Trabalho de Estrutura de Linguagens

Linguagem Haskell: História, Classificação e Expressividade

Alunos:
Bernardo Duarte
Daniel José
Vinícius Soares

História de Haskell

- Haskell tem suas principais inspirações centradas em 2 ideias: **Programação Funcional** e **Avaliação Preguiçosa**.
- A Programação Funcional, implementada primeiramente na linguagem **Lisp**, tem sua origem na **Programação Declarativa.**
- A Programação Declarativa é um paradigma de programação focado em descrever logicamente **o que** deve ser feito por um programa sem descrever um **fluxo de controle**.
- A Avaliação Preguiçosa é a ideia de funções que **não avaliam seus parâmetros imediatamente**, e começou por volta dos anos 70.
- Em 1987, em uma conferência sobre Programação Funcional, os presentes decidiram criar uma **linguagem funcional unificada.**
- Então, eles se tornaram o "**Haskell Committee**", e este comitê publicou o documento que determinou a criação da linguagem Haskell em 1º de Abril de 1990.

Classificação e Usos de Haskell

- Haskell é uma linguagem **funcional pura**, utilizando conceitos de **programação declarativa** e de **avaliação preguiçosa** como bases da linguagem.
- Haskell é uma linguagem estritamente **estática.**
- Possui tipagem **forte**.
- É uma linguagem **compilada**.
- Não pode fazer **avaliação** ou **compilação** de código durante o *runtime*.
- Haskell é utilizado, em geral, para tarefas com **grande carga de trabalho**, com uso de **programação paralela**.
- Haskell também é bastante utilizado para o desenvolvimento de **ferramentas**.
- Exemplos:
    - Sistema de filtragem de *Spam* do Facebook
    - Sistema de Suporte de Infraestrutura de TI da Google
    - Bond, o Sistema de Serialização de Dados da Microsoft

Funções de Alta Expressividade: Tipos Algébricos de Dados

- São tipos de dados que podem assumir **diferentes formas** dependendo do **modo como eles são construídos**.
- Podem ter um, nenhum ou múltiplos campos.
- São criados pela palavra chave **data** que determina o nome do tipo, este tendo que começar com letra maiúscula.
- Definido pelos seus construtores que são expressões que definem os tipos que serão utilizados pelos seus campos.
- Possui tanto a funcionalidade de **structs** de C quanto **enums** de C.
- Pode ser usada para criar **genéricos** e para fazer **polimorfismo**.
- Pode ser declarada a partir de **construtores**, por meio de **parâmetros**.
- Há também a possibilidade de serem feitos tipos de definição **recursiva**, como **listas** e **árvores**.
- Com isso, é possível desenvolver uma **Linguagem Específica de Domínio**.

Funções de Alta Expressividade: Tipos Algébricos de Dados - Exemplos

```haskell
type Ponto = (Double, Double)
type Centro = Ponto
type Raio = Double

data FormatoGeometrico
    = Circle Centro Raio
    | Poligono [ Ponto ]
    deriving (Show)
```

+ Ambos apresentam uma legibilidade similar.

- Em haskell, uma vez criado um objeto do tipo algébrico não se pode alterar a sua forma.

- Em C pode-se alterar, porém o comportamento é indefinido.

```c
struct Ponto {
    double x, y;
};
struct Circulo {
    Ponto centro;
    double raio;
};
struct Poligono {
    int numVertices;
    Ponto *vertices;
};
union FormatoGeometrico {
    Circulo circulo;
    Poligono poligono;
};
```

Funções de Alta Expressividade: Genéricos

- Uma variável genérica é uma variável que aceita valores de **quaisquer tipos**, sem restringir a uma definição de tipo específica.
- Em Haskell, tipos de dado genéricos são determinados pelo tipo **data**.
- Em Haskell, tipos genéricos são feitos por meio de **polimorfismo**, pelo uso de **tipos algébricos**.
- O tipo **data** determina um tipo que, por definição, aceita qualquer valor.
- Com a ideia de **tipos algébricos de dados**, é possível também limitar os dados que podem ser atribuídos a uma declaração de tipo genérica.
- As **funções** em Haskell são **naturalmente genéricas.**
- O compilador assume que, caso não haja uma declaração direta de tipo, os atributos são do **tipo mais genérico possível**.

Funções de Alta Expressividade: Genéricos - Exemplos

```haskell
troca (x,y) = (y,x)
val1 = 1
val2 = 2
val3 = 'a'
val4 = [1,2,3]
main :: IO()
main = do let result = (troca(val1,val2))
          putStrLn(show result)
          let result = (troca(val2,val3))
          putStrLn(show result)
          let result = (troca(val3,val4))
          putStrLn(show result)
```

