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

Requisitos de um 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?

Requisitos de um tipo de dado

Um inteiro é adequado para representar a quantidade de pessoas em um planeta? E um natural de 32 bits? E um natural?

- Um inteiro não é adequado pois um número inteiro pode ser negativo mas a quantidade de pessoas em um planeta não pode, ou seja, o tipo de dado permite a representação de valores inválidos.
- Uma 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, nem todos os valores válidos podem ser representados.
- 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.

Requisitos de um tipo de dado

Diretrizes para projeto de tipos de dados:

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

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** ou **estruturas**.

Em Racket utilizamos a forma especial `struct` para definir estruturas.

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

Definição

```
(struct ponto (x y))
```

Construção

```
(define p1 (ponto 3 4))

(define p2 (ponto 8 2))
```

Decomposição

```
> (ponto-x p1)
3
> (ponto-y p1)
4
> (ponto-x p2)
8
```

Teste de tipo

```
> (ponto? p1)
#t
> (ponto? "ola")
#f
```

Sintaxe de `struct`

Uma aproximação da sintaxe do `struct` é

```
(struct <id-estrutura> (<id-campo-1> ...))
```

Funções definidas na criação de uma estrutura

Funções definidas com `struct`

```
;; Construtor
id-estrutura

;; Predicado que verifica se um objeto
;; é do tipo da estrutura
id-estrutura?

;; Seletores
id-estrutura-id-campo
```

Por exemplo, a estrutura

```
(struct ponto (x y))
```

Define as funções

```
;; Construtor
ponto

;; Predicado
ponto?

;; Seletores
ponto-x
ponto-y
```

Note que o construtor, o predicado de tipo e os seletores criados por `struct` são funções comuns, e portando são utilizados como todas as outras funções.

```
> (struct ponto (x y))
> ponto
#<procedure:ponto>
> ponto?
#<procedure:ponto?>
> ponto-x
#<procedure:ponto-x>
> ponto-y
#<procedure:ponto-y>
```

Por padrão, ao exibir uma instância de uma estrutura o Racket não exibe o valor dos campos (para preservar o encapsulamento).

```
(struct ponto (x y))
```

```
> (ponto (+ 1 2) 4)
#<ponto>
```

Podemos usar a palavra chave `#:transparent` para tornar a estrutura “transparente” (podemos ver os valores dos campos).

```
(struct ponto (x y) #:transparent)

; mesmo formato de criação e de exibição
> (ponto (+ 1 2) 4)
(ponto 3 4)
```

Estruturas transparentes e a função `equal?`

Além de mudar a forma que o ponto é exibido, a palavra chave `#:transparent` também altera o funcionamento da função **`equal?`**.

```
;; Por padrão, dois pontos são iguais se eles são
;; o mesmo ponto.
(struct ponto (x y))

(define p1 (ponto 3 4))
(define p2 (ponto 3 4))

> (equal? p1 p2)
#f
> (equal? p1 p1)
#t
```

Estruturas transparentes e a função `equal?`

```
;; Com :#transparent, dois pontos são iguais se os seus
;; campos são iguais.
(struct ponto (x y) #:transparent)

(define p1 (ponto 3 4))
(define p2 (ponto 3 4))

> (equal? p1 p2)
#t
> (equal? p1 p1)
#t
```

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

```
(struct ponto (x y))
;; Ponto representa um ponto no plano cartesiano
;;   x : Número - a coordenada x
;;   y : Número - a coordenada y
```

Podemos utilizar os seletores para 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`.

```
> (define p1 (ponto 3 4))
> (define p2 (ponto (ponto-x p1) 5))
> p2
(ponto 3 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

Racket oferece a forma especial `struct-copy` (referência), que facilita este tipo de operação.

Alterando dados estruturados

```
> (define p1 (ponto 3 4))
> (define p2 (struct-copy ponto p1 [y 5]))
> p2
(ponto 3 5)

> (define p3 (struct-copy ponto p2 [x 4]))
> p3
(ponto 4 5)

