Programação Funcional
Aula 19 — Raciocinar sobre programas

Pedro Vasconcelos
DCC/FCUP

2021

Raciocínio equacional

Para simplificar expressões matemáticas podemos usar igualdades algébricas.

Este tipo manipulação chama-se raciocínio equacional.

Algumas igualdades algébricas

$$a \cdot b = b \cdot a \qquad \text{comutatividade de } \cdot$$

$$(a + b) \cdot c = a \cdot c + b \cdot c \qquad \text{distributividade de } \cdot \text{ sobre } +$$

Podemos usar estas igualdades para substituir os lados esquerdos pelos lados direitos ou vice-versa.

Exemplo

$$
\begin{array}{cl}
 & (x+1)\cdot(x+2) \\
= & \{\text{ distributividade }\} \\
 & (x+1)\cdot x+(x+1)\cdot 2 \\
= & \{\text{ distributividade }\} \\
 & x^2+1\cdot x+x\cdot 2+1\cdot 2 \\
= & \{\text{ comutatividade, distributividade }\} \\
 & x^2+(1+2)\cdot x+1\cdot 2 \\
= & \{\text{ aritmética }\} \\
 & x^2+3x+2
\end{array}
$$

Raciocínio equacional sobre programas

Podemos mostrar propriedades de programas em Haskell usando as definições das funções como igualdades algébricas.

Raciocínio equacional sobre programas (cont.)

Vamos mostrar que

```
reverse [x] = [x]
```

usando as definições seguintes:

```
reverse []     = []                   -- reverse.1
reverse (x:xs) = reverse xs ++ [x]    -- reverse.2

[]     ++ ys = ys                     -- ++.1
(x:xs) ++ ys = x:(xs++ys)             -- ++.2
```

Exemplo

Começando pelo lado esquerdo:

$$
\begin{array}{cl}
 & \textit{reverse}\ [x] \\
= & \{\ \text{notação}\ \} \\
 & \textit{reverse}\ (x : []) \\
= & \{\ \text{reverse.2}\ \} \\
 & \textit{reverse}\ []\ +\!\!+\ [x] \\
= & \{\ \text{reverse.1}\ \} \\
 & []\ +\!\!+\ [x] \\
= & \{\ +\!\!+.1\ \} \\
 & [x]
\end{array}
$$

Obtemos a expressão do lado direito.

Porquê provar propriedades de programas?

- Verificação formal da correcção
  1. provar propriedades universais
  2. garantia de resultados correctos para *quaisquer* valores
  3. garantia de terminação e ausência de erros
- Simplificação e transformação
  1. transformar programas usando igualdades
  2. sintetizar programas a partir de especificações
  3. obter um programa eficiente a partir de um mais simples

"Testing shows the presence, not the absence of bugs."
— *E. Disjkstra*

Porquê em Haskell?

Podemos usar raciocínio equacional sobre programas Haskell porque são definidos usando *equações*.

Por contraposição, os programas imperativos são definidos por *sequências de instruções* — não são equações.

Exemplo

Após a instrução

```
n = n+1;    // em C,C++,Java...
```

não podemos substituir n por $n + 1$ — trata-se duma atribuição e não duma equação.

Recursão e indução

Em programação funcional usamos recursão para definir funções sobre números naturais, listas, árvores, etc.

Além de raciocínio equacional, necessitamos de indução matemática para provar propriedades dessas funções.

Exemplo: números naturais

```
data Nat = Zero | Succ Nat
```

Construidos apartir do zero aplicando o sucessor:

```
Zero
Succ Zero
Succ (Succ Zero)
Succ (Succ (Succ Zero))
⋮
```

Cada natural é finito mas há uma infinidade de números naturais.

Indução sobre naturais

Para provar $P(n)$ basta:

1. mostrar $P(\texttt{Zero})$
2. mostrar $P(\texttt{Succ}\ n)$ usando a hipótese de indução $P(n)$

Formalmente

$$\frac{\begin{array}{l} P(\texttt{Zero}) \\ P(n) \implies P(\texttt{Succ}\ n) \quad \text{para todo } n \end{array}}{P(n) \quad \text{para todo } n}$$

Adição de naturais

```
(+) :: Nat -> Nat -> Nat
Zero + m   = m                  -- +.1
Succ n + m = Succ (n + m)       -- +.2
```

Vamos mostrar

```
n + Zero = n
```

usando indução sobre *n*.

Adição de naturais (cont.)

Obrigações de prova:

caso base: `Zero + Zero = Zero`

caso indutivo: `n + Zero = n` $\Longrightarrow$ `Succ n + Zero = Succ n`

Prova por indução

Caso base

```
  Zero + Zero
=    { +.1 }
  Zero
```

Prova por indução (cont.)

Caso indutivo

Hipótese: `n + Zero = n`

Tese: `Succ n + Zero = Succ n`

```
  Succ n + Zero
    { +.2 }
= Succ (n + Zero)
    { hipótese de indução }
= Succ n
```

Outro exemplo

Exercício: provar a associatividade da adição

$$x + (y + z) = (x + y) + z$$

por indução sobre x.

Indução sobre inteiros

Podemos também usar indução sobre inteiros pré-definidos.
Nesse caso temos de escolher um valor base (por ex.: 0).

$$\frac{\begin{array}{l} P(0) \\ P(n) \implies P(n+1) \quad \text{para todo } n \geq 0 \end{array}}{P(n) \quad \text{para todo } n \geq 0}$$

A propriedade fica provada apenas para os inteiros não-negativos.

