Capítulo 5

Estruturas de dados funcionais

Definição Em programação funcional, o termo estrutura de dados funcional refere-se a estruturas que são imutáveis e projetadas para serem usadas em um estilo funcional. Ou seja, uma estrutura de dados puramente funcional possui as seguintes características:

- **Imutabilidade**. As estruturas de dados funcionais são imutáveis, ou seja, uma vez criadas, não podem ser alteradas. Qualquer modificação resulta em uma nova estrutura, preservando a original.
- **Persistência**. As estruturas de dados funcionais mantêm versões anteriores, permitindo acesso a estados anteriores sem custo adicional significativo. Isso é possível porque as modificações criam novas versões que compartilham partes da estrutura original.
- **Compartilhamento estrutural**. As estruturas de dados funcionais utilizam compartilhamento estrutural para evitar cópias desnecessárias. Isso significa que, ao criar uma nova versão de uma estrutura, apenas as partes modificadas são copiadas, enquanto as partes não alteradas são compartilhadas com a versão anterior.
- **Casamento de padrões**. Muitas estruturas de dados funcionais são projetadas para serem usadas com correspondência de padrões, o que facilita a extração de informações e a manipulação de dados de forma declarativa.
- **Funções de ordem superior**. As estruturas de dados funcionais frequentemente utilizam funções de ordem superior para operar sobre os dados, permitindo expressar operações como mapeamento, filtragem e redução de forma concisa e legível.

Estruturas funcionais em Scala Scala possui implementações de diversas estruturas funcionais, organizadas nas coleções imutáveis que são parte da biblioteca padrão. Essa biblioteca inclui tipos como `List`, `Set`, `Map` e `Vector`. Por padrão, quando utilizamos quaisquer dessas estruturas, estamos utilizando as implementações imutáveis. Caso desejemos utilizar as versões mutáveis, devemos importar

explicitamente o pacote `scala.collection.mutable`, que contém as versões mutáveis dessas mesmas coleções.

Estrutura de dados puramente funcional Uma estrutura de dados puramente funcional é aquela que compartilha as características mencionadas acima, além de garantir a transparência referencial das estruturas, isto é, a estrutura de dados não deve ter efeitos colaterais e deve sempre retornar o mesmo resultado para os mesmos argumentos.

Uma estrutura funcional pode prover uma API de estilo funcional mas ainda utilizar mutabilidade e efeitos colaterais (e.g. entrada e saída, tratamento de exceções, etc.) internamente. No entanto, para ser considerada puramente funcional, a estrutura deve garantir que todas as operações sejam realizadas de forma imutável e sem efeitos colaterais visíveis para o usuário.

As coleções imutáveis de Scala são um exemplo de estruturas de dados funcionais, mas não puramente funcionais. Por questões de desempenho e pragmatismo, algumas dessas estruturas podem utilizar mutabilidade e efeitos colaterais internamente, mas expõem uma API que atende a todas as características de uma estrutura de dados funcional. Eventuais anomalias oriundas da impureza são tratadas internamente e não afetam o usuário da coleção.

Vantagens de estruturas funcionais As estruturas de dados funcionais oferecem várias vantagens em relação às estruturas de dados tradicionais:

- **Segurança e previsibilidade**. A imutabilidade garante que os dados não serão alterados inesperadamente, o que reduz bugs e facilita a depuração.
- **Facilidade de raciocínio**. Como as estruturas são imutáveis, é mais fácil entender o comportamento do código, pois não há efeitos colaterais ocultos.
- **Concorrência segura**. A imutabilidade torna as estruturas de dados funcionais seguras para uso em ambientes concorrentes, pois não há risco de condições de corrida.
- **Composição fácil**. As estruturas funcionais são projetadas para serem compostas facilmente, permitindo a criação de pipelines de processamento de dados concisos e legíveis.

Desvantagens de estruturas funcionais Apesar das vantagens, as estruturas de dados funcionais também apresentam algumas desvantagens:

- **Desempenho**. Em alguns casos, as estruturas de dados funcionais podem ser menos eficientes em termos de desempenho do que suas contrapartes mutáveis, especialmente em operações que exigem muitas modificações.
- **Complexidade**. A implementação de estruturas de dados puramente funcionais pode ser mais complexa do que as implementações tradicionais, exigindo um entendimento mais profundo dos conceitos de programação funcional.

- **Curva de aprendizado**. Para desenvolvedores acostumados a paradigmas imperativos, a transição para estruturas de dados funcionais pode exigir uma curva de aprendizado significativa.

- **Sobrecarga de memória**. Embora o compartilhamento estrutural ajude a economizar memória, as estruturas funcionais podem consumir mais memória em alguns casos, especialmente quando muitas versões diferentes da mesma estrutura são mantidas.

5.1 Coleções em Scala

Scala oferece um conjunto amplo de implementações prontas de estruturas de dados, organizadas em uma hierarquia rica e unificada, que se estende pelos pacotes principais:

- `scala.collection` (Traits Abstratos): Este pacote define as traits (interfaces) que são os blocos de construção para todas as coleções, tanto mutáveis quanto imutáveis. Ele contém a maior parte da lógica e das operações que são comuns a diferentes tipos de coleções. Por exemplo, `Iterable`, `Seq`, `Set` e `Map` são definidos aqui.

- `scala.collection.immutable` (Implementações Imutáveis): Este pacote contém as implementações concretas das coleções imutáveis, que são o foco principal do Scala. Exemplos incluem `List`, `Vector`, `Set` e `Map`. São as mais recomendadas para a maioria dos casos de uso.

  `scala.collection.mutable` (Implementações Mutáveis): Aqui você encontra as implementações concretas das coleções mutáveis, como `ArrayBuffer`, `mutable.Map` e `mutable.Set`. Use-as com cautela e apenas quando a mutabilidade for realmente necessária.

  `scala.collection.concurrent` (Coleções Concorrentes): Este pacote contém coleções projetadas para serem usadas em ambientes concorrentes, como `TrieMap` e `ConcurrentHashMap`. Elas são otimizadas para acesso seguro em múltiplas threads.

Uma visão geral sobre as relações entre esses pacotes pode se encontrada na documentação oficial[1] da Scala. Nas próximas seções, vamos estudar algumas estruturas funcionais importantes presentes na biblioteca padrão do Scala. Essas coleções são amplamente utilizadas e fornecem uma base sólida para a programação funcional em Scala.

5.2 Estudo de caso: `immutable.List`

A classe `scala.immutable.List` é uma implementação concreta de uma lista encadeada funcional. A implementação concreta é baseada em lista encadeada simples.

[1] `https://docs.scala-lang.org/overviews/collections-2.13/overview.html`

Já vimos brevemente em capítulos anteriores como criar e manipular os componentes de uma lista. O código 5.1 mostra alguns exemplos de como criar e manipular listas.

Código 5.1: Exemplos de criação e manipulação de listas

```scala
// Creating lists with elements
val numbers = List(1, 2, 3, 4, 5)
val fruits = List("apple", "banana", "orange")
val booleans = List(true, false, true)

// Creating empty lists
val emptyList1: List[Int] = List()
val emptyList2: List[String] = Nil
val emptyList3 = List.empty[Double]

// Extracting head and tail
val firstNumber = numbers.head // 1
val remainingNumbers = numbers.tail // List(2, 3, 4, 5)

// Checking list length
val numbersLength = numbers.length // 5
val fruitsLength = fruits.size // 3 (size is equivalent to length)
val emptyLength = emptyList1.length // 0

// Adding elements to the beginning using cons (::) -- prepend
val originalList = List(2, 3, 4)

// Prepending single elements
val withOne = 1 :: originalList
val withZero = 0 :: withOne

// Prepending multiple elements
val withNegatives = -2 :: -1 :: withZero
println(s"After prepending -2 and -1: $withNegatives")

// Building a list from scratch using cons
val builtFromScratch = 1 :: 2 :: 3 :: Nil

// Adding elements to the end using append (:+)
val appendedList = originalList :+ 5 // List(2, 3, 4, 5)

// Concatenating two lists
val listA = List(1, 2, 3)
val listB = List(4, 5, 6)
val concatenatedList = listA ::: listB // List(1, 2, 3, 4, 5, 6)
val concatenatedList2 = listA ++ listB // List(1, 2, 3, 4, 5, 6)
```

IMUTABILIDADE E PERSISTÊNCIA A classe `List` é imutável, o que significa que qualquer operação que modifique a lista retorna uma nova lista, preservando a ori-

ginal. Por exemplo, ao adicionar um elemento no início da lista, a operação não altera a lista original, mas cria uma nova lista com o elemento adicionado. Isso permite que versões anteriores da lista sejam mantidas e acessadas facilmente. No exemplo do código 5.1, todas as operações de adição de elementos criam novas listas, preservando a lista original.

Eficiência de tempo das operações Por ser uma lista de encadeamento simples com acesso ao ponteiro do primeiro elemento, a classe `List` oferece operações eficientes para adicionar elementos no início da lista e acessar o primeiro elemento. No entanto, outras operações, como acessar o último elemento ou adicionar elementos no final da lista, são menos eficientes, pois exigem percorrer toda a lista. A tabela 5.1 resume a eficiência das principais operações na classe `List`.

Tabela 5.1: Eficiência das operações na classe `List`. Nos operadores ++ e :+, n é o tamanho do operando esquerdo.

Operação	Complexidade tempo
Acesso ao primeiro elemento (`head`)	$O(1)$
Acesso ao último elemento (`last`)	$O(n)$
Adicionar elemento no início (`::`)	$O(1)$
Adicionar elemento no final (`:+`)	$O(n)$
Concatenar listas (`++`)	$O(n)$
Remover primeiro elemento (`tail`)	$O(1)$
Remover último elemento (`init`)	$O(n)$
Buscar elemento (`contains`)	$O(n)$

Complexidade de espaço Devido às otimizações de compartilhamento estrutural, a classe `List` busca ser eficiente em termos de espaço. Por exemplo, quando utilizamos o operador cons (`::`) para adicionar um elemento no início da lista, a nova lista compartilha a mesma estrutura da lista original, economizando espaço. Por outro lado, operações que exigem a criação de uma nova lista, como adicionar elementos no final ou concatenar listas, podem consumir mais espaço, pois criam novas estruturas que não compartilham partes da lista original.

Em vista disso, determinar a complexidade de espaço de uma operação na classe `List` depende do tipo de operação realizada. Como mecanismo envolvido em cada um desses tipos de operações pode ser bastante complexo, vamos adiar a discussão desses custos quando estudarmos mais a fundo a implementação de estruturas de dados funcionais.

5.2.1 Processamento recursivo de listas

As listas são estruturas de dados com um papel muito importante na programação funcional. Vamos praticar nosso conhecimento em processamento de listas implementando alguns algoritmos recursivos comuns. Esses algoritmos são fundamentais para entender como manipular listas de forma funcional e eficiente.

Definição recursiva Uma lista encadeada possui uma definição essencialmente recursiva, conforme a fórmula 5.1.

$$\text{Lista} = \begin{cases} \texttt{Nil} & \text{se não houver elementos} \\ \texttt{head::tail} & \text{se houver elementos} \end{cases} \tag{5.1}$$

Onde:

- `Nil` representa uma lista vazia.
- `head` é o primeiro elemento da lista.
- `tail` é a lista restante, que pode ser vazia ou conter mais elementos.
- `::` é o operador de construção de listas, que combina a cabeça e a cauda para formar uma lista não vazia.

Nessa definição, o caso base é a lista vazia, que não possui cabeça nem cauda. O caso recursivo é a lista não vazia, que possui uma cabeça (o primeiro elemento da lista) e uma cauda (o restante da lista). Numa lista com um elemento, a cabeça é o único elemento e a cauda é a lista vazia.

Células cons A classe `List` é uma classe abstrata, portanto não pode ser instanciada diretamente. Essa classe possui duas classes concretas que a estendem: `Nil` e `Cons`. A implementação (simplificada) dessas classes é ilustrada no código 5.2.

Código 5.2: Implementação das classes `Nil` e `Cons`.

```scala
package scala.collection.immutable

sealed abstract class List[+A] extends Iterable[A] {
  val head: A
  val tail: List[A]

  // Outros métodos e definições...
}

final case class ::[+A](override val head: A, override val tail: List[A])
  extends List[A] {
  override def isEmpty: Boolean = false
  // Outros métodos e definições...
}

final case object Nil extends List[Nothing] {
  override def head: Nothing = throw new NoSuchElementException("head of
    empty list")
  override def tail: List[Nothing] = throw new NoSuchElementException("tail
    of empty list")
  // Outros métodos e definições...
}
```

Como podemos observar, tanto `Nil` quanto `::` estendem a classe abstrata `List`. A classe `Nil` representa uma lista vazia, enquanto a classe `::` representa uma célula cons, que contém um elemento (`head`) e uma referência para o restante da lista (`tail`). De fato, tanto `Nil` quanto `::` são o que comumente chamamos de "Nó" ou "Célula" de uma lista encadeada. Em suma, trabalhar com listas em Scala equivale a trabalhar diretamente com os "nós" da lista. A implementação dessas classes é fundamental para entender como as listas são construídas e manipuladas em Scala.

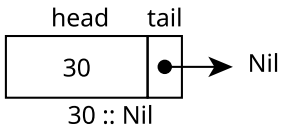

Figura 5.1: Representação gráfica da lista `30 :: Nil`

A figura 5.1 ilustra a representação gráfica da lista `30 :: Nil`. Nela, podemos ver que a célula cons (`::`) contém o valor 30 como cabeça (`head`) e uma referência para a lista vazia (`Nil`) como cauda (`tail`). Essa representação é fundamental para entender como as listas são construídas e manipuladas em Scala.

5.2.2 Tamanho da lista

Para determinar o tamanho de uma lista, a opção mais eficiente é armazenar uma constante com o esse valor no próprio TAD (Tipo Abstrato de Dados) da lista, no momento da criação. Porém, em Scala, as listas não são definidas como TADs, mas sim recursivamente, e não possuem uma propriedade que armazena o tamanho da lista. Podemos ver isso claramente na implementação da célula cons, onde não há nenhuma propriedade que armazena o tamanho da lista.

Por essa razão, para determinar o tamanho de uma lista, precisamos percorrer toda a lista. É isso exatamente o que o método `List.length` faz. Quando aplicamos esse método, ele inicia, a partir do nó atual, um percurso até o final da lista, contando quantos nós foram percorridos antes de atingir `Nil`. Essa operação é linear em relação ao tamanho da lista, ou seja, $O(n)$, onde n é o número de elementos na lista.

Para compreender melhor como esse método funciona, vamos implementar nossa própria função para calcular o tamanho de uma lista. A definição recursiva dessa operação é bastante simples, conforme a fórmula 5.2.

$$\text{tamanho}(\text{lista}) = \begin{cases} 0 & \text{se a lista for vazia} \\ 1 + \text{tamanho}(\text{tail}) & \text{se a lista não for vazia} \end{cases} \tag{5.2}$$

A implementação dessa função recursiva em Scala é apresentada no programa 5.3.

