Programowanie Funkcyjne 2020

Lista zadań nr 6 dla grup mabi, mbu, ppo i efes

Na zajęcia 24 i 25 listopada 2020

Zadanie 1 (2p). Interesującą techniką w programowaniu funkcyjnym jest oddzielenie rekursji od reszty definicji. Dla funkcji rekurencyjnych możemy to osiągnąć definiując *kombinator punktu stałego*

```
let rec fix f x = f (fix f) x
```

a pozostałe funkcje rekurencyjne definiować przy jego pomocy. Na przykład naiwna definicja funkcji obliczającej liczby Fibonacciego może wyglądać następująco.

```
let fib_f fib n =
  if n <= 1 then n
  else fib (n-1) + fib (n-2)

let fib = fix fib_f
```

Zaletą tego podejścia jest to, że łatwo teraz zmienić kod tak, by każde wywołanie funkcji rekurencyjnej wykonywało dodatkową pracę — wystarczy użyć innego kombinatora punktu stałego. Zdefiniuj następujące wersje kombinatora punktu stałego:

- `fix_with_limit : int -> (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b` — działa jak zwykły `fix`, ale dostaje dodatkowy parametr oznaczający maksymalną głębokość rekursji. W przypadku przekroczenia limitu funkcja powinna zgłosić wyjątek.
- `fix_memo : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b` — kombinator, który dodatkowo implementuje spamiętywanie, tzn. zapamiętuje wyniki wszystkich dotychczasowych wywołań. Gdy dana funkcja była kiedyś wołana z danym parametrem, to nie powinniśmy jej liczyć po raz kolejny, tylko od razu zwrócić zapamiętaną wartość. Np. gdy zdefiniujemy `fib` jako

  ```
  let fib = fix_memo fib_f
  ```

  to dla danego n pierwsze wywołanie `fib` n powinno policzyć się w czasie liniowym względem n, a każde następne w czasie stałym. Zapoznaj się z modułem `Hashtbl` z biblioteki standardowej w celu efektywnej implementacji funkcji `fix_memo`.

Zadanie 2 (2p). Zaimplementuj funkcję `fix` na dwa sposoby nie używając jawnej rekursji (tj. formy `let rec`):

- używając typów rekurencyjnych;
- używając mutowalnego stanu (możesz użyć funkcji `Obj.magic` albo wyjątku, jeśli uzasadnisz że jest to bezpieczne).

Zadanie 3 (1p). Zdefiniuj parę funkcji `next: unit -> int` oraz `reset: unit -> unit`, które odpowiednio zwracają wartość pewnego licznika, zwiększając go o jeden oraz resetują licznik. Zrób to tak, żeby nie definiować dodatkowych zmiennych globalnych, tzn. rozwiązanie

```
let cnt = ref 0
let next () =
  let r = !cnt in
  cnt := r + 1;
  r
let reset () =
  cnt := 0
```

nie jest akceptowalne, bo definiuje zmienną `cnt`, która jest widoczna dla użytkownika.

Zadanie 4 (2p). Zdefiniuj typ do reprezentacji nieskończonych ciągów elementów dowolnego typu (strumieni). Następnie:

- zdefiniuj strumień przybliżający wartość liczby π z rosnącą dokładnością, korzystając ze wzoru Leibniza:

$$\frac{\pi}{4} = \frac{1}{1} - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \ldots,$$

- napisz funkcję, która dla danej funkcji f przekształca strumień $x_1, x_2, x_3, \ldots$ w strumień postaci $f\ x_1\ x_2\ x3, f\ x_2\ x_3\ x_4, f\ x_3\ x_4\ x_5, \ldots,$
- zastosuj tę funkcję aby stworzyć nowy strumień, zbiegający znacznie szybciej do liczby π, przy użyciu transformacji Eulera:

$$F\ x\ y\ z = z - \frac{(y-z)^2}{x - 2y + z}.$$

Zadanie 5 (4p). Leniwe listy dwukierunkowe możemy wyrazić następującym typem.

```
type 'a dllist = 'a dllist_data lazy_t
and 'a dllist_data =
  { prev : 'a dllist
  ; elem : 'a
  ; next : 'a dllist
  }
```

