Programowanie Funkcyjne 2018

Lista zadań nr 4

7 listopada 2018

Zadanie 1 (2p). Napisz funkcję rozstrzygającą czy dana lista jest palindromem, ale taką, która wykonuje co najwyżej $\lceil n/2 \rceil$ wywołań rekurencyjnych, gdzie n jest nieznaną długością listy (nie potrzeba i nie wolno jej liczyć). Funkcja nie powinna również alokować nowych list.

Zadanie 2 (2p). Rozważmy typ danych dla drzew binarnych, zdefiniowany następująco:

```
type 'a btree = Leaf | Node of 'a btree * 'a * 'a btree
```

Mówimy, że drzewo jest zbalansowane, jeśli dla każdego węzła v liczby węzłów w lewym i prawym poddrzewie zakorzenionym w v różnią się co najwyżej o 1.

1. Napisz efektywną funkcję sprawdzającą czy dane drzewo jest zbalansowane.
2. Napisz funkcję, która dla zadanej listy etykiet tworzy zbalansowane drzewo, dla którego tę listę można otrzymać przechodząc je w porządku preorder.

Zadanie 3 (2p). Napisz możliwie efektywne funkcje przejścia wszerz oraz przejścia w głąb (preorder) dla obu reprezentacji drzew wielokierunkowych podanych na wykładzie (podanych również poniżej).

```
type 'a mtree  = MNode of 'a * 'a forest
and  'a forest = EmptyForest | Forest of 'a mtree * 'a forest

type 'a mtree_lst = MTree of 'a * ('a mtree_lst) list
```

Zadanie 4 (6p). Zdefiniuj typ służący do reprezentacji formuł rachunku zdań składających się ze zmiennych zdaniowych, negacji, koniunkcji i alternatywy.

1. Napisz funkcję sprawdzającą, czy dana formuła jest tautologią. W tym celu należy generować kolejne wartościowania zmiennych zdaniowych występujących w danej formule i sprawdzać dla nich wartość formuły. W przypadku, gdy formuła nie jest tautologią, funkcja powinna, oprócz tej informacji, podać jedno z wartościowań, dla których formuła nie jest prawdziwa.
2. Napisz funkcję, która przekształca zadaną formułę w formułę jej równoważną w negacyjnej postaci normalnej, tj. w której negacja występuje tylko przy zmiennych zdaniowych.
3. Napisz funkcję, która przekształca zadaną formułę w formułę jej równoważną w koniunkcyjnej postaci normalnej.
4. Napisz funkcję sprawdzającą (syntaktycznie), czy zadana formuła jest tautologią, korzystając z faktu, że każdą formułę można przedstawić w koniunkcyjnej postaci normalnej.
5. Napisz funkcję, która przekształca zadaną formułę w formułę jej równoważną w dyzjunkcyjnej postaci normalnej.
6. . Napisz funkcję sprawdzającą (syntaktycznie), czy zadana formuła jest sprzeczna, korzystając z faktu, że każdą formułę można przedstawić w dyzjunkcyjnej postaci normalnej.

Uwaga: w miejscach gdzie uznasz to za przydatne, powinieneś zdefiniować dodatkowe typy danych.

Zadanie 5 (3p). Funkcja jest napisana w stylu przekazywania kontynuacji (continuation-passing style, CPS), jeżeli przyjmuje dodatkowy argument funkcyjny zwany kontynuacją, który reprezentuje całą resztę obliczeń jakie mają zostać przeprowadzone po powrocie z tej funkcji. W konsekwencji, funkcje w CPS-ie zwracają wynik wywołując swoją kontynuację, a wszystkie wywołania w programie w CPS-ie są ogonowe. Na przykład, funkcja licząca silnię napisana w CPS-ie wygląda następująco:

```
let rec fact_cps n k = if n = 0 then k 1 else fact_cps (n-1) (fun v
    -> k (n*v))
```

Kontynuacja początkowa przekazana funkcji `fact_cps` mówi co zrobić z wynikiem obliczenia silni zadanej liczby. Typowe wywołanie funkcji `fact_cps` dostaje identyczność jako kontynuację początkową (gdy silnia jest obliczona, wystarczy ją zwrócić):

```
let fact n = fact_cps n (fun v -> v)
```

Dla drzew binarnych z zadania 2, napisz funkcję `prod : int btree -> int`, która liczy iloczyn wszystkich wartości w drzewie. Zapisz tę funkcję w CPS-ie, a następnie zmodyfikuj otrzymaną funkcję tak by w przypadku napotkania wartości `0` funkcja wykonała bezpośredni skok do miejsca swego wywołania, bez krokowego powracania z rekursji.

Zadanie 6 (5p). Gdy modyfikujemy strukturę danych, często chcemy wykonać więcej niż jedną operację w pobliskich miejscach. W przypadku czysto funkcyjnych struktur może to skutkować niepotrzebnie długim czasem wykonania, spowodowanym przechodzeniem i odtwarzaniem struktury danych przy każdej zmianie. Aby zapobiec temu problemowi, możemy rozbić operację modyfikacji struktury na dwie części: znajdowania miejsca w strukturze, w którym mamy wykonać zmianę, oraz przeprowadzenia zmiany w konkretnym miejscu; musimy też znaleźć odpowiednią pośrednią reprezentację dla takich miejsc.

1. Zdefiniuj typ danych `'a place` jako pośrednią reprezentację dla typu `'a list`.
2. Zaimplementuj funkcje `findNth : 'a list -> int -> 'a place` oraz `collapse : 'a place -> 'a list`. Pierwsza powinna lokalizować `n`-te miejsce na liście, druga — zapominać informację o miejscu i zwracać listę na której działamy.
3. Zaimplementuj funkcje `add : 'a -> 'a place -> 'a place` oraz `del : 'a place -> 'a place`, odpowiednio dodającą i usuwającą element w odpowiednim miejscu listy. Funkcje powinny działać w czasie stałym, oraz spełniać następujące równości:

   ```
   collapse (add 3 (findNth [1;2;4] 2)) == [1;2;3;4]
   collapse (del (findNth [1;2;4] 2)) == [1;2]
   del (add x p) == p
   ```

   dla dowolnych `x : 'a` i `p : 'a place`
4. Zaimplementuj funkcje `next: 'a place -> 'a place` oraz `prev : 'a place -> 'a place`, służące do nawigacji w strukturze listy. Te funkcje również powinny działać w czasie stałym.
5. Listy nie są jedynym typem danych dla którego możemy definiować struktury pamiętające miejsce w którym pracujemy. Zdefiniuj typ danych `'a btplace` będący pośrednią reprezentacją pozwalającą na modyfikacje drzew binarnych z Zadania 2 (nie musisz definiować na nim odpowiednich operacji).