Haskell

Haskell is a general purpose, purely functional programming language incorporating many recent innovations in programming language design. Haskell provides higher-order functions, non-strict semantics, static polymorphic typing, user-defined algebraic datatypes, pattern-matching, list comprehensions, a module system, a monadic I/O system, and a rich set of primitive datatypes, including lists, arrays, arbitrary and fixed precision integers, and floating-point numbers. Haskell is both the culmination and solidification of many years of research on lazy functional languages.

(The Haskell 98 Report)

Maria João Frade
DI - Universidade do Minho
2004/05

Valores & Expressões

Os **valores** são as entidades básicas da linguagem Haskell. São os elementos atómicos.

As **expressões** são obtidas aplicando funções a valores ou a outras expressões.

O interpretador Haskell actua como uma calculadora *(read--evaluate--print loop)*:

lê uma expressão, calcula o seu valor e mostra o resultado.

Exemplos:

```
>  5
5

>  3.5  +  6.7
10.2

>  2 < 35
True

>  not True
False

>  not ((3.5+6.7) > 23)
True
```


Tipos

Os tipos servem para **classificar** entidades (de acordo con as suas caracteristicas).

Em Haskell *toda a expressão tem um **tipo**.*

`e :: T` significa que a expressão e *tem* tipo T
é do tipo

Exemplos:

```
58      :: Int
'a'     :: Char
[3,5,7] :: [Int]
(8,'b') :: (Int,Char)
```

Inteiro
Caracter
Lista de inteiros
Par com um inteiro e um caracter

Em Haskell, a verificação de tipos é feita durante a compilação.

O Haskell é uma linguagem fortemente tipada, com um sistema de tipos muito evoluído (como veremos).

Tipos Básicos

Bool	Boleanos	`True, False`
Char	Caracteres	`'a', 'b', 'A', '1', '\n', ...`
Int	Inteiros de tamanho limitado	`1, -3, 234345, ...`
Integer	Inteiros de tamanho ilimitado	`2, -7, 75756850013434682, ...`
Float	Números de vírgula flutuante	`3.5, -6.53422, 51.2E7, 3e4, ...`
Double	Núm. vírg. flut. de dupla precisão	`3.5, -6.5342, 51.2E7, ...`
()	*Unit*	`()` é o seu único elemento

Tipos Compostos

Produtos Cartesianos `(T1,T2, ...,Tn)`

`(T1,T2,...,Tn)` Tipo dos tuplos com o 1º elemento do tipo `T1`, 2º elemento do tipo `T2`, etc.

Exemplos:
```
(1,5)         :: (Int,Int)
('a',6,True) :: (Char,Int,Bool)
```

Listas `[T]`

`[T]` Tipo da listas cujos elementos são *todos* do tipo `T`.

Exemplos:
```
[2,5,6,8]         :: [Integer]
['h','a','s']     :: [Char]
[3.5,86.343,1.2] :: [Float]
```

Funções `T1 -> T2`

`T1 -> T2` Tipo das funções que *recebem* valores do tipo `T1` e *devolvem* valores do tipo `T2`.

Exemplos:
```
not :: Bool -> Bool
ord :: Char -> Int
```


Funções

A operação mais importantes das funções é a sua **aplicação**.

Se `f :: T1 -> T2` e `a :: T1` então `f a :: T2`

Exemplos:
```
>  not True
False :: Bool

>  ord 'a'
97 ::Int

>  ord 'A'
65 :: Int

>  chr 97
'a' :: Char
```

Preservação de tipos

O tipo das expressão é preservado ao longo do processo de cálculo.

Qual será o tipo de `chr` ?

Novas definições de funções deverão que ser escritas num ficheiro, que depois será carregado no interpretador.

Definições

Uma definição associa um nome a uma expressão. *nome = expressão*

nome tem que ser uma palavra começada por letra minúscula.

A definição de funções pode ainda ser feita por um conjunto de **equações** da forma:

nome arg1 arg2 ... argn = expressão

Quando se define uma função podemos incluir *informação sobre o seu tipo.* No entanto, essa informação não é obrigatória.

