Programação Funcional com Haskell

Paulo Torrens

paulotorrens@gnu.org

Departamento de Ciência da Computação
Centro de Ciências e Tecnológias
Universidade do Estado de Santa Catarina

2019/02

Introdução

- A maioria das linguagens de programação populares se enquadram no paradigma imperativo
  - **Em linguagens imperativas, o programador controla o estado da aplicação explicitamente através de comandos**, que podem ter efeitos colaterais arbitrários
  - **Efeitos colaterais são as ações executadas além do retorno da função e que manipulam o estado do programa: alterar a memória, escrever um arquivo, imprimir algo no terminal, etc**
  - Dentre os possíveis comandos também temos atribuições a variáveis e estruturas de fluxo de controle (`while`, `for`, etc), que especificam como executar um algoritmo
  - Exemplos: C, C++, Objective-C, Fortran, Pascal, Java, JavaScript, Python, Swift, Rust, etc

Introdução

- A maioria das linguagens de programação populares se enquadram no paradigma imperativo
  - **Em linguagens imperativas, o programador controla o estado da aplicação explicitamente através de comandos**, que podem ter efeitos colaterais arbitrários
  - **Efeitos colaterais são as ações executadas além do retorno da função e que manipulam o estado do programa: alterar a memória, escrever um arquivo, imprimir algo no terminal, etc**
  - Dentre os possíveis comandos também temos atribuições a variáveis e estruturas de fluxo de controle (`while`, `for`, etc), que especificam como executar um algoritmo
  - Exemplos: C, C++, Objective-C, Fortran, Pascal, Java, JavaScript, Python, Swift, Rust, etc

Introdução

- A maioria das linguagens de programação populares se enquadram no paradigma imperativo
  - **Em linguagens imperativas, o programador controla o estado da aplicação explicitamente através de comandos**, que podem ter efeitos colaterais arbitrários
  - **Efeitos colaterais são as ações executadas além do retorno da função e que manipulam o estado do programa: alterar a memória, escrever um arquivo, imprimir algo no terminal, etc**
  - Dentre os possíveis comandos também temos atribuições a variáveis e estruturas de fluxo de controle (`while`, `for`, etc), que especificam como executar um algoritmo
  - Exemplos: C, C++, Objective-C, Fortran, Pascal, Java, JavaScript, Python, Swift, Rust, etc

Introdução

- A maioria das linguagens de programação populares se enquadram no paradigma imperativo
  - **Em linguagens imperativas, o programador controla o estado da aplicação explicitamente através de comandos**, que podem ter efeitos colaterais arbitrários
  - **Efeitos colaterais são as ações executadas além do retorno da função e que manipulam o estado do programa: alterar a memória, escrever um arquivo, imprimir algo no terminal, etc**
  - Dentre os possíveis comandos também temos atribuições a variáveis e estruturas de fluxo de controle (`while`, `for`, etc), que especificam como executar um algoritmo
  - Exemplos: C, C++, Objective-C, Fortran, Pascal, Java, JavaScript, Python, Swift, Rust, etc

Introdução

- A maioria das linguagens de programação populares se enquadram no paradigma imperativo
  - **Em linguagens imperativas, o programador controla o estado da aplicação explicitamente através de comandos**, que podem ter efeitos colaterais arbitrários
  - **Efeitos colaterais são as ações executadas além do retorno da função e que manipulam o estado do programa: alterar a memória, escrever um arquivo, imprimir algo no terminal, etc**
  - Dentre os possíveis comandos também temos atribuições a variáveis e estruturas de fluxo de controle (`while`, `for`, etc), que especificam como executar um algoritmo
  - Exemplos: C, C++, Objective-C, Fortran, Pascal, Java, JavaScript, Python, Swift, Rust, etc

Introdução

```java
// Java
int my_number = 0;

int getNumber() {
  System.out.println("Getting a number!");
  return my_number++;
}

void showNumber() {
  System.out.println("Got: " + getNumber());
  System.out.println("Got: " + getNumber());
  System.out.println("Got: " + getNumber());
}
```

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

- Em contraste às linguagens imperativas, existem as linguagens que se enquadram no paradigma declarativo
  - **Linguagens declarativas desencorajam ou proíbem o uso de código que manipule explicitamente estado ou com efeitos colaterais**
  - Funções são definidas apenas em respeito aos seus parâmetros de entrada e ao seu resultado
- Dentro das linguagens declarativas, temos o paradigma funcional, onde funções são tratadas como declarações matemáticas
  - **Linguagens funcionais são linguagens declarativas que priorizam a manipulação de funções**
  - **Funções de alta ordem (ou de ordem superior) podem receber outras funções como argumento ou retornar funções como resultado**
  - Exemplos: Common Lisp, Scheme, Standard ML, OCaml, Erlang, Elixir, etc

