Capítulo 6

Tipos de dados algébricos

Os tipos de dados algébricos são um conceito fundamental na programação funcional e na teoria dos tipos. Eles representam um recurso poderoso para modelar dados complexos de forma segura e expressiva. O termo *algébrico* refere-se à capacidade de serem manipulados de modo análogo às operações algébricas: soma (para escolhas restritas) e produto (para combinações estruturadas).

Os tipos de dados algébricos possuem uma definição combinatória dos tipos envolvidos, ou seja, são definidos em termos da cardinalidade do domínio dos tipos envolvidos. Em outras palavras, eles são definidos com base no número de valores possíveis que podem ser combinados ou escolhidos entre os tipos envolvidos.

- **Tipos soma**. O tipo soma é definido como uma união discriminada de tipos, ou seja, uma variável que pode assumir valores de diferentes tipos, mas apenas um de cada vez.

  Se $A, B, \ldots, Z$ são tipos com $n, m, \ldots, z$ valores possíveis, respectivamente, então o tipo soma $A+B+\ldots+Z$ terá $n+m+\ldots+z$ valores possíveis. Isso significa que uma variável desse tipo pode conter um valor de A, um valor de B, e assim por diante, mas não pode conter valores de mais de um tipo ao mesmo tempo.

- **Tipos produto**. O tipo produto é definido como uma agregação de diferentes tipos, onde cada tipo pode ser composto por outros tipos, formando uma estrutura mais complexa.

  Se $A, B, \ldots, Z$ são tipos com $n, m, \ldots, z$ valores possíveis, respectivamente, então o tipo produto $A \times B \times \ldots \times Z$ terá $n \times m \times \ldots \times z$ valores possíveis. Isso significa que uma variável desse tipo pode conter um valor de A e um valor de B, e assim por diante, ao mesmo tempo, permitindo combinações de valores de todos os tipos envolvidos.

Tipos abstratos vs. tipos algébricos É importante distinguir entre tipos abstratos e tipos algébricos:

- **Tipos abstratos**: São tipos definidos por interfaces ou traits, que especificam um conjunto de operações que podem ser realizadas sobre eles, mas não fornecem uma implementação concreta. Eles são usados para definir contratos estruturais, permitindo que diferentes implementações sejam usadas de forma intercambiável.

- **Tipos algébricos**: São tipos concretos que combinam outros tipos de dados, permitindo a construção de estruturas complexas. Eles são usados para modelar dados tipados de modo seguro, associados a recursos robustos de manipulação de dados, como casamento de padrões e recursão estrutural.

De modo resumido, podemos dizer que os tipos abstratos especificam o *comportamento* de um tipo de dados, enquanto os tipos algébricos especificam a *estrutura* e a composição dos dados de um tipo. Inclusive, é perfeitamente possível combinar ambos os conceitos, criando tipos abstratos que utilizam tipos algébricos como base para suas implementações.

Tipos algébricos em Scala Por ser uma linguagem que se apoia na forte integração entre orientação a objetos e programação funcional, Scala implementa os tipos algébricos também fortemente apoiada em recursos de orientação a objetos. Dentre esses recursos podemos mencionar as `case classes` e os `sealed traits`, que serão analisados nas próximas seções.

6.1 Tipos produto

Um tipo produto é uma estrutura que combina múltiplos valores (componentes) de diferentes tipos em uma única entidade. Uma instância de um tipo produto contém todos os valores de seus componentes, e cada componente pode ser acessado individualmente.

Conjunção A lógica utilizada nos tipos produto é baseada na conjunção lógica (AND), daí o nome ”produto”. Isso significa que todos os componentes devem estar presentes simultaneamente para formar uma instância válida do tipo produto. Por exemplo, se definimos um tipo produto sobre os tipos A, B e C, isso significa que uma instância desse tipo produto conterá um valor do tipo A, um valor do tipo B e um valor do tipo C ao mesmo tempo.

Produto cartesiano O produto cartesiano é uma operação matemática que combina dois conjuntos para formar um novo conjunto, onde cada elemento do novo conjunto é uma tupla contendo um elemento de cada conjunto original. Em termos de tipos de dados, o produto cartesiano é representado por tipos produto, onde cada componente da tupla corresponde a um tipo diferente. Se o tipo A tem n valores possíveis e o tipo B tem m valores possíveis, então o tipo-produto é o produto cartesiano $A \times B$, que terá $n \times m$ combinações possíveis.

Tipos produto em Scala Scala possui dois recursos principais para representar tipos produto: as `case classes` e as tuplas. Analisaremos cada um deles a seguir.

6.1.1 Tuplas

As tuplas são uma forma mais leve, mas também menos poderosa, de representar tipos produto em Scala. Elas permitem agrupar valores de diferentes tipos em uma estrutura. A sintaxe para criar uma tupla é simples, como mostrado no código 6.1.

Código 6.1: Criação de uma tupla chamada `point` contendo dois valores: 3.0 e 4.0.

```
val point = (3.0, 4.0)
```

No exemplo do código 6.1, a tupla `point` contém dois valores, 3.0 e 4.0, que podem ser acessados usando a sintaxe de índice, como `point._1` e `point._2`, conforme mostrado no código 6.2.

Código 6.2: Acesso aos elementos de uma tupla.

```
println(point._1) // Acessa o primeiro elemento da tupla
println(point._2) // Acessa o segundo elemento da tupla
```

As tuplas são imutáveis, portanto não é possível alterar seus elementos após a criação. Se precisarmos de uma nova tupla com valores modificados, devemos criar uma nova instância, conforme o código 6.3.

Código 6.3: Criação de uma nova tupla com um elemento modificado.

```
val updatedPoint = point.copy(_2 = 5.0) // Cria uma nova tupla com o segundo
↪  elemento modificado
println(updatedPoint)
```

Além disso, as tuplas podem conter valores de tipos diferentes, o que as torna bastante adequadas para representar os tipos produto. Considere por exemplo o código 6.4, que cria uma tupla contendo um inteiro, um string e um booleano.