Exemplos:
```
pi = 3.1415

areaCirc x = pi * x * x

areaQuad = \x-> x*x

areaTri b a = (b*a)/2

volCubo :: Float -> Float
volCubo y = y * y * y
```

O tipo de cada função é inferido automáticamente.

Exemplo:

Para a função g assim definida:
```
g x = not (65 > ord x)
```

O tipo inferido é
```
g :: Char -> Bool
```

Porquê ?

Mas, há funções às quais é possível associar *mais do que um* tipo concreto.

Exemplos:
```
id x = x

nl y = '\n'
```

Qual será o tipo destas funções ?

O problema é resolvido recorrendo a **variáveis de tipo**.

Uma variável de tipo representa um tipo qualquer.

```
id :: a -> a

nl :: a -> Char
```

Em Haskell:

As variáveis de tipo representam-se por nomes começados por letras minúsculas (normalmente `a`, `b`, `c`, ...).

Os tipos concretos usam nomes começados por letras maiúsculas (ex: `Bool`, `Int`, ...).

Quando as funções são usadas, as variáveis de tipos são substituídas pelos tipos concretos adquados.

Exemplos:
```
id True
id 'a'
nl False
nl (volCubo 3.2)
```

```
id :: Bool -> Bool
id :: Char ->Char
nl :: Bool -> Char
nl :: Float -> Char
```


Funções cujos tipos têm variáveis de tipo são chamadas **funções polimórficas**.

Um tipo pode conter diferentes variáveis de tipo.

Exemplo:
```
fst (x,y) = x

fst :: (a,b) -> a
```

Inferência de tipos
O tipo de cada função é inferido automáticamente.
O Haskell infere o *tipo mais geral* de qualquer expressão.

É possivel associar a uma função um tipo *mais especifico* do que o tipo inferido automáticamente.

Exemplo:
```
seg :: (Bool,Int) -> Int
seg (x,y) = y
```


O Haskell tem um enorme conjunto de definições (que está no módulo Prelude) que é carregado por defeito e que constitui a base da linguagem Haskell.

Alguns operadores:

Lógicos: **&&** (e), || (ou), **not** (negação)

Numéricos: +, -, *, / (divisão de reais), ^ (exponenciação com inteiros), **div** (divisão inteira), **mod** (resto da divisão inteira), ** (exponenciações com reais), **log**, **sin**, **cos**, **tan**, ...

Relacionais: == (igualdade), /= (desigualdade), <, <=, >, >=

Condicional: **if** ... **then** ... **else** ...

:: Bool **:: a**

Exemplo:
```
> if (3>=5) then [1,2,3] else [3,4]
[3,4]
> if (ord 'A' == 65) then 2 else 3
2
```

As funções `test` e `test'` são muito parecidas mas há uma diferença essencial:

Têm tipos diferentes !

```
test (x,y) = [ not x, y||x, x && y ]

test' x y = [ not x, y||x, x && y ]
```

A função `test` recebe **um único** argumento (que é um par de booleanos) e devolve uma lista de booleanos.

```
test :: (Bool,Bool) -> [Bool]

> test (True,False)
```

A função `test'` recebe **dois** argumentos, cada um do tipo `Bool`, e devolve uma lista de booleanos.

```
test' :: Bool -> Bool -> [Bool]

> test' True False
```

A função `test'` recebe um valor de cada vez. Realmente, o seu tipo é:

```
test' :: Bool -> (Bool -> [Bool])

> (test' True) False
```

Mas os parentesis podem ser dispensados !

O tipo função associa à direita e a aplicação de funções é associativa à esquerda.

Isto é, `f :: T1 -> T2 -> ... -> Tn -> T`

é uma forma abreviada de escrever

```
f :: T1 -> (T2 -> (... -> (Tn -> T)...))
```

e `f x1 x2 ... xn`

é uma forma abreviada de escrever

```
(...((f x1) x2) ...) xn
```


Equações e Funções

Uma função pode ser definida por equações que relacionam os seus argumentos com o resultado pretendido.

