Atividade VII

Problema 1

A

Em Haskell, o termo "classe" refere-se a classes de tipos, que são um conceito fundamental na definição de tipos polimórficos. Classes de tipos definem uma interface que especifica algum comportamento. Tipos que implementam esse comportamento são considerados instâncias dessa classe de tipo.

1. **Classes Primitivas (ou Básicas)**:
   - São classes de tipos fundamentais em Haskell.

   Exemplos:
   1. **Eq**: Suporta comparação de igualdade.
      - Métodos:
        - `(==)`: Testa se dois valores são iguais.
        - `(/=)`: Testa se dois valores são diferentes.

   2. **Ord**: Suporta comparação de ordenação.
      - Métodos:
        - `compare`: Compara dois valores e retorna um dos seguintes: `LT`, `EQ` ou `GT`.
        - `(<)`, `(<=)`, `(>)`, `(>=)`: Funções de comparação padrão.

   3. **Show**: Permite que os valores sejam apresentados como strings.
      - Métodos:
        - `show`: Retorna uma representação em string do valor.

2. **Classes Secundárias**:
   - São classes de tipos que dependem de outras classes de tipo. Ou seja, para um tipo ser uma instância de uma classe secundária, ele deve primeiro ser uma instância de outra classe (ou classes).

   Exemplos:
   1. **Enum**: Usado para tipos que têm sucessores e predecessores definidos.
      - Métodos:
        - `succ`: Retorna o sucessor de um valor.
        - `pred`: Retorna o predecessor de um valor.

   2. **Bounded**: Usado para tipos que têm um valor mínimo e máximo.
      - Métodos:
        - `minBound`: Retorna o menor valor do tipo.
        - `maxBound`: Retorna o maior valor do tipo.

3. **Num**: Classe numérica básica.
   - Métodos:
     - `(+)`, `(-)`, `(*)`: Operações aritméticas básicas.
     - `abs`: Valor absoluto.
     - `signum`: Indica o sinal do número.
     - `fromInteger`: Converte um valor inteiro para o tipo numérico.
   - Nota: Para que um tipo seja uma instância de `Num`, ele geralmente também será uma instância de outras classes, como `Eq` e `Show`.

B

Haskell tem um sistema bem definido de classes numéricas no módulo Prelude. Este sistema permite operações matemáticas e aritméticas em diversos tipos numéricos, e é projetado de forma hierárquica, onde classes mais específicas são subconjuntos de classes mais gerais.

Classe Num:

- É a classe numérica básica.
- Métodos:
  - (+), (-), (*): Operações aritméticas básicas.
  - negate: Retorna o valor negativo.
  - abs: Valor absoluto.
  - signum: Indica o sinal do número (retorna -1 para números negativos, 0 para zero e 1 para números positivos).
  - fromInteger: Converte um valor inteiro para o tipo numérico.
- Todos os números, sejam eles inteiros ou de ponto flutuante, são instâncias dessa classe.

Classe Integral:

- Representa números inteiros.
- É uma subclasse de Num.
- Métodos:
  - div, mod: Divisão e módulo.
  - divMod: Retorna um par contendo o resultado de div e mod.
  - toInteger: Converte para um valor inteiro.
- Os tipos padrão que são instâncias dessa classe são Int e Integer.

Classe Fractional:

- Representa números fracionários, ou seja, números que podem ter partes fracionárias.

- É uma subclasse de Num.
- Métodos:
  - (/): Divisão.
  - recip: Recíproco (inverso) de um número.
  - fromRational: Converte um valor racional para o tipo numérico.
- Tipos comuns que são instâncias desta classe incluem Float e Double.

Classe Floating:

- Representa números de ponto flutuante.
- É uma subclasse de Fractional.
- Métodos:
  - pi: O valor de π.
  - exp, log, sqrt: Funções exponencial, logarítmica e raiz quadrada.
  - (**): Exponenciação.
  - logBase: Logaritmo em uma base especificada.
  - sin, cos, tan: Funções trigonométricas básicas.
  - asin, acos, atan: Funções trigonométricas inversas.
- Float e Double são instâncias típicas desta classe.

Classe Real:

- Representa números reais.
- É uma subclasse de Num e Ord (para ordenação).
- Métodos:
  - toRational: Converte um número real para uma fração (racional).

Classe RealFrac:

- Representa a combinação de números reais e fracionários.
- É uma subclasse de Real e Fractional.
- Métodos:
  - properFraction: Separa o número em sua parte inteira e fracionária.
  - truncate, round, ceiling, floor: Funções de arredondamento.

Classe RealFloat:

- Representa a combinação de números reais e de ponto flutuante.
- É uma subclasse de RealFrac e Floating.
- Métodos:
  - floatRadix: Base do sistema de representação.
  - floatDigits: Número de dígitos na mantissa.
  - floatRange: Exponente mínimo e máximo.

- decodeFloat: Separa o número em sua mantissa e expoente.

