ML演習 第 3 回

2007/06/19

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

今日の内容

- 例外
- 副作用を利用したプログラミング
  - Reference
  - 代入可能フィールド
  - Value restriction

今日の内容

- 例外
- 副作用を利用したプログラミング
  - Reference
  - 代入可能フィールド
  - Value restriction

例外処理とは?

- エラーが発生したときに現在の計算を中断してエラー処理用のコードにジャンプする機構
  - 開こうとしているファイルが見つからない
  - 配列の境界を越えてアクセスした
  - ユーザの入力がおかしい
  - 関数に渡されるべきでない値が渡された
  - etc.
- C++ や Java にも同様の機構がある

例外処理の動作(1)

```
let f x =
  ...
  try
    g x y
  with
    ExceptionC -> ...
  | ExceptionB -> ...
```

```
let g a b =
  try
    if invalid a then
      raise ExceptionA
    else if b < 0 then
      raise ExceptionB
    else
      ...
  with
    ExceptionA -> ...
```

例外処理の動作(2)

```
let f x =
  ...
  try
    g x y
  with
    ExceptionC -> ...
  | ExceptionB -> ...
```

ここにジャンプ

```
let g a b =
  try
    if invalid a then
      raise ExceptionA
    else if b < 0 then
      raise ExceptionB
    else
      ...
  with
    ExceptionA -> ...
```

ここには無いので

例外機構の使用例 (1)

例外の定義

例外の送出

- (0, 0) が与えられたら例外を投げる

```
# exception ZeroVector;;
exception ZeroVector
# let normalize (x, y) =
    let n = sqrt (x *. x +. y *. y) in
    if n = 0.0 then raise ZeroVector
    else (x /. n, y /. n);;
val normalize :
  float * float -> float * float = <fun>
# normalize (3.0, 4.0);;
- : float * float = 0.6, 0.8
# normalize (0.0, 0.0);;
Exception: ZeroVector.
```

例外機構の使用例 (2)

エラー処理を記述

```
# let angle_str v1 v2 =
    try
      let ((x, y), (x', y')) =
          (normalize v1, normalize v2)
      in string_of_float
        (acos(x *. x' +. y *. y'))
    with ZeroVector -> "Not defined.";;
  val angle_str : float * float ->
    float * float -> string = <fun>
  # angle_str (0.0, 0.0) (0.0, 0.5);;
  - : string = "Not defined."
```

例外のまとめ (1)

- 例外の定義
  - `exception ZeroVector`
  - `exception BadArg of float`
    - 引数つきの例外も可能 (バリアントと同じ)
- 例外の送出
  - `raise ZeroVector`
  - `raise (BadArg 3.0)`

例外のまとめ (2)

- 例外のキャッチ (例外ハンドラの登録)
  - `try … with ZeroVector -> …`
  - `try … with BadArg x -> …`

ここで x が使える

- with の後はパターンマッチになっている
  - ```
    try … with
       ZeroVector -> …
     | BadArg x -> …
     | _ -> …
    ```
    ```

今日の内容

- 例外
- 副作用を利用したプログラミング
  - Reference
  - 代入可能フィールド
  - Value restriction

unit 型

- () を唯一の値として持つ型

```
# ();;
- : unit = ()
```

- 用途
  - C の void 型に相当
  - 副作用以外に意味のない関数や式の返り値
    - print_string や reference への代入(後述) など
  - 引数の不要な関数に与えるダミー引数

Reference

- 中身を変更できるセル (“箱”)
  - C や Scheme の変数に相当

```
# let a = ref 0;; (* 0 で初期化した参照を作る *)
val a : int ref = {contents = 0}
# !a;; (* 参照先を取り出す *)
- : int = 0
# a := 5;; (* 参照に代入 *)
- : unit = ()
# !a;; (* 再び参照先の値を取り出す *)
- : int = 5
```

返り値は unit 型

Mutable Field

- 値を変更できる record 中のフィールド

