Programowanie Funkcyjne 2018

Lista zadań nr 2 dla grupy TWI: zabawy w permutowanie i sortowanie list

Na zajęcia 18 października 2018

W poniższych zadaniach możesz używać funkcji zdefiniowanych w modułach `Pervasives` i `List`.

Zadanie 1 (2 pkt). Zaprogramuj funkcję:

```
ins_everywhere : 'a -> 'a list -> 'a list list
```

która tworzy listę wszystkich list powstałych przez wstawienie podanego elementu we wszystkie możliwe miejsca w podanej liście, np.

```
ins_everywhere 0 [1;2;3] = [[0;1;2;3];[1;0;2;3];[1;2;0;3];[1;2;3;0]]
```

Postaraj się napisać najprostsze, najbardziej zrozumiałe rozwiązanie. Użyjesz przy tym prawdopodobnie rekursji nieogonowej. Zastanów się wówczas, jaka jest głębokość rekursji względem a) rozmiaru danych wejściowych, b) rozmiaru danych wyjściowych, c) czasu działania tej funkcji? Ile nieużytków generuje twoja funkcja? Zaprogramuj następnie ogonową wersję tej funkcji. Czy warto było?

Zadanie 2 (1 pkt). Algorytm permutowania przez wstawianie. Zaprogramuj funkcję:

```
iperm : 'a list -> 'a list list
```

tworzącą listę wszystkich permutacji podanej listy (w dowolnej kolejności). Skorzystaj przy tym z funkcji `ins_everywhere` z poprzedniego zadania. Czy warto tu dbać o rekursję ogonową? Czy warto dbać o to, by nie generować nieużytków?

Zadanie 3 (1 pkt). Z generowaniem permutacji poprzez wstawianie jest związany algorytm sortowania poprzez wstawianie — wystarczy wstawiać nie wszędzie, tylko tak, by utworzona lista była posortowana. Zaprogramuj funkcję:

```
isort : 'a list -> 'a list
```

sortującą listy w ten sposób (wewnątrz niej zaprogramujesz pewnie funkcję pomocniczą `ins_ord` wstawiającą element do listy w taki sposób, żeby zachować porządek). Nie staraj się na siłę użyć rekursji ogonowej w żadnej z tych funkcji.

Zadanie 4 (2 pkt). Napisz ogonową wersję funkcji z poprzedniego zadania, tj. funkcję sortującą przez wstawianie i korzystającą ze stałej ilości miejsca na stosie. Oszacuj głębokość rekursji wersji nieogonowej i odpowiedz na pytanie, czy warto ją było eliminować? Jak to wpłynęło na czytelność programu? A jego efektywność, w tym ilość generowanych nieużytków?

Zadanie 5 (1 pkt). Zaprogramuj funkcję:

```
sel_anything : 'a list -> ('a * 'a list) list
```

tworzącą dla podanej listy listę par złożonych z wybranego elementu tej listy i listy powstałej przez usunięcie tego elementu z oryginalnej listy, np.

```
sel_anything [1;2;3;4] = [(1,[2;3;4]);(2,[1;3;4]);(3,[1;2;4]);(4,[1;2;3])]
```

Zadanie 6 (2 pkt). Algorytm permutowania przez wybieranie. Zaprogramuj funkcję:

```
sperm : 'a list -> 'a list list
```

tworzącą listę wszystkich permutacji podanej listy (w dowolnej kolejności). Skorzystaj przy tym z funkcji `sel_anything` z poprzedniego zadania. Czy warto tu dbać o rekursję ogonową? Czy warto dbać o to, by nie generować nieużytków?

Zadanie 7 (1 pkt). Z generowaniem permutacji poprzez wybieranie jest związany algorytm sortowania poprzez wybieranie — wystarczy w każdej iteracji wybierać nie dowolny element, tylko zawsze najmniejszy. Zaprogramuj funkcję:

```
ssort : 'a list -> 'a list
```

sortującą listę w ten sposób (wewnątrz niej zaprogramujesz pewnie funkcję pomocniczą `sel_min` wybierającą najmniejszy element z listy). Nie staraj się na siłę użyć rekursji ogonowej w żadnej z dwóch funkcji.

Zadanie 8 (2 pkt). Napisz ogonową wersję funkcji z poprzedniego zadania, tj. funkcję sortującą przez wybieranie i korzystającą ze stałej ilości miejsca na stosie. Oszacuj głębokość rekursji wersji nieogonowej i odpowiedz na pytanie, czy warto ją było eliminować? Jak to wpłynęło na czytelność programu? A jego efektywność, w tym ilość generowanych nieużytków? Czy łatwo można tu zapewnić stabilność algorytmu sortowania?