Código 6.4: Criação de uma tupla com elementos de tipos diferentes.

```
val mixedTuple = (42, "Hello", true)
println(mixedTuple._1) // Acessa o primeiro elemento (Int)
println(mixedTuple._2) // Acessa o segundo elemento (String)
println(mixedTuple._3) // Acessa o terceiro elemento (Boolean)
```

O tipo inferido de `mixedTuple` é `(Int, String, Boolean)`. Adotando a definição de produto cartesiano, o tipo `(Int, String, Boolean)` representa o produto cartesiano dos tipos `Int`, `String` e `Boolean`, ou seja, todas as combinações possíveis desses três tipos.

Comparação com outras estruturas De início, podemos pensar que qualquer estrutura que combina diferentes tipos de dados pode ser considerada um tipo produto. Por exemplo, uma lista que pode armazenar diferentes tipos de dados, como `List[Int|String]`, poderia ser vista como um tipo produto. No entanto, essa premissa está incorreta, pois um tipo produto deve garantir as seguintes condições:

- **Número fixo de componentes**: Um tipo produto deve ter um número fixo de componentes, cada um com um tipo específico. Por exemplo, uma tupla de dois elementos tem exatamente dois componentes, enquanto uma lista pode ter qualquer número de elementos. Numa lista, não é possível determinar qual o produto cartesiano dos tipos dos elementos em tempo de compilação, pois o número de elementos só pode ser determinado em tempo de execução.

- **Ordem dos componentes é relevante**: A ordem dos componentes em um tipo produto é relevante e deve ser respeitada. Em uma tupla, a ordem dos componentes é fixa e conhecida, enquanto em uma lista a ordem dos elementos pode ser alterada, o que não garante a preservação da ordem dos componentes. Por exemplo, o tipo `(Int, String)` é diferente de `(String, Int)`, pois a ordem dos componentes é importante. Em contraste, uma lista não possui essa restrição, e a ordem dos elementos pode ser alterada sem afetar o tipo da lista.
- **Todos os componentes presentes**: Um tipo produto deve garantir que todos os seus componentes estejam presentes em tempo de compilação. Em uma tupla, todos os componentes são obrigatórios e devem ser fornecidos na criação da instância. Em contraste, uma lista pode ser vazia ou conter um número variável de elementos, o que não garante a presença de todos os componentes.

Embora tenhamos usado as listas como contra-exemplo, o mesmo raciocínio vale para outras estruturas de dados, como arrays, conjuntos ou mapas, que também não garantem um número fixo de componentes ou o tipo determinístico dos elementos.

Limitações Embora as tuplas tenham todas as características necessárias para representar tipos produto (como imutabilidade, acesso aos componentes e suporte a tipos diferentes), elas não possuem recursos desejáveis para a manipulação de dados complexos de modo conciso e expressivo. Devido a essas limitações, os tipos produto em Scala são geralmente representados por `case classes`, que analisaremos na próxima seção.

6.1.2 `case classes`

As `case classes` representam uma abordagem orientada a objetos para representar os tipos produto. A sintaxe para definir uma `case class` é bastante simples, conforme ilustra o código 6.5.

Código 6.5: Definição de uma `case class` chamada `Point` com dois componentes: `x` e `y`.

```
1 case class Point(x: Double, y: Double)
```

Quando criamos uma `case class` em Scala, o compilador gera automaticamente os seguintes recursos:

- **Imutabilidade**: As instâncias de `case class` são imutáveis por padrão, o que significa que os componentes do tipo produto são representados como propriedades imutáveis da classe. Note que, diferentemente de classes regulares, não é necessário definir explicitamente os modificadores de propriedades `val` para os componentes, pois o compilador já assume que são propriedades imutáveis da classe.
- **Construtor primário**: Um construtor primário público que permite criar instâncias da `case class` com os valores dos componentes.

- **Persistência**: O método `copy` permite criar uma nova instância da `case class` com alguns componentes modificados, mantendo os demais inalterados.

- **Métodos de acesso**: Acessores para os valores dos componentes, utilizando o *Uniform Access Principle* (UAP).

- **Método aplicador**: O compilador gera um companion object que permite criar instâncias da `case class` usando a sintaxe de chamada de função, como se fosse uma função normal. Por exemplo, podemos criar uma instância de `Point` usando `Point(3.0, 4.0)`.

- **Métodos de comparação**: Implementação automática dos métodos `equals` e `hashCode`, permitindo comparar instâncias da `case class` com base nos valores dos componentes.

- **Método `toString`**: Uma representação em string da instância, facilitando a depuração e o registro.

O código 6.6 ilustra como instanciar e utilizar a `case class` `Point` definida anteriormente.

Código 6.6: Exemplo de uso da `case class` `Point`.

```
1  val p1 = Point(3.0, 4.0)
2  val p2 = Point(5.0, 6.0)
3
4  println(p1)
5  println(p1.x)
6  println(p1.y)
7  println(p1 == p2)
8
9  val p3 = p1.copy(y = 5.0)
10
11 println(p1)
12 println(p2)
13 println(p3)
```

```
Point(3.0,4.0)
3.0
4.0
false
Point(3.0,4.0)
Point(5.0,6.0)
Point(3.0,5.0)
```

Em resumo, as `case classes` são uma maneira poderosa e expressiva de representar tipos produto em Scala, oferecendo recursos avançados para manipulação de dados complexos. Elas permitem criar tipos de dados imutáveis com propriedades bem definidas, facilitando a construção de estruturas de dados robustas e seguras.

6.2 Tipos soma

Um tipo soma é uma estrutura que permite definir um valor que pode pertencer a diferentes tipos, mas apenas um tipo em um dado momento.

UNIÃO DISJUNTIVA A lógica utilizada nos tipos soma é baseada na união disjuntiva, daí o nome "soma". A união disjuntiva é similar ao operador lógico XOR, porém estendida para mais de dois operandos. Isso significa que uma instância de um tipo soma pode ser de um tipo ou de outro, mas não ambos ao mesmo tempo.

Por exemplo, se definimos um tipo soma sobre os tipos A e B, isso significa que uma instância desse tipo soma conterá um valor do tipo A ou um valor do tipo B, mas não ambos simultaneamente. Podemos dizer que os tipos são mutuamente exclusivos, ou seja, não podem ser combinados em uma única instância.

