プログラミング言語処理系 インタプリタ演習 & 型推論演習
パート試験

工学部 情報学科 計算機科学コース
学籍番号: 1029358455 氏名: 登古紘平

today

今回のインタプリタ演習は、ソフトウェア実験の課題 3 で作成したソースコードを基に行っている。

1 文字列型の実装

文字列定数 ‘”hogehoge”‘，文字列の連結 ‘s1 ^ $s2$‘，文字列の n 文字目を返す操作 ‘$s.[n]$‘，文字列の出力関数 ‘*print_strings*‘をサポートできればよい.

1.1 プログラムの設計方針

字句・構文解析器と抽象構文木を拡張し、文字列操作の構文を定義した。次に TyString 型と型付け規則を導入して静的な型安全性を保証し、最後に評価器では OCaml の組み込み関数に処理を委ねることで効率的な実行を実現した。

1.2 実装の説明

```
lexer.ml------

let reservedWords = [
  ・・・
  ("print_string", Parser.PRINT_STRING);
]

|"^" { Parser.CONCAT } (* s1^s2 *)
|".[" { Parser.DOT_LBRACKET } (* s[i] *)
|'"' [^'"']*'"'
    { let s = Lexing.lexeme lexbuf in
      let len = String.length s in
      Parser.STRINGV (String.sub s 1 (len - 2)) }

syntax.ml------

type exp =
  ・・・
  | SLit of string
  | StrConcatExp of exp * exp
  | StrGetExp of exp * exp
  | PrintStrExp of exp

type ty =
  ・・・
  | TyString （* 文字列型を表すコンストラクタ *）
```

```
let rec string_of_ty = function
  ・・・
  | TyString -> "string"  (* 文字列型を "string" として返す *)

let rec freevar_ty ty =
  ・・・
  | TyString -> MySet.empty  (* 文字列型には型変数が含まれないので空集合を返す *)

let rec pp_ty typ =
  match typ with
   ・・・
  | TyString -> print_string "string"  (* 文字列型を "string" として表示 *)

perser.mly------

%token <string> STRINGV
%token CONCAT
%token PRINT_STRING
%token DOT_LBRACKET

PExpr :
    l=PExpr PLUS r=CONCATExpr { BinOp (Plus, l, r) }
  | e=CONCATExpr { e }

CONCATExpr :
    l=CONCATExpr CONCAT r=MExpr { StrConcatExp (l, r) } (* 文字列連結 *)
  | e=MExpr { e } (* 文字列連結がない場合はそのまま *)

AppExpr :
  ・・・
  | e1 = AExpr DOT_LBRACKET e2=AExpr RBRACKET { StrGetExp (e1, e2) }  (* 文字列のインデックス取得 *)
  | PRINT_STRING e=AExpr { PrintStrExp e }  (* 文字列出力 *)

AExpr :
   ・・・
  | s=STRINGV { SLit s }

eval.ml------
```

```
type exval =
   ・・・
  | StringV of string (* 文字列値 *)

let rec string_of_exval = function
    ・・・
  | StringV s  -> "\"" ^ s ^ "\"" (* 文字列値の文字列表現 *)

let rec eval_exp env = function
    ・・・
  | SLit s -> StringV s  (* 文字列リテラルの評価 *)
  | StrConcatExp (exp1, exp2) -> (* 文字列連結の評価 *)
      let str1 = eval_exp env exp1 in
      let str2 = eval_exp env exp2 in
      (match str1, str2 with
         StringV s1, StringV s2 -> StringV (s1 ^ s2) (* 文字列を連結 *)
       | _, _ -> err "Both arguments must be strings: ^")
  | StrGetExp (exp1, exp2) -> (* 文字列のインデックス取得 *)
      let str = eval_exp env exp1 in
      let index = eval_exp env exp2 in
      (match str, index with
         StringV s, IntV i ->
          if i >= 0 && i < String.length s then
            StringV (String.sub s i 1) (* インデックスが有効な場合、文字を取得 *)
          else
            err ("Index out of bounds: " ^ string_of_int i)
       | _, _ -> err "First argument must be a string and second must be an integer: .[i]")
  | PrintStrExp exp -> (* 文字列を出力する *)
      let str_val = eval_exp env exp in
      (* 文字列値であることを確認して出力 *)
      (match str_val with
         StringV str_val ->
          print_string str_val;
          flush_all ();
          StringV str_val
        | _ -> err "Argument must be a string: print_string")

typing.ml------

let rec freevars ty =
```

```
  match ty with
  ・・・
  | TyString -> [] (* 文字列型 TyString には型変数は含まれないので空リスト [] を返す *)

let rec ty_exp tyenv exp =
  match exp with
  ・・・
  | SLit _ -> ([], TyString)
  | StrConcatExp (exp1, exp2) -> (* 文字列連結の型推論 *)
      let (s1, ty1) = ty_exp tyenv exp1 in (*  exp1 の型推論 *)
      let (s2, ty2) = ty_exp tyenv exp2 in (*  exp2 の型推論 *)
      let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ (* s1, s2 の型代入を制約集合に変
換し，両方が文字列型である制約を追加 *)
                [(ty1, TyString); (ty2, TyString)] in
      let s3 = unify eqs in (* 全制約を単一化し，型代入 s3 を得る *)
      (s3, TyString) (* 型代入 s3 と、結果の型 TyString を返す *)
  | StrGetExp (exp1, exp2) -> (* 文字列のインデックス取得の型推論 *)
      let (s1, ty1) = ty_exp tyenv exp1 in (* exp1 の型推論 *)
      let (s2, ty2) = ty_exp tyenv exp2 in (* exp2 の型推論 *)
      let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) @ (* s1, s2 の型代入を制約集合に変
換し，exp1 が文字列型、exp2 が整数型である制約を追加 *)
                [(ty1, TyString); (ty2, TyInt)] in
      let s3 = unify eqs in (* 全制約を単一化し，型代入 s3 を得る *)
      (s3, TyString)
  | PrintStrExp exp -> (* 文字列の出力の型推論 *)
      let (s, ty) = ty_exp tyenv exp in
      let eqs = (eqs_of_subst s) @ [(ty, TyString)] in
      let s2 = unify eqs in
      (* print 式は NilV を返すので、その型である 'a list を表現 *)
      (s2, TyList (TyVar (fresh_tyvar ())))
```

