Tipos de dados

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

Qual é a segunda etapa no processo de projeto de funções? Definição de tipos de dados.

Qual o propósito dessa etapa? Identificar as informações e definir como elas serão representadas.

Essa etapa pode ter parecido, até então, muito simples ou talvez até desnecessária, isto porque as informações que precisávamos representar eram “simples”.

No entanto, essa etapa é muito importante no projeto de programas, de fato, vamos ver que para muitos casos, os tipos de dados vão guiar o restante das etapas do projeto.

Vamos começar com a definição do que é um tipo de dado.

Um **tipo de dado** é um conjunto de valores que uma variável pode assumir.

Exemplos

- Booleano $= \{verdadeiro, falso\}$
- Natural = $\{0, 1, 2, \dots\}$
- Inteiro = $\{\dots, -2, -1, 0, 1, 2, \dots\}$
- String = { ’ ‘, ’a’, ‘b’, . . . }
- String que começa com a = { ‘a’, ‘aa’, ‘ab’, . . . }

Adequação de tipo de dado

Durante a etapa de definição de tipos de dados identificamos as informações e definimos como elas são representadas no programa.

Como determinar se um tipo de dado **é adequado** para representar uma informação?

Adequação de tipo de dado

Um inteiro é adequado para representar a quantidade de pessoas em um planeta?

- Não é adequado pois ele pode ser negativo, mas a quantidade de pessoas em um planeta não pode, ou seja, o tipo *permite representar valores inválidos*.

E um natural de 32 bits?

- Não é adequado pois o valor máximo possível é 4.294.967.295, mas o planeta terra tem mais pessoas que isso, ou seja, o tipo *não permite representar todos os valores válidos*.

E um natural?

- É adequado. Cada valor do conjunto dos naturais representa um valor válido de informação, e cada possível valor de informação pode ser representado por um número natural.

Diretrizes para o projeto de tipos de dados

Diretrizes para o projeto de tipos de dados:

- Faça os valores válidos representáveis.
- Faça os valores inválidos irrepresentáveis.

Vamos aplicar esses princípios a uma série de exemplos.

No exemplo da escolha do combustível, nós definimos os seguintes tipos:

```
/// O preço do litro do combustível, deve ser um número positivo.
type Preco = Float

/// O tipo do combustível, deve "Álcool" ou "Gasolina".
type Combustivel = String
```

Esses tipos estão de acordo com as diretrizes para o projeto de tipos de dados?

Não!

Vamos resolver essa questão começando com `Combustivel`.

Enumerações

Em um **tipo enumerado** todos os valores do tipo são enumerados explicitamente.

A forma geral para definir tipos enumerados é:

```
[pub] type NomeDoTipo {
  Valor1
  Valor2
  ...
}
```

Cada tipo enumerado pode ter 0 ou mais valores. O nome e os valores do tipo devem começar com letra maiúscula.

Quando usar tipos enumerados?

Quando todos os valores válidos para o tipo podem ser nomeados.

Vamos definir um tipo enumerado para representar o tipo combustível.

```
pub type Combustivel {
  Alcool
  Gasolina
}
```

Podemos utilizar os operadores `==` e `!=` e a função `string.inspect` com os valores de tipos enumerados.

```
> Alcool == Gasolina
False
> Alcool != Gasolina
True
> string.inspect(Gasolina)
"Gasolina"
```

Assim como para valores do tipo `Bool`, podemos utilizar a expressão `case` para processar valores de tipos enumerados. De fato, o tipo `Bool` é um tipo enumerado com os valores `True` e `False` pré-definido na linguagem.

```
pub fn msg_combustivel(
  c: Combustivel
) -> String {
  case c {
    Alcool -> "Use álcool."
    Gasolina -> "Use gasolina."
  }
}
```

A análise dos casos precisa ser exaustiva

```gleam
pub fn msg_combustivel(c: Combustivel) -> String {
  case c {
    Alcool -> "Use álcool."
  }
}
```

```
  ┌─ src/arquivo.gleam:2:3
  │
2 │ +   case c {
3 │ │     Alcool -> "a"
4 │ │   }
  │ +───^
```

This case expression does not have a pattern for all possible values. If it is run on one of the values without a pattern then it will crash.
The missing patterns are:
    Gasolina

Exemplo - tíquete do RU

O RU da UEM cobra um valor por tíquete que depende da relação do usuário com a universidade. Para alunos e servidores que recebem até 3 salários mínimos o tíquete custa R$ 5,00, para servidores que recebem acima de 3 salários mínimos e docentes, R$ 10,00, para pessoas da comunidade externa, R$ 19,00. Como parte de um sistema de cobrança você deve projetar uma função que determine quanto deve ser cobrado de um usuário por um quantidade de tíquetes.

Exemplo - tíquete do RU

Análise

- Determinar quanto deve ser cobrado de um usuário por uma quantidade de tíquetes
- O usuário pode ser aluno ou servidor (até 3 sal) - R$ 5, servidor (acima de 3 sal) ou docente - R$ 10, ou externo - R$ 19.

Definição de tipos de dados

- As informações são a quantidade, o tipo de usuário e o valor que deve ser cobrado.

