函数式编程原理

Lecture 3

参考CMU 15-150 Lecture03-05-slides.pdf

上节课内容回顾

- 函数式语言的特点
- ML编程基础

  基础类型、元组(*)、表(list, ::, @)、函数(->)
- 表达式：表达式求值的结果为一个值(if it terminates)
- 声明：声明产生名字和值的绑定
- 模式：模式匹配成功将产生值的绑定

本次课主要内容

- 代码说明　　　(Specifications)
- 程序的正确性验证：归纳法
- 程序的有效性验证：

  基于归纳法的时间复杂度分析

代码说明(Specifications)

- 函数定义前，用注释信息描述函数功能，形如(* comments*)：
  - 函数名字和类型 (类型定义)
  - REQUIRES：参数说明 (明确参数范围)
  - ENSURES：函数在有效参数范围内的执行结果 (函数功能)

```
fun eval ([ ]:int list) : int = 0
| eval (d::L) = d + 10 * (eval L);

(* eval : int list -> int  *)

(* REQUIRES:   *)
(*    every integer in L is a decimal digit    *)

(* ENSURES:                              *)
(*    eval(L) evaluates to a non-negative integer *)
```

```
fun decimal (n:int) : int list =
  if n<10  then [n]
           else (n mod 10) :: decimal (n div 10);

(* decimal : int -> int list  *)

(* REQUIRES: n >= 0     *)

(* ENSURES:                                   *)
(*  decimal(n) evaluates to a list L of decimal digits *)
(*      such that eval(L) = n *)
```

代码说明的作用

- 确保函数行为的正确性
- 确保在允许的参数范围内能得到正确的结果

  - 如何证明函数能按说明的内容正确的执行？

    ——程序正确性证明

- 基于等式或推导的方式进行数学证明
- 程序结构作为指导：

程序语法	推导
if-then-else	布尔分析
case p of …	case分析
fun f(x) = …f…	归纳法

归纳法(Induction)

- 常见的几种归纳法：
  - 简单归纳法 (Simple (mathematical) induction)
  - 完全归纳法 (Complete (strong) induction)
  - 结构归纳法 (Structural induction)
  - 良基归纳法 (Well-founded induction)

简单归纳法(Simple (mathematical) induction)

证明对所有非负整数n，P(n)都成立

- 基本情形(base case)：证明P(0)成立
- 推导过程(inductive step)：

    假设对任意k(≥0)，P(k)成立，则P(k+1)也成立

用简单归纳法证明

```
fun f(x:int):int =
    if x=0 then 1 else f(x-1) + 1
(* REQUIRES x ≥0 *)
(* ENSURES f(x) = x+1 *)
```

试证明：对所有整数 x，当x≥0时, f(x) = x+1

用简单归纳法证明

```
(* eval : int list -> int        *)
(* REQUIRES:                     *)
(*  every integer in L is a decimal digit   *)
(* ENSURES:                                  *)
(*    eval(L) evaluates to a non-negative integer *)
```

```
fun eval ([ ]:int list) : int = 0
  | eval (d::L) = d + 10 * (eval L);
```

试证明：对所有值 L:int list，
存在一个整数 n，使eval L =>* n

(size = length of argument list, decreases by 1)

如何证明？
为什么不能用？

```
fun decimal (n:int) : int list =
  if n<10
    then [n]
    else (n mod 10) :: decimal (n div 10)
```

```
(* decimal : int -> int list          *)
(* REQUIRES: n >= 0                   *)
(* ENSURES:                            *)
(*  decimal(n) evaluates to a list L of decimal digits *)
(*      such that eval(L) = n *)
```

简单归纳法的适用范围

- **适用于涉及自然数的递归函数**
  - 参数为非负整数
  - f(x)的递归调用形如f(y),且size(y)=size(x)-1

完全归纳法(Complete (strong) induction)

证明对所有非负整数n，P(n)都成立

- 将P(k)简化为k个子问题： P(0), P(1), … , P(k-1)，且它们均成立时，可以利用{P(0), P(1), … , P(k-1)}推导出P(k)也成立
  - 如：P(0)成立
    P(1)可由P(0)推导出来
    P(2)可由P(0)， P(1)推导出来
    P(3)可由P(0)， P(1)， P(2)推导出来
    ……
    P(k)可由P(0), P(1), … , P(k-1)推导出来

完全归纳法的适用范围

- **适用于涉及自然数的递归函数**
  - 参数为非负整数
  - f(x)的递归调用形如f(y),且size(y)<size(x)

用完全归纳法证明

```
(* decimal : int -> int list  *)
(* REQUIRES: n >= 0    *)
(* ENSURES:                                   *)
(*    decimal(n) evaluates to a list L of decimal digits, *)
(*       such that eval(L) = n *)
```

```
fun decimal (n:int) : int list =
  if n<10 then [n]
           else (n mod 10) :: decimal (n div 10)
```

试证明：对所有值 n:int (n≥0)，eval(decimal n) = n

```
fun eval ([ ]:int list) : int = 0
  | eval (d::L) = d + 10 * (eval L);
```