1.3 実装について工夫した点

モジュール化された既存の枠組みを拡張する形で実装を行い、簡潔に記述した。

1.4 プログラムのテスト

```
# "hogehoge";;
val - : string = "hogehoge"
# "hoge" ^ "hoge";;
```

```
val - : string = "hogehoge"
# "hogehoge".[2];;
val - : string = "g"
# print_string "hogehoge";;
hogehogeval - : ('e list) = "hogehoge"
```

これらのテストコードにより、所望の動作を得られていることが分かる。

2 文字列型の実装

ペア型の値とそれに対する操作を実装しなさい．型推論までサポートすること．ペアを作る操作 ‘(e1,e2)‘, ペアの第一要素をとってくる操作 ‘proj1 e‘ と第二要素をとってくる操作 ‘proj2 e‘, をサポートできればよい.

2.1 プログラムの設計方針

任意の二つの値を組み合わせるため、構文・型・実行時値の定義にペア用のコンストラクタ (PairExp, TyPair, PairV) を導入した。評価器はパターンマッチで安全な要素の取り出しを保証し、型推論器は unify 関数自体を拡張することで、int * string のような異なる型同士のペアも正しく扱える一般的な型付けルールを実現した。

2.2 実装の説明

```
lexer.ml------

let reservedWords = [
  ・・・
  ("proj1", Parser.PROJ1);
  ("proj2", Parser.PROJ2);
]

| "," { Parser.COMMA }

syntax.ml------

type exp =
  ・・・
  | PairExp of exp * exp
  | Proj1Exp of exp
  | Proj2Exp of exp
```

```
type ty =
  ・・・
  | TyPair of ty * ty (* ペア型を表すコンストラクタ *)

let rec string_of_ty = function
  ・・・
  | TyPair (t1, t2) -> (* ペア型 t1 * t2 を括弧で囲んで表現し、両方の型を再帰的に文字列化 *)
     "(" ^ string_of_ty t1 ^ " * " ^ string_of_ty t2 ^ ")"

let rec freevar_ty ty =
  ・・・
  | TyPair (t1, t2) ->  MySet.union (freevar_ty t1) (freevar_ty t2) (* ペア型では両方の
要素型から型変数を収集し、和集合を計算 *)

let rec pp_ty typ =
  match typ with
    ・・・
    | TyPair (t1, t2) -> (* ペア型 t1 * t2 を (t1 * t2) として表示 *)
     print_string "(";
     pp_ty t1;
     print_string " * ";
     pp_ty t2;
     print_string ")"

perser.mly------

%token COMMA PROJ1 PROJ2

AppExpr :
  ・・・
  | PROJ1 e=AExpr { Proj1Exp e } (* ペアの第 1 要素を取得 *)
  | PROJ2 e=AExpr { Proj2Exp e } (* ペアの第 2 要素を取得 *)

AExpr :
    ・・・
  | LPAREN e1=Expr COMMA e2=Expr RPAREN { PairExp (e1, e2) } (* ペア式の追加 *)

eval.ml------
type exval =
  ・・・
```

```
  | PairV of exval * exval (* ペア値 *)

let rec string_of_exval = function
    ・・・
  | PairV (v1, v2) ->  (* ペア値の文字列表現 *)
     "(" ^ string_of_exval v1 ^ ", " ^ string_of_exval v2 ^ ")"

let rec eval_exp env = function
    ・・・
  | PairExp (exp1, exp2) -> (* ペアの評価 *)
    (* exp1 と exp2 の評価を行い，それぞれの値を取得 *)
    let v1 = eval_exp env exp1 in
    let v2 = eval_exp env exp2 in
    PairV (v1, v2)
  | Proj1Exp exp -> (* ペアの第 1 要素の取得 *)
    (* exp の評価を行い，ペア値を取得 *)
    (match eval_exp env exp with PairV (v, _) -> v | _ -> err "Argument of proj1 must be a pair")
