Compilation : TP5

19 décembre 2014

Valentin Esmieu
Corentin Nicole
groupe 1.2

1 Code

Listing 1 – **typing.ml**

```
open Print_ast
open Ast




let rec typ_of_pattern : ml_pattern -> TypEnv.t * Ast.typ =
 function
 | Ml_pattern_var(s,ty) -> TypEnv.singleton s ty , ty
 | Ml_pattern_bool b -> TypEnv.empty , Tbool
 | Ml_pattern_int i  -> TypEnv.empty , Tint
 | Ml_pattern_pair(p1,p2) ->
   let env1,typ1 = typ_of_pattern p1 in
   let env2,typ2 = typ_of_pattern p2 in
   TypEnv.add_all env1 env2, TPair(typ1,typ2)
 | Ml_pattern_nil typ -> TypEnv.empty, typ
 | Ml_pattern_cons(x,l) ->
  let envx,typx = typ_of_pattern x in
  let envl,typl = typ_of_pattern l in
  if typl = TList typx then  TypEnv.add_all envx envl, TList typx
  else failwith "incompatibles types in the list"

let rec wt_expr (env:TypEnv.t) = function
 | Ml_int i -> Tint
 | Ml_bool b -> Tbool
 | Ml_nil ty   ->  ty
 | Ml_pair(e1,e2) ->
  let typ1 = wt_expr env e1 in
  let typ2 = wt_expr env e2 in
  TPair(typ1,typ2)
 | Ml_cons(e1,le1) ->
  let typee1 = wt_expr env e1 in
  let typele1 = wt_expr env le1 in
  if typele1 = TList typee1 then typele1
  else failwith "typ -> cons "
 | Ml_unop(op,e) ->
  let tyExp = wt_expr env e in
  (match op, tyExp with
   | Ml_fst,TPair(x,y) -> x
   | Ml_snd, TPair(x,y) -> y
   | _ -> failwith "typ -> unop")
 | Ml_binop(op,e1,e2) ->
   let tyE1 = wt_expr env e1 in
   let tyE2 = wt_expr env e2 in
  (match op,tyE1,tyE2 with
   | (Ml_add | Ml_sub | Ml_mult),Tint,Tint -> Tint
   | Ml_less, Tint, Tint -> Tbool
   | Ml_eq, x, y -> if x = y then Tbool
                    else failwith "not two same type for equal"
   | _ -> failwith "operation illegal")
```

```
| Ml_var x -> TypEnv.find x env
| Ml_if(e1,e2,e3) -> failwith "todo"
| Ml_fun l -> failwith "TODO"
| Ml_app(e1,e2) ->
  let typE1 = wt_expr env e1 in
  let typE2 = wt_expr env e2 in
  (match typE1 with
    TFun(typEnt,typRes) ->
      if(typE2 = typEnt)
      then typRes
      else failwith ("cannot apply")
  | _ -> failwith ("cannot apply, use a function !"))
| Ml_let (x,e1,e2) ->
  let t1 = wt_expr env e1 in
  wt_expr (TypEnv.update x t1 env) e2
| Ml_letrec(x,typ,e1,e2) ->
  let newEnv = TypEnv.update x typ env in
  let newTyp = wt_expr newEnv e1 in
  if newTyp = typ then wt_expr newEnv e2
  else failwith "incompatible types in let rec"

let wt_ast tenv ast =
  match ast with
  | Ml_expr e -> wt_expr (!tenv) e
  | Ml_definition(s,e) ->
   let ty' = wt_expr !tenv e in
   tenv := TypEnv.update s ty' !tenv ;
   ty'
  | Ml_definitionrec (s,ty',e) ->
   let ty = wt_expr (TypEnv.update s ty' !tenv) e in
   if ty = ty'
   then
    begin
     tenv := TypEnv.update s ty !tenv ;
     ty'
    end
   else failwith (Printf.sprintf "Type error: let rec with incompatible types
      %s and %s" (string_of_typ ty) (string_of_typ ty'))
```

