プログラム言語論

亀山幸義

筑波大学 情報科学類

No. 5 (ヒープと関数型言語その 2)

「対」のデータ型の利用 (1)

Fibonacci 関数の素朴な定義:

```
let rec fib n =
  if n < 2 then 1
  else (fib (n-2)) + (fib (n-1))
in fib 5
==> 8
```

Fibonacci 関数の改善した定義:

```
let rec fib n =
  if n = 0 then (1,1)
  else
    let p = fib (n - 1) in
      (snd p, fst p + snd p)
in fst (fib 5)
==> 8
```


「対」のデータ型の利用 (2)

最大公約数を求める関数:

```
let rec gcd m n =
  if m = n then m
  else if m > n then
    gcd (m-n) n
  else
    gcd (n-m) m
in gcd 100 35
==> 5
```


「対」のデータ型の処理 (1)

OCaml プログラム

```
let add p q =
   (fst p + fst q,
    snd p + snd q)
in let v = (10,20)
in add v v
==> (20,40)
```

C プログラム: NG!

```
int* add (int *p, int *q) {
  int r[2];
  *r = *p + *q;
  *(r+1) = *(p+1) + *(q+1);
  return r;
}
int main () {
  int v[2];
  int *q;
  v[0] = 10; v[1] = 20;
  q = add(v, v);
  printf ("(%d,%d)\n", *q, *(q+1));
}
```

「対」のデータ型の処理 (2)

OCaml プログラム

```
let add p q =
   (fst p + fst q,
    snd p + snd q)
in let v = (10,20)
in add v v
==> (20,40)
```

C プログラム

```
int* add (int *p, int *q) {
  int* r =
   (int*)malloc(sizeof(int)*2);
  *r = *p + *q;
  *(r+1) = *(p+1) + *(q+1);
  return r;
}
int main () {
  int v[2];
  int *q;
  v[0] = 10; v[1] = 20;
  q = add(v, v);
  printf ("(%d,%d)\n", *q, *(q+1));
}
```

malloc for memory allocation.

データ構造

- MiniML 言語の「対 (ペア)」データ型を使えば、様々なデータ構造を表現できる。C 言語で同様なことをするためには、構造体 (struct 型) などを使う。
  - MiniML 言語では、一度作ったペア (のためのメモリ) は、どうなるだろうか？ ペア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 言語処理系において「ペア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。

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

ただし _ というのは、「無名の変数」のこと (後で使わない変数)。

- 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 (heap)

- 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ものを格納するためのメモリ。(永続的なデータ).
- ヒープに格納されるデータ (例)
  - 対 (ペア) など構造を持つデータ
  - 文字列
  - リスト
  - 関数 (正確には関数クロージャ)
  - オブジェクト指向言語でのオブジェクト
  - 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
- x と y の対の生成の処理
  - 対のためのメモリをヒープから取得する.
  - その場所に (x, y) を書きこむ.
  - それへのポインタ (x へのポインタ) を返す.

ヒープ

- C 言語ではプログラマがメモリ管理を行う。(ヒープ上のメモリ確保 malloc(), メモリ解放 free())
- OCaml などの言語では、処理系が自動的にやってくれる。(対を実行すればメモリが確保され，対を利用することがなくなれば，いつの間にかメモリが解放される．)
- 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。
- 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、現在でも更なる改善のための研究がなされている。(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。)

データ型の意義

C 言語でのデータ型

- 比較的少数 (配列，ポインタ，struct 型など).
- メモリ確保と解放はプログラマの責任.
- 低レベルの詳細なコントロールが可能.

OCaml 等でのデータ型

- 非常に豊富なデータ型を用意 (関数，直積，レコード，バリアント，帰納的データ型など)
- メモリ確保と解放は自動.
- 「ごみ集め」の仕組みが必要.

※ C 言語は自前でメモリ管理をしたいプログラマ向け，OCaml 等はメモリ管理をシステム (処理系) に委ねたい人向け.

まとめ

- データ構造，永続的なデータ，ヒープ
- メモリ管理

関数をデータとして扱う-1

プログラム言語における「第一級のもの (first-class citizen)」

- 通常のデータと同様に扱われるもの．変数の値になったり，関数の引数や返り値になれるもの．
- C 言語: 整数などのほか，ポインタが first-class.
- Java: 整数などのほか，オブジェクトが first-class.
- OCaml, Haskell など: 整数などのほか，関数が first-class.
- Scheme: 整数やシンボルのほか，S 式が first-class.

関数をデータとして扱う-2

処理系内部では,「関数を表す式を計算した結果の値」が必要.
動的束縛の場合 (昔の Lisp, 今の emacs lisp)

- $\lambda x.e$ を計算した結果は，$\lambda x.e$ そのものでよい.

