Capítulo 4

Recursividade

A recursividade é um conceito fundamental na programação funcional, sendo o principal recurso para resolver problemas que envolvem repetições, como processamentos iterativos ou processamento de estruturas de dados. O principal aspecto que a torna útil para a programação funcional é que a recursividade modela repetições sem a necessidade de operar variáveis mutáveis.

Repetições imperativas No paradigma de programação imperativa, repetições são frequentemente implementadas por meio de laços de repetição. Por exemplo, considere o problema de somar os números de a até b, definido pela fórmula 4.1.

$$S(a, b) = a + (a + 1) + (a + 2) + \ldots + b \tag{4.1}$$

Em Scala, podemos propor uma solução imperativa para esse problema utilizando um laço `while`, conforme ilustrado no programa 4.1.

Código 4.1: Solução imperativa para o problema de somar os números de `a` até `b`.

```scala
def imperativeSum(a: Int, b: Int): Int = {
  if (a > b) {
    return 0
  }

  var sum = 0
  var i = a

  while (i <= b) {
    sum += i
    i += 1
  }

  return sum
}

@main def hello(): Unit = {
  println(imperativeSum(1, 10)) // Output: 55
  println(imperativeSum(5, 15)) // Output: 110
  println(imperativeSum(0, 100)) // Output: 5050
```

```
21 }
```

Como podemos observar no programa 4.1, a solução imperativa depende fortemente de variáveis mutáveis para controle da repetição e construção do resultado. No exemplo, há mutabilidade em dois pontos:

- **Contador**. A variável `i` é utilizada para controlar o índice do laço de repetição, iniciando em `a` e incrementando até que atinja o valor de `b`.
- **Acumulador**. A variável `sum` é utilizada para acumular o resultado da soma, iniciando em `0` e sendo atualizada a cada iteração do laço.

Em suma, o paradigma imperativo apoia-se fortemente no uso de variáveis mutáveis para controlar o fluxo de execução. Já no paradigma funcional, a pureza dos programas é em parte mantida pela ausência de mutabilidade. Para que seja possível realizar processamentos repetitivos, é necessário que o paradigma funcional utilize outros mecanismos, como a recursividade.

4.1 Funções recursivas

Uma função recursiva é uma função que chama a si mesma, direta ou indiretamente. As principais características de uma função recursiva são:

- **Auto-referência**. A função deve chamar a si mesma em algum ponto de sua execução.
- **Caso base**. A função deve ter um ou mais casos base que definem quando a recursão deve parar. Esses casos base são essenciais para evitar chamadas infinitas e garantir que a função eventualmente retorne um resultado.
- **Caso recursivo**. A função deve reduzir o problema em cada chamada recursiva, até atingir o caso base. Isso geralmente envolve dividir o problema em subproblemas menores.

SOMATÓRIO RECURSIVO O somatório pode ser definido recursivamente, conforme a fórmula 4.2. Essa fórmula representa um modo clássico de expressar matematicamente uma função recursiva, utilizando a notação de função por partes.

$$S(a,b) = \begin{cases} 0 & \text{se } a > b \\ a + S(a+1,b) & \text{caso contrário} \end{cases} \tag{4.2}$$

Temos dois componentes fundamentais de uma recursão:

- **Caso-recursivo**. Reduzimos o problema do somatório a soma do elemento atual `a` com o somatório dos elementos seguintes, ou seja, `a + S(a + 1, b)`. Em suma, a cada passo da recursão, refazemos o processo com um subproblema menor: o somatório de todos os elementos restantes, excetuando-se o atual.

- **Caso-base**. Definimos que, se a for maior que b, o somatório é 0, ou seja, não há mais números a serem somados. Esse caso base é essencial para evitar chamadas infinitas e garantir que a função eventualmente retorne um resultado. Ou melhor, indica quando o problema foi reduzido ao menor subproblema possível.

Uma importante vantagem de uma definição recursiva é que podemos mapeá-la muito diretamente para um algoritmo recursivo, como ilustrado no programa 4.2. Note que, nesse programa, os casos são mapeados muito diretamente para a expressão condicional `if`.

Código 4.2: Solução recursiva para o problema de somar os números de `a` até `b`.

```scala
package recursivesum

def recursiveSum(a: Long, b: Long): Long =
  if (a > b) 0
  else a + recursiveSum(a + 1, b)

@main def run(): Unit = {
  println(recursiveSum(1, 10)) // Output: 55
  println(recursiveSum(5, 15)) // Output: 110
  println(recursiveSum(0, 100)) // Output: 5050
}
```

```
55
110
5050
```

Comparando-se com a solução imperativa, a solução recursiva apresenta as seguintes vantagens:

- **Imutabilidade**. A solução recursiva não utiliza variáveis mutáveis, o que a torna mais alinhada com os princípios da programação funcional.
- **Declaratividade**. A solução recursiva expressa o problema de forma mais próxima da definição matemática, facilitando a compreensão e a manutenção do código.

Traço de execução O ambiente de execução implementa recursividade por meio de uma estrutura de dados chamada pilha de chamadas (*call stack*). Cada vez que uma função é chamada, um novo quadro de pilha (*stack frame*) é adicionado à pilha, contendo informações sobre a chamada da função, como por exemplo os parâmetros e o ponto de retorno. Quando a função retorna, o quadro correspondente é removido da pilha. Para ilustrar esse comportamento, considere a chamada da função `recursiveSum(2, 5)`. A pilha de chamadas seria construída da seguinte forma (o programa foi incluído novamente para fácil referência):

```
recursiveSum(2, 5)
  2 + recursiveSum(3, 5)
    3 + recursiveSum(4, 5)
      4 + recursiveSum(5, 5)
        5 + recursiveSum(6, 5)
          (a > b, returns 0)
        returns 5 + 0 = 5
      returns 4 + 5 = 9
    returns 3 + 9 = 12
  returns 2 + 12 = 14
```

```
def recursiveSum(a: Int, b: Int): Int
↪  =
  if (a > b) 0
  else a + recursiveSum(a + 1, b)
```

Na notação de traço que utilizamos, cada linha representa uma chamada de função ou a avaliação de uma expressão relevante para o resultado. O aumento do nível de indentação indica uma chamada de função recursiva, enquanto a diminuição do nível de indentação indica o retorno de uma função. A quantidade de níveis de indentação reflete a profundidade da recursão e, por conseguinte, o tamanho da pilha de chamadas.

Como podemos observar, o algoritmo cria uma série de quadros na pilha de execução, cada um com o valor de `a` incrementado em uma unidade. Quando o valor de `a` ultrapassa o valor de `b`, a função retorna `0`, e os quadros da pilha são removidos um a um, retornando os valores acumulados até chegar ao resultado final. O processo de desmontagem da pilha de chamadas para a construção do resultado é denominado desenrolamento (*unwinding*) da pilha.

