

Arquivos de código e tipos

Felipe A. Lessa

Departamento de Ciência da Computação
Universidade de Brasília

Assuntos de hoje

1

Arquivos

- O básico
- Mais funções

2

Tipos

- Descobrimdo
- Funções
- Assinaturas e inferência

3

Finalizando

- Palavra-final

Frases do dia

*Para quem só tem martelo,
tudo parece prego.*

Assuntos de hoje

1

Arquivos

- O básico
- Mais funções

2

Tipos

- Descobrimdo
- Funções
- Assinaturas e inferência

3

Finalizando

- Palavra-final

Criando arquivos

- Nós precisamos salvar nossas funções para uso posterior.
- Arquivos contendo código-fonte Haskell possuem a extensão `.hs` e são bem úteis.
- Como exemplo, crie o arquivo `Teste.hs` com um editor de texto (e.g. gedit, emacs, ...) contendo

```
1 area r = pi * r^2
```

- Sim, `pi` já está definido por padrão =).

Arquivos no interpretador

- Entre no interpretador e use o comando `:load` para carregar o arquivo que você acabou de criar:

```
1 Prelude> :l Teste
2 [1 of 1] Compiling Main (Teste.hs, interpreted)
3 Ok, modules loaded: Main.
4 *Main>
```

- Agora você pode usar o que você definiu!

```
4 *Main> area 5
5 78.53981633974483
```

- Após modificar o arquivo, você pode recarregá-lo usando `:reload` ou apenas `:r`.

Sem `let`

- Se estiveram atentos, no último slide não usamos `let`.
- Nós usamos `let` para muitas coisas, mas não nas definições top-level em arquivos.
- Errado:

```
1 let area r = pi * r^2
```

- Correto

```
1 area r = pi * r^2
```

- Já veremos onde usa-se `let`.

Múltiplas declarações

- Para várias declarações, apenas escreva-as:

```
1 areaCirc r = pi * r^2
2 areaTri b h = (b * h) / 2
```

- A ordem não faz diferença:

```
3 areaQuad l = areaRect l l
4 areaRect b h = b * h
```

- Não podemos ter múltiplas definições!

```
5 r = 5.0
6 r = 25.0 -- Errado!
```


Assuntos de hoje

1

Arquivos

- O básico
- Mais funções

2

Tipos

- Descobrimdo
- Funções
- Assinaturas e inferência

3

Finalizando

- Palavra-final

Expressões `if`

- Haskell possui `if ... then ... else ...` como a maioria das linguagens de programação.
- Podemos definir uma função que retorna -1 , 0 ou 1 dependendo do sinal do seu argumento (essa função já existe e se chama `signum`):

```
1 sinal x =  
2     if x < 0  
3     then -1  
4     else if x > 0  
5     then 1  
6     else 0
```

- Usando:

```
1 *Main> sinal (-1)  
2 -1
```

Expressões `if` II

- Uma peculiaridade: o **else** é obrigatório.
- Por exemplo, em **if** `cond` **then** `a` **else** `b`,
 - Primeiro é verificado se `cond` é **True** ou **False** (`cond` obrigatoriamente é um booleano).
 - Se `cond` é **True**, o resultado da expressão é `a`.
 - Caso contrário, o resultado da expressão é `b`.
- Observe que falamos de “resultado da expressão”. Qual seria tal resultado se não houvesse **else**? =)

Expressões `case`

- Também possuímos expressões `case`.
- Elas são bem poderosas, trabalhando em qualquer tipo de dados e em qualquer profundidade (depois =).
- Curiosidade: o GHC transforma todas as operações que precisam verificar o valor de um dado em `cases`.
- Como Haskell é preguiçosa, algo é computado apenas quando seu valor é necessário. Como as operações que precisam dos valores se reduzem todas a `cases`, diz-se que apenas um `case` pode realizar algum cálculo. =]

Expressões `case` II

- Um exemplo com pouco sentido:

```
1 f x = case x of
2       0 → 1
3       1 → 5
4       2 → 2
5       _ → -1
```

- A indentação é importante!
- Os casos são verificados na ordem definida.
- O `_` é chamado “wildcard” e pode ser qualquer coisa. Uma variável como `z` poderia ser usada, mas com o wildcard você deixa claro que não precisa daquele valor.

