value: $value")
        case Left(error) => println(s"Composed value is not positive:
        ↪  $error")
    }
}
```

```
Composed value is not positive: RightOperandNegative(-3)
```

Note que agora, diferentemente do que obtivemos com o padrão de acumulador, os erros propagam-se para o resultado da composição. Isso é mais uma evidência do maior poder das mônadas para encadear operações de forma segura.

8.4.5 Compreensões de mônadas

A notação de composição explícita com `flatMap` e `map` pode ser verbosa e difícil de ler, especialmente quando há várias composições. Para tornar o código mais legível, Scala oferece uma sintaxe especial chamada compreensão de mônadas (*monad comprehension*). Essa sintaxe permite escrever composições de forma mais concisa e expressiva.

Uma compreensão em Scala é um mero açúcar sintático para composições de mônadas, ou seja, para a utilização sequencial de operações `map`, `flatMap` e condições de guarda. Uma compreensão de mônada é definida usando a palavra-chave `for` seguida por uma ou mais expressões `yield`. A sintaxe básica de uma compreensão de mônada é:

```
for {
    a <- fa
    b <- fb
} yield (a, b)
```

Onde:

- `fa` e `fb` são geradores, ou expressões que produzem valores dentro de um contexto monádico.
- `a` e `b` são variáveis que recebem os valores extraídos dos geradores.
- `yield` é a expressão que produz o valor final da compreensão.

Compreensão em listas Precisamos lembrar que o tipo `List` é um tipo monádico. Como tal, podemos usar compreensões de mônadas para trabalhar com listas de forma mais expressiva. Por exemplo, considere o código 8.26 que define uma compreensão de mônada para gerar pares de todos os elementos.

Código 8.26: Exemplo de compreensão de mônada com listas

```
@main def listComprehension2 = {
    val nums1 = List(1, 2)
    val nums2 = List(10, 20)

    // For-comprehension
    val forResult = for {
        x <- nums1
        y <- nums2
    } yield (x, y)
```

```
10
11      // Equivalent with flatMap/map/filter
12      val explicitResult = nums1.flatMap(x =>
13          nums2.map(y => (x, y))
14      )
15
16      println(s"For-comprehension: $forResult")  // List(20, 40)
17      println(s"Explicit:          $explicitResult")
18      println(s"Equal: ${forResult == explicitResult}")
19  }
```

```
For-comprehension: List((1,10), (1,20), (2,10), (2,20))
Explicit: List((1,10), (1,20), (2,10), (2,20))
Equal: true
```

Nesse exemplo, a compreensão de mônada gera todos os pares possíveis de números das listas `nums1` e `nums2`, sendo que o primeiro elemento é retirado de `nums1` e o segundo de `nums2`. O resultado é uma nova lista contendo os pares.

Notação sequencial Não por acaso, o uso de compreensões de mônadas se assemelha à notação sequencial, onde as operações são encadeadas de forma linear. Essa semelhança torna o código mais legível e expressivo, permitindo que os desenvolvedores pensem nas operações de forma mais intuitiva. O nome `for` faz referência a laços de repetição, comuns em linguagens imperativas, mas aqui o significado é diferente: estamos definindo uma *sequência* de operações que serão aplicadas a valores encapsulados em um contexto monádico.

`flatMap` e `map` As operações intermediárias da compreensão de mônada são traduzidas para chamadas a `flatMap` e `map`. Cada expressão na compreensão que envolve a extração de um valor do contexto monádico (usando o operador '<-') é traduzida para uma chamada a `flatMap`. A última expressão, que produz o valor final (após a palavra-chave `yield`), é traduzida para uma chamada a `map`. Condições de guarda (usando a palavra-chave `if`) são traduzidas para chamadas a `filter`.

Forma geral De modo geral, uma compreensão de mônada no seguinte formato:

```
for {
  x1 <- generator1
  x2 <- generator2 if condition
  x3 <- generator3
  ...
  xn <- generatorN
} yield result
```

É convertida para uma composição de chamadas `flatMap`, `map` and `withFilter`:

```
generator1.flatMap { x1 =>
  generator2.withFilter { x2 => condition(x2) }.flatMap { x2 =>
    generator3.flatMap { x3 =>
```