RECURSÃO E IMUTABILIDADE Ao construirmos soluções recursivas para os problemas de repetição, conseguimos evitar o uso de variáveis mutáveis. Isso é possível pois, a cada passo recursivo, novas variáveis são criadas com os valores necessários para a próxima chamada da função. A figura 4.1 ilustra a pilha de execução no seu estado de maior profundidade, para o exemplo `recursiveSum(2, 5)`.

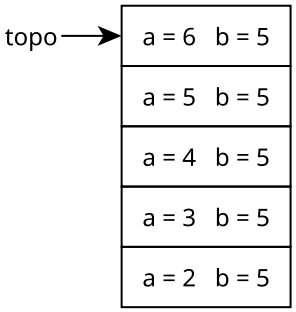

Figura 4.1: Pilha de execução (completa) da função `recursiveSum(2, 5)`.

Cada quadro na pilha contém uma amarração separada entre o parâmetro `a` e o argumento correspondente à chamada da função naquele ponto da execução. Por exemplo, o quadro mais profundo na pilha contém a amarração `a = 6`, que é a última chamada antes de atingir o caso base. Esse comportamento mostra como o

mecanismo de execução de recursões automaticamente cria novas variáveis específicas para cada passo de recursão, promovendo imutabilidade.

CONSUMO DE MEMÓRIA Devido ao comportamento de empilhamento, a recursividade pode consumir uma quantidade significativa de memória, especialmente em casos de recursão profunda. Para o algoritmo de somatório, podemos notar, pelo traço do algoritmo que, para somar os inteiros de 2 a 5, ou seja, um total de 4 inteiros, foi necessário criar 5 quadros na pilha de chamadas. Analisando assintoticamente, podemos afirmar que o consumo de memória é proporcional à profundidade da recursão, ou seja, ao número de chamadas recursivas necessárias para resolver o problema. No caso do somatório, a profundidade da recursão é dada por $b - a + 1$.

Cada chamada recursiva adiciona um novo quadro à pilha, e se a profundidade da recursão for muito alta, isso pode levar a um estouro de pilha (*stack overflow*). Em implementações modernas da JVM, por exemplo, o tamanho máximo da pilha é limitado (cerca de 1 MiB por thread) e, portanto, a profundidade máxima da recursão é limitada. Se a profundidade da recursão exceder esse limite, ocorrerá um erro de estouro de pilha. Por exemplo, o programa 4.2 ilustra a execução de `recursiveSum(1, 1_000_000)` que, devido a uma profundidade muito alta, resulta em um estouro de pilha.

Código 4.3: Solução recursiva para o problema de somar os números de `a` até `b` com profundidade alta.

```
1  package recursivestackoverflow
2
3  import recursivesum.recursiveSum
4
5  @main def run(): Unit = {
6      try {
7          println(recursiveSum(1, 1_000_000))
8      } catch {
9          case e: StackOverflowError => {
10             println("Stack overflow occurred")
11         }
12     }
13 }
```

```
Stack overflow occurred
```

Esse problema ocorre basicamente pelo modo como a recursão foi modelada. No algoritmo, a chamada recursiva é feita pela expressão `a + recursiveSum(a + 1, b)`. Ou seja, a avaliação da expressão ocorrerá na seguinte ordem:

1. `a + 1` será avaliado
2. a função `recursiveSum` é aplicada com o novo valor de `a`
3. o resultado da avaliação de `recursiveSum` será somado ao valor de `a`

Para o passo 3 se complete, é necessário aguardar o resultado do passo 2. Para isso, o sistema de execução "grava" o estado da execução atual na pilha de chamadas, aguardando a conclusão da chamada recursiva. Como esse processo se repete para cada chamada recursiva, a pilha de chamadas cresce rapidamente, levando ao estouro de pilha. Para evitar esse problema, é necessário que a função recursiva seja projetada de forma a não depender do resultado de chamadas recursivas anteriores para calcular o resultado final.

4.2 Recursão na cauda

Recursão na cauda é uma forma especial de recursão na qual a chamada recursiva é absolutamente a última operação realizada pela função antes de retornar um resultado. Essa característica é crucial pois permite que os compiladores e ambientes de execução efetuem otimizações para diminuir o consumo de memória das chamadas recursivas e evitem o estouro de pilha.

Chamada na cauda Uma função apresenta chamada na cauda (*tail call*) quando:

- **A última operação é uma chamada de função**. A chamada de função deve ser a última operação executada pela função antes de retornar um resultado. Isso significa que não pode haver mais cálculos ou operações após a chamada de função.
- **Não há dependência de resultados intermediários**. A função não deve depender de resultados intermediários de chamadas anteriores para calcular o resultado final. Ou seja, a função não deve realizar operações adicionais com o resultado da chamada de função.

Considere por exemplo as funções definidas no programa 4.4.

Código 4.4: Exemplo de chamada na cauda.

```
// no tail call
def f(x: Int, y: Int): Int = {
    x + y
}

// tail call
def g(x: Int): Int = {
    val y = 10

    f(x, y)
}

// no tail call
def h(y: Int): Int = {
    val x = 5

```

```
17      f(x, y) + x
18  }
19
20  // tail call
21  def m(z: Int): Int = {
22      f(z, h(z + 1))
23  }
24
25  // mixed: tail call and no tail call
26  def p(x: Int): Int = {
27      if (x <= 0) f(x, 0)
28      else x + f(x - 1, 0)
29  }
```

Do ponto de vista de chamada na cauda, podemos classificar as funções do programa 4.4 da seguinte forma:

- `f` não apresenta chamada na cauda, pois a última operação é uma soma, que não é uma chamada de função.
- `g` é uma chamada na cauda, pois a última operação é a chamada da função `f`.
- `h` não apresenta chamada na cauda pois, embora haja uma chamada da função `f`, essa chamada não é a última operação executada. Após a chamada de `f`, ainda há uma soma com o valor de `x`, ou seja, `f(x, y) + x`.
- `m` apresenta chamada na cauda, pois a última operação é a chamada da função `f`, mesmo que essa chamada dependa do resultado de outra função (`h`). A função `h` não é uma chamada na cauda, mas isso não impede que `m` seja uma chamada na cauda, pois a última operação de `m` é a chamada de `f`.
- `p` apresenta chamada na cauda em apenas um dos ramos do `if`. No ramo `x <= 0`, a chamada de `f` é a última operação executada, portanto é uma chamada na cauda. No ramo `x > 0`, a chamada de `f` não é a última operação executada, pois há uma soma com o valor de `x`, portanto não é uma chamada na cauda.

FUNÇÕES RECURSIVAS NA CAUDA Uma função é recursiva na cauda quando todas as chamadas recursivas são chamadas na cauda. Considere as funções definidas no programa 4.5.

