Arquivos de código e tipos

Felipe A. Lessa

Departamento de Ciência da Computação
Universidade de Brasília

Assuntos de hoje

Frase do dia

Para quem só tem martelo,
tudo parece prego.

Assuntos de hoje

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

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 **`case`**s.
- Como Haskell é preguiçosa, algo é computado apenas quando seu valor é necessário. Como as operações que precisam dos valores se reduzem todas a **`case`**s, diz-se que apenas um **`case`** pode realizar algum cálculo. =]

Expressões `case` II

- Um exemplo com pouco sentido:

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

- A identação 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:

```
sinal' x =
  case compare x 0 of
    LT → -1
    EQ → 0
    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 identação

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

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

- Uma forma simples de usar o layout é identar 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 identa nao importa, desde que idente!

Definições por partes

- O `case` pode se disfarçar numa definição:

```
f 0 = 1
f 1 = 5
f 2 = 2
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(g(x))` é o mesmo que `(f . g) x`.
- 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:

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

- Funções também!

```
Prelude> let a `mais2` b = a + b + b
Prelude> mais2 4 5
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

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:

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

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

```
Prelude> :t "Hello World"
"Hello World" :: [Char]
```

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

Explorando II

- Tipos booleanos:

```
Prelude> :t True
True :: Bool

Prelude> :t False
False :: Bool
```

- Tuplas:

```
Prelude> :t ('a', True)
('a', True) :: (Char, Bool)

Prelude> :t ('a', True, "oi")
('a', True, "oi") :: (Char, Bool, [Char])
```

Explorando III

- Números:

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

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

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

Exercício izi

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

Assuntos de hoje

Setas

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

```
Prelude> :t not
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'`:

```
Prelude> unwords ["CIC", "MAT", "FIS"]
"CIC MAT FIS"
Prelude> unlines ["CIC", "MAT", "FIS"]
"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:

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

- Note que setas têm associatividade à direita:

```
Prelude> let f g = g (g True)
Prelude> :t f
f :: (Bool → Bool) → Bool
Prelude> f not
True
```

Currying

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

```
Prelude> nand True True
False
Prelude> (nand True) True
False
Prelude> :t (nand True)
(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 = nand True x`:

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

- Aliás, é isso que nos permite transformar `h x = (g . f) x` em apenas `h = g . f`, assim como `g x = nand True x` é equivalente a `g = 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`**:

```
Prelude> length [1,2,3]
3
Prelude> length "Felipe"
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:

```
Prelude> :t length
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:

```
const   :: a → b → a
snd     :: (a, b) → b
reverse :: [a] → [a]
all     :: (a → Bool) → [a] → Bool
```

Assuntos de hoje

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

```haskell
-- Aqui restringimos o tipo a Double apenas,
-- e nao ao tipo polimorfico com classes
-- que seria inferido. Isso pode aumentar
-- a performance porque o compilador gera
-- instrucoes de maquina de Double apenas.
quadrado    :: Double → Double
quadrado x  = x * x

-- Ops! O compilador vai pegar estaticamente
-- o erro que cometemos abaixo! =)
first       :: (a, b) → a
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?

```haskell
quadrado    :: Double → Double
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

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/`