SOMA DE DOMÍNIOS A soma de domínios é uma operação matemática que combina dois conjuntos para formar um novo conjunto, onde cada elemento do novo conjunto é um elemento de um dos conjuntos originais. Se o tipo A tem n valores possíveis e o tipo B tem m valores possíveis, então o tipo-soma é a soma de domínios A + B, que terá n + m combinações possíveis.

Os tipos soma são frequentemente usados para representar valores que podem ser de diferentes tipos, como resultados de operações que podem falhar ou eventos que podem ocorrer em diferentes estados. Em Scala, os tipos soma são geralmente representados pelo uso combinado de `sealed traits` e `case classes` ou `case objects`, que garantem que todas as subclasses sejam conhecidas em tempo de compilação.

6.2.1 Definição de tipos soma

Um tipo soma é definido como um tipo que pode conter um valor de um conjunto finito de tipos possíveis. Em Scala, isso é frequentemente alcançado usando um `sealed trait` para definir o tipo soma e, em seguida, criando subclasses (geralmente `case classes` ou `case objects`) para cada um dos tipos possíveis. Isso garante que todas as subclasses sejam conhecidas em tempo de compilação, permitindo que o compilador verifique a exaustividade em correspondências de padrões.

Por exemplo, considere o código 6.7, que define um tipo soma para representar resultados de uma operação que pode ser bem-sucedida ou falhar.

Código 6.7: Definição de um tipo soma usando `sealed trait` e `case classes`.

```scala
sealed trait Result[T]
case class Success[T](value: T) extends Result[T]
case class Failure[T](error: String) extends Result[T]
```

No exemplo do código 6.7, o tipo soma `Result[T]` pode ser de dois tipos: `Success[T]`, que contém um valor do tipo `T`, ou `Failure`, que contém uma mensagem de erro. Podemos usar esse tipo soma para representar o resultado de uma operação que pode ser bem-sucedida ou falhar, como mostrado no código 6.8.

Código 6.8: Exemplo de uso do tipo soma `Result[T]`.

```scala
def divide(a: Int, b: Int): Result[Int] = {
  if (b == 0) Failure("Division by zero")
  else Success(a / b)
}

@main def testResult = {
    println(divide(10, 2))
    println(divide(10, 0))
}
```

```
Success(5)
Failure(Division by zero)
```

`sealed trait` E EXAUSTIVIDADE O uso de `sealed` antes da definição do `trait` restringe a extensão do trait apenas às subclasses definidas no mesmo arquivo. Isso garante que todas as subclasses de `Result[T]` sejam conhecidas em tempo de compilação, permitindo que o compilador verifique a exaustividade dos subtipos, o que é importante para operacionalizar o casamento de padrões (veremos essa técnica adiante). Em suma, o `sealed trait` permite definir uma hierarquia fechada de tipos, garantindo que todas as possibilidades sejam tratadas em correspondências de padrões.

TIPOS SOMA VS. POLIMORFISMO DE SUBTIPOS O polimorfismo de subtipos é frequentemente usado para representar diferentes comportamentos de um tipo. Ou seja, definimos um tipo base (supertipo) e criamos subtipos que estendem esse tipo base, cada um com seu próprio comportamento. Em orientação a objetos, os supertipos são definidos como superclasses e os subtipos são definidos como subclasses. Em outros paradigmas, as técnicas de implementação variam. Embora sejam conceitos que possuam semelhanças, os tipos soma e o polimorfismo de subtipos são usados para propósitos diferentes.

Enquanto o polimorfismo de subtipos define uma hierarquia aberta, que pode ser estendida por outras classes em outros arquivos, os tipos soma definem uma hierarquia fechada, onde todas as subclasses são conhecidas em tempo de compilação. Isso permite que o compilador verifique a exaustividade das correspondências de padrões, garantindo que todos os casos possíveis sejam tratados. Por fim, é perfeitamente possível utilizar polimorfismo de subtipos em conjunto com tipos soma, mas é importante entender as diferenças entre os dois conceitos.

PADRÃO DE PROJETO O padrão de projeto adotado para criar tipos soma em Scala é definido do seguinte modo:

- **Definir um `sealed trait`**: Crie um `sealed trait` que represente o tipo soma. Esse trait servirá como a base para os diferentes tipos possíveis. É naturalmente um tipo abstrato, para o qual deveremos definir exaustivamente todos os subtipos concretos possíveis.
- **Criar subclasses com `case classes` ou `case objects`**: Para cada tipo possível no tipo soma, crie uma subclasse usando `case class` ou `case object`. A

decisão de uso de `case class` ou `case object` pode ser defendida como segue:

- **`case class` para múltiplas instâncias**: Se o tipo soma precisa armazenar informações adicionais, como um valor ou uma mensagem de erro, use `case class`. Isso vai permitir que haja instâncias do tipo. Por exemplo, `Success[T]` contém um valor do tipo `T`, enquanto `Failure` contém uma mensagem de erro do tipo `String`.
- **`case object` para única instância**: Se o tipo soma não precisa armazenar informações adicionais e é apenas um marcador ou um estado, use `case object`. Isso é útil para representar estados que não têm dados associados, como `Failure` sem um valor de erro específico. Essa restrição deve-se ao fato do object ser um tipo singleton, ou seja, não é possível criar instâncias de `case object`, o que é desejável quando o tipo não contém dados.

Composição entre tipos produto e tipos soma É importante notar que os tipos soma podem conter tipos produto como componentes. Naturalmente, no exemplo do código 6.7, o tipo `Success[T]` é um tipo produto, pois contém um valor do tipo `T`. Analogamente, `Failure` é um tipo produto que contém uma mensagem de erro do tipo `String`.

Simetricamente, um tipo produto pode conter tipos soma como componentes. Por exemplo, considere o código 6.9, que define um tipo produto que contém um valor do tipo `Result[T]` como um de seus componentes.

Código 6.9: Definição de um tipo produto que contém um tipo soma como componente.

```
1 case class OperationResult[T](result: Option[T], timestamp: Long)
```

No exemplo do código 6.9, o tipo produto `OperationResult[T]` contém um componente do tipo `Option[T]`, que é um tipo soma. Isso demonstra a flexibilidade dos tipos algébricos em Scala, permitindo combinar tipos de forma expressiva e segura, analogamente à composição de tipos em álgebra.

