函数式编程原理

Lecture 3

值绑定

- 【x_1:v_1, ..., x_k:v_k 】： 表示值绑定(value bindings)的集合

  x, x_1, ... ：表示变量 (Variables)

  v, v_1, ... ： 表示值 ((syntactic) Values)

  e, e_1, ... : 表示表达式 (Expressions)

  t, t_1, ... ： 表示类型 (Types)

- 可终止状态(Termination):

  $e \downarrow$ when $\exists v.\ e \Rightarrow^* v$

- 不可终止状态(Non-termination):

  $e \uparrow$

- **TYPE SAFETY**
- 严格类型检查，**well-typed**特性

表达式推导

$[\![x:2]\!] (x + x)$ is $(2 + 2)$

$[\![x:2]\!] (\text{fn } y \Rightarrow x + y)$ is $(\text{fn } y \Rightarrow 2 + y)$

$[\![x:2]\!] (\text{if } x>0 \text{ then } 1 \text{ else } f(x-1))$

is $(\text{if } 2>0 \text{ then } 1 \text{ else } f(2-1))$

求值符号的使用

- $e \Rightarrow e'$　一次推导
- $e \Rightarrow^* e'$　有限次推导
- $e \Rightarrow^+ e'$　至少一次推导

- 如：fun f(x:int):int = f x

$$
\begin{aligned}
f\ 0 &\Rightarrow^+ (\text{fn } x \Rightarrow f\ x)\ 0 \\
&\Rightarrow^* [x{:}0]\ f\ x \\
&\Rightarrow^* f\ 0
\end{aligned}
$$

因此：$f\ 0 \Rightarrow^+ f\ 0$

$(f\ 0)\uparrow$

- =>和=>*可以精确反映程序行为
- 某些时候,计算顺序可以忽略,如

For all e_1, e_2 : int and all v:int

if $e_1 + e_2 \Rightarrow^* v$ then $e_2 + e_1 \Rightarrow^* v$

此时,我们关注计算结果多于计算过程

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)

范例1： 函数eval的说明

```
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 *)
```

范例2： 函数decimal的说明

```
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

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$x \geq 0$时, $f(x) = x+1$

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 *)
```

- To prove:

  For all values x:int

  such that $x \geq 0$, $f(x) = x+1$

用简单归纳法证明

- Let P(n) be f(n) = n+1
- Base case: we prove P(0), i.e. f(0) = 0+1

f 0 = (fn x => if x=0 then 1 else f(x-1)+1) 0
= ⟦x:0⟧(if x=0 then 1 else f(x-1)+1)
= **if** 0=0 **then** 1 **else** f(0-1) + 1
= **if true then** 1 **else** f(0-1) + 1
= 1

0+1 = 1

So f(0) = 0+1

用简单归纳法证明

- Let P(n) be f(n) = n+1
- Inductive step:
  let k≥0, assume P(k), prove P(k+1).
  Let v be the numeral for k+1.

f(k+1) = **if** v=0 **then** 1 **else** f(v-1) + 1

= **if false then** 1 **else** f(v-1) + 1

= f(v-1) + 1

= f(k) + 1 since v=k+1

= (k + 1) + 1 by assumption P(k)

So P(k+1) follows from P(k)

用简单归纳法证明

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

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

试证明：对所有值 L:int list，

存在一个整数 n，使eval L =>* n

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

?为什么不能用?

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

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

(when $n \geq 10, 0 \leq n \text{ div } 10 < n$)

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

试证明：对所有值 n:int (n≥0),

eval (decimal n) = n

```
fun decimal (n:int) : int list =
   if n<10 then [n] else (n mod 10) :: decimal (n div 10);
fun eval ([ ]:int list) : int = 0
  | eval (d::L) = d + 10 * (eval L);
```

用完全归纳法证明

对所有值 n:int (n≥0)，eval (decimal n) = n

当0 <= n < 10时，decimal(n)=[n], 要证明eval (decimal n) = n

eval (decimal n) = eval([n])=n+10*eval([ ])=n+0=n

当0 > 10时，有eval(decimal(m))=m,

要证明eval (decimal m+1) = m+1

设 n=m+1, x = m mod 10，y= m div 10

eval (decimal m+1) = eval((m+1 mod 10) :: decimal (m+1 div 10))

= x+1+10*eval(y)

eval(y)=eval(m div 10), y是一个个位数

=x+1 + 10* y = x + 10 * y +1 = m+1 = n

用完全归纳法证明

```
fun decimal (n:int) : int list =
  if n<10 then [n] else (n mod 10) :: decimal (n div 10);
fun eval ([ ]:int list) : int = 0
  | eval (d::L) = d + 10 * (eval L);
```

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

当0 <= n < 10时， decimal(n)=[n], 要证明eval (decimal n) = n

eval (decimal n) = eval([n])=n+10*eval([ ])=n+0=n

当0 > 10时， 对于any 0<=m<n,

有eval(decimal(m))=m, 要证明eval (decimal n) = n

设x = n mod 10， y= n div 10

eval (decimal n) = eval((n mod 10) :: decimal (n div 10))

= eval(x:: decimal y)n+10*eval([ ])

=x + 10* decimal y = x + 10 * y = n

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： 1) 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，

