Acumuladores

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

Falta de contexto na recursão

Até agora não nos preocupamos com o contexto do uso quando criamos funções recursivas, não importa se é a primeira vez que a função está sendo chamada ou se é a 100ª.

Este princípio de independência do contexto facilita a escrita de funções recursivas, mas pode gerar problemas em algumas situações.

Vamos ver um exemplo.

Exemplo

Dado uma lista de distâncias relativas entre pontos (começando da origem) em uma linha, defina uma função que calcule a distância absoluta a partir da origem.

Exemplo

```
;; Lista(Número) -> Lista(Número)
;; Converte uma lista de distâncias relativas para uma lista
;; de distâncias absolutas. O primeito item da lista
;; representa a distância da origem.
(examples
 (check-equal? (relativa->absoluta empty) empty)
 (check-equal? (relativa->absoluta
                (list 50 40  70  30  30))
                (list 50 90 160 190 220)))

(define (relativa->absoluta lst)
  (cond
    [(empty? lst) ...]
    [else
     (... (first lst)
          (relativa->absoluta (rest lst)))]))
```

Exemplo

Para a entrada `(list 50 40 70 30 30)` a função deve produzir como saída `(list 50 90 160 190 220)`.

Como combinar `(first lst)` – `50` – com `(relativa->absoluta (rest lst))` – `(list 40 110 140 170)` – para obter a resposta para `lst`?

```
(list 50 40 70 30 30)   ->   (list 50 90 160 190 220)

                                50  (list 40 110 140 170)
                                 |  |
                       (first lst)  (relativa->absoluta (rest lst))
```

Somando `50` a cada elemento de `(list 40 110 140 170)`

```
(cons 50 (map (curry + 50) (list 40 110 140 170)))
```

Exemplo

```
;; Lista(Número) -> Lista(Número)
;; Converte uma lista de distâncias relativas
;; para uma lista de distâncias absolutas. O
;; primeito item da lista representa a
;; distância da origem.
(examples
 (check-equal? (relativa->absoluta
                (list 50 40  70  30  30))
                (list 50 90 160 190 220)))
(define (relativa->absoluta lst)
  (cond
    [(empty? lst) empty]
    [else
     (cons
       (first lst)
       (map (curry + (first lst))
            (relativa->absoluta (rest lst))))]))
```

Qual é o problema dessa função?

Ela realiza muito trabalho! O tempo de execução é $\Theta(n^2)$.

Podemos melhorar? Sim!

Como resolveríamos o problema manualmente?

Somando a distância absoluta de um ponto com a distância relativa do próximo.

Vamos tentar definir uma função mais parecida com este método manual.

Exemplo

Começamos com o modelo

```
(define (relativa->absoluta lst)
  (cond
    [(empty? lst) ...]
    [else (... (first lst)
               (relativa->absoluta (rest lst)))]))
```

Exemplo

Como seria a avaliação de
`(relativa->absoluta (list 3 2 7))`?

```
(relativa->absoluta (list 3 2 7))
```

```
(cons ... 3 ...
  (relativa->absoluta (list 2 7)))
```

```
(cons ... 3 ...
  (cons ... 2 ...
    (relativa->absoluta (list 7))))
```

```
(cons ... 3 ...
  (cons ... 2 ...
    (cons ... 7 ...
      emtpy)))
```

Qual é e como obter o primeiro item da resposta? 3 e calculamos diretamente.
Qual é e como obter o segundo item da resposta. 5 e obtemos com `(+ 3 2)`. Na segunda chamada de `relativa->absoluta` obtemos o 2 com `(first lst)`, mas como obtemos o 3? Não temos como obter o 3 pois a recursão é **independe** do que "aconteceu" antes, ou seja, é independente do contexto! O mesmo acontece para o terceiro item, temos que obter `(+ 5 7)` mas não temos acesso ao 5.
Como resolver esse problema, isto é, como acessar a distância absoluta anterior para calcular a distância atual? Adicionando um novo parâmetro que representa a distância absoluta anterior, ou seja, um contexto para a chamada da função.

Exemplo

Com um novo parâmetro, o início da implementação fica

```
;; acc-dist é a distância absoluta do item
;; anterior que estava antes do primeiro elemento de lst.
(define (relativa->absoluta lst acc-dist)
  (cond
    [(empty? lst) ...]
    [else
     (... acc-dist
          (first lst)
          (relativa->absoluta (rest lst) ...))]))
```

O parâmetro `acc-dist` é um **acumulador**, isto é, uma variável que representa o contexto que a função está sendo chamada. Em geral, um acumulador é um resultado parcial do processamento das chamadas recursivas anteriores.

Exemplo

Para a entrada `(list 3 2 7)`, qual deve ser a chamada inicial?
`(relativa->absoluta (list 3 2 7) ...)`

Durante a chamada `(relativa->absoluta (list 3 2 7) ...)`, como é a chamada recursiva para `(rest (list 3 2 7))`?

