TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Lehrstuhl für Sprachen und Beschreibungsstrukturen
Einführung in die Informatik 2
Prof. Dr. Helmut Seidl, Ralf Vogler, Stefan Schulze Frielinghaus

WS 2015/16
Übungsblatt 6

Aufgabe 6.1 [13 Punkte] **OCaml-Hausaufgabe: Listen und Records**

Zur Abgabe melden Sie sich mit Ihrer TUM-Kennung auf `https://vmnipkow3.in.tum.de` an und laden ihre `ha2.ml`-Datei (nicht umbenennen!) hoch. Die Ergebnisse der Tests sind sichtbar sobald die Abgabe abgearbeitet wurde. Die endgültige Bewertung erfolgt aber erst nach der Frist.

In der Datei gibt es drei Module mit Funktionen die zu implementieren sind:

a) `MyList`: mehr Funktionen auf polymorphen und Integer-Listen,

b) `NonEmptyList`: einige Funktionen für einen Listen-Typ der keine leeren Listen erlaubt,

c) `Db`: nutzen Sie Ihre oben definierten Funktionen um Berechnungen auf Listen von Records durchzuführen.

Lösungsvorschlag 6.1

Siehe Moodle.

Aufgabe 6.2 OCaml-Tutoraufgabe: Summentypen

a) OCaml hat `let` und `let rec` — was sind die Unterschiede? In Haskell sind `let` immer rekursiv — was ist dadurch nicht mehr möglich?

b) Implementieren Sie folgende Funktionen. Welchen Vorteil hat das Einführen neuer Typen?

```
type n = float
type k = K of n (* SI: Kelvin *)
type c = C of n (* Celsius *)
let c_to_k : c -> k = ??
let k_to_c : k -> c = ??
```

c) Implementieren Sie für folgende Typen eine Funktion zum Auswerten von arithmetischen Ausdrücken (mit m können Variablen Werten zugewiesen werden):

```
type c = int (* only integer constants *)
and v = string (* variables *)
and m = v -> c (* environment mapping variables to values *)
and unop = Neg  (* unary operations *)
and binop = Add | Sub | Mul | Div (* binary operations *)
and e = C of c | V of v | Unop of unop*e | Binop of binop*e*e (* expressions *)

(* evaluate expressions *)
let rec eval_e : m -> e -> c = ??
```

d) Wir erweitern das vorherige Schema nun um Anweisungen (Zuweisungen und Ausgabe von Variablen). Implementieren Sie nun Funktionen zum Auswerten von Anweisungen und ganzen Programmen (Liste von Anweisungen).

```
type c ...
and s = Asn of v*e | Print of v (* statements: assignment, print variable *)
and p = s list (* a program is a list of statements *)

(* evaluate a statement and return environment *)
let eval_s : m -> s -> m = ??

(* evaluate a program (with side-effects) and return environment *)
let eval_p : m -> p -> m = ??
```

Lösungsvorschlag 6.2

a) `let v = e1 in e2` wertet `e1` aus und bindet den Wert an `v` in `e2`. `let rec v = e1 in e2` wertet `e1` aus und bindet den Wert an `v` in `e1` und `e2`. Der Ausdruck `let x=1 in let x=x+1 in x` liefert in OCaml 2, führt aber zu Nicht-Terminierung in Haskell.

b) Wir können nun ausdrücken dass eine Funktion eine bestimmte Einheit erwartet und dies den Compiler überprüfen lassen.

```
type n = float (* we use float as a base numeric type *)
type k = K of n (* SI: Kelvin *)
type c = C of n (* Celsius *)
(* type f = F of n (* Fahrenheit *) *)

(* type safe conversion *)
let c_to_k (C x) = K (x +. 273.15)
let k_to_c (K x) = C (x -. 273.15)
(* some constants *)
let freezing = C 0.
let absolute_zero = K 0.
let body = C 37.
let white_point = K 6500.

(* now we want a temperature type with both variants... *)
type temp = TK of k | TC of c (* temperature *)
let to_k = function TK x -> x | TC x -> c_to_k x
let to_c = function TK x -> k_to_c x | TC x -> x
```

c)

d)
```
type c = int (* only integer constants *)
and v = string (* variables *)
and m = v -> c (* environment mapping variables to values *)
and unop = Neg  (* unary operations *)
and binop = Add | Sub | Mul | Div (* binary operations *)
and e = C of c | V of v | Unop of unop*e | Binop of binop*e*e (* expressions *)
and s = Asn of v*e | Print of v (* statements: assignment, print variable *)
and p = s list (* a program is a list of statements *)
[@@deriving show] (* alternatively write your own *)
```

```
let unop = function Neg -> ( * ) (-1)
let binop = function | Add -> (+) | Sub -> (-) | Mul -> ( * ) | Div -> (/)
(* order of operations is given by the tree... *)

(* evaluate expressions *)
let rec eval_e m = function
  | C x -> x
  | V x -> m x
  | Unop (o,e) -> unop o (eval_e m e)
  | Binop (o,e1,e2) -> binop o (eval_e m e1) (eval_e m e2)

open Printf
(* evaluate a statement and return environment *)
let eval_s m = function
  | Print v -> printf "V %s = %i\n" v (m v); m
  | Asn (v,e) -> fun v' -> if v'=v then eval_e m e else m v'

(* evaluate a program (with side-effects) and return environment *)
let eval_p = List.fold_left eval_s

(* empty mapping fails when undefined variables ase accessed *)
let m v = failwith (v^" is undefined!")
let () =
  (* test expressions *)
  let test1 = Binop (Add, C 1, C 1) in
  let test2 = Binop (Mul, C 21, Binop (Sub, C 1, Unop (Neg, C (-1)))) in
  let test3 = Binop (Add, V "x", V "y") in
  let test_e e m =
    printf "evaluating expression %s = %i\n\n" (show_e e) (eval_e m e)
  in
  test_e test1 m;
  test_e test2 m;
  test_e test3 (function "x" -> 1 | "y" -> 2 | v -> m v);
  let test4 = [Asn ("x", C 1); Asn ("y", C 2); Asn ("z", test3); Print "z"] in
  let test_p p m =
    printf "evaluating program %s:\n" (show_p p);
    let _ = eval_p m p in ()
  in
  test_p test4 m;
```