Programação Funcional

(COMP0393)

Leila M. A. Silva

dcomp
departamento de computação

Indução Forte e Recursão Geral

(COMP0393)

Aula 8

dcomp departamento de computação

Indução Matemática

- Nas aulas anteriores estudamos indução fraca e como este princípio indutivo pode ser usado na construção de funções com recursão primitiva.
- Na indução fraca o passo indutivo é de um, $P(k) \Rightarrow P(k+1)$.
- Mas para alguns problemas precisamos de um passo maior. Neste caso, dizemos que a indução é **forte.**

Indução Matemática (Forte)

- Seja $P(n)$ um predicado definido para os inteiros n e sejam a e b dois inteiros fixos, sendo $a \leq b$.
- Suponha que as duas afirmações seguintes sejam verdadeiras:
  - $P(a)$, $P(a+1)$,.., $P(b)$ são V. (Casos base)
  - Para qualquer inteiro $k \geq b$,

    se $P(i)$ é V para $a \leq i < k$, então $P(k)$ é V, isto é, $P(i) \Rightarrow P(k)$.
- Logo, a afirmação para todos inteiros $n \geq a$, $P(n)$ é V.
- A suposição que $P(i)$ é V para $a \leq i < k$ é chamada de **hipótese indutiva.**

Indução Matemática (Forte)

- Prova por indução forte é similar ao caso anterior de indução fraca:
  - Defina a variável de indução;
  - Estabeleça e prove o(s) *caso(s) base;*
  - Estabeleça a Hipótese de Indução (HI);
  - Estabeleça e prove o Passo Indutivo ou Caso Geral.
- A diferença maior reside no fato a hipótese de indução deve ser suposta verdadeira não só para o elemento anterior ao que se quer provar, mas para **todos** os anteriores a ele entre o caso base e ele.

Aplicando o Princípio de Indução Forte

- Prove que qualquer inteiro maior que 1 é divisível por um número primo.
- Prova:
  - Caso base: n=2. A prova do caso é válida porque 2 é divisível por 2 e o número 2 é primo.
  - Hipótese de Indução: Vamos supor que para todos inteiros i, $2 \leq i < k$, i é divisível por um número primo.
  - Caso geral: Supondo HI, desejamos provar que k é divisível por um número primo.

Aplicando o Princípio de Indução

- Caso geral: Supondo HI, desejamos provar que k é divisível por um número primo, ou seja, se a propriedade é válida para $2 \leq i < k$, então é válida para k.

  Seja k um inteiro, $k > 2$. Temos dois casos a considerar:

  1. k é primo ou
  2. k não é primo.

  No primeiro caso, k é divisível por um primo que é ele próprio, já que todo inteiro é divisível por si mesmo.

  No segundo caso, se k não é primo então $k = u \,.v$, onde u e v são inteiros tais que $2 \leq u < k$ e $2 \leq v < k$. Pela hipótese indutiva, u é divisível por um número primo p. Como se um número é divisível por outro qualquer múltiplo dele também é então k também é divisível por p. Assim, independente se k é primo ou não, k é divisível por um primo. ■

Observe que não podemos usar a indução fraca porque não podemos garantir que u ou v sejam $k-1$. Só podemos garantir que ele será um número entre 2 e $k-1$.

Exemplos

Seja a sequência a_1, a_2, a_3,... definida como:

$a_1 = 0$;

$a_2 = 2$;

$a_k = 3.\ a_{\lfloor k/2 \rfloor} + 2,\ k \geq 3$

Prove que a_n é par para $n \geq 1$.

- Caso base: $n=1$, $n=2$ e $n=3$. Nestes casos os valores são pares pois $a_1 = 0$, $a_2 = 2$ e $a_3 = 2$.
- Hipótese de Indução: Supomos que a_i é par para todos os inteiros i, $1 \leq i < k$
- Caso Geral: Se a *HI* é V então a asserção é valida para k, ou seja, a_k é par.

Exemplos

- Caso Geral: Se a *HI* é V então a asserção é valida para k, ou seja, a_k é par.

