Wstęp do Programowania potok funkcyjny

Marcin Kubica

2010/2011

Outline

Dwustronne kolejki ze scalaniem i odwracaniem

Example

Dwustronne kolejki + sklejanie + odwracanie.

```
module type QUEUE = sig
  type 'a queue
  exception EmptyQueue
  val init : unit -> 'a queue
  val is_empty : 'a queue -> bool
  val put_first : 'a queue -> 'a -> unit
  val put_last : 'a queue -> 'a -> unit
  val first : 'a queue -> 'a
  val last : 'a queue -> 'a
  val remove_first : 'a queue -> unit
  val remove_last : 'a queue -> unit
  val merge : 'a queue -> 'a queue -> unit
  val rev : 'a queue -> unit
end;;
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Technika atrap i strażników.
- Lista dwukierunkowa, bez określania kierunków dowiązań.
- Atrapy na końcach.
- Lista, to para dowiązań do atrap.

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Opcjonalna wartość — atrapa nie ma wartości:

```
type 'a opt = Null | Val of 'a
```

- Elementy listy:

```
type 'a elem = {
  mutable l1 : 'a elem;
  mutable l2 : 'a elem;
  v : 'a opt
}
```

- Kolejka:

```
type 'a queue = {
  mutable front : 'a elem;
  mutable back : 'a elem
}
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Gdy kolejka niespodziewanie pusta:
  `exception EmptyQueue`
- Wartość elementu:

```
let value e =
  match e.v with
    Val x -> x |
    Null  -> raise EmptyQueue
```

- Predykat sprawdzający czy kolejka jest pusta:

```
let is_empty q =
  let f = q.front
  and b = q.back
  in (f.l2 == b)
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Konstruktor pustej kolejki:

```
let init () =
  let rec g1 = { l1 = g1; l2 = g2; v = Null}
  and     g2 = { l1 = g2; l2 = g1; v = Null}
  in { front = g1; back = g2 }
```

- Wstawia nowy element z wartością x pomiędzy elementy p i q:

```
let put_between p q x =
  let r = { l1 = p; l2 = q; v = Val x }
  in begin
    if p.l1 == q then p.l1 <- r else p.l2 <- r;
    if q.l1 == p then q.l1 <- r else q.l2 <- r
  end
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Wstawienie na początek:

```
let put_first q x =
  let f = q.front
  in put_between f f.l2 x
```

- Wstawienie na koniec:

```
let put_last q x =
  let b = q.back
  in put_between b b.l2 x
```

- Pierwszy element:

```
let first q = value q.front.l2
```

- Ostatni element:

```
let last q = value q.back.l2
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Usuwa element (nie będący atrapą):

```
let remove e =
  let e1 = e.l1
  and e2 = e.l2
  in begin
    if e1.l1 == e then e1.l1 <- e2 else e1.l2 <- e2;
    if e2.l1 == e then e2.l1 <- e1 else e2.l2 <- e1
  end
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Usunięcie pierwszego elementu:

```
let remove_first q =
  if is_empty q then raise EmptyQueue
  else remove q.front.l2
```

- Usunięcie ostatniego elementu:

```
let remove_last q =
  if is_empty q then raise EmptyQueue
  else remove q.back.l2
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Sklejenie dwóch kolejek, druga kolejka staje się pusta:

```
let merge q1 q2 =
  let f1 = q1.front and b1 = q1.back
  and f2 = q2.front and b2 = q2.back
  in  let e1 = b1.l2 and e2 = f2.l2
      in begin
        if e1.l1 == b1 then e1.l1 <- e2
        else e1.l2 <- e2;
        if e2.l1 == f2 then e2.l1 <- e1
        else e2.l2 <- e1;
        q1.back <- b2;
        f2.l2 <- b1;
        b1.l2 <- f2;
        q2.back <- b1
      end
```

Dwustronne kolejki ze scalaniem i odwracaniem

Example

- Odwrócenie kolejki:

```
let rev q =
  let f = q.front
  and b = q.back
  in begin
    q.front <- b;
    q.back <- f
  end
```

W oczywisty sposób, złożoność czasowa wszystkich operacji to: $\Theta(1)$.

Drzewa Find-Union

Example

- Struktura danych do reprezentowania relacji równoważności oraz jej klas abstrakcji.
- Alternatywnie: graf nieskierowany i spójne składowe.
- Interfejs:
  - typ reprezentujący elementy i klasy,
  - konstruktor elementów i par,
  - sprawdzenie równoważności,
  - reprezentant i elementy klasy abstrakcji,
  - liczba klas abstrakcji.

Drzewa Find-Union

Example

```
module type FIND_UNION = sig
  type 'a set
  val make_set : 'a -> 'a set
  val find : 'a set -> 'a
  val equivalent : 'a set -> 'a set -> bool
  val union : 'a set -> 'a set -> unit
  val elements : 'a set -> 'a list
  val n_of_sets : unit-> int
end;;
```

Drzewa Find-Union

Example

- α `set` — elementy klas abstrakcji etykietowane wartościami typu α.
  Różne (w sensie ==) wartości sklejamy w jedną klasę abstrakcji.
- Każde wywołanie `make_set` tworzy nowy element z podaną etykietą.
  Nawet jeśli dwa razy wywołamy z tą samą etykietą.
  Etykiety powinny być unikalne.
- Elementy x i y są w tej samej klasie abstrakcji gdy `find` $x =$ `find` y.
  (Chyba, że etykiety nie są unikalne.)

Drzewa Find-Union

Example

- Implementacja takiej struktury danych **musi być imperatywna**.
- `union` wpływa nie tylko na swoje argumenty, ale również na wszystkie wartości scalanych klas abstrakcji.
- Efektywna implementacja wymaga zastosowania wskaźnikowej struktury danych.

Drzewa Find-Union

Example

```
let a = make_set "a"
and b = make_set "b"
and c = make_set "c"
and d = make_set "d";;
n_of_sets();;
- : int = 4

