Wstęp do Programowania potok funkcyjny

Marcin Kubica

2010/2011

Outline

Wstęp

- Kilka powtarzających się schematów procedur przetwarzających listy.
- Procedury wyższych rzędów ujmujące te schematy.
- Istnieje zestaw standardowych procedur wyższych rzędów przeznaczonych do przetwarzania list.
- Moduł `List`

fold_left

- Przeglądamy kolejne elementy listy (od lewej do prawej).
- Przechowujemy wynik pośredni (dla przejrzanych elementów).
- Zaczynamy od wyniku dla listy pustej.
- W każdym kroku „dorzucamy” jeden element listy do wyniku pośredniego.
  Kumulujemy wpływ kolejnych elementów listy na wynik.
- Po przejrzeniu wszystkich elementów listy mamy gotowy wynik.
- Intuicja: kula śnieżna, zwijanie dywanu.

fold_left

Definition (fold_left)

Parametry:

- wynik dla pustej listy (a),
- procedura *kumulująca* wpływ kolejnych elementów na wynik (f),
- lista do przetworzenia (l).
- fold_left f a $[x_1; x_2; \ldots; x_n] = f\ (\ldots (f\ (f\ a\ x_1)\ x_2) \ldots)\ x_n$

```
let rec fold_left f a l =
  match l with
    [] -> a |
    h::t -> fold_left f (f a h) t;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

(Rekurencja ogonowa)

fold_left

Example

Suma elementów listy:

```
let sum l = fold_left (+) 0 l;;
val sum : int list -> int = <fun>
```

Example

Iloczyn elementów listy:

```
let prod l = fold_left ( * ) 1 l;;
val prod : int list -> int = <fun>
```

fold_left

Example

Suma elementów listy:

```
let sum l = fold_left (+) 0 l;;
val sum : int list -> int = <fun>
```

Example

Iloczyn elementów listy:

```
let prod l = fold_left ( * ) 1 l;;
val prod : int list -> int = <fun>
```

fold_left

Example

Długość listy:

```
let length l = fold_left (fun x _ -> x + 1) 0 l;;
val length : 'a list -> int = <fun>
```

Wartości elementów nie mają znaczenia, tylko ich liczba.

Example

Odwracanie listy:

```
let rev l = fold_left (fun a h -> h::a) [] l;;
val rev : 'a list -> 'a list = <fun>
```

Kumulowany wynik może być złożoną wartością.

fold_left

Example

Długość listy:

```
let length l = fold_left (fun x _ -> x + 1) 0 l;;
val length : 'a list -> int = <fun>
```

Wartości elementów nie mają znaczenia, tylko ich liczba.

Example

Odwracanie listy:

```
let rev l = fold_left (fun a h -> h::a) [] l;;
val rev : 'a list -> 'a list = <fun>
```

Kumulowany wynik może być złożoną wartością.

fold_left2

Definition (fold_left2)

Wersja fold_left dla dwóch list tej samej długości:

```
let rec fold_left2 f a l1 l2 =
  match (l1, l2) with
    ([], [])           -> a |
    (h1::t1, h2::t2) -> fold_left2 f (f a h1 h2) t1 t2 |
    _                  -> failwith "Error ...";;
val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a ->
                          'b list -> 'c list -> 'a = <fun>
```

fold_left2

Example

Iloczyn skalarny wektorów reprezentowanych jako listy:

```
let iloczyn_skalarny l1 l2 =
  fold_left2 (fun a x y -> a + x * y) 0 l1 l2;;
val iloczyn_skalarny : int list -> int list -> int = <fun>
```

fold_right

Definition

- `fold_right` działa analogicznie do `fold_left`,
- `fold_right` przetwarza elementy od prawej do lewej.
- Uwaga: inna kolejność argumentów.
- `fold_right` $f\ [x_1; x_2; \ldots; x_n]\ a =$
  $f\ x_1\ (\ldots (f\ x_{n-1}\ (f\ x_n\ a))\ldots)$

```
let rec fold_right f l a =
  match l with
    [] -> a |
    h::t -> f h (fold_right f t a);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Rekurencja nieogonowa.

fold_right

Example

Suma i iloczyn elementów:

```
let sum l = fold_right (+) l 0;;
val sum : int list -> int = <fun>

let prod l = fold_right ( * ) l 1;;
val prod : int list -> int = <fun>
```

Example

Długość listy:

```
let length l = fold_right (fun _ x -> x + 1) l 0;;
val length : 'a list -> int = <fun>
```

Mniej efektywne pamięciowo niż za pomocą `fold_left`.