Exemplo - tíquete do RU

Como representar um tipo de usuário? Criando um tipo enumerado com os valores possíveis para o tipo.

```
/// O tipo de usuário do RU da UEM.
pub type Usuario {
  Aluno
  // Servidor que recebe até 3 salários mínimos.
  ServidorAte3
  // Servidor que recebe mais do que 3 salários mínimos.
  ServidorMais3
  Docente
  Externo
}
```

Especificação

```
/// Determina o custo de *quant* tíquetes para um usuário do tipo *usuario*.
/// O custo de um tíquete é
/// - Aluno           5,0
/// - ServidorAte3    5,0
/// - ServidorMais3  10,0
/// - Docente        10,0
/// - Externo        19,0
pub fn custo_tiquetes(usuario: Usuario, quant: Int) -> Float
```

Não vamos tratar quantidades menores ou iguais a zero.

Quantos exemplos são necessários para funções que processam valores de tipos enumerados?

Pelo menos um para cada valor da enumeração.

```
pub fn custo_tiquetes_examples() {
  check.eq(custo_tiquetes(Aluno, 3), 15.0)
  check.eq(custo_tiquetes(ServidorAte3, 2), 10.0)
  check.eq(custo_tiquetes(ServidorMais3, 2), 20.0)
  check.eq(custo_tiquetes(Docente, 3), 30.0)
  check.eq(custo_tiquetes(Externo, 4), 76.0)
}
```

Como iniciamos a implementação de uma função que processa um valor de tipo enumerado?

Criando um caso para cada valor da enumeração.

Implementação

```
pub fn custo_tiquetes(usuario: Usuario, quant: Int) -> Float {
  case usuario {
    Aluno -> todo
    ServidorAte3 -> todo
    ServidorMais3 -> todo
    Docente -> todo
    Externo -> todo
  }
}
```

Agora completamos o corpo considerando cada forma de resposta dos exemplos.

Implementação

```
pub fn custo_tiquetes(usuario: Usuario, quant: Int) -> Float {
  case usuario {
    Aluno -> 5.0 *. int.to_float(quant)
    ServidorAte3 -> 5.0 *. int.to_float(quant)
    ServidorMais3 -> 10.0 *. int.to_float(quant)
    Docente -> 10.0 *. int.to_float(quant)
    Externo -> 19.0 *. int.to_float(quant)
  }
}
```

A implementação está correta? Sim.

Estruturas

Os tipos de dados que vimos até agora são atômicos, isto é, não podem ser decompostos.

Agora veremos como representar dados onde dois ou mais valores devem ficar juntos:

- Registro de um aluno;
- Placar de um jogo de futebol;
- Informações de um produto.

Chamamos estes tipos de dados de **dados compostos**, **registros** ou **estruturas**.

A forma geral para definir um **dado composto** é:

```
[pub] type NomeDoTipo {
  NomeDoTipo([campo1:] Tipo1, [campo2:] Tipo2, ...)
}
```

Quando usar dados compostos?

Quando a informação consiste de dois ou mais itens que juntos descrevem uma entidade.

Vamos definir uma estrutura para representar um ponto em um plano cartesiano.

Estruturas - operações

Definição

```
type Ponto {
  Ponto(x: Int, y: Int)
}
```

Construção

```
> let p1: Ponto = Ponto(x: 3, y: 4)
> let p2 = Ponto(8, 2)
> p2
Ponto(x: 8, y: 2)
```

Acesso aos campos

```
> p1.x + p1.y
7
```

Desestruturação

```
> // pela posição
> let Ponto(x, y) = p2
> x
8
> y
2

// pelo rótulo
> let Ponto(y: a, ..) = p2
> a
2
> let Ponto(y:, ..) = p2
> y
2
```

Estruturas - operações

Definição

```
type Ponto {
  Ponto(x: Int, y: Int)
}
```

Construção

```
> let p1: Ponto = Ponto(x: 3, y: 4)
> let p2 = Ponto(8, 2)
> p2
Ponto(x: 8, y: 2)
```

Acesso aos campos

```
> p1.x + p1.y
7
```

Comparação

```
> p1 == Ponto(3, 4)
True
> p1 != p1
False
> p1 != p2
True
```

Inspeção

```
> string.inspect(p1)
"Ponto(x: 3, x: 4)"
```

Junto com a definição de uma estrutura, também faremos a descrição do seu propósito e do seus campos.

```
/// Um ponto no plano cartesiano.
type Ponto {
  // x e y são as coordenadas dos pontos.
  Ponto(x: Int, y: Int)
}
```

Podemos consultar o valor de um campo, mas como alterar o valor de um campo? Não tem como! Lembrem-se, estamos estudando o paradigma funcional, onde não existe mudança de estado!

Ao invés de modificar o campo de uma instância da estrutura, criamos uma cópia da instância com o campo alterado.

Vamos criar um ponto `p2` que é como `p1`, mas com o valor `5` para o campo `y`.

```
> let p1 = Ponto(3, 4)
> let p2 = Ponto(p1.x, 5)
> p2
Ponto(x: 3, y: 5)
```

Quais são as limitações desse método?

