Tipos de dados

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

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

Estamos interessados em 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 **estruturas**.

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

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

Estruturas

Definição

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

Construção

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

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

Seletores

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

Verificação 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
```

Estruturas transparentes

Por padrão, ao exibir um dado estruturado o interpretador não exibe os campos do dado (para preservar o encapsulamento)

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

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

Estruturas transparentes

Podemos usar a palavra chave `#:transparent` para tornar a estrutura “transparente”

```
(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?`

Estruturas transparentes e a 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 propósito e campos da estrutura.

```
(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)
```

Este método é limitado

- 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.

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

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

> (define p4 (struct-copy ponto p2 [y 9] [x 6]))
> p4
(ponto 6 9)
```

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)
  (sqrt (+ (sqr (ponto-x p))
           (sqr (ponto-y p)))))
```

Exemplo contagem

Em um sistema de enquete cada possível resposta é identificada por uma cor: verde, vermelho, azul ou branco. Após todos os participantes responderem a enquete, é necessário contabilizar a quantidade de vezes que cada resposta foi selecionada. Como parte desse sistema, você deve projetar uma função que receba a contabilização atual das respostas e uma nova resposta e produza a contabilização atualizada.

Tipos enumerados

Como representar uma cor que pode ser verde, vermelho, azul ou branco?

Enumerando os seus valores em um tipo enumerado.

Embora o Racket não suporte a definição de tipos enumerados, podemos registrar em forma de comentários os possíveis valores de um “tipo”.

```
;; Cor é um dos valores:
;; - "verde"
;; - "vermelho"
;; - "azul"
;; - "branco"
```

```
(struct contagem (verde vermelho azul branco) #:transparent)
;; Uma contagem das respostas de cada cor
;;   verde: Número - número de respostas verde
;;   vermelho: Número - número de respostas vermelho
;;   azul: Número - número de respostas azul
;;   branco: Número - número de respostas branco

;; Resposta é um dos valores
;; - "verde"
;; - "vermelho"
;; - "azul"
;; - "branco"
```

```
;; Resposta Contagem -> Contagem
;; Atualiza a contagem cont adicionando a resposta res.
(define (atualiza-contagem res cont) ...)
```

Exemplos

```
(examples
 (check-equal? (atualiza-contagem "verde" (contagem 4 5 1 2))
               (contagem 5 5 1 2))

               ;; (struct-copy contagem (contagem 4 5 1 2)
               ;;               [verde (add1 (contagem-verde (contagem 4 5 1 2)))])

 (check-equal? (atualiza-contagem "vermelho" (contagem 4 5 1 2))
               (contagem 4 6 1 2))
 ...
```

Quantos exemplos são necessários para funções que processam valores de tipos enumerados?
Pelo menos um para cada valor da enumeração.

Como iniciamos a implementação de uma função que processa um valor de tipo enumerado?
Criando um caso para cada valor da enumeração.

```
(define (atualiza-contagem res cont)
  (cond
    [(equal? res "verde")


    [(equal? res "vermelho")


    [(equal? res "azul")


    [(equal? res "branco")

                                                    ]))
```

```
(define (atualiza-contagem res cont)
  (cond
    [(equal? res "verde")
     (struct-copy contagem
                  cont [verde (add1 (contagem-verde cont))])]
    [(equal? res "vermelho")
     (struct-copy contagem
                  cont [vermelho (add1 (contagem-vermelho cont))])]
    [(equal? res "azul")
     (struct-copy contagem
                  cont [azul (add1 (contagem-azul cont))])]
    [(equal? res "branco")
     (struct-copy contagem
                  cont [branco (add1 (contagem-branco cont))])]))
```

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 um tempo específico 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 tempo msg_sucesso codigo_err msg_err))
;; Representa o estado de uma tarefa
;; executando: Bool - #t se a tarefa está em execução, #f caso contrário
;; tempo: 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 um tempo e uma mensagem
- Falhado, 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.

Vamos ver uma analogia para nos auxilar a entender o conceito de união.

Se consideramos um tipo de dado como um conjunto de possíveis valores daquele tipo, então podemos dizer que:

- 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.

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. Antes de definir o tipo que representa o estado da tarefa, precisamos definir os tipos para sucesso e erro.

```
(struct sucesso (tempo msg))
;; Representa o estado de uma tarefa que finalizou a execução com sucesso
;; tempo: 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
    [(and (string? estado) (string=? estado "Executando"))
    ]
    [(sucesso? estado)
     ... (sucesso-tempo estado)
     ... (sucesso-msg estado)
    ]
    [(erro? estado)
     ... (erro-codigo estado)
     ... (erro-msg estado)
    ]))
```

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

União em outras linguagens

A união de tipos é bastante útil e com a popularização da programação funcional, também tem sido adicionada a diversas linguagens de programação. Vamos ver algumas delas.

```python
from dataclasses import dataclass
from typing import Literal

@dataclass
class Sucesso:
    duracao: int
    msg: str

@dataclass
class Erro:
    codigo: int
    msg: str

EstadoTarefa = Literal["Executando"] | Sucesso | Erro
```

```
def mensagem(estado: EstadoTarefa) -> str:
    if isinstance(estado, str):
        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 str(estado):
            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}'
```

```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(tempo, msg) =>
            format!("A tarefa finalizou com sucesso ({tempo}s): {msg}"),
        EstadoTarefa::Erro(codigo, msg) =>
            format!("A tarefa falhou (erro {codigo}): {msg}"),
    }
}
```

```java
sealed interface EstadoTarefa permits Executando, Sucesso, Erro {};
record Executando() implements EstadoTarefa {};
record Sucesso(int tempo, String sucesso) 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.tempo(), s.sucesso())
        case Erro e ->
            String.format("A tarefa falhou (erro %d): %s", e.erro(), e.msg());
    };
}
```

Revisão

Vimos duas formas diferentes de definir novos tipos de dados:

- Estruturas: quando diversos valores relacionados são agrupados para representar uma entidade
- Uniões: quando uma entidade é descrita pela união de diversas classes de valores

No contexto de programação funcional, essas construções de tipos são chamadas de tipos de dados algébricos

- As estruturas são chamadas de tipos produto
- As uniões são chamadas de tipos somas

Essa “analogia” com a álgebra é interessante pois nos permite entender mais facilmente alguns aspectos da construção de tipos.

Referências

Referências

Básicas

- Vídeos Compound Data
- Vídeos Reference
- Seções 5.1 do Guia Racket

Complementares

- Seções 4.1 da Referência Racket

Leitura recomendada

- Expression problem