Código 5.3: Função recursiva para calcular o tamanho de uma lista.

```
1 package listSize
```

```
2
3  def listSize(xs: List[A]): Int = {
4    if (xs == Nil) 0
5    else 1 + listSize(xs.tail)
6  }
7
8  @main def testListSize(): Unit = {
9    println(listSize(Nil))
10   println(listSize(List(1, 2, 3)))
11   println(listSize(List("a", "b")))
12   println(listSize(List()))
13 }
```

```
0
3
2
0
```

Como podemos observar no programa 5.3, a função `listSize` é capaz de calcular o tamanho de listas de diferentes tipos, incluindo listas vazias, listas de números e listas de strings. O problema dessa implementação é que ela não é recursiva na cauda, pois a chamada recursiva ocorre após a soma com 1. Podemos facilmente converter essa função para uma versão recursiva na cauda, por meio do padrão Accumulator, conforme o programa 5.4.

Código 5.4: Função recursiva na cauda para calcular o tamanho de uma lista.

```
1  def listSizeTailRec(xs: List[Any], acc: Int = 0): Int = {
2    if (xs.isEmpty) acc
3    else listSizeTailRec(xs.tail, acc + 1)
4  }
5  @main def testListSizeTailRec(): Unit = {
6    println(listSizeTailRec(Nil))
7    println(listSizeTailRec(List(1, 2, 3)))
8    println(listSizeTailRec(List("a", "b")))
9    println(listSizeTailRec(List()))
10 }
```

```
0
3
2
0
```

A complexidade de tempo das duas versões do algoritmo é $O(n)$, onde n é o número de elementos na lista. A complexidade de espaço da versão não recursiva na cauda é $O(n)$, devido à profundidade da pilha de chamadas recursivas. Já a versão recursiva na cauda tem complexidade de espaço $O(1)$, pois não utiliza a pilha de chamadas para armazenar os resultados intermediários.

5.2.3 Soma dos elementos de uma lista

Para calcular a soma dos elementos de uma lista, o padrão de projeto clássico consiste em definir um redutor. Por exemplo, `List(1, 2, 3).reduce((a, b) => a + b)` retorna 6. Existe também um redutor especializado para somar os elementos de uma lista, que é `List(1, 2, 3).sum`, que também retorna 6. Esses redutores são implementados na classe `List` e utilizam a recursão interna para percorrer a lista e acumular a soma dos elementos.

No entanto, como estamos praticando a implementação de algoritmos recursivos, vamos implementar nossa própria função para calcular a soma dos elementos de uma lista. Podemos definir uma função recursiva que percorre a lista e acumula a soma dos elementos. A definição recursiva dessa operação é semelhante à do tamanho da lista, conforme a fórmula 5.3.

$$\text{soma}(\text{lista}) = \begin{cases} 0 & \text{se a lista for vazia} \\ \text{head} + \text{soma}(\text{tail}) & \text{se a lista não for vazia} \end{cases} \tag{5.3}$$

A implementação dessa função recursiva em Scala é apresentada no programa 5.5.

Código 5.5: Função recursiva para calcular a soma dos elementos de uma lista.

```scala
package ListSum

def listSum(xs: List[Int]): Int = {
  if (xs.isEmpty) 0
  else xs.head + listSum(xs.tail)
}

@main def testListSum(): Unit = {
  println(listSum(List(1, 2, 3)))
  println(listSum(List(4, 5, 6)))
  println(listSum(Nil))
  println(listSum(List(10)))
}
```

```
0
6
15
10
```

Mais uma vez, nossa primeira versão não é recursiva na cauda, pois a chamada recursiva ocorre após a soma com o elemento da cabeça da lista. O padrão Accumulator resolve o nosso problema, conforme o programa 5.6.

Código 5.6: Função recursiva na cauda para calcular a soma dos elementos de uma lista.

```scala
def listSumTailRec(xs: List[Int]): Int = {
  def loop(xs: List[Int], acc: Int = 0): Int = {
    if (xs.isEmpty) acc
```

```
    else loop(xs.tail, acc + xs.head)
  }
  loop(xs)
}

@main def testListSumTailRec(): Unit = {
  println(listSumTailRec(List(1, 2, 3)))
  println(listSumTailRec(List(4, 5, 6)))
  println(listSumTailRec(Nil))
  println(listSumTailRec(List(10)))
}
```

```
6
15
0
10
```

ADICIONAR ELEMENTO NO FINAL Adicionar um elemento no final de uma lista é uma operação que, na implementação padrão da classe `List`, não é eficiente, pois requer a criação de uma nova lista com o elemento adicionado. A classe List possui a operação `:+` para adicionar um elemento no final da lista, que tem complexidade $O(n)$, onde n é o número de elementos na lista.

Para ficar mais claro a razão dessa operação consumir tempo linear, basta visualizar em linhas gerais que o algoritmo precisa percorrer toda a lista para encontrar o último elemento, para só então adicionar o novo elemento. Além disso, como a lista é imutável, o algoritmo precisa criar uma nova lista contendo todos os elementos da lista original mais o novo elemento. Suponha que temos o seguinte trecho de código:

```
val xs = List(1, 2, 3)
val ys = xs :+ 4
```

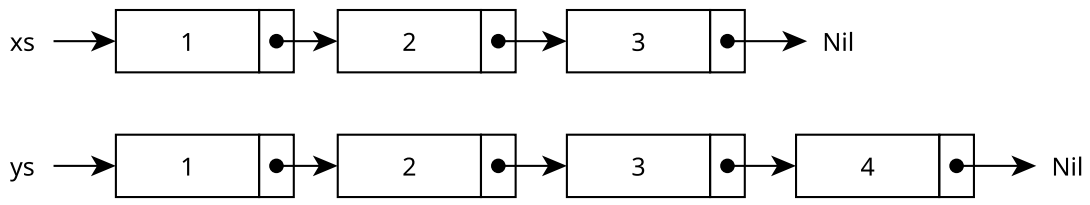

Figura 5.2: Adicionar elemento no final de uma lista

A figura 5.2 ilustra os conteúdos das listas `xs` e `ys` após a execução do código acima. A lista `xs` contém os elementos 1, 2 e 3. Quando aplicamos a operação `:+` para adicionar o elemento 4, o algoritmo percorre a lista `xs`, ao mesmo tempo em duplica todos os nós por onde passou. Quando encontra o último nó (3), o algoritmo cria um novo nó com o elemento 4 e o adiciona como cauda do último nó. Perceba que nessa operação, não há compartilhamento estrutural, pois o algoritmo cria uma nova lista com todos os elementos da lista original mais o novo elemento.

Com isso, essa operação tem complexidade de espaço $O(n)$, pois o algoritmo precisa criar uma nova lista com todos os elementos da lista original mais o novo elemento. A definição recursiva dessa operação é apresentada na fórmula 5.4.

$$\text{append}(\text{lista}, \text{elemento}) = \begin{cases} \text{Lista}(\text{elemento}) & \text{se a lista for vazia} \\ \text{head} :: \text{append}(\text{tail}, \text{elemento}) & \text{se a lista não for vazia} \end{cases} \tag{5.4}$$

A implementação dessa função recursiva em Scala é apresentada no programa 5.7.

Código 5.7: Função recursiva para adicionar um elemento no final de uma lista.

```scala
package append

def append[A](xs: List[A], elem: A): List[A] = {
  if (xs.isEmpty) List(elem)
  else xs.head :: append(xs.tail, elem)
}
@main def testAppend(): Unit = {
  println(append(List(1, 2, 3), 4)) // List(1, 2, 3, 4)
  println(append(List("a", "b"), "c")) // List("a", "b", "c")
  println(append(Nil, 42)) // List(42)
}
```

```
List(1, 2, 3, 4)
List(a, b, c)
List(42)
```

O traço desse algoritmo para a lista `List(1, 2, 3)` é o seguinte:

```
1 :: append(List(2, 3), 4)
  2 :: append(List(3), 4)
    3 :: append(Nil, 4)
      List(4)
    List(3, 4)
  List(2, 3, 4)
List(1, 2, 3, 4)
```

Essa implementação percorre toda a lista, adicionando o elemento no final. A complexidade de tempo dessa operação é $O(n)$, onde n é o número de elementos na lista. Discutimos acima que o resultado dessa operação é uma nova lista, com todos os elementos da lista original mais o novo elemento, o que nos dá uma complexidade de espaço também $O(n)$. Além disso, outro fator que caracteriza a complexidade de espaço dessa operação é a profundidade da pilha de chamadas recursivas, que também é $O(n)$, pois cada chamada recursiva adiciona um novo quadro na pilha.

Podemos diminuir o fator constante da complexidade de espaço, tornando a função recursiva na cauda. Para isso, podemos utilizar um acumulador para armazenar os elementos da lista enquanto percorremos a lista original. A implementação dessa função recursiva na cauda é apresentada no programa 5.8.

Código 5.8: Função recursiva na cauda para adicionar um elemento no final de uma lista.

```scala
def appendTailRec[A](xs: List[A], elem: A): List[A] = {
  @scala.annotation.tailrec
  def loop(xs: List[A], acc: List[A]): List[A] = {
    if (xs.isEmpty) (elem :: acc).reverse
    else loop(xs.tail, xs.head :: acc)
  }
  loop(xs, Nil)
}
@main def testAppendTailRec(): Unit = {
  println(appendTailRec(List(1, 2, 3), 4)) // List(1, 2, 3, 4)
  println(appendTailRec(List("a", "b"), "c")) // List("a", "b", "c")
  println(appendTailRec(Nil, 42)) // List(42)
}
```

```
List(1, 2, 3, 4)
List(a, b, c)
List(42)
```

Essa versão recursiva na cauda utiliza um acumulador para construir a lista de forma eficiente. Note um detalhe importante: diferentemente da versão ingênua, essa versão não adiciona o elemento no final da lista, mas sim no início do acumulador. Isso é possível porque a ordem dos elementos não importa, já que estamos apenas adicionando um único elemento no final da lista. No entanto, para manter a ordem correta dos elementos, precisamos inverter o acumulador ao final do processamento, utilizando o método `reverse`.

Embora o método `reverse` exiga mais uma passada completa na lista, a complexidade de tempo permanece $O(n)$, onde n é o número de elementos na lista original. Ao final, a complexidade da operação fica:

- Complexidade de tempo: $O(n)$, onde n é o número de elementos na lista original.
- Complexidade de espaço: $O(n)$, devido à criação de uma nova lista com todos os elementos da lista original mais o novo elemento.

INVERTER UMA LISTA Inverter uma lista é uma operação comum em programação funcional. Por exemplo, no algoritmo de inserção no final da lista, recorremos à operação `List.reverse`, que gera uma lista com os elementos na ordem inversa, em tempo $O(n)$, onde n é o número de elementos na lista. Por exemplo, `List(1, 2, 3).reverse` retorna `List(3, 2, 1)`. Note que `reverse` é um método sem parâmetros, ou seja, podemos omitir os parênteses da chamada.

Considere que efetuamos, em Scala, a seguinte operação:

```scala
val xs = List(1, 2, 3)
val ys = xs.reverse
```

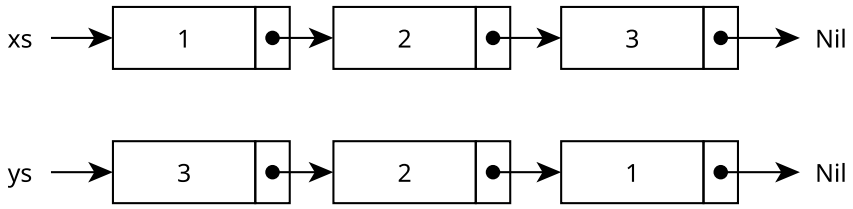

Figura 5.3: Inverter uma lista

A figura 5.3 ilustra os conteúdos das listas `xs` e `ys` após a execução do código acima. A lista `xs` contém os elementos 1, 2 e 3. Quando aplicamos a operação `reverse`, o algoritmo percorre a lista `xs`, ao mesmo tempo em que duplica todos os nós por onde passou. Quando chega ao último nó (3), o algoritmo cria uma nova lista com os elementos na ordem inversa, resultando na lista `ys` com os elementos 3, 2 e 1. Note que, de modo análogo ao que ocorre na operação de adicionar elemento no final, não há compartilhamento estrutural entre as listas `xs` e `ys`. Ou seja, a lista `ys` é uma nova lista, com todos os elementos da lista original na ordem inversa.

Implementação ingênua. Para praticar a implementação dessa operação "do zero", podemos partir da definição recursiva dessa operação, conforme a fórmula 5.5.

$$\text{inverter}(\text{lista}) = \begin{cases} \text{Nil} & \text{se a lista for vazia} \\ \text{inverter}(\text{tail}) :+ \text{head} & \text{se a lista não for vazia} \end{cases} \tag{5.5}$$

Nessa fórmula, o operador `:+` é o operador nativo para inserir no final lista. A implementação dessa função recursiva em Scala é apresentada no programa 5.9.

Código 5.9: Função recursiva para inverter uma lista.

```
def reverseNaive[A](xs: List[A]): List[A] = {
  if (xs.isEmpty) Nil
  else reverseNaive(xs.tail) :+ xs.head
}

@main def testReverseNaive(): Unit = {
  println(reverseNaive(List(1, 2, 3))) // List(3, 2, 1)
  println(reverseNaive(List("a", "b", "c"))) // List(c, b, a)
  println(reverseNaive(Nil)) // Nil
}
```

```
List(3, 2, 1)
List(c, b, a)
Nil
```

Eficiência da implementação ingênua. A implementação ingênua da função `reverseNaive` tem complexidade de tempo $O(n^2)$, onde n é o número de elementos na lista. Isso ocorre porque a operação `:+` percorre toda a lista parcial para adicionar o elemento no final. Podemos constatar isso quando analisamos o traço da pilha de execução da função `reverseNaive` para a lista `List(1, 2, 3)`.

```
reverseNaive(List(1, 2, 3))
  reverseNaive(List(2, 3)) :+ 1
    reverseNaive(List(3)) :+ 2
      reverseNaive(Nil) :+ 3
        Nil :+ 3
      List(3) :+ 2
    List(3, 2) :+ 1
  List(3, 2, 1)
```

No processo de desempilhamento (*unwinding*), o operador de inserção no final (`:+`) percorre toda a lista parcial para adicionar o elemento da cabeça. No total, o operador percorre $0 + 1 + 2 + ... + (n - 1) \approx O(n^2)$ elementos, onde n é o número de elementos na lista.

Versão iterativa. Para melhorarmos esse desempenho podemos mudar a ordem de construção da lista. O problema principal do algoritmo recursivo ingênuo é que a lista é encadeada ao desenrolar da pilha, o que nos obriga a recorrer à operação custosa de adicionar elementos no final da lista. Podemos evitar essa operação utilizando uma abordagem iterativa, que constrói a lista na ordem correta desde o início, utilizando o operador cons (`::`). A figura 5.4 ilustra a intuição desse algoritmo.