- Se a estrutura tem muitos campos e desejamos alterar apenas um campo, temos que especificar a cópia de todos os outros;
- Se a estrutura é alterada pela adição ou remoção de campos, então, todas as operações de "cópia" da estrutura no código devem ser alteradas.

Alterando dados estruturados

Gleam tem uma sintaxe especial para atualização de estruturas.

```
> let p1 = Ponto(3, 4)
> let p2 = Ponto(..p1, y: 5)
> p2
Ponto(x: 3, y: 5)

> let p3 = Ponto(..p1, x: 7)
> p3
Ponto(x: 7, y: 4)

> // Podemos atualiza mais que um campo (não faz sentido nesse exemplo)
> Ponto(..p1, x: 1, y: 2)
Ponto(x: 1, y: 2)
```

Exemplo - outras linguagens

A ideia de estruturas imutáveis que são “atualizadas” através de cópias está presente em diversas linguagens. A seguir temos um exemplo em Python e outro em Rust.

```python
from dataclasses import dataclass, replace

@dataclass(frozen=True)
class Ponto:
    x: int
    y: int

>>> p = Ponto(10, 20)

>>> # Atribuição inválida, p é imutável
>>> p.x = 8

>>> # Cria uma cópia de p alterando x para 8
>>> p1 = replace(p, x=8)
>>> p1
Ponto(x=8, y=20)
```

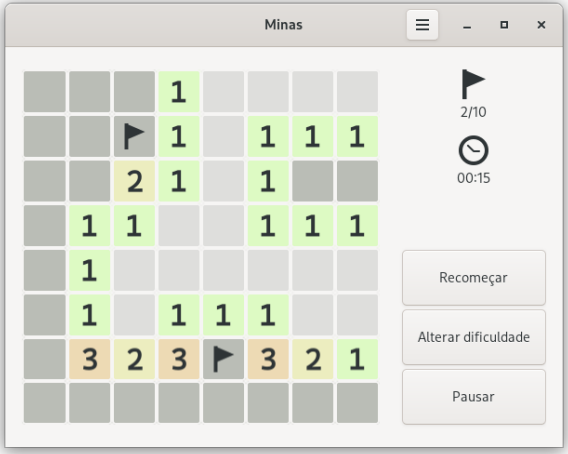
```rust
struct Ponto {
    x: i32,
    y: i32,
}

pub fn main() {
    let p = Ponto { x: 10, y: 20 };

    // Atribuição inválida, p é imutável
    p.x = 8;

    // Cria uma cópia de p alterando x para 8
    let p1 = Ponto {x: 8, ..p};
    assert_eq!(p1.x, 8);
    assert_eq!(p1.y, 20);
}
```

Exemplo - Campo minado

Campo minado é um famoso jogo de computador. O jogo consiste de um campo retangular de quadrados que podem ou não conter minas escondidas. Os quadrados podem ser abertos clicando sobre eles. O objetivo do jogo é abrir todos os quadrados que não têm minas. Se o jogador abrir um quadrado com uma mina, o jogo termina e o jogador perde.

Como guia para explorar o campo, cada quadrado aberto exibe o número de minas nos quadrados ao seu redor (no máximo 8). Quando um quadrado sem minas ao redor é aberto, todos os quadrados ao seu redor também são abertos. O usuário pode colocar uma bandeira sobre um quadrado fechado para sinalizar uma possível mina e impedir que ele seja aberto. Uma bandeira também pode ser removida de um quadrado.

Exemplo - Campo minado



Exemplo - Campo minado

Projete um tipo de dado para representar um quadrado em um jogo de campo minado. Não é necessário armazenar o número de bombas ao redor do quadrado pois esse valor pode ser calculado dinamicamente.

Exemplo - Campo minado

Em uma primeira tentativa poderíamos pensar: o quadrado pode ter uma mina ou não, pode estar fechado ou aberto e pode ter uma bandeira ou não. Como são três itens relacionados, então definiríamos uma estrutura. Além disso, cada item tem dois estados possíveis, então poderíamos usar booleano para representar cada estado.

```
/// Um quadrado no jogo campo minado.
pub type Quadrado {
  Quadrado(mina: Bool, aberto: Bool, bandeira: Bool)
}
```

Nós vimos duas diretrizes para o projeto de tipo de dado

- Faça os valores válidos representáveis.
- Faça os valores inválidos irrepresentáveis.

A definição de `Quadrado` está de acordo com essas diretrizes? Vamos verificar!

Quantas possíveis instâncias distintas existem de `Quadrado`? São três campos, cada um pode assumir dois valores, portanto, $2 \times 2 \times 2 = 8$.

Vamos listar essas instâncias e analisar se todas são válidas.

`mina?`	`aberto?`	`bandeira?`	Válido?
F	F	F	Sim
F	F	V	Sim
F	V	F	Sim
F	V	V	Não
V	F	F	Sim
V	F	V	Sim
V	V	F	Sim
V	V	V	Não

Temos dois estados inválidos!

Como evitar estes estados inválidos? Primeiro temos que entender o problema.

A questão é que apenas 3 das 4 possíveis combinações dos valores dos campos `aberto?` e `bandeira?` são válidos: aberto, fechado ou fechado com bandeira.

