

Linguagens de Programação

Algebraic Data Types e Pattern Matching

Samuel da Silva Feitosa

Aula 9

Tuplas, ADTs e Pattern Matching



Tuplas

- Tupla é uma estrutura de dados formada por uma sequência de valores **possivelmente de tipos diferentes**.
- Uma tupla é formada por:
 - Sequência de valores de expressões separadas por vírgula e delimitada por parênteses.
 - Exemplo: (exp1, ..., expn)
- O tipo de uma tupla é o **produto cartesiano** dos tipos dos seus componentes.

Tuplas

- A tabela a seguir mostra alguns exemplos:

tupla	tipo
<code>('A', 't')</code>	<code>(Char, Char)</code>
<code>('A', 't', 'o')</code>	<code>(Char, Char, Char)</code>
<code>('A', True)</code>	<code>(Char, Bool)</code>
<code>("Joel", 'M', True, "COM")</code>	<code>(String, Char, Bool, String)</code>
<code>(True, ("Ana", 'f'), 43)</code>	<code>Num a => (Bool, (String, Char), a)</code>
<code>()</code>	<code>()</code>
<code>("nao eh tupla")</code>	<code>String</code>

- Não existe tupla com um único elemento.
- Principais operações com tuplas: *fst* e *snd*.

Trabalhando com Tipos de Dados

- Haskell oferece tuplas para agrupar um número fixo de componentes com tipos diferentes.
- E também oferece listas que podem armazenar elementos com tipos homogêneos.
- Parece suficiente para iniciar a modelagem de um sistema.
 - Por exemplo: um cliente de nome Paulo, com 25 anos de idade e comprador de dois produtos.

```
Prelude> ("Paulo", 25, ["Refrigerante", "Salgadinho"]  
("Paulo",25,["Refrigerante","Salgadinho"]))
```

Trabalhando com Tipos de Dados

- Dois problemas com a abordagem anterior.
 - Código fica difícil de ler, por conta das chamadas de `fst`, `snd`, e `head`.
 - Perde-se a força do sistema de tipos, pois não é possível distinguir (a partir dos tipos) um cliente de um peixe com nome, comprimento e oceanos que habita.

```
Prelude> ("Tubarao-Martelo", 355, ["Atlantico", "Indico"])  
("Tubarao-Martelo",355,["Atlantico","Indico"])
```

Algebraic Data Types (ADTs)

- O tipo de dado mais básico que podemos criar em Haskell.
- Dividido em duas partes
 - Um nome para o tipo que será usado para representar seus valores.
 - Um conjunto de construtores que serão usados para construir novos valores. Estes construtores podem ter argumentos que guardam valores de tipos específicos.

Algebraic Data Types (ADTs)

- Para deixar as ideias mais claras, vamos iniciar a modelagem de **Clientes**.
- Nosso sistema terá três tipos de clientes.
 - **Organização governamental**, que são conhecidas pelo seu nome.
 - **Empresas**, para o qual poderemos informar um nome, um código de identificação, uma pessoa de contato, e o cargo dessa pessoa na empresa.
 - **Clientes individuais**, conhecidos pelo nome, sobrenome, e se ele quer receber mais informações sobre ofertas e descontos.

Algebraic Data Types (ADTs)

- Representaremos clientes em Haskell com o seguinte código.
 - Vamos criar uma nova pasta para armazenar os códigos, e criar o arquivo **Cliente.hs**.

```
data Cliente = OrgGov String
              | Empresa String Integer String String
              | Indivíduo String String Bool
```

- Vamos executar alguns testes no GHCi.
 - Podemos testar o tipo de uma expressão.
 - Porém, se tentarmos imprimir algum resultado na tela, receberemos uma mensagem de erro.
- Para evitar o problema, vamos usar *deriving Show*.

Algebraic Data Types (ADTs)

- Podemos utilizar um ADT dentro de outro.
 - Vamos criar um novo tipo de dado para representar uma pessoa e usar esse tipo dentro de Cliente.

```
data Cliente = OrgGov String
              | Empresa String Integer String String
              | Indivíduo Pessoa Bool
              deriving Show
```

```
data Pessoa = Pessoa String String
              deriving Show
```

- Em um módulo, todos os nomes de construtores devem ter nomes diferentes.

Algebraic Data Types (ADTs)

- Algumas vezes podemos estar interessados apenas em alternativas, sem nenhuma informação extra nos construtores.
 - Por exemplo, você pode adicionar a informação de gênero de uma pessoa.

```
data Genero = Masculino | Feminino | Outro
            deriving Show
```

- **Mini exercício:**
 - Adicione o argumento **Genero** para pessoa.
 - Teste o uso deste novo tipo de dado no GHCi.