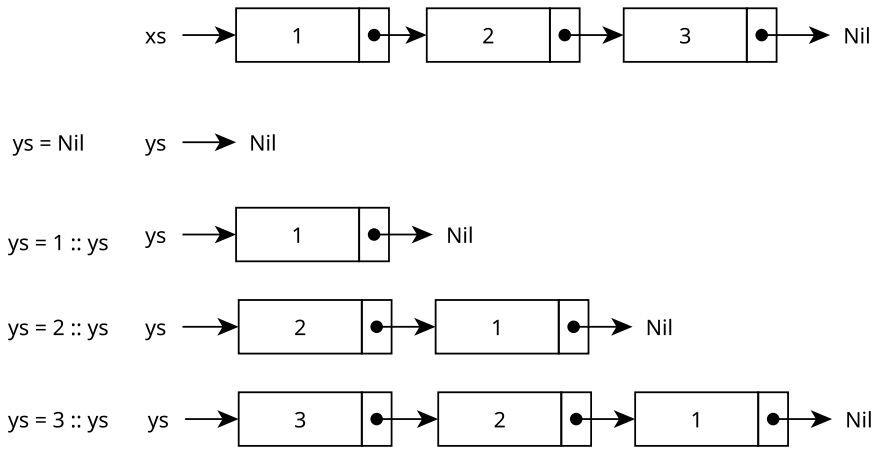

Figura 5.4: Inverter uma lista de forma iterativa

O processo consiste em ter uma variável mutável (`ys`), que inicia com a lista vazia (`Nil`), e iterar sobre os elementos da lista original (`xs`). A cada iteração, o elemento atual é adicionado no início da lista `ys` utilizando o operador cons (`::`). Dessa forma, ao final do processo, a lista `ys` conterá os elementos da lista original na ordem inversa. O código 5.10 apresenta uma versão iterativa da função `reverse`.

Código 5.10: Função iterativa para inverter uma lista.

```
1 def reverseIterative[A](xs: List[A]): List[A] = {
2   var ys: List[A] = Nil
3   var remaining = xs
4   while (!remaining.isEmpty) {
5     ys = remaining.head :: ys
```

```
    remaining = remaining.tail
  }
  ys
}
@main def testReverseIterative(): Unit = {
  println(reverseIterative(List(1, 2, 3))) // List(3, 2, 1)
  println(reverseIterative(List("a", "b", "c"))) // List(c, b, a)
  println(reverseIterative(Nil)) // Nil
}
```

```
List(3, 2, 1)
List(c, b, a)
Nil
```

Como já fizemos outras vezes, podemos facilmente converter essa função iterativa para uma versão recursiva na cauda, conforme o programa 5.11.

Código 5.11: Função recursiva na cauda para inverter uma lista.

```
def reverseTailRec[A](xs: List[A]): List[A] = {
  def loop(remaining: List[A], ys: List[A]): List[A] = {
    if (remaining.isEmpty) ys
    else loop(remaining.tail, remaining.head :: ys)
  }
  loop(xs, Nil)
}
@main def testReverseTailRec(): Unit = {
  println(reverseTailRec(List(1, 2, 3))) // List(3, 2, 1)
  println(reverseTailRec(List("a", "b", "c"))) // List(c, b, a)
  println(reverseTailRec(Nil)) // Nil
}
```

```
List(3, 2, 1)
List(c, b, a)
Nil
```

A complexidade de tempo dessa operação é $O(n)$, onde n é o número de elementos na lista. A complexidade de espaço também é $O(n)$, devido à criação de uma nova lista com os elementos da lista original na ordem inversa. A profundidade da pilha de chamadas recursivas é $O(1)$, pois a função é recursiva na cauda e não utiliza a pilha para armazenar os resultados intermediários.

CONCATENAR DUAS LISTAS Concatenar duas listas é uma operação comum em programação funcional. Em Scala, a operação `List.concat` ou o operador `++` podem ser usados para concatenar duas listas. Por exemplo, `List(1, 2) ++ List(3, 4)` retorna `List(1, 2, 3, 4)`. Considere que efetuamos, em Scala, a seguinte operação:

```
val xs = List(1, 2)
val ys = List(3, 4)
val zs = xs ++ ys
```

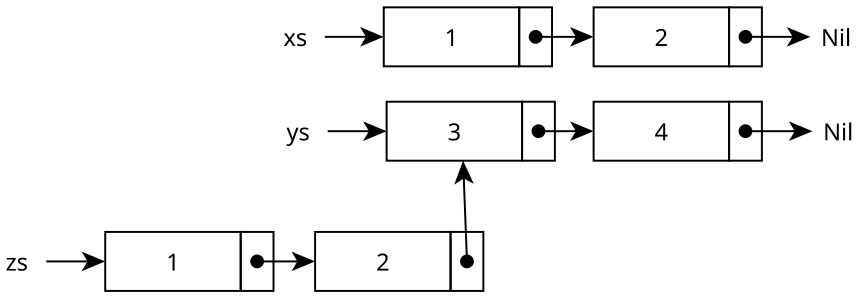

Figura 5.5: Concatenar duas listas

A figura 5.5 ilustra os conteúdos das listas `xs`, `ys` e `zs` após a execução do código acima. A lista `xs` contém os elementos 1 e 2, enquanto a lista `ys` contém os elementos 3 e 4. Quando aplicamos a operação de concatenação (++), o algoritmo percorre a lista `xs`, ao mesmo tempo em que duplica todos os nós por onde passou. Em seguida, ele adiciona a lista `ys` como cauda da lista resultante. Note que, diferentemente das operações anteriores, há compartilhamento estrutural, mesmo que parcial. Embora `xs` seja duplicada, a lista `ys` não é duplicada, mas sim ligada ao final da lista resultante. Isso significa que a lista `zs` é uma nova lista, com todos os elementos da lista `xs` mais os elementos da lista `ys`, mas sem duplicar os elementos da lista `ys`.

VERSÃO RECURSIVA CLÁSSICA Para concatenar duas listas, podemos pensar no seguinte algoritmo recursivo:

$$\text{concat}(\text{lista1}, \text{lista2}) = \begin{cases} \text{lista2} & \text{se a lista1 for vazia} \\ \text{lista1.head} :: \text{concat}(\text{lista1.tail}, \text{lista2}) & \text{se a lista1 não for vazia} \end{cases} \tag{5.6}$$

A implementação dessa função recursiva em Scala é apresentada no programa 5.12.

Código 5.12: Função recursiva para concatenar duas listas.

```scala
package concat

def concatenate[A](xs: List[A], ys: List[A]): List[A] = {
  if (xs.isEmpty) ys
  else xs.head :: concatenate(xs.tail, ys)
}
@main def testConcatenate(): Unit = {
  println(concatenate(List(1, 2), List(3, 4))) // List(1, 2, 3, 4)
  println(concatenate(List("a", "b"), List("c", "d"))) // List(a, b, c, d)
  println(concatenate(Nil, List(42))) // List(42)
}
```

```
List(1, 2, 3, 4)
List(a, b, c, d)
```

```
List(42)
```

O traço desse algoritmo para a lista `List(1, 2)` e `List(3, 4)` é o seguinte:

```
concatenate(List(1, 2), List(3, 4))
  1 :: concatenate(List(2), List(3, 4))
    2 :: concatenate(Nil, List(3, 4))
      List(3, 4)
    List(2, 3, 4)
  List(1, 2, 3, 4)
```

Essa implementação percorre toda a primeira lista, adicionando os elementos da cabeça na nova lista. A complexidade de tempo dessa operação é $O(n)$, onde n é o número de elementos na primeira lista. A complexidade de espaço é $O(n)$, devido à profundidade da pilha de chamadas recursivas e à criação de uma nova lista.

VERSÃO ITERATIVA Podemos melhorar a eficiência dessa operação utilizando uma abordagem iterativa, que constrói a lista na ordem correta desde o início, utilizando o operador cons (`::`). A figura 5.6 ilustra a intuição desse algoritmo.

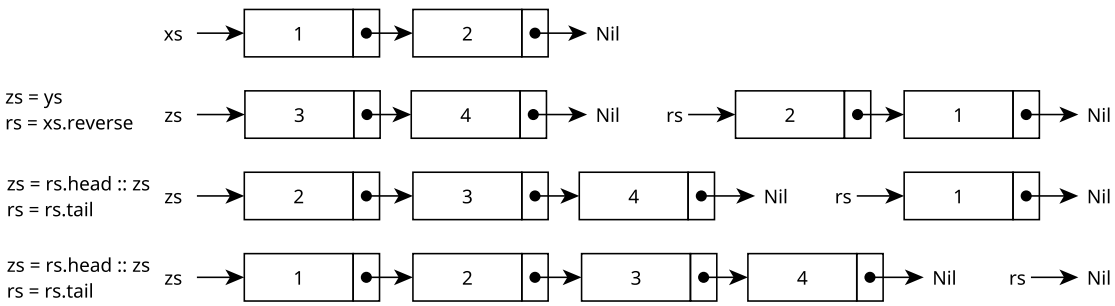

Figura 5.6: Concatenar duas listas de forma iterativa

O processo consiste em ter uma variável mutável (`zs`), que inicia com a lista da direita (`ys`), e uma variável mutável (`rs`) que inicia com a lista da esquerda (`xs`). Em seguida, o algoritmo entra em um laço de repetição (`while`) para percorrer a lista da esquerda (`rs`) até que ela esteja vazia. A cada iteração, o elemento atual da cabeça da lista `rs` é adicionado no início da lista `zs` utilizando o operador cons (`::`). Em seguida, o algoritmo atualiza a lista `rs` para a cauda da lista atual. Dessa forma, ao final do processo, a lista `zs` conterá os elementos da lista original na ordem correta. O código 5.13 apresenta uma versão iterativa da função `concatenate`.

Código 5.13: Função iterativa para concatenar duas listas.

```
1 def concatenateIterative[A](xs: List[A], ys: List[A]): List[A] = {
2   var zs = ys
3   var rs = xs.reverse
4   while (rs.nonEmpty) {
5     zs = rs.head :: zs
6     rs = rs.tail
7   }
```

```
  zs
}
@main def testConcatenateIterative(): Unit = {
  println(concatenateIterative(List(1, 2), List(3, 4))) // List(1, 2, 3, 4)
  println(concatenateIterative(List("a", "b"), List("c", "d"))) // List(a, b,
  ↪  c, d)
  println(concatenateIterative(Nil, List(42))) // List(42)
}
```

```
List(1, 2, 3, 4)
List(a, b, c, d)
List(42)
```

VERSÃO RECURSIVA NA CAUDA Podemos facilmente converter essa função iterativa para uma versão recursiva na cauda, utilizando o padrão Accumulator, conforme o programa 5.14.

Código 5.14: Função recursiva na cauda para concatenar duas listas.

```
def concatenateTailRec[A](xs: List[A], ys: List[A]): List[A] = {
  @scala.annotation.tailrec
  def loop(rs: List[A], zs: List[A]): List[A] = {
    if (rs.isEmpty) zs
    else loop(rs.tail, rs.head :: zs)
  }
  loop(xs.reverse, ys)
}
@main def testConcatenateTailRec(): Unit = {
  println(concatenateTailRec(List(1, 2), List(3, 4))) // List(1, 2, 3, 4)
  println(concatenateTailRec(List("a", "b"), List("c", "d"))) // List(a, b,
  ↪  c, d)
  println(concatenateTailRec(Nil, List(42))) // List(42)
}
```

```
List(1, 2, 3, 4)
List(a, b, c, d)
List(42)
```

Essa versão recursiva na cauda utiliza um acumulador para construir a lista de forma eficiente. Para que a construção, usando o operador cons (`::`), ocorra na ordem correta, precisamos inverter a primeira lista antes de concatená-la com a segunda lista. A complexidade de tempo permanece $O(n)$, onde n é o número de elementos na primeira lista. A complexidade de espaço da pilha é reduzida para $O(1)$, pois não há profundidade de pilha adicional. Porém, ainda precisamos considerar que precisamos duplicar a lista da esquerda, o que nos mantém em uma complexidade de espaço $O(n)$, onde n é o número de elementos na primeira lista.

COMPARTILHAMENTO ESTRUTURAL Como discutimos na implementação das operações, algumas delas envolvem compartilhamento estrutural, enquanto outras não. Em particular, conseguimos obter compartilhamento estruturas nas operações

que envolvem modificações no início da lista, como:

- Adicionar elemento no início da lista (`::`).
- Extrair a cauda da lista (`tail`).
- Concatenação de duas listas (`++` ou `:::`).

Já as outra operações (`append` ou `:+`, `reverse`, etc.) não conseguem compartilhar estrutura, devido à própria natureza da operação.

RESUMO DAS COMPLEXIDADES A tabela 5.2 resume as complexidades de tempo e espaço das operações discutidas neste capítulo.

Operação	Complexidade de Tempo	Complexidade de Espaço
Tamanho da lista	$O(n)$	$O(n)$
Soma dos elementos	$O(n)$	$O(n)$
Adicionar elemento no final	$O(n)$	$O(n)$
Inverter lista	$O(n)$	$O(n)$
Concatenar duas listas	$O(n)$	$O(n)$

Tabela 5.2: Complexidades das operações em listas. Na operação de concatenar duas listas, n é o número de elementos na primeira lista.

5.3 Outras estruturas em `scala.collection.immutable`

Scala possui outras estruturas bastane interessantes na biblioteca padrão, como `Set`, `Map` e `ArraySeq` e `Vector`. Vamos analisar em mais detalhes as duas últimas.

5.3.1 `ArraySeq`

`ArraySeq` é uma estrutuda de dados que, além de imutável, também é indexável — isto é, permite acesso aleatório em tempo constante ($O(1)$) aos seus elementos. Ela é implementada como um *wrapper* em torno de um array mutável de Java, o que permite acesso rápido aos elementos, mas sem a mutabilidade dos arrays tradicionais.

OPERAÇÕES BÁSICAS A manipulação básica de ArraySeq, a princípio, é similar à de uma lista. Considere o código 5.15.

Código 5.15: Operações básicas com `ArraySeq`.

```scala
import scala.collection.immutable.ArraySeq

val numbers = ArraySeq(1, 2, 3, 4, 5)
println(numbers(0)) // Acessa o primeiro elemento: 1
println(numbers(2)) // Acessa o terceiro elemento: 3
println(numbers.length) // Tamanho da ArraySeq: 5
println(numbers.isEmpty) // Verifica se a ArraySeq está vazia: false
```

```
println(numbers.contains(3)) // Verifica se o elemento 3 está presente: true
println(numbers.indexOf(4)) // Encontra o índice do elemento 4: 3
println(numbers.lastIndexOf(2)) // Encontra o último índice do elemento 2: 1
```

```
1
3
5
false
true
3
1
```

Persistência Todas as operações providas por `ArraySeq` garantem sua imutabilidade e persistência. Por exemplo, considere o código 5.16.

Código 5.16: Exemplo de uso de `ArraySeq`.

```
import scala.collection.immutable.ArraySeq

val numbers = ArraySeq(1, 2, 3, 4, 5)
println(numbers) // ArraySeq(1, 2, 3, 4, 5)
val updatedNumbers = numbers.updated(2, 10)
println(updatedNumbers) // ArraySeq(1, 2, 10, 4, 5)
println(numbers) // ArraySeq(1, 2, 3, 4, 5)
```

```
ArraySeq(1, 2, 3, 4, 5)
ArraySeq(1, 2, 10, 4, 5)
ArraySeq(1, 2, 3, 4, 5)
```

Perceba que, ao atualizar o elemento na posição 2, o array original `numbers` permanece inalterado. A operação `updated` retorna uma nova instância de `ArraySeq` com o elemento atualizado, preservando a imutabilidade da estrutura original.

