プログラム言語論-第 7 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 6 月 1 日

概要

前回まで
- プログラムの構文
- プログラムの意味と評価 (処理) 方法

今回
- 型システム (続き)
- 抽象データ型
- オブジェクト指向

先々週の補足

```
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
```

「このようなプログラムを実行しても、stack overflow を起こさず実行が終了する」理由:
- ペア $(x, x+1)$ (生成されては捨てられる) は stack 上に格納されない。
- 「stack 上の stack frame が、関数呼び出しごとに 1 つずつ増えていく」ことはない。

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
```

```
                ---
                x=2  ...
          ---   ---
          x=1   x=1  ...
     ---  ---   ---
     x=0  x=0   x=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)`
末尾呼出しの例:
`let rec f x y = if x=0 then y else f (x-1) (x*y)`

型システムの設計

現代のプログラム言語の設計において、型システムをどう設計するかは極めて重要。

型 (Type) とは?

「型」は「データの集合」の一種ではあるが、データの集合がすべて型になるとは限らない。

- コンピュータ (ハードウェア) で扱うことのできるデータの種類のこと。
- 同じ演算たちが適用できるデータの集まり。

型システム： どのようなプログラムにどのような型がつくか、定めるための体系。通常は、プログラムの構文の定義と一緒に与えられる。

具体的に‥

基本型 (atomic type)

- 例: `int`, `bool`, `string`
- 基本型の要素、基本型に対する演算などはプログラム言語ごとにあらかじめ与えられる。

複合的な型: 既にある型と型構成子 (type constructor) を使って構成。

- 例
  - 配列型: `int` の配列型、`string` の配列型
  - ポインタ型： 「`int` のポインタ型」
  - 構造体型、共用型、関数型 etc.
- 複合的な型の要素に対する演算は、型構成子ごとに与えられる。
  - 例. ポインタ型のデータを作る。`p=&x;`
  - 例. ポインタ型のデータを使う。`x=*p;`

再帰的に適用可能な型構成子が 1 つでもあると、型は無限個存在する。

型構成子の例

- C 言語: 構造体 (strucut)、共用 (union)、ポインタ、(関数) など。
- ML 言語: 直積、レコード (record)、バリアント (variant), 参照、関数、リスト、再帰的な型など。

再帰的な型の例 (OCaml での 2 分木):

```
type binary_tree =
  Leaf of int
| Node of binary_tree * binary_tree

Node(Leaf(1), Node(Leaf(2),Leaf(3)))
```

cf. BNF での構文定義: $e ::= 0 \mid 1 \mid (e + e)$.

型があることの利点 (メリット)

Benjamin Pierce, "Types and Programming Languages", MIT Press から引用 (一部)。

- Documentation
- Error Detection
- Efficiency
- Abstraction, Language Safety
- etc.

Types for Documentation-1

```
goo (s) { return *s; }
foo (p, f()) { return *(f(&p)); }
main () {
  x = 10;
  printf ("%d\n", foo(&x,&goo));
}

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


Types for Documentation-2

```
int * goo (int **s) { return *s; }
int foo (int *p, int *f(int **)) {
  return *(f(&p)); }
int main () {
  int x = 10;
  printf ("%d\n", foo(&x,&goo));
}

fun f -> (fun x -> f (f (f (x+1))))
  : (int -> int) -> (int -> int)  ;;
fun y -> (fun x -> if x then y else y + 1)
  : int -> (bool -> int) ;;
```


Types for Error Detection-1

```
int goo (int x) { return x+1;}
int foo (int x, int *p, int f(int)) {
  return (x + f(*p));
}
int main () {
  int x = 10; int y = 20;
  printf ("%d\n", foo(&x,x,&goo));
  printf ("%d\n", foo(x,&x,&foo));
  printf ("%d\n", foo(&x,&goo));
}
```


Types for Error Detection-2

```
# fun y -> (fun x -> if x then x else y + 1) ;;
  Characters 37-42:
  fun y -> (fun x -> if x then x else y + 1) ;;
                                       ^^^^^
This expression has type int but is here used with type bool
```

ソフトウェア作りの鉄則

- エラーは早い段階 (上流) で見つければ見つけるほど、良い (修正のための手間が少ない)。
- 実行時 (動的) に見つけるよりも、コンパイル時 (静的) に見つける方がよい。
- たまたまその部分を実行したときに見つかるよりも、実行するかどうかにかかわらず見つけた方がよい。

Types for Efficiency-1

MiniC 言語の (OCaml で書いた) インタプリタ (簡略版):