Código 4.5: Exemplos de funções recursivas na cauda.

```
1  // Not tail recursive
2  def a(n: Int): Int =
3    if (n <= 1) n
4    else a(n - 1) + a(n - 2)
5
6  // Tail recursive
7  def b(n: Int, x: Int = 0, y: Int = 1): Int =
8    if (n == 0) x
```

```
  else b(n - 1, y, x + y)

// Not tail recursive
def c(x: Int, n: Int): Int =
  if (n == 0) 1
  else x * c(x, n - 1)

// Tail recursive
def d(x: Int, n: Int, acc: Int = 1): Int =
  if (n == 0) acc
  else d(x, n - 1, acc * x)

// Modified: two tail recursive cases
def e(n: Int, acc: Int = 0): Int =
  if (n == 0) acc
  else if (n % 2 == 0) e(n / 2, acc + n)
  else e(n - 1, acc + 1)

// f: two recursive cases, one tail recursive, one not
def f(n: Int, acc: Int = 0): Int =
  if (n <= 0) acc
  else if (n % 2 == 0) f(n - 1, acc + n) // tail recursive
  else n + f(n - 2, acc) // not tail recursive
```

Analisando as funções do programa 4.5, podemos classificar as funções da seguinte forma:

- `a` não é recursiva na cauda, pois o caso recursivo envolve uma recursão dupla, cujos resultados precisam ser somados após o término de ambas as chamadas recursivas. Ou seja, a chamada recursiva não é a última operação executada.
- `b` é recursiva na cauda, pois no único caso recursivo a chamada recursiva é a última operação executada.
- `c` não é recursiva na cauda, pois a chamada recursiva ocorre antes da multiplicação com o valor de `x`.
- `d` é recursiva na cauda, pois a chamada recursiva é a última operação executada no caso recursivo.
- `e` é recursiva na cauda, pois todos os casos recursivos são chamadas na cauda.
- `f` não é totalmente recursiva na cauda, pois o ramo que trata números ímpares não é uma chamada na cauda. No entanto, o ramo que trata números pares é uma chamada na cauda.

EXEMPLO: SOMATÓRIO RECURSIVO NA CAUDA Podemos reescrever o algoritmo de somatório para que ele seja recursivo na cauda, conforme o programa 4.6.

Código 4.6: Solução recursiva na cauda para o problema de somar os números de `a` até `b`.

```scala
package tailrecursivesum

def tailRecursiveSum(a: Long, b: Long, acc: Long = 0): Long = {
    if (a > b) acc
    else tailRecursiveSum(a + 1, b, acc + a)
}

@main def run(): Unit = {
    println(tailRecursiveSum(1, 10)) // Output: 55
    println(tailRecursiveSum(5, 15)) // Output: 110
    println(tailRecursiveSum(0, 100)) // Output: 5050
    println(tailRecursiveSum(1, 1_000_000)) // Output: 500000500000
}
```

```
55
110
5050
500000500000
```

A mudança consiste basicamente numa pequena mudança na ordem de avaliação da expressão do caso recursivo. No algoritmo original, o caso recursivo primeiramente avalia a chamada recursiva para somente depois efetuar a soma que acumula o resultado. Podemos facilmente resolver esse problema "adiantando" o processo: criamos um parâmetro adicional `acc` que, na chamada recursiva, recebe como argumento o valor da soma parcial `acc + a`. Assim, a chamada recursiva é a última operação executada, tornando a função recursiva na cauda.

Padrão de projeto *Accumulator* No exemplo do código 4.6, nós aplicamos um padrão de projeto muito comum para transformar nosso código recursivo em recursivo na cauda. O padrão Accumulator (ou acumulador) consiste em adicionar um ou mais parâmetros adicionais na função recursiva, que acumulam resultados parciais da computação à medida que a recursão progride. Essa medida, embora bastante simples, garante que a avaliação da expressão acumulativa ocorra antes da chamada recursiva, garantindo que a chamada seja recursiva na cauda.

Otimização de chamada na cauda Em Scala, quando uma função é recursiva na cauda, o compilador pode aplicar uma otimização chamada *tail call optimization* (TCO). Essa otimização permite que a chamada recursiva seja transformada em um salto direto para a função, evitando a criação de novos quadros na pilha de chamadas. Isso reduz significativamente o consumo de memória e evita o estouro de pilha. Como podemos observar na saída do programa 4.6, a função `tailRecursiveSum` pode lidar com profundidades muito maiores sem causar estouro de pilha, mesmo para valores altos de `b`, como por exemplo, `1_000_000`.

Traço de execução O efeito prático da otimização de chamada na cauda é que, ao invés de empilhar novos quadros na pilha de chamadas, o ambiente de execução reutiliza o quadro atual para a próxima chamada recursiva. Isso significa que a pilha de chamadas não cresce com a profundidade da recursão. O traço de execução para a chamada `tailRecursiveSum(2, 5)` seria:

```
tailRecursiveSum(2, 5, 0)
tailRecursiveSum(3, 5, 2)
tailRecursiveSum(4, 5, 5)
tailRecursiveSum(5, 5, 9)
tailRecursiveSum(6, 5, 14)
(a > b, returns 14)
```

```
def tailRecursiveSum(a: Long, b:
↪ Long, acc: Long = 0): Long = {
    if (a > b) acc
    else tailRecursiveSum(a + 1, b,
    ↪ acc + a)
}
```

Como podemos observar, a pilha de chamadas não cresce com a profundidade da recursão. Em vez disso, o quadro atual é reutilizado para cada chamada recursiva, e o valor do acumulador `acc` é atualizado a cada passo. Isso reduz significativamente o consumo de memória e evita o estouro de pilha. A figura 4.2 ilustra a pilha de execução a cada passo do exemplo `tailRecursiveSum(2, 5)`.

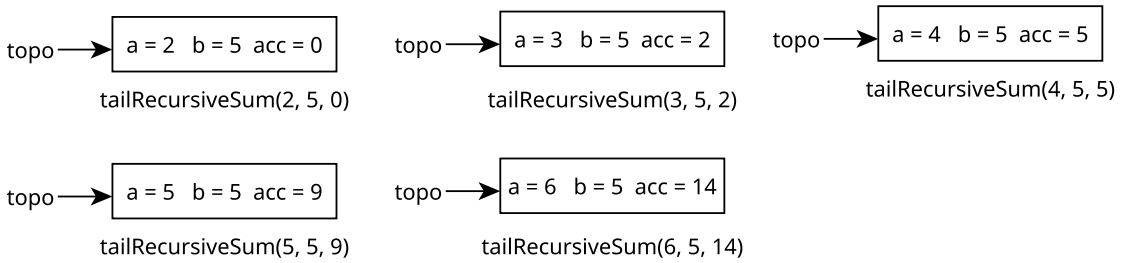

Figura 4.2: Pilha de execução (completa) da chamada `tailRecursiveSum(2, 5)`.

O comportamento esperado da pilha de execução é sempre usar o mesmo quadro cada passo da recursão. Com isso, de modo simplificado, a pilha de execução mantém o tamanho constante e o ambiente de execução é capaz de eliminar o uso excessivo de memória que seria inerente à versão recursiva do algoritmo.

