Lista 6

Leniwa ewaluacja i memoizacja

Memoizacja

Terminem *memoizacja* określa się technikę optymalizacji polegającą na przechowywaniu wyników kosztownych obliczeniowo wywołań funkcji i zwrócenie przechowywanych, wcześniej obliczonych wartości, gdy nastąpi ponowne wywołanie funkcji z tym samym wejściem. Do przechowywania wyników można wykorzystać np. **tablice haszujące (OCaml, Scala).**

Zadania 1 i 2 należy wykonać obu językach programowania: OCaml oraz Scala.

Zadanie 1

a) Napisz rekurencyjną, binarną funkcję `stirling` obliczającą liczby Stirlinga drugiego rodzaju. Liczby te określają liczbę podziałów n-elementowego zbioru na m niepustych zbiorów. Funkcję tę można wyrazić następującym rekurencyjnym wzorem:

$$S(n, m) = S(n - 1, m - 1) + m \cdot S(n - 1, m)$$

Dodatkowo, należy pamiętać, że dla dowolnego n $S(n, 1) = 1$ oraz $S(n, m) = 1$ dla $n = m$.

b) Następnie, skonstruuj funkcję `memoized_stirling`, która wykorzystuje technikę memoizacji do zoptymalizowania rekurencyjnych wywołań funkcji. W tym celu zadbaj, aby dla dowolnej pary argumentów (*n*,*m*) funkcja wywołała się tylko raz, a wyniki zostały zapamiętane i w przypadku kolejnego wywołania pozyskane z pamięci.

Zadanie 2

Zdefiniuj funkcję `make_memoize`. Niech funkcja ta:

- przyjmuje dowolną (czystą) funkcję fun jako argument,
- zwraca funkcję, która będzie działać dokładnie tak samo jak fun, z tą różnicą, że będzie wywoływać oryginalną funkcję fun jeden raz dla danego argumentu,

przechowa wewnątrz wynik, a następnie będzie zwracać przechowany wynik za każdym razem, kiedy zostanie wywołana z tym samym argumentem.

Przetestuj funkcję na dowolnej, napisanej przez siebie funkcji o dużej złożoności obliczeniowej.

Leniwa ewaluacja

Leniwa ewaluacja (*call by need*) jest strategią, w ramach której wyrażenie nie jest ewaluowane do momentu pierwszego użycia (tzn. jest postponowane do momentu, kiedy nastąpi potrzeba zażądania wartości). W językach Scala i OCaml domyślnym sposobem jest zachłanna ewaluacja, ale przy użyciu słowa kluczowego *lazy* można ją odroczyć.

Zadanie 3.

Pokaż, jak działa leniwa ewaluacja w Scala oraz OCaml (z wykorzystaniem modułu Lazy). W tym celu:

- w środowisku REPL utwórz leniwą zmienną i przypisz do niej wartość poprzez wywołanie kosztownej funkcji (np. niezmemoizowaną wersję funkcji `stirling`),
- wypisz wartość zmiennej na ekran i pokaż, że ewaluacja następuje w sposób odroczony.

Strumienie

Struktury danych mogą być definiowane w sposób rekurencyjny. Przykładowo,

```
type 'a mylist = Nil | Cons of 'a * 'a mylist
```

powoduje zdefiniowanie typu algebraicznego realizującego listę typów ‘a , który posiada dwa warianty: Nil oraz Cons, przy czym ten druga zdefiniowany jest jako para typu ‘a * ’a mylist.

Taka definicja pozwala na na zdefiniowanie trójelementowej listy z elementami 1,2,3 w następujący sposób:

```
Cons (1, Cons (2, Cons(3, Nil)))
```

Strumienie są (potencjalnie nieskończonymi) ciągami, które zawierają elementy tego samego rodzaju. Obliczenie kolejnych części strumienia jest wykonywane "na życzenie", za każdym razem, gdy wymagana jest obecna wartość. Z tego powodu, strumienie stanowią *leniwe struktury danych*.
Implementacja strumienia realizującego nieskończony ciąŋ może zostać zrealizowana poprzez zdefiniowanie rekurencyjnego typu danych, który posiada sygnaturę:

```
type 'a sequence = Cons of 'a * (unit -> 'a sequence)
```

tzn. konstruuje pary, gdzie pierwszym elementem jest wartość elementu, a drugim *funkcja* odwzorowująca jednostkę w strumień (tzw. *thunk*).
W ten sposób, można przykładowo zdefiniować rekurencyjną funkcję, która rekurencyjnie generuje następniki poprzez parę:

```
let rec from n = Cons (n, fun () -> from (n + 1))
```

a na jej podstawie (potencjalnie nieskończony) strumień liczb naturalnych:

```
let natural = from 0
```

Zadanie 4.

a) W języku OCaml, skonstruuj strumień `bell` pozwalający na generowanie kolejnych liczb Bella. Liczby Bella są definicyjnie ściśle powiązane z liczbami Stirlinga drugiego rodzaju – n'tą liczbę Bella definiuje się jako sumę:

$$\sum_{k=0}^{n} S(n,k)$$

b) Napisz funkcje stream_head oraz stream_tail pozyskujące odpowiednio głowę i ogon ze strumienia. Przez głowę należy rozumieć n'ty element strumienia, a przez ogon, *thunk*, który będzie generował kolejne elementy.

c) Korzystając z napisanych wcześniej funkcji i struktur, napisz następujące funkcję:
   i) funkcja zwracająca listę kolejnych n elementów strumienia,
   ii) funkcja zwracająca listę n elementów ze strumienia, ale pomija co drugi element

iii) funkcja, która sprawia, że strumień pomija n elementów (elementy są pobierane i porzucane),

iv) funkcja, która przyjmuje strumień s_1 i s_2, liczbę naturalną n i zwraca strumień par kolejnych n elementów pobranych z obu strumieni postaci $[(s_1[0], s_2[0]), (s_1[1], s_2[1]), ..., (s_1[n], s_2[n])]$. Przetestuj funkcję na strumieniach `natural` oraz `bell`.

v) funkcja wyższego rzędu, która przyjmuje dowolną unarną funkcję f i strumień s a następnie zwraca strumień s' odwzorowujący funkcję f na kolejnych elementach strumienia s.