Tipos de Dados Algébricos

Programação Funcional

Prof. Maycon Amaro

Revisão: Registros

Chamado também de *structs*, é a funcionalidade de várias linguagens para definir tipos abstratos de dados.

```
typedef struct {
  char* nome;
  int idade;
} TPessoa;
```

–

O tamanho da estrutura precisa ser conhecido, de forma que usa-se ponteiros para criar estruturas recursivas, como listas encadeadas.

```
typedef struct celula {
  int elemento;
  struct celula *prox;
} TLista;
```

Enumerações

Tipos que possuem poucos valores constantes. Algumas linguagens permitem associar valores adicionais em cada construtor.

```
enum TCorSemaforo {Vermelho, Azul, Amarelo};
```

Variants

Um tipo que pode possuir apenas um valor dentre dois (ou mais) tipos informados.

```
union TIdentificao {
  int codigo;
  char* nome;
}
```

A Álgebra de Tipos

O Zero

Um tipo sem valores (um conjunto vazio) é um tipo que não pode ser construído.

Se ignorarmos a possibilidade do `NULL`, não há valores para o seguinte tipo:

```
struct true_void {
  struct true_void *a;
}
```

Vamos ignorar a existência da constante `NULL` até o fim dessa aula.

O Um

Um tipo com apenas um possível valor (um conjunto unitário).

Em C, o tipo `void` é unitário. Funções do tipo `void` são funções que retornam uma constante predefinida pelo compilador, e é invisível ao programador.

```
void funcao(){
  return;
}
```

O Um

Mas para facilitar, vamos usar como representante um enum de um elemento só:

```
enum unit { unitValor };
```

A Soma (Disjunção)

Um tipo cujos valores possíveis são os valores de dois tipos quaisquer, ou seja, $A \cup B$. Variants são exatamente isso.

```
union TIdentificao {
  int codigo;
  char* nome;
}
```

O Produto (Conjunção)

Um tipo cujos valores são um par de dois valores de dois tipos quaisquer, ou seja, $A \times B$. Tuplas são exatamente isso. Podemos representá-las em C com uma `struct`:

```
struct Tupla {
  int x;
  char y;
}
```

Álgebra

$0 + 1 = 1$

```
union T1 {
  true_void x;
  unit y;
}
```

Se não der pra construir um `true_void`, então esse tipo só tem um único valor, o `unitValor` através de `y`.

Álgebra

$0 \times 1 = 0$

```
struct Tupla {
  true_void x;
  unit a;
}
```

Se não der pra construir um `true_void`, então não dá pra construir essa tupla também.

Álgebra

$1 + 1 = 2$

```
union {
  unit x;
  unit y;
}
```

Esse tipo tem dois possíveis valores: o `unitValor` construído a partir de `x` e o `unitValor` construído a partir de `y`.

Álgebra

$1 \times 1 = 1$

```
struct {
  unit x;
  unit y;
}
```

Esse tipo só tem um valor possível: `(unitValor, unitValor)`.

Álgebra

No λ-cálculo, tipos mais complexos são construídos utilizando essa *álgebra*, essa combinação de tipos através de disjunções e tuplas.

Em Haskell, isso já está bem mais abstraído e facilitado!

Haskell

A palavra reservada `data` serve para criar um tipo de dado algébrico.

```
data NomeDoTipo = ...
```

Já a palavra reservada `type` serve para criar um sinônimo de um tipo já existente.

```
type PontoCartesiano = (Int, Int)
```

O tipo `String` é na verdade um sinônimo de `[Char]`.

Em Haskell o operador | é usado para disjunção.

```
union TIdentificao {
  int codigo;
  char* nome;
}
```

```
data Identificacao = Codigo Int | Nome String
```

Disjunções estão no prelúdio como `Either`.

Uma enumeração nada mais é que uma disjunção de várias constantes.

```
enum CorSemaforo {Vermelho, Azul, Amarelo};
```

```
data CorSemaforo = Vermelho | Azul | Amarelo
```

Para agrupar elementos em uma conjunção, para separá-los por espaço.

```
struct Tupla {
  int x;
  int y;
}
```

```
data Tupla = Tupla Int Int
```

Tuplas já estão definidas no prelúdio como `( , )`.