union a b;  union c d;  n_of_sets();;
- : int = 2

assert (not (a == d) && not (a == b));;
assert ((equivalent a b) && not (equivalent a d));;
union b c;;
assert (equivalent a d);;
```

Implementacja drzew Find-Union

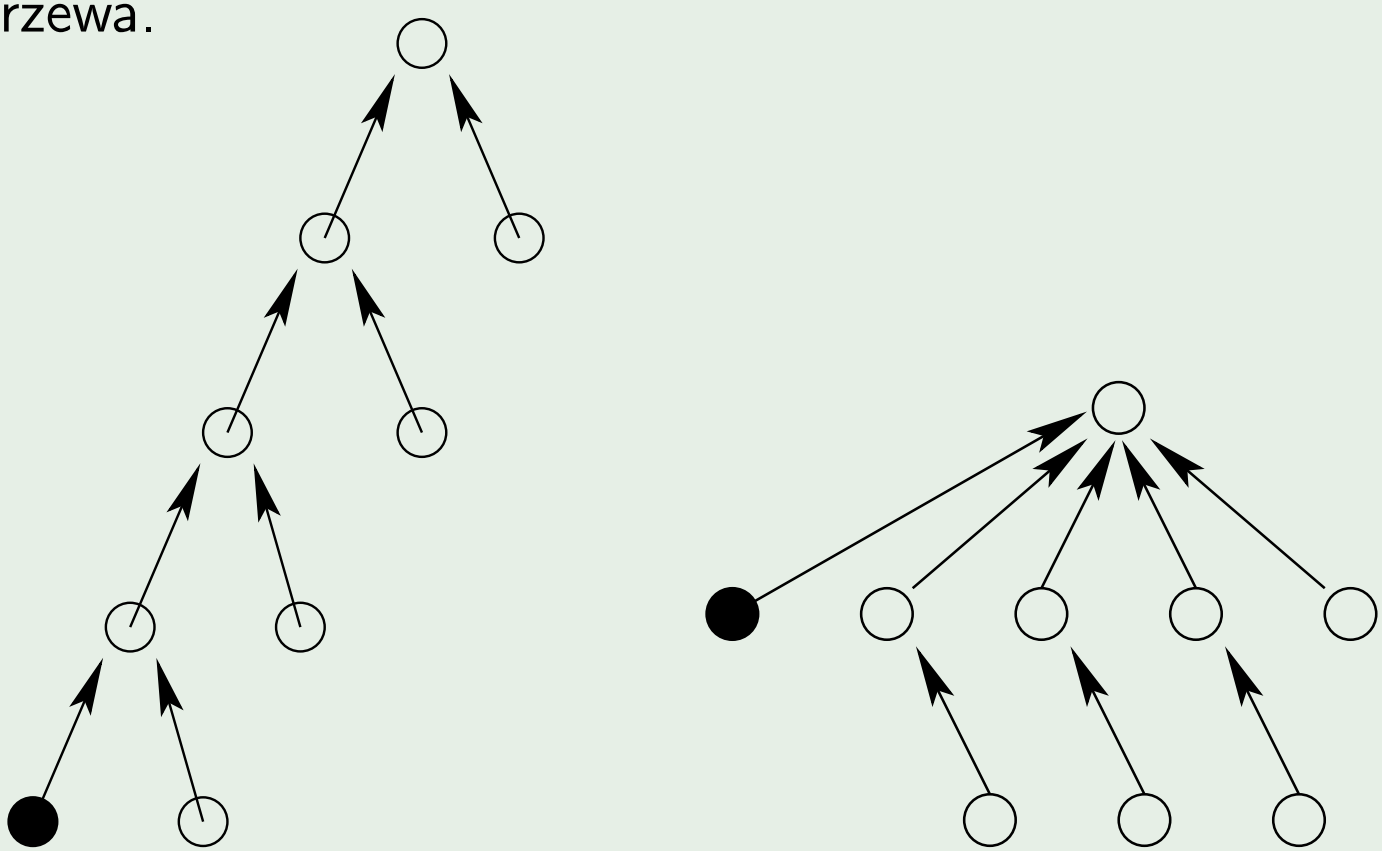
Example

- Klasy abstrakcji = drzewa.
  Węzły = elementy klas abstrakcji.
- Wskaźniki od węzła do ojca.
  Wskaźnik z korzenia prowadzi do niego samego.
- Korzeń drzewa = reprezentant klasy.
- `make_set` tworzy wierzchołek z „pętelką”.
- `find` znajduje korzeń drzewa i zwraca jego etykietę.
- `union` znajduje korzenie dwóch drzew i podłącza jednego jako syna drugiego.
  Korzeń mniejszego drzewa podczepiamy pod korzeń większego.

Kompresja ścieżek

Example

- Gdy wyszukujemy reprezentanta klasy, we wszystkich odwiedzanych węzłach przestawiamy wskaźniki na korzeń drzewa.



Wybór nowego korzenia

Example

- Gdy scalamy dwa drzewa — który korzeń ma się stać korzeniem nowego drzewa?
- Generalnie, podłączamy mniejsze drzewo do większego.
- Nie pamiętamy ani wielkości, ani wysokości drzew.
- Każdemu wierzchołkowi przypisujemy *rangę*, liczbę całkowitą, górne ograniczenie na wysokość wierzchołka.
- Ranga pojedynczego wierzchołka wynosi 0.
- Kompresja ścieżek nie wpływa na rangi wierzchołków.
- Scalając podłączamy korzeń o mniejszej randze, do korzenia o większej randze.
  Jeżeli oba korzenie mają tę samą rangę, to wybieramy którykolwiek i zwiększamy jego rangę o 1.

Wyliczanie elementów klas i licznik klas

Example

- Jak wyliczyć elementy klas abstrakcji?
- Potrzebujemy wskaźników idących w dół drzewa.
- Dla każdej klasy abstrakcji utrzymujemy również drzewo „rozpinające" elementy klasy.
- Korzeniem tego drzewa jest reprezentant klasy abstrakcji.
- W każdym węźle drzewa pamiętamy listę jego następników.
- Kształty drzew tworzonych przez wskaźniki idące „w górę" i „w dół" nie muszą się pokrywać.
- Dodatkowo, utrzymujemy licznik klas abstrakcji.

Implementacja drzew Find-Union

Example

Struktura danych:

