ML 演習 第 4 回

2007/06/26

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

ICFP Programming Contest

- 過去の O'Caml プログラムの実績
  - 1998: 2位 (ENS Camlist, France)
  - 1999: 1位 (Camls 'R Us, O'Caml 作者グループ)
  - 2000: 1位 (PLClub, U-Penn, 米澤研 住井, 細谷 参加)
    2位 (Camls 'R Us)
  - 2001: 入賞選外 (3位タイ)
  - 2002: 1位 (TAPLAS, 米澤研: 大岩, 関口, 住井)
  - 2003: 入賞選外
  - 2004: 入賞選外 (1位から4位まで Haskell)
  - 2005: 入賞選外 (1位: Haskell, 6位：O'Caml)
  - 2006: 入賞選外 (1位: Haskell, 4位,6位,7位: O'Caml)

等値演算子 (1)

- 2 種類の等値演算子がある
  - = (否定: <>):「構造的な一致」
    - 複合データの中身まで探索
    - Scheme の equal? に相当
  - == (否定: !=):「物理的な一致」
    - 「アドレス」のみを見る
    - Scheme の eq? に相当
  - == の方が識別力が強い
    - (x == y) が成り立てば (x = y) も成り立つ
      - ただし例外もある（nanなど）

より詳しい定義は
マニュアル参照

等値演算子 (2)

```
let test x y = (x = y, x == y);;
val test : 'a -> 'a -> bool * bool = <fun>
# test 1 1;;
- : bool * bool = true, true
# test 1.0 1.0;;
- : bool * bool = true, false
# test "string" "string";;
- : bool * bool = true, false
```

float や string の定数は
別々の場所に確保される
ことがある

等値演算子 (3)

```
# test (ref 1) (ref 1);;
- : bool * bool = true, false
# let r = (ref 1) in test r r;;
- : bool * bool = true, true
# (fun x -> x) = (fun x -> x);;
Exception: Invalid_argument
       "equal: functional value"
# (fun x -> x) == (fun x -> x);;
- : bool = false
# let f = (fun x -> x) in test f f;;
Exception: Invalid_argument
       "equal: functional value"
```

参照先が等しければ参照どうしも等しい

異なる場所に確保された値への参照なのでfalse

関数どうしが内容的に等しいかどうかは一般に決定不能

比較演算子

- <, >, <=, >=
  - 型: 'a -> 'a -> bool
    - 何でも比べられる
  - = と対応した演算子
  - 整数・実数 → 数値で比較
  - 文字列・文字 → 辞書式順序で比較
  - その他のオブジェクト → 実装依存
    - 注: 循環参照を持つデータでは止まらないことがある

識別子について

- 利用可能文字
  - 先頭文字: A〜Z, a〜z, _ (小文字扱い)
  - 2文字目以降: A〜Z, a〜z, 0〜9, _, ' (プライム)
- 先頭の文字の case で2つに区別
  - 小文字: 変数, 型名, レコードの field 名
    (ラベル, クラス名, クラスメソッド名)
  - 大文字: コンストラクタ名, モジュール名
  - 任意: モジュール型名

alias pattern

- パターンマッチの結果に別名を与える

```
# match (1, (2, 3)) with (x, (y, z as a)) -> a
- : int * int = (2, 3)
```

  - 結合が弱いので注意
    - 上の例では y, (z as a) ではなく
      (y, z) as a と結合している

今回の内容

- O'Caml のモジュールシステム
  - structure
  - signature
  - functor
- O'Caml コンパイラの利用

□今日使うソースは演習ホームページに置いてあります

今回の内容

- O’Caml のモジュールシステム
  - structure
  - signature
  - functor
- O’Caml コンパイラの利用

大規模ソフトウェアのプログラミングは何故難しいか?

- 人間が記憶できるプログラムの量には限界があるから
  - 例1: O'Caml 処理系のソースプログラム全てを記憶している人は (多分) いない
  - 例2: Linux カーネルのソースプログラム全てを記憶している人は (多分) いない

ではどうするか?

- 答: 複数人でプログラミングする
  - 10人でやれば1人あたりの量は10分の1に
  - 100人でやれば100分の1に
  - 1000000人でやれば1000000分の1に...
  - ...