Exemplos:

```
triplo x = 3 * x
dobro y = y + y
perimCirc r = 2*pi*r
perimTri x y z = x+y+z
minimo x y = if x>y them y else x
```

As equações definem regras de cálculo para as funções que estão a ser definidas.

nome *arg1 arg2 ... argn* = expressão

Nome da função (iniciada por letra minúscula).

Argumentos da função. Cada argumento é um **padrão**. *(cada variável não pode ocorrer mais do que uma vez)*

O tipo da função é inferido tendo por base que ambos os lados da equação têm que ter o mesmo tipo.

Padrões *(patterns)*

Um **padrão** é uma variável, uma constante, ou um "esquema" de um valor atómico *(isto é, o resultado de aplicar construtores básicos dos valores a outros padrões).*

No Haskell, um padrão **não** pode ter variáveis repetidas *(padrões lineares).*

Exemplos:

Padrões	Tipos
`x`	`a`
`True`	`Bool`
`4`	`Int`
`(x,y,(True,b))`	`(a,b,(Bool,c))`
`('A',False,x)`	`(Char,Bool,a)`
`[x,'a',y]`	`[Char]`

Não padrões

```
[x,'a',1]

(4*6, y)
```

Porquê ?

Quando não nos interessa dar nome a uma variável, podemos usar _ que representa uma variável anónima nova.

Exemplos:

```
snd (_,x) = x

segundo (_,y,_) = y
```

Exemplos:

```
soma :: (Int,Int) -> Int -> (Int,Int)
soma (x,y) z = (x+z, y+z)
```

outro modo seria

```
soma w z = ((fst w)+z, (snd w)+z)
```

Qual é mais legível ?

```
exemplo :: (Bool,Float) -> ((Float,Int), Float) -> Float
exemplo (True,y) ((x,_),w) = y*x + w
exemplo (False,y) _ = y
```

em alternativa, poderiamos ter

```
exemplo a b = if (fst a) then (snd a)*(fst (fst b)) + (snd b)
                         else (snd a)
```


Redução

O cálculo do valor de uma expressão é feito usando as equações que definem as funções como regras de cálculo.

Uma **redução** é um passo do processo de cálculo (é usual usar o símbolo ⇒ denotar esse passo)

Cada redução resulta de substituir a *instância* do lado esquerdo da equação (o redex) pelo respectivo lado direito (o contractum).

Exemplos:

`triplo 7` ⇒ `3*7` ⇒ `21`

A instância de `(triplo x)` resulta da *substituição* `[7/x]`.

`snd (9,8)` ⇒ `8`

A instância de `snd (_,x)` resulta da *substituição* `[9/_,8/x]`.

A expressão `dobro (triplo (snd (9,8)))` pode reduzir de três formas distintas:

`dobro (triplo (snd (9,8)))` ⇒ `dobro (triplo 8)`

`dobro (triplo (snd (9,8)))` ⇒ `dobro (3*(snd (9,8)))`

`dobro (triplo (snd (9,8)))` ⇒ `(triplo (snd (9,8)))+(triplo (snd (9,8)))`

A estratégia de redução usada para o cálculo das expressões é uma característica essencial de uma linguagem funcional.

O **Haskell** usa a estratégia ***lazy evaluation*** *(call-by-name)*, que se caracteriza por escolher para reduzir sempre o redex mais externo. Se houver vários redexes ao mesmo nível escolhe o redex mais à esquerda *(outermost; leftmost)*.

Uma outra estratégia de redução conhecida é a ***eager evaluation*** *(call-by-value)*, que se caracteriza por escolher para reduzir sempre o redex mais interno. Se houver vários redexes ao mesmo nível escolhe o redex mais à esquerda *(innermost; leftmost)*.

Lazy Evaluation *(call-by-name)*

```
dobro (triplo (snd (9,8))) ⇒ (triplo (snd (9,8)))+(triplo (snd (9,8)))
                           ⇒ (3*(snd (9,8))) + (triplo (snd (9,8)))
                           ⇒ (3*(snd (9,8))) + (3*(snd (9,8)))
                           ⇒ (3*8) + (3*(snd (9,8)))
                           ⇒ 24 + (3*(snd (9,8)))
                           ⇒ 24 + (3*8)
                           ⇒ 24 + 24
                           ⇒ 48
```

Com a estrategia *lazy* os parametros das funções só são calculados se o seu valor fôr mesmo necessário.