```
type 'a set = {
  elem          : 'a;                    Etykieta
  up            : 'a set ref;            Przodek
  mutable rank  : int;                   Ranga
  mutable next  : 'a set list            Lista potomków
}

let sets_counter = ref 0
```

Implementacja drzew Find-Union

Example

```
let n_of_sets () = !sets_counter

let make_set x =
  let rec v = { elem=x; up=ref v; rank=0; next=[] }
  in begin
    sets_counter := !sets_counter + 1;
    v
  end

(* Znajduje korzeń drzewa, kompresując ścieżkę. *)
let rec go_up s =
  if s == !(s.up) then s
    else begin
      s.up := go_up !(s.up);
      !(s.up)
    end
```

Implementacja drzew Find-Union

Example

```
let find s = (go_up s).elem

let equivalent s1 s2 = go_up s1 == go_up s2

let union x y =
  let fx = go_up x   and fy = go_up y
  in
    if not (fx == fy) then begin
      if fy.rank > fx.rank then begin
        fx.up := fy;  fy.next <- fx :: fy.next
      end else begin
        fy.up := fx;  fx.next <- fy :: fx.next;
        if fx.rank = fy.rank then fx.rank <- fy.rank+1
      end;
      sets_counter := !sets_counter - 1
    end
```

Implementacja drzew Find-Union

Example

