Recursão generativa

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Introdução

Vimos anteriormente como explorar a forma como um dado com autorreferência é definido para implementar funções que processam esse tipo de dado:

- Uma autorreferência na definição do tipo do dado sugere uma chamada recursiva na implementação da função

Como nesses casos a chamada recursiva é feita em uma parte da estrutura do dado a recursão é chamada de **recursão estrutural**.

Também vimos anteriormente que a recursão estrutural tem limitações e nem todos os problemas podem ser resolvidos com ela.

Discutimos rapidamente que para esses problemas precisamos utilizar outra abordagem:

- Decompor o problema inicial em subproblemas
- Resolver os subproblemas
- Combinar as soluções dos subproblemas em uma solução para o problema inicial

Se alguns dos subproblemas gerados são do mesmo tipo do problema inicial, podemos usar chamadas recursivas para resolver esses subproblemas, nesses casos, a recursão é chamada de **recursão generativa**.

A recursão generativa é mais poderosa que a recursão estrutural, porém, projetar funções que usam recursão generativa não é um processo tão direto quando funções que usam recursão estrutural. A etapa principal é “gerar” os subproblemas, e isto pode requerer um momento “eureka”.

De qualquer forma, o processo de projeto de funções, com alguns ajustes, também serve para projetar funções com recursão generativa.

Vamos ver alguns exemplos.

Projeto de funções generativas

Exemplo: agrupamento

Dado uma lista de números e um número positivo *n*, projete uma função que agrupe os elementos da lista de entrada em grupos (listas) de *n* elementos.

Exemplo: agrupamento

```
;; (Lista Número) InteiroPositivo -> (Lista (Lista Número))
;;
;; Agrupa os elementos de lst em listas
;; com n elementos. Apenas a última lista
;; do resultado pode ficar com menos que n
;; elementos.
(examples
 (check-equal? (agrupa empty 2) empty)
 (check-equal? (agrupa (list 4 1 5) 1)
               (list (list 4) (list 1) (list 5)))
 (check-equal? (agrupa (list 4 1 5 7 8) 2)
               (list (list 4 1) (list 5 7) (list 8)))
 (check-equal? (agrupa (list 4 1 5 7 8) 3)
               (list (list 4 1 5) (list 7 8))))
```

Exemplo: agrupamento

```
(define (agrupa lst n)
  (cond
    [(empty? lst) empty]
    [(< (length lst) n) (list lst)]
    [else
     (cons (take lst n)
           (agrupa (drop lst n) n))]))
```

Exemplo: quicksort

Defina uma função que ordene uma lista de números usando o algoritmo de ordenação *quicksort*.

Qual é a ideia do *quicksort*?

- Separar os elementos da entrada (não trivial) em duas listas: uma com os menores do que o primeiro e outra com os maiores do que o primeiro
- Ordenar as duas listas recursivamente
- Juntar a ordenação dos menores, com o primeiro e com a ordenação dos maiores.

Exemplo: quicksort

```
;; Lista(Número) -> Lista(Número)
;; Ordena lst em ordem não decrescente usado o quicksort.
(examples
 (check-equal? (quicksort empty)
               empty)
 (check-equal? (quicksort (list 3))
               (list 3))
 (check-equal? (quicksort (list 10 3 -4 5 9))
               (list -4 3 5 9 10))
 (check-equal? (quicksort (list 3 10 0 5 9))
               (list 0 3 5 9 10)))
```

Exemplo: quicksort

```
(define (quicksort lst)
  (cond
    [(empty? lst) empty]
    [else
     (define pivo (first lst))
     (append (quicksort (filter (curry > pivo) lst))
             (list pivo)
             (quicksort (filter (curry < pivo) lst)))]))
```

O que acontece se a lista tiver elementos repetidos? Faça o teste no DrRacket e veja!

O que acontece se alterarmos `>` para `>=`? A função não termina (discutido em sala).

Exemplo: quicksort

Versão corrigida.

```
(define (quicksort lst)
  (cond
    [(empty? lst) empty]
    [else
     (define pivo (first lst))
     (append (quicksort (filter (curry >= pivo) (rest lst)))
             (list pivo)
             (quicksort (filter (curry < pivo) (rest lst))))]))
```

O que precisamos ajustar no processo de projeto de funções?

Na etapa de implementação temos que:

- Determinar se o problema pode ser resolvido diretamente
- Determinar como resolver o problema diretamente
- Definir como decompor o problema em subproblemas que são mais facilmente resolvidos do que o problema original
- Definir combinar as soluções dos subproblemas para resolver o problema inicial
- Argumentar que a função termina para todas as entradas

Referências

Básicas

- Parte 5 do livro HTDP.