结构归纳法(Structural induction)

完全归纳法在其他数据类型上的推广

- 基本情形： P([ ])
- 归纳步骤：对具有类型t的所有元素y和t list类型的数ys，都有P(ys)成立时， P(y::ys)成立

  $\forall\, i < k$, P(i)成立的条件下有P(k)

适用于涉及表和树的递归函数

良基归纳法(Well-founded induction)

关系≺是良基的：

不存在无穷降序链：…≺Xn≺…≺X2≺X1,

对所有y’ ≺y，有P(y)，则P(y’)成立

可以处理广泛的可终止计算问题

近似运行时间

- 反映基于大批量数据的程序运行性能
  - 假设基本操作为常量执行时间(Assume basic ops take *constant* time)
  - 用O记号表示算法的时间性能(Give big-O classification)

- f(n) 为O(g(n))：
  - 存在整数N和c，满足

    $\forall n \geq N,\ f(n) \leq c\ g(n)$

为什么叫“近似”？

- 加法中的常数加不考虑(Additive constants don’t matter)

  $n^5+1000000$ is $O(n^5)$
- 乘法中的常数乘不考虑(Multiplicative constants don’t matter)

  $1000000n^5$ is $O(n^5)$
- g(n)尽可能精确(Be as accurate as you can)

时间复杂度 (big-O)

- 时间复杂度也称渐近时间复杂度，表示为T(n)=O(f(n))，其中f(n)为算法中频度最大的语句频度。
  - 程序的执行时间依赖于具体的软硬件环境，不能用执行时间的长短来衡量算法的时间复杂度，而要通过基本语句执行次数的数量级来衡量。
  - 算法中语句的频度与问题规模有关，一般考虑问题规模趋向无穷大时，该程序时间复杂度的数量级。
  - 一般仅考虑在最坏情况下的时间复杂度，以保证算法的运行时间不会比它更长。
    - 给定函数f, g : int -> int，f的时间复杂度为O(g)表示：
      存在常量c和整数N，对所有n ≥ N, 有|f(n)| ≤ c * |g(n)|.

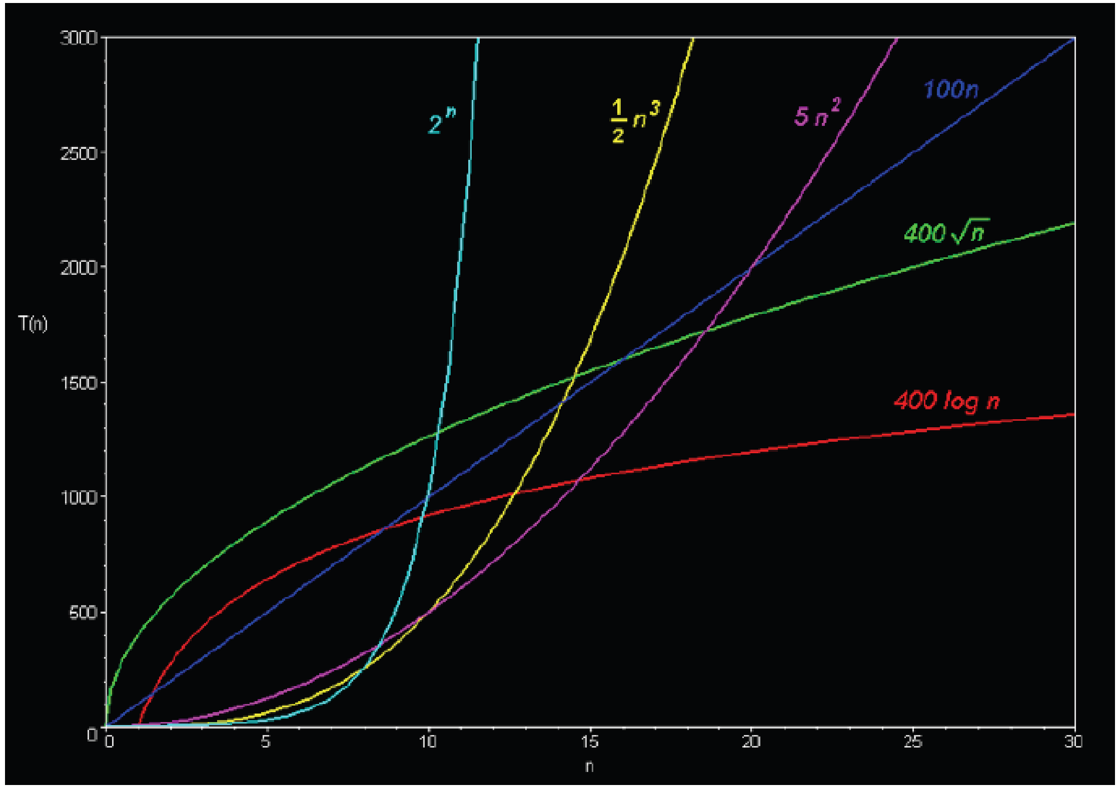
近似运行时间分析

- 求解步骤：
  1. 找出算法中的基本语句：算法中执行次数最多的那条语句就是基本语句，通常是最内层循环的循环体
  2. 计算基本语句的执行次数的数量级：忽略所有低次幂和最高次幂的系数，保证基本语句执行次数的函数中的最高次幂正确
  3. 用O记号表示算法的时间性能：将基本语句执行次数的数量级放入O记号中。