Para resolver a situação podemos "juntar" os campo `aberto?` e `bandeira?` em um campo `estado` que pode assumir um desses três valores.

```gleam
/// O estado de um quadrado no
/// campo do jogo.
pub type Estado {
  Aberto
  Fechado
  FechadoComBandeira
}

/// Um quadrado no campo de jogo.
pub type Quadrado {
  // True se tem mina,
  // False caso contrário.
  Quadrado(mina: Bool, estado: Estado)
}
```

Quantas possíveis instâncias distintas existem de `Quadrado`? O campo `mina` pode assumir dois valores e o campo `estado` 3, portanto, $2 \times 3 = 6$, que são os seis estados válidos que identificamos anteriormente.

Agora que temos uma representação adequada para um quadrado, podemos avançar e projetar uma função que determina como um quadrado ficará após a ação de um usuário. O usuário pode fazer uma ação para abrir um quadrado, adicionar uma bandeira ou remover uma bandeira.

Exemplo - Ação campo minado

Análise

- Determinar o novo estado de um quadrado a partir da ação do usuário

Definição de tipos de dados

```
/// Uma ação do usuário no jogo.
pub type Acao {
  Abrir
  AdicionarBandeira
  RemoverBandeira
}
```

Exemplo - Ação campo minado

Especificação

```
/// Atualiza o estado do quadrado *q* dado a *acao* do usuário... completar.
pub fn atualiza_quadrado(q: Quadrado, acao: Acao) -> Quadrado
```

Qual é o campo do quadrado de entrada que pode mudar? Apenas o estado.

Do que depende a mudança do estado? Do estado atual e da ação.

Se o comportamento de uma função depende apenas de um valor enumerado, quantos exemplos precisamos colocar na especificação? Um para cada valor da enumeração.

A função que estamos projetando depende de apenas um valor enumerado? Não. Depende de dois, o valor do estado e o valor da ação.

Quantos exemplos precisamos nesse caso? Pelo menos $3 \times 3 = 9$ exemplos. Vamos fazer uma tabela para não esquecer de nenhum caso!

Exemplo - Ação campo minado

estado/ação	abrir	adicionar	remover
aberto	-	-	-
fechado	aberto	fechado com bandeira	-
fechado com bandeira	-	-	fechado

```
check.eq(
  atualiza_quadrado(Quadrado(False, Aberto), Abrir),
  Quadrado(False, Aberto)
) // q

check.eq(
  atualiza_quadrado(Quadrado(False, Fechado), Abrir),
  Quadrado(False, Aberto)
) // Quadrado(..q, estado: Aberto)

// restante dos exemplos
```

Implementação

Se o comportamento de uma função depende apenas de um valor enumerado, qual é a estrutura inicial do corpo da função? Um caso para cada valor enumerado.

A função que estamos projetando depende de dois valores enumerados, qual deve ser a estrutura inicial do corpo da função? Uma seleção de dois níveis, cada nível para um valor enumerado; ou; uma seleção com uma condição para cada par dos valores enumerados.

Exemplo - Ação campo minado

```gleam
pub fn atualiza_quadrado(q: Quadrado, acao: Acao) {
  case q.estado {
    Aberto -> case acao {
      Abrir -> todo
      AdicionarBandeira -> todo
      RemoverBandeira -> todo
    }
    Fechado -> case acao {
      Abrir -> todo
      AdicionarBandeira -> todo
      RemoverBandeira -> todo
    }
    FechadoComBandeira -> case acao {
      Abrir -> todo
      AdicionarBandeira -> todo
      RemoverBandeira -> todo
    }
  }
}
```

```gleam
pub fn atualiza_quadrado(q: Quadrado, acao: Acao) {
  case q.estado, acao {
    Aberto, Abrir -> todo
    Aberto, AdicionarBandeira -> todo
    Aberto, RemoverBandeira -> todo
    Fechado, Abrir -> todo
    Fechado, AdicionarBandeira -> todo
    Fechado, RemoverBandeira -> todo
    FechadoComBandeira, Abrir -> todo
    FechadoComBandeira, AdicionarBandeira -> todo
    FechadoComBandeira, RemoverBandeira -> todo
  }
}
```

estado/ação	abrir	adicionar	remover
aberto	-	-	-
fechado	aberto	fechado-com-bandeira	-
fechado-com-bandeira	-	-	fechado

Se olharmos a tabela de exemplos, vamos notar que em apenas 3 casos precisamos atualizar o quadrado, então, não é necessário colocar explicitamente no código os 9 casos, podemos simplificar o código antes mesmo de escrevê-lo!