Tipo unitário.

```
enum unit { unitValor };
```

```
data Unit = UnitValor
```

O tipo unitário em Haskell está definido como `()` e seu único valor possível também é `()`.

Tipo vazio (que não pode ser construído)

```c
struct true_void {
  struct true_void *a;
}
```

```haskell
data TrueVoid = TrueVoid TrueVoid
```

O tipo vazio está no prelúdio como `Void`.

Existe uma função no prelúdio `absurd :: Void -> a`.

Olhando pelo tipo, caso seja fornecido um valor `Void` como parâmetro (o que é impossível), essa função seria capaz de retornar um elemento de *qualquer* tipo.

A implementação dela é um *loop* infinito, forçado a ser avaliado estritamente.

Potenciação

Pela álgebra dos tipos, a potenciação seriam as funções.

Uma função `Int -> Int` tem exatamente n^n possíveis mapeamentos distintos, em que n é o número de elementos de `Int`.

Uma função `f :: () -> ()` tem exatamente 1 mapeamento único. $1^1 = 1$.

```
f :: () -> ()
f () = ()
```

Como `absurd` é a única possível implementação de uma função `Void -> Void`, **pela álgebra de tipos**, $0^0 = 1$.

Isso **não** é uma prova de que isso é verdade na matemática.

Definindo Tipos

Tipo Pessoa com um nome e uma idade.

```
data Pessoa = Pessoa Int String

exemplo :: Pessoa
exemplo = Pessoa 27 "Amy Winehouse"
```

Vamos melhorar isso em breve, com uso de `Records`.

Lista encadeada de inteiros.

```haskell
data ListaInt = Vazia | Add Int ListaInt

exemplo :: ListaInt
exemplo = Add 1 (Add 2 (Add 3 Vazia))
```

A definição de Lista do prelúdio é parecido com isso, mas ela é *genérica*.

Note que Haskell sabe lidar com tipos recursivos. Não precisa se preocupar com ponteiros.

Árvore Binária de inteiros.

```haskell
data TreeInt = EmptyNode | Node TreeInt Int TreeInt

exemplo :: TreeInt
exemplo =
  Node (Node EmptyNode 3 EmptyNode)
       5
       (Node EmptyNode 7 EmptyNode)
```

Casamento de Padrão

O casamento de padrão funciona com os tipos de dados algébricos.

```
arvoreEhVazia :: TreeInt -> Bool
arvoreEhVazia EmptyNode = True
arvoreEhVazia _ = False

profundidade :: TreeInt -> Int
profundidade EmptyNode = 0
profundidade (Node l x r) =
  1 + max (profundidade l) (profundidade r)
```

Records

Para tipos de dados algébricos da forma

```
data Tipo = Tipo Campo1 Campo2 Campo3 ...
```

podemos dar nome aos campos:

```
data Pessoa = Pessoa { idade :: Int, nome :: String }

exemplo :: Pessoa
exemplo =
  Pessoa { nome = "Amy Winehouse", idade = 27 }
```

O Casamento de Padrão funciona normalmente.

```
nomePessoa :: Pessoa -> String
nomePessoa (Pessoa {nome = x, idade = _}) = x
```

Ao criar um Record, o compilador automaticamente criará uma *função de projeção* para cada campo. Com o record

```
data Pessoa = Pessoa { idade :: Int, nome :: String }
```

automaticamente são criadas as funções

```
idade :: Pessoa -> Int
nome :: Pessoa -> String
```

Podemos criar uma nova instância de um Record reutilizando os valores de outra.

```
amy :: Pessoa
amy = Pessoa { nome = "Amy", idade = 27 }

wine :: Pessoa
wine = amy { nome = "Whinehouse" }
```

A idade de `wine` será 27.

Em Outras Linguagens

C++

Com uso de classes (Programação Orientada à Objetos), C++ e outras linguagens OO fornecem uma outra forma de criar estruturas.

Versões recentes da STL de C++ oferecem ainda os seguintes tipos:

```
std::pair<A, B> //tuplas
std::variant<A, B> // disjunção
```

Rust

`Enums` e `Structs` são funcionalidades separadas. Os `Enums` permitem valores associados, então funcionam como variants. Há também tuplas.

```
enum Identificacao {
  Codigo(i32),
  Nome(String),
}
```