ML 演習 第 7 回

新井淳也、中村宇佑、前田俊行

2011/05/31

言語処理系の作成

- 第 4 回～第 7 回の予定
  - 第 4 回: 基本的なインタプリタの作成
    - 字句解析・構文解析・簡単な言語の処理系の作成
  - 第 5 回: 関数型言語への拡張
    - 関数を作成し呼び出せるように
  - 第 6 回: 言語処理系と型システム
    - ML 風の型推論の実装
  - 第 7 回: インタプリタの様々な拡張
    - 式の評価順序に関する考察

今ここ

今回の内容

- 式の評価戦略 (evaluation strategy)
  - さまざまな評価戦略
    - Call by value
    - Call by name
    - Call by need

式の評価戦略とは

- 式の部分式を評価する順番・手法のこと

```
(e1, e2) ;;
fst (e1, e2) ;;
let x = e1 in e2;;
e1 e2 e3 e4 e5 e6;;
```

どの式 (e_i) を
どのタイミングで
評価するか

評価戦略に関する関数型言語の性質

- 基本的には式の評価戦略は
  評価結果に影響しない

  - 例外 1: 副作用のある式

```
fst (5, print_int 3)
```

  - 例外 2: 止まらない評価

```
let rec loop x = loop x in
  fst (5, loop 1)
```

代表的な三つの評価戦略

- Call by value
- Call by name
- Call by need

} 遅延評価 (lazy evaluation)

Call by value

- 関数などの引数を適用の前に評価

```
let double n = n + n in
double (2 * 3)
→ double 6
→ 6 + 6
→ 12
```

Call by value の利点

- 効率のよい実装が可能
  - 関数に渡されるのは評価後の値のみ
- 評価順がわかりやすい
  - 副作用が扱いやすい

Call by value の欠点

- 他の評価戦略では評価が止まる式でも評価が止まらないことがある

```
let rec loop x = loop x in
fst (5, loop 1)
```
→ 発散

  - `if` や `||` などを関数としては表現できない
    - `if` や `||` などに対しては通常の関数とは異なる特別なプリミティブを用意しないといけない

Call by name

- 関数適用を引数より先に評価

```
let double n = n + n in
double (2 * 3)
→ (2 * 3) + (2 * 3)
→ 6 + (2 * 3)
→ 6 + 6
→ 12
```

Call by name の利点

- 不要な評価を避けることができる
  - `let rec loop x = loop x in`
    `fst (5 + 3, loop 1)`
    → `5 + 3`
    → `8`
  - `if` を関数として実装できる
  - 他の評価戦略で評価が止まる式は call by name でも必ず評価が止まる

Call by name の欠点

- 式の評価の効率が悪くなる
  - 常に「式」の形で評価を進める必要がある
    - 関数に直接式を渡さなければならない
  - 同じ式を何回も評価する
- 式の評価回数やタイミングが制御困難
  - 副作用がある言語では使いづらい

Call by need

- 関数適用を引数より先に評価
  - ただし式は一度だけ評価し、結果を使い回す

```
let double n = n + n in
double (2 * 3)
→ (2 * 3) + (2 * 3)
→ 6 + 6
→ 12
```

(2 * 3) は
元々一つの式なので
一度しか評価されない

  - Haskell が採用している評価戦略

Call by need の利点

- 同じ式を何度も評価しない点では効率が良い
- 不要な評価を避けられる
  - Call by name と同じ
  - 他の評価戦略で評価が止まる式は
    call by need でも必ず評価が止まる

Call by need の欠点

- 式の評価の実装が依然複雑
  - 常に「式」の形で評価を進める必要がある
    - 関数に直接式を渡さなければならない
- 式の評価のタイミングが依然制御困難
  - Call by name より更に複雑