`(relativa->absoluta (rest (list 3 2 7)) 3)`.

Durante a chamada `(relativa->absoluta (list 2 7) 3)`, como é chamada a recursiva para `(rest (list 2 7))`?

`(relativa->absoluta (rest (list 2 7)) 5)`. Como obtemos 5? `(+ 2 3)`

De forma geral, como é a chamada recursiva?

```
(relativa->absoluta (rest lst) (+ (first lst) acc-dist))
```

Para a entrada `(list 3 2 7)`, qual deve ser a chamada inicial?
`(relativa->absoluta (list 3 2 7) 0)`

Completando a função obtemos

```
(define (relativa->absoluta lst acc-dist)
  (cond
    [(empty? lst) empty]
    [else
     (cons (+ (first lst) acc-dist)
           (relativa->absoluta (rest lst)
                               (+ (first lst) acc-dist)))]))
```

Note que o parâmetro `acc-dist` não é relevante para o problema, apenas para a solução.

Então podemos encapsular a solução sem expor o existência do argumento `acc-dist`.

Exemplo

```
(define (relativa->absoluta lst0)
  (define (iter lst acc-dist)
    (cond
      [(empty? lst) empty]
      [else
       (cons (+ (first lst) acc-dist)
             (iter (rest lst)
                   (+ (first lst) acc-dist)))]))
  (iter lst0 0))
```

Por convenção, chamamos a função com o acumulador de `iter`, em breve veremos porque.

Falta de contexto na recursão

No exemplo `relativa->absoluta` vimos que a falta de contexto durante a recursão tornou a função mais complicada e mais lenta do que o necessário.

Agora veremos um exemplo em que a falta de contexto faz uma função usar mais memória do que o necessário.

Processos iterativos e recursivos

Considere as seguintes implementações para a função que soma dois números naturais utilizando a função `add1`, `sub1` e `zero?`

```
(define (soma a b)
  (if (zero? b)
      a
      (add1 (soma a (sub1 b)))))

(define (soma-alt a b)
  (if (zero? b)
      a
      (soma-alt (add1 a) (sub1 b))))
```

Qual é o processo gerado quando cada função é avaliada com os parâmetros 4 e 3?

```
(define (soma a b)
  (if (zero? b)
      a
      (add1
       (soma a (sub1 b)))))
```

```
(soma 4 3)
(add1 (soma 4 2))
(add1 (add1 (soma 4 1)))
(add1 (add1 (add1 (soma 4 0))))
(add1 (add1 (add1 4)))
(add1 (add1 5))
(add1 6)
7
```

Este é um **processo recursivo**. Ele é caracterizado por uma sequência de operações adiadas e tem um padrão de “cresce e diminui”.

```
(define (soma-alt a b)
  (if (zero? b)
      a
      (soma-alt (add1 a) (sub1 b))))
```

```
(soma-alt 4 3)
(soma-alt 5 2)
(soma-alt 6 1)
(soma-alt 7 0)
7
```

Este é um **processo iterativo**. Nele o "espaço" necessário para fazer a substituição não depende do tamanho da entrada.

Na avaliação da expressão `(soma-alt 4 3)` no exemplo anterior, o valor de `a` foi usado como um acumulador, armazenando a soma parcial.

O uso de acumulador neste problema reduziu o uso de memória.

Recursão em cauda

Uma **chamada em cauda** é a chamada de uma função que acontece como última operação dentro de uma função.

Uma **função recursiva em cauda** é aquela em que todas as chamadas recursivas são em cauda.

A forma de criar processos iterativos em linguagens funcionais é utilizando recursão em cauda.

Os compiladores/interpretadores de linguagens funcionais otimizam as recursões em cauda de maneira que não é necessário manter a pilha das chamadas recursivas, o que torna a recursão tão eficiente quanto um laço em uma linguagem imperativa. Esta técnica é chamada de **eliminação da chamada em cauda**.

Projetando funções com acumuladores

Usar acumuladores é algo que fazemos **depois** que definimos a função e não antes.

As etapas para projetar funções com acumuladores são

- Identificar que a função se beneficia ou precisa de um acumulador
  - Torna a função mais simples
  - Diminui o tempo de execução
  - Diminui o consumo de memória
- Entender o que o acumulador significa e determinar
  - A inicialização
  - A atualização
- Determinar o resultado da função a partir do acumulador

Vamos reescrever diversas funções utilizando acumuladores.

Exemplo - tamanho

```
;; Lista -> Natural
;; Conta a quantidade de elementos de lst.
(examples
 (check-equal? (tamanho empty) 0)
 (check-equal? (tamanho (list 4)) 1)
 (check-equal? (tamanho (list 4 7)) 2)
 (check-equal? (tamanho (list 4 8 -4)) 3))

(define (tamanho lst)
  (cond
    [(empty? lst) 0]
    [else (add1 (tamanho (rest lst)))]))
```

Exemplo - tamanho

Existe algum benefício em utilizar acumulador?