Exemplo - Ação campo minado

```
/// Atualiza o estado do quadrado *q* dado a *acao* do usuário. A atualização é
/// feita conforme a tabela a seguir, onde - significa que o quadrado permanece
/// como estava.
///
/// | estado/ação          |  abrir  |      adicionar       | remover |
/// |---------------------:|:-------:|:--------------------:|:-------:|
/// | aberto               |   -     |          -           |    -    |
/// | fechado              | aberto  | fechado-com-bandeira |    -    |
/// | fechado-com-bandeira |   -     |          -           | fechado |
pub fn atualiza_quadrado(q: Quadrado, acao: Acao) -> Quadrado {
  case q.estado, acao {
    Fechado, Abrir -> Quadrado(..q, estado: Aberto)
    Fechado, AdicionarBandeira -> Quadrado(..q, estado: FechadoComBandeira)
    FechadoComBandeira, RemoverBandeira -> Quadrado(..q, estado: Fechado)
    _, _ -> q
  }
}
```

Uniões

Exemplo - Estado tarefa

Projete uma função que exiba uma mensagem sobre o estado de uma tarefa. Uma tarefa pode estar em execução, ter sido concluída em uma duração específica e com um mensagem de sucesso, ou ter falhado com um código e uma mensagem de erro.

Como representar o estado de uma tarefa?

Vamos tentar uma estrutura.

Exemplo - Estado tarefa

```
/// O estado de uma tarefa.
type EstadoTarefa {
  EstadoTarefa(
    // True se a tarefa está em execução, False caso contrário.
    executando: Bool,
    // Em caso de sucesso
    duracao: Int,
    msg_sucesso: String,
    // Em caso de erro
    codigo_erro: Int,
    msg_erro: String
  )
}
```

Qual é o problema dessa representação? Possíveis estados inválidos. O que significa `EstadoTarefa(True, 10, "Ótimo desempenho", 123, "Falha na conexão")`?

Exemplo - Estado tarefa

Analisando a descrição do problema conseguimos separar o estado da tarefa em três casos:

- Em execução
- Sucesso, com uma duração e uma mensagem
- Falha, com um código e uma mensagem

Esses casos são excludentes, ou seja, se a tarefa se enquadra em um deles, não devemos armazenar informações sobre os outros (caso contrário, seria possível criar um estado inconsistente).

E como expressar esse tipo de dado? Usando união de tipos.

Definimos anteriormente um tipo de dado como um conjunto de possíveis valores, agora vamos discutir qual é a relação entre definição de tipos de dados e operações com conjuntos.

- Os valores possíveis para um tipo definido por uma estrutura (**tipo produto**) é o produto cartesiano dos valores possíveis de cada um do seus campos;
- Os valores possíveis para um tipo definido por uma união (**tipo soma**) é a união dos valores de cada tipo (classe de valores) da união.
- Chamamos de **tipo algébrico de dado** um tipo soma de tipos produtos.

Entender essa relação pode nos ajudar na definição dos tipos de dados, como foi para o quadrado do campo minado e como é para o caso do estado da tarefa.

Uniões

Algumas linguagens, como Rust e Python, tem maneiras diferentes para definir tipos de dados.

A maioria das linguagens funcionais, incluindo o Gleam, tem apenas uma.

A forma geral para definição de tipos de dados em Gleam é

```
[pub | pub opaque] type NomeDoTipo {
  Caso1[([campo1:] Tipo1, [campo1:] Tipo2, ...)]
  Caso2[([campo1:] Tipo1, [campo1:] Tipo2, ...)]
  ...
}
```

```
[pub] type NomeDoTipo {
  Valor1
  ...
}
```

```
[pub] type NomeDoTipo {
  NomeDoTipo([campo1:] Tipo1, [campo2:] Tipo2, ...)
}
```

Exemplo - Estado tarefa

```
/// O estado de uma tarefa
type EstadoTarefa {
  // A tarefa está em execução
  Executando
  // A tarefa finalizou com sucesso
  Sucesso(duracao: Int, msg: String)
  // A tarefa finalizou com falha
  Falha(codigo: Int, msg: String)
}
```

```
> let tarefa: EstadoTarefa = Executado

> tarefa.msg

1 |     tarefa.msg
  |           ^^^^ This field does not exist


> let tarefa = Sucesso(10, "Recuperação exitosa.")

> tarefa.msg

1 |     tarefa.msg
  |           ^^^^ This field does not exist
```

Como podemos acessar os campos então!? Usando casamento de padrão com o **`case`**.

Exemplo - Estado tarefa

```gleam
/// O estado de uma tarefa
type EstadoTarefa {
  // A tarefa está em execução
  Executando
  // A tarefa finalizou com sucesso
  Sucesso(duracao: Int, msg: String)
  // A tarefa finalizou com falha
  Falha(codigo: Int, msg: String)
}
```

```gleam
> // Devolve -1 se não tem duracao.
> pub fn duracao(tarefa: EstadoTarefa) -> Int {
  case tarefa {
    Sucesso(duracao, _) -> duracao
    _ -> -1
  }
}


> duracao(Executando)
-1
> duracao(Sucesso(10, "Recuperação exitosa."))
10
> duracao(Falha(-23, "Arquivo não existente."))
-1
```

Exemplo - Estado tarefa

Agora podemos retornar e concluir o projeto.

Projete uma função que exiba uma mensagem sobre o estado de uma tarefa. Uma tarefa pode estar em execução, ter sido concluída em uma duração específica e com um mensagem de sucesso, ou ter falhado com um código e uma mensagem de erro.

Especificação

```
/// Produz uma string amigável para o usuário para descrever o estado da tarefa.
pub fn msg(tarefa: EstadoTarefa) -> String
```

O exercício não é muito específico sobre a saída (o foco é no projeto de dados), por isso usamos a criatividade para definir a saída nos exemplos a seguir.