```
      ...
      generatorN.map { xn =>
        result(x1, x2, ..., xn)
      }
    }
  }
}
```

COMPREENSÕES DE MÔNADAS `Either` Por serem tipos monádicos, tanto `Option` quanto `Either` suportam compreensões de mônadas. Por exemplo, considere o código 8.27 que define uma compreensão de mônada para somar números positivos, reaproveitando definições que vimos em seções anteriores.

Código 8.27: Exemplo de compreensão de mônada com `Either` (caso de sucesso)

```
@main def testEitherComprehensionSuccess = {
    val result: Either[AddError, Int] = for {
        sum1 <- addPositive2(1, 2)
        sum2 <- addPositive2(sum1, 3)
        sum3 <- addPositive2(sum2, 4)
    } yield sum3

    result match {
        case Right(value) => println(s"Final composed value: $value")
        case Left(error) => println(s"Composed value is not positive:
        ↪  $error")
    }
}
```

```
Final composed value: 10
```

Nesse exemplo, a compreensão de mônada `Either` foi capaz de encadear várias operações que podem falhar, retornando um valor de sucesso no final. Na presença de erros, a compreensão de mônada `Either` também pode ser usada para encadear operações que podem falhar. Por exemplo, considere o código 8.28 que define uma compreensão de mônada para somar números positivos, mas agora com um erro.

Código 8.28: Exemplo de compreensão de mônada com `Either` (caso de falha)

```
@main def testEitherComprehensionFailure = {
    val result: Either[AddError, Int] = for {
        sum1 <- addPositive2(1, 2)
        sum2 <- addPositive2(sum1, -3) // This will cause an error
        sum3 <- addPositive2(sum2, 4)
    } yield sum3

    result match {
        case Right(value) => println(s"Final composed value: $value")
        case Left(error) => println(s"Composed value is not positive:
        ↪  $error")
```

```
    }
}
```

```
Composed value is not positive: RightOperandNegative(-3)
```

Repare que, neste caso, a compreensão de mônada `Either` foi capaz de encadear operações que podem falhar, retornando um valor de erro no final. Isso demonstra como as mônadas podem ser usadas para lidar com falhas de forma elegante e composicional.

COMPREENSÕES VS. FUNÇÕES DE DOBRA Os exemplos do código 8.27 e do código 8.28 podem ser facilmente convertidos para uma versão com função de dobra, pois todas as computações efetuadas na compreensão são do mesmo tipo. No entanto, as compreensões são mais poderosas que as funções de dobra, pois permitem a composição de operações heterogêneas, que podem envolver diferentes tipos de estado ou diferentes tipos de erros. Por exemplo, considere o código 8.29, que define uma compreensão de mônada que envolve operações heterogêneas.

Código 8.29: Exemplo de compreensão de mônada com operações heterogêneas

```
def validateRange(x: Int): Either[String, Int] = {
    if (x >= 1 && x <= 100) Right(x)
    else Left(s"Number $x out of range [1,100]")
}

@main def testComprehensionHeterogeneous = {
    val result: Either[AddError | String, Int] = for {
        sum1 <- addPositive2(1, 2)
        sum2 <- addPositive2(sum1, -3) // This will cause an error
        sum3 <- addPositive2(sum2, 4)
        result <- validateRange(sum3) // Different type of computation
    } yield result