A ANOTAÇÃO `@tailrec` Scala possui uma anotação especial para indicar que uma função deve ser otimizada para recursão na cauda. O efeito prático dessa anotação é que o compilador verifica se a função é realmente recursiva na cauda e, se não for, gera um erro de compilação. Isso ajuda a garantir que a função seja otimizada corretamente e evita problemas de estouro de pilha.

Para verificar esse efeito, basta tentarmos desfazer a recursão na cauda, conforme o exemplo 4.7.

Código 4.7: Somatório recursivo sem recursão na cauda.

```
@tailrec
def tailRecursiveSum(a: Long, b: Long, acc: Long = 0): Long = {
    if (a > b) acc
    else tailRecursiveSum(a + 1, b, acc + a) + 0
}
```

```
Cannot rewrite recursive call: it is not in tail position
```

Como podemos observar, a corretude da função não é comprometida ao somarmos zero ao resultado, pois a operação não afeta o resultado final. Porém, do ponto de vista da execução, não será possível otimizar a chamada. Com isso, o compilador emite uma mensagem de erro. É sempre recomendável usar a anotação `@tailrec` nas chamadas recursivas, pois desse modo o programador é alertado sobre possíveis problemas de desempenho que podem ser gerados por uma modelagem inadequada do problema.

4.3 Conversão de loops para funções recursivas

Em um cenário ideal, podemos propor definições recursivas para todos os problemas, utilizando notação matemática, conforme o exemplo que vimos na fórmula 4.2. No entanto, muitas vezes o problema que estamos solucionando é mapeado trivialmente para esse tipo de notação.

Na prática, podemos facilmente converter um loop imperativo para um algoritmo recursivo, seguindo os seguintes passos:

- As variáveis de controle do loop tornam-se parâmetros da função.
- O corpo do loop torna-se o passo recursivo.
- A condição do loop torna-se o caso base.
- O resultado do loop é o resultado final acumulado no momento em que o caso base é satisfeito.

Por exemplo, considere o loop do exemplo 4.1. Temos duas variáveis mutáveis: o contador `i` e o acumulador `sum`. O primeiro passo da conversão consiste em converter o loop while para uma função recursiva na cauda, conforme o exemplo 4.8.

Código 4.8: Função recursiva auxiliar para emular um loop imperativo.

```
@tailrec
def loopHelper(i: Long, sum: Long, b: Long): Long = {
  if (i > b) sum
  else loopHelper(i + 1, sum + i, b)
}
```

Perceba que a transformação envolveu a representação do contador `i` e do acumulador `sum` como parâmetros da função recursiva `loopHelper`. A cada chamada recursiva, os argumentos realizam as operações que ocorriam no interior do loop original (incremento do contador, e atualização do acumulador). O caso base corresponde à condição de parada original do loop (ou o inverso dela, nesse caso).

Agora basta integrar essa função auxiliar na função principal, conforme o código 4.9.

Código 4.9: Função recursiva completa com função auxiliar emuladora de loop.

```
def declarativeSum(a: Long, b: Long): Long = {
  @tailrec
```

```
  def loopHelper(i: Long, sum: Long, b: Long): Long = {
    if (i > b) sum
    else loopHelper(i + 1, sum + i, b)
  }

  if (a > b) 0
  else loopHelper(a, 0, b)
}

@main def runDeclarativeSum(): Unit = {
  println(declarativeSum(1, 10)) // Output: 55
  println(declarativeSum(5, 15)) // Output: 110
  println(declarativeSum(0, 100)) // Output: 5050
  println(declarativeSum(1, 1_000_000)) // Output: 500000500000
}
```

```
55
110
5050
500000500000
```

No exemplo 4.9, optamos por manter a função auxiliar `loopHelper` como função aninhada de `declarativeSum`. Essa decisão de projeto garante que somente `declarativeSum` tenha acesso a `loopHelper`, o que promove o princípio de ocultação de informação. O corpo da função `declarativeSum`, efetivamente, restringe-se a fazer a verificação inicial da faixa de valores (de modo análogo ao que era feito no algoritmo imperativo) e, por fim, fazer a chamada inicial para `loopHelper`.

Aplicabilidade Para loops de complexidade baixa, ou até média, a técnica de conversão de loops imperativos para recursivos é bastante eficiente e simples. No entanto, para loops mais complexos, ou que envolvam múltiplas variáveis de controle, a conversão pode se tornar mais complicada e menos intuitiva. Nesse caso, é importante avaliar se a conversão realmente traz benefícios em termos de legibilidade e manutenção do código.

Caso a conversão não traga benefícios claros, pode ser mais apropriado repensar o problema conceitualmente para se chegar em uma solução recursiva mais direta. Em linguagens multiparadigma, em particular, uma solução muito comum é simplesmente manter a solução imperativa pois, para alguns problemas, ela é mais eficiente e mais fácil de entender.

4.4 Estudo de caso: fatorial

A função fatorial possui uma definição matemática recursiva bastante simples, conforme a fórmula 4.3.

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1)! & \text{caso contrário} \end{cases} \tag{4.3}$$

Implementação recursiva ingênua Podemos, ingenuamente, fazer a conversão quase direta da definição matemática para uma função recursiva, conforme o programa 4.10.

Código 4.10: Função recursiva para calcular o fatorial de um número.

```
def factorialNaive(n: BigInt): BigInt = {
  if (n <= 1) 1
  else n * factorialNaive(n - 1)
}
@main def runFactorialNaive(): Unit = {
  println(factorialNaive(0))
  println(factorialNaive(1))
  println(factorialNaive(5))
  println(factorialNaive(10))
  println(factorialNaive(21))
}
```

```
1
1
120
3628800
51090942171709440000
```

Embora simples de implementar a partir da definição matemática, essa implementação é computacionalmente ineficiente, pois não apresenta recursão na cauda. Precisamos torná-la recursiva na cauda para evitar o estouro de pilha e melhorar a eficiência do algoritmo.

Implementação recursiva na cauda Podemos aplicar o padrão de projeto *Accumulator* para transformar a função fatorial em uma função recursiva na cauda, conforme o programa 4.11.

Código 4.11: Função recursiva na cauda para calcular o fatorial de um número.

```
def factorialTailRecursive(n: BigInt, acc: BigInt = 1): BigInt = {
  if (n <= 1) acc
  else factorialTailRecursive(n - 1, acc * n)
}
@main def runFactorialTailRecursive(): Unit = {
  println(factorialTailRecursive(0))
  println(factorialTailRecursive(1))
  println(factorialTailRecursive(5))
  println(factorialTailRecursive(10))
  println(factorialTailRecursive(21))
}
```

```
1
1
120
3628800
51090942171709440000
```

VERSÃO ITERATIVA E CONVERSÃO PARA RECURSIVA Em muitas situações, torna-se difícil partir de uma definição recursiva do problema para que consigamos implementar uma solução recursiva. Nesse caso, podemos partir de uma solução imperativa/iterativa e, a partir dela, converter para uma solução recursiva.