```
let elements s =
  let acc = ref []
  in
    let rec traverse s1 =
      begin
        acc := s1.elem :: !acc;
        List.iter traverse s1.next
      end
    in begin
      traverse (go_up s);
      !acc
    end
```

Analiza kosztu

Lemma

Dla każdego węzła, który nie jest korzeniem, jego ranga jest mniejsza od rangi jego ojca.

Dowód.

- Dowód przebiega indukcyjnie po historii obliczeń.
- Operacja `union` podłączając jeden korzeń do drugiego zapewnia, że ojciec ma większą rangę niż nowy syn.
- Kompresja ścieżek zachowuje powyższą własność.

Analiza kosztu

Lemma

Dla dowolnego drzewa, którego korzeń ma rangę r,
liczba węzłów w drzewie wynosi przynajmniej 2^r.

Dowód (indukcyjny po historii obliczeń):

- Jednoelementowe drzewa mają rangę 0.
- Kompresja ścieżek nie zmienia liczby węzłów, ani rangi korzenia.
- Jeżeli scalamy drzewa, których korzenie mają różne rangi, to własność jest zachowana.
- Jeżeli scalamy drzewa, których korzenie mają rangę r,
  to każde z tych drzew ma przynajmniej 2^r węzłów.
  Po ich scaleniu powstaje drzewo, które zawiera przynajmniej 2^{r+1} elementów i którego korzeń ma rangę $r+1$. □

Analiza kosztu

Fact

Dla dowolnej rangi r istnieje co najwyżej $\frac{n}{2^r}$ węzłów rangi $\geq r$.

Dowód.

- Ranga wierzchołków rośnie w wyniku operacji `union`.
- Usuńmy z obliczeń wszystkie operacje `union`, które tworzą wierzchołki o randze $> r$. Ich ranga pozostanie $= r$.
- Każdy wierzchołek rangi r jest korzeniem drzewa zawierającego przynajmniej 2^r węzłów.
- Ponieważ wszystkich wierzchołków jest n, więc wierzchołków rangi r (lub większej) może być co najwyżej $\frac{n}{2^r}$.

□

Analiza kosztu

Wniosek

Rangi węzłów nie przekraczają $\lfloor \log_2 n \rfloor$.

Definition

Funkcja $\log_2^i x$ jest zdefiniowana standardowo.
Może ona być nieokreślona.
Funkcja $\log^*$ jest zdefiniowana następująco:

$$\log^* x = \min\{i \geq 0 : \log_2^i x \leq 1\}$$

Example

$$\log^* 4 = 2$$

$$\log^* 65000 = 4$$

Analiza kosztu

Theorem

Łączny koszt wykonania m operacji na drzewach find-union (bez operacji `elements`), spośród których n to operacje `make_set` wynosi $O(m \cdot \log^* n)$.

Definition

Zanim przystąpimy do dowodu, zdefiniujemy funkcję B, odwrotną do funkcji $\log^*$:

$$B(j) = \begin{cases} \underbrace{2^{\cdot^{\cdot^{\cdot^{2}}}}}_{j \text{ razy}} & \\ 2 & \text{jeśli } j \geq 1 \\ 1 & \text{jeśli } j = 0 \\ -1 & \text{jeśli } j = -1 \end{cases}$$

Analiza kosztu

Dowód

- Wierzchołki dzielimy na *bloki*, ze względu na $\log^*$ ich rang.
  Bloki numerujemy tymi wartościami.
  Blok numer j zawiera rangi od $B(j-1)+1$ do $B(j)$.
- Bloki mają numery od 0 do co najwyżej
  $\log^*(\log_2 n) = \log^* n - 1$.
  Wszystkich bloków jest $\leq \log^* n$.
- Pomijając koszt operacji go_up, koszt operacji wynosi $O(m)$.
  Możemy więc skupić się na koszcie operacji go_up.
- Pojedyncza operacja go_up przebiega ścieżkę od danego węzła do syna korzenia: $(x_0, x_1, \ldots, x_l)$.
  Rangi wierzchołków na ścieżce rosną.
  Wierzchołki na ścieżce należące do tego samego bloku występują po kolei.

Analiza kosztu

Dowód

- Koszt go_up przypisujemy wierzchołkom na ścieżce.
  Dzielimy go na koszt *związany ze ścieżką* i *związany z blokami*.
- Koszt związany z blokami przypisujemy ostatnim w blokach wierzchołkom na ścieżce, oraz synowi korzenia, x_{l-1}.
- Koszt związany z blokami jednej operacji go_up wynosi $O(1 + \log^* n)$.
  Koszt związany z blokami wszystkich operacji go_up wynosi $O(m \cdot \log^* n)$.
- Należy jeszcze pokazać, że łączny koszt związany ze ścieżkami jest również rzędu $O(m \cdot \log^* n)$.

Analiza kosztu

Dowód

- Jeżeli jakiemuś wierzchołkowi został raz przydzielony koszt za blok, to już więcej nie będzie mu przydzielony koszt za ścieżkę.
- Jeżeli wierzchołkowi został w danej operacji go_up przydzielony koszt za ścieżkę, to w wyniku tej operacji jego ojciec zmienił się na taki o wyższej randze.
- Jeżeli ten nowy ojciec należy do innego bloku, to wierzchołek już więcej nie będzie miał przypisanego kosztu za ścieżkę.
- Jeżeli dany wierzchołek należy do bloku nr j, to może on mieć przypisany koszt za ścieżkę co najwyżej $B(j) - B(j-1) - 1$ razy.

Analiza kosztu

Fact

Oznaczmy przez $N(j)$ liczbę wierzchołków w bloku nr j.
Mamy: $N(j) \leq \frac{n}{B(j)}$.

Dowód lematu.

Korzystając z faktu, że węzłów o randze $\geq r$ jest $\leq \frac{n}{2^r}$ mamy:

$$\begin{aligned} N(0) &\leq \frac{n}{2^0} = \frac{n}{B(0)} \\ N(j) &\leq \frac{n}{2^{B(j-1)+1}} = \frac{n}{2B(j)} < \frac{n}{B(j)} \end{aligned}$$

Analiza kosztu

Dowód.

Łączną liczbę opłat za ścieżki możemy oszacować:

$$\sum_{j=0}^{\log^* n-1} (N(j) \cdot (B(j) - B(j-1) - 1)) \leq$$

$$\leq \sum_{j=0}^{\log^* n-1} \left(\frac{n}{B(j)} \cdot (B(j) - B(j-1) - 1) \right) \leq$$

$$\leq \sum_{j=0}^{\log^* n-1} \left(\frac{n}{B(j)} \cdot B(j) \right) = \sum_{j=0}^{\log^* n-1} (n) \leq n \cdot \log^* n$$

Stąd, łączny koszt wszystkich operacji jest rzędu $O(m \log^* n)$. □

Analiza kosztu

- W praktyce czynnik $\log^* n$ można przyjąć za stały.
- Nie przekracza on 5 dla wszelkich praktycznych wielkości.
- $B(5) \geq 10^{19\,728}$
- Liczba atomów we Wszechświecie jest szacowana na 10^{80}.