Programowanie Funkcyjne 2023

Lista zadań nr 6

Na zajęcia 29 listopada i 1 grudnia 2023

Zadanie 1 (3 pkt). Chcemy zdefiniować funkcję `sprintf` znaną z języka C, tak by np. wyrażenie

```
sprintf "Ala ma %d kot%s"
```

miało typ `int -> string -> string` i pozwalało zdefiniować funkcję

```
fun n -> sprintf "Ala ma %d kot%s." n
  (if n = 1 then "a" else if 1 < n && n < 5 then "y" else "ów").
```

Na pierwszy rzut oka wydaje się, że rozwiązanie tego zadania wymaga typów zależnych, ponieważ typ funkcji `sprintf` zależy od wartości pierwszego argumentu. Okazuje się jednak, że polimorfizm parametryczny wystarczy. Jeśli dyrektywy formatujące będą opisane pewnym typem `('a, 'b) format` z dwoma parametrami, to funkcja `sprintf` będzie miała typ `('a, string) format -> 'a`.

Zdefiniuj typ `('a, 'b) format`, funkcję `sprintf` oraz poniższe dyrektywy formatujące:

- `lit : string -> ('a, 'a) format` — stała napisowa (nie oczekuje żadnego parametru),
- `int : (int -> 'a, 'a) format` — liczba typu `int`,
- `str : (str -> 'a, 'a) format` — napis typu `string`,
- `(^^)` — konkatenacja dyrektyw formatujących.

Jeśli zrobisz to dobrze, to przykład podany na początku treści zadania może być zapisany następująco

```
sprintf (lit "Ala ma " ^^ int ^^ lit " kot" ^^ str ^^ lit ".").
```

Wskazówka: Dyrektywy powinny być funkcjami transformującymi kontynuacje, a operator `(^^)` zwykłym złożeniem takich funkcji. Na przykład `int` (po rozwinięciu definicji typu `format`) powinien mieć typ `(string -> 'a) -> string -> int -> 'a` (argumentem jest kontynuacja oczekująca napisu, ale o nieokreślonym typie odpowiedzi, natomiast wynikiem ma być kontynuacja oczekująca napisu i produkująca funkcję, która oczekuje na liczbę). Może okazać się potrzebna definicja pomocniczej funkcji `ksprintf` typu `('a, 'b) format -> (string -> 'b) -> 'a`

Zadanie 2 (3 pkt). W tym zadaniu postaramy się oddzielić składnię od semantyki dyrektyw formatujących. W tym celu zdefiniuj typ danych `('a, 'b) format`, który ma cztery konstruktory: `Lit`, `Int`, `Str` oraz `Cat` odpowiadające podstawowym dyrektywom formatującym oraz operacji konkatenacji. Oczywiście, żeby typy tych konstruktorów były odpowiednie, należy użyć GADT. Następnie napisz interpreter dyrektyw formatujących w postaci funkcji

```
ksprintf : ('a, 'b) format -> (string -> 'b) -> 'a
```

i użyj jej do napisania funkcji `sprintf`. Takie podejście ma sporo zalet: teraz dyrektywy formatujące są zwykłymi danymi, które można przetwarzać. W szczególności możemy dostarczyć alternatywny interpreter. Napisz funkcję `kprintf`, o sygnaturze

```
kprintf : ('a, 'b) format -> 'b -> 'a,
```

która od razu drukuje wynik na standardowe wyjście bez wcześniejszego konstruowania całego napisu. Użyj jej do zaimplementowania funkcji `printf`.

Zadanie 3 (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 4 (3p). Zaimplementuj wersję funkcji `fold_left`, która jest w stylu kontynuacyjnym (CPS, *continuation passing style*). Jej pierwszy parametr, który jest funkcją, też powinien być w stylu kontynuacyjnym, 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 5 (3p). Styl kontynuacyjny, choć nie jest najwygodniejszy, jest bardzo atrakcyjny, bo pozwala wyrazić niemalże dowolny efekt sterowania poprzez odpowiednie manipulowanie kontynuacjami. Wykonaj jeszcze raz polecenie z zadania 3, 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 6 (4p). 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. 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 `send`, które odpowiednio odbierają dane z wejścia i wysyłają na wyjście (`recv` jest funkcją, bo oczekuje swojej kontynuacji) oraz operator `(>|>)`, który zastępuje bieżący proces dwoma nowymi połączonymi ze sobą. Dodatkowo, funkcja `run` 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 po kolei 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 7 (1p). Wyjaśnij implementację biblioteki Proc z poprzedniego zadania.