Ausência de compartilhamento estrutural Devido à simplicidade da sua implementação, isto é, um *wrapper* em torno de um array mutável, `ArraySeq` não possui compartilhamento estrutural. Cada operação que modifica a estrutura resulta na criação de uma nova instância, sem reutilização de elementos da instância original.

Complexidade de tempo e espaço A tabela 5.3 resume as complexidades de tempo e espaço das operações mais comuns em `ArraySeq`.

Como podemos observar na tabela, `ArraySeq` destaca-se basicamente pelo rápido acesso aleatório que, por ser basicamente um wrapper sobre um array, compartilha a mesma complexidade de tempo para acesso a elementos. No entanto, as operações de atualização, inserção e remoção são custosas, pois exigem a criação de uma nova instância da estrutura, o que resulta em complexidade linear tanto em tempo quanto em espaço.

Operação	Tempo	Espaço
Acesso a elemento	$O(1)$	$O(1)$
Atualização de elemento	$O(n)$	$O(n)$
Inserção no início	$O(n)$	$O(n)$
Inserção no final	$O(n)$	$O(n)$
Inserção no meio	$O(n)$	$O(n)$
Remoção de elemento	$O(n)$	$O(n)$
Concatenação com outra ArraySeq	$O(n)$	$O(n)$

Tabela 5.3: Complexidades das operações em `ArraySeq`.

5.3.2 `Vector`

`Vector` é uma estrutura de dados imutável bastante sofisticada, projetada para oferecer um bom equilíbrio entre eficiência de acesso aleatório e operações de modificação. Ela é implementada como uma árvore balanceada com alto fator de ramificação, o que permite acesso rápido aos elementos, além de suportar modificações eficientes.

Árvore balanceada A implementação de `Vector` é baseada numa *radix tree* balanceada com um fator de ramificação 32. Isso significa que cada nó interno pode armazenar 32 valores e pode ter até 32 filhos. Essa estrutura permite que a árvore possui uma altura logarítmica e baixa profundidade, o que resulta em acesso rápido aos elementos.

Operações básicas A manipulação básica de `Vector` é similar à de uma lista, mas com acesso aleatório eficiente. Considere o código 5.17.

Código 5.17: Operações básicas com `Vector`.

```scala
import scala.collection.immutable.Vector

val numbers = Vector(1, 2, 3, 4, 5)
println(numbers(0))
println(numbers(2))
println(numbers.length)
println(numbers.isEmpty)
println(numbers.contains(3))
println(numbers.indexOf(4))
println(numbers.lastIndexOf(2))

println(numbers.updated(2, 10))
println(numbers :+ 6)
println(6 +: numbers)
println(numbers ++ Vector(7, 8))
println(numbers)
```

```
1
3
5
```

```
false
true
3
1
Vector(1, 2, 10, 4, 5)
Vector(1, 2, 3, 4, 5, 6)
Vector(6, 1, 2, 3, 4, 5)
Vector(1, 2, 3, 4, 5, 7, 8)
Vector(1, 2, 3, 4, 5)
```

Complexidade das operações A tabela 5.4 resume as complexidades de tempo e espaço das operações mais comuns em `Vector`.

Operação	Tempo	Espaço
Acesso a elemento (`apply`)	$O(\log n)$	$O(1)$
Atualização de elemento (`updated`)	$O(\log n)$	$O(\log n)$
Inserção no início (`+:`)	$O(\log n)$	$O(\log n)$
Inserção no final (`:+`)	$O(\log n)$	$O(\log n)$
Inserção no meio (`patch`)	$O(\log n)$	$O(\log n)$
Remoção de elemento (`filterNot`)	$O(n)$	$O(n)$
Concatenação (`++`)	$O(\min(n1, n2))$	$O(\min(n1, n2))$

Tabela 5.4: Complexidades das operações em `Vector`. Na operação de concatenação, $n1$ é o número de elementos no primeiro `Vector` e $n2$ é o número de elementos no segundo `Vector`.

Como podemos observar na tabela, `Vector` oferece acesso aleatório eficiente, com complexidade logarítmica tanto para acesso quanto para atualização de elementos. Podemos dizer que as operações logarítimos "efetivamente constantes", pois o fator de ramificação é alto (32). Por exemplo $\log_3 2(1000000) = 4.64$, o que significa que, mesmo para listas com milhões de elementos, o tempo de acesso é muito baixo.

Quanto à complexidade de espaço, as operações de atualização e inserção também são logarítmicas, pois essa estrutura utiliza fortememente compartilhamento estrutural. Quando adicionamos elementos no início ou no final, ou atualizamos elementos, a estrutura necessita criar um novo ramo da árvore, que é ligado à árvore existente, preservando a imutabilidade e garantindo a persistência. Isso significa que as operações de modificação não duplicam toda a estrutura, mas apenas os ramos afetados, resultando em uma complexidade de espaço logarítmica.

Rationale Devido ao bom desempenho em suas operações, `Vector` costuma ser uma boa escolha para a maioria dos problemas que envolvem estruturas de dados imutáveis. Naturalmente, existem problemas em que uma lista encadeada ou um array imutável são mais adequados. Por exemplo, problemas que envolvem muitas inserções ou remoções no início da lista podem ser mais eficientes com listas encadeadas, enquanto problemas que exigem acesso rápido a elementos em posições específicas (com poucas alterações) podem se beneficiar de arrays imutáveis. No entanto, `Vector` oferece um bom balanço entre vantagens e desvantagens. Mais informações sobre o desempenho das operações de `Vector` (e outras coleções) podem

ser encontradas na documentação oficial Scala[2]

Outras estruturas imutáveis Recomenda-se ao leitor interessado, consultar a documentação oficial Scala[3] para conhecer outras estruturas imutáveis disponíveis na biblioteca padrão, como `Set`, `Map`, `Range` e `Queue`. Essas estruturas oferecem diferentes características e funcionalidades, permitindo que você escolha a mais adequada para suas necessidades específicas.

5.4 Tipos abstratos de dados: TADs imutáveis

Estruturas de dados são comumente modeladas como tipos abstratos de dados (TADs), que definem uma interface para manipulação dos dados, sem expor a implementação interna. Em resumo, um tipo abstrato de dados é composto por:

- Um conjunto de dados;
- Um conjunto de operações (interface ou API) abstratas que podem ser realizadas sobre esses dados.

Independência de implementação A principal característica de um TAD, que deriva do adjetivo "abstrato", é que a interface do TAD não revela aspectos de implementação, tanto dos dados quanto das operações. Isso permite que o TAD seja implementado de diferentes maneiras, desde que a interface seja respeitada. Por isso, dizemos quem um TAD é independente de implementação. Essa independência é fundamental para garantir a flexibilidade e a reutilização do código, pois permite que diferentes implementações sejam trocadas sem afetar o código que utiliza o TAD.

Ocultamento da informação O ocultamento da informação é uma característica importante dos TADs, pois permite que a implementação interna dos dados e das operações seja escondida do usuário. Isso significa que o usuário do TAD não precisa conhecer os detalhes de como os dados são armazenados ou como as operações são implementadas, apenas precisa saber como utilizá-los através da interface definida. O ocultamento da informação ajuda a reduzir a complexidade do código e a aumentar a segurança, pois impede que o usuário acesse ou modifique diretamente os dados internos.

Técnicas de implementação TADs podem ser implementados de diferentes modos, porém uma técnica bastante comum é a utilização de classes e objetos em linguagens que operam sob o paradigma da orientação a objetos. Em linguagens funcionais que promovem a imutabilidade, a orientação a objetos deve atender a algumas restrições, como discutiremos a seguir.

[2] `https://docs.scala-lang.org/overviews/collections-2.13/performance-characteristics.html`

[3] `https://docs.scala-lang.org/overviews/collections-2.13/immutable-collections.html`

5.4.1 Classes e objetos imutáveis

Um objeto imutável é aquele cujo estado não pode ser modificado após sua criação. Em suma:

- As propriedades do objeto são todas imutáveis;
- O objeto não provê quaisquer métodos mutadores, ou seja, métodos que alteram o estado do objeto.

Uma classe imutável é uma classe que define objetos imutáveis. Em Scala, a notação para criação de uma classe é bastante simples, como podemos ver no código 5.18.

Código 5.18: Definição de uma classe imutável para representar um par ordenado.

```scala
class Pair[T](val first: T, val second: T) {

    def copy(first: T = this.first, second: T = this.second): Pair[T] = {
        new Pair(first, second)
    }

    def name = "Pair"

    override def toString: String = s"${name}($first, $second)"
}

@main def testPair(): Unit = {
  val pair = new Pair(1, 2)
  println(pair) // Pair(1, 2)

  val modifiedPair = pair.copy(first = 3)
  println(modifiedPair) // Pair(3, 2)

  println(pair.first) // 1
  println(pair.second) // 2
}
```

```
Pair(1, 2)
Pair(3, 2)
1
2
```

Construtor primário Em Scala, definimos a propriedade de um objeto no construtor primário da classe. A classe `Pair` define um construtor primário que recebe dois parâmetros, `first` e `second`, ambos do tipo genérico `T`. Os parêmetros do construtor primário podem conter modificadores:

- `val` or `var`: na presença de desses modificadores, os parâmetros do construtor primário se tornam propriedades da classe. Se o modificador for `val`, a propriedade é imutável; se for `var`, a propriedade é mutável. Na ausência desses

modificadores, os parâmetros do construtor primário são apenas parâmetros do método construtor e não fazem parte do estado do objeto.

- `private` or `protected`: esses modificadores controlam a visibilidade das propriedades. Se um parâmetro for declarado como `private`, ele só pode ser acessado dentro da classe; se for `protected`, ele pode ser acessado dentro da classe e em suas subclasses. Na ausência desses modificadores, os parâmetros do construtor primário são públicos e podem ser acessados de fora da classe.

Em `Pair`, ambos parâmetros são declarados como *val*, o que significa que são imutáveis e não podem ser alterados após a criação do objeto.

IMUTABILIDADE E PERSISTÊNCIA As principais características que devem estar presentes em uma classe imutável são a imutabilidade e a persistência. A imutabilidade é garantida pelo fato de que as propriedades do objeto são todas privadas e imutáveis. Consequentemente, não é possível a existência de métodos mutadores. A persistência é garantida pelo fato de que a classe provê um método `copy`, que permite criar uma nova instância do objeto com algumas propriedades modificadas, preservando o estado original. Desse modo, podemos criar uma nova instância do objeto com as propriedades desejadas, sem alterar o estado do objeto original.

OCULTAMENTO DA INFORMAÇÃO Em linguagens orientadas o objeto e imperativas, como Java, é comum definir métodos acessores e mutadores para acessar e modificar as propriedades de um objeto. Por outro lado, quando trabalhamos com objetos imutáveis, esses métodos perdem grande parte de sua utilidade, pois não podemos modificar o estado do objeto. Note que não definimos modificadores de acesso para as propriedades `first` e `second`, consequentemente elas são públicas.

CÓDIGO NO CONSTRUTOR PRIMÁRIO Métodos mutadores podem também ser usados para validação de dados, mas em objetos imutáveis, essa validação deve ser feita diretamente no construtor primário da classe. Qualquer trecho de código que esteja no corpo de uma classe, mas fora dos métodos, é executado no construtor primário da classe. Isso significa que podemos realizar validações e inicializações de propriedades diretamente no construtor, garantindo que o objeto seja criado em um estado válido. Por exemplo, podemos garantir que os elementos do par sejam do tipo `Int` ou que não sejam nulos, conforme o código 5.19.

Código 5.19: Definição de uma classe imutável para representar um par ordenado de inteiros positivos.

```
1 class PositivePair(override val first: Int, override val second: Int) extends
  ↪  Pair[Int](first, second) {
2     require(first > 0 && second > 0, "Both elements must be positive
      ↪  integers")
3
4     override def name = "PositivePair"
5 }
6
7 @main def testPositivePair(): Unit = {
8   val pair = new PositivePair(1, 2)
```

```
  println(pair) // PositivePair(1, 2)
  val modifiedPair = pair.copy(first = 3)
  println(modifiedPair) // PositivePair(3, 2)
  println(pair.first) // 1
  println(pair.second) // 2
  // O seguinte código lançará uma exceção, pois os elementos não são
  ↪ positivos
  // val invalidPair = new PositivePair(-1, 2) // Lança
  ↪ IllegalArgumentException
}
```

```
PositivePair(1, 2)
PositivePair(3, 2)
1
2
```

No exemplo, utilizamos uma função auxiliar `require` para validar as propriedades do construtor. Essa função lança uma exceção se a condição não for satisfeita, garantindo que os objetos da classe `PositivePair` sempre terão propriedades válidas, ou seja, ambos os elementos serão inteiros positivos.

Herança Note que a classe `PositivePair` estende a classe `Pair`, o que significa que ela herda as propriedades e métodos da classe `Pair`. Uma consequência direta é que podemos utilizar o método `copy` da classe `Pair` para criar uma nova instância de `PositivePair` com algumas propriedades modificadas.

Sobrescrita de propriedades A classe `PositivePair` sobrescreve as propriedades `first` e `second` da classe `Pair` para garantir que elas sejam do tipo `Int`. Isso é necessário para que possamos validar as propriedades no construtor da classe. A sobrescrita é feita utilizando a palavra-chave `override`, que indica que estamos substituindo a implementação original da propriedade na classe mãe. Caso omitíssemos o modificador `override`, o compilador geraria um erro, pois estaríamos tentando redefinir uma propriedade com um nome que já definido na classe mãe.

Sobrescrita de métodos A classe `PositivePair` sobrescreve o método `name` da classe `Pair` para fornecer um nome mais específico para a classe. Isso é útil para identificar o tipo de objeto quando imprimimos ou registramos informações sobre ele. O efeito direto da sobrecrita ocorre na chamada do método `toString`, que utiliza o método `name` para formatar a saída. Assim, quando imprimimos um objeto da classe `PositivePair`, obtemos uma representação mais informativa, como `PositivePair(1, 2)`, embora o método `toString` não tenha sido explicitamente sobrescrito.

Opacidade referencial A validação das propriedades é feita diretamente no construtor da classe, utilizando a função `require`. Essa função é chamado durante o construtor primário da classe, e lança uma exceção se a condição não for satisfeita. Isso garante que os objetos da classe `PositivePair` sempre terão propriedades válidas, ou seja, ambos os elementos serão inteiros positivos.

Por outro lado, como já estudamos, o lançamento de exceções é considerado um efeito colateral, tornando o nosso contrutor referencialmente opaco. Por ora, de-

vemos estar cientes dessa limitação, porém posteriormente veremos técnicas mais declarativas para lidar com tratamento de erros, que não implicam em efeitos colaterais.

Vantagens de TADs imutáveis A utilização de TADs imutáveis traz todas as vantagens da imutabilidade que já discutimos extensivamente no curso: segurança em concorrência, facilidade de depuração, previsibilidade e simplicidade. Além disso, TADs imutáveis em muitos contextos podem ser compreendidos como estruturas de dados persistentes, pois permitem a criação de novas versões do objeto sem alterar o original. Isso é especialmente útil em aplicações que exigem histórico de mudanças ou versões, como sistemas de controle de versão, bancos de dados e aplicações reativas.