Exemplo

```
length :: [a] -> Int
length []     = 0                    -- length.1
length (x:xs) = 1 + length xs        -- length.2

replicate :: Int -> a -> [a]
replicate 0 x  = []                  -- replicate.1
replicate n x | n>0
    = x : replicate (n-1) x          -- replicate.2)
```

Vamos mostrar que

```
length (replicate n x) = n
```

usando indução sobre `n`.

Prova por indução

Caso base

```
length (replicate 0 x) = 0
```

```
  length (replicate 0 x)
=   { replicate.1 }
  length []
=   { length.1 }
  0
```

Prova por indução (cont.)

Case indutivo

Hipótese: `length (replicate n x) = n`

Tese: `length (replicate (1+n) x) = 1+n`

```
  length (replicate (1+n) x)
=   { replicate.2 }
  length (x :  replicate n x)
=   { length.2 }
  1 + length (replicate n x)
=   { hipótese de indução }
  1 + n
```

Indução sobre listas

Também podemos provar propriedades usando indução sobre listas.

Caso base: $P(\texttt{[]})$

Caso indutivo: $P(xs) \implies P(x : xs)$ para todos x, xs

Formalmente

$$\frac{\begin{array}{l} P(\texttt{[]}) \\ P(xs) \implies P(x : xs) \quad \text{para todo } x, xs \end{array}}{P(xs) \quad \text{para todo } xs}$$

Exemplo

Vamos mostrar que

```
xs ++ [] = xs
```

por indução sobre `xs`.

Exemplo

Caso base:

```
   [] ++ []
=  { ++.1 }
   []
```

Caso indutivo:

Hipótese: `xs ++ [] = xs`

Tese: `(x:xs) ++ [] = (x:xs)`

```
    (x:xs) ++ []
 =    { ++.2 }
    x :  (xs ++ [])
 =    { hipótese de indução }
    x:xs
```

Segundo exemplo

Mostrar

```
reverse (reverse xs) = xs
```

por indução sobre `xs`.

Caso base

```
  reverse (reverse [])
=   { reverse.1 interior }
  reverse []
=   { reverse.1 }
  []
```

Caso indutivo

Hipótese: `reverse (reverse xs) = xs`

Tese: `reverse (reverse (x:xs)) = x:xs`

```
   reverse (reverse (x:xs))
=    { reverse.2 interior }
   reverse (reverse xs ++ [x])
```

- Não conseguimos avançar!
- Por vezes é necessário algum resultado auxiliar para continuar

Dois lemas auxiliares

Distributividade de `reverse` sobre `++`

```
reverse (xs ++ ys) = reverse ys ++ reverse xs
```

Atenção à inversão da ordem dos argumentos!

Para provar o lema acima, necessitamos de mostrar:

Associatividade de `++`

```
(xs ++ ys) ++ zs = xs ++ (ys ++ zs)
```

Exercício: provar estes lemas usando indução.

De regresso à prova

```
  reverse (reverse (x:xs))
=   { reverse.2 interior }
  reverse (reverse xs ++ [x])
=   { distributividade reverse sobre ++ }
  reverse [x] ++ reverse (reverse xs)
=   { reverse [x] = [x] }
  [x] ++ reverse (reverse xs)
=   hipótese de indução
  [x] ++ xs
=   { ++.2, ++.1 }
  x:xs
```

Outros tipos de dados recursivos

Podemos associar um regra de prova por indução a cada tipo de dados recursivo.

Exemplo: ao tipo de árvores binárias

```
data Arv a = Vazia | No a (Arv a) (Arv a)
```

corresponde a regra

$$\frac{\begin{array}{l} P(\texttt{Vazia}) \\ P(esq) \wedge P(dir) \implies P(\texttt{No}\ x\ esq\ dir) \end{array}}{P(t) \qquad \text{para toda a árvore } t}$$

(Veremos mais exemplos nas aulas seguintes.)

Sintetizar programas

Podemos usar raciocínio equacional e indução para sintetizar um programa a partir de outro.

Exemplo: transformar um programa noutro equivalente mas mais eficiente.

Sintetizar programas (cont.)

A definição natural de `reverse` é ineficiente por causa do uso de ++ na recursão.

```
reverse :: [a] -> [a]
reverse []       = []
reverse (x:xs) = reverse xs ++ [x]
```

Vamos obter uma versão mais eficiente eliminando as concatenações.

Eliminar concatenações

Vamos encontrar uma função

```
revacc :: [a] -> [a] -> [a]
```

tal que

```
revacc xs ys = reverse xs ++ ys      -- especificação
```

Queremos obter uma definição recursiva de `revacc` sem usar `reverse` e `++`.

Eliminar concatenações

Caso o 1º argumento seja `[]`.

```
  revacc [] ys
=   { especificação }
  reverse [] ++ ys
=   { reverse.1 }
  [] ++ ys
=   { ++.1 }
  ys
```

Eliminar concatenações

Caso o 1º argumento seja `x:xs`.

```
  revacc (x:xs) ys
=   { especificação }
  reverse (x:xs) ++ ys
=   { reverse.2 }
  (reverse xs ++ [x]) ++ ys
=   { associatividade de ++ }
  reverse xs ++ ([x] ++ ys)
=   { ++.2, ++.1 }
  reverse xs ++ (x:ys)
=   { especificação }
  revacc xs (x:ys)
```

Eliminar concatenações

Combinando os dois casos obtemos a definição recursiva de `revacc`:

```
revacc :: [a] -> [a] -> [a]
revacc []     ys = ys
revacc (x:xs) ys = revacc xs (x:ys)
```

Concluimos definindo `reverse` usando `revacc`.

```
reverse :: [a] -> [a]
reverse xs = revacc xs []
```