Type e newtype

- Existem outras duas formas de introduzir novos tipos em Haskell.
- *type*: introduz um sinônimo para um tipo e usa os mesmos construtores de dados.

```
type Name = String
```

- *newtype*: introduz uma mudança de nome em um tipo, e requer que sejam providos novos construtores.

```
newtype FirstName = FirstName String
```

Pattern Matching

- Agora vamos aprender a criar funções usando nossos novos tipos de dados.
 - Para acessar as informações internas de um tipo de dados, utilizaremos o conceito de pattern matching.
- Usando este conceito, podemos visualizar a estrutura de um valor, incluindo o construtor que foi usado para criá-lo, e associar esses valores a ligações locais.

Pattern Matching

- Vamos criar uma função para que retorne o nome de um cliente.

```
nomeCliente :: Cliente -> String
nomeCliente cliente = case cliente of
    OrgGov nome                -> nome
    Empresa nome id resp cargo -> nome
    Indivíduo pessoa ads       ->
        case pessoa of
            Pessoa pNome sNome g -> pNome ++ " " ++ sNome
```

- No caso de uma organização governamental ou empresa, o nome do cliente aparece no primeiro componente do construtor.
- No caso do indivíduo, é preciso olhar dentro de **Pessoa** e concatenar o nome e sobrenome.

Pattern Matching

- É possível especificar padrões mais complexos, onde partes internas também são tipos de dados.
 - Vamos reescrever a função anterior de forma distinta.

```
nomeCliente' :: Cliente -> String
nomeCliente' cliente =
  case cliente of
    OrgGov nome          -> nome
    Empresa nome _ _ _   -> nome
    Indivíduo (Pessoa pNome sNome _) _ -> pNome ++ " " ++ sNome
```

- Aqui estamos fazendo o casamento de padrão do tipo **Indivíduo** e **Pessoa** na mesma linha de código.
- Note também o uso de *underline* no lugar dos parâmetros.

Pattern Matching

- Mas o que acontece se esquecermos de tratar algum dos construtores de um tipo de dado?

- Vejamos um exemplo:

```
nomeEmpresa :: Cliente -> String
nomeEmpresa cliente = case cliente of
  Empresa nome _ _ _ -> nome
```

- E vamos executar alguns exemplos:

```
*Cliente> nomeEmpresa (Empresa "IFSC" 1 "Samuel" "Professor")
"IFSC"
*Cliente> nomeEmpresa (OrgGov "IFSC")
*** Exception: /home/samuel/HaskellProjects/aula9/Cliente.hs:(34,23)-(35,50):
Non-exhaustive patterns in case
```


Pattern Matching

- Quando funções não são definidas sobre o domínio completo dos argumentos, chamamos esta função de *parcial*.
 - Haskell possui um tipo especial muito usado em funções parciais, chamado **Maybe**.
 - Podemos usar: **Maybe Int**, **Maybe String**, etc.

```
nomeEmpresa' :: Cliente -> Maybe String
nomeEmpresa' cliente = case cliente of
    Empresa nome _ _ _ _ _ -> Just nome
    _                        -> Nothing
```

Pattern Matching

- Como o uso de expressões **case** é muito comum para trabalhar com pattern matching, podemos codificar o padrão diretamente na definição da função.
 - Vamos reescrever a função **nomeCliente**.

```
nomeCliente'' :: Cliente -> String
nomeCliente'' (OrgGov nome)           = nome
nomeCliente'' (Empresa nome _ _ _)    = nome
nomeCliente'' (Individuo (Pessoa pn sn _) _) = pn ++ " " ++ sn
```

Considerações Finais

- Nesta aula estudamos basicamente três novos conceitos em linguagens funcionais.
 - Uso de tuplas, e a comparação do seu funcionamento com relação a listas.
 - Criação de novos tipos de dados compostos, conhecidos como *Algebraic Data Types* (ADTs).
 - Utilização de casamento de padrão (*pattern matching*) para obter informações dentro de tipos de dados algébricos.
- Todos estes conceitos são muito importantes na programação funcional e serão mais explorados.

Exercícios de Fixação

- Escreva uma tupla com três elementos, onde o primeiro é 4, o segundo é “Olá” e o terceiro é True.
- Use a combinação de **fst** e **snd** para extrair o número 4 da seguinte tupla: ((“Hello”, 4), True).
- Faça testes com as funções desenvolvidas em aula.
- Escreva outras funções que consultem informações sobre os tipos de dados criados.
 - Por exemplo: obter o nome de uma pessoa, obter o gênero de uma pessoa, obter o nome e cargo do responsável pela empresa, obter o indicador de recebimento de promoções de um indivíduo, etc.
 - Utilize ambos os formatos (usando **case** e pattern matching diretamente nos parâmetros da função).

Desafio

- Implementar o tipo algébrico **Nat** para representar números naturais.
 - Deve possuir dois construtores de tipo: **Zero** e o sucessor de um número (**Suc**).
 - Este tipo deve ser recursivo.
 - Exemplos:
 - **Zero** representa o número zero.
 - **Suc Zero** representa o número um.
 - **Suc (Suc Zero)** representa o número dois.
- Crie algumas funções sobre esse tipo:
 - Representação dos números de 1 à 4.
 - Conversões **nat2integer** e **integer2nat**.
 - Funções: **natAdd**, **natSub**, **natMul**.