Exemplo - Estado tarefa

Quantos exemplos são necessários? Pelo menos um para cada classe de valor.

```
pub fn msg_examples() {
  check.eq(
    mensagem(Executando),
    "A tarefa está em execução."
  )
  check.eq(
    mensagem(Sucesso(12, "Os resultados estão corretos.")),
    "Tarefa concluída (12s): Os resultados estão corretos."
  )
  check.eq(
    mensagem(Erro(123, "Número inválido '12a'.")),
    "A tarefa falhou (erro 123): Número inválido '12a'."
  )
}
```

Mesmo sem saber detalhes da implementação, podemos definir a estrutura do corpo da função baseado apenas no tipo do dado, no caso, `EstadoTarefa`. São três casos:

```
pub fn mensagem(estado: EstadoTarefa) -> String {
  case estado {
    Executando -> todo

    Sucesso(duracao, msg) -> todo

    Erro(codigo, msg) -> todo

  }
}
```

Mesmo sem saber detalhes da implementação, podemos definir a estrutura do corpo da função baseado apenas no tipo do dado, no caso, `EstadoTarefa`. São três casos:

```
pub fn mensagem(estado: EstadoTarefa) -> String {
  case estado {
    Executando ->
      "A tarefa está em execução."
    Sucesso(duracao, msg) ->
      "Tarefa concluída (" <> int.to_string(duracao) <> "s): " <> msg
    Erro(codigo, msg) ->
      "A tarefa falhou (erro " <> int.to_string(codigo) <> "): " <> msg
  }
}
```

Podemos usar tipos algébricos em outras linguagens? Sim, de fato, com o aumento do uso do paradigma funcional, muitas linguagens, mesmo algumas mais antigas como Java e Python, ganharam suporte a essa forma de definição de tipo de dados.

Vamos ver alguns exemplos.

```python
@dataclass
class Executando:
    pass

@dataclass
class Sucesso:
    duracao: int
    msg: str

@dataclass
class Erro:
    codigo: int
    msg: str

EstadoTarefa = Executando | Sucesso | Erro
```

```python
def mensagem(estado: EstadoTarefa) -> str:
    if isinstance(estado, Executando):
        return 'A tarefa está em execução'
    elif isinstance(estado, Sucesso):
        return 'A tafera finalizou com sucesso ({}s): {}'.format(estado.duracao,
                                                                 estado.msg)
    else:
        return 'A tafera falhou (error {}): {}'.format(estado.codigo, estado.msg)
```

```python
def mensagem(estado: EstadoTarefa) -> str:
    match estado:
        case Executando():
            return 'A tarefa está em execução'
        case Sucesso(duracao, msg):
            return f'A tafera finalizou com sucesso ({duracao}s): {msg}'
        case Erro(codigo, msg):
            return f'A tafera falhou (error {codigo}): {msg}'
```

Aqui usamos **casamento de padrões** para decompor cada tipo produto em seus componentes.

```rust
pub enum EstadoTarefa {
    Executando,
    Sucesso(u32, String),
    Erro(u32, String),
}
pub fn mensagem(estado: &EstadoTarefa) -> String {
    match estado {
        EstadoTarefa::Executando =>
            "A tarefa está em execução".to_string(),
        EstadoTarefa::Sucesso(duracao, msg) =>
            format!("A tarefa finalizou com sucesso ({duracao}s): {msg}"),
        EstadoTarefa::Erro(codigo, msg) =>
            format!("A tarefa falhou (erro {codigo}): {msg}"),
    }
}
```

Usamos novamente casamento de padrões para decompor `Sucesso` e `Erro` em seu componentes.

```java
sealed interface EstadoTarefa permits Executando, Sucesso, Erro {};
record Executando() implements EstadoTarefa {};
record Sucesso(int duracao, String msg) implements EstadoTarefa {};
record Erro(int erro, String msg) implements EstadoTarefa {};

static String mensagem(EstadoTarefa estado) {
    return switch (estado) {
        case Executando e ->
            "A tarefa está executando";
        case Sucesso s ->
            String.format("A tarefa foi concluída (%ds): %s", s.duracao(), s.msg());
        case Erro e ->
            String.format("A tarefa falhou (erro %d): %s", e.erro(), e.msg());
    };
}
```

A JEP 405, que ainda está em *preview*, permite o uso de padrões para decompor registros.

Valores opcionais e erros

Pendências

Nós aplicamos com sucesso as diretrizes para projeto de tipos de dados no exemplo do combustível, quadrado do campo minado e estado da tarefa. Mas ainda temos alguns pontos para resolver.

No problema do combustível usamos `Float` para representar o preço do combustível, mas não garantimos que o preço é maior do que zero.

No problema do estado da tarefa, usamos `Int` para representar a duração da tarefa no caso de sucesso, mas não garantimos que a duração é maior ou igual a zero.

Na função exemplo `duracao(EstadoTarefa) -> Int`, devolvemos `-1` para representar que o estado da tarefa não tem informação de duração.