Expressões case III

- Um exemplo um pouco melhor:

```
1 sinal' x =  
2   case compare x 0 of  
3     LT → -1  
4     EQ → 0  
5     GT → 1
```

- Verifiquem se `sinal x == sinal' x`.
- Aliás, como ficaria `f x = if c x then a else b` usando apenas `case`?

Um pouco sobre indentação

- Você pode usar a “regra do layout” ou ponto-e-vírgulas explícitos. O primeiro é preferível na maioria dos casos.
- Exemplo sem layout:

```

1 sinal' x = case compare x 0 of {LT → -1; EQ →
    0; GT → 1}
2 f x = case x of { 0 → 1 ;
3     1 → 5 ; 2 → 2
4     ; _ → -1 }
```

- Uma forma simples de usar o layout é indentar um código sempre mais do que a linha que contém o início da expressão onde ele está.
- Em geral a regra faz sentido e você não se preocupa =).
- O quanto você indenta não importa, desde que idente!

Definições por partes

- O **case** pode se disfarçar numa definição:

$$1 \quad f \ 0 = 1$$

$$2 \quad f \ 1 = 5$$

$$3 \quad f \ 2 = 2$$

$$4 \quad f \ _ = -1$$

- As mesmas regras do **case** valem aqui.
Essa definição é *totalmente equivalente* à anterior.
- Definições assim são extremamente comuns,
principalmente por serem fáceis de ler.

Composição

- Na matemática, a função $h(x) = f(g(x))$ é chamada a composição de f com g e escrevemos $h = f \circ g$.
Ou seja, $f(g(x)) = (f \circ g)(x)$.
- Em Haskell, podemos fazer o mesmo com a função $(.)$.
Basicamente, $f \circ g$ é o mesmo que $(f . g)$.
- Por que iríamos querer isso? Para acabar com parênteses ou com argumentos, e assim aumentar a clareza (usem o bom senso!).

Composição II

- Exemplo:

```
1 quadrado x = x ^ 2
```

```
2 mais_um x = x + 1
```

```
4 f1 x = quadrado (mais_um x)
```

```
5 f2 x = (quadrado . mais_um) x
```

```
6 f3 = quadrado . mais_um
```

- Ok, essa foi fácil. Mas acreditem, é bem útil hehe.

let em funções

- Usamos **let** para criar variáveis locais.
- Exemplo:

```
1 raizes a b c =  
2   ((-b + sqrt (b*b - 4*a*c)) / (2*a),  
3   (-b - sqrt (b*b - 4*a*c)) / (2*a))
```

- Urgh, quanta repetição. Bem melhor:

```
1 raizes a b c =  
2   let disc = sqrt (b*b - 4*a*c)  
3       dois_a = 2*a  
4   in ((-b + disc) / dois_a,  
5       (-b - disc) / dois_a)
```

- O valor das variáveis só é calculado se necessário!

Operadores

- Operadores são funções também. As diferenças são:
 - Eles possuem apenas símbolos no nome.
 - Por padrão são usados na forma infixada.
- Uma função pode ser usada na forma infixada usando `

```
1 Prelude> 10 `compare` 5
2 GT
```

- Operadores podem ser usados na forma prefixada com parênteses:

```
1 Prelude> (>) 10 5
2 True
```

Operadores II

- Pode-se definir operadores de qualquer forma:

```
1 Prelude> let a # b = (a + b) / 2
2 Prelude> let (?) a b = 2 / (recip a + recip b)
3 Prelude> (#) 3 4
4 3.5
5 Prelude> 3 ? 4
6 3.428571428571429
```

- Funções também!

```
1 Prelude> let a `mais2` b = a + b + b
2 Prelude> mais2 4 5
3 14
```

Exercício

- O código de `raizes` não funciona se não há raízes.
- Faça uma função que retorne uma lista de raízes únicas.
- Não calcule os mesmos valores duas vezes!
- Utilize algo ``compare`` 0 como fizemos antes.