A relação entre essas classes é hierárquica. Por exemplo, todos os tipos Integral são Num, e todos os tipos Floating são Fractional. Isso significa que se um tipo é uma instância da classe Floating, ele também deve ser uma instância da classe Fractional, e assim por diante. Essa estrutura permite que Haskell defina operações gerais na classe Num e operações mais específicas em subclasses como Integral ou Floating.

Problema 2

O polimorfismo em Haskell permite a criação de funções e estruturas de dados que podem operar em diversos tipos. O polimorfismo paramétrico permite a definição de funções que são agnósticas ao tipo de dados com o qual estão operando, enquanto o polimorfismo ad-hoc permite a definição de funções que se comportam de maneira diferente dependendo do tipo de dados. As classes de tipos em Haskell fornecem uma forma poderosa de definir funções sobrecarregadas, permitindo a definição de comportamentos específicos do tipo sem comprometer a segurança ou a expressividade do tipo.

Polimorfismo Universal:

- Refere-se à capacidade de escrever código que pode ser aplicado a qualquer tipo de dados.

a. Polimorfismo Paramétrico:

- Descreve a situação em que uma função ou tipo de dados pode operar em qualquer tipo, sem depender da semântica desse tipo.
- Exemplo em Haskell:
  - length :: [a] -> Int
  - length [] = 0
  - length (_:xs) = 1 + length xs
- A função length pode operar em listas de qualquer tipo.

b. Polimorfismo por Inclusão (ou Subtipo):

- Não é intrínseco a Haskell, mas refere-se à capacidade de usar um valor de um subtipo (ou subclasse) em lugar de um valor de um tipo base. Isso é mais comum em linguagens orientadas a objetos.
- Haskell não tem subtipagem no sentido tradicional das linguagens OOP, então não encontramos exemplos diretos disso em Haskell.

Polimorfismo Ad-hoc:

- Refere-se à capacidade de dar múltiplas implementações dependendo do tipo de dados com o qual estamos trabalhando.

a. Polimorfismo de Sobrecarga:

- Em Haskell, isso é realizado através de classes de tipos. Uma função que é definida para vários tipos através de uma classe de tipo é chamada de função sobrecarregada.
- Exemplo em Haskell:
  - class Printable a where
  - toString :: a -> String
  - instance Printable Int where
  - toString x = show x
  - instance Printable Bool where
  - toString x = if x then "true" else "false"
- A função toString é sobrecarregada para tipos Int e Bool.

b. Polimorfismo de Coerção:

- Refere-se à capacidade de converter automaticamente um tipo em outro tipo.
- Em Haskell, isso é menos comum do que em outras linguagens, pois Haskell valoriza a segurança de tipo. No entanto, existem funções explícitas para conversão de tipos, como fromIntegral que converte um número integral em um número mais geral.

Problema 7

O'Haskell foi uma tentativa de combinar características de programação orientada a objetos com o paradigma funcional de Haskell. Ele estendeu Haskell com um modelo de objetos, suporte para subtipagem, herança, encapsulamento e concorrência. No entanto, apesar de seus conceitos interessantes, O'Haskell permaneceu uma extensão experimental e não alcançou adoção generalizada. Haskell padrão continuou a evoluir e incorporou muitas características avançadas ao longo dos anos, embora não exatamente da mesma maneira que O'Haskell propôs.

Diferenças chave entre O'Haskell e Haskell padrão:

- Modelo de Objetos: O'Haskell incorporou um modelo de objetos, permitindo a definição e manipulação de objetos de uma maneira que é típica em linguagens orientadas a objetos.
- Subtipagem: Uma das características centrais da programação orientada a objetos é a subtipagem e o polimorfismo. Em O'Haskell, foi introduzido um mecanismo de subtipagem que permitia que um tipo fosse considerado um subtipo de outro.
- Herança: Outro conceito fundamental da programação orientada a objetos é a herança, onde um tipo (ou classe) pode herdar propriedades e comportamentos de outro tipo. O'Haskell incluiu suporte para herança, permitindo a reutilização e extensão de código.
- Encapsulamento e Modificação de Estado: Enquanto Haskell padrão enfatiza a imutabilidade e funções puras, O'Haskell introduziu maneiras de encapsular e modificar o estado dentro de objetos, de maneira controlada e localizada.
- Threads e Concorrência: O'Haskell também introduziu um modelo de concorrência baseado em threads, permitindo a execução paralela de código.

Problema 8

Em Haskell, enumerações são frequentemente representadas usando tipos de dados algébricos.
Um tipo de dados algébrico que tem vários construtores sem argumentos pode ser pensado como uma enumeração.
O uso de deriving (Show, Eq, Enum, Ord) automaticamente nos dá funções de exibição (Show), comparação de igualdade (Eq), operações de enumeração (Enum), e operações de ordenação (Ord).

Exemplos:

```
- data Day = Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday
-     deriving (Show, Eq, Enum, Ord)
-
- -- Retorna o dia seguinte
- nextDay :: Day -> Day
- nextDay Sunday = Monday  -- Wrap around
- nextDay day    = succ day
-
- isWeekend :: Day -> Bool
- isWeekend day = day == Saturday || day == Sunday
-
```