プログラム言語論-第 6 週-

亀山幸義

筑波大学コンピュータサイエンス専攻

筑波大学 情報科学類 講義, 2009 年 5 月 25 日

プログラミング・コンテスト

ACM 主催: International Collegiate Programming Contest

- 国内予選⇒アジア予選⇒世界大会 (旅費支給!)
- 3 名一組 (大学院 1 年生まで OK)
- 詳細は、Google ACM Japan で。
- 興味がある人は、亀山あてにメイルしてください。

今回の概要

以下の概念を説明する.

- 関数クロージャ (関数の処理における Access Link のかわり).
- 評価順序 (キーワード: 名前呼び, 値呼び, 必要呼び)
- 制御構造 (キーワード: 末尾再帰, 例外機構、継続)
- データ構造 (キーワード: ヒープ, ごみ集め)
- 型システム (型検査, 型推論)

全てを詳しく説明するのは無理なので、主要部分 (赤字で表示) のみ詳しく。
今回の授業で説明しきれない題材については、2 学期の「宣言型プログラム論」(南出先生) や「ソフトウェアサイエンス実験: J-8 関数プログラミング」(亀山) を参照。

関数クロージャ-1

関数クロージャ (function closure)

- 静的束縛のプログラム言語における「計算結果 (値) としての関数」のこと。
- 具体的には、「関数」と「環境」(あるいは「状態」) のセット: (`fun` x `->` e, σ).
- 関数を得たとき (式を計算して, その結果が関数になったとき) の環境を保存しておく。

関数クロージャ-2

```
let x=1 in
let f=(fun y-> x+y) in
let x=2 in
  f 3
```

式 (fun y→ x+y) の計算結果は、((fun y→ x+y), (x=1)) である。これにより、x の (静的束縛による) 値が保存され、後にこの関数が使われるときに、関数を定義した瞬間の環境における x の値が使われる。
動的束縛のときは, 関数クロージャは必要ない.

評価順序とは

先週の課題から:

- OCaml 処理系を使って、ML 言語の構文や意味を理解し、以下の事を調べよ。
  - 静的束縛であるか、動的束縛であるか。
  - 関数呼び出しの際の評価順序は、名前呼びであるか、値呼びであるか。
  - (e1 e2) では、評価順序は、e1 が先か e2 が先か。

評価順序 (evaluation order, 計算の順序): 1 つのプログラムにおいて、どの部分 (部分プログラム) から計算するか。
評価戦略 (evaluation strategy) とも言う。

評価順序-例 1

式 $((1+2)*(3+4))*0$

- $(1+2)$ から計算する。
- $(3+4)$ から計算する。
- $(1+2)$ と $(3+4)$ から (2 つ同時に) 計算する。
- $\cdots*0$ から計算する。

$+$ のような primitive function (プログラム言語にもともと備わっている関数) については、それごとに評価順序が決まっている。

評価順序-例 2

式 (fun x → x*3) (1+2)

- (1 + 2) から計算して、(fun x → x*3)3 を得る。(値呼び計算)
- 関数適用から計算して、(1+2)*3 を得る。(名前呼び計算)

名前呼び計算 call-by-name

関数呼出し e_1 e_2 の計算 (静的束縛言語と仮定)。

- e_1 を計算して，値 v_1 を得る．
- v_1 が関数でないとき，エラー．
- v_1 が関数クロージャ(fun x -> e_0, σ') のとき，環境 σ' に $x = e_2$ を追加．
- その環境で e_0 を計算して，その結果を全体の答えとする．

(fun x-> x+x+x) (power 2 10) の計算で、「2 の 10 乗」は 3 回計算される。
(fun x-> 0) (power 2 10) の計算で、「2 の 10 乗」は計算されない。
C 言語のマクロ展開は、名前呼びの一種と考えられる。

理論と実装

理論的には、引数の計算をなるべく繰返さない計算方式が優秀⇒必要呼びが最善の戦略。
実装上は、名前呼び、必要呼びは必ずしも効率良く実装できない。⇒引数を (値ではなく) 式のまま保存しておく必要がある。
実用的なプログラム言語の多くにおいて、基本的な関数呼び出しは値呼び。

データ構造

先週の課題から:

- MiniML 言語の「対 (ペア)」データ型を使えば、2 分木を表現することができる。C 言語でも構造体 (struct 型) を使えば、ほぼ同様のことができるが、少し違いがある。
  - MiniML 言語では、一度作った 2 分木 (のためのメモリ) は、どうなるだろうか？ 2 分木を大量に作り続ける関数を作成して呼びだして、実験せよ。
  - MiniML 言語処理系における「2 分木などのデータを覚えておくためのメモリ領域」は、スタック上に取られるかどうか考えなさい。