Introdução

```
1 (* Standard ML *)
2 fun fib 0 = 0
3   | fib 1 = 1
4   | fib n = fib (n - 1) + fib (n - 2)
```

Exercícios de Recursão

1. Uma função que some os números $1 + 2 + 3 + \ldots + n$.

```
int recsum(int n)
```

2. Uma função que encontre o menor elemento de uma lista.

```
int findmin(int list[], int len)
```

3. Uma função que encontre o maior elemento de uma lista.

```
int findmax(int list[], int len)
```

4. Uma função que verifique se uma palavra é um palíndromo.

```
_Bool palin(char word[], int len)
```

Exercícios de Recursão

```c
// C
int recsum(int n) {
  if(n == 0) {
    return 0;
  }
  return n + recsum(n - 1);
}
```

Exercícios de Recursão

```python
# Python
def recsum(n):
    if n == 0:
        return 0
    return n + recsum(n - 1)
```

Exercícios de Recursão

```c
// C
int findmin(int list[], int len) {
  if(len == 1) {
    return list[0];
  }

  int top = list[len - 1];
  int aux = findmin(list, len - 1);
  if(top > aux) {
    return aux;
  } else {
    return top;
  }
}
```

Exercícios de Recursão

```python
# Python
def findmax(items):
  if len(items) == 1:
    return items[0]
  else:
    top = items[0]
    aux = findmax(items[1:])
    if top < aux:
      return aux
    else:
      return top
```

Exercícios de Recursão

```c
// C
_Bool palin(char word[], int len) {
  if(len < 2) {
    return 1;
  }
  if(word[0] != word[len - 1]) {
    return 0;
  }
  return palin(word + 1, len - 2);
}
```

Exercícios de Recursão

```python
# Python
def palin(array):
  if len(array) < 2:
    return True
  if array[0] != array[len(array) - 1]:
    return False
  return palin(array[1:-1])
```

Haskell

"Haskell é uma linguagem de programação puramente funcional. Em linguagens imperativas você faz as coisas dando ao computador uma sequência de tarefas e então as executando. Enquanto você as executa, o estado pode ser alterado. Por exemplo, você seta a variável a para 5 e então faz outras coisas e então a seta para algo diferente. Você tem estruturas de fluxo de controle para fazer algumas ações várias vezes. Em uma linguagem puramente funcional você não diz ao computador o que fazer dessa forma mas ao invés disso você diz o que uma coisa é." [1, tradução livre]

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

Haskell

- Haskell é uma linguagem **puramente funcional**
  - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencoragem efeitos colaterais, porém os permitem
  - Como consequência, estruturas de dados em Haskell são imutáveis (não podem ser alteradas!)
  - Problemas são geralmente resolvidos com recursão em linguagens puramente funcionais
  - Outras linguagens puramente funcionais: Clean, Idris, etc
- Similar ao cálculo lambda, a execução de um programa em Haskell pode ser vista como a reescrita de um termo
  - O programador declara uma sequência de equações representando igualdades, as quais serão reduzidas durante a execução do programa

```haskell
take :: Int -> [a] -> [a]
take 0 xs = [] -- Se o primeiro arg for zero
take n [] = [] -- Se o segundo arg estiver vazio
take n (x:xs) = x : take (n - 1) xs

{- Exemplo de redução:
   take 3 [1, 3, 5, 7, 9] =
       1 : take 2 [3, 5, 7, 9] =
       1 : 3 : take 1 [5, 7, 9] =
       1 : 3 : 5 : take 0 [7, 9] =
       1 : 3 : 5 : [] =
       [1, 3, 5]
-}
```

Haskell

- Programas escritos em uma linguagem puramente funcional como Haskell possuem **transparência referencial**
  - A transparência referencial diz que qualquer expressão em Haskell pode ser trocada por seu valor, sem alterar o resultado
  - Isto é, funções respeitam a definição matemática de igualdade; se $a = b$, então é possível substituir qualquer a por b e vice-versa (se $a = b$, então $b = a$)
  - **Isso só é possível graças à falta de efeitos colaterais**
- **Haskell é uma linguagem não-estrita**, fazendo uso da chamada **avaliação preguiçosa**; parâmetros são executados apenas quando são necessários e se forem necessários
  - Isso permite a definição de estruturas infinitas, e, ao longo que apenas uma porção finita dela seja necessária para a execução, o programa não entrará em um *loop* infinito

- Programas escritos em uma linguagem puramente funcional como Haskell possuem **transparência referencial**
  - A transparência referencial diz que qualquer expressão em Haskell pode ser trocada por seu valor, sem alterar o resultado
  - Isto é, funções respeitam a definição matemática de igualdade; se $a = b$, então é possível substituir qualquer a por b e vice-versa (se $a = b$, então $b = a$)
  - **Isso só é possível graças à falta de efeitos colaterais**
- **Haskell é uma linguagem não-estrita**, fazendo uso da chamada **avaliação preguiçosa**; parâmetros são executados apenas quando são necessários e se forem necessários
  - Isso permite a definição de estruturas infinitas, e, ao longo que apenas uma porção finita dela seja necessária para a execução, o programa não entrará em um *loop* infinito