Em suma, para a programação funcional, TADs imutáveis devem ser a escolha padrão para modelagem de estruturas de dados. Eles oferecem uma maneira segura e eficiente de manipular dados, mantendo a integridade do estado e permitindo fácil compartilhamento e reutilização de estruturas. Veremos mais adiante que podemos também trabalhar com um conceito mais avançado e idiomático, que são os tipos algébricos de dados — cuja sigla também é TADs — que são uma forma mais poderosa de modelar dados complexos em programação funcional.

5.4.2 Contratos estruturais: *traits*

Contratos estruturais são uma forma de definir um conjunto de métodos que uma ou mais classes devem implementar. Em Scala, os contratos estruturais podem ser definidos por meio de *traits* ou de classes abstratas.

Um *trait* é uma coleção de métodos e propriedades que podem ser "misturados" a uma classe, permitindo que essa classe implemente o comportamento definido pelo *trait*. A principal vantagem de traits sobre classes abstratas é que uma classe pode estender múltiplos traits, enquanto pode estender apenas uma classe abstrata. Isso permite uma maior flexibilidade na definição de comportamentos e na reutilização de código.

`traits` e TAD API Na prática, utilizamos traits para definir a API de um TAD, ou seja, o conjunto de métodos que devem ser implementados por uma classe que estende o trait. Isso permite que diferentes classes implementem o mesmo comportamento de maneiras distintas, respeitando o contrato definido pelo trait. Isso garante a independência de implementação e o ocultamento da informação, pois a classe que estende o trait não precisa conhecer os detalhes de implementação dos métodos definidos no trait, apenas precisa implementá-los conforme o contrato.

Definição de um *trait* Um *trait* é definido utilizando a palavra-chave `trait`, seguida pelo nome e pelo corpo do trait. O código 5.20 apresenta um exemplo de definição de um trait para uma estrutura de dados do tipo pilha imutável.

Código 5.20: Definição e implementação de um *trait*

```
1 trait Stack[T] {
2   def push(x: T): Stack[T]
```

```
  def pop: (T, Stack[T])
  def isEmpty: Boolean
  def size: Int
}
```

Um trait pode conter métodos abstratos (sem implementação) e métodos concretos (com implementação). Para a nossa discussão, vamos focar apenas em métodos abstratos, que devem ser implementados pelas classes que estenderem o trait. No exemplo, definimos métodos típicos de uma pilha, como `push` (empilhar), `pop` (desempilhar), `isEmpty` (consulta se vazia) e `size` (consulta tamanho). Esses métodos devem ser implementados por qualquer classe que estenda o trait `Stack`. Por exemplo, considere o código 5.21.

Código 5.21: Implementação de um *trait* para pilha.

```
class ListStack[T] extends Stack[T] {
  override def push(x: T): Stack[T] = ???
  override def pop: (T, Stack[T]) = ???
  override def isEmpty: Boolean = ???
  override def size: Int = ???
}

class ArrayStack[T] extends Stack[T] {
  override def push(x: T): Stack[T] = ???
  override def pop: (T, Stack[T]) = ???
  override def isEmpty: Boolean = ???
  override def size: Int = ???
}
```

Múltiplas implementações No código 5.21, temos duas classes que implementam o trait `Stack`: `ListStack` e `ArrayStack`. Cada implementação, a princípio, utilizará uma estrutura de dados subjacente diferente para armazenar os elementos da pilha (lista e array, respectivamente). Por ora, omitimos as implementações dos métodos para focar na estrutura do código. A notação `???` indica ao compilador que os métodos ainda não foram implementados, mas que serão implementados posteriormente. Temos aqui a principal vantagem dos traits: a possibilidade de definir um contrato comum para diferentes implementações, permitindo que diferentes classes implementem o mesmo comportamento de maneiras distintas.

Sobrescrita obrigatória É fundamental que uma classe que estenda um trait implemente todos os métodos abstratos definidos no trait. Caso contrário, ocorrerá um erro de compilação. No nosso exemplo, tanto `ListStack` quanto `ArrayStack` devem implementar os métodos `push`, `pop`, `isEmpty` e `size`. A palavra-chave `override` é utilizada para indicar que estamos implementando um método definido no trait. Essa palavra-chave é opcional para implementação dos métodos abstratos (mas obrigatória para os métodos concretos), mas é uma boa prática utilizá-la para deixar claro que estamos implementando um método de um trait ou de uma superclasse: caso esse método não exista no trait, o compilador emitirá um erro, ajudando a evitar erros de digitação ou confusão na implementação.

5.5 Tratamento declarativo de erros

Em linguagens orientadas a objetos, o tratamento de erros é comumente feito por meio de exceções. No entanto, em programação funcional, preferimos evitar efeitos colaterais e utilizar abordagens mais declarativas para lidar com erros. Nas abordagens declarativas, ao invés de situações excepcionais, os erros são tratados como cidadãos de primeira classe, ou seja, são representados como valores na lógica formal do programa. Isso permite que o código seja mais previsível, testável e composicional. Por exemplo, considere o código 5.22.

Código 5.22: Tratamento de erros com exceções.

```scala
def divide(x: Int, y: Int): Int = {
  if (y == 0) throw new ArithmeticException("Division by zero")
  x / y
}
@main def testDivide(): Unit = {
  try {
    println(divide(10, 2)) // 5
    println(divide(10, 0)) // Lança ArithmeticException
  } catch {
    case e: ArithmeticException => println(e.getMessage) // Division by zero
  }
}
```

```
5
Division by zero
```

O exemplo acima define uma função `divide` que recebe dois inteiros e retorna o resultado da divisão. Se o divisor for zero, a função lança uma exceção `ArithmeticException`. No método `testDivide`, utilizamos um bloco `try-catch` para capturar a exceção e imprimir uma mensagem de erro.

O PROBLEMA DAS EXCEÇÕES Já discutimos no início do curso que as exceções são consideradas efeitos colaterais, devido aos seguintes problemas:

- **Interrupção do fluxo normal do programa**. Quando uma exceção é lançada, o fluxo de execução é interrompido e o controle é transferido para o bloco `catch` mais próximo. Isso pode dificultar a compreensão do programa pois expressões podem não ser avaliadas como esperado, e o fluxo de controle pode se tornar confuso.

- **Dificuldade de testagem**. Exceções podem ser difíceis de depurar, especialmente se não forem tratadas adequadamente. O rastreamento de pilha pode ser confuso e não fornecer informações suficientes sobre o contexto em que a exceção ocorreu.

- **Opacidade referencial**. O disparo de uma exceção pode prevenir uma função de gerar um valor válido dentro do seu contradomínio. Isso dificulta a componibilidade do programa. Por exemplo, a expressão `divide(10, 0)`

`+ divide(3, 2)` não pode ser avaliada, pois a primeira chamada à função `divide` lança uma exceção, impedindo que o resultado da segunda chamada seja calculado.

- **Complexidade adicional**. O uso de exceções pode tornar o código mais complexo, especialmente em programas grandes e fracamente modularizados. O tratamento de exceções pode exigir a adição de muitos blocos `try-catch` e a definição de exceções personalizadas, o que pode aumentar a complexidade do código e torná-lo mais difícil de entender.

- entre outros

Funções parciais e funções totais Em programação funcional, podemos classificar as funções em dois tipos:

- **Funções totais**: são aquelas que sempre retornam um valor válido dentro do seu contradomínio, ou seja, para todos os argumentos possíveis.

  Exemplos:

  - `def add(x: Int, y: Int): Int = x + y` (sempre retorna a soma de dois inteiros).
  - `def square(x: Double): Double = x * x` (sempre retorna o quadrado de um número real).

- **Funções parciais**: são aquelas que podem não retornar um valor válido para todos os argumentos possíveis, ou seja, existem casos em que a função não é definida ou não retorna um valor válido.

  Exemplos:

  - `def divide(x: Int, y: Int): Int = if (y == 0) throw new ArithmeticException("Division by zero") else x / y` (não retorna um valor válido quando o divisor é zero).
  - `def getElementAtIndex[T](list: List[T], index: Int): T = list(index)` (não retorna um valor válido se o índice estiver fora dos limites da lista).

Dentro de um programa, a presença de funções parciais trazem problemas. Por exemplo, se uma expressão contém uma função parcial, como `divide(10, 0)`, essa expressão não pode ser avaliada, pois a função não retorna um valor válido. Isso impede que o programa seja composto de forma previsível e segura, uma vez que o resultado de uma função parcial pode não existir em determinadas situações.

Dentro do paradigma imperativo, os casos de exceção das funções parciais são tratados com exceções, o que pode levar aos problemas que já discutimos. Para que possamos operar dentro do estilo puramente funcional, precisamos definir recursos que permitam converter funções parciais em funções totais.

Tratamento de erros com tipos Em programação funcional, preferimos utilizar tipos de dados especiais para representar os casos de exceção de uma função parcial. A definição desses tipos especiais pode ser feita para cada domínio de interesse, no entanto o mais comum é adotarmos padrões de projeto funcionais para esse problema. Dentre esses padrões de projeto, o uso de tipos opcionais e tipos de resultado são os mais comuns. Em Scala, podemos utilizar os tipos `Option` e `Either` para representar valores opcionais e resultados, respectivamente. Veremos mais sobre esses tipos nas próximas seções.

5.5.1 Tipos opcionais: `Option`

Os tipos opcionais em Scala são representados pelo tipo `Option`, cuja semântica permite lidar com a ausência de valores de forma segura e declarativa. Ao invés de utilizar `null`, ou similares, como fazemos em linguagens imperativas, o tipo `Option` nos permite representar a presença ou ausência de um valor de forma explícita.

Definição A definição simplificada do tipo `Option` consta no código 5.23.

Código 5.23: Definição simplificada do tipo `Option`.

```scala
sealed trait Option[+A]
case class Some[A](value: A) extends Option[A]
case object None extends Option[Nothing]
```

A definição do código 5.23 utiliza o conceito de tipos algébricos, que ainda não abordamos. No entanto, para os fins de nossa discussão, basta saber que um tipo abstrato `Option` só pode assumir dois tipos concretos:

- `Some[A]`: representa a presença de um valor do tipo `A`. O construtor `Some` recebe um valor do tipo `A` e o encapsula dentro de uma instância de `Option[A]`;
- `None`: representa a ausência de um valor. É um objeto singleton, ou seja, existe apenas uma instância de `None` em todo o programa. Ele é utilizado para indicar que não há um valor presente.

Declaração de tipos opcionais Para demonstrar a declaração de tipos opcionais, vamos adaptar o código 5.22 para utilizar o tipo `Option` no lugar de exceções. O código 5.24 apresenta essa adaptação.

Código 5.24: Tratamento de erros com tipos opcionais.

```scala
def divide(x: Int, y: Int): Option[Int] = {
  if (y == 0) None
  else Some(x / y)
}
@main def testDivideOption(): Unit = {
  println(divide(10, 2)) // Some(5)
  println(divide(10, 0)) // None
}
```

```
Some(5)
None
```

No código 5.24, a função `divide` agora retorna um valor do tipo `Option[Int]`. Se o divisor for zero, a função retorna `None`, indicando que não há um valor válido. Caso contrário, ela retorna `Some(x / y)`, encapsulando o resultado da divisão dentro de uma instância de `Option`.

Função total A função `divide` agora é uma função total, pois sempre retorna um valor do tipo `Option[Int]`, independentemente dos argumentos fornecidos. Isso significa que podemos compor essa função com outras funções de forma segura, sem nos preocuparmos com exceções ou interrupções no fluxo do programa. Naturalmente, mudamos o contradomínio da função de `Int` para `Option[Int]`: as situações de erro, que antes implícitas no corpo da função, agora fazem parte da assinatura função. Isso pode parecer uma desvantagem, pois precisamos lidar com o tipo `Option[Int]` ao invés de um valor direto do tipo `Int`. No entanto, essa mudança é intencional e necessária para garantir a segurança e a componibilidade do programa.

Componibilidade A principal vantagem de utilizar o tipo `Option` é que ele permite compor funções, mantendo a transparência referencial, mesmo em situações de erro. Por exemplo, suponha que queiramos avaliar a expressão a seguir:

$$x/z + x/y \tag{5.7}$$

Essa expressão não está definida caso `z` ou `y` sejam iguais a zero, pois o operador de divisão é uma função parcial. No entanto, suponha que, ao invés de utilizar o operador de divisão, utilizemos a função `divide` definida anteriormente. Nesse caso, podemos reescrever a expressão como:

$$divide(x, z) + divide(x, y) \tag{5.8}$$

Poderíamos implementar uma função que avalia essa expressão, como no código 5.25.

Código 5.25: Tentativa de composição insegura de funções.

```
1 def expression1(x: Int, y: Int, z: Int): Option[Int] = {
2   divide(x, y) + divide(x, z)
3 }
```

```
console error: value + is not a member of Option[Int]
  divide(x, y) + divide(x, z)
```

Como podemos observar no erro de compilação, a expressão `divide(x, y) + divide(x, z)` não é válida, pois o operador `+` não está definido para o tipo `Option[Int]`. Isso ocorre porque o operador `+` é um método da classe `Int`, e não do tipo `Option[Int]`. Portanto, a soma entre um Int e um `Option[Int]` não é válida. Para conseguir fazer isso, precisamos extrair o valor dentro do `Option` e realizar a operação.

MÉTODOS EXTRATORES Uma alternativa seria usar alguns dos métodos de extração do tipo `Option`, como `getOrElse`, para obter o valor dentro do `Option` e realizar a operação, conforme o código 5.26.

Código 5.26: Composição segura de funções com `Option` via métodos extratores.

```
def expression2(x: Int, y: Int, z: Int): Int = {
  divide(x, y).getOrElse(0) + divide(x, z).getOrElse(0)
}
@main def main(): Unit = {
  println(expression2(10, 2, 3))
  println(expression2(10, 2, 0))
}
```

```
8
5
```

O método `getOrElse` é um método extrator que permite obter o valor dentro do `Option`, ou um valor padrão caso o `Option` seja `None`. No exemplo, se a divisão resultar em `None`, o valor padrão 0 será utilizado na soma. Assim, a expressão `divide(10, 2) + divide(10, 0)` pode ser avaliada de forma segura, sem lançar exceções. Por outro lado, assumir um valor padrão zero quando a divisão estiver indefinida não é matematicamente preciso, pois a função está gerando um valor válido (5) para uma expressão que não está definida, ou seja, que não deveria ter valor.

Em suma, para esse problema, o mais adequado não seria assumir um valor padrão para as operações individuais, mas sim garantir que o resultado da expressão também seja do tipo `Option`. Nesse caso, quando avaliamos a expressão `divide(10, 0) + divide(10, 2)`, o resultado deve ser do tipo `None`, e não `20`. Para isso, precisamos reorganizar a avaliação da expressão para considerar a presença ou ausência de valores em cada operação.

