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.

```
(struct ponto (x y))
(define p1 (ponto 3 4))
(define p2 (ponto 8 2))
> (ponto-x p1)
3
> (ponto-y p1)
4
> (ponto-x p2)
8
> (ponto-y p2)
2
> (ponto? p1)
#t
> (ponto? "ola")
#f
```

Sintaxe de `struct`

Uma aproximação da sintaxe do `struct` é

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

Funções definidas pelo `struct`

```
;; Construtor
id-estrutura

;; Predicado que teste se o valor é do tipo definido
id-estrutura?

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

Funções definidas na criação de uma estrutura

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 3 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 3 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
```

Alterando dados estruturados

Se quisermos mudar um campo de um dado estruturado, temos que criar uma cópia 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, todas as operações de “cópia” 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.
(examples
 (check-equal? (distancia-origem (ponto 0 7)) 7)
 (check-equal? (distancia-origem (ponto 1 0)) 1)
 (check-equal? (distancia-origem (ponto 3 4)) 5))

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

Exercício: classificação retângulo

Defina uma estrutura para representar um retângulo. Em seguida defina uma função que classifique um retângulo em largo (largura maior que altura), alto (altura maior que largura) ou quadrado (altura igual a largura).

Exercício: horas, minutos e segundos

Defina uma função que converta uma quantidade de segundos para uma quantidade de horas, minutos e segundos equivalente. A quantidade de segundos e de minutos da resposta deve ser menor que 60.

Exercício: conflito

Uma determinada sala de reunião pode ser usada das 8:00h às 18:00h. Cada interessado em utilizar a sala faz uma reserva indicando o intervalo de tempo que gostaria de utilizar a sala. Como parte de um sistema de reservas, você deve projetar uma função que verifique se duas reservas podem ser atendidas, ou seja, não têm conflito de horário.

Exercício: 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.

Como representar a classificação de um retângulo em alto, largo ou quadrado?

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"
```

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? (Segunda etapa do processo de projeto de funções - Definição dos tipos de dados)

Vamos tentar uma estrutura.

```
(struct estado-tarefa (executando tempo msg_sucesso codigo_err msg_err))
;; Representa o estado de uma tarefa
;; executando: Bool - true se a tarefa está em execução, false 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 true 10 "Ótimo desempenho" 123 "Falha na conexão)`?

Como evitar esse problema?

Podemos criar uma união de classes de valores. Veja o vídeo da aula para mais detalhes!

```
(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

;; 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
```

Agora podemos ir para a especificação da função.

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

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. Note também que como `EstadoTarefa` é definido como a união de três classes de valores, então precisamos de pelo menos um exemplo para cada classe de valores.

Agora vamos para a implementação.

Mesmo sem saber detalhes da implementação, podemos definir uma 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)]))
```

Agora é só preencher as lagunas!

Uniões

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

```rust
pub enum EstadoTarefa {
    Executando,
    Sucesso(u32, String),
    Erro(u32, String)
}

pub fn mensagem(estado: &EstadoTarefa) -> String {
    use EstadoTarefa::*;
    match estado {
        Executando =>
            "A tarefa está em execução".to_string(),
        Sucesso(tempo, msg) =>
            format!("A tarefa foi concluída ({}s): {}", tempo, msg),
        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());
    };
}
```

Tipos de dados

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