- Programas escritos em uma linguagem puramente funcional como Haskell possuem **transparência referencial**
  - A transparência referencial diz que qualquer expressão em Haskell pode ser trocada por seu valor, sem alterar o resultado
  - Isto é, funções respeitam a definição matemática de igualdade; se $a = b$, então é possível substituir qualquer a por b e vice-versa (se $a = b$, então $b = a$)
  - **Isso só é possível graças à falta de efeitos colaterais**
- **Haskell é uma linguagem não-estrita**, fazendo uso da chamada **avaliação preguiçosa**; parâmetros são executados apenas quando são necessários e se forem necessários
  - Isso permite a definição de estruturas infinitas, e, ao longo que apenas uma porção finita dela seja necessária para a execução, o programa não entrará em um *loop* infinito

- Programas escritos em uma linguagem puramente funcional como Haskell possuem **transparência referencial**
  - A transparência referencial diz que qualquer expressão em Haskell pode ser trocada por seu valor, sem alterar o resultado
  - Isto é, funções respeitam a definição matemática de igualdade; se $a = b$, então é possível substituir qualquer a por b e vice-versa (se $a = b$, então $b = a$)
  - **Isso só é possível graças à falta de efeitos colaterais**
- **Haskell é uma linguagem não-estrita**, fazendo uso da chamada **avaliação preguiçosa**; parâmetros são executados apenas quando são necessários e se forem necessários
  - Isso permite a definição de estruturas infinitas, e, ao longo que apenas uma porção finita dela seja necessária para a execução, o programa não entrará em um *loop* infinito

- Programas escritos em uma linguagem puramente funcional como Haskell possuem **transparência referencial**
  - A transparência referencial diz que qualquer expressão em Haskell pode ser trocada por seu valor, sem alterar o resultado
  - Isto é, funções respeitam a definição matemática de igualdade; se $a = b$, então é possível substituir qualquer a por b e vice-versa (se $a = b$, então $b = a$)
  - **Isso só é possível graças à falta de efeitos colaterais**
- **Haskell é uma linguagem não-estrita**, fazendo uso da chamada **avaliação preguiçosa**; parâmetros são executados apenas quando são necessários e se forem necessários
  - Isso permite a definição de estruturas infinitas, e, ao longo que apenas uma porção finita dela seja necessária para a execução, o programa não entrará em um *loop* infinito

- Programas escritos em uma linguagem puramente funcional como Haskell possuem **transparência referencial**
  - A transparência referencial diz que qualquer expressão em Haskell pode ser trocada por seu valor, sem alterar o resultado
  - Isto é, funções respeitam a definição matemática de igualdade; se $a = b$, então é possível substituir qualquer a por b e vice-versa (se $a = b$, então $b = a$)
  - **Isso só é possível graças à falta de efeitos colaterais**
- **Haskell é uma linguagem não-estrita**, fazendo uso da chamada **avaliação preguiçosa**; parâmetros são executados apenas quando são necessários e se forem necessários
  - Isso permite a definição de estruturas infinitas, e, ao longo que apenas uma porção finita dela seja necessária para a execução, o programa não entrará em um *loop* infinito

```
1  (* Standard ML, uma linguagem funcional impura *)
2  fun sum a b = (
3      print ("Summing " ^ Int.toString a ^
4                 " and " ^ Int.toString b ^ "\n");
5      a + b);
6
7  val n = let
8      (* Não existe transparência referencial *)
9      val x = sum 10 20
10     (* O compilador não pode trocar a seguinte
11        linha para y = x por causa do efeito! *)
12     val y = sum 10 20
13 in
14     x + y
15 end;
```

```haskell
to_infinity :: Int -> [Int]
to_infinity n = n : to_infinity (n * 2)

{- Se tentarmos imprimir o termo to_infinity 1
   teremos um loop; o termo irá reduzir para uma
   lista infinita [1, 2, 4, 8, 16, ...].

   Porém...
   take 2 (to_infinity 10) =
     take 2 (10 : to_infinity 20) =
     10 : take 1 (to_infinity 20) =
     10 : take 1 (20 : to_infinity 40) =
     10 : 20 : take 0 (to_infinity 40) =
     10 : 20 : [] =
     [10, 20]
-}
```

Bibliografia

Lipovaca, M.
Learn You a Haskell for Great Good!: A Beginner's Guide, 1st ed.
No Starch Press, San Francisco, CA, USA, 2011.