Como o tamanho da resposta não depende do tamanho da entrada, esta função está usando mais memória do que é necessário, portanto ela pode beneficiar-se do uso de acumulador.

Qual o significado do acumulador? A quantidade de elementos já “vistos”.

Qual é o valor inicial do acumulador? `0`.

Como atualizar o acumulador? Somando 1.

Qual é a resposta da função? O valor do acumulador.

Exemplo - tamanho

```
(define (tamanho lst0)
  ;; acc - a quantidade de elementos de lst0 já visitados
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (add1 acc))]))
  (iter lst0 0))
```

Exemplo - soma

```
;; Lista(Número) -> Número
;; Soma os elementos de lst.
(examples
 (check-equal? (soma empty)0)
 (check-equal? (soma (list 3)) 3)
 (check-equal? (soma (list 3 5)) 8)
 (check-equal? (soma (list 3 5 -2)) 6))

(define (soma lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst)
             (soma (rest lst)))]))
```

Exemplo - soma

Existe algum benefício em utilizar acumulador?

Como o tamanho da resposta não depende do tamanho da entrada, esta função está usando mais memória do que é necessário, portanto ela pode beneficiar-se do uso de acumulador.

Qual o significado do acumulador? A soma dos elementos já “vistos”.

Qual é o valor inicial do acumulador? `0`.

Como atualizar o acumulador? Somando o primeiro da lista de entrada.

Qual é a resposta da função? O valor do acumulador.

Exemplo - soma

```
(define (soma lst0)
  ;; acc - a soma dos elementos de lst0 já visitados
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (+ (first lst) acc))]))
  (iter lst0 0))
```

Exemplo - inverte

```
;; Lista -> Lista
;; Inverte a ordem dos elementos de lst.
(examples
 (check-equal? (inverte empty) empty)
 (check-equal? (inverte (list 2)) (list 2))
 (check-equal? (inverte (list 2 8 9)) (list 9 8 2)))

(define (inverte lst)
  (cond
    [(empty? lst) empty]
    [else (append (inverte (rest lst))
                  (list (first lst)))]))
```

Existe algum benefício em utilizar acumulador?

Neste caso a função é mais complicada do que o necessário. Isto porque o resultado da chamada recursiva é processada por outra função recursiva (**append**). Além disso, o tempo de execução desta função é $\Theta(n^2)$, o que intuitivamente é muito para inverter uma lista.

Qual o significado do acumulador? Os elementos que já foram visitados em ordem reversa.

Qual é o valor inicial do acumulador? `empty`.

Como atualizar o acumulador? Colocando o primeiro da entrada como primeiro do acumulador.

Qual é a resposta da função? O valor do acumulador.

Exemplo - inverte

```
(define (inverte lst0)
  ;; acc - os elementos já visitados de lst0 em ordem inversa
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst)
                  (cons (first lst) acc))]))
  (iter lst0 empty))
```

Função `foldl`

Função `foldl`

Vamos observar as semelhanças das funções `tamanho`, `soma` e `inverte`.

Função `foldl`

```
(define (tamanho lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (add1 acc))]))
  (iter lst0 0))
```

Função `foldl`

```
(define (soma lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (+ (first lst) acc))]))
  (iter lst0 0))
```

Função `foldl`

```
(define (inverte lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst) (cons (first lst) acc))]))
  (iter lst0 empty))
```

Função `foldl`

Vamos criar uma função chamada `reduz-acc` (pré-definida em Racket com o nome `foldl`) que abstrai este comportamento.

Função reduz-acc / foldl

```
;; (X Y -> Y) Y Lista(X) -> Y
;; A chamada
;; (reduz-acc f base (list x1 x2 ... xn) produz
;; (f xn ... (f x2 (f x1 base)))
(define (reduz-acc f base lst0)
  (define (iter lst acc)
    (cond
      [(empty? lst) acc]
      [else (iter (rest lst)
                  (f (first lst) acc))]))
  (iter lst0 base))
```

Função `foldl`

Redefinimos as funções em termos de `reduz-acc`

```
(define (tamanho lst)
  (reduz-acc (λ (_ tam) (add1 tam))  0 lst))

(define (soma lst)
  (reduz-acc + 0 lst))

(define (inverte lst)
  (reduz-acc cons empty lst))
```

`foldr` vs `foldl`

`foldr` vs `foldl`

`foldr` e `foldl` produzem o mesmo resultado se a função `f` for associativa.

Quando possível, utilize a função `foldl`, pois ela pode utilizar menos memória.

Não tenha receio de utilizar a função `foldr`, muitas funções ficam mais complicadas, ou não podem ser escritas em termos de `foldl`, como por exemplo, `map` e `filter`.

Referências

Referências

Básicas

- Capítulos 31 e 32 do livro HTDP.
- Seção 1.2 do livro SICP.