Assuntos de hoje

1 Arquivos

- O básico
- Mais funções

2 Tipos

- Descobrimos
- Funções
- Assinaturas e inferência

3 Finalizando

- Palavra-final

O que são tipos?

- Nós trabalhamos com dados.
- Cada dado possui uma representação e um conjunto de operações que podem ser aplicadas a ele.
- Um *tipo* basicamente representa um conjunto de dados.
- Nós gostamos de tipos porque eles nos asseguram que tudo o que estamos fazendo está de acordo com o conjunto onde estamos.
- “Se $n \in \mathbb{N}$, como $n = \frac{7}{4}$?” Uma contradição, ou seja, um erro de compilação =).

Explorando

- Dentro do GHCi usamos `:type` para saber o tipo de uma expressão qualquer:

```
1 Prelude> :t 'H'  
2 'H' :: Char
```

- Lemos a resposta do GHCi como “`' H'` é do tipo **Char**”.
- Outro teste:

```
1 Prelude> :t "Hello_World"  
2 "Hello_World" :: [Char]
```

- Usamos `[a]` para descrever o tipo composto por listas com elementos do tipo `a`.

Explorando II

- Tipos booleanos:

```
1 Prelude> :t True
```

```
2 True :: Bool
```

```
4 Prelude> :t False
```

```
5 False :: Bool
```

- Tuplas:

```
1 Prelude> :t ('a', True)
```

```
2 ('a', True) :: (Char, Bool)
```

```
4 Prelude> :t ('a', True, "oi")
```

```
5 ('a', True, "oi") :: (Char, Bool, [Char])
```

Explorando III

- Números:

```
1 Prelude> :t 5
2 5 :: (Num t) => t
```

```
4 Prelude> :t pi
5 pi :: (Floating a) => a
```

- Números usam classes (“type classes”), veremos isso outro dia hehe.

Exercício 12

- Dê o tipo de
 - 1 `[('s', 'b')]`
 - 2 `(['s'], ['s'])`
 - 3 `(['a', 'b'], 'ab')`
- Verifique com o GHCi.

Arquivos

- O básico
- Mais funções

Tipos

- Descobrimos
- **Funções**
- Assinaturas e inferência

Finalizando

- Palavra-final

Setas

- Funções também possuem tipos (e também são dados).
- Usamos uma seta (\rightarrow) para criar tipos funcionais:

```
1 Prelude> :t not  
2 not :: Bool → Bool
```

Setas II

- Por exemplo, **unwords** e **unlines** “pegam uma lista de strings e retornam uma string”. A primeira concatena as strings da lista com espaços, e a segunda com ‘\n’:

```
1 Prelude> unwords ["CIC", "MAT", "FIS"]  
2 "CIC_MAT_FIS"  
3 Prelude> unlines ["CIC", "MAT", "FIS"]  
4 "CIC\nMAT\nFIS\n"
```

- Não precisamos nem olhar para saber que o tipo de ambas é **[String]** → **String**. (**String** é um sinônimo para **[Char]**)

Múltiplos argumentos

- Funções com vários argumentos simplesmente recebem várias setas no seu tipo:

```
1 Prelude> let nand a b = not (a && b)
2 Prelude> :t nand
3 nand :: Bool → Bool → Bool
```

- Note que setas têm associatividade à direita:

```
1 Prelude> let f g = g (g True)
2 Prelude> :t f
3 f :: (Bool → Bool) → Bool
4 Prelude> f not
5 True
```


Currying

- Continuando com a associatividade: então podemos dizer que `nand :: Bool → (Bool → Bool)`.
- Opa! Quer dizer que `nand` retorna uma função?

```
1 Prelude> nand True True
2 False
3 Prelude> (nand True) True
4 False
5 Prelude> :t (nand True)
6 (nand True) :: Bool → Bool
```

Currying II

- O que acabamos de ver se chama “currying” e permite aplicação a parcial de argumentos.
- Por exemplo, não precisamos de $g\ x = \text{nand True } x$:

```
1 Prelude> f (nand True)
2 True
```

- Aliás, é isso que nos permite transformar $h\ x = (g\ .\ f)\ x$ em apenas $h = g\ .\ f$, assim como $g\ x = \text{nand True } x$ é equivalente a $g = \text{nand True}$.