- O código em Haskell é mais **conciso** e tem melhor **redigibilidade**.
- Neste caso, ele também é mais **legível**, pois não é necessário ler várias definições de pares antes de cada chamada de função.
- A lógica também é mais simples de entender, pois é uma lógica **direta**, sem necessidade de **templates** e de **inverter os templates** para garantir que cada elemento de um par pode ser de um tipo diferente.

```cpp
#include <iostream>

template <typename T, typename G>
struct par {
    T um;
    G dois;
};

template <typename T, typename G>
par<G,T> troca (par<T,G> p) {
    par<G,T> pInv;
    pInv.um = p.dois;
    pInv.dois = p.um;
    return pInv;
}

int main(void) {
    par<int,int> par1;
    par1.um = 1; par1.dois = 2;
    par1 = troca(par1);
    std::cout << "(" << par1.um << ", " << par1.dois << ")";
    par<int,char>par2;
    par2.um = 1; par2.dois = 'a';
    par<char,int>par3;
    par3 = troca(par2);
    std::cout << "(" << par3.um << ", " << par3.dois << ")";
    par<char,int*>par4;
    int vet[] = {1,2,3};
    par4.um = 'a'; par4.dois = vet;
    par<int*,char>par5;
    par5 = troca(par4);
    std::cout << "(" << par5.um << ", " << par5.dois << ")";
}
```

Funções de Alta Expressividade: Tipos e Funções Polimórficas

```haskell
data Forma = Triangulo Float Float | Retangulo Float Float | Quadrado Float
area :: Forma -> Float
area (Triangulo x y) = (x*y)/2
area (Quadrado x) = x*x
area (Retangulo x y) = x*y
t = Triangulo 3 5
r = Retangulo 5 10
q = Quadrado 3
main = do let result = (area(t))
          putStrLn(show result)
          let result = (area(r))
          putStrLn(show result)
          let result = (area(q))
          putStrLn(show result)
```

- Polimorfismo é a ideia de criar um tipo "**mais genérico**" que engloba um ou mais tipos "**mais especializados**".
- Em Haskell, tipos polimórficos são feitos por meio do uso do conceito de genéricos e de tipos algébricos de dados.
- É usado o tipo **data** com limitações impostas por definições algébricas para definir o tipo "mais genérico" e os tipos "mais especializados".
- É uma parte intrínseca da definição da linguagem Haskell, e portanto é simples e intuitivo de ser utilizado.
- O código em C++ precisa do uso de **classes** e **herança**.
- O código em C++ é mais **longo** e tem lógica mais **complexa**.
- O código em Haskell é tanto mais **legível** e tem mais **redigibilidade**, além de ser mais **simples** de entender logicamente, com mais **expressividade**.

```cpp
class Forma {
    public:
        virtual float area() = 0;
};
class Triangulo : public Forma {
    public:
        float base;
        float altura;
        Triangulo(float b, float a) {
            base = b;
            altura = a;
        }
        float area() {
            return (base*altura)/2;
        }

};
class Retangulo : public Forma {
    public:
        float compr;
        float altura;
        Retangulo(float c, float a) {
            compr = c;
            altura = a;
        }
        float area() {
            return compr * altura;
        }
};
class Quadrado : public Forma {
    public:
        float lado;
        Quadrado(float l) {
            lado = l;
        }
        float area() {
            return lado * lado;
        }
};
int main(void) {
    Triangulo t(3, 5);
    Retangulo r(5, 10);
    Quadrado q(3);
    Forma* f[3];
    f[0] = &t;
    f[1] = &r;
    f[2] = &q;
    for (int i = 0; i < 3; i++) {
        std::cout << f[i]->area() << std::endl;
    }
}
```

Funções de Alta Expressividade: Avaliação Preguiçosa

```haskell
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

```cpp
int fib(int n)
{
    if (n <= 1)
        return n;

    int table[n + 1];
    table[0] = table[1] = 1;
    for (int i = 2; i <= n; ++i) {
        table[i] = table[i-1] + table[i-2];
    }
    return table[n];
}
```

Lazy Evaluation, também conhecida como **call-by-need**, consiste em apenas calcular o valor de uma expressão quando seu valor realmente é necessário.

- Evita que sejam realizados cálculos desnecessários.
- Torna-se possível, portanto:
  - Definir estruturas infinitas.
  - Definir suas próprias expressões de fluxo de controle.
- Legibilidade do código é prejudicada.
- O custo para se armazenar uma expressão pode superar de maneira considerável o custo para se armazenar o resultado dela.

Funções de Alta Expressividade: Compreensão de Listas

```haskell
quicksort [] = []
quicksort (x:xs) = quicksort [a | a <- xs, a <= x] ++ [x] ++ quicksort [a | a <- xs, a > x]
```

- É um tipo de sintaxe baseada em conjuntos, que permite a criação de listas utilizando outras listas já existentes.
- Fazendo uso da compreensão de listas, o Quick Sort pode ser feito em Haskell usando apenas duas linhas.
- Utilizando esse recurso o código se torna mais fácil de escrever e em alguns casos até mesmo mais fácil de ler.
- A implementação do algoritmo do quicksort em haskell é muito mais próximo de como se explicaria o algoritmo em linguagem natural do que a versão em C.

```c
void quicksort0(int arr[], int a, int b) {
    if (a >= b)
        return;

    int key = arr[a];
    int i = a + 1, j = b;
    while (i < j) {
        while (i < j && arr[j] >= key)
            --j;
        while (i < j && arr[i] <= key)
            ++i;
        if (i < j)
            swap(arr, i, j);
    }
    if (arr[a] > arr[i]) {
        swap(arr, a, i);
        quicksort0(arr, a, i - 1);
        quicksort0(arr, i + 1, b);
    } else {
        quicksort0(arr, a + 1, b);
    }
}
```