プログラム言語論

亀山幸義

筑波大学 情報科学類

関数型言語の続き・制御構造

先週

関数型言語の導入
「組・対」のデータ構造とヒープとメモリ管理
関数型言語に対する抽象機械

関数型プログラミング言語

特徴: 単一代入，高階関数、再帰呼出しの多用，データ型の活用、etc.

- 単一代入⇒副作用の分離・明示 (変換や検証がやりやすい)
- 高階関数 (記述力・表現力の向上⇒プログラムの抽象化)
- データ型: たとえば「対」
- 再帰呼出し: 自然な関数定義

書きやすくなったり、検証しやすくなったといっても、処理がとても大変になっていないか？

- 単一代入⇒処理系にとっては、うれしい :)
- 高階関数⇒関数クロージャ
- データ型⇒ヒープ
- 再帰呼出し⇒末尾再帰

検証や変換がやりやすいとは？

関数型言語 (および論理学言語など) は、「宣言的プログラム」を書ける
検証がしやすい: 今日のソフトウェアでは「正しさ」はとてもとてもとても大事。(ハードウェア性能が向上し、ちょっとしたスピードを稼いでもしょうがない。書きやすく読みやすく、テストしやすく、正しさを保証しやすいプログラムを書きたい。)
⇒では、今日は、検証の話を …
しません。来週、出張なので、南出先生に Isabelle/HOL を使った検証の話をしていただきます。(Coq に興味がある人は、学生主導で勉強会をやっていますので参加してください。)

例

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

このプログラムを OCaml で実行しても、overflow のエラーを起こさない
理由: 「ペア $(x, x+1)$ は heap に格納される」
その理由だけでは説明がつかない。

スタックからペア $(x, x+1)$ へのポインタがあるなら、(スタックがまきもどされない以上) このペアも、「ごみ集め」の対象にならないのでは？

確かにそうだが、_ という特殊な変数は値を記憶しない (スタック上に変数の領域が取られない)。

C 言語での末尾再帰関数

```
#include <stdio.h>
int foo (int x) {
 int a[100];
 if (x > 0) return foo(x-1);
 else return 1;
}
int main () {
 printf ("%d\n", foo(1000000));
}
```

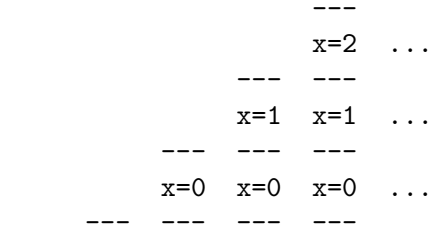

以前の例 (続き)

「対」が heap に取られてごみ集めの対象となってメモリ領域が回収されるとしても、stack に積まれる stack frame が、関数呼び出しごとに 1 つずつ多くなり、いつか stack overflow になるのでは？

その「からくり」を考えるのが、この章の目的。

関数呼出し時のスタック-1

```
let rec f x =
 if ... then ...
 else f (x+1)
in f 0
```



これでは、いつか、stack overflow になる。

関数呼出し時のスタック-2

```
let rec f x =
 if ... then ...
 else f (x+1)
in f 0
```

```
     ---   ---   ---
     x=0   x=1   x=2   ...
---   ---   ---   ---   ---
```


末尾呼び出し

関数 f の本体で、関数呼出し (g e) を行なうとき、(g e) の結果が、そのまま関数 f の結果となるとき、この関数呼出しを末尾呼出し (tail call) と言う。

末尾呼出しは、「それより後で関数 f の計算はない」ので、関数 f (の現在の呼出し) に対する stack frame は消してしまってよい。(while ループ等と同じ処理)

末尾呼出しでない例:

```
let rec f x = if x=0 then 1 else x * f (x-1)
let rec f x = if x=0 then 0 else f (x-1) + 0
```

末尾呼出しの例:

```
let rec f x = if x=0 then 1 else f x
let rec f x y = if x=0 then y else f (x-1) (x*y)
```

両方混在している例:

```
let rec f x = if x<2 then 1 else f(f(x-1))
```


関数型言語と末尾呼び出し

C 言語等における「ループ」を、関数型言語では、通常、「関数の再帰呼出し」で実現する。

再帰呼出しは、ループよりも表現力が高いが、その反面、実行効率がわるい (stack に stack frame をどんどん積む必要があるため。)

しかし、再帰呼出しが末尾再帰であれば、(また、処理系が末尾再帰最適化を組みこんでいれば)、コンパイラが、「ループ」として実現するので、実行効率はループと同等になる。

