Ozyegin University
CS 321 Programming Languages
Sample Problems on Type Checking

Reference

Typing rules of the Deve language are given below.

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

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

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

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

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

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

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

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

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

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

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 bofy."
  | 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
```

Solution:

- 1: $[\mathtt{x} \mapsto \mathit{int}]$
- 2: $[\mathtt{f} \mapsto (\mathit{int} \to \mathit{int}), \mathtt{x} \mapsto \mathit{int}]$
- 3: $[\mathtt{x} \mapsto \mathit{int}, \mathtt{f} \mapsto (\mathit{int} \to \mathit{int}), \mathtt{x} \mapsto \mathit{int}]$
- 4: $[\mathtt{y} \mapsto \mathit{int}, \mathtt{x} \mapsto \mathit{int}, \mathtt{f} \mapsto (\mathit{int} \to \mathit{int}), \mathtt{x} \mapsto \mathit{int}]$

(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
```

Solution:

- 1: $[\mathtt{x} \mapsto \mathit{int}]$
- 2: $[\mathtt{x} \mapsto \mathit{int}, \mathtt{x} \mapsto \mathit{int}]$
- 3: $[\mathtt{y} \mapsto \mathit{int}, \mathtt{x} \mapsto \mathit{int}]$

(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
```

Solution:

- 1: $[\mathtt{add} \mapsto (\mathit{int} \to \mathit{int} \to \mathit{int})]$
- 2: $[\mathtt{foo} \mapsto (\mathit{int} \to \mathit{int}), \mathtt{add} \mapsto (\mathit{int} \to \mathit{int} \to \mathit{int})]$
- 3: $[\mathtt{baz} \mapsto \mathit{int}, \mathtt{foo} \mapsto (\mathit{int} \to \mathit{int}), \mathtt{add} \mapsto (\mathit{int} \to \mathit{int} \to \mathit{int})]$

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.

Solution:

$$\frac{\rho \vdash e_1 : \text{int} \qquad \rho \vdash e_2 : \text{int}}{\rho \vdash \mathtt{min(}e_1\mathtt{,}e_2\mathtt{)} : \text{int}}$$ (and similarly for `max`)

For the implementation, add a new Binary operator case for each.

```
| "min", IntTy, IntTy -> BoolTy
| "max", IntTy, IntTy -> BoolTy
```

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)`

Solution:

$$\frac{\rho \vdash e_1 : \tau \qquad \rho \vdash e_2 : \tau}{\rho \vdash e_1 \texttt{=} e_2 : \text{bool}}$$

For the implementation, add a new Binary operator case.

```
| "=", _, _ when t1 = t2 -> BoolTy
```

Or:

```
| "=", _, _ -> if t1 = t2 then BoolTy
               else failwith "Bad use of = operator."
```

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.

Solution:

$$\frac{\rho \vdash e : (\tau_1 \times \tau_2)}{\rho \vdash \texttt{fst(}e\texttt{))} : \tau_1} \qquad \frac{\rho \vdash e : (\tau_1 \times \tau_2)}{\rho \vdash \texttt{snd(}e\texttt{))} : \tau_2}$$

For the implementation, add a new case.