    result match {
        case Right(value) => println(s"Final composed value: $value")
        case Left(error) => println(s"Composed value is not positive:
        ↪  $error")
    }
}
```

CURTO-CIRCUITO . Um comportamento fundamental das mônadas `Either` e `Option` é o curto-circuito. Quando uma operação falha em uma cadeia de computações que resulte em `Option` ou `Either`, as operações subsequentes não são executadas. Isso é especialmente útil em cenários onde várias validações ou transformações são encadeadas, pois evita o processamento desnecessário de dados inválidos. Esse comportamento é inerente às monadas de tratamento de erros, e está presente tanto em seu uso nas funções de dobra quanto nas compreensões de mônadas.

RESUMO As mônadas são um recurso poderoso para lidar com computações

que podem falhar ou que dependem de um estado. Elas permitem encadear operações de forma segura e expressiva, evitando a necessidade de verificações manuais de erros ou estados intermediários. Além disso, a notação de compreensão de mônadas torna o código mais legível e fácil de entender, permitindo que os desenvolvedores pensem nas operações de forma sequencial e intuitiva.

8.5 Padrões funcionais e bibliotecas

Os padrões funcionais são soluções recorrentes para problemas comuns em programação funcional, que enfatizam o uso de funções puras, imutabilidade e funções de primeira classe para resolver esses problemas. Podemos dizer que os padrões funcionais tem um papel semelhante aos padrões de projeto em orientação a objetos: melhorar a qualidade do código aplicando soluções reutilizáveis e composicionais. Os princípios fundamentais dos padrões funcionais incluem:

- **Pureza e previsibilidade.** Os padrões funcionais enfatizam o uso de funções puras, que não têm efeitos colaterais e sempre retornam o mesmo resultado para os mesmos argumentos. Isso torna o código mais previsível e fácil de entender.

- **Componibilidade.** Os padrões funcionais promovem a criação de funções pequenas e reutilizáveis que podem ser combinadas de várias maneiras para construir comportamentos mais complexos. Isso facilita a composição de funções e a criação de pipelines de dados.

- **Segurança.** Os padrões funcionais ajudam a garantir a segurança do código, uma vez que a imutabilidade e a ausência de efeitos colaterais reduzem a possibilidade de bugs e comportamentos inesperados.

Nós já estudamos alguns padrões funcionais simples, como o uso combinado de funções de ordem superior, como `map`, `filter` e `reduce`. Ou mesmo o projeto de algoritmos que permitam recursão na cauda. Temos também o uso de tipos opcionais, como `Option` e `Either`. Esses são alguns exemplos de padrões funcionais que podem ser aplicados em diferentes contextos.

Para a aplicação de padrões funcionais, podemos implementá-los de modo direto, e integrado ao problema específico que estamos resolvendo. Ou, alternativamente, especializar uma abstração genérica para o problema em questão. Aqui, adotaremos a segunda abordagem, utilizando uma biblioteca de programação funcional.

Bibliotecas de programação puramente funcional Uma biblioteca de programação puramente funcional visa prover abstrações e tipos de dados que podem ser utilizados para escrever código puramente funcional com segurança de tipos, componibilidade e ausência de efeitos colaterais. Por não ser uma linguagem puramente funcional, Scala não inclui essas abstrações em sua biblioteca padrão. No entanto, existem várias bibliotecas de terceiros que fornecem essas abstrações,

como Cats, Scalaz e ZIO. Essas bibliotecas oferecem uma ampla gama de tipos e abstrações funcionais, incluindo funções de ordem superior, tipos de dados imutáveis, e estruturas de controle para lidar com efeitos colaterais.

BIBLIOTECA CATS Uma das bibliotecas mais populares para programação puramente funcional em Scala é o *Cats* (*Category Theory for Scala*), que fornece uma série de abstrações, como funtores, monóides e mônadas.

INSTALAÇÃO DA CATS Para utilizar a biblioteca Cats, precisamos adicionar a dependência da biblioteca no nosso projeto. Se estivermos usando o SBT, podemos adicionar a seguinte linha ao arquivo `build.sbt`:

```
libraryDependencies += "org.typelevel" %% "cats-core" % "2.10.0"
```

Após efetuar o rebuild do projeto, podemos começar a utilizar a biblioteca Cats em nosso código.

8.6 Mônada de estados

Agora que já entendemos o conceito de mônadas, podemos aplicar esse conhecimento para resolver o problema de computações com estado. A mônada de estados é uma abstração que permite encapsular computações com estado de forma modular e reutilizável.

A composição de estados por meio de mônadas envolve a construção de computações complexas por meio do encadeamento de funções de mudança de estado. Esta técnica é mais poderosa que a agregação de estados, pois permite a construção de computações que podem ser combinadas de forma modular e reutilizável.

ESTADO COMO FUNÇÃO A ideia central é representar uma computação com estado como uma função do seguinte tipo:

