Raciocinar Sobre Programas

* Material baseado no livro "Programming in Haskell", Graham Hutton.

- Raciocínio Equacional

  - Regras algébricas básicas:

    1. $xy = yx$
    2. $(x+y)+z = x+(y+z)$
    3. $x(y+z) = xy + xz$
    4. $(x+y)z = xz + yz$

Se quisermos provar $x^2 + (a-b)x - x(a+x) = -bx$ podemos usar estas regras:

$$
\begin{aligned}
& x^2 + (a-b)x - x(a+x) \\
= & \{4\} \\
& x^2 + ax - bx - x(a+x) \\
= & \{3\} \\
& x^2 + ax - bx - \overset{\frown}{xa} - x^2 \\
= & \{2,1\} \\
& \cancel{x^2} - \cancel{x^2} + \cancel{ax} - \cancel{ax} - bx \\
= & \{\text{Aritmética}\} \\
& -bx
\end{aligned}
$$

Raciocinar em Haskell

```haskell
double :: Int -> Int
double x = x*x
```

Definição da função é Propriedade que pode ser usada no raciocínio sobre esta função.

Pode ser usada da esquerda para a direita (ED) ou da direita para a esquerda (DE).

Cuidado com funções definidas por várias equações:

```haskell
isSingleton :: [a] -> Bool
isSingleton [x] = True
isSingleton xs  = False
```

Não podem ser vistas como propriedades isoladas. Têm que ser interpretadas de acordo com a forma e ordem em que os padrões aparecem.

Para simplificar o processo de raciocinar sobre programas, é boa prática escrever padrões que não fazem "overlap"! Eg, usar guardas:

```haskell
isSingleton xs | (length xs /= 0) = False
isSingleton [x] = True
```

A ordem já não importa.

Exemplos

Reverse :: [a] → [a]

① Reverse [ ] = [ ]

Reverse (x:xs) = Reverse xs ++ [x]

→ Provar que aplicar o Reverse a uma lista unitária não tem qualquer efeito:

i.e. $\boxed{\text{Reverse } [x] = [x]}$

e.g.: Reverse [1] = [1]

Reverse [x]

= { Notação de listas [x] = x : [ ] }

Reverse (x : [ ])

= { aplicar Reverse }

Reverse [ ] ++ [x]

①

= { aplicar Reverse }

[ ] ++ [x]

= { aplicar ++ }

[x]

⊜

Conclusão:

$\boxed{\text{Sempre que Reverse [x] ocorre num programa, pode ser substituído por [x].}}$

Não altera o significado mas melhora a eficiência.

→ Por vezes também se usa análise de casos. Por exemplo:

not :: Bool → Bool

not False = True

not True = False

Por estar definida através de "pattern-matching", provar propriedades sobre a função not é normalmente feito com análise de casos.

Exemplo, provar que "not" é a sua própria inversa: $\boxed{\text{not (not } x) = x}$

• Caso: x = True

not (not True)

= { aplicar o "not" interno }

not False

= { aplicar "not" }

True ✓

⊜

Caso: x = False

not (not False)

= { aplicar o "not" interno }

not True

= { aplicar o "not" }

False ✓

⊜

Indução sobre números

Raciocínio sobre programas recursivos é normalmente feito através de indução.
Exemplo, consideremos o tipo dos números naturais: (consideramos aqui apenas aplicações finitas de Succ)

$$\text{data Nat} = \underbrace{\text{Zero}}_{\text{caso base}} \mid \underbrace{\text{Succ Nat}}_{\text{caso recursivo}}$$

Exemplo, se quiser representar o número 3: Succ (Succ (Succ Zero))

Para provar que uma propriedade é verdadeira para todos os números naturais, então basta provar que a propriedade é verdadeira para Zero (caso base) e que é preservada pela aplicação de Succ (caso indutivo/passo de indução).

Consideremos a função:

```
add :: Nat -> Nat -> Nat
add Zero n = n   *
add (Succ m) n = Succ (Add m n)
```

H.I. (Hipótese de Indução)

Provar que: $\boxed{\text{add n Zero} = n}$ Não é imediato pela definição:

⇒ Assumir que $\boxed{\text{add n Zero} = n.}$ H.I.

- Caso base: Zero

  add Zero Zero
  = { aplicar add } *
  Zero

Passo de Indução: Succ n

add (Succ n) Zero
= { aplicar add } → H.I.: add n Zero = n
Succ ($\boxed{\text{add n Zero}}$)
= { H.I }
Succ n

A mesma abordagem funciona para os valores do tipo de inteiros do Haskell.