> ; podemos especificar o novo valor de mais de um campo
> ; não faz sentido para ponto... mas vale o exemplo!
> (define p4 (struct-copy ponto p2 [y 9] [x 6]))
> p4
(ponto 6 9)
```

Exemplo - outras linguagens

A ideia de estruturas imutáveis que são “atualizadas” através de cópias está presentes 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,
}

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 - distância

Defina uma função que calcule a distância de um ponto a origem.

Exemplo - distância

```
;; Ponto -> Número
;; Calcula a distância do ponto p a origem.
;; A distância de um ponto (x, y) até a origem é calculada
;; pela raiz quadrada de x^2 + y^2.
(examples
 (check-equal? (distancia-origem (ponto 0 7)) 7)
 (check-equal? (distancia-origem (ponto 1 0)) 1)
 ;; (sqrt (+ (sqr 3) (sqr 4))
 (check-equal? (distancia-origem (ponto 3 4)) 5))
(define (distancia-origem p) 0)

(define (distancia-origem p)
  (sqrt (+ (sqr (ponto-x p))
           (sqr (ponto-y p)))))
```

Enumerações

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.

Como representar um tipo de usuário?

Criando um tipo **enumeração** com os valores possíveis para o tipo.

O Racket não suporta a criação de tipos enumerados, mas mesmo assim podemos utilizar o conceito.

Vamos ver exemplos em Python e Rust e depois veremos como fazer em Racket.

```python
class TipoUsuario(Enum):
    '''
    Representa um tipo de usuário
    do RU da UEM.
    '''
    ALUNO = auto()
    # Servidores que recebem até
    # 3 salários mínimos.
    SERVIDOR_ATE_3 = auto()
    # Servidores que recebem mais
    # do que 3 salários mínimos.
    SERVIDOR_MAISQ_3 = auto()
    DOCENTE = auto()
    EXTERNO = auto()
```

```python
tp: TipoUsuario = TipoUsuario.ALUNO
# Comparação por igualdade
assert tp != TipoUsuario.DOCENTE

# Representação textual
assert tp.name == "ALUNO"

# Verificação de tipo (mypy)
# atribuição inválida
tp = "externo"
```

```rust
/// Representa o tipo de um usuário do
/// RU da UEM.
#[derive(PartialEq, Debug)]
enum TipoUsuario {
    Aluno,
    // Servidor que recebe até 3
    // salários mínimos.
    ServidorAte3,
    // Servidor que recebe mais
    // do que 3 salários mínimos.
    ServidorMaisq3,
    Docente,
    Externo,
}
```

```rust
let mut tp = TipoUsuario::ServidorAte3;
// Comparação por igualdade
assert!(tp == TipoUsuario::ServidorAte3);

// Verificação de tipo
// Atribuição inválida
tp = "servidor";
```

Embora o Racket não suporte a definição de tipos enumerados, podemos registrar em forma de comentários os possíveis valores para o tipo (como fizemos com combustível e alinhamento). Mesmo que o Racket não “entenda” os comentários, eles são úteis pois registram a intenção do projetista.

```
;; TipoUsuario representa um tipo de usuário do RU da UEM.
;; TipoUsuario é um dos valores:
;; - "aluno"
;; - "servidor<=3" - servidor que recebe até 3 salários mínimos
;; - "servidor>3" - servidor que recebe acima de 3 salários mínimos
;; - "docente"
;; - "externo"
```

Especificação

```
;; TipoUsuario InteiroPositivo -> NúmeroPositivo
;; Determina o custo de *quant* tíquetes para um usuário do tipo *tu*.
;; O custo de um tíquete é
;; - "aluno"        5,0
;; - "servidor<=3"  5,0
;; - "servidor>3"  10,0
;; - "docente"     10,0
;; - "externo"     19,0
(define (custo-tiquetes tu quant) ...)
```

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

Pelo menos um para cada valor da enumeração.

