Programowanie Funkcyjne 2018

Lista zadań nr 5 dla grupy TWI: odroczenia, strumienie i inne nieskończone typy danych

Na zajęcia 22 listopada 2018

Zadanie 1 (7 pkt). Zaprogramuj moduł `Streams` o sygnaturze

```
module type STREAMS =
sig
   type 'a t = 'a cell Lazy.t
    and 'a cell = Nil | Cons of 'a * 'a t
   val hd : 'a t -> 'a
   val tl : 'a t -> 'a t
   val (++) : 'a t -> 'a t -> 'a t
   val init : (int -> 'a) -> 'a t
   val map : ('a -> 'b) -> ('a t -> 'b t)
   val map2 : ('a -> 'b -> 'c) -> ('a t -> 'b t -> 'c t)
   val combine : 'a t -> 'b t -> ('a * 'b) t
   val split : ('a * 'b) t -> 'a t * 'b t
   val filter : ('a -> bool) -> 'a t -> 'a t
   val find_opt : ('a -> bool) -> 'a t -> 'a option
   val concat : 'a t t -> 'a t
   val nth : int -> 'a t -> 'a
   val take : int -> 'a t -> 'a t
   val drop : int -> 'a t -> 'a t
   val list_of_stream : 'a t -> 'a list
   val stream_of_list : 'a list -> 'a t
end
```

Wszystkie funkcje, z wyjątkiem czterech ostatnich, powinny mieć podobne znaczenie, jak funkcje o tej samej nazwie z modułu `List`, przy czym funkcja `init` tworzy nieskończony strumień $\langle f\,0, f\,1, f\,2, \ldots\rangle$ wartości funkcji f podanej jej jako parametr, funkcja `take` tworzy skończony strumień będący prefiksem podanego strumienia o określonej długości, a `drop` odrzuca prefiks podanego strumienia o określonej długości. Obie funkcje, podobnie jak funkcja `nth`, zgłaszają wyjątek `Failure` z parametrem, odpowiednio, `"Streams.take"`, `"Streams.drop"` i `"Streams.nth"`, jeśli strumień nie ma dostatecznie wielu elementów oraz wyjątek `Invalid_argument` z takim samym parametrem jeśli podana liczba elementów bądź indeks są ujemne. Ostatnie dwie funkcje dokonują konwersji pomiędzy listami i strumieniami.

Które funkcje są monolityczne a które inkrementacyjne?

Powyższa sygnatura znajduje się w pliku `streams.mli` dostępnym na stronie zajęć. Opis sposobu tworzenia modułów i rozdzielnej kompilacji znajduje się w rozdziale 2.5 oficjalnego podręcznika języka. TL;DR: należy utworzyć plik `streams.ml` zawierający deklaracje odpowiadające wszystkim specyfikacjom umieszczonym w pliku `streams.mli` i skompilować oba pliki poleceniami

```
$ ocamlc -c streams.mli
$ ocamlc -c streams.ml
```

Skompilowany moduł ładuje się do środowiska interaktywnego poleceniem

```
# #load "streams.cmo";;
```

Zadanie 2 (2 pkt). Często przytaczana definicja nieskończonych strumieni:

```
type 'a bad_stream = BadNil | BadCons of 'a * 'a stream Lazy.t
```

ma poważną usterkę i działa czasami niezgodnie z intuicją i zamierzeniem programisty. Zajrzyj do artykułu: Philip Wadler, Walid Taha, David MacQueen, *How to Add Laziness to a Strict Language Without Even Being Odd*, Workshop on Standard ML, Baltimore, September 1998. Skonstruuj w Ocamlu przykład podobny do opisanego w artykule.

Zadanie 3 (2 pkt). Zaprogramuj funkcję

```
(><) : 'a t -> 'b t -> ('a * 'b) t
```

która tworzy produkt kartezjański dwu strumieni, tj. strumień wszystkich możliwych par elementów obu strumieni w porządku przekątniowym Cantora:

$$\langle x_1, x_2, x_3, \ldots\rangle \times \langle y_1, y_2, y_3, \ldots\rangle \quad = \quad \langle (x_1,y_1),(x_1,y_2),(x_2,y_1),(x_1,y_3),(x_2,y_2),(x_3,y_1),\ldots\rangle$$

Zauważ, że strumienie mogą być skończone bądź nie.

