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

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

Exemplo

```gleam
/// Converte a lista *lst* de distâncias relativas
/// para uma lista de distâncias absolutas. O
/// primeiro item da lista representa a distância
/// da origem.
fn relativa_absoluta(
  lst: List(Int)
) -> List(Int) {
  todo
}

fn relativa_absoluta_examples() {
  check.eq(
    relativa_absoluta([50, 40, 70, 30, 30]),
    [50, 90, 160, 190, 220]
  )
}
```

Como resolvemos este problema?

Começando com o modelo!

```gleam
fn relativa_absoluta(
  lst: List(Int)
) -> List(Int) {
  case {
    [] -> todo
    [primeiro, ..resto] -> {
      todo
      primeiro
      relativa_absoluta(resto)
    }
  }
}
```

Exemplo

```
relativa_absoluta([50, 40, 70, 30, 30])        ->   [50, 90, 160, 190, 220]

                  50  [40, 110, 140, 170]
                   |  |
            primeiro  relativa_absoluta(resto)
```

Como combinar o primeiro com a resposta da chamada recursiva para obter a resposta da função?

Juntando `50` com a soma de `50` a cada elemento de `[40, 110, 140, 170]`.

```
[primeiro, ..list.map(relativa_absoluta(resto), int.add(_, primeiro))]
```

Exemplo

```gleam
fn relativa_absoluta(
  lst: List(Int)
) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] -> [
      primeiro,
      ..list.map(
          relativa_absoluta(resto),
          int.add(_, primeiro),
      )
    ]
  }
}
```

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 relativa do ponto com a distância absoluta do ponto anterior.

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

Exemplo

Como queremos que a função funcione?

```
relativa_absoluta([50, 40, 70, 30, 30])
   -> [50, 90, 160, 190, 220]

[50,
 ..relativa_absoluta([40, 70, 30, 30])]

[50, 90,
 ..relativa_absoluta([70, 30, 30])]

[50, 90, 160,
 ..relativa_absoluta([30, 30])]

...

[50, 90, 160, 190, 220,
 ..relativa_absoluta([])]
```

É possível implementar a função para que ela funcione *exatamente* dessa forma? Não! Por que não? Não sabemos qual é a distância que precisa ser somada no primeiro elemento, ou seja, não temos um contexto da chamada da função.

Como resolver esse problema, isto é, como acessar a distância absoluta anterior para calcular a distância atual? Adicionando um novo parâmetro para a distância absoluta anterior, ou seja, um contexto para a chamada da função.

Exemplo

Como queremos que a função funcione?

```
relativa_absoluta([50, 40, 70, 30, 30])
   -> [50, 90, 160, 190, 220]

[50,
 ..relativa_absoluta([40, 70, 30, 30])]

[50, 90,
 ..relativa_absoluta([70, 30, 30])]

[50, 90, 160,
 ..relativa_absoluta([30, 30])]

...

[50, 90, 160, 190, 220,
 ..relativa_absoluta([])]
```

```
relativa_absoluta([50, 40, 70, 30, 30], 0)

[50,
 ..relativa_absoluta([40, 70, 30, 30], 50)]

[50, 90,
 ..relativa_absoluta([70, 30, 30], 90)]

[50, 90, 160,
 ..relativa_absoluta([30, 30], 160)]

...

[50, 90, 160, 190, 220,
 ..relativa_absoluta([], 220)]
```

Exemplo

```
fn relativa_absoluta(
  lst: List(Int),
  dist: Int,
) -> List(Int) {
  // dist é a distância absoluta até o ponto
  // anterior ao primeiro de lst.
  case lst {
    [] -> []
    [primeiro, ..resto] -> [
      primeiro + dist,
      ..relativa_absoluta(resto,
                          primeiro + dist)
    ]
  }
}
```

Qual o problema com essa função?

Ela precisa de um parâmetro extra, que não faz parte do problema.

Vamos separar em duas funções, uma para o usuário da função e outra com a implementação usando um acumulador.

```
fn relativa_absoluta(lst) {
  relativa_absoluta_loop(lst, 0)
}
fn relativa_absoluta_loop(
  lst: List(Int),
  dist: Int,
) -> List(Int) {
  case lst {
    [] -> []
    [primeiro, ..resto] -> [
      primeiro + dist,
      ..relativa_para_absoluta_loop(resto,
                                    primeiro + dist)
    ]
  }
}
```

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.

```
fn soma(a: Int, b: Int) -> Int {
  case b {
    0 -> a
    _ -> 1 + soma(a, b - 1)
  }
}
```

```
fn soma_alt(a: Int, b: Int) -> Int {
  case b {
    0 -> a
    _ -> soma_alt(a + 1, b - 1)
  }
}
```