OUTRAS FORMAS IDIOMÁTICAS A abordagem mais concisa e idiomática para lidar com expressões envolvendo tipos opcionais é utilizar os métodos de ordem superior do tipo `Option`, como `map` e `flatMap`. Veremos adiante mais sobre esses métodos, no contexto do estudo das mônadas. Aqui, apenas daremos uma prévia para o leitor analisar como podemos processar valores opcionais de modo mais elegante e seguro. Por exemplo, podemos reescrever a expressão 5.8 utilizando os métodos `flatMap` e `map` do tipo `Option`, conforme o código 5.27.

Código 5.27: Composição segura de funções com `Option` via padrão de `flatMaps` a `map` aninhados.

```
def expression4(x: Int, y: Int, z: Int): Option[Int] = {
  divide(x, y).flatMap { quotient1 =>
    divide(x, z).map { quotient2 =>
      quotient1 + quotient2
    }
  }
}
@main def testExpression4(): Unit = {
```

```
  println(expression4(10, 2, 3)) // Some(8)
  println(expression4(10, 2, 0)) // None
}
```

```
Some(8)
None
```

Nesse exemplo, utilizamos o padrão de projeto de combinar aninhamentos de `flatMap` e `map` para compor as operações de divisão. Essa construção pode ser mais simplificada ainda utilizando *for-comprehensions*, que são um açúcar sintático para o mesmo padrão de projeto. Por exemplo, a expressão 5.8 poderia ser reescrita como no código 5.28.

Código 5.28: Composição segura de funções com `Option` via compreensões `for`.

```
def expression5(x: Int, y: Int, z: Int): Option[Int] = {
  for {
    quotient1 <- divide(x, y)
    quotient2 <- divide(x, z)
  } yield quotient1 + quotient2
}
@main def testExpression5(): Unit = {
  println(expression5(10, 2, 3)) // Some(8)
  println(expression5(10, 2, 0)) // None
}
```

```
Some(8)
None
```

Esse tipo de construção pode parecer mais desafiadora nesse momento, pois se apóia em algumas construções que ainda não estudamos, como os tipos monádicos e as *for-comprehensions*. No entanto, é importante ressaltar que essas construções são bastante comuns em Scala e em outras linguagens funcionais, e são uma forma idiomática de lidar com tipos opcionais. Estudaremos essas e outras construções relevantes em aulas posteriores.

EXEMPLO: `PositivePair` OPCIONAL Vamos lembrar que nossa classe `PositivePair`, definida no código 5.19, possuia uma impureza funcional no construtor primário, pois lançava uma exceção se os elementos não fossem positivos. Isso claramente viola a transparência referencial do objeto, pois o construtor não é uma função total. Podemos reescrever a classe `PositivePair` para que ela crie um `Option[PositivePair]` ao invés de lançar uma exceção. Para isso, podemos utilizar um recurso importante de Scala, que são os *companion objects*.

COMPANION OBJECT Em Scala, temos um padrão de projeto bastante recorrente, amplamente utilizado nas coleções Scala, que é o *companion object* (objeto associado ou objeto auxiliar). O *companion object* é um objeto com as seguintes características:

- É definido na mesma unidade de compilação que a classe, ou seja, no mesmo arquivo fonte;

- Possui o mesmo nome da classe, mas com a primeira letra maiúscula;
- Pode acessar os membros privados da classe, e vice-versa.

CONSTRUTOR PRIVADO O primeiro passo para melhorar a classe `PositivePair` é eliminar o disparo de exceções no construtor primário. Porém, isso traz de volta a vulnerabilidade de termos objetos inválidos. Para evitar que isso aconteça, vamos tornar o construtor primário privado e a classe privados, prevenindo que os usuários da classe criem instâncias diretamente, conforme o código 5.29.

Código 5.29: Definição de uma classe `PositivePair` com construtor privado.

```
class PositivePair2 private (override val first: Int, override val second:
↪  Int) extends Pair[Int](first, second) {
    (...)

    override def name = "PositivePair"
}
```

IMPLEMENTAÇÃO DO COMPANION OBJECT Em seguida, criamos um *companion object* para a classe `PositivePair`, que será responsável por criar instâncias válidas da classe, conforme o código 5.30.

Código 5.30: Definição de um *companion object* para `PositivePair`.

```
object PositivePair2 {
    def apply(first: Int, second: Int): Option[PositivePair2] = {
        if (first >= 0 && second >= 0) {
            Some(new PositivePair2(first, second))
        } else {
            None
        }
    }
}
```

```
PositivePair2(1, 2) // Some(PositivePair2(1, 2))
PositivePair2(3, 2) // Some(PositivePair2(3, 2))
PositivePair2(-1, 2) // None
PositivePair2(1, -2) // None
PositivePair2(0, 2) // None
PositivePair2(1, 0) // None
```

Com isso, eliminamos a impureza funcional do construtor primário, pois agora ele não lança exceções. Em vez disso, o *companion object* fornece um método `apply` que retorna um `Option[PositivePair2]`. Se os valores fornecidos forem válidos, o método retorna um `Some(PositivePair2)`, caso contrário, retorna `None`.

`Option` VS `Null` O uso de `Option` é preferível ao uso de `Null` por várias razões:

- **Segurança de tipos**: `Option` é um tipo verificado em tempo de compilação, enquanto `Null` é um valor verificado em tempo de execução.

Consequentemente, `Null` pode levar a erros difíceis de reproduzir (e.g. `NullPointerException`);

- **Componibilidade**: `Option` permite compor funções de forma segura e declarativa, enquanto `Null` pode levar a erros de composição e dificultar a legibilidade do código.
- **Expressividade**: `Option` expressa claramente a intenção de que um valor pode estar ausente, enquanto `Null` é ambíguo e pode ser confundido com um valor válido.
- **Propagação**: quando uma subexpressão avalia a um `Option`, o resultado da expressão é automaticamente do tipo `Option`, permitindo que a ausência de valores seja propagada de forma segura ao longo do programa. Mais importante, a função que contém o `Option` é efetivamente avaliada até o fim, sem a término prematuro da execução, como ocorre com exceções.
- **Previsibilidade**: `Option` permite que o programador lide com a ausência de valores de forma explícita. O valor `Null`, por outro lado, não faz parte da verificação de tipos do compilador, e pode levar a erros em tempo de execução se não for tratado corretamente.

null safety Em linguagens imperativas, existe uma tendência em trazer recursos de *null safety* para evitar problemas relacionados ao uso de `null`. Por exemplo, em Kotlin e em TypeScript, temos o conceito de tipos anuláveis, o operador de chamada segura (`?.`), o operador Elvis (`?:`) e outros recursos que permitem lidar com `null` de forma mais segura.

Java, juntamente com as funcionalidades de programação funcional, também introduziu o tipo `Optional` na versão 8, que é similar ao tipo `Option` de Scala. O código 5.31 mostra como poderíamos implementar a função `divide` em Java utilizando o tipo `Optional`.

Código 5.31: Tratamento de erros com `Optional` em Java.

```java
import java.util.Optional;

public class Main {
    public static Optional<Integer> divide(int x, int y) {
        if (y == 0) {
            return Optional.empty();
        } else {
            return Optional.of(x / y);
        }
    }

    public static void main(String[] args) {
        System.out.println(divide(10, 2)); // Optional[5]
        System.out.println(divide(10, 0)); // Optional.empty
    }
}
```

```
Optional[5]
Optional.empty
```

Embora assemelhe-se ao tipo `Option` de Scala, o tipo `Optional` de Java não é um tipo monádico, pois ainda lida com `null`s internamente (essa é uma longa discussão que foge ao escopo do nosso tema). No entanto, ele ainda oferece uma maneira segura de lidar com a ausência de valores, evitando o uso explício de `null`.

5.5.2 Tipos de resultado: `Either`

Os tipos de resultado são uma forma de representar o resultado de uma função em termos de sucesso ou falha, sem recorrer a exceções. Em relação aos tipos opcionais (`Option`), os tipos de resultado são mais flexíveis, pois permitem representar não apenas a ausência de um valor, mas também discriminar um erro ou uma condição especial. Em Scala, o tipo `Either` é comumente utilizado para representar tipos de resultado. O código 5.32 apresenta uma definição simplificada do tipo `Either`.

Código 5.32: Definição simplificada do tipo `Either`.

```scala
sealed trait Either[+E, +A]
case class Left[E](value: E) extends Either[E, Nothing]
case class Right[A](value: A) extends Either[Nothing, A]
```

O tipo abstrato `Either` pode assumir dois tipos concretos:

- `Left[E]`: representa um resultado de erro, onde `E` é o tipo do erro. O construtor `Left` recebe um valor do tipo `E` e o encapsula dentro de uma instância de `Either[E, Nothing]`;
- `Right[A]`: representa um resultado de sucesso, onde `A` é o tipo do valor de sucesso. O construtor `Right` recebe um valor do tipo `A` e o encapsula dentro de uma instância de `Either[Nothing, A]`.

Convenção de sucesso "à direita" A convenção é que o tipo de sucesso seja representado pelo tipo `Right`, enquanto o tipo de erro seja representado pelo tipo `Left`. Essa convenção é útil pois simplifica os algoritmos dos tipos monádicos (e.g. `map`, `flatmap`, `filter`, etc.), que operam sobre o valor da direita por padrão. Assim, quando utilizamos o tipo `Either`, podemos pensar que o valor de sucesso está sempre à direita, enquanto o valor de erro está sempre à esquerda.

Declaração de tipos de resultado Para demonstrar a declaração de tipos de resultado, vamos adaptar o código 5.22 para utilizar o tipo `Either` no lugar de exceções. O código 5.33 apresenta essa adaptação.

Código 5.33: Tratamento de erros com tipos de resultado.

```scala
def divide(x: Int, y: Int): Either[String, Int] = {
  if (y == 0) Left("Division by zero")
  else Right(x / y)
```

```
}
@main def testDivideEither(): Unit = {
  println(divide(10, 2)) // Right(5)
  println(divide(10, 0)) // Left(Division by zero)
}
```

```
Right(5)
Left("Division by zero")
```

No código 5.33, a função `divide` agora retorna um valor do tipo `Either[String, Int]`. Se o divisor for zero, a função retorna `Left("Division by zero")`, indicando que ocorreu um erro. Caso contrário, ela retorna `Right(x / y)`, encapsulando o resultado da divisão dentro de uma instância de `Either`.

`Option` vs `Either` A principal vantagem de `Either` em relação a `Option` é que contamos com mais recursos para indicar a natureza do problema que causou a ausência de valor. No exemplo, optamos por indicar o erro com uma mensagem de texto, mas poderíamos utilizar um tipo mais específico para representar o erro, como um tipo enumerado ou um tipo de erro personalizado. Isso permite que o programador forneça informações mais detalhadas sobre o erro, facilitando a depuração e o tratamento de erros. Por exemplo, considere os tipos de erro definidos no código 5.34.

Código 5.34: Definição de tipos de erro.

```
sealed trait ArithmeticError
case object DivisionByZero extends ArithmeticError
case class NegativeNumberError(value: Int) extends ArithmeticError
case class OverflowError(value: Int) extends ArithmeticError
```

Poderíamos usar esses tipos de erro para representar os erros de forma mais específica, como no código 5.35.

Código 5.35: Tratamento de erros com tipos de resultado e tipos de erro.

```
def dividePositive(x: Int, y: Int): Either[ArithmeticError, Int] = {
  if (y == 0) Left(DivisionByZero)
  else if (x < 0 || y < 0) Left(NegativeNumberError(x))
  else Right(x / y)
}

def multiply(x: Int, y: Int): Either[ArithmeticError, Int] = {
  if (x > Int.MaxValue / y) Left(OverflowError(x))
  else Right(x * y)
}

@main def testDivideEitherWithErrors(): Unit = {
  println(dividePositive(10, 2))
  println(dividePositive(10, 0))
  println(dividePositive(-10, 2))
  println(multiply(10, 2))
```

```
17    println(multiply(Int.MaxValue, 2))
18  }
```

```
Right(5)
Left(DivisionByZero)
Left(NegativeNumberError(-10))
Right(20)
Left(OverflowError(2147483647))
```

TIPO MONÁDICO Assim como o tipo `Option`, o tipo `Either` é um tipo monádico, que possui métodos padronizados de ordem superior que facilitam a composição de funções. Ou seja, podemos utilizar as mesmas técnicas de componibilidade que utilizamos com o tipo `Option`, ou seja, flatMap, map, filter, for comprehensions, entre outros.

5.6 Estudo de caso: pilha persistente

Uma pilha é uma estrutura de dados que segue o princípio LIFO (*Last In, First Out*), ou seja, o último elemento adicionado é o primeiro a ser removido. Em programação funcional, podemos modelar uma pilha utilizando TADs imutáveis, onde cada operação de modificação resulta em uma nova instância da pilha, preservando a versão anterior.

API As operações fundamentais de uma pilha são definidas no código 5.36.

Código 5.36: Definição de uma pilha funcional.

```
1  trait Stack[T] {
2    def push(x: T): Stack[T]
3    def pop: (Option[T], Stack[T])
4    def peek: Option[T]
5    def isEmpty: Boolean
6    def size: Int
7  }
```

A seguir detalhamos a semântica de cada operação:

- `push(x: T): Stack[T]`: adiciona o elemento `x` ao topo da pilha e retorna uma nova instância da pilha com o elemento adicionado. A pilha original permanece inalterada.
- `pop: (Option[T], Stack[T])`: remove o elemento do topo da pilha e retorna uma tupla contendo um tipo opcional representando o elemento removido e uma nova instância da pilha sem o elemento (potencialmente) removido. Perceba que, no caso da pilha estar vazia, não é possível remover um elemento, por isso precisamos especificar que o retorno é do tipo `Option[T]`.
- `peek: Option[T]`: retorna um tipo opcional representando o elemento do topo da pilha, sem removê-lo.

- `isEmpty: Boolean`: verifica se a pilha está vazia, retornando `true` ou `false`;
- `size: Int`: retorna o número de elementos na pilha.

Tipos opcionais Nem sempre é possível remover ou consultar um elemento do topo da pilha, como no caso de uma pilha vazia. Em programas que não são puramente funcionais, esse caso é comumente tratado disparando uma exceção, como `NoSuchElementException`. No entanto, essa abordagem não é adequada para programação funcional, pois, como já amplamente discutimos, as exceções introduzem efeitos colaterais e dificultam a composibilidade do programa.

Por isso, utilizamos o tipo opcional `Option[T]` para representar o elemento removido. Assim, se a pilha estiver vazia, o retorno será `None`, indicando que não há um elemento a ser removido. Caso contrário, o retorno será `Some(x)`, onde `x` é o elemento removido.

Implementação com listas imutáveis A implementação de uma pilha funcional pode ser feita utilizando diferentes estruturas de dados subjacentes, como listas, arrays ou vectors. No entanto, a lista encadeada simples é uma estrutura bastante adequada para implementar uma pilha funcional, pois as operações eficientes dessa estrutura (inserção e remoção no início) também respeitam a lógico LIFO, de acordo com a correspondência estabelecida na tabela 5.5.

Tabela 5.5: Correspondência entre operações de pilha e operações de lista.