ならない

複数人でのプログラミング
最悪のシナリオ

- 他人の書いたプログラムは読みにくい

→ 自分で書いたほうが早い

→ 全員同じようなプログラムを書くことになる

- 似たようなプログラムがたくさんできる

→ しかもそれぞれ微妙に違っている

→ プログラムの改善・修正がとても大変になる

大規模プログラミングとモジュール

- O'Caml は大規模プログラミングに有用な機能をモジュールとして提供している
  - 仕様と実装の切り分けの明確化
    - 細かい実装の変更から利用者を守る
    - 仕様を変えない範囲で実装の変更を自由にする
  - 部品の再利用
    - 同じ構造を持つコードを共通化する
  - 名前の衝突の回避
    - 適切な「名前空間」の分離

O'Caml の モジュールシステム

- structure：名前空間を提供
  - プログラムの実装をモジュールとして分離
- signature：モジュールの仕様を定義
  - プログラムの実装 (値・型など) を隠蔽
- functor：structure を受け取る関数 のようなもの
  - 共通の構造をもった structure を生成できる

structure の書き方

- 変数や型などの定義の集合
  - 例: MultiSet (lecture4-1.ml)

```
module MultiSet =
  struct
    ...
    ...
  end
```

ここに
変数や型などを定義

structure の使い方(1)

- 内部の変数には．(ドット) 表記でアクセス

```
# MultiSet.empty;;
- : 'a MultiSet.set = MultiSet.Leaf
# let a = MultiSet.add MultiSet.empty 5;;
val a : int MultiSet.set = MultiSet.Node
              (5, MultiSet.Leaf, MultiSet.Leaf)
# MultiSet.member a 5;;
- : bool = true
```

Multiset モジュールの empty という変数にアクセス

structure の使い方(2)

- open: structure を「開く」
  - structure 内の定義を ． 無しでアクセス

```
# open MultiSet;;
# add empty 5;;
- : int MultiSet.set = Node (5, Leaf, Leaf)
# member (add empty 5) 10;;
- : bool = false
```

Multiset. を付けなくても
Multiset.add にアクセスできる

signature の書き方

- structure に対する「型」
  - 公開する/隠蔽する変数や型を指定できる
  - 例: MULTISET: 重複集合の抽象化

```
module type MULTISET =
  sig
    ...
    ...
  end
```

ここに
変数や型などを定義

MULTISETによる変数や型の隠蔽

- `type 'a set` は存在だけが示されている
  - モジュールの外からは `'a set` の定義が見えない
    - `'a set` の実装が変わっても, 使う側には影響ナシ
- `remove_top` は MULTISET にはない
  - `remove_top` はモジュールの内側でしか見えない

signature の適用 (1)

- signature を structure に適用
  = structure の中身を (一部) 隠す
  - (例) MULTISET で MultiSet に制限をかける

```
# module AbstractMultiSet =
                        (MultiSet : MULTISET);;
module AbstractMultiSet : MULTISET
# let a = AbstractMultiSet.empty;;
val a : 'a AbstractMultiSet.set = <abstr>
# let b = AbstractMultiSet.add a 5;;
val b : int AbstractMultiSet.set = <abstr>
```

抽象データ型の内容は隠蔽される

signature の適用 (2)

```
# open AbstractMultiSet;;
# let a = add (add empty 5) 10;;
val a : int AbstractMultiSet.set = <abstr>
# AbstractMultiSet.remove_top;;
Unbound value AbstractMultiSet.remove_top;;
# MultiSet.remove_top a;;
This expression has type int AbstractMultiSet.set
  but it is used with type 'a MultiSet.set
```

remove_top は
外から見えない

int AbstractMultiSet.set と
'a MultiSet.set は違う型!!

functor

- structure から structure への関数のようなもの
  - 例: lecture4-2.ml
    - signature ORDERED_TYPE
      - 一般の全順序・等値関係つきの型
    - functor MultiSet2
      - ORDERED_TYPE を持つ structure に対する集合の定義

functor の書き方

- 例: MultiSet2 (lecture4-2.ml)

```
module MultiSet2 =
   functor (Elt : ORDERED_TYPE) ->
   struct
      ...
      ...
   end
```

functor と signature

- functor にも signature を定義できる
  - SETFUNCTOR: MultiSet2 に対する functor signature
    - elem の型は concrete (Elt.t)
    - t の型は abstract
  - AbstractSet2: SETFUNCTOR で制限した functor MultiSet2

functor の signature の書き方

- 例: SETFUNCTOR (lecture4-2.ml)

```
module type SETFUNCTOR =
   functor (Elt : ORDERED_TYPE) ->
   sig
      ...
      ...
   end
```

functor と signature (2)

```
# module AbstractStringSet =
    AbstractSet2(OrderedString);;
module AbstractStringSet : sig ... end
# let sa = AbstractStringSet.add
  AbstractStringSet.empty "Ocaml";;
val sa : AbstractStringSet.t = <abstr>
# AbstractStringSet.member sa "ocaml";;
- : bool = false
```

functor と signature (3)

```
# module NCStringSet = AbstractSet2(NCString);;
module NCStringSet : sig ... end
# let sa = NCStringSet.add NCStringSet.empty
  "Ocaml";;
val sa : NCStringSet.t = <abstr>
# NCStringSet.member sa "ocaml";;
- : bool = true
# AbstractStringSet.add sa "ocaml";;
This expression has type NCStringSet.t =
  AbstractSet2(NCString).t but is here used with
  type AbstractStringSet.t =
  AbstractSet2(OrderedString).t
```