Zadanie 9 (1 pkt). Zaprogramuj funkcję:

```
monotone : 'a list -> bool
```

odpowiadającą na pytanie, czy podana lista jest uporządkowana rosnąco. Użyj standardowych operatorów porównania. Funkcja powinna być efektywna, tzn. nie generować nieużytków i korzystać z rekursji ogonowej. Wykorzystaj ją do zaprogramowania następującej funkcji:

```
perm_sort : ('a list -> 'a list list) -> ('a list -> 'a list)
```

Jeśli `p : 'a list -> 'a list list` jest funkcją tworzącą listę wszystkich permutacji podanej listy, to `sort_perm p` jest funkcją, która sortuje listę w ten sposób, że generuje wszystkie jej permutacje i wybiera listę posortowaną. Powyższa funkcja przydaje się do testowania wydajności systemu! Uruchom ją dla algorytmu generowania permutacji przez wstawianie i przez wybieranie. Jak zapewnić stabilność takiego algorytmu sortowania?

Zadanie 10 (1 pkt). Zaprogramuj funkcję:

```
split : 'a list -> 'a list * 'a list
```

która dzieli listę na połowy równej (z dokładnością do jednego elementu) długości, np.:

```
split [1;2;3;4;5;6;7;8;9] = ([1;2;3;4],[5;6;7;8;9])
```

Pierwsza z list powinna być skopiowana, druga — współdzielona. Nie ulegnij pokusie korzystania z funkcji `List.length` — zwyczajna rekursja względem listy powinna wystarczyć!

Zadanie 11 (1 pkt). Zaprogramuj funkcję:

```
(<+>) : 'a list -> 'a list -> 'a list
```

scalającą posortowane listy. Wykorzystaj ją oraz funkcję z poprzedniego zadania do zaprogramowania funkcji sortującej:

```
msort : 'a list -> 'a list
```

Postaraj się napisać najprostsze, najbardziej zrozumiałe rozwiązanie. Użyjesz przy tym prawdopodobnie rekursji nieogonowej.

Zadanie 12 (1 pkt). Usunięcie rekursji nieogonowej poprzez dodanie akumulatora do funkcji (<+>) powoduje, że otrzymujemy scaloną listę w odwróconej kolejności. Zaprogramuj taką funkcję

```
rev_merge : ('a -> 'a -> bool) -> ('a list -> 'a list -> 'a list)
```

wykorzystującą rekursję ogonową i nie generującą nieużytków, że dla dowolnych list `xs` i `ys` posortowanych rosnąco mamy:

```
rev_merge (<=) xs ys = List.rev (xs <+> ys)
rev_merge (>) (List.rev xs) (List.rev ys) = (xs <+> ys)
```

Zadanie 13 (1 pkt). Zmodyfikuj implementację funkcji `msort` tak, by wykorzystywała funkcję `rev_merge` z poprzedniego zadania i korzystała z funkcji `List.rev` do odwracania jej wyniku. Zamieniliśmy tym samym nieogonowość na generowanie nieużytków. Czy warto było?

Zadanie 14 (3 pkt). Okazuje się, że używając funkcji `rev_merge` możemy zjeść ciasteczko i jednocześnie je mieć: aby pozbyć się nieogonowości i nie generować nieużytków możemy zdefiniować dwie wzajemnie rekurencyjne funkcje

```
msort_down : int -> 'a list -> 'a list
msort_up : int -> 'a list -> 'a list
```

z których pierwsza sortuje listy malejąco, a druga rosnąco. Aby pozbyć się generowania nieużytków poprzez kopiowanie połowy listy w funkcji `split` zaimplementuj lepsze rozwiązanie: funkcje `msort_down` i `msort_up` mają dodatkowy parametr, który mówi, jak długi prefiks argumentu należy posortować, np.:

```
msort_down 3 [1;7;2;8;4;5] = [7;2;1]
```

Zaimplementuj to rozwiązanie. Potrzebne będą pewnie takie funkcje standardowe, jak `List.length`, `List.rev` itp. Ile nieużytków generuje twoja funkcja sortująca? Jaka jest maksymalna głębokość rekursji?

Niniejszy tekst jest kompilacją listy zadań przygotowanej przez Filipa Sieczkowskiego.