```
S => (S, A)
```

onde S é o tipo do estado e A é o tipo do resultado da computação. A função $S => (S, A)$ recebe um estado do tipo S e retorna uma tupla com o novo estado do tipo S e o resultado da computação do tipo A. Ou seja, a função recebe um estado como argumento e retorna um resultado e um novo estado.

Os tipos monádicos que já estudamos, como `List`, `Option` e `Either`, foram implementados usando tipos algébricos de dados, ou seja, tipos produto e tipos soma. A mônada de estados, por outro lado, é implementada usando uma função que encapsula o estado. Essa função é um tipo de dados que representa uma computação com estado, e pode ser usada para compor computações de forma modular e reutilizável. Isso nos mostra que podemos utilizar diferentes tipos de dados para implementar mônadas, desde que satisfaçam as leis de mônadas.

8.6.1 Operações básicas

Considere o problema de manter um contador que avança ou regride a um passo configurável. Poderíamos definir o contador conforme o tipo produto do código 8.30.

Código 8.30: Definição do tipo `Counter`

```scala
case class Counter(count: Int, step: Int)
```

Definição do estado O primeiro passo para usar a mônada de estados é especializar o tipo do estado. No nosso caso, o estado é representado pelo tipo `Counter` e o valor das operações é um inteiro. Isso está refletido no código 8.31.

Código 8.31: Definição do tipo de estado

```scala
type CounterState[A] = State[Counter, A]
```

Operações de modificação de estado O próximo passo é definir as operações que modificam o estado. No nosso caso, temos duas operações: avançar e regredir o contador. Essas operações podem ser definidas conforme o código 8.32.

Código 8.32: Definição das operações de estado

```scala
object CounterOps {
    type CounterState[A] = State[Counter, A]

    def increment: CounterState[Int] = State { counter =>
        val newCounter = counter.copy(count = counter.count + counter.step)
        (newCounter, newCounter.count)
    }

    def decrement: CounterState[Int] = State { counter =>
        val newCounter = counter.copy(count = counter.count - counter.step)
        (newCounter, newCounter.count)
    }

    val get: CounterState[Int] = State.inspect(_.count)
}
```

No código 8.32, definimos o objeto `CounterOps` que encapsula as operações de modificação de estado para o contador. As operações `increment` e `decrement` são definidas como funções que retornam um valor do tipo `CounterState[Int]`, que é um alias para `State[Counter, Int]`. Usamos o método `State.apply` para criar essas operações. Esse método possui a seguinte assinatura:

```scala
def apply[S, A](f: S => (S, A)): State[S, A]
```

O método `State.apply` recebe uma função do tipo `Counter => (Counter, A)`, que recebe o estado atual do contador, aplica a operação de incremento ou decre-

mento, modifica e retorna o novo estado.

A operação `get` é definida por meio do método `State.inspect`, cuja assinatura é a seguinte:

```
def inspect[S, T](f: S => T): State[S, T]
```

Em resumo, o método recebe como parâmetro uma lambda que estrai o valor desejado do estado atual. O método `State.inspect` permite que criemos um estado que apenas lê o valor atual sem modificá-lo.

Sequência de modificações de estado Podemos criar uma sequência de operações que incrementa o contador, obtém seu valor e, em seguida, decrementa o contador. Essa sequência de operações pode ser representada como uma composição de estados usando as funções `flatMap` e `map`. Ou, mais convenientemente, usando uma compreensão `for`. O código 8.33 ilustra como podemos usar o objeto `CounterOps` para criar uma sequência de operações que modificam o estado do contador.

Código 8.33: Compondo estados com Cats State

```
@main def testCounter = {
    import CounterOps._

    val program = for {
        _ <- increment
        _ <- increment
        lastCount <- decrement
    } yield lastCount