```
| Plus (e1,e2) ->
   let v1 = (eval e1 env) in
   let v2 = (eval e2 env) in
     match (v1,v2) with
       | (IntValue(i1), IntValue(i2)) -> IntValue (i1+i2)
       | _ -> failwith "non-int value in arithmetic"
```

($e1 + e2$) という式の評価:

- $e1$ を計算して、結果を $v1$ とする。
- $e2$ を計算して、結果を $v2$ とする。
- $v1$, $v2$ がともに整数値であれば、加算をおこない、その結果を整数値として返す。

実行時に、データの型をチェックしている。
このようなチェックをしない場合、たとえば、「単なる整数値を、ポインタとして使う (メモリ領域上のとんでもない場所を指してしまうかもしれない)」という Segmentation Fault を起こす危険

Types for Efficiency-2

C 言語や ML 言語の処理系: MiniC, MiniML とは異なり、コンパイル時に型検査、型推論を行なう。
($e1 + e2$) という式の評価:

- $e1$ を計算して、結果を $v1$ とする。
- $e2$ を計算して、結果を $v2$ とする。
- $v1$, $v2$ はともに整数値であることはチェック済みなので、そのまま加算をおこない、その結果を返す。

コンパイル時に、整数型であることをチェックしている。
補足: C 言語の + は、int 型だけでなく、float 型にも使われる (overloading) ので、コンパイル時に、int の加算か、float の加算かも判定する。また、必要ならば、int 型のデータを float 型のデータに変換するコードを挿入する。

型があることの利点 (再掲)

Benjamin Pierce, "Types and Programming Languages", MIT Press から引用 (一部)。

- Documentation (文書⇒理解しやすさ)
- Error Detection (エラーの早期発見)
- Efficiency (実行効率の良さ)
- Abstraction (抽象化), Language Safety(言語の安全性)
- etc.

静的な型システムの利点

Type Soundness (型に関する健全性):

- コンパイル時に型が整合していれば、(型検査あるいは型推論に合格すれば)、実行時には型に関するエラーを起こさない。

この性質が満たされるプログラム言語では、実行時に型のチェックをする必要がない。
この性質自体は、「証明」する必要がある。(現実のプログラム言語では、Standard ML などごく一部の言語に対してしか証明されていない。)

型検査

- 与えられたプログラム p に含まれる変数や関数の型が全て宣言されているとき (型が既知のとき)、p の型が整合してているかどうかを検査すること。
- 型検査の答えは YES or NO である。(判定問題)
- C 言語や Java 言語は、基本的には、すべての変数、関数 (あるいはクラス) の型が宣言されているため、コンパイル時に型検査を行う。

型推論

- 与えられたプログラム p に含まれる変数や関数の型がわからないものもあるとき (型が未知のとき)、p に型がつくかどうかを検査し、また、型がつく場合はその型が何であるかを推論すること。
- 型推論の答えは (YES と p の型) or NO である。(正確には、YES の場合は、p に含まれる変数等の型も推論される。)
- ML 言語は、変数の型で宣言されていないものがあるため、コンパイル時に型推論を行う。

型検査と型推論:

- 一般に、型検査問題より、型推論問題の方が困難。
- ML 言語では、どのような「プログラムらしきもの」を与えられても、型推論できるかどうかが有限時間で判定できる。
- 型システムを複雑にしていくと、型推論や型検査が有限時間内に判定できなくなることがある。

動的な型システム

静的な型システムを持たない言語でも、動的な型システムを持つことが多い。

- 型の整合性という概念はある。(整数と文字列を加えるとエラーなど)
- ただし、コンパイル時に全ての整合性をチェックするのではなく、実行時に (も) 型検査を行なう。
- 単純な効率やプログラムの理解のしやすさの観点からは、静的型システムに比べて不利。
- 静的な型システムで記述できないような、柔軟なプログラミングができる可能性がある。(実行結果によって、式の型が変化するプログラム。) cf. Ruby

多相型 (Polymorphism)

```
void swap (int *p, int *q) {
  int *r;
  r = p; p = q; q = r;
}
```

swap 関数は (int *) 型だけでなく、どんな型でも使える。

```
void swap (T *p, T *q) {
  T *r;
  r = p; p = q; q = r;
}
for any T.