  2) 证明P(y::ys)成立，

换句话说，在$\forall i < k$, P(i)成立的条件下证明P(k)成立。

适用于涉及表和树的递归函数

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)的近似运行时间为O(g(n))：
  - 存在整数N和c，满足

    ∀n≥N, f(n)的近似运行时间≤ 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)

近似运行时间分析

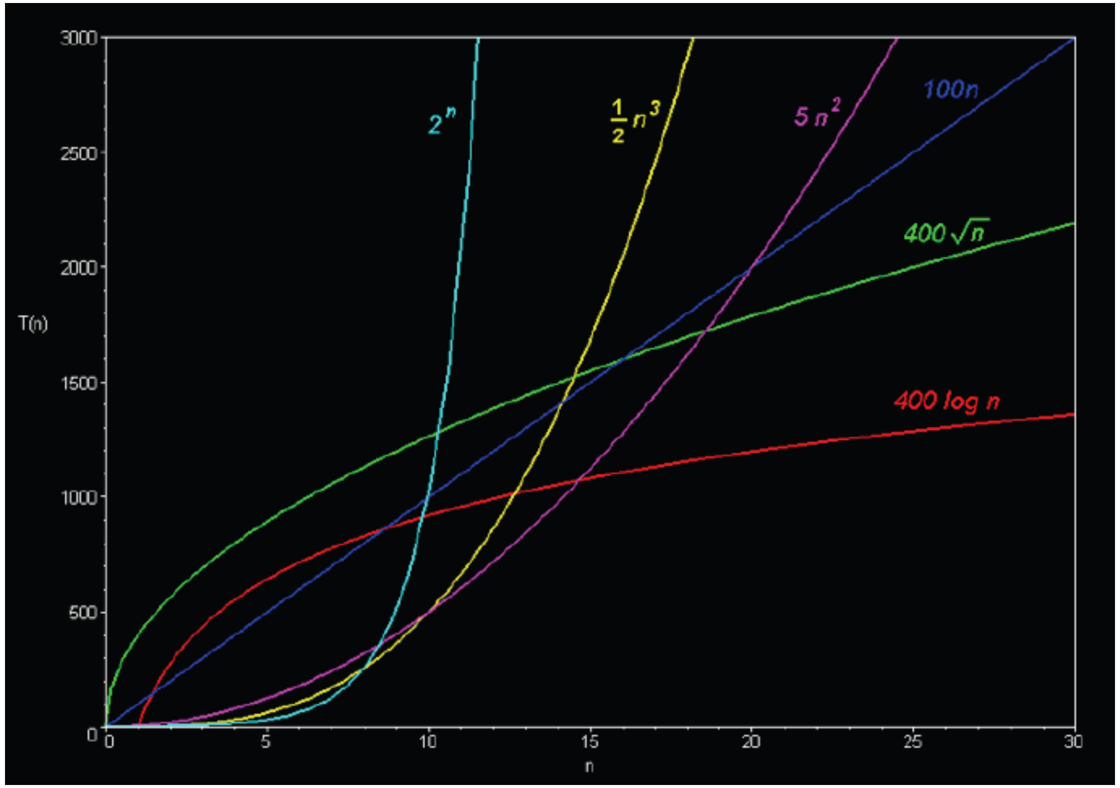
- 求解步骤：

  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(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 (P(0))：评估基本操作的执行 (Estimates the number of basic operations)
  - Inductive case (P(x))：
    - 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)

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)

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。

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);
```

程序执行时间随n值的增加线性增长

用$W_{exp}(n)$表示程序 exp(n)的执行时间

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

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

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

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?

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

程序执行情况分析：Can it be faster?

- The definition of exp relies on the fact that

$$2^n = 2\ (2^{n-1})$$

- Everybody knows that

$$2^n = (2^{n \text{ div } 2})^2 \qquad \text{if } n \text{ is even}$$

程序执行情况分析：Can it be faster?

```
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)

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

程序执行情况分析：Can it be faster?

```
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)
```

- Let $W_{fastexp}(n)$ be the runtime for fastexp(n)

$W_{fastexp}(0) = k_0$

$W_{fastexp}(n) = W_{fastexp}(n \text{ div } 2) + k_1$ for n>0, even

$W_{fastexp}(n) = W_{fastexp}(n-1) + k_2$ for n>0, odd

for some constants k_0, k_1, k_2

程序执行情况分析：Can it be faster?

```
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)
```

- Let $W_{fastexp}(n)$ be the runtime for fastexp(n)

  $W_{fastexp}(0) = c_0$

  $W_{fastexp}(1) = c_1$

  $W_{fastexp}(n) = W_{fastexp}(n \text{ div } 2) + c_2$ for n>1, even

  $W_{fastexp}(n) = W_{fastexp}(n \text{ div } 2) + c_3$ for n>1, odd

  for some constants c_0, c_1, c_2, c_3

程序执行情况分析： Can it be faster?

- Let $T_{fastexp}(n)$ be given by

$$T_{fastexp}(0) = 1$$

$$T_{fastexp}(1) = 1$$

$$T_{fastexp}(n) = T_{fastexp}(n \text{ div } 2) + 1 \quad \text{for } n>1$$

$T_{fastexp}(n) \leq c \log_2(n)$

for all large enough n