Lista zadań nr 7

Zadanie 1. (2 pkt)

Zmodyfikuj interpreter tak, by operatory logiczne działały leniwie. Oznacza to, że jeśli po obliczeniu wartości pierwszego argumentu znamy już odpowiedź, nie obliczamy wartości drugiego argumentu. Przykładowo, wyrażenie `false && 10 / 0` powinno obliczyć się do fałszu.

Jaki związek ma leniwa semantyka z łącznością operatorów `&&` i `||`?

Wskazówka: Zadanie to można rozwiązać na dwa sposoby: modyfikując ewaluator albo potraktować operatory logiczne jako lukier syntaktyczny rozwijany do wyrażeń warunkowych.

Zadanie 2. (1 pkt)

Język PAIR z ćwiczeń (czyli język LET wyposażony w pary i typ unit), posiada dostatecznie dużo struktury, żeby zdefiniować listy. Konstruktor `::` możemy symulować przy użyciu pary, a `[]` przy użyciu wartości typu unit. Niestety, w tej chwili nie mamy sposobu, żeby odróżnić listę pustą od niepustej, co w OCamlu możemy zrobić dopasowaniem wzorca.

Dodaj do języka następujące konstrukcje sprawdzające typy wartości:

- `number? e` sprawdza czy wartością wyrażenia `e` jest liczba,
- `boolean? e` sprawdza czy wartością wyrażenia `e` jest wartość boolowska,
- `pair? e` sprawdza czy wartością wyrażenia `e` jest para,
- `unit? e` sprawdza czy wartością wyrażenia `e` jest unit.

Przykładowo, wyrażenie `number? (fst (1+1, false))` powinno obliczyć się do prawdy, a `unit? ((),())` – do fałszu.

Zadanie 3. (2 pkt)

Dodaj do języka lukier syntaktyczny dla list, podobny do tego w OCamlu. Przykładowo, wyrażenie `[e1; e2; e3]` powinno rozwijać się w fazie analizy składniowej (parsowania) do wyrażenia `(e1, (e2, (e3, ())))`.

Zadanie 4. (2 pkt)

Póki co, nie mamy jeszcze funkcji, a tym bardziej funkcji rekurencyjnych, ale możemy dodać do interpretera konstrukcję eliminującą listy poprzez ich zwijanie z prawej strony (niezbyt wygodny w użyciu odpowiednik funkcji `fold_right`). Składnia:

```
fold e1 with (x, acc) -> e2 and e3
```

gdzie:

- `e1`, `e2` i `e3` to wyrażenia
- `x` i `acc` to identyfikatory

Semantyka jest taka sama jak Ocamlowe `fold_right (fun x acc -> e2) e1 e3`. Na przykład, sumowanie elementów na liście można zapisać następująco:

```
fold (1, (10, (100, ()))) with (x, y) -> x + y and 0
```

Zadanie 5. (2 pkt)

Używając konstrukcji `fold` z poprzedniego zadania, napisz w naszym języku program, który odwraca listę.

Innymi słowy, zdefiniuj ciąg znaków `prog` taki, że następujący program w OCamlu odwraca listę intów:

```
let prog_of_list (xs : int list) : string = List.fold_right
  (fun x s -> "(" ^ string_of_int x ^ "," ^ s ^ ")")
  xs "()"

let rec list_of_val (v : value) : int list =
  match v with
  | VPair (VInt n, vs) -> n :: list_of_val vs
  | VUnit -> []
  | _ -> failwith "error"

let reverse (xs : int list) : int list =
  ("fold" ^ prog_of_list xs ^ prog) |> interp |> list_of_val
```

Zbadaj różnicę czasu wykonania powyższej definicji `reverse` i standardowej funkcji `List.reverse`.

Zadanie 6. (1 pkt)

Na wykładzie padła propozycja alternatywnej wersji semantyki dla konstrukcji `let`:

```
let rec eval (e : expr) : value =
  match e with
  | ...
  | Let (x, e1, e2) -> eval (subst x e1 e2)
```

W ten sposób dostalibyśmy leniwą semantykę `let`-a i uniknęlibyśmy potrzeby reifikacji wartości do wyrażeń.

Niestety, przy aktualnej definicji podstawień (funkcja `subst`) taka definicja byłaby nieprawidłowa. Dlaczego? Napisz program, który przy użyciu powyższej reguły nie zachowuje się zgodnie z oczekiwaniami.

Zadanie 7. (2 pkt)

Popraw definicję interpretera tak, żeby ewaluator działał prawidłowo przy zastosowaniu reguły z poprzedniego zadania.