Processamento simultâneo

Programação Funcional

Marco A L Barbosa
malbarbo.pro.br

Departamento de Informática
Universidade Estadual de Maringá

Como implementar uma função que consome dois argumentos e os dois são de tipos com autorreferência? Temos algumas possibilidades:

1) Tratar um dos argumentos como atômico e utilizar o modelo de função para o outro argumento.
2) Processar os dois argumentos de forma sincronizada.
3) Combinar os modelos de funções para os dois argumentos.

Exemplo caso 1: concatenação

Projete uma função que concatene duas listas de números.

1) *Tratar um dos argumentos como atômico e utilizar o modelo de função para o outro argumento.*
2) Processar os dois argumentos de forma sincronizada.
3) Combinar os modelos de funções para os dois argumentos.

Exemplo: concatenação

```gleam
/// Produz uma nova lista com os elementos
/// de *lsta* seguido dos elementos de *lstb*.
fn concatena(
  lsta: List(a),
  lstb: List(a),
) -> List(a) {
  todo
}
```

```gleam
fn concatena_examples() {
  check.eq(concatena([], [10, 4, 6]),
           [10, 4, 6])
  check.eq(concatena([3], [10, 4, 6]),
           [3, 10, 4, 6])
  check.eq(concatena([7, 3], [10, 4, 6]),
           [7, 3, 10, 4, 6])
}
```

Pelo propósito e pelos exemplos, qual dos argumentos pode ser tratado como atômico, isto é, não precisa ser decomposto? `lstb`.

Então usamos o modelo para processar `lsta`.

Exemplo: concatenação

```
/// Produz uma nova lista com os elementos
/// de *lsta* seguido dos elementos de *lstb*.
fn concatena(
  lsta: List(a),
  lstb: List(a),
) -> List(a) {
  case lsta {
    [] -> { todo lstb }
    [primeiro, ..resto] -> {
      todo
      primeiro
      concatena(resto, lstb)
    }
  }
}
```

```
fn concatena_examples() {
  check.eq(concatena([], [10, 4, 6]),
           [10, 4, 6])
  check.eq(concatena([3], [10, 4, 6]),
           [3, 10, 4, 6])
  check.eq(concatena([7, 3], [10, 4, 6]),
           [7, 3, 10, 4, 6])
}
```

Exemplo: concatenação

```gleam
/// Produz uma nova lista com os elementos
/// de *lsta* seguido dos elementos de *lstb*.
fn concatena(
  lsta: List(a),
  lstb: List(a),
) -> List(a) {
  case lsta {
    [] -> lstb
    [primeiro, ..resto] -> {
      todo
      primeiro
      concatena(resto, lstb)
    }
  }
}
```

```gleam
fn concatena_examples() {
  check.eq(concatena([], [10, 4, 6]),
           [10, 4, 6])
  check.eq(concatena([3], [10, 4, 6]),
           [3, 10, 4, 6])
  check.eq(concatena([7, 3], [10, 4, 6]),
           [7, 3, 10, 4, 6])
}
```

Exemplo: concatenação

```gleam
/// Produz uma nova lista com os elementos
/// de *lsta* seguido dos elementos de *lstb*.
fn concatena(
  lsta: List(a),
  lstb: List(a),
) -> List(a) {
  case lsta {
    [] -> lstb
    [primeiro, ..resto] ->
      [primeiro,
        ..concatena(resto, lstb)]
  }
}
```

```gleam
fn concatena_examples() {
  check.eq(concatena([], [10, 4, 6]),
           [10, 4, 6])
  check.eq(concatena([3], [10, 4, 6]),
           [3, 10, 4, 6])
  check.eq(concatena([7, 3], [10, 4, 6]),
           [7, 3, 10, 4, 6])
}
```

Exemplo caso 2: soma ponderada

Projete uma função que calcule a soma ponderada a partir de uma lista de números e uma lista de pesos.

1) Tratar um dos argumentos como atômico e utilizar o modelo de função para o outro argumento.
2) *Processar os dois argumentos de forma sincronizada.*
3) Combinar os modelos de funções para os dois argumentos.

