HW1 - Type Inference for Explicit-Lang
Deadline: 16 March 2019

This assignment asks you to implement a type-inference algorithm for a language called Explicit-Lang. We next present the concrete syntax of this language, its abstract syntax, the typing rules (these will serve as guide to type-inference) and then the structure of the files you have to complete. Sample output is given at the end of this document.

1 Concrete Syntax of Explicit-Lang

```
<expr> ::=
    | ()
    | <integer>
    | <identifier>
    | <expr> + <expr>
    | <expr> - <expr>
    | <expr> * <expr>
    | <expr> / <expr>
    | let <identifier>=<expr> in <expr>
    | letrec <identifier>(<identifier>)=<expr> in <expr>
    | letrec <identifier>(<identifier>:<texpr>):<texpr>=<expr> in <expr>
    | proc(<identifier>:<texpr>) {<expr>}
    | proc (<identifier>) { <expr> }
    | (<expr> <expr>)
    | zero?(<expr>)
    | newref(<expr>)
    | deref(<expr>)
    | setref(<expr>,<expr>)
    | if <expr> then <expr> else <expr>
    | begin <expr>; ...; <expr> end
    | (<expr>)

<texpr>:
    | <identifier>
    | unit
    | int
    | bool
    | <texpr> -> <texpr>
    | ref <texpr>
    | (<texpr>)
```

If a term is decorated with type expressions, then the type inference function should verify that the annotated type is an instance of the one inferred.

2 Abstract syntax of Explicit-Lang

```
type expr =
  | Var of string
  | Int of int
  | Add of expr*expr
  | Sub of expr*expr
  | Mul of expr*expr
  | Div of expr*expr
  | Let of string*expr*expr
  | IsZero of expr
  | ITE of expr*expr*expr
  | Proc of string*texpr*expr
  | ProcUntyped of string*expr
  | App of expr*expr
  | Letrec of texpr*string*string*texpr*expr*expr
  | LetrecUntyped of string*string*expr*expr
  | BeginEnd of expr list
  | NewRef of expr
  | DeRef of expr
  | SetRef of expr*expr
and
  texpr =
  | IntType
  | BoolType
  | UnitType
  | VarType of string
  | FuncType of texpr*texpr
  | RefType of texpr

type prog = AProg of expr
```

3 Typing rules of Explicit-Lang

$$\frac{}{\texttt{tenv} \vdash \texttt{n} :: \texttt{int}}\ \textsf{TConst} \qquad \frac{\texttt{tenv(x)=t}}{\texttt{tenv} \vdash \texttt{x} :: \texttt{t}}\ \textsf{TVar} \qquad \frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{int}}{\texttt{tenv} \vdash \texttt{zero?(e)} :: \texttt{bool}}\ \textsf{TZero}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{int} \qquad \texttt{tenv} \vdash \texttt{e2} :: \texttt{int} \qquad \texttt{op} \in \{+,-,*,/\}}{\texttt{tenv} \vdash \texttt{e1 op e2} :: \texttt{int}}\ \textsf{TOp}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{bool} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{t} \quad \texttt{tenv} \vdash \texttt{e3} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{if e1 then e2 else e3} :: \texttt{t}}\ \textsf{TIf}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t1} \quad \texttt{[var=t1]tenv} \vdash \texttt{e2} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{let var=e1 in e2} :: \texttt{t2}}\ \textsf{TLet}$$

$$\frac{\texttt{tenv} \vdash \texttt{rator} :: \texttt{t1} \rightarrow \texttt{t2} \quad \texttt{tenv} \vdash \texttt{rand} :: \texttt{t1}}{\texttt{tenv} \vdash \texttt{(rator rand)} :: \texttt{t2}}\ \textsf{TApp}$$

$$\frac{\texttt{[var = t1]tenv} \vdash \texttt{e} :: \texttt{t2}}{\texttt{tenv} \vdash \texttt{proc (var:t1) \{e\}} :: \texttt{t1} \rightarrow \texttt{t2}}\ \textsf{TProc}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{newref(e)} :: \texttt{ref(t)}}\ \textsf{TNewRef}$$

$$\frac{\texttt{tenv} \vdash \texttt{e} :: \texttt{ref(t)}}{\texttt{tenv} \vdash \texttt{deref(e)} :: \texttt{t}}\ \textsf{TDeref}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{ref(t)} \quad \texttt{tenv} \vdash \texttt{e2} :: \texttt{t}}{\texttt{tenv} \vdash \texttt{setref(e1,e2)} :: \texttt{unit}}\ \textsf{TSetRef}$$