```
(examples
  (check-equal? (custo-tiquetes "aluno" 3) 15.0) ; (* 3 5.0)
  (check-equal? (custo-tiquetes "servido<=3" 2) 10.0) ; (* 2 5.0)
  (check-equal? (custo-tiquetes "servido>3" 2) 20.0) ; (* 2 10.0)
  ...)
```

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

Criando um caso para cada valor da enumeração.

Implementação

```
(define (custo-tiquetes tu quant)
  (cond
    [(equal? tu "aluno")                    ]
    [(equal? tu "servidor<=3")              ]
    [(equal? tu "servidor>3")                ]
    [(equal? tu "docente")                   ]
    [(equal? tu "externo")                   ]))
```

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

Implementação

```
(define (custo-tiquetes tp quant)
  (cond
    [(equal? tu "aluno")       (* quant 5.0)]
    [(equal? tu "servidor<=3") (* quant 5.0)]
    [(equal? tu "servidor>3")  (* quant 10.0)]
    [(equal? tu "docente")     (* quant 10.0)]
    [(equal? tu "externo")     (* quant 19.0)]))
```

Podemos simplificar? Sim, podemos agrupas os casos iguais. Fica como atividade.

O que acontece se esquecermos de tratar um caso de um tipo enumerado na implementação de uma função em Racket?

O programa continua funcionando como o esperado se aquela entrada nunca for utilizada. No entanto, o código irá falhar ou produzir uma resposta incorreta se a entrada para o caso que está faltando for utilizada.

Os testes podem nos ajudar nesses casos, mas não é garantido. Por exemplo, suponha que existam várias funções que processam dados do tipo `TipoUsuario`, e que adicionamos mais um valor possível para o tipo, por exemplo `"visitante"`, como saber todos os lugares que o código precisa ser atualizado? Os testes e o Racket não nos ajudam nesse casos...

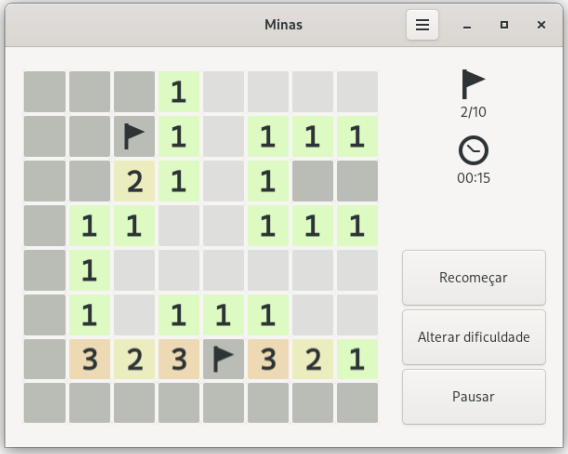
Já o mypy e o compilador do Rust sinalizam casos que não estão sendo tratados. Como são analisadores estáticos, elas fazem isso sem precisarem executar o código.

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.

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 item relacionados, então definiríamos uma estrutura. Além disso, cada item tem dois estados possíveis, então poderíamos usar booleano para representar cada estado.

```
(struct quadrado (mina? aberto? bandeira?) #:transparent)
;; Representa um quadrado no jogo campo minado.
;;  mina?    : Bool - #t se tem uma mina no quadrado, #f caso contrário
;;  aberto?  : Bool - #t se o quadrado está aberto, #f caso contrário
;;  bandeira?: Bool - #t se tem uma bandeira no quadrado, #f caso contrário
```

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`	`#t`	Sim
`#f`	`#t`	`#f`	Sim
`#f`	`#t`	`#t`	Não
`#t`	`#f`	`#f`	Sim
`#t`	`#f`	`#t`	Sim
`#t`	`#t`	`#f`	Sim
`#t`	`#t`	`#t`	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.

```
;; Estado é um dos valores:
;; - "aberto"
;; - "fechado"
;; - "fechado-com-bandeira"

(struct quadrado (mina? estado) #:transparent)
;; Representa um quadrado no jogo campo minado.
;;  mina? : Bool - #t se tem uma mina no quadrado, #f caso contrário
;;  estado: Estado - o estado do quadrado
```

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

```
;; Acao é um dos valores:
;; - "abrir"
;; - "adicionar-bandeira"
;; - "remover-bandeira"
```

Exemplo - Ação campo minado

Especificação

Quais são as entradas para a função? Um quadrado e uma ação.

Qual é a saída da função? Um 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