fold_right

Example

`flatten` przekształca listę list sklejając listy składowe.

```
let flatten l = fold_right (@) l [];;
val flatten : 'a list list -> 'a list = <fun>

flatten [[1;2]; []; [3]; []; [4;5;6]];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Dlaczego rozwiązanie tego zadania za pomocą `fold_left` byłoby mniej efektywne?

`fold_left` i `fold_right`

- `fold_left` i `fold_right` to najbardziej elementarne spośród standardowych procedur wyższych rzędów przetwarzających listy.
- `fold_right` można zdefiniować za pomocą `fold_left` i odwrotnie.
- `fold_left` jest bardziej elementarna — definiując `fold_left` za pomocą `fold_right` tracimy ogonowość.

fold_left i fold_right

Example

fold_right za pomocą fold_left:

```
let fold_right f l a =
  let rev = fold_left (fun a h -> h::a) []
  in fold_left (fun x h -> f h x) a (rev l);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Wystarczy odwrócić listę i przetwarzać elementy od lewej do prawej.

fold_left i fold_right

Example

`fold_left` za pomocą `fold_right`:

- Chcemy obliczyć:
  `fold_left` $f\ a\ [x_1; x_2; \ldots; x_n] = f\ (\ldots (f\ (f\ a\ x_1)\ x_2) \ldots)\ x_n$
- Przetwarzamy elementy w kolejności: $x_n, x_{n-1}, \ldots, x_1$.
- Kumulowane wartości to procedury postaci:

$$
\begin{array}{lcl}
\texttt{fun}\ x & \rightarrow & x \\
\texttt{fun}\ x & \rightarrow & f\ x\ x_n \\
\texttt{fun}\ x & \rightarrow & f\ (f\ x\ x_{n-1})\ x_n \\
 & \vdots & \\
\texttt{fun}\ x & \rightarrow & f\ (\ldots (f\ x\ x_2) \ldots)\ x_n \\
\texttt{fun}\ x & \rightarrow & f\ (\ldots (f\ (f\ x\ x_1)\ x_2) \ldots)\ x_n
\end{array}
$$

fold_left i fold_right

Example

- Kumulowane procedury przekształcają wynik dla początkowego fragmentu listy $[x_1; x_2; \ldots; x_{i-1}]$ w wynik dla całej listy.
- Na koniec, wystarczy przekazać wynik dla pustej listy (a).

```
let fold_left f a l =
  fold_right (fun h p -> (function x -> p (f x h)))
             l (fun x -> x) a;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

fold_right2

Definition

fold_right ma swój odpowiednik do przetwarzania dwóch list równej długości:

```
let rec fold_right2 f l1 l2 a =
  match (l1, l2) with
    ([], [])           -> a |
    (h1::t1, h2::t2) -> f h1 h2 (fold_right2 f t1 t2 a) |
    _                  -> failwith "Error ...";;
val fold_right2 : ('a -> 'b -> 'c -> 'c) ->
                  'a list -> 'b list -> 'c -> 'c = <fun>
```

map

Definition

Procedura `map` to schemat, w którym każdy element listy jest przetwarzany niezależnie:

$$\texttt{map}\ f\ [x_1; x_2; \ldots; x_n] = [f\ x_1; f\ x_2; \ldots; f\ x_n]$$

```
let map f l =
  fold_right (fun h t -> (f h)::t) l [];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

map

Example

Przykłady użycia map:

```
map abs [6; -9; 4; -2; 0];;
- : int list = [6; 9; 4; 2; 0]

map rev [[1;2]; []; [3]; []; [4;5;6]];;
- : int list list = [[2; 1]; []; [3]; []; [6; 5; 4]]
```

map

Definition

Procedura map ma swój odpowiednik do przetwarzania dwóch list równej długości:

```
let map2 f l1 l2 =
  fold_right2 (fun h1 h2 t -> (f h1 h2)::t) l1 l2 [];;
val map2 : ('a -> 'b -> 'c) ->
                    'a list -> 'b list -> 'c list = <fun>
```

Example

Suma wektorów reprezentowanych jako listy:

```
let suma_wektorow l1 l2 =
  map2 (+) l1 l2;;
val suma_wektorow : int list -> int list -> int list = <fun>
```

map

Definition

Procedura map ma swój odpowiednik do przetwarzania dwóch list równej długości:

```
let map2 f l1 l2 =
  fold_right2 (fun h1 h2 t -> (f h1 h2)::t) l1 l2 [];;
val map2 : ('a -> 'b -> 'c) ->
                    'a list -> 'b list -> 'c list = <fun>
```