O fatorial possui uma definição recursiva simples e direta, que talvez seja mais conhecida do que a definição iterativa. No entanto, podemos definir a função fatorial como um multiplicatório, conforme a fórmula 4.4.

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1 & \text{caso contrário} \end{cases} \tag{4.4}$$

Com base na definição iterativa, podemos implementar o fatorial de forma imperativa, conforme o programa 4.12.

Código 4.12: Função imperativa para calcular o fatorial de um número.

```
def factorialImperative(n: BigInt): BigInt = {
  var result: BigInt = 1
  var i: BigInt = n

  while (i > 1) {
    result *= i
    i -= 1
  }
  result
}
@main def runFactorialImperative(): Unit = {
  println(factorialImperative(0))
  println(factorialImperative(1))
  println(factorialImperative(5))
  println(factorialImperative(10))
  println(factorialImperative(21))
}
```

```
1
1
120
3628800
5109094217170944000
```

Para converter essa função imperativa em uma função recursiva, podemos seguir os passos descritos anteriormente. A variável de controle do loop `i` torna-se um parâmetro da função recursiva, e o corpo do loop torna-se o passo recursivo. O caso base é a condição de parada do loop, e o resultado final é o valor acumulado no momento em que o caso base é satisfeito. O programa 4.13 apresenta a conversão do fatorial imperativo para uma função recursiva na cauda.

Código 4.13: Função recursiva na cauda para calcular o fatorial de um número, a partir de uma função imperativa.

```
def factorialFromImperative(n: BigInt): BigInt = {
```

```
  @tailrec
  def loopHelper(i: BigInt, acc: BigInt): BigInt = {
    if (i <= 1) acc
    else loopHelper(i - 1, acc * i)
  }
  loopHelper(n, 1)
}
@main def runFactorialFromImperative(): Unit = {
  println(factorialFromImperative(0))
  println(factorialFromImperative(1))
  println(factorialFromImperative(5))
  println(factorialFromImperative(10))
  println(factorialFromImperative(21))
}
```

```
1
1
120
3628800
51090942171709440000
```

Como podemos observar, o código 4.13 apresenta uma função auxiliar `loopHelper` que emula o comportamento do loop imperativo. A variável de controle `i` e o acumulador `acc` são passados como parâmetros da função recursiva. O caso base ocorre quando `i` é menor ou igual a 1, momento em que o acumulador `acc` é retornado como resultado final. Podemos constatar que essa versão é muito similar à versão recursiva na cauda do fatorial apresentada no programa 4.11, o que demonstra a flexibilidade da recursão na cauda para emular o comportamento de loops imperativos.

4.5 Estudo de caso: números de Fibonacci

A sequência de Fibonacci é uma famosa sequência matemática definida recursivamente, onde cada número é a soma dos dois números anteriores. A definição matemática da sequência de Fibonacci é apresentada na fórmula 4.5.

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-1) + F(n-2) & \text{caso contrário} \end{cases} \tag{4.5}$$

De acordo com essa definição, a sequência de números de Fibonacci começa com 0 e 1, e os números subsequentes são gerados pela soma dos dois números anteriores. Assim, a sequência é: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

4.5.1 Implementação recursiva ingênua

Podemos implementar a sequência de Fibonacci de forma recursiva, seguindo a definição matemática, conforme o programa 4.14.

Código 4.14: Função recursiva para calcular o n-ésimo número de Fibonacci.

```
def fibonacciNaive(n: BigInt): BigInt = {
  if (n == 0) BigInt(0)
  else if (n == 1) BigInt(1)
  else fibonacciNaive(n - 1) + fibonacciNaive(n - 2)
}

@main def runFibonacciNaive(): Unit = {
  println((0 until 15).map(n => fibonacciNaive(n)))
}
```

```
Vector(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377)
```

Embora essa implementação seja simples e direta, ela é extremamente ineficiente para valores maiores de n, pois a função faz muitas chamadas recursivas redundantes. Esse comportamento anômalo fica claro quando analisamos o traço de execução da chamada `fibonacciNaive(5)`, conforme a figura 4.3.

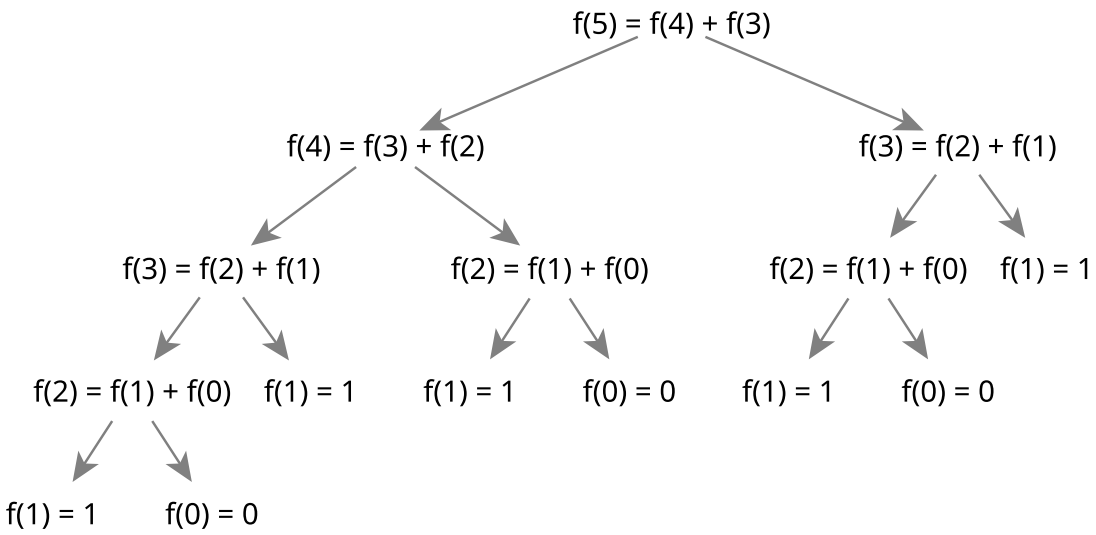

Figura 4.3: Árvore de recursão da chamada `fibonacciNaive(5)`.

COMPLEXIDADE DE TEMPO E ESPAÇO O algoritmo ingênuo utiliza recursão dupla, o que implica em diversas operações repetidas. Como podemos observar na árvore de recursão da figura 4.3, o valor de `fibonacciNaive(3)` é calculado duas vezes, assim como `fibonacciNaive(2)`. Esse comportamento leva a uma complexidade exponencial, $O(2^n)$. Ou seja, no pior caso, as recursões percorreriam uma árvore binária completa de altura n, o que totalizaria $2^n - 1$ chamadas recursivas. Essa complexidade exponencial torna essa implementação impraticável para valores grandes de n.

Além da complexidade de tempo, que é bastante alta, ainda temos o problema da complexidade de espaço, que é $O(n)$ (correspondente à altura da árvore) devido

à profundidade da pilha de chamadas, pois o algoritmo não pode ser otimizado para recursão na cauda.