```
(examples
  ; q
  (check-equal? (atualiza-quadrado (quadrado #f "aberto") "abrir")
                (quadrado #f "aberto"))
  ; (struct-copy quadrado q [estado "aberto"])
  (check-equal? (atualiza-quadrado (quadrado #f "fechado") "abrir")
                (quadrado #f "aberto"))
  ...)
```

Implementação

Se o comportamento de uma função depende apenas de um valor enumerado, qual é a estrutura inicial do corpo da função? Uma seleção com uma condição 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

```
(define (atualiza-quadrado q acao)
  (define estado (quadrado-estado q))
  (cond
    [(equal? estado "aberto")
     (cond
       [(equal? acao "abrir") ...]
       [(equal? acao "adicionar-bomba") ...]
       [(equal? acao "remover-bomba") ...])]
    [(equal? estado "fechado")
     (cond
       [(equal? acao "abrir") ...]
       [(equal? acao "adicionar-bomba") ...]
       [(equal? acao "remover-bomba") ...])]
    [(equal? estado "fechado-com-bandeira")
     (cond
       [(equal? acao "abrir") ...]
       [(equal? acao "adicionar-bomba") ...]
       [(equal? acao "remover-bomba") ...])]))
```

```
(define (atualiza-quadrado q acao)
  (define estado (quadrado-estado q))
  (cond [(and (equal? estado "aberto")
              (equal? acao "abrir")) ...]
        [(and (equal? estado "aberto")
              (equal? acao "adicionar-bomba")) ...]
        [(and (equal? estado "aberto")
              (equal? acao "remover-bomba")) ...]
        [(and (equal? estado "fechado")
              (equal? acao "abrir")) ...]
        [(and (equal? estado "fechado")
              (equal? acao "adicionar-bomba")) ...]
        [(and (equal? estado "fechado")
              (equal? acao "remover-bomba")) ...]
        [(and (equal? estado "com-bomba")
              (equal? acao "abrir")) ...]
        [(and (equal? estado "com-bomba")
              (equal? acao "adicionar-bomba")) ...]
        [(and (equal? estado "com-bomba")
              (equal? acao "remover-bomba")) ...]))
```

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

```
(define (atualiza-quadrado q acao)
  (define estado (quadrado-estado q))
  (cond
    [(and (equal? estado "fechado")
          (equal? acao "abrir"))
     (struct-copy quadrado q [estado "aberto"])]
    [(and (equal? estado "fechado")
          (equal? acao "adicionar-bandeira"))
     (struct-copy quadrado q [estado "fechado-com-bandeira"])]
    [(and (equal? estado "fechado-com-bandeira")
          (equal? acao "remover-bandeira"))
     (struct-copy quadrado q [estado "fechado"])]
    [else q]))
```

Uniões

Problema - 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.

```
(struct estado-tarefa (executando duracao msg_sucesso codigo_err msg_err))
;; Representa o estado de uma tarefa.
;; executando: Bool - #t se a tarefa está em execução, #f caso contrário
;; duracao: Número - tempo que durou a execução da tarefa
;; msg_sucesso: String - mensagem caso a tarefa tenha sido executada com sucesso
;; codigo_err: Número - código de erro se a execução da tarefa falhou
;; msg_err: String - mensagem de erro se a execução da tarefa falhou
```

Qual é o problema dessa representação?

Possíveis estados inválidos. O que significa

```
(estado-tarefa #t 10 "Ótimo desempenho" 123 "Falha na conexão")
```

?

Como evitar esse problema?

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

- Em execução
- Sucesso, com uma duração e uma mensagem
- Erro, 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 conjunto.

- 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.

Antes de vermos como expressar uniões em Racket, vamos ver como uniões funcionam em um sistema estático de tipo (discutido em sala).

Agora podemos prosseguir com o projeto do programa em Racket.

```
(struct executando ())
;; Representa que uma tarefa está em execução.

