- Final Exam: 2013/11/06 (Wednesday) 1:10–3:10pm
- Review Tutorial: 2013/11/01 (this Friday) 8:00am
- Practise Haskell/Scheme and Prolog by yourself, e.g. solve the Eight Queens Problem.

Solution by Xiao Jia

Homework 6

Problem 1. (55 points) A language P is defined as follows.

```
e ::= x | n | true | false | succ | pred | iszero |
      if e then e else e | fn x => e | e e | rec x => e | (e)
```

The above grammar is quite ambiguous. We can resolve ambiguities by adopting the following conventions:

- Function application associates to the left, e.g. `e f g` is `(e f) g`, not `e (f g)`.
- Function application binds tighter than `if`, `fn`, and `rec`, e.g. `fn f => f 0` is `fn f => (f 0)`, not `(fn f => f) 0`.

Evaluation Rules

```
(1) n => n, for any non-negative integer literal n

(2) true => true          false => false

(3) error s => error s

(4) succ => succ          pred => pred          iszero => iszero

       b => true          e1 => v
(5)   ---------------------------
       if b then e1 else e2 => v

       b => false         e2 => v
(6)   ---------------------------
       if b then e1 else e2 => v

       e1 => succ         e2 => n
(7)   ---------------------------
             e1 e2 => n+1
```

```
       e1 => pred     e2 => 0         e1 => pred     e2 => n+1
(8)   ------------------------     --------------------------
           e1 e2 => 0                    e1 e2 => n

       e1 => iszero   e2 => 0         e1 => iszero   e2 => n+1
(9)   ------------------------     --------------------------
           e1 e2 => true                e1 e2 => false

(10)  (fn x => e) => (fn x => e)

         e1 => (fn x => e)      e2 => v1    e[x:=v1] => v
(11)  ------------------------------------------------------
                           e1 e2 => v

       e[x:=(rec x => e)] => v
(12)  -------------------------
         (rec x => e) => v
```

Typing

```
                    t ::= 'a | int | bool | t -> t | (t)
```

Here `'a` is a *type variable*, used for polymorphic types.

```
             E(x) = t
(ID)         ----------
             E |- x : t

(NUM)        E |- n : int

(BOOL)       E |- true : bool

             E |- false : bool

(BASE)       E |- succ : int -> int

             E |- pred : int -> int

             E |- iszero : int -> bool

             E |- e1 : bool  E |- e2 : t  E |- e3 : t
(IF)         ----------------------------------------
             E |- if e1 then e2 else e3 : t
```

```
            E[x : t1] |- e : t2
(-> INTRO)  -------------------------
            E |- fn x => e : t1 -> t2

            E |- e1 : t1 -> t2  E |- e2 : t1
(-> ELIM)   --------------------------------
            E |- e1 e2 : t2

            E[x : t] |- e : t
(REC)       ------------------
            E |- rec x => e : t
```

(a) (10 points) Implement a function `sum` in P, which adds two numbers `x` and `y` together.

`rec sum => fn x => fn y =>` (your code here)

(b) Infer the principal type of `fn f => fn x => f (f x)`

(c) Infer the principal type of `fn f => fn g => fn x => f (g x)`

(d) Infer the principal type of `fn b => if b then 1 else 0`

(e) Infer the principal type of `rec f => fn b => if b then 1 else f true`

(f) Infer the principal type of `rec f => fn x => f x`

(g) (10 points) Infer the principal type of

```
rec m => fn x => fn y => if iszero y then x
                         else m (pred x) (pred y)
```

(h) (10 points) Infer the principal type of

```
rec even => fn n => if iszero n then true
                    else if iszero (pred n) then false
                    else even (pred (pred n))
```

Solution.

(a) `rec sum => fn x => fn y => if iszero x then y else sum (pred x) (succ y)`

(b) `('a -> 'a) -> 'a -> 'a`

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

(d) `bool -> int`

(e) `bool -> int`

(f) `'a -> 'b`

(g) `int -> int -> int`

(h) `int -> bool`

□

Problem 2. (20 points) Now we want to extend the language P with `let` expressions.

```
e ::= ...   | let x = e in e end
```

For example, `let z = 2 in succ z end` is allowed. Rather than creating new rules for such expressions, we treat `let` expressions as *syntactic sugar*. In particular, we treat

```
let x = e1 in e2 end
```

as if it were

```
(fn x => e2) e1
```

(a) A bit of thought should convince you that this function application has exactly the same meaning as the `let` expression. However, this function application may not be typeable. Explain why with respect to the following program as an example.

```
let
  twice = fn f => fn x => f (f x)
in
  twice twice twice succ 0
end
```

(b) Write a short program in P without `let` that runs without errors but is not typeable with respect to the given rules in Problem 1.

Solution.

(a) Rewrite the example using shorter variable names for better readability:

$$\begin{aligned}&\texttt{let } t = \lambda f.\lambda x.f\ (f\ x)\\ &\ \texttt{in } (((t\ t)\ t)\ s)\ 0\\ &\texttt{end}\end{aligned}$$

This is translated to

$$(\lambda t.(((t\ t)\ t)\ s)\ 0)\ (\lambda f.\lambda x.f\ (f\ x))$$

The type of $\lambda f.\lambda x.f\ (f\ x)$ is $(\alpha \to \alpha) \to \alpha \to \alpha$, so in the subexpression $(t\ t)$ the variable t has type $(\alpha \to \alpha) \to \alpha \to \alpha$, and also type $\alpha \to \alpha$, which is a contradiction.

(b) `fn x => x x`

□

Problem 3. (25 points)

(a) Is there a type that is a subtype of every other type? Is there an arrow type that is a supertype of every other arrow type? Explain with examples.

(b) Write a short Java program involving arrays that type-checks but fails (by raising an `ArrayStoreException`) at run-time.

Solution.

(a) There is a type that is a subtype of every other type: Bot (as defined in the lecture).

If there is an arrow type $t_1 \to t_2$ that is a supertype of every other arrow type, then for any t_1' and t_2', $t_1' \to t_2' <= t_1 \to t_2$, so $t_1 <= t_1'$ and $t_2' <= t_2$. Now we have (1) for any t_1', $t_1 <= t_1'$ so $t_1 \equiv \text{Bot}$, and (2) for any t_2', $t_2' <= t_2$ so $t_2 \equiv \text{Top}$. So $\text{Bot} \to \text{Top}$ is the only possible arrow type that is a supertype of every other arrow type. However, this is no such a value that has the type $\text{Bot} \to \text{Top}$ since Bot has no values.

More on Bot: Suppose T is a function that maps each type to a set of values with that type. Then $T(\texttt{int}) = \{..., -2, -1, 0, 1, 2, ...\}$ and $T(\texttt{bool}) = \{\texttt{true}, \texttt{false}\}$. Also $T(t_1 \times t_2) = T(t_1) \times T(t_2)$ where $t_1 \times t_2$ represents a pair type and the later $\times$ represents Cartesian product, and $T(t_1 \to t_2) = [T(t_1) \to T(t_2)] \subseteq T(t_1) \times T(t_2)$. Since we have $T(\text{Bot}) = \emptyset$, it is obvious that $T(\text{Bot} \to t) \subseteq \emptyset \times T(t) = \emptyset$ where t is an arbitrary type.

(b)
```
$> cat > Test.java
public class Test {
  public static void main(String[] args) {
    Object[] x = new String[3];
    x[0] = new Integer(0);
  }
}
$> javac Test.java
$> java Test
Exception in thread "main" java.lang.ArrayStoreException: java.lang.Integer
  at Test.main(Test.java:4)
```

□