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

```
fn soma(a: Int, b: Int) -> Int {
  case b {
    0 -> a
    _ -> 1 + soma(a, b - 1)
  }
}
```

```
soma(4, 3)
1 + soma(4, 2)
1 + {1 + soma(4, 1)}
1 + {1 + {1 + soma(4, 0)}}
1 + {1 + {1 + 4}}
1 + {1 + 5}
1 + 6
7
```

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

```
fn soma_alt(a: Int, b: Int) -> Int {
  case b {
    0 -> a
    _ -> soma_alt(a + 1, b - 1)
  }
}
```

```
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 um 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

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

```gleam
/// Conta a quantidade de elementos de *lst*.
fn tamanho(lst: List(a)) -> Int {
  case lst {
    [] -> 0
    [_, ..r] -> 1 + tamanho(r)
  }
}

fn tamanho_examples() {
  check.eq(tamanho([]), 0)
  check.eq(tamanho([4]), 1)
  check.eq(tamanho([7, 1]), 2)
}
```

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 um 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

```gleam
/// Conta a quantidade de elementos de *lst*.
fn tamanho(lst: List(a)) -> Int {
  tamanho_loop(lst, 0)
}

fn tamanho_loop(lst: List(a), acc: Int) -> Int {
  // acc é a quantidade de elementos já processados
  case lst {
    [] -> acc
    [_, ..r] -> tamanho_loop(r, acc + 1)
  }
}
```

Exemplo - soma

```gleam
/// Soma os elementos de *lst*.
fn soma(lst: List(Int)) -> Int {
  case lst {
    [] -> 0
    [p, ..r] -> p + soma(r)
  }
}

fn soma_examples() {
  check.eq(soma([]), 0)
  check.eq(soma([4]), 4)
  check.eq(soma([7, 1]), 8)
}
```

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 um 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

```gleam
/// Soma os elementos de *lst*.
fn soma(lst: List(Int)) -> Int {
  soma_loop(lst, 0)
}

fn soma_loop(lst: List(Int), acc: Int) -> Int {
  // acc é a soma dos elementos já processados
  case lst {
    [] -> acc
    [p, ..r] -> soma_loop(r, acc + p)
  }
}
```

Exemplo - inverte

```gleam
/// Inverte os elementos de *lst*.
fn inverte(lst: List(a)) -> List(a) {
  case lst {
    [] -> []
    [p, ..r] -> list.append(inverte(r), [p])
  }
}

fn inverte_examples() {
  check.eq(inverte([]), [])
  check.eq(inverte([7, 1]), [1, 7])
}
```

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 processado por outra função recursiva (`list.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? `[]`.
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

```
/// Inverte os elementos de *lst*.
fn inverte(lst: List(a)) -> List(a) {
  inverte_loop(lst, [])
}

fn inverte_loop(lst: List(a), acc: List(a)) -> List(a) {
  // acc é a lista dos elementos já analisados em ordem reversa
  case lst {
    [] -> acc
    [p, ..r] -> inverte_loop(r, [p, ..acc])
  }
}
```

Função `fold` (left)

Função `fold` (left)

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

Função `fold` (left)

```
fn tamanho(lst: List(a)) -> Int {
  tamanho_loop(lst, 0)
}

fn tamanho_loop(lst: List(a), acc: Int) -> Int {
  case lst {
    [] -> acc
    [_, ..r] -> tamanho_loop(r, acc + 1)
  }
}
```

Função `fold` (left)

```
fn soma(lst: List(Int)) -> Int {
  soma_loop(lst, 0)
}

fn soma_loop(lst: List(Int), acc: Int) -> Int {
  case lst {
    [] -> acc
    [p, ..r] -> soma_loop(r, acc + p)
  }
}
```

Função `fold` (left)

```gleam
fn inverte(lst: List(a)) -> List(a) {
  inverte_loop(lst, [])
}

fn inverte_loop(lst: List(a), acc: List(a)) -> List(a) {
  case lst {
    [] -> acc
    [p, ..r] -> inverte_loop(r, [p, ..acc])
  }
}
```

Função `fold` (left)

Vamos criar uma função chamada `reduz_acc` (pré-definida em Gleam com o nome `list.fold`) que abstrai este comportamento.

Função reduz_acc / fold

```gleam
fn reduz_acc(lst: List(a), acc: b, f: fn(b, a) -> b) -> b {
  case lst {
    [] -> acc
    [p, ..r] -> reduz_acc(r, f(acc, p), f)
  }
}

fn tamanho(lst: List(a)) -> Int {
  reduz_acc(lst, 0, fn(acc, _) { acc + 1})
}

fn soma(lst: List(Int)) -> Int {
  reduz_acc(lst, 0, fn(acc, e) { acc + e })
}

fn inverte(lst: List(a)) -> List(a) {
  reduz_acc(lst, [], fn(acc, e) { [e, ..acc] })
}
```

`fold_right` vs `fold`

`fold_right` vs `fold`

`fold_right` e `fold` produzem o mesmo resultado se a função `f` for associativa.

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

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

Referências

Referências

Básicas

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