Exemplo: soma ponderada

```gleam
/// Calcula a soma ponderada dos valores de
/// *lst* considerando que cada elemento de
/// *lst* tem como peso o elemento correspon-
/// dente em *pesos*. Devolve Error(Nil) se
/// *lst* e *pesos* tem quantidade diferente
/// de elementos.
fn soma_ponderada(
  lst: List(Float),
  pesos: List(Float),
) -> Result(Float, Nil) {
  todo
}
```

```gleam
fn soma_ponderada_examples() {
  check.eq(soma_ponderada([], []), Ok(0.0))
  check.eq(soma_ponderada([], [1.0]), Error(Nil))
  check.eq(soma_ponderada([1.0], []), Error(Nil))
  check.eq(soma_ponderada([4.0], [2.0]), Ok(8.0))
  check.eq(
    soma_ponderada([3.0, 4.0], [5.0, 2.0]),
    Ok(23.0))
  check.eq(
    soma_ponderada([5.0, 3.0, 4.0], [1.0, 5.0, 2.0]),
    Ok(28.0))
}
```

Existem uma correspondência entre os elementos de `lst` e `pesos`. Usamos essa correspondência para o caso base (listas vazias) e para a chamada recursiva (para os restos das listas).

Exemplo: soma ponderada

```gleam
/// Calcula a soma ponderada dos valores de
/// *lst* considerando que cada elemento de
/// *lst* tem como peso o elemento correspon-
/// dente em *pesos*. Devolve Error(Nil) se
/// *lst* e *pesos* tem quantidade diferente
/// de elementos.
fn soma_ponderada(
  lst: List(Float),
  pesos: List(Float),
) -> Result(Float, Nil) {
  case lst, pesos {
    [], [] -> todo
    [valor, ..rlst], [peso, ..rpesos] -> {
      todo valor peso
           soma_pondera(rlst, rpesos)
    }
    _, _ -> todo
  }
}
```

```gleam
fn soma_ponderada_examples() {
  check.eq(soma_ponderada([], []), Ok(0.0))
  check.eq(soma_ponderada([], [1.0]), Error(Nil))
  check.eq(soma_ponderada([1.0], []), Error(Nil))
  check.eq(soma_ponderada([4.0], [2.0]), Ok(8.0))
  check.eq(
    soma_ponderada([3.0, 4.0], [5.0, 2.0]),
    Ok(23.0))
  check.eq(
    soma_ponderada([5.0, 3.0, 4.0], [1.0, 5.0, 2.0]),
    Ok(28.0))
}
```

Exemplo: soma ponderada

```gleam
/// Calcula a soma ponderada dos valores de
/// *lst* considerando que cada elemento de
/// *lst* tem como peso o elemento correspon-
/// dente em *pesos*. Devolve Error(Nil) se
/// *lst* e *pesos* tem quantidade diferente
/// de elementos.
fn soma_ponderada(
  lst: List(Float),
  pesos: List(Float),
) -> Result(Float, Nil) {
  case lst, pesos {
    [], [] -> Ok(0.0)
    [valor, ..rlst], [peso, ..rpesos] -> {
      todo valor peso
           soma_pondera(rlst, rpesos)
    }
    _, _ -> Error(Nil)
  }
}
```

```gleam
fn soma_ponderada_examples() {
  check.eq(soma_ponderada([], []), Ok(0.0))
  check.eq(soma_ponderada([], [1.0]), Error(Nil))
  check.eq(soma_ponderada([1.0], []), Error(Nil))
  check.eq(soma_ponderada([4.0], [2.0]), Ok(8.0))
  check.eq(
    soma_ponderada([3.0, 4.0], [5.0, 2.0]),
    Ok(23.0))
  check.eq(
    soma_ponderada([5.0, 3.0, 4.0], [1.0, 5.0, 2.0]),
    Ok(28.0))
}
```

Exemplo: soma ponderada

```gleam
/// Calcula a soma ponderada dos valores de
/// *lst* considerando que cada elemento de
/// *lst* tem como peso o elemento correspon-
/// dente em *pesos*. Devolve Error(Nil) se
/// *lst* e *pesos* tem quantidade diferente
/// de elementos.
fn soma_ponderada(
  lst: List(Float),
  pesos: List(Float),
) -> Result(Float, Nil) {
  case lst, pesos {
    [], [] -> Ok(0.0)
    [valor, ..rlst], [peso, ..rpesos] ->
      case soma_pondera(rlst, rpesos) {
        Ok(r) -> Ok(valor *. peso +. r)
        _ -> Error(Nil)
      }
    _, _ -> Error(Nil)
  }
}
```

```gleam
fn soma_ponderada_examples() {
  check.eq(soma_ponderada([], []), Ok(0.0))
  check.eq(soma_ponderada([], [1.0]), Error(Nil))
  check.eq(soma_ponderada([1.0], []), Error(Nil))
  check.eq(soma_ponderada([4.0], [2.0]), Ok(8.0))
  check.eq(
    soma_ponderada([3.0, 4.0], [5.0, 2.0]),
    Ok(23.0))
  check.eq(
    soma_ponderada([5.0, 3.0, 4.0], [1.0, 5.0, 2.0]),
    Ok(28.0))
}
```

