

Ozyegin University

CS 321 Programming Languages

Sample Problems on Type Checking

Reference

Typing rules of the Deve language are given below.

$$\frac{}{\rho \vdash i : \text{int}} \quad (\text{rule 1}) \qquad \frac{}{\rho \vdash b : \text{bool}} \quad (\text{rule 2}) \qquad \frac{\rho(x) = \tau}{\rho \vdash x : \tau} \quad (\text{rule 3})$$

$$\frac{\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 + e_2 : \text{int}} \quad (\text{rule 4}) \quad (\text{and similarly for } -, *, /)$$

$$\frac{\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 < e_2 : \text{bool}} \quad (\text{rule 5}) \quad (\text{and similarly for } \leq)$$

$$\frac{\rho \vdash e_1 : \tau_1 \quad \rho \vdash e_2 : \tau_2}{\rho \vdash (e_1, e_2) : (\tau_1 \times \tau_2)} \quad (\text{rule 6})$$

$$\frac{\rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : \tau \quad \rho \vdash e_3 : \tau}{\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \quad (\text{rule 7})$$

$$\frac{\rho \vdash e_1 : \tau_1 \quad [x \mapsto \tau_1] + \rho \vdash e_2 : \tau_2}{\rho \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2} \quad (\text{rule 8})$$

$$\frac{[x \mapsto \tau_1] + \rho \vdash e : \tau_2}{\rho \vdash \text{fun } (x : \tau_1) \rightarrow e : (\tau_1 \rightarrow \tau_2)} \quad (\text{rule 9})$$

$$\frac{\rho \vdash e_1 : (\tau_2 \rightarrow \tau_1) \quad \rho \vdash e_2 : \tau_2}{\rho \vdash e_1 e_2 : \tau_1} \quad (\text{rule 10})$$

$$\frac{\rho \vdash e_1 : (\tau_1 \times \tau_2) \quad [x \mapsto \tau_1, y \mapsto \tau_2] + \rho \vdash e_2 : \tau}{\rho \vdash \text{match } e_1 \text{ with } (x, y) \rightarrow e_2 : \tau} \quad (\text{rule 11})$$

$$\frac{[f \mapsto (\tau_1 \rightarrow \tau_2), x \mapsto \tau_1] + \rho \vdash e_1 : \tau_2 \quad [f \mapsto (\tau_1 \rightarrow \tau_2)] + \rho \vdash e_2 : \tau}{\rho \vdash \text{let rec f } (x : \tau_1) : \tau_2 = e_1 \text{ in } e_2 : \tau} \quad (\text{rule 12})$$

The `typeOf` function is given below.

```

type typ = IntTy
         | BoolTy
         | PairTy of typ * typ
         | FunTy of typ * typ

(* typeOf: exp -> (string * typ) list -> typ *)
let rec typeOf e tyEnv =
  match e with
  | CstI i -> IntTy
  | CstB b -> BoolTy
  | Var x -> lookup x tyEnv
  | Binary(op, e1, e2) ->
    let t1 = typeOf e1 tyEnv in
    let t2 = typeOf e2 tyEnv in
    (match op, t1, t2 with
     | "+", IntTy, IntTy -> IntTy
     | "-", IntTy, IntTy -> IntTy
     | "*", IntTy, IntTy -> IntTy
     | "/", IntTy, IntTy -> IntTy
     | "<", IntTy, IntTy -> BoolTy
     | "<=", IntTy, IntTy -> BoolTy
     | ",", _, _ -> PairTy(t1, t2)
     | _ -> failwith ("Bad use of the binary operator: " ^ op))
  | LetIn(x, e1, e2) ->
    let t = typeOf e1 tyEnv
    in let tyEnv' = (x, t)::tyEnv
       in typeOf e2 tyEnv'
  | LetRec(f, (x,t1), retTy, e1, e2) ->
    let tBody = typeOf e1 ((f, FunTy(t1,retTy))::(x,t1)::tyEnv)
    in if tBody = retTy then
       typeOf e2 ((f, FunTy(t1,retTy))::tyEnv)
     else failwith "Return type of the rec. function should agree with the type of the body."
  | If(e1, e2, e3) -> (match typeOf e1 tyEnv with
                       | BoolTy -> let t2 = typeOf e2 tyEnv in
                                   let t3 = typeOf e3 tyEnv in
                                   if t2 = t3 then t2
                                   else failwith "Branch types of an if-then-else must agree."
                       | _ -> failwith "Condition should be a bool.")
  | MatchPair(e1, x, y, e2) ->
    (match typeOf e1 tyEnv with
     | PairTy(t1, t2) -> typeOf e2 ((x,t1)::(y,t2)::tyEnv)
     | _ -> failwith "Pair pattern matching works on pair values only (obviously)!"
    )
  | Fun((x, t), e) ->
    let tBody = typeOf e ((x,t)::tyEnv)
    in FunTy(t, tBody)
  | App(e1, e2) ->
    (match typeOf e1 tyEnv with
     | FunTy(t2, t1) ->
       if t2 = typeOf e2 tyEnv then t1
       else failwith "Function parameter type should agree with the argument type."
     | _ -> failwith "Application wants to see a function!")
  )

```

Questions

1. For each of the program points below, write down the *type environment*. Assume that we start with the empty environment.