値呼び計算 call-by-value

関数呼出し e_1 e_2 の計算 (静的束縛言語と仮定)。

- まず，e_1 と e_2 を計算して，値 v_1 と v_2 を得る．(ここで，e_1 が先かどうかで 2 通りの順序がある．)
- v_1 が関数でないとき，エラー．
- v_1 が関数クロージャ(fun x -> e_0, σ') のとき，環境 σ' に $x = v_2$ を追加．
- その環境で e_0 を計算して，その結果を全体の答えとする．

(fun x→ x+x+x) (power 2 10) の計算で、「2 の 10 乗」は 1 回計算される。
(fun x-> 0) (power 2 10) の計算で、「2 の 10 乗」は 1 回計算される。
多くのプログラム言語 (C, Java, Scheme, ML 等) の関数呼出しが値呼び。

必要呼び計算 call-by-need

値呼びと名前呼びの「良いとこどり」: 名前呼びと同様に計算するが、引数の値を 1 回計算したらその結果を覚えておいて、2 回目以降の計算で使う。
(fun x-> x+x+x) (power 2 10) の計算で、「2 の 10 乗」は 1 回計算される。
(fun x-> 0) (power 2 10) の計算で、「2 の 10 乗」は計算されない。
一部のプログラム言語 (Haskell 等) の関数呼出しは必要呼び。
cf. Java の Just-in-Time Compiler: 各クラスは、それが必要になるまで、compile しない。ただし 1 度 compile したら、2 回目以降の呼出しでは compiled code を使う。

マクロと関数

```
#define foo(x) (x+x)
int goo(int x) {
  return x+x;
}
int main () {
  int y = 0;
  y = foo(power(2,10));
  y = goo(power(2,10));
}
```

マクロ展開は、名前呼びと見なせる。関数呼び出しは、値呼びである。

プログラム例

対 (ペア) を作る操作を多数回繰返すプログラム:

```
let limit=10000000 in
  let rec f = (fun x ->
              if x=limit then "ok"
              else let _ = (x,x+1) in f (x+1))
  in f 0
```

これだけ繰返しても、エラーは起きず、計算が正常に終了する。

- ペア $(x, x+1)$ のデータは、スタック上に取られるのではない。
- ペア $(x, x+1)$ のデータを格納するためのメモリ領域は、このペアが不要になったら自動的に回収され、他のペアのために再利用される。

プログラム実行時のメモリ (再掲)

- Register
- Program Counter
- Code
- Environment Pointer (スタックを指す)
- Data:
  - Stack (スタック)
  - Heap (ヒープ)

ヒープ-1

ヒープ (heap): プログラム実行時に、データを格納しているメモリ領域で、スタック以外の領域。

- 構造を持つデータを格納する目的で使用される。
- そのデータを生成した関数が終了しても、そのデータが使われる可能性がある。
- ヒープに格納されるデータの例
  - ペア
  - 文字列
  - リスト
  - 関数 (正確には関数クロージャ)
  - その他、固定長のメモリ (通常は、32bit や 64bit) にはいりきらないデータすべて
- 式 `let y = (x,x+1) in ...` の計算
  - ペア $(x, x+1)$ はヒープに置かれる。
  - 局所変数 y の値は、そのペアへのポインタであり、スタックに置かれる。

ヒープ-2

先週の演習により、わかったこと (OCaml の長所)：

- ペアを作る時 (式 $(x, x+1)$ の評価)
  - 処理系は、ヒープの中のあいている場所を 1 つ確保し、このペアを格納する。その場所へのポインタを返す。
  - C 言語の malloc 関数は不要 (処理系が自動的に行う)。
- ペアを使う時: ポインタをたどる。
- ペアを使わなくなった時
  - どこからも参照されないペアを格納する場所は、いつか、回収される。
  - C 言語の free 関数は不要 (処理系が自動的に行なう)。
  - 不要になった瞬間に回収するか、ヒープが不足したときに一斉に回収するか、という選択肢がある。

ヒープ-3

- C 言語ではプログラマが行っているメモリ管理の仕事を、OCaml などの関数型言語では、処理系がやってくれる。
- 論理式、数式、プログラム、XML データなど、構造のあるデータ (特に、可変長のデータ) を扱う際には、上記の 2 機能があるプログラム言語を使うことは、ほぼ必須。
- ヒープは、後で回収することも考えると、単なる「配列」状ではなく、「リンクをもつリスト」状の構造を持つ。(長く使い続けていると、ヒープ全体が「使用領域」と「未使用領域」による「まだら模様」になっていく。)