(struct sucesso (duracao msg))
;; Representa o estado de uma tarefa que finalizou a execução com sucesso.
;; duracao: Número - tempo de execução em segundos
;; msg    : String - mensagem de sucesso gerada pela tarefa

(struct erro (codigo msg))
;; Representa o estado de uma tarefa que finalizou a execução com falha.
;; código: Número - o código da falha
;; msg   : String - mensagem de erro gerada pela tarefa
```

Agora podemos definir o tipo para estado da tarefa como uma união de três casos:

```
;; EstadoTarefa é um dos valores:
;; - (executando)             A tarefa está em execução
;; - (sucesso Número String)  A tarefa finalizou com sucesso
;; - (erro Número String)     A tarefa finalizou com falha
```

```
;; EstadoTarefa -> String
;; Produz uma string amigável para o usuário para descrever o estado da tarefa.
(define (msg-usuario estado) "")
```

Quantos exemplos são necessários? Pelo menos um para cada classe de valor. (Note que 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)

```
(examples
 (check-equal? (msg-usuario (executando))
               "A tarefa está em execução.")
 (check-equal? (msg-usuario (sucesso 12 "Os resultados estão corretos"))
               "Tarefa concluída (12s): Os resultados estão corretos.")
 (check-equal? (msg-usuario (erro 123 "Número inválido '12a'"))
               "A tarefa falhou (err 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, `EstatoTarefa`. São três casos, dependendo do caso, podemos usar seletores específicos.

```
(define (msg-usuario estado)
  (cond
    [(executando? estado)
     ...]
    [(sucesso? estado)
     ...
     ... (sucesso-duracao estado)
     ... (sucesso-msg estado)]
    [(erro? estado)
     ...
     ... (erro-codigo estado)
     ... (erro-msg estado)]))
```

```
(define (msg-usuario estado)
  (cond
    [(executando? estado)
     "A tarefa está em execução."]
    [(sucesso? estado)
     (format "Tarefa concluída (~as): ~a."
             (sucesso-duracao estado)
             (sucesso-msg estado))]
    [(erro? estado)
     (format "A tarefa falhou (err ~a): ~a."
             (erro-codigo estado)
             (erro-msg estado))]))
```

Considerações

Nos vimos que os tipos algébricos de dados podem ser usados para modelar informações de forma mais precisa, aumentando a confiabilidade do programa.

Mas a sua utilidade pode ser ampliada se a linguagem oferecer algum tipo de verificação estática que suporte tipos algébricos.

Considere por exemplo uma alteração nos requisitos do nosso projeto: as tarefas agora podem ficar em uma fila antes de iniciar a execução.

Supondo que o programa utilize `EstadoTarefa` em mais que um lugar, como podemos saber todos os lugares que precisamos alterar o código para levar em consideração o novo estado "Fila"?

Em Racket não podemos... mas em Typed Racket podemos!

Uniões em Racket tipado (typed racket)

Considere as seguintes definições

```
#lang typed/racket

(struct executando ())

(struct sucesso ([duracao : Number]
                 [msg : String]))

(struct erro ([codigo : Number]
              [msg : String]))

(define-type EstadoTarefa
  (U executando sucesso erro))
```

E a função

```
(: msg-usuario (-> EstadoTarefa String))
(define (msg-usuario estado)
  (cond
    [(executando? estado)
     "A tarefa está em execução"]
    [(sucesso? estado)
     (format "A tarefa finalizou com sucesso (~as): ~a."
             (sucesso-duracao estado)
             (sucesso-msg estado))]
    [(erro? estado)
     (format "A tarefa falhou (erro ~a): ~a."
             (erro-codigo estado)
             (erro-msg estado))]))
```

O que acontece se alteramos a definição do estado da tarefa da seguinte maneira?

```
(struct fila ())
(define-type EstadoTarefa (U fila executando sucesso erro))
```

O analisador estático do Racket indica um erro no **cond**, pois nem todos os casos são tratados.

```
Type Checker: type mismatch
  expected: String
  given: Void
```

Outras linguagens

União em outras linguagens

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
```

```
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.

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.

Por fim, vimos que um analisador estático amplia bastante a utilidade dos tipos algébricos de dados, garantindo que o código trate todos os casos na análise de tipos soma.

Referências

Referências

Básicas

- Vídeos Compound Data
- Vídeos Reference
- Vídeo Making Impossible States Impossible
- Seções 5.1 a 5.5 do Guia Racket

Leitura recomendada

- Expression problem