近似运行时间分析

```
for (i=1; i<=n; i++)
        x++;

for (i=1; i<=n; i++)
        for (j=1; j<=n; j++)
          x++;
```



$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n\log n) < O(n^2) < O(n^3) < \ldots < O(2^n) < O(n!)$

对数时间 (logarithmic)　平方根时间 (square root)　线性时间 (linear)　Quadratic　Cubic　多项式时间(polynomial)

指数时间 (exponential)

递推分析实例

```
fun exp (n:int):int =
  if n=0 then 1 else 2 * exp (n-1);
```

M：(fn n => if n=0 then 1 else 2 * exp(n-1))

exp 4 $\Rightarrow^{(1)}$ M 4 $\Rightarrow^{(4)}$ 2 * (M 3)

$\Rightarrow^{(4)}$ 2 * (2 * (M 2))

$\Rightarrow^{(4)}$ 2 * (2 * (2 * (M 1)))

$\Rightarrow^{(4)}$ 2 * (2 * (2 * (2 * (M 0))))

$\Rightarrow^{(2)}$ 2 * (2 * (2 * (2 * 1)))

$\Rightarrow^{(4)}$ 16

M 4 => if 4=0 then ...
=> 2 * exp (4-1)
=> 2 * M (4-1)
=> 2 * M 3

由此可推出：

for all n≥0, exp n $\Rightarrow^{(5n+3)}$ 2^n

近似运行时间为：O(n)

递推分析(recurrences)

- 递归函数的定义给出了程序的递推关系，执行情况用*work*表示
  (A *recursive function definition* suggests a ***recurrence relation*** for *work*, or *runtime*)
  - W(n)表示参数规模为n的程序的执行情况work (W(n) = work on inputs of size n)

- W(n)的推导：
  - Base cases：评估基本操作的执行 (Estimates the number of basic operations)
  - Inductive case：
    - 用归纳法得到W(n)的表达式 (Try to find a *closed form* solution for W(n) using *induction*)
    - 对表达式进行简化，得到一个具有相同渐近属性的表达式(Find solution to a *simplified* recurrence with the same asymptotic properties)

注意：推导过程要规范(Appeal to table of standard recurrences)

程序执行情况W(n)分析

```
fun exp (n:int):int =
   if n=0 then 1 else 2 * exp (n-1);
```

用$W_{exp}(n)$表示程序 exp(n)的执行时间

$$W_{exp}(0) = c_0$$

$$W_{exp}(n) = c_0 + n\,c_1 \quad (n>0)$$

程序执行时间随n值的增加线性增长

For all $n \geq 0$, $W_{exp}(n) \leq c\,n \rightarrow O(n)$

能否缩短程序运行时间、

提高效率？

fastexp

```
fun square(x:int):int = x * x
fun fastexp (n:int):int =
    if n=0 then 1 else
    if n mod 2 = 0 then square(fastexp (n div 2))
                   else 2 * fastexp(n-1)
```

$W_{fastexp}(n)$如何推导？

$W_{fastexp}(0)=c_0$

N为偶数： $W_{fastexp}(n)= W_{fastexp}(n \text{ div } 2) + c_1$

N为奇数： $W_{fastexp}(n)= W_{fastexp}(n-1) + c_2$

$W_{fastexp}(n)$： $O(\log n)$

```
fastexp 4 = square(fastexp 2)
          = square(square (fastexp 1))
          = square(square (2 * fastexp 0))
          = square(square (2 * 1))
          = square 4 =16
```

能否再快一点？

pow

```
fun pow (n:int):int =
    case n of
          0 => 1
        | 1 => 2
        | _ => let
                 val k = pow(n div 2)
               in
                 if n mod 2 = 0 then k*k else 2*k*k
               end
```

$W_{pow}(n)$?

$W_{pow}(0)=c_0$
$W_{pow}(1)=c_1$
$W_{pow}(n)= W_{pow}(n \text{ div } 2) + c_2$

$W_{pow}(n)$： $O(\log n)$

badpow

$W_{badpow}(n)$?

```
fun badpow (n:int):int =
    case n of
         0 => 1
        | 1 => 2
        | _ => let
              val k2 = badpow(n div 2)*badpow(n div 2)
            in
              if n mod 2 = 0 then k2 else 2*k2
            end
```

$W_{badpow}(0)=c_0$
$W_{badpow}(1)=c_1$
$W_{badpow}(n)= 2*W_{badpow}(n \text{ div } 2) + c_2$

$W_{badpow}(2^m)$: $O(2^m)$
$W_{badpow}(n)$: $O(n)$

fib

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

$W_{fib}(n)$?