多くの関数型言語は、末尾再帰の最適化を組みこんでいる。(Scheme, SML/NJ, OCaml ..... ただし C 言語処理系は通常，末尾再帰の最適化はしない．)

本当に C の処理系は末尾再帰最適化をしないのだろうか？

C 言語の処理系

C 言語での末尾再帰関数:

```
#include <stdio.h>
int foo (int x) {
 int a[100];
 if (x > 0) return foo(x-1);
 else return 1;
}
int main () {
 printf ("%d\n", foo(1000000));
}
```

上記のテスト結果 (thanks to 鳥居君):

```
% gcc --version
gcc (Debian 4.7.2-5) 4.7.2
% gcc -o foo0 foo.c
% ./foo0
Segmentation fault
% gcc -o foo1 foo.c -O2
% ./foo1
```

教訓

先生 (皆さんより年寄り) の言うことを盲目的に信じてはいけません。
昔の知識のまま、確かめもせずに何か言っているかもしれません。
自分の手と頭を使って確認しましょう。

関連して … :

研究討論において、「インターネットにこう書いてあった」といってそれを根拠に議論を進める人がいます。
「インターネットの知識」すべてを集めたら、簡単に矛盾が導けます。
誰がどういう根拠をもってそう言っているのか、必ず確認しましょう。
とはいえ、根拠まで確認することは簡単ではありません。その場合でも、「誰が」そう言っているのか、を必ず付けておきましょう。(できれば、「XXX という論文誌にのった YYY さんの論文でそう言っている」とか「ZZZ さんがブログの記事でそう言っている」とかまで)

情報の精度には、非常に敏感になってほしいです。

制御構造 FORTRAN

```
10  IF (X .GT. 0.000001) GO TO 20
    X=-X
11  Y=X*X-SIN(Y)/(X+1)
    IF (X .LT. 0.000001) GO TO 50
20  IF (X*Y .LT. 0.000001) GO TO 30
    X=X-Y-Y
30  X=X+Y
    ...
50  CONTINUE
    X=A
    Y=B-A+C*C
    GO TO 11
```

スパゲッティ・コード (Mitchell, “Concepts in PL”, 2003 より)

構造化プログラミング

Dijkstra, “GO TO CONSIDERED HARMFUL” (1968)

go to 文を多用したプログラムは理解しづらい.
かわりに, より「構造的」な制御文を使うべき.
if-then-else, while, for, case ...

リストの要素の積を返す関数

```
open List;;  (* おまじない *)
let rec mult x =
  if x=[] then 1
  else (hd x) * (mult (tl x))
in mult [2; -3; 5]  ;;
```

hd はリストの先頭要素を取る関数.
tl はリストの先頭要素を除いた残り (リスト) を取る関数.

```
mult [2;-3;5] => -30
```

例外的な状況

mult 関数で、リストの要素に負の数があったら，(積ではなく) その要素を返すことにしたい.

```
let rec mult2 x =
  if x=[] then 1
  else if (hd x) < 0 then (hd x)
  else (hd x) * (mult2 (tl x))
in mult2 [2; -3; 5]  ;;
```

これではうまくいかない.

例外的な状況

```
let rec mult3 x =
  if x=[] then 1
  else if (hd x) < 0 then (hd x)
  else
     let res = mult3 (tl x) in
     if res < 0 then res
     else (hd x) * res
in mult3 [2; -3; 5]  ;;
```

再帰呼出しをするごとに,「例外的な状況が起きたかどうか」をチェックしないといけない.
⇒プログラムが読みづらくなるし，効率も悪い.

例外機構の利用

```
exception Negative of int ;; (例外の宣言)

try
  let rec mult4 x =
    if x=[] then 1
    else if (hd x) <= 0 then
        raise (Negative (hd x))
    else (hd x) * (mult4 (tl x))
  in
    mult4 [2; -3; 5]
with
  Negative n -> n ;;
```

raise (例外の発生) を実行すると，対応する try-with (例外の処理) まで一気にジャンプする.

例外機構: 別の例

map 関数の利用:

```
open List;;
let inc x = x +. 1.0;;
map inc [1.0; 2.0; 3.0];;
map sqrt [1.0; 2.0; 3.0];;
map sqrt [1.0; -2.0; 3.0];;
```

負の数があったら，例外を出したい.

```
exception Neg of float;;
let f r =
   if r < 0.0 then raise (Neg r)
   else sqrt r ;;
try
    map f [1.0; -2.0; 3.0]
with
    Neg r -> [r] ;;
```

例外機構

深い関数呼出しから一気に抜ける.

「内から外」の方向のみ可能.