Como podemos resolver essas questões? Vamos começar com a função `duracao`.

```gleam
/// Devolve -1 se não tem duracao.
pub fn duracao(tarefa: EstadoTarefa) -> Int {
  case tarefa {
    Sucesso(duracao, _) -> duracao
    _ -> -1
  }
}

> duracao(Executando)
-1
> duracao(Sucesso(10, "Recuperação exitosa."))
10
> duracao(Falha(-23, "Arquivo não existente."))
-1
```

Como representar um inteiro que pode ou não estar presente?

São dois casos distintos, ou existe um valor, ou não existe nenhum. Então podemos criar um tipo união.

```gleam
type Opcional {
    Nenhum
    Algum(Int)
}
```

```
pub fn duracao(tarefa: EstadoTarefa) -> Opcional {
  case tarefa {
    Sucesso(duracao, _) -> Algum(duracao)
    _ -> Nenhum
  }
}
```

```
> duracao(Executando)
Nenhum
> duracao(Sucesso(10, "Recuperação exitosa."))
Algum(10)
> duracao(Falha(-23, "Arquivo não existente."))
Nenhum
```

Quais as vantagens dessa abordagem?

O código é mais claro.

O usuário da função tem que tratar de forma explícita os dois casos, ele não pode usar por “acidente” o valor -1 como se existisse uma duração.

```
> 2 * duracao(Executado)

The * operator expects arguments of
this type:

    Int

But this argument has this type:

    Opcional
```

Soma um

Projete uma função que receba um opcional e some 1 ao valor se ele estiver presente.

```
/// Soma 1 ao valor opcional de *a*.
pub fn soma1(a: Opcional) -> Opcional {
  todo
}

pub fn soma1_examples() {
  check.eq(soma1(Nenhum), Nenhum)
  check.eq(soma1(Algum(10)), Algum(11))
}
```

```
/// Soma 1 ao valor opcional de *a*.
pub fn soma1(a: Opcional) -> Opcional {
  case a {
    Nenhum -> Nenhum
    Algum(x) -> Algum(x + 1)
  }
}
```

Primeiro string

Projete uma função que devolva o primeiro caractere de uma string.

```
type Opcional {
  Nenhum
  Algum(String)
}

/// Devolve o primeiro caractere
/// de *s* ou Nenhum se *s* é vazia.
pub fn primeiro(s: String) -> Opcional {
  todo
}

pub fn primeiro_examples() {
  check.eq(primeiro(""), Nenhum)
  check.eq(primeiro("casa"), Algum("c"))
}
```

```
/// Devolve o primeiro caractere
/// de *s* ou Nenhum se *s* é vazia.
pub fn primeiro(s: String) -> Opcional {
  case s {
    "" -> Nenhum
    _ -> Algum(string.slice(s, 0, 1))
  }
}
```

Existe algum problema com a implementação?

A string em `Opcional` ainda pode ser vazia.

Este é o mesmo problema do preço e da idade...

Gleam tem na biblioteca padrão o tipo `Option` para representar valores opcionais.

O tipo `Option` é definido como

```
type Option(a) {
  None
  Some(a)
}
```

O nome `a` é um parâmetro de tipo.

Os parâmetros de tipos são escritos com letra minúscula.

Um parâmetro de tipo pode ser instanciado com qualquer tipo.

```
import gleam/option.{type Option, Some, None}

pub fn soma1(a: Option(Int)) -> Option(Int) {
  case a {
    None -> None
    Some(x) -> Some(x + 1)
  }
}

pub fn primeiro(s: String) -> Option(String) {
  case s {
    "" -> None
    _ -> Some(string.slice(s, 0, 1))
  }
}
```

As linguagens Rust e Java, entre outras, também têm o tipo `Option`.

Em Rust o tipo `Option` é bastante utilizando na biblioteca padrão para representar valores que podem estar ausentes, como na saída de funções semelhantes a função `primeiro`.

Em Gleam é mais comum utilizar o tipo `Result`, que vamos discutir a seguir.

Como lidar com funções que podem falhar?

Por exemplo, uma função que converte uma string para um número pode falhar, pois nem todas as strings representam números válidos, como lidar com isso?

Estratégias comumente utilizadas incluem

- Finalizar o programa
- Lançar exceção (Python, Java)
- ...
- Devolver um valor indicando erro

Nos vimos que as linguagens puramente funcionais não têm efeitos colaterais, então a opção mais viável é a última.

Uma possibilidade é utilizar `Option` como resultado sendo que o `None` representa que a função falhou e `Some(val)` que a função executou corretamente e produziu `val` como resposta.

Em que situações o tipo `Option` não seria adequado? Quando existe mais de uma possível causa para a falha da função e queremos distinguir entre as falhas.

Por exemplo, uma função para escrever em um arquivo pode falhar porque o arquivo não existe, o usuário não tem permissão para escrever no arquivo, o disco está cheio, etc.

Como podemos fazer nesse caso?

Definimos uma enumeração com dois casos, uma para erro com um valor associado, e um para sucesso com o valor associado.

