ML 演習 第 6 回

2007/07/10

飯塚 大輔, 後藤 哲志, 前田 俊行

http://www.yl.is.s.u-tokyo.ac.jp/~sgotou/lecture/caml-enshu

補足: 標準出力関係

- print_int, print_string, print_float, print_char: それぞれの型の値を出力
- print_newline: 改行
- print_endline: 文字列を出力し改行
- Printf.printf: C 言語の printf 相当
  - 引数はきちんと型チェックされる
    - printf 用のちょっと特殊な型規則が存在
- その他はマニュアルを参照

今回の内容

- 言語処理系の実装 (2)
  - 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
  - (e_1, e_2)
  - $\texttt{fst}\ (e_1, e_2)$
  - $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$
  - $e_1\ e_2\ e_3\ e_4\ e_5\ e_6$

どの式 (e_i) を
どのタイミングで
どうやって
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3つの評価戦略

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

Call by value

- 関数などの引数を適用の前に評価
  - fib (1 + 1) → fib 2 →
    if 2 < 2 then 1
    else fib (2 - 1) + fib (2 - 2)
    → if false then 1
    else fib (2 - 1) + fib (2 - 2)
    → fib (2 - 1) + fib (2 - 2)
    → fib 1 + fib (2 - 2)
    → (if 1 < 2 then 1 else …)
    + fib (2 - 2)
    →* 1 + fib (2 - 2) → 1 + fib 0
    →* 1 + 1 → 2

Call by value の利点

- 高速な実装が可能
  - ほとんどの値は計算中に即値として現れる
- 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

- 結果が定まっている式の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

- 対策: いくつかの special form を用意
  - if は条件節と必要な方の節しか評価しない
    - if は関数ではない
  - ||, &&, ; なども同様

Call by name

- 外側の関数適用を引数より先に評価
  - `let f x = x * x in f (5 + 3)`
    - → `f (5 + 3)`
    - → `(5 + 3) * (5 + 3)`
    - → `8 * (5 + 3)`
    - → `8 * 8`
    - → `64`

Call by name の利点

- ある評価戦略で式の評価が止まるならば call by name で評価しても必ず止まる

```
  let rec loop x = loop x in fst (5 + 3, loop x)
→ fst (5+3, loop x) → 5 + 3 → 8
```

  - if も通常の組み込み関数として定義可能
    - let (if_f) b x y = { b を評価し x か y を返す }
    - (if_f) true (5 + 3) (loop x)
      →* 5 + 3 → 8

Call by name の欠点

- 実装が遅くなる
  - 常に「式」の形で評価を進める必要がある
- 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

- 外側の関数適用を先に評価
- 同じ式は1回だけ評価し結果を使い回す

```
let f x = x * x in f (5 + 3)
→ f (5 + 3)
→ (5 + 3) * (5 + 3)   (* 2つの (5+3) は同一 *)
→ 8 * 8
→ 64
```

- cf. Haskell

Call by need の利点

- ある評価戦略で式の評価が止まるならば call by need で評価しても必ず止まる (Call by name と同じ)
- 1つの式の評価は1回で済む

Call by need の欠点

- 実装が遅くなる
- 式の評価タイミングは依然制御困難
  - Sharing があるのでさらに複雑に

Call by value で call by name や call by need の真似をするには

- λ抽象を評価順の制御に使えばよい
  - `let if_f b x y = if b then x else y`
  - `if_f true 5 (loop x)` → 発散
  - `let if_d b x y =`
    `  if b then x () else y ()`
  - `if_d true (fun () → 5) (fun () → loop x)`
    `→* if true then (fun () → 5) ()`
    `  else (fun () → loop x) ()`
    `→ (fun () → 5) () → 5`

遅延評価の実装

- module Delayed: Call by name の実現
  - delay : 遅延評価される式を表すデータを生成
    - 使い方: **delay (fun () -> 式)**
  - force : delay された式の実際の値を得る

遅延評価の実装を用いた if 式の実現例

- if の2つの選択肢を遅延評価

```
# open Delayed;;
# let lazy_if b x y =
    if b then (force x) else (force y)
val lazy_if : bool → 'a Delayed.delayed →
                     'a Delayed.delayed → 'a =<fun>
# let lazy_fact x =
    lazy_if (x = 0)
        (delay (fun () -> 1))
        (delay (fun () -> x * lazy_fact (x - 1)));;
val lazy_fact : int -> int = <fun>
# lazy_fact 10;;
- : int = 3628800
```

第 6 回 課題

締め切り: 7/24 13:00

課題1（必須）

- module Delayed を Call by need で再実装せよ。
  - 1回 force された値を記憶しておき同じ値が2回以上 force された場合でも評価が1回しか起こらないようにする。
  - 現状:
    - let p = delay (fun () -> print_endline "eval!"; 5 + 3) in (force p) * (force p) とやると eval! が2回表示される。

課題 1（ヒント）

- データ構造: Reference
  - 評価される前の値：`unit -> 'a`
  - 評価された後の値：`'a`

  を格納できる reference を作って
  前者を1回評価したら後者に書き換える

  - global な記憶を作る戦略はうまくいかない
    - 型が違う delayed value を扱えない

課題 2 (必須)

- 遅延 (無限) リストを表現する module Sequence を実装せよ
  - tail 部分を遅延させる
    - `head: 'a seq -> 'a` (* 先頭要素を取得 *)
    - `tail: 'a seq -> 'a seq` (* 先頭を除いた Seq. *)
    - `nil: 'a seq` (* 空 Sequence *)
    - `cons: 'a -> 'a seq delayed -> 'a seq`
    - `take: 'a seq -> int -> 'a list`
      - take s n : s の先頭最大 n 要素を取り出す

課題 3 (必須)

- 課題 2 で実装した
  モジュール Sequence を用いて
  小さい順に素数が並んだ
  無限リスト primes を作れ

```
# take primes 10;;
- : int list = [2; 3; 5; 7; 11; 13; 17; 19; 23; 29]
```

課題 4 (必須)

- 前回のインタプリタを Call by name で再実装せよ
  - データ構造は任意: 必ずしも配布ソースの `type lazymlvalue` に従わなくてもよい
    - データ型を変えた際は make clean; make で `Reader` などを作り直す必要がある

課題 4 (ヒント)

- 戦略
  - force → 式の形を見ながら
    今すぐに必要なところだけを評価
    - Primitive: 引数がすべて必要
    - Cons, Pair: 引数は delay されていてよい
    - Apply: 左辺の値 (λ式) が必要
    - Match: パターンマッチに必要な部分だけは
      force されていないといけない
      - 変数や _ にマッチする部分は force しなくてよい

課題 5 (Optional)

- 課題4で実装したインタプリタを Call by need で再実装せよ