Operação de pilha	Operação de lista	Complexidade
`push(x: T)`	`x :: list`	$O(1)$
`pop`	`(list.head, list.tail)`	$O(1)$
`peek`	`list.head`	$O(1)$
`isEmpty`	`list.isEmpty`	$O(1)$
`size`	`list.length`	$O(n)$

Implementação do TAD Para implementar o TAD, vamos definir a classe `ListStack`, que estende o trait `Stack`. A estrutura básica do TAD é apresentada no código 5.37.

Código 5.37: Implementação de uma pilha funcional com listas.

```
class ListStack[T](val elements: List[T], override val size: Int) extends
↪  Stack[T]  {
  def this() = ???
  def this(elements: List[T]) = ???
  override def push(x: T): Stack[T] = ???
  override def pop: (Option[T], Stack[T]) = ???
  override def peek: Option[T] = ???
  override def isEmpty: Boolean = ???
  override def toString: String = ???
}
```

Na estrutura básica, ainda não definimos as implementações, portanto todos os métodos estão marcados com `???`. Além de sobrescrever os métodos do trait `Stack`, também sobrescrevemos o método `toString` (originalmente de `AnyRef`) para fornecer uma representação textual para facilitar a depuração da pilha em um dado momento. Além dos métodos tradicionais, utilizamos o construtor padrão da classe para definir duas propriedades adicionais:

- `val elements: List[T]`: uma lista imutável que armazena os elementos da pilha. Essa lista é inicializada pelos construtores e utilizada nas operações da pilha;
- `override val size: Int`: o tamanho da pilha, que é sobrescrito do trait `Stack`. Declaramos esse valor no construtor para permitir que ele seja sempre quando um elemento é adicionado ou removido da pilha. Assim, não precisamos percorrer toda a pilha para calcular o tamanho.

Ambas as propriedades são marcadas como `val` para garantir que sejam imutáveis, preservando a semântica funcional da pilha.

Construtores A classe `ListStack` possui os seguintes construtores:

- **Construtor primário**. O construtor primário é o construtor padrão da classe, que recebe uma lista de elementos e um tamanho. Esse construtor é definido na própria assinatura da classe. É preciso que o programador esteja atento para que os parâmetros tenham valores mutuamente consistentes, ou seja, o tamanho deve corresponder ao número de elementos na lista.
- **Construtores secundários**. Temos dois construtores secundários, um que não recebe parâmetros, e outro que recebe uma lista de elementos como parâmetros. A implementação desse construtores é ilustrada no código 5.38.

Código 5.38: Implementação dos construtores secundários da pilha.

```
1 def this() = this(List.empty[T], 0)
2 def this(elements: List[T]) = this(elements, elements.length)
```

Em ambos os casos, o tamanho da pilha é calculado a partir da lista de elementos. O construtor secundário sem parâmetros é útil para criar uma pilha vazia, enquanto o construtor com lista permite criar uma pilha a partir de uma lista existente. Perceba que todo construtor secundário deve chamar o construtor primário, passando os argumentos necessários. Assim, a pilha é sempre inicializada com uma lista de elementos e um tamanho.

Operações de consulta básicas As operações de consulta da pilha são implementadas conforme o código 5.39.

Código 5.39: Implementação das operações de consulta da pilha.

```
1 override def isEmpty: Boolean = elements.isEmpty
2 override def toString: String = s"top -> (${elements.mkString(", ")})"
```

Temos dois métodos de consulta: `isEmpty` e `toString`. O método `isEmpty` verifica se a lista de elementos está vazia, retornando `true` ou `false`. O método `toString` retorna uma representação textual da pilha, mostrando os elementos do topo para a base, o que facilita a depuração.

Uniform Access Principle Perceba que na nossa classe `ListStack` o método `size` (originalmente definido no trait `Stack`) é sobrescrito como uma propriedade, ou seja, uma função foi sobrescrita como uma propriedade (o mais intuitivo seria uma função sobrescrever outra função). Isso é possível porque Scala adota o *Uniform Access Principle*, que permite tratar propriedades e métodos uniformemente. Em resumo, um método sem parâmetros pode ser tratado como uma propriedade, e uma propriedade pode ser tratada como um método sem parâmetros.

Assim, podemos acessar a propriedade `size` como se fosse um método, sem precisar utilizar parênteses. Essa abordagem torna o código mais legível e expressivo, pois não precisamos nos preocupar com a diferença entre propriedades e métodos em algumas situações bem específicas.

CONSULTA AO TOPO A operação de consulta ao topo da pilha (`peek`) retorna o elemento do topo da pilha, sem removê-lo. Para implementar essa operação, verificamos se a lista `elements` está vazia. Se não estiver, retornamos o primeiro elemento da lista como um tipo opcional `Some`. Caso contrário, retornamos `None`, indicando que a pilha está vazia. O código 5.40 apresenta a implementação do método `peek`.

Código 5.40: Implementação da operação de consulta ao topo.

```
override def peek: Option[T] = {
  if (elements.isEmpty) None
  else Some(elements.head)
}
```

EMPILHAMENTO A operação de empilhamento (`push`) adiciona um elemento ao topo da pilha. Para implementar essa operação, utilizamos o operador cons para adicionar o elemento no início da lista `elements`. O código 5.41 apresenta a implementação do método `push`.

Código 5.41: Implementação da operação de empilhamento.

```
override def push(x: T): Stack[T] = {
  new ListStack(x :: elements, size + 1)
}
```

A operação `push` cria uma nova instância de `ListStack`, adicionando o elemento `x` no início da lista `elements` e incrementando o tamanho da pilha em 1. A lista original permanece inalterada, mantendo a persistência.

COMPARTILHAMENTO ESTRUTURAL Note que, como estamos utilizando uma lista imutável, a operação de empilhamento não cria uma cópia completa da lista. Em vez disso, ela compartilha a estrutura da lista original, criando uma nova instância de `ListStack` que aponta para a mesma lista de elementos, mas com o novo

elemento adicionado no início. Isso é possível porque as listas imutáveis em Scala são projetadas para serem compartilhadas de forma eficiente, economizando memória e tempo de execução. Por exemplo, considere a sequência de empilhamentos apresentada no código 5.42.

Código 5.42: Compartilhamento estrutural de listas imutáveis.

```
val s1 = new ListStack[Int]()
val s2 = stack1.push(1)
val s3 = stack2.push(2)
val s4 = stack3.push(3)
```

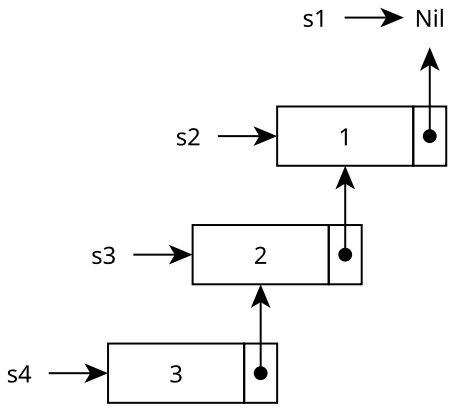

Figura 5.7: Compartilhamento estrutural em pilha persistente.

A sequência de operações gera a estrutura representada na figura 5.7. Note que, apesar de termos quatro instâncias de `ListStack`, elas compartilham nós da lista `elements`. Assim, a pilha `s4` contém os elementos `3, 2, 1`, enquanto a pilha `s3` contém os elementos `2, 1`, e assim por diante. Isso permite que as pilhas sejam persistentes e eficientes em termos de memória, pois não precisamos duplicar toda a lista a cada operação de empilhamento.

Desempilhamento A principal ação da operação de desempilhamento (`pop`) é remover o elemento do topo da pilha. Como estamos desenvolvendo uma estrutura imutável e persistente, é esperado que essa operação retorne uma nova instância da pilha, preservando a versão anterior. Por outro lado, também é esperado, pelas convenções de uma pilha, que essa operação retorne o elemento removido. Como temos a necessidade de retornar dois valores a partir da função, podemos resolver esse problema definindo o retorno da função como uma tupla, conforme o código 5.43.

Código 5.43: Implementação da operação de desempilhamento.

```
override def pop: (Option[T], Stack[T]) = {
  if (elements.isEmpty) (None, this)
  else (Some(elements.head), new ListStack(elements.tail, size - 1))
```

```
}
```

A operação `pop` verifica se a lista `elements` está vazia. Se estiver, retorna uma tupla contendo `None` e a própria pilha (`this`), indicando que não há elemento para remover. Caso contrário, retorna uma tupla contendo o elemento do topo da pilha (como um tipo opcional `Some`) e uma nova instância de `ListStack`, que representa a pilha sem o elemento removido, decrementando o tamanho em 1.

Companion object A API da nossa implementação de pilha tem uma vulnerabilidade no construtor primário, pois podemos passar valores inconsistentes para os parâmetros `elements` e `size`. Além disso, do ponto de vista de ocultamento de informação, estamos desnecessariamente expondo detalhes da implementação interna, ou melhor, o usuário da pilha não precisa ter de lidar com listas para criar a pilha. Para evitar esse problema, podemos adotar uma estratégia de encapsulamento que permita restringir a criação de instâncias da pilha apenas através de um construtor seguro.

As coleções Scala utilizam o *companion object* para fornecer métodos de fábrica que permitem criar instâncias da coleção de uma sintaxe bastante conveniente. Por exemplo, quando criamos uma lista com o método `List(1, 2, 3)`, estamos utilizando o *companion object* da classe `List` para criar uma instância da lista. Podemos utilizar esse mesmo recurso para melhorar a API da nossa pilha.

Construtores privados Primeiramente, vamos aplicar ocultamento de informação na nossa classe `ListStack`, tornando tanto a classe quanto os seus construtores (primário e secundários) privados, conforme o código 5.44. O efeito dessa modificação é que a classe não pode mais ser acessada de fora da unidade de compilação (o arquivo fonte), ou do pacote (caso esteja definido) e, ainda, a classe não pode ser instanciada diretamente.

Código 5.44: Definição de uma pilha funcional com construtores privados.

```
private class ListStack[T] private (val elements: List[T], override val size:
↪  Int) extends Stack[T] {
  private def this() = this(Nil, 0)
  private def this(elements: List[T]) = this(elements, elements.size)
  (...)
```

A princípio, isso pode parecer um contrassenso, pois estamos tornando a classe e os construtores privados, o que impede a criação de instâncias da pilha fora da própria classe. No entanto, isso é intencional: devemos lembrar que um *companion object* que seja homônimo à classe possui privilégios que o permitem acessar os membros privados da classe, e vice-versa.

Implementação do *companion object* A seguir, vamos implementar o *companion object* da classe `ListStack`, que será responsável por fornecer métodos de fábrica para criar instâncias da pilha de forma segura. O código 5.45 apresenta a implementação do *companion object*.

Código 5.45: Implementação do *companion object* da pilha funcional.

```scala
object ListStack {
  def apply[T](): ListStack[T] = new ListStack[T]()
  def apply[T](elements: T*): ListStack[T] = new ListStack(elements.toList)
}
```

O *companion object* da classe `ListStack` define dois métodos `apply` que permitem criar instâncias da pilha com uma notação simplificada, visto que a chamada explícita a esses métodos são opcionais. Além disso, esses métodos são chamados sem a necessidade de instanciar o *companion object* pois, por definição, ele é um singleton.

- O primeiro método `apply` não recebe parâmetros e cria uma pilha vazia, utilizando o construtor primário da classe. Esse método pode ser usado da seguinte forma:

```scala
val emptyStack = ListStack[Int]()
```

- O segundo método `apply` um parâmetro variádico do tipo genérico `T` e cria uma pilha a partir desses elementos, convertendo-os em uma lista.

```scala
val stackFromElements = ListStack(1, 2, 3)
```

Esses métodos garantem que as instâncias da pilha sejam criadas de forma consistente, evitando problemas de inconsistência entre os parâmetros `elements` e `size`.

Outras estruturas subjacentes Embora tenhamos implementado a pilha utilizando listas, poderíamos utilizar outras estruturas subjacentes, cada uma delas teria vantagens e desvantagens. Por exemplo:

- **`ArraySeq`**. Utilizar um `ArraySeq` como estrutura subjacente pode ser vantajoso em termos de desempenho, pois permite acesso rápido aos elementos e ocupa menos memória do que uma lista encadeada. Poderíamos, por exemplo, empilhar elementos sempre "ao final" do array, enquanto houver posições livres. No entanto, a pilha ficaria mais restrita, pois necessariamente a pilha teria uma limitação de tamanho (`ArraySeq` não é de tamanho dinâmico). O `ArraySeq` poderia ser modificado para permitir o redimensionamento, mas isso tornaria a implementação mais complexa e, além disso, quando elementos forem removidos, a memória ocupada por esses elementos não seria liberada.

- **`Vector`**. Utilizar um `Vector` como estrutura subjacente pode ser vantajoso em termos de desempenho, pois permite acesso rápido aos elementos e ocupa menos memória do que uma lista encadeada. Poderíamos modelar como convenção que o topo da pilha é o primeiro ou o último elemento do vector: a escolha é arbitrária, pois todas as operações teriam tempo logarítmico. Embora tenhamos ciência de que as operações do vector são "efetivamente constantes", a lista encadeada ainda é preferível pois o custo das operações é constante de fato.

5.7 Estudo de caso: fila persistente

Uma fila é uma estrutura de dados que segue o princípio FIFO (*First In, First Out*), ou seja, o primeiro elemento adicionado é o primeiro a ser removido. Analogamente à pilha, uma fila persistente é implementada utilizando estruturas subjacentes imutáveis, sejam elas listas, arrays ou vectors. Na nossa discussão, vamos focar em listas imutáveis como estrutura subjacente.

API As operações fundamentais de uma fila são definidas no código 5.46.

Código 5.46: Definição de uma fila funcional.

```
trait Queue[T] {
    def enqueue(element: T): Queue[T]
    def dequeue: (Option[T], Queue[T])
    def peekFront: Option[T]
    def peekBack: Option[T]
    def isEmpty: Boolean
    def size: Int
    def toString: String
}
```

A seguir detalhamos a semântica de cada operação:

- `enqueue(element: T): Queue[T]`: adiciona o elemento `element` ao final da fila e retorna uma nova instância da fila com o elemento adicionado. A fila original permanece inalterada.
- `dequeue: (Option[T], Queue[T])`: remove o elemento do início da fila e retorna uma tupla contendo um tipo opcional representando o elemento removido e uma nova instância da fila sem o elemento (potencialmente) removido. Assim como na pilha, utilizamos o tipo opcional para representar a ausência de um elemento, caso a fila esteja vazia.
- `peekFront: Option[T]`: retorna um tipo opcional representando o elemento do início da fila, sem removê-lo.
- `peekBack: Option[T]`: retorna um tipo opcional representando o elemento do final da fila, sem removê-lo.
- `isEmpty: Boolean`: verifica se a fila está vazia, retornando `true` ou `false`.
- `size: Int`: retorna o número de elementos na fila.
- `toString: String`: retorna uma representação textual da fila, facilitando a depuração.

5.7.1 Implementação com uma lista imutável

A implementação de uma fila persistente utilizando uma única lista imutável é considerada ingênua, pois a operação de enfileiramento é ineficiente. Primeiramente,

vamos desenvolver essa implementação e, posteriormente, vamos analisar a razão dessa implementação ser ineficiente.