Em Gleam, este é o tipo `Result`, pré-definido como:

```
type Result(ok, error) {
    Ok(ok)
    Error(error)
}
```

De acordo com https://hexdocs.pm/gleam_stdlib/gleam/option.html:

In other languages failible functions may return either `Result` *or* `Option` *depending on whether there is more information to be given about the failure. In Gleam all failible functions return* `Result`*, and* `Nil` *is used as the error if there is no extra detail to give. This consistency removes the boilerplate that would otherwise be needed to convert between* `Option` *and* `Result` *types, and makes APIs more predictable.*

```
> int.parse("10.1")
Error(Nil)
> int.parse("241")
Ok(241)
```

```
> int.divide(25, 3)
Ok(8)
> int.divide(12, 0)
Error(Nil)
```

```
> float.square_root(25.0)
Ok(5.0)
> float.square_root(-1.0)
Error(Nil)
```

```
> string.first("")
Error(Nil)
> string.first("casa")
Ok("c")
```

Exemplo soma de string

Projete uma função que receba como parâmetro duas strings, e se as duas representarem inteiros, devolva a soma dos valores em forma de string.

Exemplo soma de string

```gleam
pub fn soma(
  a: String,
  b: String,
) -> Result(String, Nil) {
  todo
}

pub fn soma_examples() {
  check.eq(soma("31", "4"), Ok("35"))
  check.eq(soma("31", "a"), Error(Nil))
  check.eq(soma("a", "4"), Error(Nil))
  check.eq(soma("a", "b"), Error(Nil))
}
```

```gleam
pub fn soma(a, b) -> Result(String, Nil) {
  case int.parse(a) {
    Ok(a) -> case int.parse(b) {
      Ok(b) -> Ok(int.to_string(a + b))
      Error(_) -> Error(Nil)
    }
    Error(_) -> Error(Nil)
  }
}
```

```gleam
pub fn soma(a, b) -> Result(String, Nil) {
  case int.parse(a), int.parse(b) {
    Ok(a), Ok(b) -> Ok(int.to_string(a + b))
    _, _ -> Error(Nil)
  }
}
```

Como podemos utilizar o tipo `Result` para lidar com a questão do preço, que deve ser positivo?

A opção mais direta é validar o preço na função `seleciona_combustivel` e devolver `Error` se um dos preços não for positivo.

```gleam
/// O preço do litro do combustível,
/// deve ser um número positivo.
type Preco = Float

pub fn seleciona_combustivel(
  preco_alcool: Preco,
  preco_gasolina: Preco,
) -> Result(Combustivel, Nil) {
  case preco_alcool <= 0.0 ||
       preco_gasolina <= 0.0 {
    True -> Error(Nil)
    False -> todo
  }
}
```

Qual é a limitação dessa abordagem?

Em todos os lugares que `Preco` é utilizado precisamos fazer a validação; ou podemos assumir que o preço foi validado anteriormente.

Podemos melhorar? Sim!

Validação

A ideia é definir um TAD, e fazer a validação do valor no construtor do tipo.

Dessa forma, não é possível construir uma instância do tipo que seja inválida.

Usamos a palavra chave **opaque** para criar um TAD em Gleam.

Apenas o módulo que define um tipo **opaque** tem acesso aos seus componentes.

Validação

```gleam
/// O preço do litro do combustível.
pub opaque type Preco {
    Preco(valor: Float)
}

/// Devolve Ok(Preco) com o valor *v* se
/// v > 0, Error(Nil) caso contrário.
pub fn preco(v: Float) -> Result(Preco, Nil) {
  case v >. 0.0 {
    True -> Ok(Preco(v))
    False -> Error(Nil)
  }
}

/// Devolve o valor em *p*.
pub fn valor(p: Preco) -> Float {
  p.valor
}
```

```gleam
pub fn seleciona_combustivel(
  preco_alcool: Preco,
  preco_gasolina: Preco,
) -> Combustivel {
  case valor(preco_alcool) <=.
         0.7 *. valor(preco_gasolina) {
    ...
  }
}

pub fn seleciona_combustivel_examples() {
  let assert Ok(alcool) = preco(4.2)
  let assert Ok(gasolina) = preco(6.1)
  check.eq(
    seleciona_combustivel(alcool, gasolina),
    Alcool,
  )
}
```

Revisão

Vimos com mais detalhes como desenvolver a etapa de definição de tipos de dados.

Aprendemos que devemos considerar dois princípios no projeto de tipos de dados

- Faça os valores válidos representáveis.
- Faça os valores inválidos irrepresentáveis.

Vimos como definir novos tipos de dados usando tipos algébricos:

- Estruturas (tipo produto)
- Uniões e enumerações (tipo soma)

Discutimos como os tipos de dados guiam o processo de projeto de programas:

- Um tipo soma com N casos sugere pelo menos N exemplos;
- Um tipo soma com N casos sugere um corpo com uma análise de N casos.

Vimos como usar tipos somas para lidar com valores opcionais, erros e validação:

- O tipo `Option` é usado para valores opcionais;
- O tipo `Result` é utilizado para representar sucesso ou falha de uma função;
- Tipos opacos podem ser utilizados para representar valores que foram validados.

Referências

Referências

Básicas

- Tipos de dados em Gleam
- Tipos opacos em Gleam
- Vídeo Making Impossible States Impossible
- Parse, don't validade

Leitura recomendada

- Expression problem