(a) `let x = 9 in`
 (program point 1 *)*
 `let f y = x * y in`
 (program point 2 *)*
 `let x = 4 in`
 (program point 3 *)*
 `let y = 7 in`
 (program point 4 *)*
 `f x`

(b) `let x = 9 in`
 (program point 1 *)*
 `let y = let x = 13 in`
 (program point 2 *)*
 `x + 2 in`
 (program point 3 *)*
 `y + x`

(c) `let add x y = x + y in`
 (program point 1 *)*
 `let foo = add 10 in`
 (program point 2 *)*
 `let baz = foo 20 in`
 (program point 3 *)*
 `baz`

2. Suppose we had “min” and “max” as binary operators. Define typing rules for them and also show how the `typeOf` function would be implemented.

3. Suppose we had “=” as a binary operator for equality checking. Define typing rules for this operator and also show how the `typeOf` function would be implemented. “=” works for between any pair of values as long as they have the same type. E.g. These are fine: `4 = 6`, `(4<5) = true`, `(4,5) = (3+1,10/2)`

4. Suppose we had unary operators in the language, represented with the `Unary of string * exp` constructor. Define typing rules for the “fst” and “snd” unary operators and also show how the `typeOf` function would be implemented.

5. Using the Deve typing rules, show the type derivation tree for the type judgment given below.

$$[] \vdash \text{let } x = 1 \text{ in } x < 2 : \text{bool}$$

6. Using the Deve typing rules, show the type derivation tree for the type judgment given below.

$$[] \vdash \text{let } z = 1 < 2 \text{ in if } z \text{ then } 3 \text{ else } 4 : \text{int}$$

7. Each of the following expressions has a problem that prevents it from being accepted by the Deve type system. In other words, it is impossible to construct a type derivation tree. Explain at which rule your attempt to build a tree would fail, and why.

- $[y \mapsto \text{bool}] \vdash y < 42 : \text{bool}$

- $[] \vdash \text{let } x = 17 \text{ in } x \text{ } 25 : \text{int}$

- $[x \mapsto \text{int}] \vdash \text{if } x < 0 \text{ then } 54 \text{ else false} : \text{int}$

8. Using the Deve typing rules, show the type derivation tree for the type judgment given below.

$[] \vdash \text{let } x = 3+5 \text{ in if } x < 0 \text{ then } (\text{fun } n \rightarrow n*2) \text{ else } (\text{fun } z \rightarrow z-x) : \text{int} \rightarrow \text{int}$

9. Using the Depe typing rules, show the type derivation tree for the type judgment given below.

$[] \vdash \text{let rec fib (n:int) :int = if n<2 then n else fib(n-1) + fib(n-2) in fib 42 : int}$

10. What are the types of the following OCaml expressions? Give types that are as general as possible. You may use Greek letters (e.g. $\alpha, \beta, \gamma, \delta$ etc.) or quoted letters (e.g. 'a', 'b', 'c', 'd' etc.) for polymorphic types. If there is an error, write ERROR and explain the problem.

(a) `let q3 f = f(f(f(1)))`

(b) `let q4 f n = f(f(f(n)))`

(c) `let q6 p1 p2 = (snd p2, fst p2, snd p1, fst p1)`

(d) `let rec graph f lst =
 match lst with
 | [] -> []
 | (x::xs) -> (x,f x) :: graph f xs`

(e) `let rec fold f a lst =
 match lst with
 | [] -> a
 | x::xs -> fold f (f a x) xs`

(f) `type 'a tree = Leaf of 'a
 | Node of ('a * 'a tree * 'a tree)

let rec flatten t =
 match t with
 | Leaf n -> [n]
 | Node(n, t1, t2) -> flatten t1 @ (n::flatten t2);;`

(g) `let p = (34, true);;`

(h) `let f x = (x, (x+5, x > 0));;`

(i) `let f x y = (y, x);;`

(j) `let f (x,y) = (y, x);;`

(k) `let f x = List.map (fun y -> y*y) x;;`

(l) `let f x g b = List.fold_left g b x;;`

(m) `let rec f p =
 match p with
 | [] -> []
 | x::xs -> (x+x)::f xs;;`

(n) `let f = let max n m = if n - m > 0 then n else m
 in max 10;;`

(o) `let f g x = g(g(g(x)));;`

(p) `let apply f x y = f x y;;`

(q) `let compose f g x = f(g(x));;`

(r)

```
let rec g f a lst =  
  match lst with  
  | [] -> a  
  | x::xs -> g f (f a x) xs;;
```

(s)

```
let f x = if x > 0 then Some x else None;;
```

(t)

```
let rec last p lst =  
  match lst with  
  | [] -> None  
  | x::xs -> (match last p xs with  
              | None -> if p x then Some x else None  
              | Some y -> Some y);;
```

(u)

```
let rec f lst a =  
  match lst with  
  | [] -> a  
  | x::xs -> f xs (x::a);;
```

(v)

```
let rec gee f xs =  
  match xs with  
  | [] -> []  
  | y::ys -> (y, f y)::(gee f ys)
```

(w)

```
let rec f n = f (n+1)
```

(x)

```
let rec foo x y z =  
  match y with  
  | [] -> z * z  
  | b::bs -> x b (foo x z bs)
```