  Prova:

  Pela definição, $a_k = 3.\ a_{\lfloor k/2 \rfloor} + 2,\ k \geq 3$. Como $1 \leq k/2 < k$, por HI $a_{\lfloor k/2 \rfloor}$ é par. Logo, $3.\ a_{\lfloor k/2 \rfloor}$ é par, porque qualquer número múltiplo de par é par. Como 2 é par e soma de pares é par, a_k é par. ■

Observe que não podemos usar a indução fraca porque $1 \leq k/2 < k$.

Exemplos

Seja a sequência a_1, a_2, a_3,... definida como:

$a_1 = 1; a_2 = 2; a_3 = 3;$

$a_k = a_{k-1} + a_{k-2} + a_{k-3}, k \geq 4$

Prove que $a_n < 2^n$, $n \geq 1$.

- Caso base: $n=1$, $n=2$, $n=3$ e $n=4$. Nestes casos a asserção é válida, ou seja $a_n < 2^n$ porque 1<2, 2<4, 3<8 e 6<16.
- Hipótese de Indução: Supomos que $a_i < 2^i$ para todos os inteiros i, $1 \leq i < k$
- Caso Geral: Se a *HI* é V então a asserção é valida para k, ou seja, $a_k < 2^k$.

Exemplos

- Caso Geral: Se a *HI* é V então a asserção é valida para *k*, ou seja, $a_k < 2^k$.

  Prova:

  Por definição, $a_k = a_{k-1} + a_{k-2} + a_{k-3}$. Como *k-1*, *k-2* e *k-3* são menores que *k*, por HI vale que $a_{k-1} < 2^{k-1}$, $a_{k-2} < 2^{k-2}$ e $a_{k-3} < 2^{k-3}$. Logo,

$$a_k = a_{k-1} + a_{k-2} + a_{k-3}$$
$$< 2^{k-1} + 2^{k-2} + 2^{k-3} = 2^{k-3}(2^2 + 2 + 1) = 7 \cdot 2^{k-3}$$
$$< 8 \cdot 2^{k-3} = 2^3 \, 2^{k-3} = 2^k \quad \blacksquare$$

Observe que não podemos usar a indução fraca porque preciso de três termos menores e não um só!

Exercícios Recomendados

- Seja a sequência a_1 , a_2 , a_3 ,... definida como:

  $a_1 = 1; a_2 = 2;$

  $a_k = a_{k-2} + 2. a_{k-1}, k \geq 3$

  Prove que a_n é ímpar para todos os inteiros $n \geq 1.$

- Seja a sequência a_0 , a_1 , a_2 ,... definida como:

  $a_0 = 12; a_1 = 29;$

  $a_k = 5a_{k-1} - 6\, a_{k-2}, k \geq 2$

  Prove que a_n é $5. 3^n + 7. 2^n$, para todos os inteiros $n \geq 0.$

Recursão Geral

- As funções construídas usando o princípio de indução forte resultam também em funções recursivas e como o passo é maior que um, a esta recursão chamamos de **recursão geral.**
- Da mesma forma que na indução fraca, podemos usar a indução forte para a construção de funções recursivas.

Exemplos

- Sequência de Fibonacci
  - A sequência de Fibonacci começa com 0 e 1 e depois os próximos números são a soma dos dois anteriores.

    0,1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...
- A função para calcular o *n*-ésimo número de Fibonacci pode ser construída por indução forte da seguinte maneira:
  - Casos base: *n=0* e *n=1*. Nestes casos a função retornaria 0 e 1, respectivamente.
  - HI: Sei calcular a função para todos os inteiros *i*, $0 \leq i < k$.
  - Caso geral: calcular a função para o *k*-ésimo termo

Exemplos

- Caso geral: calcular a função para o *k*-ésimo termo
  - Estratégia: *fib (k) = fib(k-1) + fib(k-2)*. Como , $0 \leq k-1 < k$ e , $0 \leq k-2 < k$ então posso aplicar a HI e obter o resultado. Para determinar *fib(k)* é só somar os resultados obtidos por indução.
  - Observe que preciso de indução forte, porque não consigo resolver o problema somente com a instância *k-1*.
  - Podemos escrever a função abaixo para o caso de sucesso.

```
fib :: Integer -> Integer
fib 0 = 0
fib 1 = 1
fib k = fib (k-1) + fib(k-2)
```