    val (finalState, result) = program.run(Counter(0, 2)).value

    println(s"Final State: $finalState")
    println(s"Result: $result")
}
```

```
Final State: Counter(2,2)
Result: 2
```

No programa 8.33, definimos uma sequência de operações que incrementa o contador duas vezes, decrementa uma vez e, em seguida, obtém o valor atual do contador. A sequência de operações é representada como uma compreensão `for` que encadeia as operações usando `flatMap` e `map` implicitamente.

Computação adiada e `run` Cada operação do programa é encapsulada em uma função que retorna um valor do tipo `CounterState`. Isso permite que as operações sejam combinadas de forma mais flexível e composicional, sem a necessidade de executar a lógica imediatamente. Em vez disso, a execução real ocorre quando chamamos `run` no programa, passando o estado inicial do contador.

O método `run` efetivamente executa os seguintes passos:

- **Estado inicial**: Recebe um estado inicial (`Counter(0, 2)` neste caso)

- **Execução passo a passo**: Executa a computação do programa passo a passo:
  - Primeiro incremento: o estado torna-se `Counter(2, 2)`, retorna `2`
  - Segundo incremento: o estado torna-se `Counter(4, 2)`, retorna `4`
  - Decremento: o estado torna-se `Counter(2, 2)`, retorna `2`
- **Retorno**: Retorna uma tupla `(finalState, result)` onde:
  - `finalState` é o estado final do contador após todas as operações
  - `result` é o valor produzido pela última operação na compreensão `for`
- **Tipo de retorno**: `run` retorna `(Counter, Int)` encapsulado no tipo valor da mônada `State`
- **A chamada `.value`**: Extrai a tupla real do encapsulamento da mônada `State`

Como efeito prático, temos um programa puramente funcional que assemelha bastante a um programa imperativo, mas sem os efeitos colaterais. Isso é possível graças à separação entre a lógica de computação e a manipulação de estado.

Sequenciamento de operações A compreensão for é útil quando temos poucas operações que podem ser expressas explicitamente no código, uma a uma. Em muitas situações, precisamos processar uma coleção de operações que podem ser aplicadas a um estado. Outro resultado desejado é o armazenamento dos resultados intermediários das computações.

Quando vimos a abordagem de agregação de estados, exploramos a possibilidade de usar funções de dobra e suas variações para resolver os dois problemas. A mônada de estados permite que sequenciemos operações de forma mais natural, sem a necessidade de manipular listas ou coleções intermediárias. Para tal, temos as operações de percurso, dentre as quais vamos enfatizar a operação `sequence`.

Código 8.34: Sequenciamento de operações com `sequence`

```scala
@main def testSequence = {
    import CounterOps._
    import cats.syntax.traverse._

    val operations: List[CounterState[Int]] = List(
        increment,
        increment,
        decrement,
        increment,
        getValue
    )

    val program: CounterState[List[Int]] = operations.sequence
    val (finalState, results) = program.run(Counter(0, 3)).value

    println(s"Initial State: Counter(0, 3)")
    println(s"Final State: $finalState")
```

```
18      println(s"Results: $results")
19  }
```

```
Initial State: Counter(0, 3)
Final State: Counter(6,3)
Results: List(3, 6, 3, 6, 6)
```

Perceba que armazenamos todos as mônadas de estado em uma lista, permitindo que as executemos em sequência de forma mais simples. Após a execução do programa representado pela lista, obtemos o estado final do contador e os resultados de cada operação.

8.6.2 Transformadores de mônadas

Uma das grandes vantagens das mônadas é que elas permitem a composição de computações heterogêneas. Por exemplo, podemos ter um programa em que algumas computações falham. No modo tradicional (sem mônadas), precisaríamos verificar manualmente se cada computação falhou antes de prosseguir para a próxima. Com mônadas, podemos compor computações que podem falhar de forma mais elegante e segura. Considere o problema de manter um contador que só pode variar de valores dentro de uma faixa específica, conforme o código 8.35.

Código 8.35: Definição do tipo `BoundedCounter` e erros associados

```
1  case class BoundedCounter(count: Int, min: Int, max: Int, step: Int)
2
3  type CounterState[A] = State[BoundedCounter, A]
4
5  sealed trait CounterError
6  case object MaxBoundReached extends CounterError
7  case object MinBoundReached extends CounterError
8
9  object BoundedCounter {
10   def apply(
11       count: Int,
12       min: Int,
13       max: Int,
14       step: Int = 1
15   ): Either[CounterError, BoundedCounter] = {
16     if (count < min) Left(MinBoundReached)
17     else if (count > max) Left(MaxBoundReached)
18     else Right(new BoundedCounter(count, min, max, step))
19   }
20 }
```

O código define um tipo de dado `BoundedCounter` que representa um contador com limites mínimo e máximo, além de um passo de incremento. A função `apply` é usada para criar instâncias desse contador, garantindo que o valor inicial esteja dentro dos limites especificados. Se o valor estiver fora dos limites, um erro apropriado é retornado.

INCREMENTO E DECREMENTO COMO FUNÇÕES PARCIAIS Se o incremento ou decremento do contador resultar em um valor fora dos limites, a operação deve falhar. Isso é tratado pelas funções `increment` e `decrement` definidas no objeto `BoundedCounterOps`. Essas funções teriam as implementações conforme o codigo 8.36.

Código 8.36: Definição das operações de incremento e decremento