Outro problema de uma implementação ingênua é que pode não considerar que os números da sequência de Fibonacci crescem rapidamente, o que pode levar a ao estouro da capacidade de inteiros — em Scala, a chamada com n = 47 já resulta em um estouro do inteiro de 32 bits. Para evitar esse problema, utilizamos o tipo `BigInt`, que permite trabalhar com números inteiros de tamanho arbitrário.

4.5.2 Implementação com memoização

Definição de memoização Memoização é uma técnica de otimização que armazena os resultados de chamadas de função para evitar cálculos repetidos. Essa técnica é especialmente útil em algoritmos recursivos, como o cálculo dos números de Fibonacci, onde muitos resultados intermediários são recalculados várias vezes. Na prática, é necessário manter memória adicional para armazenar os resultados intermediários, o que pode ser feito utilizando estruturas de dados de acesso aleatório. A memoização ocorre da seguinte forma:

1. **Cache**. O cache é implementado com uma estrutura de dados que permita acesso em tempo constante aos resultados armazenados, como um array ou um mapa.

2. **Verificação dos argumentos**. Antes de calcular o resultado, verifica-se se os argumentos da função estão presentes no cache.

3. ***Cache hit***. Caso os argumentos já estejam no cache, o resultado é retornado diretamente do cache, evitando o cálculo repetido.

4. ***Cache miss***. Se os argumentos não estiverem no cache, o resultado é calculado normalmente e armazenado no cache para futuras chamadas.

Requisitos da memoização Para que a memoização funcione corretamente, é necessário que a função a ser memoizada atenda aos seguintes requisitos:

- **Transparência referencial**. A função deve sempre retornar o mesmo resultado para os mesmos argumentos. Isso significa que a função não deve ter efeitos colaterais e deve depender apenas dos seus argumentos.

- **Imutabilidade dos argumentos**. Os argumentos da função devem ser imutáveis, ou seja, não devem ser alterados durante a execução da função. Isso garante que o cache possa ser usado de forma segura e eficiente.

- **Operação custosa**. A função deve ser computacionalmente cara, ou seja, o custo de calcular o resultado deve ser significativamente maior do que o custo de acessar o cache. Caso contrário, a memoização não trará benefícios significativos.

- **Operação frequente**. A função deve ser chamada com frequência, de modo que o cache possa ser reutilizado em várias chamadas. Se a função for chamada apenas uma vez, a memoização não trará benefícios significativos.

Aplicações de memoização As técnicas de memoização são bastante conhecidos no paradigma algorítmico de programação dinâmica, onde o objetivo é otimizar algoritmos que apresentam subproblemas sobrepostos. Suas aplicações são bastante diretas também nos problemas recursivos que também apresentam subproblemas sobrepostos, como é o caso do cálculo dos números de Fibonacci.

Estrutura de cache Para implementar memoização, precisamos utilizar uma estrutura auxiliar que atue como cache. A alternativa tradicional consiste em manter uma estrutura mutável de acesso aleatório, pois oferece o compromisso de garantir complexidade de espaço e tempo eficientes. Em Scala, comumente utiliza-se um mapa mutável, que mantém pares chave-valor. O código 4.15 demonstra como utilizar um mapa mutável para armazenar pares chave-valor.

Código 4.15: Exemplo de uso de mapa mutável em Scala.

```scala
import scala.collection.mutable

val cache: mutable.Map[String, Int] = mutable.Map()

// Adicionando valores ao mapa
cache("um") = 1
cache("dois") = 2

// Acessando valores do mapa
println(cache("um")) // Output: 1
println(cache("dois")) // Output: 2
// Verificando se uma chave existe no mapa
println(cache.contains("três")) // Output: false
// Testando e se não existe, adicionando um valor
println(cache.getOrElseUpdate("três", 3)) // Output: 3
println(cache.getOrElseUpdate("três", 4)) // Output: 3
```

```
1
2
false
3
3
```

Função memoizadora A função memoizadora é uma função de ordem superior que recebe uma função como argumento e retorna uma nova função que implementa a lógica de memoização. Para isso, a função memoizadora cria uma closure que contém a estrutura de cache, a função original e a lógica de verificação do cache. O código 4.16 apresenta uma implementação simples de uma função memoizadora.

Código 4.16: Implementação de uma função memoizadora com um argumento.

```scala
def memoize[I, O](f: I => O): I => O = {
  val cache = mutable.Map.empty[I, O]

  (arg: I) => cache.getOrElseUpdate(arg, f(arg))
}
```

A função memoizadora utiliza os tipos genéricos `I` e `O` para representar os tipos de entrada e saída da função original, respectivamente. A função `f` é a função original que será memoizada. A closure criada pela função memoizadora contém o cache, que é um mapa mutável, e a lógica de verificação do cache. A função memoizada, que é retornada pela memoizadora, verifica se o argumento já está no cache e, caso contrário, calcula o resultado e o armazena no cache.

Podemos ver o comportamento de cache hit e cache miss na função memoizada, ao incluirmos linhas de impressão no código 4.16. Para a chamada o cálculo de n = 5, o resultado seria:

```
Cache miss for argument: 1, inserting into cache with value: 1
Cache miss for argument: 0, inserting into cache with value: 0
Cache miss for argument: 2, inserting into cache with value: 1
Cache hit for argument: 1. Value: 1
Cache miss for argument: 3, inserting into cache with value: 2
Cache hit for argument: 2. Value: 1
Cache miss for argument: 4, inserting into cache with value: 3
Cache hit for argument: 3. Value: 2
Cache miss for argument: 5, inserting into cache with value: 5
```

Note que cada valor de Fibonacci é calculado apenas uma vez, e os resultados intermediários são armazenados no cache. Quando um valor já está no cache, a função memoizada simplesmente retorna o valor armazenado, evitando o cálculo redundante. A vantagem de se criar uma função memoizadora é que ela pode ser reutilizada para qualquer função que atenda aos requisitos de memoização, sem a necessidade de reescrever a lógica de cache. Isso promove a reutilização de código e torna a implementação mais modular e flexível. Além disso, não poluímos o código da função a ser memoizada com a lógica de cache, o que melhora a legibilidade e a manutenção do código. Por outro lado, note que a função memoizadora que criamos funciona apenas para funções que recebem um único argumento. Para funções com múltiplos argumentos, seria necessário adaptar a função memoizadora para lidar com múltiplas chaves no cache.

O impacto da memoização no desempenho pode ser observado na figura 4.4, que apresenta a árvore de recursão da chamada `fibonacciNaive(5)` com memoização. Como podemos observar, a árvore de recursão é significativamente reduzida, pois os resultados intermediários são armazenados no cache e reutilizados em chamadas subsequentes. Na figura, representamos os valores de cache hit em azul, enquanto os valores de cache miss estão em preto. Em cinza claro, estão os ramos que foram "podados" da árvore de recursão ingênua. Essa redução no número de chamadas recursivas resulta em uma melhoria significativa no desempenho do algoritmo.