$$\frac{}{\texttt{tenv} \vdash \texttt{()} :: \texttt{unit}}\ \textsf{TUnit}$$

$$\frac{\texttt{tenv} \vdash \texttt{e1} :: \texttt{t1} \ldots \texttt{tenv} \vdash \texttt{en} :: \texttt{tn}}{\texttt{tenv} \vdash \texttt{begin e1;...;en end} :: \texttt{tn}}\ \textsf{TBeginEnd}$$

$$\frac{\begin{array}{l}\texttt{[var=tVar][f=tVar} \rightarrow tRes\texttt{]tenv} \vdash \texttt{e} :: tRes \\ \texttt{[f=tVar} \rightarrow tRes\texttt{]tenv} \vdash \texttt{body} :: \texttt{t}\end{array}}{\texttt{tenv} \vdash \texttt{letrec } tRes \texttt{ f (var:tVar) = e in body} :: \texttt{t}}\ \textsf{TRec}$$

4 Solution Structure

Modules:

- `ast.ml` AST

- `subs.ml` Substitutions of types for variables (type environments) and also types for type variables (mgu); variables are represented as strings. Interface file (`subs.mli`) is:

```
type subst = (string,Ast.texpr) Hashtbl.t

val create : unit -> subst

val extend : subst -> string -> Ast.texpr -> unit

val remove : subst -> string -> unit

val lookup : subst -> string -> Ast.texpr option

val apply_to_texpr : subst -> Ast.texpr -> Ast.texpr

val apply_to_expr : subst -> Ast.expr -> Ast.expr

val apply_to_env : subst -> subst -> unit

val string_of_subs : subst -> string

val domain : subst -> string list

val join : subst list -> subst
```

- `unification.ml` Interface file (`unification.mli`) is:

```
type  unif_result = UOk of Subs.subst | UError of Ast.texpr*Ast.texpr

val mgu : (Ast.texpr*Ast.texpr) list -> unif_result
```

- `infer.ml`. Implement:

```
type 'a error = OK of 'a | Error of string

type typing_judgement = subst*expr*texpr

val infer' : Ast.expr -> int -> (int * typing_judgement) error
```

  Note that `infer'` takes as input, not only the expression whose type we wish to infer, but also a number `n`. This number is used for generating fresh variables as follows: it is appended to the name of newly generated variables. For example, if `n` is 10, then

```
VarType ("_V"^string_of_int n)
```

  generates a type variable `VarType "_V10"`. In order not to reuse already used numbers, `infer'` must return the current value of `n` plus 1 so that it can be passed on to further recursive calls to `infer'`.

5 Sample Output

Type the following in utop

```
utop # for i=1 to 20 do
print_string @@ inf @@ Examples.expr i;
print_string "\n";
done;;
```

The output should be

```
 x:=_V0, |- x:  _V0
 empty |- 0:  int
 x:=int, |- App(x,1):  int
 x:=int, |- Zero?(x):  bool
 x:=_V1,f:=(_V1->_V2), |- App(f,x):  _V2
 f:=(int->_V1), |- App(f,0):  _V1
 f:=(int->int), |- App(App(f,0),1):  int
Error! Cannot unify int and (_V0->_V1)
 f:=(int->(int->_V2)), |- App(App(f,0),1):  _V2
Error! Cannot unify int and (int->_V1)
 x:=_V1,y:=_V3,f:=(_V1->(_V3->(_V5->_V6))),z:=_V5, |- App(App(App(f,x),y),z):  _V6
 x:=(_V2->(_V4->_V5)),y:=_V2,f:=(_V5->_V6),z:=_V4, |- App(f,App(App(x,y),z)):  _V6
Error! Cannot unify bool and (_V2->_V3)
 x:=int,f:=(bool->_V2), |- App(f,Zero?(x)):  _V2
Error! Cannot unify _V1 and (_V1->_V2)
 empty |- Proc(x:_V0,x):  (_V0->_V0)
 y:=_V2, |- App(Proc(x:_V2,x),y):  _V2
 empty |- Proc(s:(_V1->(_V3->_V4)),Proc(x:_V1,Proc(y:_V3,App(App(s,x),y)))):  ((_V1->(_V3->_V4))->(_V1->(_V3->_V4)))
 empty |- Proc(s:(_V3->_V4),Proc(x:(_V2->_V3),Proc(y:_V2,App(s,App(x,y))))):  ((_V3->_V4)->((_V2->_V3)->(_V2->_V4)))
 empty |- Let(x,7,Let(y,2,Let(y,Let(x,App(x,1),App(x,y)),App(App(x,8),y)))):  int
```

6 What to hand in

Hand in zip file with all your sources through Canvas.