Call by value で call by name の真似をするには

- λ抽象を評価順の制御に使えばよい

  （例）if を関数として定義する

```
let if_cbn c t f = if c () then t ()
                             else f ()

if_cbn (fun () -> true) (fun () -> 1)
                        (fun () -> 2)
→ if (fun …) () then (fun …) ()
                  else (fun …) ()
→ if true then (fun …) () else (fun …) ()
→ (fun () -> 1) ()
→ 1
```

評価のタイミングを明示

λ抽象式で評価を遅延

OCaml で遅延評価を用いる方法については以下を参照:
http://caml.inria.fr/pub/docs/manual-ocaml/manual021.html#toc73
http://caml.inria.fr/pub/docs/manual-ocaml/libref/Lazy.html

第 7 回 課題

締切: 6/14 13:00 (日本標準時)

課題 1 (15点)

- 第 5 回または第 6 回のインタプリタを改造して call by name で評価を行うようにせよ
  - タプル・リストには (まだ) 対応しなくてよい

課題 1 (続き)

- 実装法の一例:
  関数適用 (呼出) で引数を評価しないようにする
  - 「環境」を改造する
    - 「変数」から「値」ではなく
      「変数」から thunk への写像とする
      - Thunk = 「(遅延評価される)式」と評価される「環境」の組
    - 「環境」から「変数」を取りだすときに
      thunk の中身を評価する

課題 1 (続き)

- 例: 環境 E における式「e_1 e_2」の評価
  - e_1 を評価して値を得る
    - もし値がクロージャでなかったらエラー
  - 引数名と thunk (e_2, E) との対応を
    クロージャに含まれる環境に追加して拡張する
    - e_2 は評価せずにそのまま環境に入れる
  - 拡張した環境でクロージャの本体を評価する

  - let 式も同様に評価できる
    - 「`let x =` e_1 `in` e_2」は
      「`(fun x ->` e_2`)` e_1」と同等とみなせるので

課題 2 (15点)

- 課題 1 のインタプリタを改造して
  タプル・リストに関する操作を
  call by name で評価するようにせよ

課題 2 (続き)

- 実装法の一例:
  タプル・リストを表す値の定義を変更する
  - タプル・リスト値中の要素を
    「値」ではなく「thunk」にする

課題 3 (20点)

- 課題 2 のインタプリタ上で
  全ての素数を小さい順に一つずつ含む
  無限リスト primes を作れ
  - イメージとしては

```
primes = [2; 3; 5; 7; 11; 13; …
```

  - 整数演算のオーバーフロー等は無視してもよい

課題 4 (20点)

- 第 5 回または第 6 回または
  課題 1 または課題 2 のインタプリタを改造して
  call by need で評価を行うようにせよ
  - タプル・リストにも対応せよ

課題 5 (20点)

- 課題 2 または課題 4 のインタプリタ上で
  全ての無限長 2 進数列からなる無限集合
  cantor_space を作れ

課題 6 (20点)

- 第 5 回または第 6 回または
  課題 1 または課題 2 のインタプリタを改造して
  以下のような演算子 (|||) を扱えるようにせよ

  e_1 ||| e_2 → { true (if e1 → true or e2 → true)
  { false (if e1 → false and e2 → false)

  – (例)

```
# let rec loop x = loop x in
  (true ||| loop false, loop false ||| true)
- : bool * bool = (true, true)
```

課題 7 (20点)

- 以下の整数の無限リストの型の定義

```
type stream =
| Cons of int * stream
```

と同様の型を再帰型を用いずに定義せよ

  - またその型の値や値に対する操作も定義せよ

課題 7 のヒント

```
type 'a t' =
| Cons of int * 'a

module E = EXIST(struct type 'a t = ('a -> 'a t') * 'a end)

type stream = E.t

val conv : ('a -> 'b) -> 'a t' -> 'b t'

val unfold : ('a -> 'a t') -> 'a -> stream

val out : stream -> stream t'

val head : stream -> int

val tail : stream -> stream
```

(ここでモジュールEXIST は、第 4 回の課題 6 のもの)