6.2.2 Exemplos de tipos soma

Tipos de resultado A biblioteca-padrão de Scala está repleta de tipos soma, alguns dos quais já usamos extensivamente. Por exemplo, o código 6.10 reproduz, de modo simplificado, os tipos `Option`, `Either` e `Try`, que são tipos soma amplamente utilizados para o tratamento de erros em Scala.

Código 6.10: Exemplos (simplificados) de tipos soma na biblioteca-padrão Scala.

```
1 sealed trait Option[T]
2 case class Some[T](value: T) extends Option[T]
3 case object None extends Option[Nothing]
```

```scala

sealed trait Either[A, B]
case class Left[A](value: A) extends Either[A, Nothing]
case class Right[B](value: B) extends Either[Nothing, B]

sealed abstract class Try[T]
case class Success[T](value: T) extends Try[T]
case class Failure[T](exception: Throwable) extends Try[T]
```

Já exploramos extensivamente os tipos de resultado `Option` e `Either`. O tipo de resultado `Try` é um tipo soma que representa uma operação que pode ser bem-sucedida ou falhar, semelhante ao `Either`, mas com foco em capturar exceções. A vantagem do tipo `Try` é que permite lidar com código legado imperativo que lança exceções, sem de fato implicar no efeito colateral de desvio de fluxo. Como consequência, o código legado torna-se mais declarativo.

O código 6.11 ilustra o uso do tipo `Try`.

Código 6.11: Exemplo de uso do tipo `Try`.

```scala
import scala.util.{Try, Success, Failure}

def divide(a: Int, b: Int): Try[Int] = Try {
  a / b
}

@main def run = {
    println(divide(10, 2))
    println(divide(10, 0))
    println(divide(10, -2))
}
```

```
Success(5)
Failure(java.lang.ArithmeticException: / by zero)
Success(-5)
```

LISTAS É bastante comum definir estruturas de dados complexas como tipos soma. Por exemplo, a própria estrutura `List` do pacote `scala.collection` é definida como um tipo soma. No código 6.12, temos uma tentativa (bem mais simplificada que a representação nativa de Scala) de representar uma lista encadeada como um tipo soma. Essa definição permite representar listas de forma recursiva, onde cada elemento da lista pode ser um valor do tipo `T` ou a lista vazia (`Nil`).

Código 6.12: Definição de uma lista como um tipo soma.

```scala
sealed trait MyList[+A]
case object MyNil extends MyList[Nothing]
case class Cons[A](head: A, tail: MyList[A]) extends MyList[A]
```

Com base nessa definição de tipo soma, pois podemos criar listas encadeadas de forma recursiva, como mostrado no código 6.13.

Código 6.13: Exemplo de uso da lista definida como tipo soma.

```scala
val list: MyList[Int] = Cons(1, Cons(2, Cons(3, Nil)))

println(list) // Exibe a lista
```

```
Cons(1,Cons(2,Cons(3,MyNil)))
```

Outra estrutura de dados definida como um tipo algébrico é `Stream`, que representa uma lista "preguiçosa" (lazy) de elementos, permitindo a avaliação sob demanda. Estudaremos essa estrutura mais adiante, quando falarmos sobre avaliação preguiçosa.

6.3 Casamento de padrões

O casamento de padrões (*pattern matching*) é uma característica poderosa das linguagens que suportam tipos algébricos. Ele permite que os desenvolvedores escrevam código que lida com diferentes formas de dados de maneira concisa e expressiva. O casamento de padrões funciona examinando um tipo algébrico e executando diferentes blocos de código com base em sua estrutura.

Motivação Na ausência de casamento de padrões, precisamos usar condicionais `if-else` para lidar com diferentes tipos ou estados. Por exemplo, considere o tipo soma para formas geométricas, conforme o código 6.14.

Código 6.14: Definição de um tipo soma para formas geométricas.

```scala
sealed trait Shape
case class Circle(radius: Double) extends Shape
case class Rectangle(width: Double, height: Double) extends Shape
case class Triangle(base: Double, height: Double) extends Shape
```

Suponha que desejamos projetar uma função capaz de calcular a área de qualquer forma geométrica. Uma primeira tentativa poderia ser a seguinte, conforme o código 6.15.

Código 6.15: Cálculo da área de uma forma geométrica usando condicionais `if-else`.

```scala
def area(shape: Shape): Option[Double] = {
  if (shape.isInstanceOf[Circle]) {
    val circle = shape.asInstanceOf[Circle]
    Some(Math.PI * circle.radius * circle.radius)
  } else if (shape.isInstanceOf[Rectangle]) {
    val rectangle = shape.asInstanceOf[Rectangle]
    Some(rectangle.width * rectangle.height)
  } else if (shape.isInstanceOf[Triangle]) {
    val triangle = shape.asInstanceOf[Triangle]
    Some(0.5 * triangle.base * triangle.height)
  } else {
```

```
    None
  }
}

@main def runAreaCalculations(): Unit = {
  val shapes: List[Shape] = List(
    Circle(5),
    Rectangle(4, 6),
    Triangle(3, 7)
  )

  shapes.foreach { shape =>
    val result = area(shape)

    println(s"Área: ${result.getOrElse("Desconhecida")}")
  }
}
```

```
Área: 78.53981633974483
Área: 24.0
Área: 10.5
```

DESVANTAGENS DO USO DE CONDICIONAIS Embora o código acima funcione, ele apresenta algumas desvantagens:

- **Verificações de tipo em tempo de execução**: O uso de `isInstanceOf` e `asInstanceOf` introduz verificações de tipo em tempo de execução, o que pode levar a erros se não forem tratadas corretamente. Por exemplo, se um novo tipo for adicionado ao tipo soma `Shape` (por exemplo, `Square`) e não for tratado na função `area`, o código falhará em tempo de execução.
- **Falta de exaustividade**: O compilador não pode garantir que todos os casos possíveis foram tratados na seleção encadeada, pois pode haver novos tipos adicionados ao tipo soma `Shape` no futuro. Isso pode levar a erros sutis e difíceis de depurar.

CASAMENTO DE PADRÕES O casamento de padrões resolve esses problemas, permitindo que o compilador verifique a exaustividade e a segurança de tipos em tempo de compilação. A seguir, o código 6.27 mostra como reescrever a função `area` usando casamento de padrões.