```
# type mutable_point =
    { mutable x : int; mutable y : int };;
type mutable_point =
    { mutable x : int; mutable y : int; }
# let p1 = { x = 5; y = 3; };;
val p1 : mutable_point = { x = 5; y = 3 }
# p1.x <- 6;;
- : unit = ()
# p1;;
- : mutable_point = { x = 6; y = 3 }
```

逐次実行

- 複数の式を順に評価
  - Scheme の begin に相当

```
# let increment x a = (x := !x + a ; !x);;
val increment : int ref -> int -> int = <fun>
# let a = ref 0;;
val a : int ref = {contents = 0}
# increment a 1;;
- : int = 1
# increment a 2;;
- : int = 3
```

最後の式の結果 (!x) が
全体の結果になる

型多相と reference は相性が悪い

- (実際にはエラーになる) 例

```
# let r1 = ref [];;
val r1 : 'a list ref = { contents = <fun> }
# let f () = List.map not (!r1);;
val f : unit → bool list = <fun>
# r1 := [5];;
- : unit = ()
# f ();;
```

- どこがおかしい？

O'Caml の解決案

- なんでもかんでも多相型にしない
- reference には「未決定な単相型」を与える

```
# let r1 = ref [];;
val r1 : '_a list ref = { content
# let f1 () = List.map not !r1;;
val f1 : unit -> bool list = <fun>
# let _ = !r1 @ [5];;
This expression has type int but is here used
with type bool
# !r1;;
  - : bool list = []
```

'_a: 一旦型が確定すると以降は他の型としては使えない型

bool に確定すると

もう int としては使えない

問題は reference 以外にも

- unit->'a->'a 型の関数に () を適用したら？
  - fun () -> (fun x -> x) なら 'a -> 'a でいいが..
  - 次の場合は単相型 '_a - >'_a でないとダメ

```
let f () =
  let r = ref None in
  fun x ->
  let old = match !r with
            None -> x | Some y -> y
  in r := Some x; old
```

O'Caml の最終的な解決案

- Value restriction:
  副作用がないと確実にわかる「値」
  にのみ多相型を与える
  - 値 = それ以上評価されることがない式
    - 結局のところ, 評価される式は副作用を持ちうるので
  - OK: 定数、fun 式、それらの tuple、
    それらからなる変更不可データ構造
  - NG: reference、let 式、関数適用 etc...

```
# (fun () x -> x) ();;
- : '_a -> '_a = <fun>
```

Value restriction: 注意すべきこと

- 部分適用が単相型になることがある

```
# let f = List.map (fun x -> (x, x));;
  (* 多相型になってほしいが、値ではないので
     未決定単相型になってしまう *)
val f : '_a list -> ('_a * '_a) list = <fun>
```

- 解決策: η 展開 (仮引数を明示する)

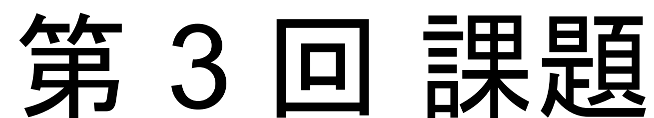
```
# let f xs = List.map (fun x -> (x, x)) xs;;
val f : 'a list -> ('a * 'a) list = <fun>
```

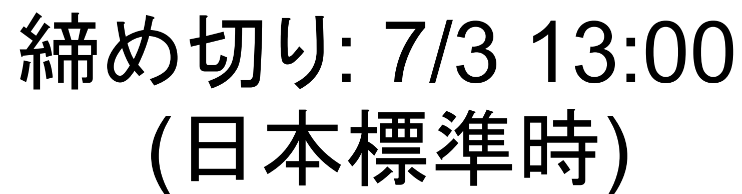
Value restriction: 参考文献

- 最初に Value restriction を提案した論文
  - Andrew K. Wright, Matthias Felleisen.
    A Syntactic Approach to Type Soundness.
    - Value restriction 以外の解決法との得失比較あり
- 提案された O'Caml の拡張
  - Jacques Garrigue. Relaxing Value Restriction.
    - OCaml 3.07 で採用

アドバイス

- 論文の探し方
  - 基本的には Google、Google Scholar
    - タイトル, 著者名, 発表された会議や雑誌で検索
  - ACM Digital Library (http://www.acm.org から)
    - ACMの学会で発表された論文ならここにもある
  - CiteSeer.IST (http://citeseer.ist.psu.edu/cis/)
    - 論文データベース
    - Google で検索すると大抵ここの検索結果がかかる
  - 著者の Web サイト

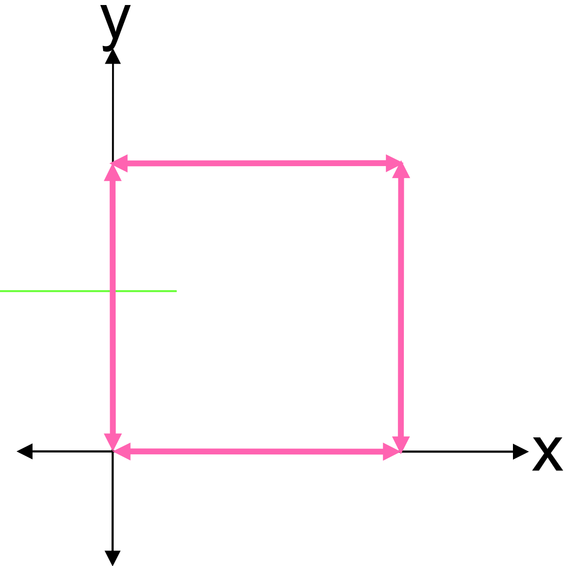

第３回 課題


締め切り: 7/3 13:00
(日本標準時)

課題 1 (必須)

- 詳細は別紙参照

課題1 (例)




