Programowanie Funkcyjne 2020

Lista zadań nr 3 dla grup mabi, mbu, ppo i efes

Na zajęcia 27 i 28 października 2020

Zadanie 1 (1p). Wyraź standardowe funkcje `length`, `rev`, `map`, `append`, `rev_append`, `filter`, `rev_map` operujące na listach za pomocą funkcji `List.fold_left` albo `List.fold_right` (wybierz badziej odpowiednią do każdego z przypadków).

Zadanie 2 (1p). Liczbę w zapisie dziesiętnym reprezentujemy w postaci listy cyfr. Na przykład lista `[3;7;8]` oznacza liczbę 378. Zaprogramuj funkcję ujawniającą liczbę całkowitą reprezentowaną przez podaną listę cyfr. Użyj przy tym a) jawnie rekursji ogonowej, b) funkcji `List.fold_left`.

Zadanie 3 (1p). Wielomiany o współczynnikach rzeczywistych reprezentujemy w postaci list współczynników w kolejności od najwyższej potęgi do najniższej. Np. `[1.;0.;-1.;2.]` oznacza wielomian $x^3 - x + 2$. Przyjmij, że lista pusta oznacza ten sam wielomian, co lista `[0.]`. Napisz funkcję

```
polynomial : float list -> float -> float
```

wykorzystującą schemat Hornera do wyznaczenia wartości wielomianu o podanych współczynnikach dla podanego argumentu. Zaprogramuj ją a) jawnie korzystając z rekursji ogonowej oraz b) używając funkcji `List.fold_left`.

Zadanie 4 (2p). Współczynniki wielomianu z zadania 3 umieszczamy teraz w kolejności rosnących potęg. Przykładowy wielomian z zadania 3 reprezentujemy teraz za pomocą listy `[2.,-1.,0.,1.]`. Zaprogramuj funkcję `polynomial` dla tej reprezentacji a) jawnie korzystając z rekursji (nieogonowej), b) używając funkcji `List.fold_right`, c) jawnie korzystając z rekursji ogonowej, d) używając funkcji `List.fold_left`.

Zadanie 5 (3p). Zauważ, że skoro w OCamlu parametry są przekazywane przez wartość, to funkcje *fold* zawsze wykonują obliczenie dla każdego elementu listy, nawet jeśli wynik jest znany wcześniej. Na przykład czas obliczenia wyrażenia

```
List.fold_left (&&) true [false;...;false]
```

jest proporcjonalny do długości listy, podczas gdy funkcja

```
let rec for_all xs =
  match xs with
  | []      -> true
  | x :: xs -> x && for_all xs
```

wywołana dla listy `[false;...;false]` zwraca wynik po wykonaniu jednego kroku (nie dochodzi do wywołania rekurencyjnego). Zatem do zaprogramowania obliczeń tego typu funkcje *fold* też nie są odpowiednie (inaczej jest w języku *non-strict*, takim jak Haskell, w którym funkcja `foldr` nadaje się do tego celu, bo jest inkrementacyjna, a funkcja `foldl` — nie, gdyż jest monolityczna).

Można jednak zaimplementować efektywną wersją funkcji `for_all` przy funkcji *fold*, ale wymaga to skorzystania z efektów sterowania, np. wyjątków. Używając wyjątków oraz funkcji `List.fold_left` zaimplementuj efektywną wersję następujących funkcji.

- `for_all : ('a -> bool) -> 'a list -> bool` — sprawdza czy wszystkie elementy listy spełniają podany predykat.
- `mult_list : int list -> int` — oblicza iloczyn elementów listy.

- `sorted : int list -> bool` — sprawdza czy podana lista jest posortowana rosnąco.

Twoja implementacja powinna przerywać przechodzenie po liście, gdy wynik jest już znany.

Zadanie 6 (3p). Funkcja jest napisana w stylu przekazywania kontynuacji (*continuation-passing style*, CPS), jeżeli przyjmuje dodatkowy argument funkcyjny zwany *kontynuacją*, który reprezentuje całą resztę obliczeń, jakie mają zostać przeprowadzone po powrocie z tej funkcji. W konsekwencji, funkcje w CPS-ie zwracają wynik wołając swoją kontynuację, a wszystkie wywołania w programie w CPS-ie są ogonowe. Na przykład, funkcja obliczająca silnię napisana w CPS-ie wygląda następująco.

```
let rec fact_cps n k =
  if n = 0 then k 1
  else fact_cps (n-1) (fun v -> k (n * v))
```

Aby wykonać funkcję napisaną w CPS-ie, należy podać kontynuację początkową. Zazwyczaj jest to funkcja identycznościowa.

```
let fact n = fact_cps n (fun x -> x)
```

Zaimplementuj wersję funkcji `fold_left`, która jest w stylu przekazywania kontynuacji. Jej pierwszy parametr, który jest funkcją, też powinien być w CPS-ie, zatem powinieneś otrzymać funkcję o następującej sygnaturze.

```
fold_left_cps :
  ('a -> 'b -> ('a -> 'c) -> 'c) -> 'a -> 'b list -> ('a -> 'c) -> 'c
```

Następnie przy jej pomocy zaimplementuj zwykłą funkcję `fold_left`.

