Homework 5
COSE212, Fall 2018

Hakjoo Oh

Due: 12/16, 24:00

Problem 1 (Simple Type System, 50pts) Consider the language:

```
type exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | READ
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

Types for the language are defined as follows:

```
type typ = TyInt | TyBool | TyFun of typ * typ | TyVar of tyvar
and tyvar = string
```

Implement the following type-inference function:

```
typeof : exp -> typ
```

which takes a program and returns its type if the program is well-typed. When the program is ill-typed, `typeof` should raise an exception `TypeError`.

Examples:

- The program

```
PROC ("f",
 PROC ("x", SUB (CALL (VAR "f", CONST 3),
                 CALL (VAR "f", VAR "x"))))
```

has type `TyFun (TyFun (TyInt, TyInt), TyFun (TyInt, TyInt))`.

- The program

  ```
  PROC ("f", CALL (VAR "f", CONST 11))
  ```

  has type `TyFun (TyFun (TyInt, TyVar "t"), TyVar "t")`, where `t` can be any type variable.

- The program

  ```
  LET ("x", CONST 1,
    IF (VAR "x", SUB (VAR "x", CONST 1), CONST 0))
  ```

  is ill-typed, so `typeof` should raise an exception `TypeError`.

As discussed in class, `typeof` is defined with two functions: one for generating type equations and the other for solving the equations. Complete the implementation of these two functions:

```
gen_equations  :  TEnv.t -> exp -> typ -> typ_eqn
        solve  :  typ_eqn -> Subst.t
```

Modules for type environments (`TEnv`) and substitutions (`Subst`), as well as the operations of applying substitutions to types (`Subst.apply`) and extending substitutions (`Subst.extend`), are provided.

Problem 2 (Let-Polymorphic Type System, 40pts) Consider the language:

```
type exp =
  | CONST of int
  | VAR of var
  | ADD of exp * exp
  | SUB of exp * exp
  | MUL of exp * exp
  | DIV of exp * exp
  | READ
  | ISZERO of exp
  | IF of exp * exp * exp
  | LET of var * exp * exp
  | LETREC of var * var * exp * exp
  | PROC of var * exp
  | CALL of exp * exp
and var = string
```

Define the function

```
expand : exp -> exp
```

that transforms an expression into a semantically-equivalent expression where every let-bound variable in the original expression gets replaced by its definition. Examples and caveat:

- Evaluating

  ```
  expand (LET ("x", CONST 1, VAR "x"))
  ```

  produces `CONST 1`.

- Evaluating

  ```
  expand (
    LET ("f", PROC ("x", VAR "x"),
     IF (CALL (VAR "f", ISZERO (CONST 0)),
      CALL (VAR "f", CONST 11),
      CALL (VAR "f", CONST 22))))
  ```

  produces

  ```
  IF (CALL (PROC ("x", VAR "x"), ISZERO (CONST 0)),
   CALL (PROC ("x", VAR "x"), CONST 11),
   CALL (PROC ("x", VAR "x"), CONST 22))
  ```

- Unused definitions should not go away. For example, evaluating

  ```
  expand (LET ("x", ADD (CONST 1, ISZERO (CONST 0)), CONST 2))
  ```

  should return `LET ("x", ADD (CONST 1, ISZERO (CONST 0)), CONST 2)`, not `CONST 2`.

As discussed in class, the function `expand` can be used for implementing the let-polymorphic type system. The type checker `typeof : exp -> typ` in Problem 1 does not support polymorphism and would not accept the program:

```
# typeof(
  LET ("f", PROC ("x", VAR "x"),
   IF (CALL (VAR "f", ISZERO (CONST 0)),
    CALL (VAR "f", CONST 11),
    CALL (VAR "f", CONST 22))));;

= Equations =
t2 = (t6 -> t7)
t7 = t6
(t5 -> bool) = t2
t5 = bool
```

```
int = int
(t4 -> t1) = t2
t4 = int
(t3 -> t1) = t2
t3 = int

The program does not have type. Rejected.
```

With `expand`, however, the same type checking algorithm will succeed:

```
# typeof(
    expand(
      LET ("f", PROC ("x", VAR "x"),
       IF (CALL (VAR "f", ISZERO (CONST 0)),
        CALL (VAR "f", CONST 11),
        CALL (VAR "f", CONST 22)))));;

= Equations =
(t8 -> bool) = (t9 -> t10)
t10 = t9
t8 = bool
int = int
(t5 -> t1) = (t6 -> t7)
t7 = t6
t5 = int
(t2 -> t1) = (t3 -> t4)
t4 = t3
t2 = int

= Substitution =
t3 |-> int
t4 |-> int
t2 |-> int
t6 |-> int
t7 |-> int
t1 |-> int
t5 |-> int
t9 |-> bool
t10 |-> bool
t8 |-> bool

Type of the given program: int
```

Problem 3 (10pts) Consider the language (called lambda calculus):

```
type lambda = V of var
            | P of var * lambda
            | C of lambda * lambda
and var = string
```

A program in lambda calculus is a variable, a procedure abstraction, or a call.

Write the function

```
check : lambda -> bool
```

that checks if a given program is well-formed. A program is said to be *well-formed* if and only if the program does not contain free variables; i.e., every variable name is bound by some procedure that encompasses the variable. For example, well-formed programs are:

- `P ("a", V "a")`
- `P ("a", P ("a", V "a"))`
- `P ("a", P ("b", C (V "a", V "b")))`
- `P ("a", C (V "a", P ("b", V "a")))`

Ill-formed ones are:

- `P ("a", V "b")`
- `P ("a", C (V "a", P ("b", V "c")))`
- `P ("a", P ("b", C (V "a", V "c")))`