```scala
object BoundedCounterOpsUnsafe {
  def increment: State[BoundedCounter, Either[CounterError, Int]] =
    State { counter =>
      val newCount = counter.count + counter.step
      if (newCount > counter.max) {
        (counter, Left(MaxBoundReached))
      } else {
        val newCounter = counter.copy(count = newCount)
        (newCounter, Right(newCount))
      }
    }

  def decrement: State[BoundedCounter, Either[CounterError, Int]] =
    State { counter =>
      val newCount = counter.count - counter.step
      if (newCount < counter.min) {
        (counter, Left(MinBoundReached))
      } else {
        val newCounter = counter.copy(count = newCount)
        (newCounter, Right(newCount))
      }
    }

  def getValue: State[BoundedCounter, Either[CounterError, Int]] =
    State.inspect(counter => Right(counter.count))
}
```

O problema dessas assinaturas é que as mônadas `State` e `Either` ficam independentes uma da outra, ou melhor, os métodos `flatMap` não podem ser mutuamente encadeados. Na prática, teríamos uma compreensão para a mônada `State` e dentro dessa compreensão precisaríamos tratar manualmente os erros da mônada `Either`, como podemos observar no código 8.37.

Código 8.37: Cliente das operações do contador com tratamento manual de erros

```scala
@main def testBoundedCounterUnsafe = {
  BoundedCounter(8, 0, 10, 2) match {
    case Left(error) => println(s"Failed to create counter: $error")
    case Right(initialCounter) =>
      import BoundedCounterOpsUnsafe._

      val program = for {
        result1 <- increment
```

```
9           result2 <- if (result1.isRight) increment else State.pure(result1)
10          result3 <- if (result2.isRight) increment else State.pure(result2) //
            ↪ This should fail (10 -> 12)
11          result4 <- if (result3.isRight) increment else State.pure(result3) //
            ↪ This should also fail
12          currentCount <- if (result4.isRight) getValue else
            ↪ State.pure(Left(result4.left.getOrElse(MaxBoundReached)))
13        } yield currentCount
14
15        val (finalCounter, result) = program.run(initialCounter).value
16
17        println(s"Initial counter: $initialCounter")
18        result match {
19          case Left(error)  => println(s"Program failed with error: $error")
20          case Right(count) => println(s"Program succeeded with count: $count")
21        }
22        println(s"Final counter: $finalCounter")
23    }
24  }
```

```
Initial counter: BoundedCounter(8,0,10,2)
Program failed with error: MaxBoundReached
Final counter: BoundedCounter(10,0,10,2)
```

Perceba que para cada mudança de estado, precisamos lidar com os possíveis erros que podem ocorrer. Isso torna o código mais verboso e difícil de ler. Em cada verificação de erro, precisamos verificar se o resultado é um `Right` ou um `Left`, e então decidir se devemos continuar ou não. Como o lado direito de cada linha da compreensão é uma computação `State`, precisamos usar o método `State.pure` para encapsular o resultado em uma computação de estado, caso contrário, não poderíamos compor as operações. Para evitar essa prolixidade, usamos transformadores de mônadas.

Transformadores de mônadas Um transformador de mônadas é uma abstração que permite combinar duas ou mais mônadas para criar uma nova mônada que encapsula o comportamento de ambas. Alguns tipos de mônadas não são facilmente componíveis. Por exemplo, caso queiramos compor uma expressão que contenha tanto `State` quanto `Either`, não poderemos fazer isso diretamente, pois essas duas mônadas não são naturalmente compatíveis. Um transformador de mônadas resolve esse problema, criando uma mônada "híbrida" que combina os efeitos de ambas.

`EitherT` e `OptionT` A implementação de `State` em Cats inclui dois importantes transformadores de mônadas: `EitherT` e `OptionT`. Esses transformadores permitem combinar a funcionalidade de `Either` e `Option` com a funcionalidade de `State`, respectivamente. Esses transformadores são definidos conforme o seguinte:

- `OptionT[F, A]` encapsula uma computação `F` que pode não retornar um valor, retornando um valor do tipo `Option[A]`, onde `A` é o tipo do valor.

- `EitherT[F, E, A]` encapsula uma computação `F` que pode falhar, retornando um valor do tipo `Either[E, A]`, onde `E` é o tipo do erro e `A` é o tipo do valor.

Exemplos de uso O código 8.38 ilustra como podemos usar o transformador de mônadas `EitherT` para compor computações com estado e erros.

Código 8.38: Exemplo de uso de `EitherT`