Tipos polimórficos

- Muitas vezes uma função precisará de operações disponíveis não apenas para um, e sim para vários tipos.
- Por exemplo, a função **length**:

```
1 Prelude> length [1,2,3]
```

```
2 3
```

```
3 Prelude> length "Felipe"
```

```
4 6
```

- Essa função requer como argumento uma lista, mas não se importa nem um pouco com seus elementos.
- Como expressar isso no seu tipo?

Tipos polimórficos II

- Usamos variáveis (de tipo) para expressar polimorfismo:

```
1 Prelude> :t length  
2 length :: [a] → Int
```

- Aqui, *a* pode assumir qualquer tipo (para restringir a certos grupos de tipos — nem só um, nem todos, e sim alguns — usamos classes).

- Mais alguns exemplos:

```
1 const      :: a → b → a  
2 snd        :: (a, b) → b  
3 reverse    :: [a] → [a]  
4 all        :: (a → Bool) → [a] → Bool
```

Arquivos

- O básico
- Mais funções

Tipos

- Descobrimos
- Funções
- Assinaturas e inferência

Finalizando

- Palavra-final

Assinando seu nome

- A assinatura de um dado (função ou variável) é composta pelo seu nome e seu tipo.
- O Haskell pode inferir os tipos para você, mas você também pode escrever os tipos se quiser.
- É comum escrever os tipos de funções top-level:
 - O tipo é uma forma de documentação.
 - O Haddock não tem inferência de tipos.
 - Você não precisa adivinhar se o compilador inferiu o tipo certo (“certo” nesse caso significa “o que você pediu”, pois o compilador não infere tipos errados).
 - Se quiser, pode escrever o seu código primeiro pelos tipos e depois o corpo das funções.

Assinando seu nome II

```
1  -- Aqui restringimos o tipo a Double apenas,  
2  -- e nao ao tipo polimorfico com classes  
3  -- que seria inferido. Isso pode aumentar  
4  -- a performance porque o compilador gera  
5  -- instrucoes de maquina de Double apenas.
```

```
6  quadrado      :: Double → Double
```

```
7  quadrado x    = x * x
```

```
9  -- Ops! O compilador vai pegar estaticamente  
10 -- o erro que cometemos abaixo! =)
```

```
11 first         :: (a, b) → a
```

```
12 first (y,x)   = x
```

Inferência

- Inferir um tipo é descobri-lo através do seu contexto.
- O compilador faz isso sempre. No slide anterior, nós usamos a função $(*)$, mas qual seu tipo?

```
1 quadrado      :: Double → Double
```

```
2 quadrado x    = x * x
```

- O compilador diz “Bom, eu não sei qual o tipo de $(*)$. Mas eu sei que `quadrado :: Double → Double` então `x :: Double`. Como aparece no código `x * x`, então `(*) :: Double → Double → Double`.”

Inferência II

- Todos os tipos podem ser inferidos sem nenhuma assinatura.
- Todos os tipos são sempre verificados estaticamente.
- O compilador sempre infere o tipo mais genérico possível (fora exceções devido à restrição de monomorfismo).
- Não seremos formais aqui, mesmo porque não vamos nos preocupar com inferência de tipos enquanto não falarmos de funções de 2ª ordem ou maior (que é um recurso que *não* faz parte do Haskell 98).

Assuntos de hoje

1 Arquivos

- O básico
- Mais funções

2 Tipos

- Descobrimos
- Funções
- Assinaturas e inferência

3 Finalizando

- Palavra-final

Dever de casa

- Vocês já têm boa parte do conhecimento básico.
- Isso lhes dá condições de começar a ler a documentação do **Prelude**. Dêem uma olhada nas funções disponíveis e tentem usá-las com alguns argumentos no GHCi.
- Sim, o **Prelude** é bem grande. Na semana que vem reservaremos um tempo para dar uma olhada nele, mas é bom descobrirem por si próprios o que existe =).
- Documentação do GHC:

<http://haskell.org/ghc/docs/latest/html/>