Example

Suma wektorów reprezentowanych jako listy:

```
let suma_wektorow l1 l2 =
  map2 (+) l1 l2;;
val suma_wektorow : int list -> int list -> int list = <fun>
```

filter

Definition

- Procedura `filter` realizuje dualny schemat do `map`.
- Elementy listy nie zmieniają się, natomiast część z nich jest usuwana.
- Parametrem jest predykat określający jakie wartości mają zostać.

```
let filter p l =
  fold_right (fun h t -> if p h then h::t else t) l [];;
val filter : ('a -> bool) -> 'a list -> 'a list = <fun>
```

filter

Example

Sito Eratostenesa:

```
let sito l =
  match l with
    [] -> [] |
    h::t -> filter (fun x -> x mod h <> 0) t;;
val sito : int list -> int list = <fun>

let gen n =
  let rec pom acc k =
    if k < 2 then acc else pom (k::acc) (k-1)
  in pom [] n;;
val gen : int -> int list = <fun>
```

filter

Example

```
let eratostenes n =
  let rec erat l =
    if l = [] then [] else (List.hd l)::(erat (sito l))
  in erat (gen n);;
val eratostenes : int -> int list = <fun>

eratostenes 42;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41]
```

Procedury wyższych rzędów przetwarzające drzewa

- Nie tylko listy przetwarzamy rekurencyjnie.
- Drzewa — typowa struktura danych przetwarzana rekurencyjnie.
- Drzewa dowolnego stopnia. W węzłach przechowujemy wartości typu 'a.

  `type` α `tree = Node of` α `*` α `tree list;;`
- Zdefiniujemy odpowiedniki procedur: `fold_right` (-up), `map`.

fold_tree

Definition

- Drzewa przetwarzamy od liści do korzenia (ang. *bottom-up*). W korzeniu uzyskujemy jeden skumulowny wynik.
- W pojedynczym kroku kumulujemy wartość z danego węzła i wartości obliczone dla jego poddrzew.

```
let rec fold_tree f (Node (x, l)) =
  f x (map (fold_tree f) l);;
val fold_tree : ('a -> 'b list -> 'b) -> 'a tree -> 'b = <fun>
```

fold_tree

Example

Wysokość drzewa:

```
let height t =
  let maxl l = fold_left max 0 l
  in
    fold_tree (fun _ l -> maxl l + 1) t;;
val height : 'a tree -> int = <fun>
```

map_tree

Definition

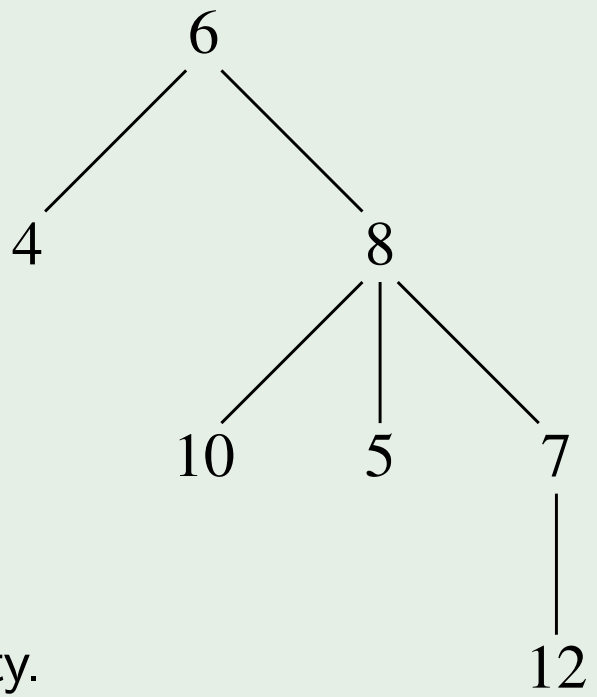
Odpowiednik map dla drzew:

```
let map_tree f t =
  fold_tree (fun x l -> Node (f x, l)) t;; val map_tree :
('a -> 'b) -> 'a tree -> 'b tree = <fun>
```

fold_tree

Example

- Wartość w węźle drzewa (typu `int tree`) jest *widoczna*, jeżeli na ścieżce do korzenia nie ma większej wartości.
- Liczba w korzeniu jest zawsze widoczna, a liczby mniejsze od niej nie są nigdy widoczne.
- Ile liczb jest widocznych?



- Przykład: 4 widoczne elementy.

fold_tree

Example

- Przetwarzamy drzewo od liści do korzenia.
- Nie wiemy jakie będzie maksimum na ścieżce do korzenia.
- Kumulowana wartość, to procedura, a nie liczba.
  Po podaniu maksimum na ścieżce do korzenia, zwraca liczbę widocznych elementów.

```
let widoczne t =
  let merge x l k =
    if x < k then
      fold_left (fun a h -> a + h k) 0 l
    else
      fold_left (fun a h -> a + h x) 0 l + 1
  in
    fold_tree merge t (-1);;
val widoczne : int tree -> int = <fun>
```