Programowanie Funkcyjne 2020

Lista zadań nr 2 dla grup mabi, mbu, ppo i efes

Na zajęcia 20 i 21 października 2020 r.

Zadanie 1 (1p). Zdefiniuj funkcję `sublists` znajdującą wszystkie podlisty (rozumiane jako podciągi, niekoniecznie kolejnych elementów) listy zadanej jako argument. Zadbaj o to by Twoja funkcja nie generowała nieużytków.

Zadanie 2 (2p). Zdefiniuj funkcję `cycle: 'a list -> int -> 'a list`, która w cykliczny sposób przesuwa listę o zadaną liczbę pozycji n, przy czym zakładamy, że długość listy jest nie mniejsza niż n. Na przykład:

```
cycle [1; 2; 3; 4] 3 = [2; 3; 4; 1]
```

Zadanie 3 (5p). Sortowanie przez scalanie.

1. Zdefiniuj funkcję `merge`, która łączy dwie listy posortowane rosnąco w pewnym porządku tak, by wynik działania funkcji był także listą posortowaną rosnąco w tym samym porządku. Argumentami funkcji merge powinny być: funkcja `cmp: 'a -> 'a -> bool` (reprezentująca porządek) oraz dwie listy elementów typu `'a`. Na przykład

   ```
   merge (<=) [1; 2; 5] [3; 4; 5] = [1; 2; 3; 4; 5; 5].
   ```

2. Zapisz tę samą funkcję używając rekursji ogonowej, a następnie porównaj działanie obu funkcji na odpowiednich przykładach. Która z nich jest lepsza?
3. Zdefiniuj funkcję `halve` dzielącą listę w połowie (przy liście nieparzystej długości druga z połówek powinna być dłuższa) tak aby nie wyliczać jawnie jej długości i użyć $\lfloor n/2 \rfloor$ wywołań rekurencyjnych (gdzie n jest długością listy)
4. Wykorzystaj funkcję `merge` do napisania funkcji `mergesort` sortującej listę przez scalanie.
5. Zaproponuj alternatywną implementację algorytmu sortowania przez scalanie, w której funkcja `merge` jest ogonowa, ale nie wykonuje odwracania list.[1] Nie przejmuj się, jeżeli otrzymasz algorytm sortowania, który nie jest stabilny. Porównaj szybkość działania tej implementacji z definicją z poprzedniego punktu.

Zadanie 4 (3p). Zdefiniuj funkcję zwracającą listę wszystkich permutacji zadanej listy.

Zadanie 5 (1p). Zdefiniuj funkcję generującą wszystkie sufiksy danej listy. Na przykład dla listy `[1; 2; 3]` Twoja funkcja powinna zwrócić listę `[[1; 2; 3]; [2; 3]; [3]; []]`. Następnie, zdefiniuj funkcję generującą wszystkie prefiksy danej listy. Na przykład dla listy `[1; 2; 3]` Twoja funkcja powinna zwrócić listę `[[]; [1]; [1; 2]; [1; 2; 3]]`.

Zadanie 6 (5p). Struktura list jest bardzo podobna do struktury liczb naturalnych reprezentowanych unarnie (ciąg następników nałożonych na zero). Jedyna różnica polega na tym, że w listach do następników (zwanych *consami*), przyczepiamy dane — elementy listy. Na poprzedniej liście zadań widzieliśmy reprezentację Churcha liczb naturalnych. W podobny sposób możemy przedstawić listy. Jak można się spodziewać, jedyna różnica polega na tym, że iterowana funkcja przyjmuje dodatkowy argument — element listy.

[1] Taka funkcja `merge` oblicza inny wynik: zacznij od wyspecyfikowania działania tej funkcji, a następnie zastanów się jak przy jej pomocy posortować listę.

```
type 'a clist = { clist : 'z. ('a -> 'z -> 'z) -> 'z -> 'z }
```

W ten sposób lista [a_1,a_2,...,a_n], jest reprezentowana przez funkcję dwuargumentową, która dla argumentów `f` oraz `z` obliczy `f` a_1 `(f` a_2 `( ... (f` a_n `z) ... ))`.

Zaimplementuj następujące operacje na listach w kodowaniu Churcha:

- `cnil : 'a clist` — lista pusta,
- `ccons : 'a -> 'a clist -> 'a clist` — dołączanie głowy do listy,
- `map : ('a -> 'b) -> 'a clist -> 'b clist` — nakładania podanej funkcji na wszystkie wartości listy,
- `append : 'a clist -> 'a clist -> 'a clist` — konkatenacja list,
- `clist_to_list : 'a clist -> 'a list` oraz `clist_of_list : 'a list -> 'a clist` — konwersja między listami Churcha, a standardowymi listami.

Zauważ, że implementacja operacji `cnil`, `ccons` oraz `append` jest bardzo podobna do implementacji `zero`, `succ` oraz `add` dla liczebników Churcha. Zaimplementuj funkcję `prod : 'a clist -> 'b clist -> ('a * 'b) clist`, która będzie analogiczna do mnożenia. Co robi ta funkcja? Czy potrafisz zaimplementować funkcję analogiczną do potęgowania?

Zadanie 7 (3p). Zauważmy, że o ile w reprezentacji Churcha liczb naturalnych łatwo było zaimplementować operacje arytmetyczne takie jak dodawanie czy mnożenie, dużo trudniej jest zaimplementować operację *poprzednika*, którą możemy zdefiniować następująco:

$$\mathrm{pred}(n) = \begin{cases} 0 & \text{gdy } n = 0 \\ m & \text{gdy } n = m + 1 \end{cases}$$

Kluczowym pomysłem, umożliwiającym zaimplementowanie poprzednika jest użycie liczby n (w naszej reprezentacji Churcha) do obliczenia *pary* liczb, z których jedna będzie reprezentowała poprzednik. Użyj tej obserwacji aby zdefiniować funkcję `pred : cnat -> cnat`, a następnie zaadaptuj swoje rozwiązanie do przypadku list definiując funkcję `ctail : 'a clist -> 'a clist` (dla listy pustej `ctail` powinien zwracać listę pustą).