渡す structure を
変えるだけで
新しい Set を作れる

今回の内容

- O’Caml のモジュールシステム
  - structure
  - signature
  - functor
- O’Caml コンパイラの利用

O'Caml のコンパイラ (1)

- モジュール単位の分割コンパイルをサポート
- Unix の実行形式ファイルを作成
  - 複数の backend
    - ocamlc: バイトコードコンパイラ
      - バイトコードインタプリタ (ocamlrun) を実行に使用
      - デバッガをサポート
    - ocamlopt: ネイティブコードコンパイラ
      - SPARC や x86 などの機械語を直接生成

O'Caml のコンパイラ (2)

- 拡張子一覧
  - ソースファイル
    - .ml → module の実装 (structure)
    - .mli → module のインタフェース (signature)
  - オブジェクトファイル
    - .cmo → 実装のバイトコード
    - .cmi → インタフェース定義のバイトコード
    - .cmx → 実装のネイティブコード
    - .cma, .cmxa → ライブラリ

分割コンパイルとモジュール

- *.ml と *.mli
  - 実装とインタフェースをそれぞれ記述
    - module SomeThing :
      sig [someThing.mli の内容] end
      = struct [someThing.ml の内容] end
      に相当 (モジュール名の先頭を小文字化)
  - .mli をコンパイル → .cmi を生成
  - .ml をコンパイル → .cmi が無ければ制約無しで生成、あれば型チェック

.mli, .ml を用いたモジュール定義の例

- mySet.mli, mySet.ml
  - module MySet の定義
- uniq.ml
  - メインプログラムのモジュール

分割コンパイルの例

```
% ocamlc -c mySet.mli
% ocamlc -c mySet.ml
% ocamlc -c uniq.ml
% ls -F *.cm*
mySet.cmi    mySet.cmo    uniq.cmi
  uniq.cmo
% ocamlc -o myuniq mySet.cmo uniq.cmo
% ls -F myuniq
     myuniq*
```

順序が重要:
モジュールの定義/依存順

実行例

```
% ./myuniq
OCaml
Standard ML
C++
OCaml
^D
      1 C++
      2 OCaml
      1 Standard ML
%
```

.cmo ファイルをインタプリタで利用する方法: #load を用いる

```
# #load "mySet.cmo";;
# MySet.empty;;
- : 'a MySet.set = <abstr>
# MySet.remove_top;;
Unbound value MySet.remove_top
# open MySet;;
# empty;;
- : 'a MySet.set = <abstr>
```

第 4 回 課題

締切: 7/10 13:00

課題1 (必須)

- myuniq の例を自分でやってみよ
  - 例に従って実行ファイル生成し、実行してみよ
  - .cmo ファイルをインタプリタで利用してみよ
  - .mli をコンパイルしないと
    どうなるか試してみよ
  - 最後のリンク時にモジュールの順番を変えると
    どうなるか試してみよ

課題2 (必須)

- リスト等の別のデータ構造を使って signature MULTISET に対する別の実装を与えよ

課題3 (必須)

- lecture4-ex3.ml は簡単な
  パスワード付き銀行口座の例であるが
  `fst a1` や `BankAccountImpl.accounts` 等で
  秘密の情報である暗証番号や口座一覧が
  操作可能であり問題である
- そこで、この moduleに適用する
  signature を作りこれらの情報を隠蔽せよ

課題3 (仕様)

- 公開すべきもの
  - `new_account, deposit, withdraw, balance, bank_statistics`
  - 型 `account` の存在
- 隠蔽すべきもの
  - 型 `account` の実装: 残高を操作できる
  - 値 `accounts`: 他口座の instance が得られる
  - その他の関数: 認証を回避できる

課題4 (必須)

- ORDERED_TYPE で表現される型の key と任意の型の値についての連想配列を作り出す functor を作れ

課題4 (例1)

```
# module NCStringAssociation =
  Association(NCString);;
module NCStringAssociation :
    sig
        type key = NCString.t
        and 'a t = 'a Association(NCString).t
        val empty : 'a t
        val add : 'a t -> key -> 'a -> 'a t
        val remove : 'a t -> key -> 'a t
        val get : 'a t -> key -> 'a
        exception Not_Found
    end
```

課題4 (例2)

```
# open NCStringAssociation;;
# let sa = add empty "C" "/* */";;
val sa : string NCStringAssociation.t = <abstr>
# let sa = add sa "Ocaml" "(* *)";;
val sa : string NCStringAssociation.t = <abstr>
# let sa = add sa "Perl" "#";;
val sa : string NCStringAssociation.t = <abstr>
# get sa "ocaml";;
- : string = "(* *)"
```

課題5 (Optional)

- 等値演算子 (=) を関数として書けるか?
  - ただし関数内で等値演算子、比較演算子を使ってはいけない