6.3.1 Expressões `match`

O principal recurso para casamento de padrões em Scala é a expressão `match`, que permite comparar um valor com diferentes padrões e executar o código correspondente ao padrão que for correspondido. A sintaxe do casamento de padrões é bastante concisa e expressiva, permitindo escrever código claro e legível. A sintaxe básica da expressão `match` é a seguinte:

```
pattern match {
  case pattern1 => expression1
  case pattern2 => expression2
  case pattern3 => expression3
  case _ => defaultExpression
}
```

Onde:

- `pattern` é o padrão a ser correspondido, que pode ser um tipo algébrico, uma tupla, uma lista, etc.
- `expression` é a expressão a ser avaliada se o padrão corresponder.
- O `case _` é um caso padrão que corresponde a qualquer valor não tratado pelos casos anteriores.

Além da expressão `match`, o casamento de padrões também pode ser usado com a sintaxe de `case` em definições de funções, e na desestruturação de tipos produto. A seguir, veremos alguns exemplos.

6.4 Casamento de padrões com tipos produto

Desestruturação Em tipos produto, o casamento de padrões permite extrair os seus componentes individuais. Essa operação é denominada desestruturação ou `destructuring`. Por exemplo, considere a `case class Point` definida anteriormente. Podemos usar casamento de padrões para extrair os valores `x` e `y` de um ponto, conforme o código 6.16.

Código 6.16: Casamento de padrões em tipos produto.

```
case class Point(x: Double, y: Double) {
  override def toString: String = this match {
    case Point(x, y) => s"Coordenadas: ($x, $y)"
  }
}

@main def runPointExample(): Unit = {
  val point = Point(3.0, 4.0)
  println(point.toString)
}
```

```
Coordenadas: (3.0, 4.0)
```

Nesse exemplo, usamos a destruturação em dois momentos dentro do método `toString` da `case class Point`

O método `apply` O método `apply` é gerado automaticamente para `case classes` e permite criar instâncias da classe de forma concisa. Por exemplo, pode-

mos criar um ponto usando `Point(3.0, 4.0)` sem precisar usar o construtor explícito. A sintaxe de desestruturação é baseada na chamada do método `apply` da `case class`, que é chamado implicitamente quando usamos `case Point(x, y)`. Isso permite extrair os valores dos componentes do tipo produto de forma concisa e legível.

O MÉTODO `unapply` O casamento de padrões em tipos produto é possível porque o compilador gera automaticamente um método `unapply` para `case classes`. Quando o compilador encontra a sintaxe do método `apply` dentro de um padrão, ele chama o método `unapply` correspondente para extrair os valores dos componentes do tipo produto. No exemplo acima, o método `unapply` é chamado implicitamente quando usamos `case Point(x, y)`.

DESESTRUTURAÇÃO EXPLÍCITA O uso de `case classes` para definir tipos algébricos aptos a participar de casamento de padrões é apenas uma conveniência, pois podemos utilizar casamento de padrões com qualquer tipo que implemente o método `unapply`. Por exemplo, podemos definir o tipo Point2 como uma classe regular e com um companion object, conforme o código 6.17.

Código 6.17: Definição de um tipo produto com método `unapply`.

```
class Point2(val x: Double, val y: Double)

object Point2 {
  def apply(x: Double, y: Double): Point2 = new Point2(x, y)

  def unapply(point: Point2): (Double, Double) = (point.x, point.y)
}

@main def testPoint2 = {
  val point = Point2(3.0, 4.0)
  val Point2(x, y) = point

  println(s"Point coordinates: x = $x, y = $y")
}
```

```
Point coordinates: x = 3.0, y = 4.0
```

Observe que, neste exemplo, definimos explicitamente o método `unapply` no companion object `Point2`. Esse método recebe uma instância de `Point2` e retorna uma tupla com os valores `x` e `y`. Isso permite que o casamento de padrões funcione da mesma forma que com `case classes`, mas sem a necessidade de usar a sintaxe de `case class`.

PADRÕES ANINHADOS A desestruturação também pode ser aplicada a tipos produto aninhados. Por exemplo, considere o código 6.18, que calcula a área de uma forma geométrica, mas agora usando casamento de padrões aninhado para extrair os componentes de tipos produto aninhados.

Código 6.18: Casamento de padrões aninhado em tipos produto.

```scala
case class Address(street: String, city: String)
case class Customer(name: String, address: Address)

def customerInfo(customer: Customer): String = customer match {
  case Customer(name, Address(street, city)) =>
    s"Nome: $name\nEndereço: $street, $city"
}

@main def runCustomerExample(): Unit = {
  val customer = Customer("Alice", Address("123 Main St", "Wonderland"))
  println(customerInfo(customer))
}
```

```
Nome: Alice
Endereço: 123 Main St, Wonderland
```

AMARRAÇÃO DO TIPO PRODUTO Além de amarrar os componentes do tipo produto, o casamento de padrões também permite amarrar o tipo produto em si. Por exemplo, podemos usar desestrutuação para extrair os componentes de um tipo produto e, ao mesmo tempo, amarrar o tipo produto a uma variável. O código 6.19 ilustra essa técnica.

Código 6.19: Amarração do tipo produto em casamento de padrões.

```scala
case class Point(x: Double, y: Double)

def printPointInfo(point: Point): String = point match {
  case p @ Point(x, y) =>
    s"Point info: ${p}, x: $x, y: $y"
}

@main def runPointExample(): Unit = {
  val point = Point(1.0, 2.0)
  println(printPointInfo(point))
}
```

```
Point info: Point(1.0,2.0), x: 1.0, y: 2.0
```

Em resumo, o casamento de padrões em tipos produto permite extrair os componentes individuais de um tipo produto, facilitando a manipulação e o acesso aos seus valores. A desestruturação é uma técnica poderosa que torna o código mais legível e expressivo, eliminando a necessidade de verificações de tipo explícitas. Além disso, o casamento de padrões pode ser usado com qualquer tipo que implemente o método `unapply`, permitindo uma flexibilidade adicional na definição de tipos algébricos.