Zdefiniuj następujące operacje na takich listach:

- `prev : 'a dllist -> 'a dllist`
- `elem : 'a dllist -> 'a`
- `next : 'a dllist -> 'a dllist`
- `of_list : 'a list -> 'a dllist`

Ostatnia z tych funkcji powinna tworzyć dwukierunkową listę cykliczną z podanej listy, o ile nie jest pusta. Zadbaj o to, by lista się nie *rozwarstwiała*, tzn. dla każdego węzła d utworzonej listy powinny zachodzić równości d `== prev (next` d`)` oraz d `== next (prev` d`)`, gdzie (`==`) oznacza równość fizyczną.

Zadanie 6 (2p). Zdefiniuj wartość `integers: int dllist` będącą nieskończoną leniwą listą dwukierunkową wszystkich liczb całkowitych. Również zadbaj o to, by lista się nie rozwarstwiała.

Zadanie 7 (4p). Zaproponuj własną implementację leniwości, definiując typ `'a my_lazy` oraz następujące operacje.

- `force : 'a my_lazy -> 'a` — działający analogicznie do `Lazy.force` z biblioteki standardowej.
- `fix : ('a my_lazy -> 'a) -> 'a my_lazy` — tworzący nowe leniwe wartości. Funkcja `fix` jako parametr przyjmuje funkcję która oblicza rozleniwioną wartość na podstawie leniwej wartości którą tworzymy. Pozwala to na tworzenie rekurencyjnych struktur danych, np.

```
let stream_of_ones = fix (fun stream_of_ones -> Cons(1, stream_of_ones))
```

Implementacja powinna sprawdzać, czy definiowane rekurencyjne wartości są *produktywne*, ale samo sprawdzanie powinno odbywać się też leniwie. Np. wyrażenie `fix (fun l -> force l)` powinno obliczyć się normalnie, ale `force (fix (fun l -> force l))` powinno zgłosić wyjątek.

Następnie przy pomocy tych operacji zdefiniuj strumień wszystkich liczb pierwszych.

Zadanie 8 (3p). Użyj mutowalnego stanu (odpowiednich komórek `ref`) aby poprawić wygodę pracy z systemem dowodzenia z poprzedniej listy. Bardziej konkretnie, zdefiniuj moduł `Interactive`, w którym:

- Przyjmij że możemy budować co najwyżej jeden dowód na raz
- Zdefiniuj typ `status` reprezentujący stan aktualnego dowodu (jeśli jakiś dowód budujemy)
- Zdefiniuj funkcję `status : unit -> status`, zwracającą aktualny stan dowodu
- Dla każdej funkcji z modułu `Proof` zdefiniuj jej wersję w module `Interact`, tak że jeśli przyjmowała ona argument typu `proof` lub `goal` to nie przyjmuje go (jeśli była jednoargumentowa to powinna przyjmować argument typu `unit`), a jeśli zwracała obiekt któregoś z tych typów — zwraca `unit`. Oczywiście, funkcje te powinny odpowiednio modyfikować stan aktywnego dowodu, rozpoczynać go lub kończyć, korzystając z mutowalnego stanu.

Docelowo dla przykładu z zad. 6 powinna być możliwa następująca interakcja (ładnie sformatowane odpowiedzi systemu wymagają zdefiniowania i zainstalowania odpowiedniego formatera, analogicznie do tych dla dowodów, formuł czy celów — ale nie są wymagane):

```
> proof [] {p → (p → q) → q}; status ();;
- : status =

========================================
p → (p → q) → q

> intro "H1"; intro "H2"; status ();;
- : status =

H2: p → q
H1: p
========================================
q

> apply_assm "H2"; status ();;
- : status =
There are 1 subgoals:
1: p()

> focus 1; apply_assm "H1"; status ();;
- : status =
No more subgoals

> let thm = qed ();;
val thm : theorem = ⊢ p → (p → q) → q

> status ();;
- : status =
There is no active proof
```