  | Proj2Exp exp -> (* ペアの第 2 要素の取得 *)
    (* exp の評価を行い，ペア値を取得 *)
    (match eval_exp env exp with PairV (_, v) -> v | _ -> err "Argument of proj2 must be a pair")

typing.ml------

let rec subst_type (subst : (tyvar * ty) list) (ty : ty) : ty =
  ・・・
    | TyPair (ty1, ty2) ->
        (* ペア型の場合、各要素型 ty1 と ty2 に再帰的に置換を適用 *)
        TyPair (apply_subset (var, ty_subset) ty1, apply_subset (var, ty_subset) ty2)

let rec freevars ty =
  match ty with
  ・・・
  | TyPair (t1, t2) -> freevars t1 @ freevars t2 (* ペア型 t1 * t2 の場合、それぞれの要素
型の自由変数を再帰的に求めて結合する *)

let rec unify eqs = (* 4.3.3 制約リスト eqs が空かどうか、またはペアの形を見てパターンマッチ
する *)
  match eqs with
  ・・・
  | (TyPair (a1, a2), TyPair (b1, b2)) :: rest -> (* ペア型のペアの場合 *)
```

```
      unify ((a1, b1) :: (a2, b2) :: rest) (* ペア型の各要素の型をペアにして残りの制約リストに追加 *)
  | (TyString, TyString) :: rest ->  (* 文字列型のペアの場合 *)
      unify rest  (* 文字列型は同じ型なので制約リストから削除して残りを処理 *)

let rec ty_exp tyenv exp =
  match exp with
  ・・・
  | PairExp (exp1, exp2) -> (* ペアの型推論 *)
      (* exp1 と exp2 の型推論を行い，それぞれの型を取得 *)
      let (s1, ty1) = ty_exp tyenv exp1 in
      let (s2, ty2) = ty_exp tyenv exp2 in
      let eqs = (eqs_of_subst s1) @ (eqs_of_subst s2) in
      let s3 = unify eqs in
      (s3, TyPair (subst_type s3 ty1, subst_type s3 ty2))
  | Proj1Exp exp -> (* ペアの第 1 要素の型推論 *)
      (* exp の型推論を行い，その型を取得 *)
      let (s1, ty1) = ty_exp tyenv exp in
      let ty_a = TyVar (fresh_tyvar ()) in
      let ty_b = TyVar (fresh_tyvar ()) in
      let eqs = (eqs_of_subst s1) @ [(ty1, TyPair (ty_a, ty_b))] in
      let s2 = unify eqs in
      (s2, subst_type s2 ty_a)
  | Proj2Exp exp -> (* ペアの第 2 要素の型推論 *)
      (* exp の型推論を行い，その型を取得 *)
      let (s1, ty1) = ty_exp tyenv exp in
      let ty_a = TyVar (fresh_tyvar ()) in
      let ty_b = TyVar (fresh_tyvar ()) in
      let eqs = (eqs_of_subst s1) @ [(ty1, TyPair (ty_a, ty_b))] in
      let s2 = unify eqs in
      (s2, subst_type s2 ty_b)
```

2.3 実装について工夫した点

proj1 と proj2 の型推論において、TyVar (fresh_tyvar ()) を使って新しい型変数を導入することで、中身の型が何であれ対応できる、非常に一般的な（多相的な）ルールを実装した。

2.4 プログラムのテスト

```
# let s = ("hoge",3);;
```

```
val s : (string * int) = ("hoge", 3)
# proj1 s;;
val - : string = "hoge"
# proj2 s;;
val - : int = 3
```

これらのテストコードにより、所望の動作を得られていることが分かる。