Impureza funcional . A função memoizada é uma closure que mantém uma estrutura de dados mutável (o cache). Isso significa que a função não é pura, pois

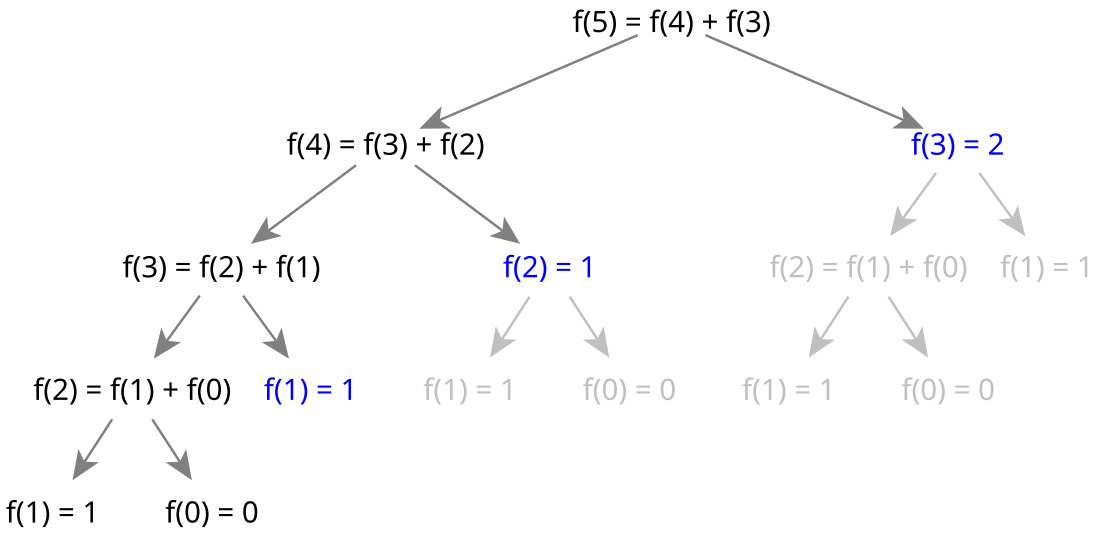

Figura 4.4: Árvore de recursão memoizada da chamada `fibonacciNaive(5)`.

o resultado da função depende do estado do cache. No entanto, essa impureza é aceitável em muitos casos, pois a memoização é uma técnica de otimização que visa melhorar o desempenho de funções computacionalmente caras. Além disso, a função original `f` mantém-se pura. Em suma, essa estatégia é amplamente utilizada, pois os benefícios de desempenho superam os malefícios da impureza funcional.

A mutabilidade do cache é controlada pela função memoizadora, o que permite que a função memoizada seja usada de forma segura em contextos onde a transparência referencial não é estritamente necessária. Por outro lado, a mutabilidade do cache pode levar a problemas de concorrência em ambientes multithreaded, onde múltiplas threads podem acessar e modificar o cache simultaneamente. Nesse caso, é possível utilizar estruturas de dados imutáveis ou técnicas de gerência de efeitos para garantir a consistência do cache. No entanto, essas abordagens podem introduzir complexidade adicional e impactar o desempenho da função memoizada.

Implementação com memoização Com a função memoizadora definida, podemos aplicá-la à função recursiva ingênua de Fibonacci para otimizar o cálculo dos números de Fibonacci. O código 4.17 apresenta a implementação da função de Fibonacci com memoização.

Código 4.17: Função recursiva de Fibonacci com memoização.

```
1 val fibonacciMemoized = memoize { n =>
2   if (n == 0) BigInt(0)
3   else if (n == 1) BigInt(1)
4   else fibonacciMemoized(n - 1) + fibonacciMemoized(n - 2)
5 }
6
7 @main def runFibonacciMemoized(): Unit = {
8   println((0 until 15).map(n => fibonacciMemoized(BigInt(n))))
```

```
}
```

```
Vector(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377)
```

COMPLEXIDADE DE TEMPO E ESPAÇO A complexidade de tempo da função de Fibonacci com memoização é linear, $O(n)$, pois cada número de Fibonacci é calculado apenas uma vez e armazenado no cache. A complexidade de espaço é também linear, $O(n)$, devido ao armazenamento dos resultados intermediários no cache. Essa implementação é muito mais eficiente do que a versão recursiva ingênua, pois evita o cálculo redundante dos números de Fibonacci.

4.5.3 Implementação iterativa

ANÁLISE DA CACHE Ao invés de usarmos um mapa, é comum utilizar um array para armazenar os resultados intermediários dos números de Fibonacci. Isso é possível porque os números de Fibonacci são inteiros não negativos e, portanto, podemos usar o valor de n como índice do array. A vantagem de usar um array é que o acesso aos elementos é feito em tempo constante, $O(1)$, o que torna a implementação mais eficiente em termos de tempo. Além do fato de um array ser mais simples de implementar do que um mapa. Porém, ao analisar o comportamento desse array ao longo de cada chamada, notamos o comportamento a seguir:

```
[1 1]
[1 1 2]
[1 1 2 3]
[1 1 2 3 5]
[1 1 2 3 5 8]
[1 1 2 3 5 8 13]
[1 1 2 3 5 8 13 21]
```

Nessa representação, cada linha representa o estado do array após cada chamda recursiva do loop. Os números em azul representam o valor do n-ésimo número de Fibonacci, enquanto os números em vermelho representam os dois últimos números da sequência. A cada iteração, o array é atualizado com o novo número de Fibonacci, e os dois últimos números são mantidos para o próximo cálculo. Como podemos observar, o array cresce a cada iteração, mas os números de Fibonacci necessários na recursão atual são sempre os dois últimos números da sequência. Isso nos leva a concluir que não precisamos manter todo o array na memória, mas apenas os dois últimos números da sequência.

CACHE COM TAMANHO CONSTANTE A observação anterior nos leva a concluir que podemos manter uma cache de tamanho constante, que armazena apenas os dois últimos números da sequência de Fibonacci. Isso reduz significativamente o uso de memória e melhora a eficiência do algoritmo.

FIBONACCI ITERATIVO. Essa estratégia é difícil de definir matematicamente, portanto vamos apenas apresentar o algoritmo iterativo, conforme o programa 4.18.

Nesse algoritmo, ao invés de usar uma estrutura separada, mantemos apenas duas variáveis para guardar os valores necessários para o cálculo de valor atual.