```scala
object BoundedCounterOps {
  def increment: EitherT[CounterState, CounterError, Int] = EitherT {
    State { counter =>
      val newCount = counter.count + counter.step
      if (newCount > counter.max) {
        (counter, Left(MaxBoundReached))
      } else {
        val newCounter = counter.copy(count = newCount)
        (newCounter, Right(newCount))
      }
    }
  }

  def decrement: EitherT[CounterState, CounterError, Int] = EitherT {
    State { counter =>
      val newCount = counter.count - counter.step
      if (newCount < counter.min) {
        (counter, Left(MinBoundReached))
      } else {
        val newCounter = counter.copy(count = newCount)
        (newCounter, Right(newCount))
      }
    }
  }

  def getValue: EitherT[CounterState, CounterError, Int] =
    EitherT.liftF(State.inspect(_.count))
}
```

O construtor `EitherT` recebe um tipo de estado e um tipo de erro, e encapsula uma computação que pode falhar. As operações `increment` e `decrement` são definidas como funções que retornam um valor do tipo `EitherT[CounterState, CounterError, Unit]`. Essas operações verificam se o novo valor do contador está dentro dos limites especificados e retornam um erro apropriado caso contrário.

Agora podemos usar essas operações de forma composicional, aproveitando a estrutura do `EitherT` para lidar com erros de forma mais elegante. O código 8.39 ilustra como podemos usar o objeto `BoundedCounterOps` para criar uma sequência de operações que modificam o estado do contador.

Código 8.39: Compondo estados com `EitherT`

```scala
@main def testBoundedCounter = {
```

```
  BoundedCounter(8, 0, 10, 2) match {
    case Left(error) => println(s"Failed to create counter: $error")
    case Right(initialCounter) =>
      import BoundedCounterOps._

      val program = for {
        _ <- increment
        _ <- increment
        _ <- increment // This should fail (10 -> 12)
        currentCount <- increment // This will short circuit
      } yield currentCount

      val (finalCounter, result) = program.value.run(initialCounter).value

      println(s"Initial counter: $initialCounter")
      result match {
        case Left(error)  => println(s"Program failed with error: $error")
        case Right(count) => println(s"Program succeeded with count: $count")
      }
      println(s"Final counter: $finalCounter")
  }
}
```

```
Initial counter: BoundedCounter(8,0,10,2)
Program failed with error: MaxBoundReached
Final counter: BoundedCounter(10,0,10,2)
```

No código 8.39, definimos um cliente para as operações do contador que encapsula as operações de incremento e decremento em uma computação de estado. Isso nos permite compor essas operações de forma mais fácil e segura, sem precisar lidar manualmente com os erros. Perceba que, mesmo sem a presença de verificações manuais, o resultado da operação aponta o erro. Além disso, há uma otimização importante: a avaliação sofre "curto-circuito", isto é, se uma operação falhar, as operações subsequentes não serão executadas. Isso é possível graças à estrutura do `EitherT`, que encapsula a lógica de erro e permite compor computações de forma mais elegante. O mesmo não aconteceria no código 8.37.

8.6.3 Estudo de caso: conta bancária com mônada de estados

Continua...

Aviso de licença de uso