

Programowanie Funkcyjne 2024

Lista zadań nr 5

Na zajęcia 12 i 14 listopada 2024

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 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 (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 skonstruowane 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). Używając drzewa z poprzedniego zadania zdefiniuj strumień wszystkich dodatnich liczb wymiernych. Użyj funkcji `Seq.interleave by` elegancko zaimplementować przechodzenie drzewa wszerz.

Zadanie 5 (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 $d == \text{prev } (\text{next } d)$ oraz $d == \text{next } (\text{prev } d)$, 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.

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 8 (2p). Przy pomocy leniwości z poprzedniego zadania zdefiniuj typ `list` leniwych, a następnie listę wszystkich liczb pierwszych. Możesz wzorować się kodem z wykładu.