6.5 Casamento de padrões com tipos soma

O casamento de padrões com tipos soma permite lidar com as diferentes variantes de um tipo soma de modo exaustivo. Por exemplo, vamos recapitular o tipo soma `Result[T]` definido anteriormente, que representa um resultado de operação que pode ser bem-sucedida ou falhar. Podemos usar casamento de padrões para lidar com essas duas variantes, conforme o código 6.20.

Código 6.20: Casamento de padrões com tipos soma.

```
def handleResult[T](result: Result[T]): String = result match {
  case Success(value) => s"Operação bem-sucedida: $value"
  case Failure(error) => s"Operação falhou: $error"
}
@main def runResultExample(): Unit = {
  val successResult: Result[Int] = Success(42)
  val failureResult: Result[Int] = Failure("Erro desconhecido")

  println(handleResult(successResult))
  println(handleResult(failureResult))
}
```

```
Operação bem-sucedida: 42
Operação falhou: Erro desconhecido
```

Exaustividade garantida O casamento de padrões garante que todos os casos possíveis sejam tratados. Se um novo tipo for adicionado ao tipo soma `Result[T]`, o compilador apontará onde o código precisa ser atualizado para lidar com a nova variante. Isso ajuda a evitar erros comuns de não tratamento de casos, melhorando a robustez do código. Por exemplo, considere que adicionamos uma nova variante `Pending` ao tipo soma `Result[T]`, conforme o código 6.21.

Código 6.21: Adicionando uma nova variante ao tipo soma `Result[T]`.

```
sealed trait Result[T]
case class Success[T](value: T) extends Result[T]
case class Failure[T](error: String) extends Result[T]
case class Pending[T](message: String) extends Result[T]
def handleResult[T](result: Result[T]): String = result match {
  case Success(value) => s"Operação bem-sucedida: $value"
  case Failure(error) => s"Operação falhou: $error"
  case Pending(message) => s"Operação pendente: $message"
}
@main def runResultExample(): Unit = {
  val successResult: Result[Int] = Success(42)
  val failureResult: Result[Int] = Failure("Erro desconhecido")
  val pendingResult: Result[Int] = Pending("Aguardando confirmação")

  println(handleResult(successResult))
  println(handleResult(failureResult))
```

```
  println(handleResult(pendingResult))
}
```

```
Operação bem-sucedida: 42
Operação falhou: Erro desconhecido
Operação pendente: Aguardando confirmação
```

Perceba que fomos obrigados a atualizar a função `handleResult` para lidar com a nova variante `Pending`. O compilador sinaliza onde o código precisa ser atualizado, garantindo que todos os casos sejam tratados. Se, por outro lado, comentarmos a linha `{case Pending(message) => s"Operação pendente: $message"}`, o compilador emitirá um aviso de que o caso `Pending` não foi tratado, alertando-nos sobre a falta de exaustividade.

APLICAÇÃO: MÁQUINA DE ESTADOS O código 6.22 ilustra como modelar uma máquina de estados finitos (FSM) usando tipos soma e a expressão `match`.

Código 6.22: Máquina de estados finitos (FSM) usando tipos soma e casamento de padrões.

```
sealed trait TrafficLight
case object Red extends TrafficLight
case object Green extends TrafficLight
case object Yellow extends TrafficLight

def nextLight(light: TrafficLight): TrafficLight = light match {
  case Red => Green
  case Green => Yellow
  case Yellow => Red
}

@main def runTrafficLights(): Unit = {
  val lights: List[TrafficLight] = List(Red, Green, Yellow)

  lights.foreach { light =>
    println(s"Luz atual: $light, Próxima luz: ${nextLight(light)}")
  }
}
```

```
Luz atual: Red, Próxima luz: Green
Luz atual: Green, Próxima luz: Yellow
Luz atual: Yellow, Próxima luz: Red
```

O CASO PADRÃO O caso padrão (`case _`) é usado para capturar qualquer valor que não corresponda aos casos anteriores. Isso é útil para lidar com casos inesperados ou para fornecer um valor padrão. No exemplo do código 6.20, o caso padrão poderia ser usado para lidar com resultados que não são `Success` ou `Failure`, como `Pending`. O código 6.23 ilustra essa técnica.

Código 6.23: Usando o caso padrão para lidar com resultados inesperados.

```scala
def handleResult[T](result: Result[T]): String = result match {
  case Success(value) => s"Operação bem-sucedida: $value"
  case Failure(error) => s"Operação falhou: $error"
  case Pending(message) => s"Operação pendente: $message"
  case _ => "Resultado desconhecido"
}
@main def runResultExample(): Unit = {
  val successResult: Result[Int] = Success(42)
  val failureResult: Result[Int] = Failure("Erro desconhecido")
  val pendingResult: Result[Int] = Pending("Aguardando confirmação")

  println(handleResult(successResult))
  println(handleResult(failureResult))
  println(handleResult(pendingResult))
  println(handleResult(null))
}
```

```
Operação bem-sucedida: 42
Operação falhou: Erro desconhecido
Operação pendente: Aguardando confirmação
Resultado desconhecido
```

Ameaça à exaustividade. É preciso usar o caso padrão com cautela, pois ele pode introduzir uma ameaça à exaustividade. Se o caso padrão for usado, o compilador não exigirá que todos os casos possíveis sejam tratados explicitamente. Isso pode levar a erros silenciosos se um novo caso for adicionado ao tipo soma e não for tratado no código. No exemplo, o uso do caso padrão permitiu capturar um resultado nulo (`null`), o que claramente tirou a segurança de nulos do código.

Ou seja, acabamos por incorporar uma das desvantagens dos condicionais, que é a falta de exaustividade. Portanto, é recomendável usar o caso padrão apenas quando muito necessário e garantir que todos os casos relevantes sejam tratados explicitamente.

Ordem dos casos A ordem dos casos no casamento de padrões é importante, pois o compilador avalia os casos na ordem em que são definidos. O primeiro caso que corresponder ao valor será executado. Portanto, é importante colocar os casos mais específicos antes dos casos mais genéricos. Por exemplo, no código 6.20, o caso `Success` deve ser colocado antes do caso `Failure` para garantir que o resultado bem-sucedido seja tratado primeiro.

