Recursão generativa

Programação Funcional

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

```gleam
/// Agrupa os elementos de *lst* em sublistas com *n* elementos. Apenas a
/// última sublista pode ter menos de *n* elementos.
fn agrupa(lst: List(a), n: Int) -> List(List(a)) {
  todo
}
fn agrupa_examples() {
  check.eq(agrupa([], 2), [])
  check.eq(agrupa([4, 1, 5], 1), [[4], [1], [5]])
  check.eq(agrupa([4, 1, 5, 7, 8], 2), [[4, 1], [5, 7], [8]])
  check.eq(agrupa([4, 1, 5, 7, 8], 3), [[4, 1, 5], [7, 8]])
}
```

Exemplo: agrupamento

```
/// Agrupa os elementos de *lst* em sublistas com *n* elementos. Apenas a
/// última sublista pode ter menos de *n* elementos.
fn agrupa(lst: List(a), m: Int) -> List(List(a)) {
  case {
    [] -> todo
    _ -> {
      // decompor em um suproblema
      // resolver recursivamente
      // estender a solução recursiva
      todo
      agrupa(todo, m)
    }
  }
}
```

Exemplo: agrupamento

```gleam
/// Agrupa os elementos de *lst* em sublistas com *m* elementos. Apenas a
/// última sublista pode ter menos de *n* elementos.
fn agrupa(lst: List(a), m: Int) -> List(List(a)) {
  case {
    [] -> []
    _ -> {
      let #(prefixo, suffixo) = list.split(lst, m)
      [prefixo, ..agrupa(suffixo, m)]
    }
  }
}
```

Qual é a equação de recorrência que descreve o tempo de execução da função agrupa?

$T(n) = T(n - m) + m$

Se $m = 1$, então $T(n) = T(n - 1) + 1 = O(n)$

Se $m \geq n$, então $T(n) = T(0) + O(n) = O(n)$

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, se ela não for 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

```gleam
/// Ordena *lst* em ordem não decrescente usando o algoritmo quicksort.
fn quicksort(lst: List(Int)) -> List(Int) {
  todo
}

fn quicksort_examples() {
  check.eq(quicksort([]), [])
  check.eq(quicksort([3]), [3])
  check.eq(quicksort([10, 3, -4, 5, 9]), [-4, 3, 5, 9, 10])
  check.eq(quicksort([3, 10, 0, 5, 9]), [0, 3, 5, 9, 10])
}
```

Exemplo: quicksort

```
/// Ordena *lst* em ordem não decrescente usando o algoritmo quicksort.
fn quicksort(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [pivo, ..resto] -> {
      // decompor em subproblemas
      // resolver os subproblemas recursivamente
      // combinar as soluções recursivas
      todo
    }
  }
}
```

Exemplo: quicksort

```gleam
/// Ordena *lst* em ordem não decrescente usando o algoritmo quicksort.
fn quicksort(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [pivo, ..resto] -> {
      let maiores = list.filter(lst, fn(x) { x >= pivo })
      let menores = list.filter(lst, fn(x) { x < pivo })
      list.append(quicksort(menores), quicksort(maiores))
    }
  }
}
```

A funciona `quicksort` funciona corretamente para qualquer entrada? Não. Se todos os elementos forem iguais, a função será executada recursivamente para a lista de entrada e não terminará a execução.

Exemplo: quicksort

```
/// Ordena *lst* em ordem não decrescente usando o algoritmo quicksort.
fn quicksort(lst: List(Int)) -> List(Int) {
  case lst {
    [] -> []
    [pivo, ..resto] -> {
      let maiores = list.filter(resto, fn(x) { x >= pivo })
      let menores = list.filter(resto, fn(x) { x < pivo })
      list.append(quicksort(menores), [pivo, ..quicksort(maiores)])
    }
  }
}
```

Qual é a equação de recorrência que descreve o tempo de execução da função agrupa?

Depende de como a lista é particionada.

No pior caso

$T(n) = T(n-1) + T(0) + O(n) = O(n^2)$

No melhor caso

$T(n) = 2T(n/2) + O(n) = O(n \lg n)$

O que precisamos ajustar no processo de projeto de funções?

Na etapa de implementação temos que:

- Definir como decompor o problema em subproblemas que são mais facilmente resolvidos do que o problema original
- Definir como 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.