ごみ集め (Garbage Collection)

- ヒープ領域において、使われなくなったデータに対応する領域を回収する (空き領域に連結する) 操作。
- 通常は、ヒープが足りなくなったときに、一斉に回収する。(その瞬間に処理系の動作が停止したように見える。cf. 昔の Lisp の主要なアプリはロボットの制御だった。。。)
- 処理速度が大事; 良いごみ集めアルゴリズム採用することが必須。
- 古くから研究されているが、現在でも、研究の対象。(高速性とともに、「絶対に間違いがあってはいけない」という意味で、正当性も非常に大事。)

プログラムの型

先週の課題から:

- (余力のある人のみ) MiniML 言語では、int や bool の定数のほかに「関数」をあらわすデータがある。このような関数の型は、どうなるであろうか？ また、MiniML 言語のプログラム中には、int や bool といった型名を書かないが、そんなことで大丈夫だろうか？ いったい OCaml 言語の処理系はどうなっているだろうか、考えてみなさい。

実行時 vs コンパイル時

式 (1+"abc") が「いつ」エラーになるか。

- MiniC や MiniML では、実行時に (動的に) エラーになる。
- C や OCaml では、コンパイル時に (静的に) エラーになる。

エラーは静的に見つかる方がよい。

- 早い段階でエラーが見つかる。
- (実行に時間がかかる場合)、速く見つかる。

型システム Type System

式 (1+"abc") の整合性に関するエラーを、静的に発見したい。一般に、(f(g(f(1),2)) + h(100,200)) のような式のエラーを、この式を実行せずに発見したい。

- 式の値を見るのではなく、式に「型」(type) を付け、その型のみを追うことによって整合性を検査する。
- 式の種類ごとに、どのような型が付くかを決めたものを「型システム」という。
- 現代のプログラム言語の多くが、強力な型システムを持っている。(例外は Prolog, Lisp/Scheme などごく一部)
- 型の整合性を検査するだけで、非常に多くのバグを発見することができる。

型検査

- C 言語では、変数の型は、その変数が使われるより前に宣言されている。
- さらに、他で定義された関数についても、その引数や返す値の型は事前に宣言する。
- ⇒型の整合性の検査は、変数や定数などのアトミックな式からはじめて、より大きな式の型が整合しているか検査する、という形式で行われる。

Java 言語なども同様。

型推論

- ML 言語では、(通常は) 変数や関数の型を宣言しない。
- さらに、他で定義された関数についても、その引数や返す値の型は宣言しない。
- ⇒型の整合性の検査は、「変数の型を推論する」という過程を含むため、それほど単純ではない。
- ML 言語では、「与えられたプログラム (らしき式) に対して、最も一般的な型を推論する」という型推論アルゴリズムが非常に重要。

```
(fun x→ x+1) :  int → int
(fun f→(fun x→ f(f(x+1)))) :  (int→ int)→(int→ int)
(fun x→ x) :  'a → 'a
(fun f → (fun x→ f (f x))) :  ('a → 'a) → ('a → 'a)
```

型推論の詳細は、「計算論理」(今年度は休講) の講義資料等を参照のこと。

やや細かい話

- 静的に (強く) 型付けされたプログラム言語: C, ML など。
- 動的に (弱く) 型付けされたプログラム言語: Lisp, Scheme, Ruby など。

Lisp, Scheme では、コンパイル時には型検査等は行われないので、($lambda(x)(+\ 1\ "abc")$) という関数を書いても、(コンパイラ時には) エラーにならない。この関数に引数を与えて実行したときに、はじめてエラーになる。ただし、型の概念がまったくないわけではない。(整数と文字列は実行時には区別される。)
Ruby でも、型エラーに相当するエラーも実行時に発生する。
考え方の相違: 信頼性 vs プログラムの書きやすさ・柔軟性

まとめ

- プログラム言語論らしい種々の概念の理解。
- たとえば，評価順序.
- たとえば，データ構造、ヒープ、ごみ集め。
- たとえば，型システム、型検査、型推論.

今日の授業の各々の話題をつきつめることが、プログラム言語論の 1 つの研究スタイル。

short quiz (3限用)

次の MiniML プログラムの型を推論しなさい。

- `fun f -> (fun x -> f (f (f x)))`
- `fun y -> (fun x -> if x then y else y + 1)`