Exemplo caso 3: prefixo

Dado duas listas `lsta` e `lstb`, defina uma função que verifique se `lsta` é prefixo de `lstb`, isto é `lstb` começa com `lsta`.

1) Tratar um dos argumentos como atômico e utilizar o modelo de função para o outro argumento.
2) Processar os dois argumentos de forma sincronizada.
3) *Combinar os modelos de funções para os dois argumentos.*

Exemplo: prefixo

```gleam
/// Devolve True se *lsta* é prefixo de *lstb*,
/// isto é, os elementos de *lsta* aparecem no
/// início de *lstb*. Devolve False, caso
/// contrário.
fn prefixo(
  lsta: List(a),
  lstb: List(a),
) -> Bool {
  todo
}
```

```gleam
fn prefixo_examples() {
  // [], []
  check.eq(prefixo([], []), True)
  // [], [_, ..]
  check.eq(prefixo([], [3, 4]), True)
  // [_, ..], []
  check.eq(prefixo([3, 4], []), False)
  // [_, ..], [_, ..]
  check.eq(prefixo([3, 4], [3, 4]), True)
  check.eq(prefixo([3, 4], [3, 4, 6, 8]), True)
  check.eq(prefixo([3, 4], [3, 5]), False)
  check.eq(prefixo([3, 4, 5], [3, 4]), False)
}
```

Exemplo: prefixo

```gleam
/// Devolve True se *lsta* é prefixo de *lstb*,
/// isto é, os elementos de *lsta* aparecem no
/// início de *lstb*. Devolve False, caso
/// contrário.
fn prefixo(
  lsta: List(a),
  lstb: List(a),
) -> Bool {
  case lsta, lstb {
    [], [] -> todo
    [], [b, ..rb] -> { todo b rb }
    [a, ..ra], [] -> { todo a ra }
    [a, ..ra], [b, ..rb] -> {
      todo
      a
      b
      prefixo(ra, rb)
    }
  }
}
```

```gleam
fn prefixo_examples() {
  // [], []
  check.eq(prefixo([], []), True)
  // [], [_, ..]
  check.eq(prefixo([], [3, 4]), True)
  // [_, ..], []
  check.eq(prefixo([3, 4], []), False)
  // [_, ..], [_, ..]
  check.eq(prefixo([3, 4], [3, 4]), True)
  check.eq(prefixo([3, 4], [3, 4, 6, 8]), True)
  check.eq(prefixo([3, 4], [3, 5]), False)
  check.eq(prefixo([3, 4, 5], [3, 4]), False)
}
```

Exemplo: prefixo

```gleam
/// Devolve True se *lsta* é prefixo de *lstb*,
/// isto é, os elementos de *lsta* aparecem no
/// início de *lstb*. Devolve False, caso
/// contrário.
fn prefixo(
  lsta: List(a),
  lstb: List(a),
) -> Bool {
  case lsta, lstb {
    [], [] -> True
    [], _ -> True
    _, [] -> False
    [a, ..ra], [b, ..rb] -> {
      todo
      a
      b
      prefixo(ra, rb)
    }
  }
}
```

```gleam
fn prefixo_examples() {
  // [], []
  check.eq(prefixo([], []), True)
  // [], [_, ..]
  check.eq(prefixo([], [3, 4]), True)
  // [_, ..], []
  check.eq(prefixo([3, 4], []), False)
  // [_, ..], [_, ..]
  check.eq(prefixo([3, 4], [3, 4]), True)
  check.eq(prefixo([3, 4], [3, 4, 6, 8]), True)
  check.eq(prefixo([3, 4], [3, 5]), False)
  check.eq(prefixo([3, 4, 5], [3, 4]), False)
}
```