# let swap (x,y) = (y,x) ;;
  val swap : 'a * 'b -> 'b * 'a = <fun>
```

多相型=「任意の型」を含む型。

型システムの設計

- プログラム言語の設計において、型システムの設計は極めて重要。
- プログラム言語の型システムの設計は、自由度が大きい。(個性があらわれる。)

Short Quiz

以下のプログラムを C 言語処理系と、miniC 言語処理系にかけると、それぞれ、どのような結果になるか。

```
int foo () {
  int y;
  y = true;
  return y;
}
int main () {
  int x;
  x = 10;
}
```


抽象化 abstraction

抽象化とは?

- ということを抽象的に 論じるのはやめて、「型による抽象化」の具体例を見ることにしよう。

抽象データ型-1

- 今までのデータ型=具体データ型 (Concrete Data Type)
  - 新しく定義したいデータ型をどう構成したいかを、具体的に (型構成子を使って) 記述した。
  - そのデータ型が、具体的にどう実現されているかがわかっている。
- 抽象データ型 (Abstract Data Type)
  - データ型の具体的な構成方法 (実現方法) は定めない。
  - データ型がどう使われるかだけを定める。
  - つまり、データの実装ではなく、データの仕様。

抽象データ型-2

stack: スタック (その要素は整数) をあらわす抽象データ型

- stack 型を操作する関数とその (具体) データ型。
  - emptystack: stack
  - push: int* stack→ stack
  - pop: stack→ int* stack
  - isempty: stack→ bool
- これらの関数が満たすべき性質。
  - isempty(emptystack)=true
  - isempty(push(x,s))=false
  - pop(push(x,s))=(x,s)

抽象データ型-3

stack の実装: 前ページの型と性質を満たす限り、どんな実装でもよい。

- stack を配列で実装。配列の第 0 要素が、スタックの底。
- stack を配列で実装。配列の最終要素が、スタックの底。
- stack をリストで実装。
- stack を配列で実装。ただし、メモリが不足すれば malloc 関数でメモリを確保。

stack の利用: 前ページの関数を使う限り、どんな使用法でもよい。

- 前ページの関数以外を使って、stack にアクセスしてはいけない。
- たとえば、stack の底のアドレスを得て、スタックの n 番目の要素にアクセスする (stack inspection) のは禁止。

モジュール (module)

ソフトウェアの構成単位 (部品)
プログラムにおける、「何らかの関心事についてのまとまり」
インタフェースと実装から構成される。

- インタフェース (interface)
  - このモジュールを使うための仕様を定めたもの。
  - 通常は、モジュールを使うための関数の名前と型、など。
  - stack の場合、push,pop,emptystack,isempty 関数とその型。
- 実装 (implementation)
  - インタフェースが定められた関数等を実現するプログラム。
  - インタフェースに従う限り、どのような実装でもよい。
  - 実装のみに現れる関数は、外からは使えない。

モジュラリティ (modularity)

モジュラリティの高いプログラム

- 関心事ごとのまとまり (モジュールなど) が、それぞれ独立性が高いこと。
- 独立性=インタフェースが実装と分離されていること。
- 「モジュラリティの高さ」についての定量的な尺度はない。

モジュラリティの高いプログラムの利点

- 各モジュールごとに独立に実装しやすい。
- プログラムの保守性・再利用性がよくなる。

ここまでのまとめ

モジュラリティ＝大規模ソフトウェア作成における重要ポイントの１つ：

- 情報の隠蔽 or インタフェースと実装の分離。
- モジュール: モジュラープログラミングに対するプログラミング言語からのサポート。

Short Quiz

大学図書館の図書管理ソフトウェアを modular に設計したい。以下のような module 設計のうち、どれを選択すると modularity がもっとも高くなりそうか、簡単な理由をつけて答えよ。

- 学群ごとに１つの module とする (A 学群の図書管理 module, B 学群の図書管理 module etc.)。
- 図書分類ごとに１つの module とする (数学分野の図書管理 module, 計算機科学分野の図書管理 module etc.)。
- 洋書か和書か、シリーズ本か単行本か、中央図書館内蔵図書か学群管理図書か、貸し出し可能かどうか、などの図書の性質で分類し、それぞれを１つの module とする。
- 図書が出版された年ごとに１つの module とする。