Programowanie Funkcyjne

Wybrane trwałe struktury danych

Piotr Polesiuk

ii-uwr

28 października 2024

Trwałe struktury danych

- Dotychczas poznany podzbiór OCamla pozwala tylko na tworzenie danych, ale nie modyfikacje.
- Nie przeszkodziło nam to w zdefiniowaniu skompilowanych struktur danych np. kopców.
- Udało się to dzięki trwałym strukturom danych, gdzie *modyfikacja* oznacza utworzenie zmodyfikowanej kopii.
- Jest to odmienne podejście niż klasyczne ulotne struktury danych.

Zalety trwałych struktur danych

- Tworzenie zmodyfikowanej kopii nie musi być drogie, bo skopiowaniu ulega tylko niewielki kawałek struktury, a reszta jest współdzielona z oryginałem.
- A zalet mamy wiele:
  - współdzielenie
  - wersjonowanie
  - łatwe wnioskowanie
  - ...

Definiowanie trwałych struktur danych

- Wiele klasycznych struktur danych elegancko da się wyrazić jako trwałe struktury danych, np.
  - kopce lewicowe
  - drzewa czerwono-czarne
  - 2-3 drzewa
  - kopce parujące
  - ...
- Często funkcyjna implementacja takich struktur jest czytelniejsza niż imperatywna.
- Ale niektóre struktury danych nie chcą wpasować się w te ramy, np.
  - kolejki
  - tablice
  - ...

Kolejki

Dwie naiwne implementacje

```
type 'a queue = 'a list

let push q x = q @ [x]

let pop q =
  match q with
  | []     -> None
  | x :: q -> Some(x, q)
```

```
type 'a queue = 'a list

let push q x = x :: q

let rec pop q =
  match q with
  | []     -> None
  | x :: q ->
    (match pop q with
    | None       -> Some(x, [])
    | Some(y, q) -> Some(y, x :: q))
```

Mieć ciastko i zjeść ciastko

- Gdy na początku listy jest przód kolejki (pierwsza implementacja) to:
  - `pop` kosztuje $O(1)$
  - `push` kosztuje $O(n)$
- Gdy na początku listy jest tył kolejki (druga implementacja) to:
  - `pop` kosztuje $O(n)$
  - `push` kosztuje $O(1)$
- A gdyby tak trzymać dwie listy?

Programowanie na żywo

Koszt zamortyzowany

- Pesymistyczny koszt operacji pop to $O(n)$
- Ale zazwyczaj kosztuje $O(1)$
- Przed kosztownym odwróceniem n-elementowej listy trzeba wykonać n (tanich) wstawień do kolejki
- Całkowity czas wykonania n operacji na kolejce będzie wynosił $O(n)$
- Zamortyzowany koszt jednej operacji to $O(1)$

Analiza zamortyzowana

- Na każdym *consie* tylnej listy kładziemy żeton —
  depozyt czasu potrzebnego na odwrócenie jednego *consa*
- Zamortyzowany koszt operacji `push` to

  faktyczny koszt operacji `push` ($O(1)$)
  $+$
  koszt jednego żetonu ($O(1)$)
- Zamortyzowany koszt operacji `pop` to $O(1)$
  — kosztowne odwracanie opłacamy w całości ze zgromadzonych środków

Analiza zamortyzowana i współdzielenie

- Podana analiza zakłada, że każdej wersji struktury użyjemy raz.
- Kopiowanie danych przez współdzielenie jest tanie ($O(1)$)
- Ale kopiowanie żetonów jest drogie.
- W szczególności można skopiować kolejkę tuż przed odwróceniem listy i wyzwolić kosztowne odwracanie wiele razy!

Kolejki Hooda-Melville'a

Dygresja

- Przy każdej operacji zamiast płacić za odwracanie listy, można faktycznie wykonać kawałek odwracania.
- Elementy w kolejce podzielmy na trzy części: f, m, r
- Kolejka Hooda-Melville'a to trójka:
  - Listę f (z pierwszym elementem z przodu)
  - Listę r (z ostatnim elementem z przodu)
  - Częściowo policzoną listę:

$$s = \texttt{rev\_append (rev } f\texttt{) (rev } m\texttt{)}$$

- Gdy s się policzy, to s zastępuje f, a r zastępuje m
- Przy każdej operacji obliczamy kawałek s, tak by zdążyć je policzyć zanim f się opróżni (dwa kroki na operację wystarczą)
- W s pamiętamy ile elementów z f już usunęliśmy

Tablice

i listy o dostępie swobodnym

Listy o dostępie swobodnym

Chcemy zaprojektować strukturę danych, która:

- pozwala na szybki dostęp do dowolnego elementu w czasie $O(\log n)$,
- pozwala na operacje `cons`, `hd` i `tl`

Można użyć słowników (drzew zbalansowanych),
ale wtedy `cons` i `tl` działają w czasie $O(\log n)$

Czy można lepiej?

Struktura list i liczb naturalnych

- Listy i liczby naturalne w zapisie unarnym mają bardzo podobną strukturę
- Głębokość (długość) liczb unarnych (i list) jest $O(n)$
- Głębokość (długość) liczb binarnych jest $O(\log n)$
- A gdyby listy miały strukturę liczb binarnych?