Código 4.18: Função iterativa para calcular o n-ésimo número de Fibonacci.

```
def fibonacciIterative(n: Int): Int = {
  if (n == 0) return 0
  if (n == 1) return 1

  var a = 0
  var b = 1
  var i = 2

  while (i <= n) {
    val temp = a + b
    a = b
    b = temp
    i += 1
  }

  return b
}

@main def runFibonacciIterative(): Unit = {
  println((0 until 15).map(fibonacciIterative))
}
```

```
Vector(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377)
```

O algoritmo iterativo explora a propriedade de um número de fibonnaci depender apenas dos dois números anteriores. Assim, podemos manter apenas duas variáveis, `a` e `b`, que armazenam os dois últimos números da sequência. A cada iteração do loop, atualizamos essas variáveis para calcular o próximo número da sequência. O resultado final é armazenado na variável `b`, que contém o n-ésimo número de Fibonacci. A figura 4.5 ilustra o traço de execução da chamada `fibonacciIterative(5)`.

Análise de complexidade Como podemos observar, o algoritmo iterativo é muito mais eficiente do que a versão recursiva ingênua. A complexidade de tempo é linear, $O(n)$, pois o loop percorre os números de 2 até `n` uma única vez. A complexidade de espaço é constante, $O(1)$, pois utilizamos apenas duas variáveis para armazenar os números anteriores.

4.5.4 Implementação recursiva na cauda

O grande problema da implementação iterativa é que, para os fins de programação funcional, ela não é pura. Para resolver esse problema, podemos aplicar a técnica de conversão de loops para funções recursivas, conforme discutido na seção anterior. A implementação recursiva na cauda do cálculo dos números de Fibonacci é apresentada no programa 4.19.

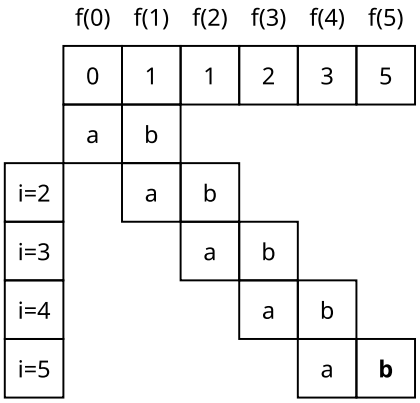

Figura 4.5: Traço de execução da chamada `fibonacciIterative(5)`.

Código 4.19: Função recursiva na cauda para calcular o n-ésimo número de Fibonacci.

```
def fibonacciTailRecursive(n: BigInt): BigInt = {
  @tailrec
  def loop(i: BigInt, a: BigInt, b: BigInt): BigInt = {
    if (i > n) b
    else loop(i + 1, b, a + b)
  }

  if (n == 0) 0
  else if (n == 1) 1
  else loop(2, 0, 1)
}

@main def runFibonacciTailRecursive(): Unit = {
  println((0 until 15).map(x => fibonacciTailRecursive(x)))
}
```

```
Vector(0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377)
```

O loop da versão iterativa executa três mutações: a troca de `a` por `b`, a atualização de `b` para `a + b`, e o incremento de `i`. Essas mutações são realizadas em cada iteração do loop, até que `i` atinja `n`. Adaptamos essas mutações para operações análogas nos argumentos da chamada recursiva, resultando na implementação da função `fibonacciTailRecursive`. Os casos base são tratados diretamente, retornando os valores de `a` e `b` quando `n` é 0 ou 1, respectivamente.

Análise de complexidade Podemos notar que o algoritmo recursivo na cauda ficou bem mais simples e conciso do que a versão iterativa, apesar de ter sido gerado a partir dela. Além disso, por ser recursivo na cauda, o compilador Scala pode aplicar a otimização de chamada na cauda, evitando o estouro de pilha e melhorando a eficiência do algoritmo. Ao final, temos um algoritmo eficiente em espaço ($O(1)$) e tempo ($O(n)$), que também é puramente funcional e declarativo.

4.5.5 Comparação entre as implementações

A tabela 4.1 apresenta uma comparação entre as diferentes implementações do cálculo dos números de Fibonacci, considerando o tempo de execução e a complexidade de espaço e tempo.

Implementação	Complexidade de tempo	Complexidade de espaço
Recursiva ingênua	$O(2^n)$	$O(n)$
Recursiva com memoização	$O(n)$	$O(n)$
Iterativa	$O(n)$	$O(1)$
Recursiva na cauda	$O(n)$	$O(1)$

Tabela 4.1: Comparação entre as implementações do cálculo dos números de Fibonacci.

Podemos também verificar os tempos de execução de cada implementação em um ambiente de configuração típico, o que leva aos resultados da tabela 4.2. Os resultados foram obtidos com base em uma única execução de cada implementação, e os tempos estão em milissegundos (ms) ou microssegundos (µs), conforme apropriado.

Tabela 4.2: Tempos de execução das implementações do cálculo dos números de Fibonacci.

n	Naive Recursive	Memoized	Tail Recursive	Iterative
10	1 ms	6 ms	147 µs	154 µs
20	3 ms	61 µs	1 µs	18 µs
30	35 ms	60 µs	2 µs	16 µs
35	126 ms	33 µs	2 µs	18 µs
40	1 s	38 µs	4 µs	28 µs
45	15 s	32 µs	2 µs	20 µs
50	—	73 µs	8 µs	22 µs
100	—	223 µs	18 µs	62 µs
500	—	1 ms	172 µs	217 µs
1000	—	543 µs	145 µs	154 µs

Como podemos observar, a implementação recursiva ingênua é extremamente ineficiente para valores maiores de `n`, enquanto as implementações com memoização, iterativa e recursiva na cauda apresentam tempos de execução muito mais baixos. As versões iterativa e recursiva na cauda são as mais eficientes e muito próximas em termos de desempenho. A implementação com memoização é eficiente, mas requer mais memória para armazenar os resultados intermediários, além da manutenção do cache consumir mais tempo. A implementação ingênua torna-se proibitiva a partir de `n = 50`, o que faz com que tenhamos omitido os tempos de execução pois a demora é muito alta.

4.6 Resumo

Neste capítulo, aprofundamos o conceito de recursividade como ferramenta central na programação funcional. Vimos como problemas tradicionalmente resolvidos com laços imperativos podem ser elegantemente expressos por meio de funções recursivas, promovendo imutabilidade e clareza na definição dos algoritmos.

Exploramos a diferença entre recursão simples e recursão na cauda, destacando a importância desta última para a eficiência e segurança do código, especialmente em linguagens como Scala, que oferecem otimização específica para esse padrão. O padrão do acumulador mostrou-se fundamental para transformar funções recursivas comuns em funções recursivas na cauda, permitindo o processamento de grandes volumes de dados sem risco de estouro de pilha.

Por meio de exemplos práticos, como somatório, fatorial e números de Fibonacci, demonstramos como a recursão pode ser aplicada a diferentes tipos de problemas, reforçando a correspondência entre definições matemáticas e implementações funcionais.

Ao dominar recursão e suas variações, o estudante está preparado para enfrentar desafios mais avançados em programação funcional, como o uso de tipos de dados algébricos e técnicas de pattern matching, que serão abordados nos próximos capítulos. A recursão não é apenas uma técnica de programação, mas uma forma de pensar sobre problemas e soluções, essencial para o desenvolvimento de software robusto e eficiente.

Aviso de licença de uso