静的束縛の場合 (ほとんどの関数型言語)

- $\lambda x.e$ を計算した結果は，$\lambda x.e$ そのものではいけない.
- C 言語の処理で，静的束縛では，access link が必要だった.

関数をデータとして扱う-3

例題:

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

上記プログラムの処理 (誤ったバージョン):

frame4: y=3
frame3: x=2
frame2: f=fun y->x+y
frame1: x=1
global:

返り値: 5

ここで frame4 における access link が指すのは，frame3 でなく，frame1 でなければならない.

関数クロージャ(関数閉包，function closure)

- 静的束縛の関数型言語で，実行時に用いられる.「関数を計算した結果 (値)」を表す.
- 関数の定義と，環境 (スタックフレームへのポインタ，あるいは，変数ごとにその値を決めるもの) をセットにしたもの: (fun x -> e, σ). ここで，σ は，環境 (へのポインタ) で，将来この関数の本体 e が実行されるときの access link となる.
- 要するに，この関数を作ったときの環境を保存しておく，ということ.

参考: closure とは，閉じたもののこと．関数 `fun x->x+y` は，変数 y が自由変数になっているので，その値とセットにして，はじめて「閉じる」ことができる．(自由変数が 1 つもない式のことをラムダ計算では，closed term と言う．)

関数クロージャを用いた処理 (2)

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

上記プログラムの処理 (正しいバージョン):

- 式 (fun y→ x+y) の値は関数クロージャ clo(fun y→ x+y, x=1 のフレームへのリンク)
- (f 3) の計算では、関数クロージャに保存されていたリンクが，access link となる.

frame4: y=3, access link ⇒ frame1	返り値: 4
frame3: x=2	返り値: 4
frame2: f=clo(fun y->x+y,frame1)	返り値: 4
frame1: x=1	返り値: 4
global:	

関数クロージャを用いた処理 (3)

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

frame4: y=3, access link ⇒ frame1	返り値: clo(...)
frame3: x=2	返り値: clo(...)
frame2: f=clo(fun y->x+y,frame1)	返り値: clo(...)
frame1: x=1frame1': y=3, access link ⇒ frame1	返り値: clo(...)
global:	

NG! frame1 はもはや存在しない。

関数クロージャはどこにあるか？

```
let f =
  let foo x =
    fun y-> x+y
  in
    foo 10
in
  f 20
```

- 関数クロージャは，C 言語の関数と違い，プログラム実行時に (動的に) 生成される.
- 関数クロージャは，スタックに積まれるのではない．(それを生成した関数呼出しが終わった後も行き残る．下記プログラムを参照)
- 関数クロージャは，ヒープに置かれる.

C 言語の「高階関数」?

質問. C 言語でも、「関数へのポインタを引数とした関数」や「関数へのポインタを返す関数」は書ける。では C 言語も関数型言語か?

- No. C 言語では、「関数を生成する」ことは (通常は) できない。
- 関数型言語では、関数を動的に生成して、(計算結果として) 返すことができる。
  `let fun f x = (fun y -> x + y)`
- (参考) オブジェクト指向言語では、オブジェクトを動的に生成して、(計算結果として) 返すことができる。

今日のまとめ

- 関数プログラミング: 単一代入，再帰呼出し，高階関数，データ型の活用
- 単一代入⇒副作用の分離・明示
- 高階関数と静的束縛⇒関数クロージャ
- データ型⇒ヒープ
- 再帰呼出し⇒末尾再帰

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。

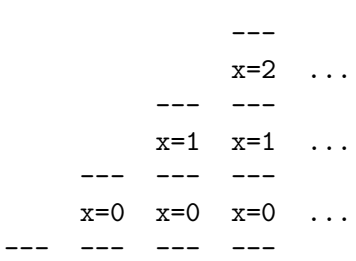
- スタックからペア $(x, x+1)$ へのポインタがあるなら、(スタックがまきもどされない以上) このペアも、「ごみ集め」の対象にならないのでは?
- 確かにそうだが、_ という特殊な変数は値を記憶しない (スタック上に変数の領域が取られない)。

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

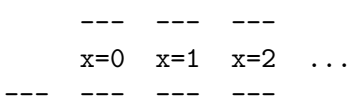
```
let rec f x =
 if ... then ...
 else f (x+1)
in f 0
```



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
```



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)
```


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．)

まとめ

- 関数プログラミング: 単一代入，再帰呼出し，高階関数，データ型の活用
- 単一代入⇒副作用の分離・明示
- 高階関数と静的束縛⇒関数クロージャ
- データ型⇒ヒープ
- 再帰呼出し⇒末尾再帰