Programowanie na żywo

Koszt operacji `cons` i `tl`

- Długie ciągi operacji `cons` potrzebują średnio $O(1)$ na operację — położymy żeton na każdej jedynce.

$$10010100111111 + 1 =$$
$$10010101000000$$

- Długie ciągi operacji `tl` potrzebują średnio $O(1)$ na operację — położymy żeton na każdym zerze.
- Ale nie wolno mieszać!

$$1111111111111 + 1 = \quad 10000000000000$$
$$10000000000000 - 1 = \quad 1111111111111$$
$$1111111111111 + 1 = \quad \ldots$$

Cyfry nadmiarowe

- Rozważmy system binarny z cyframi 0, 1, 2

$$0+1=1 \qquad 12+1=21$$
$$1+1=2 \qquad 21+1=22$$
$$2+1=11 \qquad 22+1=111$$
$$11+1=12 \qquad 111+1=112$$

- Następnik nie tworzy długich ciągów zer, kłopotliwych dla poprzednika
- Zamortyzowany koszt operacji `cons` i `tl` to $O(1)$:
  na każdym zerze i na każdej dwójce kładziemy żeton

Liczby binarne skośne

Dygresja

- Weźmy system pozycyjny, w którym cyfra na n-tej pozycji ma wagę $2^n - 1$
- Najmłodsza cyfra ma wagę 0
- Używamy cyfr 0, 1 i 2
- $2 \cdot (2^n - 1) + 1 \quad = \quad 2^{n+1} - 1$
- Każda liczba ma w zapisie dokładnie jedną dwójkę
  — jako najmłodszą niezerową cyfrę

$$2 + 1 = 12 \qquad 112 + 1 = 120$$
$$12 + 1 = 20 \qquad 120 + 1 = 200$$
$$20 + 1 = 102 \qquad 200 + 1 = 1002$$
$$102 + 1 = 112 \qquad 1002 + 1 = 1012$$

- Operacja następnika (i poprzednika) modyfikuje co najwyżej trzy cyfry!

Liczby binarne skośne — reprezentacja rzadka

Dygresja

- Operacja następnika (i poprzednika) modyfikuje co najwyżej trzy cyfry!
- Ale te cyfry mogą być w środku liczby za długimi ciągami zer
- Możemy reprezentować długie ciągi zer jako jedną liczbę — długość ciągu zer

Zippery

Motywacja

- Modyfikacja trwałej struktury danych wymaga skopiowania całej ścieżki od korzenia do miejsca modyfikacji
- Często przeprowadzamy wiele małych zmian w jednym miejscu
  - Edytowanie dokumentu tekstowego
  - Modyfikowanie drzewiastej bazy danych (XML)
  - Budowanie drzewa dowodu
- Można zmodyfikować strukturę danych tak, by mieć szybki dostęp do wybranego miejsca

Programowanie na żywo

Różniczkowanie typów

Różniczkowanie typów

Dygresja

Wyobraźmy sobie typ opisujący strukturę drzewiastą przechowującą dane typu x.
Ten typ zbudowany jest za pomocą następujących operacji na typach

- danych typu x
- stałych 1 (unit) oraz 0 (empty)
- sumy rozłącznej $t_1 + t_2$

```
type ('t1, 't2) either =
  | Left  of 't1
  | Right of 't2
```

- produktu (par) $t_1 \times t_2$

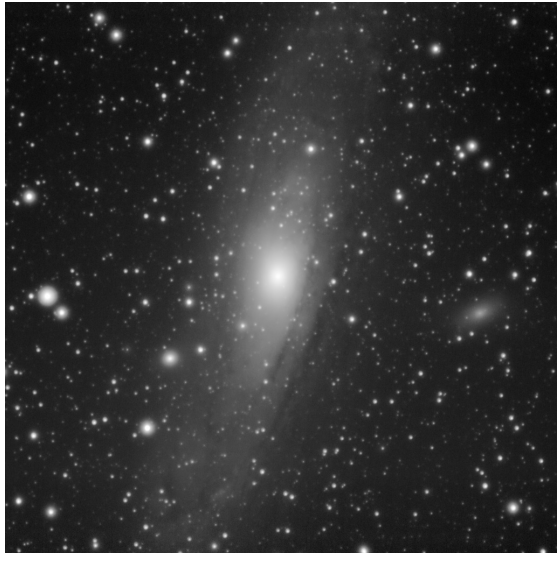
```
type ('t1, 't2) prod = 't1 * 't2
```

Różniczkowanie typów

Typ opisujący kontekst ($\frac{\partial}{\partial x} t$) dla danych typu x w typie t można wyliczyć

$$\frac{\partial}{\partial x} x = 1$$
$$\frac{\partial}{\partial x} 0 = 0$$
$$\frac{\partial}{\partial x} 1 = 0$$
$$\frac{\partial}{\partial x}(t_1 + t_2) = \frac{\partial}{\partial x} t_1 + \frac{\partial}{\partial x} t_2$$
$$\frac{\partial}{\partial x}(t_1 \times t_2) = \frac{\partial}{\partial x} t_1 \times t_2 + t_1 \times \frac{\partial}{\partial x} t_2$$

Projekt końcowy — szukajcie inspiracji



M31, ЮПИТЕР–37A + ZWO ASI 120MM, Ekspozycja: 180×19s