`nl (triplo (quad (7*45)) ⇒ '\n'`

A *lazy evaluation* faz do Haskell uma linguagem **não estrita**. Esto é, uma função aplicada a um valor indefinido pode ter em Haskell um valor bem definido.

`nl (3/0) ⇒ '\n'`

A *lazy evaluation* também vai permitir ao Haskell lidar com *estruturas de dados infinitas*.

Podemos definir uma função recorrendo a várias equações.

Exemplo:

```
h :: (Char,Int) -> Int
h ('a',x) = 3*x
h ('b',x) = x+x
h (_,x) = x
```

Todas as equações têm que ser bem tipadas e de tipos coincidentes.

Cada equação é usada como regra de redução. Quando uma função é aplicada a um argumento, a equação que é selecionada como regra de dedução é a 1ª equação (a contar de cima) cujo padrão que tem com argumento **concorda** com argumento actual (***pattern matching***).

Exemplos:

```
h ('a',5) ⇒ 3*15  ⇒ 45
h ('b',4) ⇒ 4+4  ⇒ 8
h ('B',9) ⇒ 9
```

Note: Podem existir *várias* equações com padrões que concordam com o argumento actual. Por isso, a ordem das equações é importante, pois define uma prioridade na escolha da regra de redução.

O que acontece se alterar a ordem das equações que definem h ?

Funções Totais & Funções Parciais

Uma função diz-se **total** se está definida para todo o valor do seu domínio.

Uma função diz-se **parcial** se há valores do seu domínio para os quais ela não está definida (isto é, não é capaz de produzir um resultado no conjunto de chegada).

Exemplos:

```
conjuga :: (Bool,Bool) -> Bool
conjuga (True,True) = True
conjuga (x,y) = False
```

Função total

```
parc :: (Bool,Bool) -> Bool
parc (True,False) = False
parc (True,x) = True
```

Função parcial

Porquê ?

Tipos Simónimos

O Haskell pode renomear tipos através de declarações da forma:

type *nome p1 ... pn = tipo*

parâmetros (*variáveis de tipo*)

Exemplos:

```
type Ponto = (Float,Float)
type ListaAssoc a b = [(a,b)]
```

Note que não estamos a criar tipos novos, mas apenas nomes novos para tipos já existentes. Esses nomes devem contribuir para a compreensão do programa.

Exemplo:

```
distOrigem :: Ponto -> Float
distOrigem (x,y) = sqrt (x^2 + y^2)
```

O Haskell tem pré-definido o tipo **String** como sendo `[Char]`. `-- type String = [Char]`

Os valores do tipo `String` também se escrevem de forma abreviada entre " ".

Exemplo:

`"haskell"` é equivalente a `['h','a','s','k','e','l','l']`

```
>  "Ola" == ['O','l','a']
True
```


Definições Locais

Uma definição associa um nome a uma expressão.

Todas as definições feitas até aqui podem ser vistas como **globais**, uma vez que elas são visíveis no *módulo* do programa aonde estão. Mas, muitas vezes é útil reduzir o âmbito de uma declaração.

Em Haskell há duas formas de fazer definições **locais**: utilizando expressões `let ... in` ou através de cláusulas `where` junto da definição equacional de funções.

Exemplos:

```
let  c = 10
     (a,b) = (3*c, f 2)
     f x = x + 7*c
in f a + f b
```

⇒ 242

Porquê ?

```
testa y = 3 + f y + f a + f b
  where  c = 10
         (a,b) = (3*c, f 2)
         f x = x + 7*c
```

```
> testa 5
320

> c
Variable not in scope: `c'

> f a
Variable not in scope: `f'
Variable not in scope: `a'
```

As declarações locais podem ser de funções e de identificadores (fazendo uso de padrões).