Exemplos

- Ou usando guardas...

```
fib :: Integer -> Integer
fib k
  | k == 0 = 0
  | k == 1 = 1
  | otherwise = fib (k-1) + fib(k-2)
```

- Para incluir a situação de insucesso, a construção com guarda estende mais facilmente para

```
fib :: Integer -> Integer
fib k
  | k < 0 = error "indefinido para numero negativo"
  | k == 0 = 0
  | k == 1 = 1
  | otherwise = fib (k-1) + fib(k-2)
```

Exemplos

- Esta formulação de Fibonacci não é muito eficiente porque quando calculamos `fib (k-2)` duas vezes.
- Avaliação

```
fib 4 ↝ fib 3 + fib 2 ↝
        fib 2 + fib 1 + fib 2 ↝
        fib 1 + fib 0 + fib 1 + fib 2 ↝
        1 + fib 0 + fib 1 + fib 2 ↝
        1 + 0 + fib 1 + fib 2 ↝
        1 + 0 + 1 + fib 2 ↝
        1 + 0 + 1 + fib 1 + fib 0 ↝
        1 + 0 + 1 + 1 + fib 0 ↝
        1 + 0 + 1 + 1 + 0 ↝ 3
```

- Veremos mais adiante como calcular Fibonacci de forma mais eficiente.

Exemplos

- Potência: Vamos retomar o problema de calcular a^n, $n \geq 0$, que já vimos na aula de recursão primitiva. Por convenção, $0^0 = 1$.
- Vamos tentar usar agora a seguinte propriedade para o cálculo da potência
  - $a^n = (a^{n/2})^2$, se n é par
  - $a^n = a.(a^{n/2})^2$, se n é impar
- Construindo por indução forte teríamos que:
  - Variável de indução: n
  - Casos base: $n=0$ e $n=1$. Neste caso a função retorna 1 e a pois $a^0=1$, $a^1=a$. HI: Sei calcular a^i , $0 \leq i < k$.
  - Caso Geral: Calcular a^k.

Exemplos

- Caso Geral: Calcular a^k.
  - Estratégia: Observe que usando a propriedade, nos dois casos, preciso calcular $a^{k/2}$. Mas neste termo podemos aplicar a HI pois , $0 \leq k/2 < k$. Logo, precisamos apenas checar se k é par ou não para usar a propriedade correta e chegar ao resultado.

```
pot :: Float -> Int -> Float
pot a k
   | k<0 = error "expoente negativo"
   | k == 0 = 1
   | k == 1 = a
   | ehPar k = (pot a (div k 2)) ^ 2
   | otherwise = a * (pot a (div k 2)) ^ 2
  where ehpar x = (mod x 2 == 0)
```

Exemplos

- Avaliação

```
pot 2 4 ⇝ (pot 2 2) ^ 2
          (pot 2 1)^2^2 ⇝
           2^2^2 ⇝2^4 ⇝16

pot 2 5 ⇝ 2 * (pot 2 2) ^ 2
          2* (pot 2 1)^2^2 ⇝
          2* 2^2^2 ⇝ 2* 2^4 ⇝32
```

Exemplos

- Palíndromos: são sequências que podem ser lidas da esquerda para a direita ou da direita para a esquerda.
- Ex: `"arara", [1, 2, 2, 1]`
- Elabore uma função que dada uma palavra, retorna `True` se ela for palíndromo.
  - Variável de indução: *n*, tamanho da palavra.
  - Casos base: lista vazia [] e lista unitária [x]. Por definição, retorne `True`.
  - HI: Sei calcular se a palavra é palíndromo para palavras com $0 \leq i < k$ caracteres.
  - Caso Geral: Calcular para uma palavra de *k* caracteres.

Exemplos

- Caso Geral: Calcular para uma palavra de *k* caracteres.