```
let rec typeOf e tyEnv =
  match e with
  ...
  | Unary(op, e)  ->
     let t = typeOf e tyEnv in
     (match op, t1, t2 with
      | "fst", PairTy(t1, t2) -> t1
      | "snd", PairTy(t1, t2) -> t2
      | _ -> failwith ("Bad use of the unary operator: " ^ op)
     )
```

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

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

Solution:

$$\dfrac{\dfrac{}{[\,] \vdash \texttt{1} : \mathit{int}}\,(1) \qquad \dfrac{\dfrac{[\texttt{x} \mapsto \mathit{int}](x) = \mathit{int}}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{x} : \mathit{int}}\,(3) \qquad \dfrac{}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{2} : \mathit{int}}\,(1)}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{x < 2} : \mathit{bool}}\,(5)}{[\,] \vdash \texttt{let x = 1 in x < 2} : \mathit{bool}}\,(8)$$

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

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

Solution:

$$\dfrac{\dfrac{}{[\,] \vdash \texttt{1} : \mathit{int}}\,(1) \qquad \dfrac{}{[\,] \vdash \texttt{2} : \mathit{int}}\,(1)}{[\,] \vdash \texttt{1 < 2} : \mathit{bool}}\,(5) \qquad \dfrac{\dfrac{[\texttt{z} \mapsto \mathit{bool}](\texttt{z}) = \mathit{bool}}{[\texttt{z} \mapsto \mathit{bool}] \vdash \texttt{z} : \mathit{bool}}\,(3) \qquad \dfrac{}{[\texttt{z} \mapsto \mathit{bool}] \vdash \texttt{3} : \mathit{int}}\,(1) \qquad \dfrac{}{[\texttt{z} \mapsto \mathit{bool}] \vdash \texttt{4} : \mathit{int}}\,(1)}{[\texttt{z} \mapsto \mathit{bool}] \vdash \texttt{if z then 3 else 4} : \mathit{int}}\,(7)$$
$$\text{(8)}\ \dfrac{\qquad}{[\,] \vdash \texttt{let z = 1<2 in if z then 3 else 4} : \mathit{int}}$$

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.

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

Solution: Fails when attempting to use rule 3:

$$\dfrac{\dfrac{\text{☹}}{[\texttt{y} \mapsto \mathit{bool}] \vdash \texttt{y} : \mathit{int}}\,(3) \qquad \dfrac{}{[\texttt{y} \mapsto \mathit{bool}] \vdash \texttt{42} : \mathit{int}}\,(1)}{[\texttt{y} \mapsto \mathit{bool}] \vdash \texttt{y < 42} : \mathit{bool}}\,(5)$$

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

Solution:

$$\dfrac{\dfrac{}{[\,] \vdash \texttt{17} : \mathit{int}}\,(1) \qquad \dfrac{\dfrac{\text{☹}}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{x} : (\mathit{int} \to \mathit{int})}\,(3) \qquad \dfrac{}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{25} : \mathit{int}}\,(1)}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{x 25} : \mathit{int}}\,(10)}{[\,] \vdash \texttt{let x = 17 in x 25} : \mathit{int}}\,(8)$$

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

Solution:

$$\dfrac{\dfrac{\dfrac{[\texttt{x} \mapsto \mathit{int}](\texttt{x}) = \mathit{int}}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{x} : \mathit{int}}\,(3) \qquad \dfrac{}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{0} : \mathit{int}}\,(1)}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{x < 0} : \mathit{bool}}\,(5) \qquad \dfrac{}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{54} : \mathit{int}}\,(1) \qquad \dfrac{\text{☹}}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{false} : \mathit{int}}\,(2)}{[\texttt{x} \mapsto \mathit{int}] \vdash \texttt{if x < 0 then 54 else false} : \mathit{int}}\,(7)$$

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

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

Solution: This was done in the lecture. Check your notes, or buy coffee for a friend who takes notes.

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

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

Solution: In the solution below, ρ_1 stands for the following environment: $[\texttt{fib} \mapsto (int \to int), \texttt{n} \mapsto int]$. Also, ρ_2 stands for the following environment: $[\texttt{fib} \mapsto (int \to int)]$.

$$\dfrac{\dfrac{\dfrac{\dfrac{\rho_1(\texttt{n}) = int}{\rho_1 \vdash \texttt{n} : int}\,(3) \quad \dfrac{}{\rho_1 \vdash \texttt{2} : int}\,(1)}{\rho_1 \vdash \texttt{n<2} : bool}\,(5) \quad \dfrac{\rho_1(\texttt{n}) = int}{\rho_1 \vdash \texttt{n} : int}\,(3) \quad \mathcal{C}}{\rho_1 \vdash \texttt{if n<2 then n else fib(n-1) + fib(n-2)} : int}\,(7) \quad \dfrac{\dfrac{\rho_2(fib) = (int \to int)}{\rho_2 \vdash \texttt{fib} : (int \to int)}\,(3) \quad \dfrac{}{\rho_2 \vdash \texttt{42} : int}\,(1)}{\rho_2 \vdash \texttt{fib 42} : int}\,(10)}{[\,] \vdash \texttt{let rec fib (n:int) :int = if n<2 then n else fib(n-1) + fib(n-2) in fib 42} : int}\,(12)$$

And tree $\mathcal{C}$ is:

$$\dfrac{\dfrac{\dfrac{\rho_1(\texttt{fib}) = (int \to int)}{\rho_1 \vdash \texttt{fib} : (int \to int)}\,(3) \quad \dfrac{\dfrac{\rho_1(\texttt{n}) = int}{\rho_1 \vdash \texttt{n} : int}\,(3) \quad \dfrac{}{\rho_1 \vdash \texttt{1} : int}\,(1)}{\rho_1 \vdash \texttt{n-1} : int}\,(4)}{\rho_1 \vdash \texttt{fib(n-1)} : int}\,(10) \quad \dfrac{\dfrac{\rho_1(\texttt{fib}) = (int \to int)}{\rho_1 \vdash \texttt{fib} : (int \to int)}\,(3) \quad \dfrac{\dfrac{\rho_1(\texttt{n}) = int}{\rho_1 \vdash \texttt{n} : int}\,(3) \quad \dfrac{}{\rho_1 \vdash \texttt{2} : int}\,(1)}{\rho_1 \vdash \texttt{n-2} : int}\,(4)}{\rho_1 \vdash \texttt{fib(n-2)} : int}\,(10)}{\rho_1 \vdash \texttt{fib(n-1) + fib(n-2)} : int}\,(4)$$

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.

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

Solution: `(int -> int) -> int`

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

Solution: `('a -> 'a) -> 'a -> 'a`

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

Solution: `'a * 'b -> 'c * 'd -> 'd * 'c * 'b * 'a`

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

Solution: `('a -> 'b) -> 'a list -> ('a * 'b) list`

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

Solution: `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

(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);;
```

Solution: `'a tree -> 'a list`

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

Solution: `int * bool`

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

Solution: `int -> int * (int * bool)`

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

Solution: `'a -> 'b -> 'b * 'a`

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

Solution: `'a * 'b -> 'b * 'a`

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

Solution: `int list -> int list`

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

Solution: `'a list -> ('b -> 'a -> 'b) -> 'b -> 'b`

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

Solution: `int list -> int list`

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

Solution: `int -> int`

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

Solution: `('a -> 'a) -> 'a -> 'a`

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

Solution: `('a -> 'b -> 'c) -> 'a -> 'b -> 'c`

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

Solution: `('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`

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

Solution: `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

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

Solution: `int -> int option`

(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);;
```

Solution: `('a -> bool) -> 'a list -> 'a option`

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

Solution: `'a list -> 'a list -> 'a list`

11. Give the most general (polymorphic) types for the following functions. If the function is not typable, write ERROR and explain the problem.

(a)
```
let f id = (id 5, id true)
```

Solution: ERROR. Function parameter cannot be polymorphic.

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

Solution: `('a -> 'b) -> 'a list -> ('a * 'b) list`

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

Solution: `int -> 'a`