Zadanie 7 (3p). Styl przekazywania kontynuacji, choć nie jest najwygodniejszy, jest bardzo atrakcyjny, bo pozwala wyrazić niemalże dowolny efekt sterowania, poprzez odpowiednie manipulowanie kontynuacjami. Wykonaj jeszcze raz polecenie z zadania 5, tym razem z użyciem funkcji `fold_left_cps` i bez użycia wyjątków.

Wskazówka: Funkcja przekazywana jako parametr do `fold_left_cps` powinna czasami porzucić swoją kontynuację.

Zadanie 8 (5p). Na stronie przedmiotu znajdują się pliki `proc.mli` oraz `proc.ml`, które implementują prostą bibliotekę do lekkich, kooperatywnych wątków. Biblioteka ta definiuje następujący typ

```
type ('a,'z,'i,'o) proc = ('a -> ('z,'i,'o) ans) -> ('z,'i,'o) ans,
```

który opisuje pojedynczy proces, który produkuje wartość typu `'a`, może czytać wartości typu `'i` ze swojego wejścia, i wypisywać wartości typu `'o` na swoje wyjście (`'z` jest typem ostatecznego wyniku obliczenia, patrz typ funkcji `run`). Zauważ, że typ `proc` jest typem obliczeń w CPS-ie, tzn. opisuje on funkcje które czekają na swoją kontynuację (typu `'a -> ('z,'i,'o) ans`). Biblioteka dostarcza trzy podstawowe operacje dla takich procesów: funkcje `recv : ('i,'z,'i,'o) proc` i `send : 'o -> (unit,'z,'i,'o) proc`, które odpowiednio odbierają dane z wejścia i wysyłają na wyjście (`recv` jest funkcją, bo oczekuje swojej kontynuacji), oraz operator `(<|>>) : ('z,'z,'i,'m) proc -> ('z,'z,'m,'o) proc -> ('a,'z,'i,'o) proc`) który *zastępuje* bieżący proces dwoma nowymi połączonymi ze sobą kanałem wiadomości typu `'m`. Dodatkowo, funkcja `run : ('a,'a,string,string) proc -> 'a` uruchamia podany proces, dla którego wejściem są kolejne wiersze ze standardowego wejścia, a napisy wysyłane na wyjście przekazywane są do standardowego wyjścia. Na przykład poniższy proces przekazuje swoje wejście bezpośrednio do wyjścia.

```
open Proc
let rec echo k =
  recv (fun v ->
  send v (fun () ->
  echo k))
```

Zatem obliczenie wyrażenia `run echo` nigdy się nie kończy, i wypisuje na ekranie wszystkie wiersze wprowadzone z klawiatury.

Zaimplementuj następujące procesy.

- `map : ('i -> 'o) -> ('a, 'z, 'i, 'o) proc` — proces, który nakłada podaną funkcję na wszystkie elementy przeczytane z wejścia. Np. obliczenie

  ```
  run (map String.length <|> map string_of_int)
  ```

  powinno po kolei wyświetlać długości wprowadzanych wierszy.
- `filter : ('i -> bool) -> ('a, 'z, 'i, 'i) proc` — proces, który przekazuje dalej tylko te wartości odczytane z wejścia, które spełniają podany predykat. Np.

  ```
  run (filter (fun s -> String.length s >= 5))
  ```

  powinno wyświetlać tylko te wiersze ze standardowego wejścia, które mają co najmniej 5 znaków.
- `nats_from : int -> ('a, 'z, 'i, int) proc` — proces, który dla danego n wysyła na wyjście wszystkie liczby naturalne zaczynając od n.
- `sieve : ('a, 'a, int, int) proc` — proces, który przesyła dalej pierwszą przeczytaną liczbę n, a następnie zamienia się w swoją kopię złożoną z procesem, który przepuszcza tylko liczby niepodzielne przez n (zastanów się w którą stronę lepiej jest złożyć te procesy). Takiego procesu powinno dać się użyć do wyświetlenia wszystkich liczb pierwszych:

  ```
  run (nats_from 2 <|> sieve <|> map string_of_int)
  ```

Zadanie 9 (1p). Wyjaśnij implementację biblioteki Proc z poprzedniego zadania.

Zadania 3,4 oraz 5 są kompilacją zadań przygotowanych przez Tomasza Wierzbickiego.