

Programação Funcional com Haskell

Paulo Torrens

paulotorrens@gnu.org

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

2019/02

- 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

- 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

- 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

- 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

- 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

```
1 // Java
2 int my_number = 0;
3
4 int getNumber() {
5     System.out.println("Getting a number!");
6     return my_number++;
7 }
8
9 void showNumber() {
10     System.out.println("Got: " + getNumber());
11     System.out.println("Got: " + getNumber());
12     System.out.println("Got: " + getNumber());
13 }
```

- 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

- 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

- 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

- 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

- 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

- 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

- 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

```
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 $1 + 2 + 3 + \dots + n$.

```
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)
```



```
1 // C
2 int recsum(int n) {
3     if(n == 0) {
4         return 0;
5     }
6     return n + recsum(n - 1);
7 }
```

```
1 # Python
2 def recsum(n):
3     if n == 0:
4         return 0
5     return n + recsum(n - 1)
```

Exercícios de Recursão

```
1 // C
2 int findmin(int list[], int len) {
3     if(len == 1) {
4         return list[0];
5     }
6
7     int top = list[len - 1];
8     int aux = findmin(list, len - 1);
9     if(top > aux) {
10         return aux;
11     } else {
12         return top;
13     }
14 }
```

```
1 # Python
2 def findmax(items):
3     if len(items) == 1:
4         return items[0]
5     else:
6         top = items[0]
7         aux = findmax(items[1:])
8         if top < aux:
9             return aux
10        else:
11            return top
```

```
1 // C
2 _Bool palin(char word[], int len) {
3     if(len < 2) {
4         return 1;
5     }
6     if(word[0] != word[len - 1]) {
7         return 0;
8     }
9     return palin(word + 1, len - 2);
10 }
```

```
1 # Python
2 def palin(array):
3     if len(array) < 2:
4         return True
5     if array[0] != array[len(array) - 1]:
6         return False
7     return palin(array[1:-1])
```

“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 é uma linguagem **puramente funcional**
 - Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais; linguagens funcionais impuras desencorajam 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 é uma linguagem **puramente funcional**
 - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencorajam 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 é uma linguagem **puramente funcional**
 - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencorajam 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 é uma linguagem **puramente funcional**
 - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencorajam 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 é uma linguagem **puramente funcional**
 - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencorajam 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 é uma linguagem **puramente funcional**
 - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencorajam 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 é uma linguagem **puramente funcional**
 - **Chamamos uma linguagem de puramente funcional quando ela não permite efeitos colaterais**; linguagens funcionais impuras desencorajam 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

```
1 take :: Int -> [a] -> [a]
2 take 0 xs = [] -- Se o primeiro arg for zero
3 take n [] = [] -- Se o segundo arg estiver vazio
4 take n (x:xs) = x : take (n - 1) xs
5
6 {- Exemplo de redução:
7     take 3 [1, 3, 5, 7, 9] =
8         1 : take 2 [3, 5, 7, 9] =
9         1 : 3 : take 1 [5, 7, 9] =
10        1 : 3 : 5 : take 0 [7, 9] =
11        1 : 3 : 5 : [] =
12        [1, 3, 5]
13 -}
```

- 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;
```

```
1 to_infinity :: Int -> [Int]
2 to_infinity n = n : to_infinity (n * 2)
3
4 {- Se tentarmos imprimir o termo to_infinity 1
5     teremos um loop; o termo irá reduzir para uma
6     lista infinita [1, 2, 4, 8, 16, ...].
7
8     Porém...
9     take 2 (to_infinity 10) =
10         take 2 (10 : to_infinity 20) =
11         10 : take 1 (to_infinity 20) =
12         10 : take 1 (20 : to_infinity 40) =
13         10 : 20 : take 0 (to_infinity 40) =
14         10 : 20 : [] =
15         [10, 20]
16 -}
```

Lipovaca, M.

Learn You a Haskell for Great Good!: A Beginner's Guide, 1st ed.

No Starch Press, San Francisco, CA, USA, 2011.