Listing 2 – **parser.mly**

```
%{
  open Ast
%}

%token <int> INT
%token <string> IDENT
%token TRUE FALSE
%token LET REC IN
%token FUNCTION ARROW ALTERNATIVE
%token IF THEN ELSE
%token ADD SUB MULT LESS
%token FST SND
%token EQUAL
%token LEFT_PAREN RIGHT_PAREN
```

```
%token LEFT_BRACKET RIGHT_BRACKET CONS
%token COMMA
%token END_OF_EXPRESSION
%token EOF
%token COLON
%token TBOOL TINT TLIST

%nonassoc NO_ALTERNATIVE
%nonassoc ALTERNATIVE IN
%left LESS EQUAL
%right CONS
%left ADD SUB
%left MULT
%left ARROW
%nonassoc FST SND ELSE

%start main
%type <Ast.ml_ast * StrSet.t > main
%%

main:
 | EOF { Printf.printf "\nbye"; exit 0 }
 | LET IDENT EQUAL expr END_OF_EXPRESSION { Ml_definition($2, fst $4), snd $4
    }
 | LET REC IDENT EQUAL expr END_OF_EXPRESSION { Ml_definitionrec($3, failwith
    "let rec type expected", fst $5) , snd $5 }
 | expr END_OF_EXPRESSION { Ml_expr (fst $1) , (*TODO*) StrSet.empty}
 | error {
    let bol = (Parsing.symbol_start_pos ()).Lexing.pos_bol in
    failwith ("parsing: line " ^
                (string_of_int ((Parsing.symbol_start_pos ()).Lexing.pos_lnum
                   )) ^
                " between character " ^
                (string_of_int (Parsing.symbol_start () - bol)) ^
                " and " ^
                (string_of_int ((Parsing.symbol_end ()) + 1 - bol)))
 }

expr:
 | simple_expr { $1 }
 | LEFT_PAREN expr RIGHT_PAREN { $2 }
 | expr CONS expr { Ml_cons(fst $1, fst $3) ,  StrSet.union (snd $1) (snd $3)
    }
 | LEFT_PAREN expr COMMA expr RIGHT_PAREN { Ml_pair(fst $2, fst $4) ,  StrSet.
    union (snd $2) (snd $4) }
 | FST expr { Ml_unop(Ml_fst, fst $2) , snd $2 }
 | SND expr { Ml_unop(Ml_snd, fst $2) , snd $2 }
 | expr ADD expr { Ml_binop(Ml_add, fst $1, fst $3) ,  StrSet.union (snd $1) (
    snd $3) }
 | expr MULT expr { Ml_binop(Ml_mult, fst $1, fst $3) ,  StrSet.union (snd $1)
     (snd $3) }
 | expr SUB expr { Ml_binop(Ml_sub, fst $1, fst $3),  StrSet.union (snd $1) (
    snd $3)}
 | expr LESS expr { Ml_binop(Ml_less, fst $1, fst $3), StrSet.union (snd $1) (
```

```
    snd $3) }
 | expr EQUAL expr { Ml_binop(Ml_eq, fst $1, fst $3), StrSet.union (snd $1) (
    snd $3) }
 | IF expr THEN expr ELSE expr { Ml_if(fst $2, fst $4, fst $6), StrSet.union (
    snd $2) (StrSet.union (snd $4) (snd $6))}
 | FUNCTION pattern_expr_list { Ml_fun (fst $2) , snd $2 }
 | application { List.fold_left (fun res a -> Ml_app(res, a)) (List.hd (fst $1
    )) (List.tl (fst $1)) , snd $1   }
 | LET IDENT  EQUAL expr IN expr { Ml_let($2, fst $4, fst $6) , (StrSet.union
    (snd $4) (StrSet.remove $2 (snd $6))) }
 | LET REC IDENT COLON typ EQUAL expr IN expr { Ml_letrec($3,$5, fst $7, fst
    $9) , (StrSet.union (StrSet.remove $3 (snd $7)) (StrSet.remove $3 (snd $9
    ))) }

simple_expr:
 | INT { Ml_int $1, StrSet.empty }
 | bool { Ml_bool $1, StrSet.empty }
 | LEFT_BRACKET RIGHT_BRACKET COLON typ  { Ml_nil $4 , StrSet.empty }
 | IDENT { Ml_var $1 , StrSet.singleton $1}

bool:
 | FALSE { false }
 | TRUE  { true }

pattern:
 | IDENT COLON typ       { Ml_pattern_var ($1,$3) , StrSet.singleton $1}
 | INT   { Ml_pattern_int $1 , StrSet.empty }
 | bool  { Ml_pattern_bool $1 , StrSet.empty }
 | LEFT_PAREN pattern COMMA pattern RIGHT_PAREN
   {Ml_pattern_pair(fst $2, fst $4), StrSet.union (snd $2) (snd $4)  }
 | LEFT_BRACKET RIGHT_BRACKET COLON typ { Ml_pattern_nil $4 , StrSet.empty }
 | pattern CONS pattern { Ml_pattern_cons(fst $1, fst $3), StrSet.union (snd
    $1) (snd $3) }

pattern_expr_list:
 | pattern ARROW expr pattern_expr_list_next { (fst $1, fst $3) :: fst $4 ,
    StrSet.union (snd $4) (StrSet.diff (snd $3) (snd $1))  }

pattern_expr_list_next:
 | ALTERNATIVE pattern ARROW expr pattern_expr_list_next { (fst $2, fst $4) ::
     fst $5, StrSet.union (snd $5) (StrSet.diff (snd $4) (snd $2))  }
 | %prec NO_ALTERNATIVE { [] ,  StrSet.empty }

application:
 | simple_expr_or_parenthesized_expr simple_expr_or_parenthesized_expr
    application_next { fst $1 :: fst $2 :: fst $3, StrSet.union (snd $1) (
    StrSet.union (snd $2) (snd $3))  }

simple_expr_or_parenthesized_expr:
 | simple_expr { $1 }
 | LEFT_PAREN expr RIGHT_PAREN { $2 }

application_next:
 | simple_expr_or_parenthesized_expr application_next { fst $1 :: fst $2 ,
```

```
    StrSet.union (snd $1) (snd $2) }
 | { [] , StrSet.empty }


typ :
| TBOOL { Tbool }
| TINT { Tint }
| typ ARROW typ { TFun($1,$3) }
| typ TLIST { TList $1 }
| typ MULT typ {TPair($1,$3)}
| LEFT_PAREN typ RIGHT_PAREN {$2 }
```

2 Tests

Listing 3 – **test.ml**

```
# 1+1;;
- = 2:int
# let x = 2;;
# x;;
- = 2:int
# x+3;;
- = 5:int
# x < 5;;
- = true:bool
# false;;
- = false:bool
# let x = 3 in x + 2;;
- = 5:int
# let x = 3 in x + 2;;
- = 5:int
# fst(5,false);;
- = 5:int
# 5+false;;
 error: operation illegal
```