Tipos recursivos O casamento de padrões também pode ser usado com tipos recursivos, como listas ou árvores. Por exemplo, considere o código 6.24, que calcula a soma dos elementos de uma lista usando casamento de padrões.

Código 6.24: Cálculo da soma dos elementos de uma lista usando casamento de padrões.

```scala
sealed trait MyList[+A]
case object MyNil extends MyList[Nothing]
case class MyCons[+A](head: A, tail: MyList[A]) extends MyList[A]
```

```scala

def sum(list: MyList[Int]): Int = list match {
  case MyNil => 0
  case MyCons(head, tail) => head + sum(tail)
}

@main def runListSum(): Unit = {
  val numbers: MyList[Int] = MyCons(1, MyCons(2, MyCons(3, MyNil)))
  println(s"Soma: ${sum(numbers)}")
}
```

```
Soma: 6
```

Recursão estrutural A recursão estrutural é uma técnica que permite definir funções recursivas sobre tipos algébricos recursivos. O casamento de padrões é uma ferramenta muito útil para implementar a recursão estrutural, pois permite decompor os tipos recursivos em seus componentes e aplicar a função recursivamente de modo muito mais conciso e legível. Por exemplo, no código 6.24, a função `sum` usa casamento de padrões para decompor a lista em seu `head` e `tail`, permitindo calcular a soma de forma recursiva.

Padrões de guarda Os padrões de guarda são uma extensão do casamento de padrões que permite adicionar condições adicionais aos padrões. Eles são úteis quando queremos refinar ainda mais a correspondência de padrões com base em propriedades dos dados. Considere o código 6.25, que usa padrões de guarda para categorizar itens com base em seu preço.

Código 6.25: Uso de padrões de guarda em casamento de padrões.

```scala
case class Item(name: String, price: Double)

sealed trait ItemCategory
case class Cheap(item: Item) extends ItemCategory
case class ModeratelyPriced(item: Item) extends ItemCategory
case class Expensive(item: Item) extends ItemCategory
case class Unknown(item: Item) extends ItemCategory

def categorizeItem(item: Item): ItemCategory = {
  item match {
    case Item(n, p) if p < 10.0                => Cheap(item)
    case Item(n, p) if p >= 10.0 && p <= 100.0 => ModeratelyPriced(item)
    case Item(n, p) if p > 100.0               => Expensive(item)
    case _                                     => Unknown(item)
  }
}
```

Exaustividade e condições de guarda Usamos casamento de padrões para desestruturar o item e, em seguida, aplicamos padrões de guarda para categorizar o item com base em seu preço. Isso permite que o código seja mais legível e expres-

sivo, evitando condicionais aninhadas. Por outro lado, quando usarmos condições de guarda, o compilador não é capaz de verificar a exaustividade, pois as condições de guarda podem ser arbitrárias. Portanto, é importante garantir que o caso padrão (`case _`) seja usado para capturar qualquer caso não tratado, garantindo que o código seja robusto e não falhe em tempo de execução.

6.6 Outras formas de casamento de padrões

Além da expressão `match`, o casamento de padrões também pode ser usado em outros contextos. Vamos mencionar alguns a seguir, no entanto vamos explorar outras formas a medida que avançarmos em conceitos mais avançados.

- **Desestruturação em amarrações de variáveis**: O casamento de padrões pode ser usado para desestruturar objetos em variáveis individuais. Considere o código 6.26, que extrai os valores x e y de um ponto diretamente em variáveis.

Código 6.26: Desestruturação em amarrações de variáveis.

```scala
case class Person(name: String, age: Int)

val person = Person("Alice", 30)
val Person(name, age) = person
println(s"Name: $name, Age: $age")
```

```
Name: Alice, Age: 30
```

- **Definição de funções parciais**: funções parciais são definidas por casos específicos, que precisam ser tratados separadamente. Até o momento, vimos como definir funções parciais usando condicionais `if-else`. No entanto, podemos usar casamento de padrões para definir funções parciais de forma mais concisa. Por exemplo, considere o código 6.27.

Código 6.27: Cálculo da área de uma forma geométrica usando casamento de padrões.

```scala
def area(shape: Shape): Option[Double] = shape match {
  case Circle(radius)           => Some(Math.PI * radius * radius)
  case Rectangle(width, height) => Some(width * height)
  case Triangle(base, height)   => Some(0.5 * base * height)
  case _                        => None // Caso padrão para tipos não
  ↪ tratados
}

@main def runAreaCalculations(): Unit = {
  val shapes: List[Shape] = List(
    Circle(5),
    Rectangle(4, 6),
    Triangle(3, 7)
  )
```

```
14
15    shapes.foreach { shape =>
16      val result = area(shape)
17
18      println(s"Área: ${result.getOrElse("Desconhecida")}")
19    }
20  }
```

```
Área: Some(78.53981633974483)
Área: Some(24.0)
Área: Some(10.5)
```

- **Padrões de sequência**: O casamento de padrões também pode ser usado para trabalhar com sequências, como listas e arrays. Por exemplo, podemos usar casamento de padrões para extrair elementos de uma lista ou verificar se uma lista está vazia. O código 6.28 ilustra essa técnica.

Código 6.28: Casamento de padrões em sequências.

```
1  def processList(list: List[Int]): String = list match {
2    case Nil => "Lista vazia"
3    case head :: tail => s"Cabeça: $head, Resto: ${tail.mkString(", ")}"
4  }
5  @main def runListExample(): Unit = {
6    val list1 = List(1, 2, 3)
7    val list2 = List.empty[Int]
8
9    println(processList(list1)) // Cabeça: 1, Resto: 2, 3
10   println(processList(list2)) // Lista vazia
11 }
```

```
Cabeça: 1, Resto: 2, 3
Lista vazia
```

- **Outros padrões**: Vale ressaltar que o casamento de padrões pode ser usado com outros tipos de dados, como tuplas, opções, e até mesmo tipos personalizados. Ou seja, embora nossa discussão restrinja-se ao casamento de padrões com tipos produto e tipos soma, o casamento de padrões é uma técnica geral que pode ser aplicada a uma ampla variedade de estruturas de dados em Scala. Para mais detalhes, recomenda-se ao leitor a documentação oficial do Scala sobre casamento de padrões[1], que fornece uma visão abrangente das possibilidades e técnicas disponíveis.