Exemplo: prefixo

```gleam
/// Devolve True se *lsta* é prefixo de *lstb*,
/// isto é, os elementos de *lsta* aparecem no
/// início de *lstb*. Devolve False, caso
/// contrário.
fn prefixo(
  lsta: List(a),
  lstb: List(a),
) -> Bool {
  case lsta, lstb {
    [], [] -> True
    [], _ -> True
    _, [] -> False
    [a, ..ra], [b, ..rb] ->
      a == b && prefixo(ra, rb)
  }
}
```

```gleam
fn prefixo_examples() {
  // [], []
  check.eq(prefixo([], []), True)
  // [], [_, ..]
  check.eq(prefixo([], [3, 4]), True)
  // [_, ..], []
  check.eq(prefixo([3, 4], []), False)
  // [_, ..], [_, ..]
  check.eq(prefixo([3, 4], [3, 4]), True)
  check.eq(prefixo([3, 4], [3, 4, 6, 8]), True)
  check.eq(prefixo([3, 4], [3, 5]), False)
  check.eq(prefixo([3, 4, 5], [3, 4]), False)
}
```

Exemplo: prefixo

```gleam
/// Devolve True se *lsta* é prefixo de *lstb*,
/// isto é, os elementos de *lsta* aparecem no
/// início de *lstb*. Devolve False, caso
/// contrário.
fn prefixo(
  lsta: List(a),
  lstb: List(a),
) -> Bool {
  case lsta, lstb {
    [], _ -> True
    _, [] -> False
    [a, ..ra], [b, ..rb] ->
      a == b && prefixo(ra, rb)
  }
}
```

```gleam
fn prefixo_examples() {
  // [], []
  check.eq(prefixo([], []), True)
  // [], [_, ..]
  check.eq(prefixo([], [3, 4]), True)
  // [_, ..], []
  check.eq(prefixo([3, 4], []), False)
  // [_, ..], [_, ..]
  check.eq(prefixo([3, 4], [3, 4]), True)
  check.eq(prefixo([3, 4], [3, 4, 6, 8]), True)
  check.eq(prefixo([3, 4], [3, 5]), False)
  check.eq(prefixo([3, 4, 5], [3, 4]), False)
}
```

Exemplo: k-ésimo

Defina uma função que encontre o k-ésimo elemento de uma lista.

1) Tratar um dos argumentos como atômico e utilizar o modelo de função para o outro argumento.
2) Processar os dois argumentos de forma sincronizada.
3) *Combinar os modelos de funções para os dois argumentos.*

Exemplo: k-ésimo

```gleam
/// Devolve o elemento na posição *i* de *lst*
/// (indexado a partir de 0). Devolve Error(Nil)
/// se *i* é negativo ou é maior igual a
/// quantidade de elemento de *lst*.
fn lista_get(
  lst: List(a),
  k: Int,
) -> Result(a, Nil) {
}
```

```gleam
fn lista_get_examples() {
  // [], 0
  check.eq(lista_get([], 0), Error(Nil))
  // [], > 0
  check.eq(lista_get([], 2), Error(Nil))
  // [_, ..], 0
  check.eq(lista_get([3, 2, 8], 0), Ok(3))
  // [_, ..], > 0
  check.eq(lista_get([3, 2, 8, 10], 2), Ok(8))
  check.eq(lista_get([3, 2, 8, 10], 4), Error(Nil))
  // [], < 0
  check.eq(lista_get([], -1), Error(Nil))
  // [_, ..], < 0
  check.eq(lista_get([1, 2], -3), Error(Nil))
}
```

Exemplo: k-ésimo

```gleam
/// Devolve o elemento na posição *i* de *lst*
/// (indexado a partir de 0). Devolve Error(Nil)
/// se *i* é negativo ou é maior igual a
/// quantidade de elemento de *lst*.
fn lista_get(
  lst: List(a),
  k: Int,
) -> Result(a, Nil) {
  case lst, k {
    [], 0 -> todo
    [], _ if k > 0 -> { todo k }
    [p, ..resto], 0 -> { todo p resto }
    [p, ..resto], _ if k > 0 -> {
      todo p lista_get(resto, k - 1)
    }
    _, _ -> todo
  }
}
```

```gleam
fn lista_get_examples() {
  // [], 0
  check.eq(lista_get([], 0), Error(Nil))
  // [], > 0
  check.eq(lista_get([], 2), Error(Nil))
  // [_, ..], 0
  check.eq(lista_get([3, 2, 8], 0), Ok(3))
  // [_, ..], > 0
  check.eq(lista_get([3, 2, 8, 10], 2), Ok(8))
  check.eq(lista_get([3, 2, 8, 10], 4), Error(Nil))
  // [], < 0
  check.eq(lista_get([], -1), Error(Nil))
  // [_, ..], < 0
  check.eq(lista_get([1, 2], -3), Error(Nil))
}
```