例外の種類に名前を付け，発生した例外と一致する名前の「受け手」まで飛ぶ.

例外の発生と受け手は，動的に対応付けられる.

例外機構は動的束縛 (やや高度な内容)

```
exception E3;;
try
  let f x = raise E3 in
  let g x =
    try
       f 10
    with E3 -> 20
  in
    g 0
with E3 -> 30;;
```

上記の答は「20」である. (「30」ではない)

いろいろな言語の例外機構

例外機構〜大域脱出機構

- C: setjmp(), longjmp()
- C++: try-catch, throw
- Java: try-catch-finally, throw
- ML: try-with (または handle), raise

現代のプログラム言語では，例外機構を持つことはほとんど必須と考えられる。

継続 (continuation); 発展的な内容

実行時の「残りの計算」を表す概念.

```
let rec fact n =
  if n=0 then 1
  else n * (fact (n-1))
in fact 10
;;
let rec fact2 n k =
 if n=0 then k 1
 else
   fact2 (n-1) (fun x -> k (n*x))
in fact2 10 (fun x -> x)
```

fact2 は fact と同じ計算

```
fact2 10 (fun x -> x)
fact2 9 (fun x -> 10*x)
fact2 8 (fun x -> 10*9*x)
...
```

継続 (continuation); 発展的な内容

継続=「残りの計算」: `fact2 9 (fun x -> 10*x)` の第二引数
「残りの計算」を，関数で表すことにより，末尾呼び出しの形に変形できた．(「継続渡し方式」(Continuation Passing Style) のプログラム)

疑問: fib のように、関数呼び出しを 2 回以上おこなう関数は、末尾再帰にできないのか？
答え: すべての関数は、継続渡し方式 (continuation passing style) にすることができる。

```
let rec fib_cps n f =
 if n < 2 then (f 1)
 else fib_cps (n-1)
       (fun x1 ->
        fib_cps (n-2)
         (fun x2 -> f(x1+x2)))
in  fib_cps 5 (fun x -> x)
```

CPS にしたからといって効率がよくなるわけではないが、いろいろなメリットがある (Appel, "Compiling with Continuations".)

継続

例外機構の表現:

```
let mult5 x =
  let rec f x k =
    if x=[] then (k 1)
    else if (hd x < 0) then
       (hd x)
    else f (tl x) (fun v -> k ((hd x)*v))
in f x (fun v -> v)

mult5 [1;5;-3]
=> f [1;5;-3] (fun v->v)
=> f [5;-3] (fun v->1*v)
=> f [-3] (fun v->5*1*v)
=> -3
```


継続の利用

miniC 言語のミステリー:

- miniC 言語には、例外機構等のコントロール抽象の仕組みはない。
- miniC 言語インタープリタは、継続渡し方式で実装。

C/miniC 言語の return 式:

```
int f (int x) {
  ...
  for (i=0; i<100; i++} {
    ...
    if (...) return 1;
  }
  ...
  y = x ? (return 10) : 5;
  ...
}
```

そのほかにも、continue など。

継続

継続渡し方式: 継続を関数で表現したプログラミングスタイル．
一方，継続を直接扱うことにより、継続渡し方式と同等のプログラムを簡潔に書けるようにした言語もある．

- call/cc (call-with-current-continuation; Scheme, SML/NJ, Ruby)
  call/cc を使うと，例外機構以外に，バックトラックやコルーチンなど種々の制御を表現できる．

まとめ

構造化プログラミングとプログラムの制御構造
例外機構
(継続)

問題

以下の関数の各関数呼び出しは末尾再帰かどうか判定せよ。(g は既に定義されている関数とする)

- `let rec f x = g (x + 1)` (答: g は末尾再帰)
- `let rec f x = f (g (x + 1))` (答: f は末尾再帰,g は NO)
- `let rec f x = g (f (x + 1))` (答: g は末尾再帰,f は NO)
- `let rec f x = f g` (答: f は末尾再帰, g は関数として呼び出されていない)

問題

末尾再帰や、miniC 処理系の内部で使っている継続のように、「プログラマには見えない」制御構造は、(処理速度が速くなるという) メリットがあるが、例外や call/cc など「プログラマに見える」制御構造が役に立つ理由を述べよ。また、このような制御構造は、乱用すると、かえってまずいことになる。(GO TO 文 (無制限のジャンプ命令の乱用など) どうしたらよいだろうか？

(来週までの発展課題) Scheme の call/cc や Ruby の yield などの制御機構、あるいは、「継続渡し方式」について調べ，0.5 ページ程度で、その特徴を述べなさい.