6.7 Vantagens dos tipos algébricos

As principais vantagens dos tipos algébricos decorrem de sua capacidade de modelar estruturas e relações de dados com precisão e segurança de tipos. Eles melhoram

[1] https://docs.scala-lang.org/tour/pattern-matching.html

fundamentalmente a confiabilidade, legibilidade e manutenibilidade do código. A seguir, um resumo dos principais benefícios:

1. **Eliminação de erros em tempo de execução**: O uso de valores nulos (`null`) é uma grande fonte de erros em tempo de execução, como `NullPointerExceptions` (NPEs). Tipos algébricos (por exemplo, `Option` ou `Either`) permitem representar explicitamente a ausência de valor, forçando o tratamento de casos ausentes em tempo de compilação.

2. **Testagem exaustiva**: Em situações em que o problema exige a solução em casos, tipos algébricos permitem que o compilador verifique se todos os casos foram tratados. Isso é especialmente útil em funções que operam em dados de entrada com várias formas possíveis.

3. **Documentação e legibilidade**: Tipos algébricos servem como documentação viva, tornando explícito o formato e as restrições dos dados. Isso melhora a legibilidade do código e reduz a carga cognitiva ao entender o que cada tipo representa.

4. **Segurança de tipos aprimorada**: Tipos algébricos permitem que o compilador verifique a validade dos dados em tempo de compilação, reduzindo a probabilidade de erros de tipo em tempo de execução. Isso leva a um código mais robusto e confiável.

5. **Composição e reutilização de código**: Tipos algébricos incentivam a composição de tipos simples para criar estruturas mais complexas, promovendo a reutilização de código. Isso resulta em um design mais modular e flexível.

6. **Facilidade de refatoração**: Como os tipos algébricos são explícitos sobre as formas de dados, refatorações que envolvem mudanças na estrutura dos dados são mais seguras e fáceis de realizar. O compilador ajuda a identificar onde as mudanças afetam o código.

7. **Melhor suporte à programação funcional**: Tipos algébricos se encaixam naturalmente no paradigma de programação funcional, onde funções são tratadas como cidadãos de primeira classe. Eles permitem o uso de funções puras e de ordem superior, facilitando a composição de funções e a criação de abstrações reutilizáveis.

Em essência, os tipos algébricos elevam a modelagem de dados de simplesmente agrupar campos para definir precisamente o formato e as restrições dos dados, permitindo que o compilador atue como guardião contra erros comuns.

6.8 Tipos algébricos em outras linguagens

Diferentes linguagens de programação implementam tipos de dados algébricos de maneiras diferentes, mas geralmente seguem os princípios fundamentais de soma e produto. Exemplos:

- **Haskell**: usa `data` para tipos-produto e sintaxe direta para tipos-soma.

```haskell
data Point = Point Float Float
data Result a e = Success a | Failure e
```

Para casamento de padrões, Haskell usa a sintaxe `case`.

```haskell
handleResult :: Result a e -> String
handleResult (Success value) = "Operação bem-sucedida: " ++ show value
handleResult (Failure error) = "Operação falhou: " ++ error
```

- **Rust**: usa `enum` para tipos-soma e `struct` para tipos-produto.

```rust
struct Point {
    x: f64,
    y: f64,
}
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Para casamento de padrões, Rust usa a sintaxe `match`.

```rust
fn handle_result<T, E>(result: Result<T, E>) -> String {
    match result {
        Result::Ok(value) => format!("Operação bem-sucedida: {:?}",
        ↪  value),
        Result::Err(error) => format!("Operação falhou: {:?}", error),
    }
}
```

- **Kotlin**: usa `data class` para tipos-produto e `sealed class` para tipos-soma.

```kotlin
data class Point(val x: Double, val y: Double)
sealed class Result<out T> {
    data class Success<out T>(val value: T) : Result<T>()
    data class Failure(val error: String) : Result<Nothing>()
}
```

Para casamento de padrões, Kotlin usa a sintaxe `when`.

```kotlin
fun handleResult(result: Result<Any>): String {
    return when (result) {
        is Result.Success -> "Operação bem-sucedida: ${result.value}"
        is Result.Failure -> "Operação falhou: ${result.error}"
    }
```

```
}
```

- **F#**: usa `type` para tipos-produto e `union type` para tipos-soma.

```
type Point = { X: float; Y: float }
type Result<'T, 'E> = Success of 'T | Failure of 'E
```

Para casamento de padrões, F# usa a sintaxe `match`.

```
let handleResult result =
    match result with
    | Success value -> sprintf "Operação bem-sucedida: %A" value
    | Failure error -> sprintf "Operação falhou: %s" error
```

- **Java**: usa `record` para tipos-produto e `sealed interface` para tipos-soma (a partir do Java 17).

```
public record Point(double x, double y) {}
public sealed interface Result<T, E> permits Success, Failure { }
public record Success<T>(T value) implements Result<T, E> {}
public record Failure<E>(E error) implements Result<T, E> {}
```

Além disso, casamento de padrões é suportado através de `switch expressions` e `instanceof` com padrões (a partir do Java 16). Por exemplo:

```
public String handleResult(Result<Integer, String> result) {
    return switch (result) {
        case Success<Integer> s -> "Operação bem-sucedida: " + s.value;
        case Failure<String> f -> "Operação falhou: " + f.error;
        default -> "Resultado desconhecido";
    };
}
```

- Entre outras linguagens funcionais, como OCaml e Elm, também implementam tipos algébricos de maneira semelhante.

6.9 Resumo

Os tipos algébricos são uma poderosa abstração que permite modelar dados de forma precisa e segura. Eles são compostos por tipos produto e tipos soma, que permitem representar estruturas complexas e relações entre dados. O casamento de padrões é a principal ferramenta para trabalhar com tipos algébricos, permitindo extrair valores e tratar casos de forma concisa e expressiva.

Além disso, os tipos algébricos oferecem vantagens significativas, como a eliminação de erros comuns, a garantia de exaustividade no tratamento de casos e

a melhoria da legibilidade do código. Eles são amplamente utilizados em linguagens funcionais e têm ganhado popularidade em linguagens imperativas modernas, como Scala, Kotlin e Rust.

Aviso de licença de uso