ソフトウェアサイエンス実験 S8 最終レポート

200911434 青木大祐

2012 年 11 月 11 日

1 概要

本実験では、関数プログラミング言語 Ocaml を用いて、Ocaml 自身のサブセットである「ミニ Ocaml 言語 (以下「ミニ Ocaml」)」のインタプリタおよびコンパイラを作成した。

2 Ocaml について

まず最初に、実験に用いる Ocaml について理解を深めるため、関数プログラミングの技法を学んだ。具体的には、再帰やパターンマッチを用いて Euclid の互除法から最大公約数を求める関数や、Fibonacci 数を求める関数を記述した。例を以下に示す。

```
let rec gcd (x, y) =
  if x <= 0 || y <= 0 then gcd (abs x, abs y)
  else if x = y then x
  else if x > y then gcd(x-y, y)
  else gcd(x, y-x);;

let rec fib n =
  match n with
      1 -> 1
    | 2 -> 1
    | n -> fib(n - 2) + fib(n - 1);;
```

3 ミニ Ocaml 言語

前章の内容を終えた後、本題である「ミニ Ocaml インタプリタの実装」に着手した。本実験で扱うミニ Ocaml の構文は以下のとおり[*1]である。

```
e ::= x                   変数
  | let x = e in e        let式
  | let rec f x = e in e  let-rec式
  | fun x -> e            関数
  | e e                   関数適用
  | true                  真理値リテラル(定数)
  | false                 真理値リテラル(定数)
  | if e then e else e    if式
  | n                     自然数リテラル (n は自然数 0, 1, 2, ...)
  | - e                   整数演算(符号の反転)
  | e + e                 整数演算(足し算)
  | e * e                 整数演算(かけ算)
  | e / e                 整数演算(割り算)
  | e = e                 等しさ(整数と真理値)
  | e > e                 大小比較(整数)
  | e < e                 大小比較(整数)
```

このミニ Ocaml を実装するにあたって、入力されるプログラムを扱う内部的な表現として式 (*exp*) という構造を考える。プログラムは式からなり、また式は一つないしは複数の部分的な式から成り立っている。この再帰的な構造を表現するため、Ocaml のヴァリアント型を用いて代数的データ構造を実現することにする。

上記のミニ Ocaml の仕様を満たすために、次のような式の種類を *Syntax.ml* として用いる。なお、これは講義ページで配布されたものをそのまま利用するため、上記の仕様を含む、より多種の式の定義となっている。

```
type exp =
  | Var of string         (* variable e.g. x *)
  | IntLit of int         (* integer literal e.g. 17 *)
  | BoolLit of bool       (* boolean literal e.g. true *)
  | If of exp * exp * exp (* if e then e else e *)
  | Let of string * exp * exp   (* let x=e in e *)
  | LetRec of string * string * exp * exp   (* letrec f x=e in e *)
  | Fun of string * exp   (* fun x -> e *)
  | App of exp * exp      (* function application i.e. e e *)
  | Eq of exp * exp       (* e = e *)
  | Noteq of exp * exp (* e <> e *)
  | Greater of exp * exp  (* e > e *)
  | Less of exp * exp     (* e < e *)
  | Plus of exp * exp     (* e + e *)
  | Minus of exp * exp    (* e - e *)
  | Times of exp * exp    (* e * e *)
```

[*1] 実験テキスト (http://logic.cs.tsukuba.ac.jp/ kam/jikken/mini.html) より引用

```
  | Div of exp * exp      (* e / e *)
  | Empty                 (* [ ] *)
  | Match of exp * ((exp * exp) list)    (* match e with e->e | ... *)
  | Cons of exp * exp     (* e :: e *)
  | Head of exp           (* List.hd e *)
  | Tail of exp           (* List.tl e *)
```

これらを踏まえて、与えられた式を評価して、最終的な値を計算するインタプリタを実装する。

3.1 eval 関数

式を受け取り、式の種類によって違う処理を行うため、Ocaml のパターンマッチングを用いて *eval* 関数を実装する。整数値の足し算と掛け算を評価する *eval*1 を示す。

```
let rec eval1 e =
  match e with
  | IntLit(n)    -> n
  | Plus(e1,e2)  -> (eval1 e1) + (eval1 e2)
  | Times(e1,e2) -> (eval1 e1) * (eval1 e2)
  | _ -> failwith "unknown expression"
```

もし与えられた式が整数リテラル (*IntLit*) だった場合は、その数値が式を評価した値となる。また、与えられた式が足し算 (*Plus*) だった場合は、2 つの引数として与えられた式を再帰的に評価し、その 2 値を合計し、評価した値として返す。掛け算についても同様である。
もし exp 型の値であるが $IntLit, Plus, Times$ でないものが渡された場合、最後のパターン (_) にマッチし、*unknown expression* という例外が投げられる。

この *eval*1 に、入力として 2∗2+(3+(−4)) という式を与えてみる。*exp* 型の表現では $Plus(Times(IntLit\,2, IntLit\,2), Plus(IntLit\,3, IntLit\,(-4)))$ である。実行結果を以下に示す。

```
val easy : exp =
  Plus (Times (IntLit 2, IntLit 2), Plus (IntLit 3, IntLit (-4)))
- : int = 3
```

正しく計算できていることが分かる。

3.2 値と型

*eval*1 では評価した値は Ocaml の *integer* 型であるが、ミニ Ocaml のデータ型としてはまず *Int* 型と *Bool* 型を扱いたいため、このままでは不十分である。複数のデータ型を扱えるようにするため、値 (*value*) というデータ構造を導入する。

```
type value =
  | IntVal  of int        (* integer value e.g. 17 *)
  | BoolVal of bool       (* booleanvalue e.g. true *)
```

式を評価すると値が返る。整数型が求められる式上の位置に真偽値型がある場合は型エラーであり、またその逆も同じくエラーである。例えば、条件節の真偽によって実行する処理を分岐する If 文では、条件節は真偽値型でなければならない。
以下に、*eval*1 の式に加えて真偽値によって分岐する If、式 1 と式 2 が同じ値なら真を返す Eq、式 1 が式 2 よりも大きい整数であれば真を返す *Greater* を実装した *eval*2 を以下に示す。

```
let rec eval2 e =
  match e with
  | IntLit(n)  -> IntVal(n)
  | Plus(e1,e2) ->
      begin
        match (eval2 e1, eval2 e2) with
    | (IntVal(n1),IntVal(n2)) -> IntVal(n1+n2)
          | _ -> failwith "integer values expected"
      end
  | Times(e1,e2) ->
      begin
        match (eval2 e1, eval2 e2) with
    | (IntVal(n1),IntVal(n2)) -> IntVal(n1*n2)
          | _ -> failwith "integer values expected"
      end
  | Eq(e1,e2) ->
      begin
  match (eval2 e1, eval2 e2) with
    | (IntVal(n1),IntVal(n2)) -> BoolVal(n1=n2)
```

```
    | (BoolVal(b1),BoolVal(b2)) -> BoolVal(b1=b2)
    | _ -> failwith "wrong value"
      end
  | BoolLit(b) -> BoolVal(b)
  | If(e1,e2,e3) ->
      begin
  match (eval2 e1) with
    | BoolVal(true) -> eval2 e2
    | BoolVal(false) -> eval2 e3
    | _ -> failwith "wrong value"
      end
  | Greater (e1, e2) ->
      begin
        match (eval2 e1, eval2 e2) with
          | (IntVal(n1), IntVal(n2)) -> BoolVal(n1 > n2)
          | _ -> failwith "wrong value"
      end
  | _ -> failwith "unknown expression e";;
```