ML 演習 第5回

おおいわ

June 27, 2000

今回の内容

- ocamlc: Ocaml コンパイラ
  - 分割コンパイルとモジュール
- 遅延評価による無限データ構造

ocamlc (1)

- Ocaml のコンパイラ
  - モジュール単位の分割コンパイルサポート
  - 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
    - ocamlopt: ネイティブコードコンパイラ

ocamlc (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

分割コンパイル (1)

- .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
  - .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

分割コンパイル (2)

- 例
  - mySet.mli, mySet.ml
    - module MySet の定義 (内容はほぼ第4回の実装)
  - uniq.ml
    - メインプログラムのモジュール

分割コンパイル (3)

- 実行例 (1)

```
% 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*
```

分割コンパイル (4)

- 実行例 (1)

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

分割コンパイル (5)

- .cmo のインタプリタでの利用

```
# #load “mySet.cmo”;;
# MySet.empty;;
- : ’a MySet.set = <abstr>
# MySet.remove_top;;
Unbound value MySet.remove_top
```

値の評価戦略

- 計算順序に関する戦略
  - eager な評価
    - “call by value”
    - 関数の引数などは常に先に計算してから呼出
  - lazy な評価
    - “call by name”, “call by need”
    - 値が実際に必要になるまで評価しない

評価戦略の得失

- eager な評価
  - 得: 効率がよい、実装が簡単
  - 得: semantics が単純で理解が容易
  - 失: lazy なら止まる計算が止まらないことがある

```
# let rec iter x = 1 + iter x;;  (* infinite loop *)
# let y = let _ = iter 3 in 5;;  (* y = 5 ? *)
```

評価戦略の得失 (2)

- lazy な評価
  - 得: 計算能力が高い
    - eager に計算できる式は必ず lazy でも ok
  - 失: 実装が複雑、速度が遅い
    - 結局同じだけ評価するなら、lazy にする
      余計な処理の分だけ重い
  - 失: semantics がわかりにくい
    - 特に副作用がいつ起こるかわかりにくいので
      手続き型ではほとんど使い物にならない

遅延評価 (1)

- module Delayed (lecture5.ml)
  - delay: 遅延評価される式を表す object を生成
    - 使い方: delay (fun () -> 式)
      - closure のおかげで後で評価に必要な情報はこれですべて保存されている
  - force: delayed expression を実際に評価

遅延評価 (2)

- 例 (1)

```
# let eager_if b x y = if b then x else y;;
val lazy_if : bool -> 'a -> 'a -> 'a = <fun>
# let rec eager_fact x = eager_if (x = 0) (1)
                                  (x * eager_fact (x - 1));;
val eager_fact : int -> int = <fun>
# eager_fact 10;;
(止まらない...)
```

遅延評価 (3)

- 例 (2): 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
```

遅延評価とデータ構造

- データ構造と遅延評価を組み合わせる
  - 木構造の探索などで不要な計算を省く処理を素直に記述できる
  - 無限に続くデータを記述してそれに対する処理を行うことができる
    - 例: 無限リスト構造 (Sequence と呼ぶことにする) の遅延評価による記述

Sequence (1)

- Module Sequence (lecture5.ml)
  - データ型 Sequence.seq
  - Sequence に対する操作 (1)
    - head: 先頭要素を取り出す
    - tail: 残りの要素を取り出す
    - nil: 空リスト
    - is_empty: 空リストかどうかを判定

Sequence (2)

- Module Sequence (lecture5.ml) [続き]
  - Sequence に対する操作 (2)
    - cons: Sequence を構築する
      - 引数: 先頭要素と、第2要素以降をdelay したもの
    - filter, map, append: リストの操作に対応
    - take, drop: 先頭の n 要素を取得/捨てる
    - to_list, of_list: 有限 sequence とリストの変換
    - interleave: 要素を交互に入れた sequence
    - iterate: [x, $f(x)$, $f(f(x))$, $f^3(x)$, $f^4(x)$, ...] を作る

Sequence の操作 (1)

- Sequence.iterate: 無限データを生成
  - delay した cons の第2要素で自分を再帰的に用いて無限データを順次生成

```
# let s1 = make_int_sequence 1;;
val s1 : int Sequence.seq = <abstr>
# Sequence.take s1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
# Sequence.take s1 10;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8; 9; 10];;
```

Sequence の操作 (2)

- Sequence.filter: 無限 sequence を操作
  - 無限に辿る部分は delay されている

```
# let s1 = make_int_sequence 1;;
val s1 : int Sequence.seq = <abstr>
# let s2 = Sequence.filter (fun x -> x mod 2 = 0) s1;;
val s2 : int Sequence.seq = <abstr>
# Sequence.take s2 10;;
- : int list = [2; 4; 6; 8; 10; 12; 14; 16; 18; 20];;
```

課題1

- Sequence の操作 (1) で、take s1 10 を2回行うと、lecture5 の Delayed の実装では実際に2回 Sequence を計算してしまう。

  そこで、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module Delayed を再実装せよ。

課題2

- lecture5.ml で、未実装として残されている関数群を実装せよ。
  - drop, map, of_list, append, interleave

課題3

- 「3の倍数または5の倍数」である自然数からなる無限 Sequence を生成せよ。

```
# Sequence.take s1 10;;
- : int list = [3, 5, 6, 9, 10, 12, 15, 18, 20, 21]
```

  - ヒント: 3の倍数 sequence と 5 の倍数 sequence を作り、interleave 類似の補助関数で1つにまとめるとよい。

課題4 (optional)

- 素数からなる無限sequenceを生成せよ。

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

  - ヒント: filter をうまくつかおう。

提出方法

- 〆切: 2000年7月10日 (月) 24:00
- 提出先: ml-report@yl.is.s.u-tokyo.ac.jp
- 題名: Report 5 (学生証番号)