Implementação do TAD Para implementar o TAD, vamos definir a classe `NaiveListQueue`, que estende o trait `Queue`. A estrutura básica do TAD é apresentada no código 5.47.

Código 5.47: Implementação de uma fila funcional com listas.

```
class NaiveListQueue[T](elements: List[T] = List.empty) extends Queue[T] {
    def enqueue(element: T): Queue[T] = new NaiveListQueue(elements :+
    ↪  element)

    def dequeue: (Option[T], Queue[T]) =
      if (elements.isEmpty) (None, this)
      else (Some(elements.head), new NaiveListQueue(elements.tail))

    def peek: Option[T] = elements.headOption

    def isEmpty: Boolean = elements.isEmpty

    def size: Int = elements.size

    override def toString: String = s"front -> (${elements.mkString(", ")})"
}
```

A implementação da fila utilizando apenas uma lista é relativamente simples. Vamos nos concentrar nos métodos de enfileiramento (`enqueue`) e desenfileiramento (`dequeue`), que são os mais relevantes para a discussão.

Desenfileiramento O método `dequeue` remove o elemento do início da fila. Se a fila estiver vazia, ele retorna um valor opcional vazio e a própria fila. Caso contrário, ele retorna o elemento removido e uma nova fila sem esse elemento. Esse método é eficiente porque a operação de remoção em uma lista imutável é realizada de forma constante, já que estamos apenas criando uma nova lista sem o primeiro elemento. Por exemplo, a figura 5.8 ilustra a sequência de desenfileiramentos dado pelo código 5.48.

Código 5.48: Desenfileiramento de uma fila funcional.

```
val queue = new NaiveListQueue[Int]()
val queue1 = queue.enqueue(1).enqueue(2).enqueue(3).enqueue(4)

val (deq1, queue2) = queue1.dequeue
val (deq2, queue3) = queue2.dequeue
val (deq3, queue4) = queue3.dequeue
```

Perceba que, além de eficiente em tempo, ao executar em tempo O(1), a operação de desenfileitamento também é eficiente em espaço, pois não duplicamos a lista inteira, apenas definimos uma nova lista apontando para o restante da lista original, sem o primeiro elemento. Assim, a fila `queue1` contém os elementos `1, 2, 3, 4`,

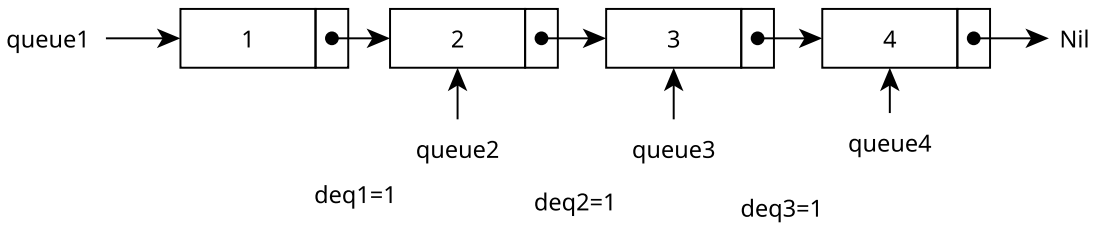

Figura 5.8: Desenfileiramentos sucessivos em uma fila persistente.

enquanto a fila queue2 contém os elementos `2`, `3`, `4`, e assim por diante.

ENFILEIRAMENTO O método enqueue adiciona um elemento ao final da fila. No entanto, essa operação é ineficiente, pois a lista imutável não permite adicionar elementos no final de forma eficiente. A operação de adição no final de uma lista imutável tem complexidade linear, pois é necessário criar uma nova lista com o novo elemento adicionado ao final. Isso significa que, a cada enfileiramento, estamos criando uma nova lista com todos os elementos anteriores e o novo elemento, o que pode ser custoso em termos de desempenho.

Por exemplo, a figura 5.9 ilustra a sequência de enfileiramentos dado pelo código 5.49.

Código 5.49: Enfileiramento de uma fila funcional.

```
1 val queue = new NaiveListQueue[Int]()
2 val queue1 = queue.enqueue(1)
3 val queue2 = queue1.enqueue(2)
4 val queue3 = queue2.enqueue(3)
5 val queue4 = queue3.enqueue(4)
```

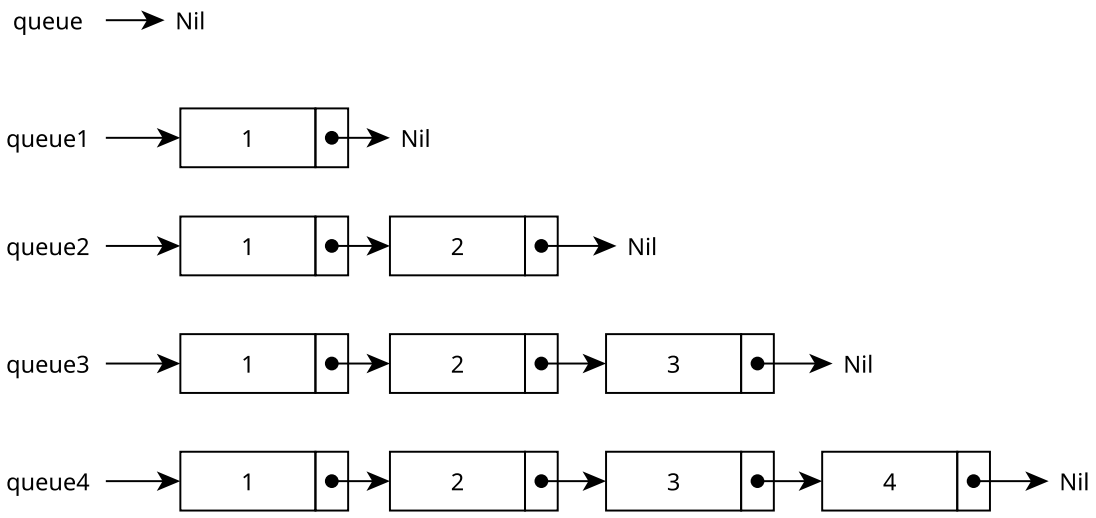

Figura 5.9: Enfileiramentos sucessivos em uma fila persistente.

Perceba que, ao executar o código 5.49, a fila queue1 contém o elemento 1, a fila queue2 contém os elementos 1, 2, e assim por diante. A operação é ineficiente tanto em tempo quanto em espaço, pois a cada enfileiramento estamos criando uma nova lista com todos os elementos anteriores e o novo elemento.

A tabela 5.6 resume as operações de enfileiramento e desenfileiramento da fila baseada em uma lista.

Tabela 5.6: Complexidade das operações de enfileiramento e desenfileiramento em uma fila baseada em lista.

Operação	Tempo	Espaço
Enfileiramento	$O(n)$	$O(n)$
Desenfileiramento	$O(1)$	$O(1)$

5.7.2 Implementação com duas pilhas

Para resolver o problema de eficiência do enfileiramento, podemos utilizar uma abordagem que combina duas pilhas. A intuição do algoritmo é a seguinte:

- Utilizamos uma pilha para armazenar os elementos que estão sendo enfileirados (`front`) e outra pilha para armazenar os elementos que estão sendo desenfileirados (`rear`).
- Quando um elemento é enfileirado, ele é adicionado à pilha `rear`.
- Quando um elemento é desenfileirado, verificamos se a pilha `front` está vazia. Se não estiver, simplesmente obtemos o primeiro elemento. Se estiver, transferimos todos os elementos da pilha `rear` para a pilha `front`, invertendo a ordem dos elementos. Isso garante que o primeiro elemento adicionado seja o primeiro a ser removido.
- Após a transferência, o elemento do topo da pilha `front` é removido e retornado como o elemento desenfileirado.

A figura 5.10 ilustra o comportamento dessa implementação para as operações listadas no código 5.50.

Código 5.50: Enfileiramento e desenfileiramento com duas pilhas.

```
1 val queue = new TwoStacksQueue[Int]()
2 queue.enqueue(1).enqueue(2).enqueue(3)
3 val (deq1, queue1) = queue.dequeue
```

Como podemos observar, a pilha `rear` armazena os elementos que estão sendo enfileirados, enquanto a pilha `front` armazena os elementos que estão sendo desenfileirados. Caso a pilha `front` esteja vazia, todos os elementos da pilha `rear` são transferidos para a pilha `front`, invertendo a ordem dos elementos. Assim, o primeiro elemento adicionado é o primeiro a ser removido, respeitando o princípio FIFO.

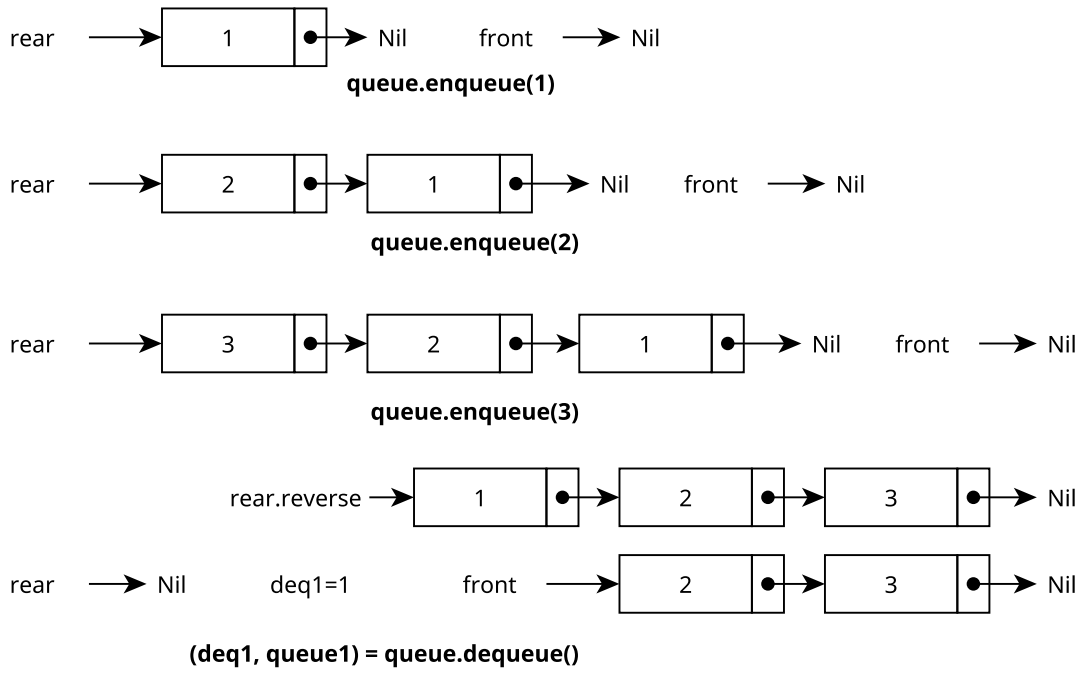

Figura 5.10: Enfileiramento e desenfileiramento com duas pilhas.

O código 5.51 apresenta a implementação do TAD `TwoStacksQueue`, que estende o trait `Queue`.

Código 5.51: Implementação de uma fila funcional com duas pilhas.

```
class TwoStacksQueue[T] (val front: List[T] = List.empty, val rear: List[T] =
↪  List.empty, override val size: Int) extends Queue[T] {

  def enqueue(element: T): TwoStacksQueue[T] = new TwoStacksQueue(front,
  ↪  element :: rear, size + 1)

  def dequeue: (Option[T], TwoStacksQueue[T]) =
      if (front.isEmpty && rear.isEmpty) (None, this)
      else if (front.isEmpty) {
        val newFront = rear.reverse
        (Some(newFront.head), new TwoStacksQueue(newFront.tail, List.empty,
        ↪  size - 1))
      } else {
        (Some(front.head), new TwoStacksQueue(front.tail, rear, size - 1))
      }

      (...)

      override def toString: String = s"front -> (${front.mkString(", ")})
      ↪  (${rear.reverse.mkString(", ")}) <- rear"
}
```

Perceba que a operação de empilhamento é bastante simples: basta adicionar o elemento ao início da pilha `rear` e incrementar o tamanho da fila. A operação de desenfileiramento é um pouco mais complexa, pois precisamos verificar se a pilha `front` está vazia. Se estiver, transferimos todos os elementos da pilha `rear` para a pilha `front`, invertendo a ordem dos elementos. Caso contrário, simplesmente removemos o elemento do topo da pilha `front`.

Complexidade do desenfileiramento Como precisamos inverter a ordem dos elementos da pilha `rear` para a pilha `front`, a operação de desenfileiramento tem complexidade linear no pior caso, pois precisamos percorrer todos os elementos da pilha `rear` para transferi-los para a pilha `front`. No entanto, essa operação é amortizada: a partir do momento em que um elemento é transferido da pilha `rear` para a pilha `front`, ele pode ser removido em tempo constante. A tabela 5.7 resume as complexidades das operações de enfileiramento e desenfileiramento da fila baseada em duas pilhas.

Tabela 5.7: Complexidade das operações de enfileiramento e desenfileiramento da fila baseada em duas pilhas.

Operação	Melhor caso	Pior caso
`enqueue`	$O(1)$	$O(1)$
`dequeue`	$O(1)$	$O(n)$
`peekFront`	$O(1)$	$O(n)$
`peekBack`	$O(1)$	$O(n)$
`isEmpty`	$O(1)$	$O(1)$
`size`	$O(1)$	$O(1)$
`toString`	$O(n)$	$O(n)$

Exercícios Recomenda-se ao leitor implementar as operações remanescentes do TAD `TwoStacksQueue`, como `peekFront` e `peekRear`. Atenção aos casos em que uma das pilhas está vazia, e como isso afeta o comportamento das operações de consulta. Além disso, recomenda-se implementar um companion object para a fila, semelhante ao que foi feito com a pilha, para fornecer métodos de fábrica que permitam criar instâncias da fila de forma segura, respeitando os princípios de independência de implementação e ocultamento de informação.

5.8 Resumo

Neste capítulo, apresentamos os fundamentos das estruturas de dados funcionais, destacando sua importância para a programação funcional moderna. Discutimos conceitos essenciais como imutabilidade, persistência e compartilhamento estrutural, além de analisar vantagens e desafios dessas abordagens. Foram exploradas as principais coleções funcionais disponíveis em Scala, com exemplos práticos e estudo de caso detalhado da estrutura `List`. Também abordamos operações fundamentais, como inserção, remoção, concatenação e inversão, sempre sob a ótica da eficiência e segurança proporcionadas pelo paradigma funcional. Ao final, o leitor

está apto a compreender, projetar e utilizar estruturas funcionais em aplicações reais, reconhecendo seu papel central em linguagens funcionais, a exemplo de Scala.

O tema de estruturas funcionais é vasto e complexo, e este capítulo serve como uma introdução aos conceitos fundamentais. Recomenda-se ao leitor explorar mais a fundo as coleções funcionais disponíveis em Scala, bem como as técnicas avançadas de programação funcional, como mônadas e funtores, que são essenciais para o desenvolvimento de aplicações robustas e eficientes. Na bibliografia, recomenda-se a leitura do livro de Okasaki, que é uma das referência clássicas nesse tema.

Aviso de licença de uso