```
# let t1 = new_turtle ();;
val t1 : turtle = { … }
# advance t1 1.0; locate t1;;
- : float * float = (1., 0.)
# rotate t1 90.0; advance t1 1.0; locate t1;;
- : float * float = (1., 1.)
# rotate t1 90.0; advance t1 1.0; locate t1;;
- : float * float = (0., 1.)
# rotate t1 90.0; advance t1 1.0; locate t1;;
- : float * float = (0., 0.)
```

(*計算誤差で値が正確に0にならないのは気にしなくて良い*)

課題2 (必須)

- スタックのデータ構造を表現する多相型を定義して次の操作を実装せよ

```
type 'a stack = { mutable s : 'a list }
val new_stack : unit -> 'a stack
(* ↑新しい stack を作成する *)
val push : 'a stack -> 'a -> unit
(* ↑要素を push する *)
val pop : 'a stack -> 'a
(* ↑要素を pop する。stack が空なら EmptyStack 例外を投げる *)
```

課題2 (例)

```
# let s = new_stack ();;
val s : '_a stack = {s = []}
# push s 1;;
- unit = ()
# push s 2;;
- unit = ()
# pop s;;
- : int = 2
# pop s;;
- : int = 1
# pop s;;
Exception : EmptyStack.
```

課題3 (必須)

- 課題2の実装で、`let s = new_stack ()` に対するインタプリタの応答が `val s : '_a stack = {s = []}` となっている。これについて以下の問いに答えよ。

  1. `'a stack` と `'_a stack` の違いを説明せよ
  2. 仮に型が `'_a stack` ではなくて `'a stack` となったとしたら、どのような問題が生じるか説明せよ

課題4 (Optional)

- 課題2と同様に、今度はキューのデータ構造を表現する多相型を定義して次の操作を実装せよ
  - ただし、各操作はキューの長さによらず定数時間 (O(1)) で終了するようにすること

```
type 'a queue = ???
val new_queue : unit -> 'a queue
(* ↑新しい queue を作成する *)
val add : 'a queue -> 'a -> unit
(* ↑要素を追加する *)
val take : 'a queue -> 'a
(* ↑要素を取り出す。queue が空なら EmptyQueue 例外を投げる *)
```

課題5 (Optional)

- 関数 f を受け取って, f を再帰的に無限回合成する関数を返す関数 inf を書け
  - 以下のようなイメージ

```
 inf f ≡ f (f (f (f (f (f (f ...)))))))
# let fib = inf (fun g n ->
                 if n <= 2 then 1
                 else g (n - 1) + g (n - 2));;
  val fib : int -> int = <fun>
# fib 10;;
  - : int = 55
```

  - **ref は使ってもよいが let rec を使ってはいけない**