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, o 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

```scala
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`).

Como podemos observar, a versão imutável do contador é mais verbosa, pois precisamos criar novos objetos (e variáveis para armazena-los) a cada incremento. 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 estado. 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

```
package mutable {
  class Account(private var balance: Double = 0.0) {

    def getBalance: Double = balance

    def deposit(amount: Double): Double = {
      balance += amount
      balance
    }

    def withdraw(amount: Double): Double = {
      balance -= amount
      balance
    }
  }
}
```

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 retor-

nar 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 seguites 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.

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 `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

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.

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

```
1 case class AccountAccumulator(balances: List[Double], account: Account) {
2     def fold(operation: Account => (Double, Account)): AccountAccumulator = {
3       val (newBalance, newAccount) = operation(account)
4       AccountAccumulator(balances :+ newBalance, newAccount)
5     }
6 }
```

O tipo produto `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 `fold` 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 `fold` aplica a operação ao estado atual do acumulador, obtendo o novo estado da conta, e retorna um novo acumulador com esse novo estado. O nome `fold` não é uma escolha arbitrária, pois é tradicionalmente usado em funções agregadoras clássicas.

Com base nesse tipo produto `AccountAccumulator`, podemos reescrever o cliente funcional ingênuo de forma mais concisa, como no código 8.8.

Código 8.8: Cliente da conta bancária com acumulador

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

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 `fold` do acumulador. Cada chamada a `fold` 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.

NOTAÇÃO CONCISA 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. Assim, podemos definir a lista de operações de forma mais compacta, como no código 8.9. O efeito é o mesmo, mas a sintaxe é mais limpa e legível.

Código 8.9: Lista de operações com notação de máscara

```
val result = AccountAccumulator(List.empty[Double], Account())
    .fold(_.deposit(100.0))
```

```
    .fold(_.deposit(50.0))
    .fold(_.withdraw(30.0))
    .fold(_.deposit(40.0))
    .fold(_.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.

Para lidar de modo mais flexível com a agregação de estados, podemos usar funções agregadoras, 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.

8.3.2 Funções agregadoras

As funções agregadoras são uma abstração poderosa para processar coleções de dados, permitindo acumular resultados de maneira eficiente e expressiva. Elas operam em uma coleção, aplicando uma função acumuladora a cada elemento da coleção, produzindo um único resultado final. Desse modo, pode ser vista como uma generalização do padrão `Accumulator`. 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 acumuladora 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:

```
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): esse 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:

```
1 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

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

```
-15
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 Computação com estados e funções agregadoras

Agora que entendemos como as funções agregadoras 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 agregação 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 que o nosso problema:

- O "acumulador" é o estado atual da conta bancária (`currentAccount`), representado por um objeto `Account`.
- O valor inicial do acumulador é uma nova instância de `Account` com saldo inicial 0.0.

- 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, balanceReport) = operations.foldLeft(
    (Account(), List.empty[Double])) { (acc, operation) =>
        val (currentAccount, report) = acc
        val (newBalance, nextAccount) = operation(currentAccount)
        (nextAccount, report :+ newBalance)
}
println(s"Final balance: ${finalAccount.balance}")
println(s"Balance report: $balanceReport")
```

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

```
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

```
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.

Discussão A abordagem de agregação de estados com `foldLeft` e `scanLeft` oferece uma maneira 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.

Temos essa abordagem como uma das possibilidades para simplificar a manutenção de estado em programação funcional. No entanto, existem outras abordagens que podem ser mais adequadas dependendo do contexto e dos requisitos do problema. Por exemplo, podemos explorar o uso de mônadas de estado, que oferecem uma maneira mais estruturada de lidar com estados em programação funcional, ou até mesmo padrões mais complexos como o modelo de ator ou programação reativa funcional.

8.4 Composição de estados

Continua...

Aviso de licença de uso