Zadanie 4 (1 pkt). Korzystając z implementacji strumieni z zadania 1 możemy zdefiniować nieskończony ciąg okresowy $\langle 1,2,3,1,2,3,1,2,3,\ldots\rangle$ np. tak:

```
let cyc3 = Streams.init (fun n -> n mod 3 + 1)
```

Mimo iż strumień jest nieskończony, to obliczenie się nie zapętli, bo tworzenie kolejnych elementów jest odraczane. Niestety na każdy element strumienia jest rezerwowana osobna pamięć i np. obliczenie

```
Streams.drop 10000002 cyc3
```

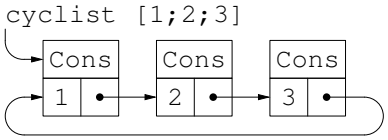
strasznie zapcha nam pamięć, choć zwraca strumień identyczny z oryginalnym. Wolelibyśmy zbudować w pamięci strukturę cykliczną, tzw. *cyklistę*, w której element 3 *wskazuje na* element 1. W Ocamlu możemy to zrobić korzystając z rekurencyjnego definiowania struktur:

```
let rec cyc3 = lazy (Cons (1, lazy (Cons (2, lazy (Cons (3, cyc3))))))
```

Zaprogramuj funkcję

```
cyclist : 'a list -> 'a Streams.t
```

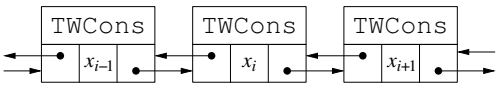
która tworzy cyklistę zawierającą elementy podanej niepustej listy. Jeśli jej argument jest listą pustą, to powinien być zgłoszony standardowy wyjątek `Invalid_argument`. Dla przykładu wywołanie `cycylist [1;2;3]` powinno zbudować w pamięci następującą strukturę:



Zadanie 5 (2 pkt). *Nieskończone strumienie dwukierunkowe* definiujemy następująco:

```
type 'a two_way_stream = 'a twcell Lazy.t
 and 'a twcell = TWCons of 'a two_way_stream * 'a * 'a two_way_stream
```

W strumieniu dwukierunkowym każdy element wskazuje na swojego następnika oraz poprzednika:



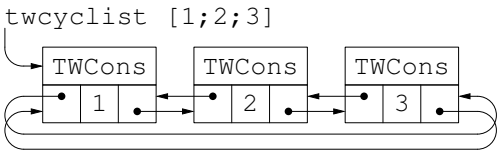
Zaprogramuj funkcje

```
tws_next : 'a two_way_stream -> 'a two_way_stream
tws_prev : 'a two_way_stream -> 'a two_way_stream
tws_elem : 'a two_way_stream -> 'a
```

Zadanie 6 (3 pkt). Zaprogramuj funkcję

```
twcyclist : 'a list -> 'a two_way_stream
```

która tworzy dwukierunkową cyklistę zawierającą elementy podanej niepustej listy. Jeśli jej argument jest listą pustą, to powinien być zgłoszony standardowy wyjątek `Invalid_argument`. Dla przykładu wywołanie `twcycylist [1;2;3]` powinno zbudować w pamięci następującą strukturę:



Zadanie 7 (2 pkt). Zaprogramuj funkcję

```
tws_init : (int -> 'a) -> 'a two_way_stream
```

tworzącą nieskończony strumień dwukierunkowy. Uwaga na pułapkę: wielokrotne naprzemienne obliczanie `tws_next` i `tws_prev` nie może tworzyć za każdym razem nowej kopii tego samego elementu! (Innymi słowy podana funkcja typu `int -> 'a` powinna być wywołana nie więcej niż raz dla tego samego indeksu).

Zadanie 8 (1 pkt). Zdefiniuj drzewa:

`inf_bin_tree` — binarne, w których mogą istnieć ścieżki nieskończonej długości,

`inf_tree` — w których wierzchołki mogą mieć nieskończenie wiele synów,

`inf_inf_tree` — w których wierzchołki mogą mieć nieskończenie wiele synów i w których mogą istnieć ścieżki nieskończonej długości.

Przyda się do tego pewnie moduł `Streams`. Zdefiniuj funkcje:

```
inf_bin_tree_init : ([bool] -> 'a) -> 'a inf_bin_tree
inf_tree_init : ([int] -> 'a) -> 'a inf_tree_init
inf_inf_tree_init : ([int] -> 'a) -> 'a inf_inf_tree_init
```

budujące takie drzewa. Ich argumentami są funkcje, które dla podanej ścieżki od korzenia ujawniają etykietę wierzchołka znajdującego się na jej końcu. Zdefiniuj następnie funkcje

```
const_inf_bin_tree : 'a -> 'a inf_bin_tree
const_inf_tree : 'a -> 'a inf_tree_init
const_inf_inf_tree : 'a -> 'a inf_inf_tree_init
```

budujące nieskończone drzewa posiadające w każdym wierzchołku tę samą podaną etykietę. Drzewa powinny być zacyklone i zużywać stałe miejsce w pamięci.

Wersja z poprawkami z 12 listopada 2018 15:33.