$[a_1, a_2, \ldots, a_{k-1}, a_k]$

se $a_1 \neq a_k$ então não é palíndromo

senão preciso checar se $[a_2, \ldots, a_{k-1}]$ é palíndromo

  - Observe que $[a_2, \ldots, a_{k-1}]$ possui tamanho *k-2* e portanto podemos aplicar a HI que irá nos retornar se é ou não palíndromo!

```
palindromo :: [Char] -> Bool
palindromo [] = True
palindromo [x] = True
palindromo xs = head xs == last xs && palindromo (init (tail xs))
```

Exemplos

- Avaliação

```
palindromo “arara” ↝ True && palindromo “rar” ↝
                        True && True && palindromo “a” ↝
                        True && True && True ↝ True
```

Cuidado!

- Precisamos definir precisamente os casos base senão podemos construir uma função que rode indefinitivamente.
- Por exemplo, suponha que você deseje implementar uma função para determinar se o número é ou não par e elabore a seguinte função

```
par :: Int -> Bool
par n
  | n == 0 = True
  | otherwise = par (n-2)
```

- Você só pensou no caso base quando *n* for par e aí a função funciona. E quando for ímpar???

```
par :: Int -> Bool
par n
  | n == 0 = True
  | n == 1 = False
  | otherwise = par (n-2)
```

Cuidado!

- Casos especiais podem afetar o passo indutivo. Considere agora o problema de dividir um número inteiro por outro usando as seguintes funções , que dão o quociente e o resto da divisão de inteiros

```
meuDiv :: Int -> Int -> Int
meuMod :: Int-> Int -> Int
```

- A ideia seria usar subtração para realizar a divisão. Por exemplo, para dividir 37 por 10 podemos ir subtraindo 10 até que o valor fique menor que 10. O resto poderia ser calculado de maneira análoga.

```
meuDiv m n
  | m < n = 0
  | otherwise = 1 + meuDiv (m-n) n
meuMod m n
  | m < n = m
  | otherwise = meuMod (m-n) n
```

- Mas, o que acontece se *n*=*0*?? E se *n* = *-4*??

Cuidado!

- As funções ficam rodando indefinidamente !!!
- Eu preciso tratar estes casos como uma das guardas da função!!

```
meuDiv m n
    | n <= 0 = error "número não positivo maior que 0"
    | m < n  = 0
    | otherwise = 1 + meuDiv (m-n) n


meuMod m n
    | n <= 0 = error "número não positivo maior que 0"
    | m < n  = m
    | otherwise = meuMod (m-n) n
```

- Observe que usamos a indução forte pois a chamada da função não é necessariamente para um valor 1 menor.

Exercício de Fixação

- Defina uma função recursiva chamada `divMod` que calcula ao mesmo o quociente e o resto da divisão inteira e fornece como saída uma tupla `(quoc, resto)`.

Recursão Mútua

- É possível definir funções recursivas que se chamem mutuamente para a solução de um problema.
- Por exemplo, digamos que dado um número desejamos determinar se ele é par usando as seguintes funções:

```
par :: Int -> Bool
par 0 = True
par n = impar (n-1)

impar :: Int -> Bool
impar 0 = False
impar n = par (n-1)
```

Recursão Mútua

- Avaliação:

```
par 5 ↝ impar 4 ↝ par 3 ↝ impar 2 ↝ par 1 ↝
      ↝ impar 0 ↝ False
```

Fibonacci usando tuplas

- A função de Fibonacci que demos anteriormente é ineficiente. É possível criar uma função mais eficiente usando recursão primitiva e tuplas. A função teria a seguinte propriedade:

  fibTupla (n) = (fib(n), fib(n+1)),

  Assim, dado o par *(u,v)* formado de dois números consecutivos na sequência obtém-se o par *(v, u+v),* que são os próximos dois números na sequência depois de *u*.

```
fibTupla :: Int -> (Int, Int)
fibTupla 0 = (0,1)
fibTupla n = (b, a+b)
   where (a, b) = fibTupla (n-1)

fib n = fst (fibTupla n)
```

Exercícios Recomendados

- Defina uma função recursiva chamada `prodMult` que dados uma lista de inteiros `xs`, e um número `z`, calcula o produto dos números que estão nas posições múltiplas de `z` em `xs`.

  `Ex: xs = [1,4,8,10,30,6] e z = 3` a função retorna

  `8*6 = 48`

- Defina uma função que dada uma palavra e um texto, determina se a palavra ocorre no texto.