# Programowanie Funkcyjne 2023

Lista zadań nr 5

Na zajęcia 22 i 24 listopada 2023

**Zadanie 1 (3p).** 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 Fibonacci'ego 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 `fin` $n$ 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 (1p).** Zaimplementuj funkcję `fix` z poprzedniego zadania na dwa sposoby, które nie używają jawnej rekursji (tj. formy `let rec`):

- używając typów rekurencyjnych;
- używając mutowalnego stanu.

**Zadanie 3 (2p).** Liczby wymierne dodatnie z przedziału $(\frac{a}{b}, \frac{c}{d})$ można nawlec na nieskończone drzewo binarne w następujący sposób: w korzeniu umieszczamy liczbę $\frac{a+c}{b+d}$, zaś lewe i prawe poddrzewo konstruujemy rekurencyjnie odpowiednio dla przedziałów $(\frac{a}{b}, \frac{a+c}{b+d})$ oraz $(\frac{a+c}{b+d}, \frac{c}{d})$. Okazuje się, że tak skonstuowane drzewo dla przedziału $(\frac{0}{1}, \frac{1}{0})$ zawiera wszystkie nieskracalne ułamki dodatnie ($\frac{1}{0}$ reprezentuje nieskończoność). Zdefiniuj typ leniwych drzew oraz drzewo wszystkich liczb wymiernych dodatnich. Liczby wymierne możesz reprezentować jako ułamki, czyli pary liczb całkowitych.

**Zadanie 4 (3p).** 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 5 (2p).** Zdefiniuj wartość `integers: int dllist` będącą nieskończoną leniwą listą dwukierunkową wszystkich liczb całkowitych. Również zadbaj o to, by lista się nie rozwarstwiała.

**Zadanie 6 (3p).** 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.
**Wskazówka:** taka leniwa wartość może być mutowalną komórką pamięci, która przechowuje jedną z trzech rzeczy: odroczenie w postaci funkcji, spamiętaną wartość, lub informację o tym, że wartość jest właśnie obliczana i ponowna próba jej wymuszenia powinna zakończyć się błędem.

**Zadanie 7 (1p).** Przy pomocy leniwości z poprzedniego zadania zdefiniuj typ list leniwych, a następnie listę wszystkich liczb pierwszych. Możesz wzorować się kodem z wykładu.

**Zadanie 8 (3p).** Rozważmy następujący typ danych opisujący pewien podzbiór typów skończonych.

```
type _ fin_type =
| Unit : unit fin_type
| Bool : bool fin_type
| Pair : 'a fin_type * 'b fin_type -> ('a * 'b) fin_type
```

Zaimplementuj funkcję `all_values: 'a fin_type -> 'a Seq.t`, która dla podanego typu zwraca wszystkie jego elementy. Na przykład, wywołanie `all_values (Pair(Unit, Bool))` powinno zwrócić sekwencję dwóch par: `((), true)` oraz `((), false)`.

**Zadanie 9 (1p).** Rozszerz typ z poprzedniego zadania o konstruktory opisujące typ pusty i typ `Either.t`. Typ pusty nie jest zdefiniowany w bibliotece standardowej, więc powinieneś go zdefiniować samemu.

```
type empty = |
```

Następnie uzupełnij definicję funkcji `all_values` o obsługę nowych przypadków.

**Zadanie 10 (1p).** Rozbuduj typ z zadania 8 o konstruktor opisujący funkcje, a następnie uzupełnij definicję funkcji `all_values`. Przyjmij, że typ `fin_type` potrafi opisywać tylko funkcje totalne, tzn. takie, że dla poprawnych danych zawsze się zakończą i zwrócą poprawny wynik. Dodatkowo przyjmij, że dwie funkcje są równe, gdy dla wszystkich poprawnych argumentów dają te same wyniki. Wtedy typ `a -> b` będzie mieć dokładnie $b^a$ elementów, gdzie $a$ i $b$ to odpowiednio liczba elementów typów `a` i `b`.

**Wskazówka:** rozwiązanie prawie na pewno wymaga zdefiniowania drugiej funkcji wzajemnie rekurencyjnej z `all_values`, której typ będzie mieć negatywne wystąpienia zmiennej typowej będącej parametrem `fin_type`. W moim rozwiązaniu jest to funkcja `equal_at: 'a fin_type -> 'a -> 'a -> bool`.