Exemplo: *k*-ésimo

```gleam
/// Devolve o elemento na posição *i* de *lst*
/// (indexado a partir de 0). Devolve Error(Nil)
/// se *i* é negativo ou é maior igual a
/// quantidade de elemento de *lst*.
fn lista_get(
  lst: List(a),
  k: Int,
) -> Result(a, Nil) {
  case lst, k {
    [], 0 -> Error(Nil)
    [], _ if k > 0 -> Error(Nil)
    [p, ..resto], 0 -> { todo p resto }
    [p, ..resto], _ if k > 0 -> {
      todo p lista_get(resto, k - 1)
    }
    _, _ -> Error(Nil)
  }
}
```

```gleam
fn lista_get_examples() {
  // [], 0
  check.eq(lista_get([], 0), Error(Nil))
  // [], > 0
  check.eq(lista_get([], 2), Error(Nil))
  // [_, ..], 0
  check.eq(lista_get([3, 2, 8], 0), Ok(3))
  // [_, ..], > 0
  check.eq(lista_get([3, 2, 8, 10], 2), Ok(8))
  check.eq(lista_get([3, 2, 8, 10], 4), Error(Nil))
  // [], < 0
  check.eq(lista_get([], -1), Error(Nil))
  // [_, ..], < 0
  check.eq(lista_get([1, 2], -3), Error(Nil))
}
```

Exemplo: *k*-ésimo

```gleam
/// Devolve o elemento na posição *i* de *lst*
/// (indexado a partir de 0). Devolve Error(Nil)
/// se *i* é negativo ou é maior igual a
/// quantidade de elemento de *lst*.
fn lista_get(
  lst: List(a),
  k: Int,
) -> Result(a, Nil) {
  case lst, k {
    [], 0 -> Error(Nil)
    [], _ if k > 0 -> Error(Nil)
    [p, ..], 0 -> Ok(p)
    [_, ..resto], _ if k > 0 -> {
      lista_get(resto, k - 1)
    }
    _, _ -> Error(Nil)
  }
}
```

```gleam
fn lista_get_examples() {
  // [], 0
  check.eq(lista_get([], 0), Error(Nil))
  // [], > 0
  check.eq(lista_get([], 2), Error(Nil))
  // [_, ..], 0
  check.eq(lista_get([3, 2, 8], 0), Ok(3))
  // [_, ..], > 0
  check.eq(lista_get([3, 2, 8, 10], 2), Ok(8))
  check.eq(lista_get([3, 2, 8, 10], 4), Error(Nil))
  // [], < 0
  check.eq(lista_get([], -1), Error(Nil))
  // [_, ..], < 0
  check.eq(lista_get([1, 2], -3), Error(Nil))
}
```

Exemplo: *k*-ésimo

```gleam
/// Devolve o elemento na posição *i* de *lst*
/// (indexado a partir de 0). Devolve Error(Nil)
/// se *i* é negativo ou é maior igual a
/// quantidade de elemento de *lst*.
fn lista_get(
  lst: List(a),
  k: Int,
) -> Result(a, Nil) {
  case lst, k {
    [p, ..], 0 -> Ok(p)
    [_, ..resto], _ if k > 0 ->
      lista_get(resto, k - 1)
    _, _ -> Error(Nil)
  }
}
```

```gleam
fn lista_get_examples() {
  // [], 0
  check.eq(lista_get([], 0), Error(Nil))
  // [], > 0
  check.eq(lista_get([], 2), Error(Nil))
  // [_, ..], 0
  check.eq(lista_get([3, 2, 8], 0), Ok(3))
  // [_, ..], > 0
  check.eq(lista_get([3, 2, 8, 10], 2), Ok(8))
  check.eq(lista_get([3, 2, 8, 10], 4), Error(Nil))
  // [], < 0
  check.eq(lista_get([], -1), Error(Nil))
  // [_, ..], < 0
  check.eq(lista_get([1, 2], -3), Error(Nil))
}
```

Referências

Básicas

- Capítulo 23 HTDP
- Vídeos 2 one-of