Indusão Sobre Listas

O nosso caso base é a lista vazia [], assume-se a propriedade para xs e mostra-se que também é verdadeira para x:xs (passo de Indusão).

Exemplo, provar que:

Reverse (xs ++ys) = Reverse ys ++ Reverse xs

Assumimos que 1. ++ é associativo e 2. [] é a identidade de ++.
Indusão sobre xs.

* Caso base: []

Reverse ([] ++ys)
= { aplicação do ++ } ([] é a identidade de ++)
Reverse ys
= { [] é a identidade de ++ }
Reverse ys ++ []
= { aplicar Reverse DE } → Direita para a Esquerda

$\xleftarrow{DE}$ Reverse [] = []

Reverse ys ++ Reverse [] ✓

* Passo de Indusão: (x:xs)

H.I. $\boxed{\text{Reverse (xs ++ys) = Reverse ys ++ Reverse xs}}$

Reverse ((x:xs) ++ys)
= { aplicar o ++ }
Reverse (x:(xs ++ys))
= { aplicar Reverse }
$\boxed{\text{Reverse (xs ++ys)}}$ ++ [x]   (H.I.)
= { H.I }
(Reverse ys ++ Reverse xs) ++ [x]
= { ++ é associativa }
Reverse ys ++ (Reverse xs ++ [x])
= { Reverse DE }
Reverse ys ++ Reverse (x:xs) ✓

se tiverem dificuldades a chegar ao objectivo da prova podem escrever o objetivo e tentar desenvolver a prova para "cima".

Quando falámos na classe Functor, vimos que a função fmap tem que satisfazer duas regras equacionais:

1. $fmap\ id = id$
2. $fmap\ (g . h) = fmap\ g\ .\ fmap\ h$

Relembrar que $\bullet$ é a composição de funções:
$(g . h) x = g\ (h x)$

Consideremos o fmap das listas:

$fmap :: (a \to b) \to [a] \to [b]$

① $fmap\ g\ [\,] = [\,]$

② $fmap\ g\ (x : xs) = g x : fmap\ g\ xs$

Vamos verificar a primeira regra: $fmap\ id\ xs = id\ xs$ para qualquer listas.
Como $id\ x = x$, então podemos simplificar o que queremos provar para

$\boxed{fmap\ id\ xs = xs}$ Hipótese de Indução.

* Caso base: $[\,]$

$fmap\ id\ [\,]$

$=$ { aplicar fmap ① }

$[\,]$

* Passo de Indução: $(x : xs)$

$fmap\ id\ (x : xs)$

$=$ { aplicar fmap ② }

$id\ x : fmap\ id\ xs$

$=$ { aplicar id (i.e. $id\ x = x$) }

$x : \boxed{fmap\ id\ xs}$ id

$=$ { H.I }

$(x : xs)$

Vamos agora provar a segunda regra:

$$fmap\ (g.h)\ xs = (fmap\ g\ .\ fmap\ h)\ xs \quad \text{para qualquer lista } xs.$$

Como $(g.h)\,x = g\,(h\,x)$, podemos simplificar esta equação para:

$$\boxed{fmap\ (g.h)\ xs = fmap\ g\ (fmap\ h\ xs)} \quad H.I.$$

* Caso base: [ ]

$fmap\ (g.h)\ [\,]$

$= \{$ aplicar fmap $\}$

$[\,]$

$= \{$ fmap DE $(fmap\ g\ [\,] = [\,])$ $\}$

$fmap\ g\ [\,]$

$= \{$ fmap DE $(fmap\ f\ [\,] = [\,])$ $\}$

$fmap\ g\ (fmap\ f\ [\,])$ ✓

* Passo de Indução: $(x:xs)$

$fmap\ (g.h)\ (x:xs)$

$= \{$ aplicar fmap $\}$

$(g.h)\,x : \boxed{fmap\ (g.h)\ xs}$ H.I

$= \{$ H.I. $\}$

$(g.h)\,x : fmap\ g\ (fmap\ h\ xs)$

$= \{$ aplicação do . $\}$

$g\,(h\,x) : fmap\ g\ (fmap\ h\ xs)$

$= \{$ fmap DE $\}$

$fmap\ g\ (h\,x : fmap\ h\ xs)$

$= \{$ fmap DE $\}$

$fmap\ g\ (fmap\ h\ (x:xs))$ ✓