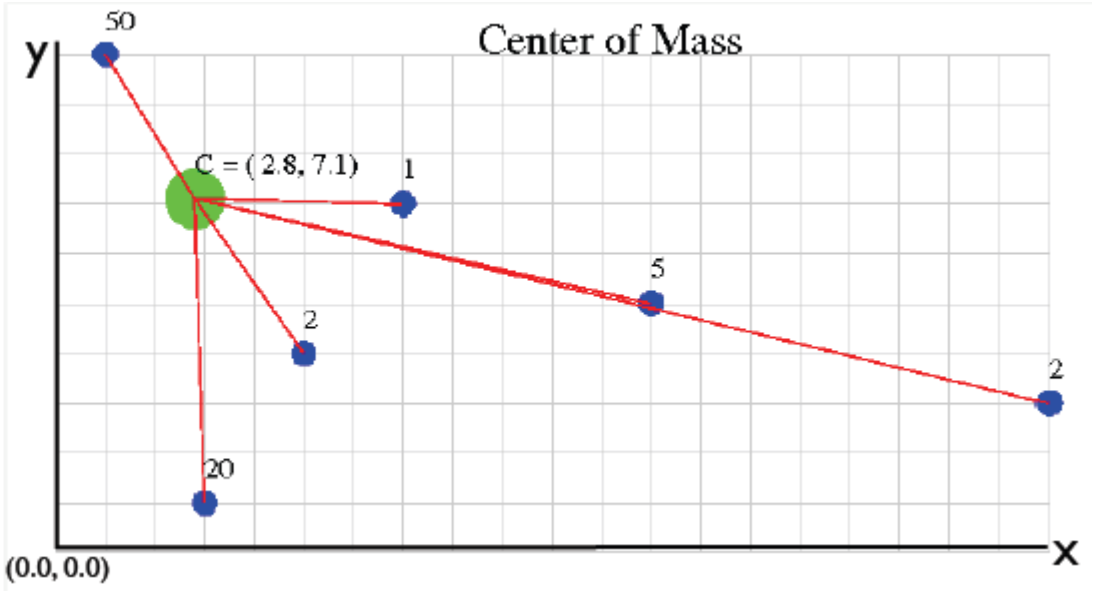
函数式编程原理

Lecture 8

剧情预告

- “多态”的力量(The power of polymorphism)
- 高阶函数编程实例
- 找零问题

高阶函数应用2——求解点集中心



给定点集： $[(m_1,(x_1,y_1)),\ldots,(m_n,(x_n,y_n))]$

求解中心点(X, Y): real * real，满足

$$X = (m_1*x_1 + \ldots + m_n*x_n)/M$$

$$Y = (m_1*y_1 + \ldots + m_n*y_n)/M$$

$$M = m_1 + \ldots + m_n$$

给定点集： $[(m_1,(x_1,y_1)),\ldots,(m_n,(x_n,y_n))]$

点集中心的求解

求解中心点(X, Y): real * real，满足

$X = (m_1 * x_1 + \ldots + m_n * x_n)/M$

$Y = (m_1 * y_1 + \ldots + m_n * y_n)/M$

$M = m_1 + \ldots + m_n$

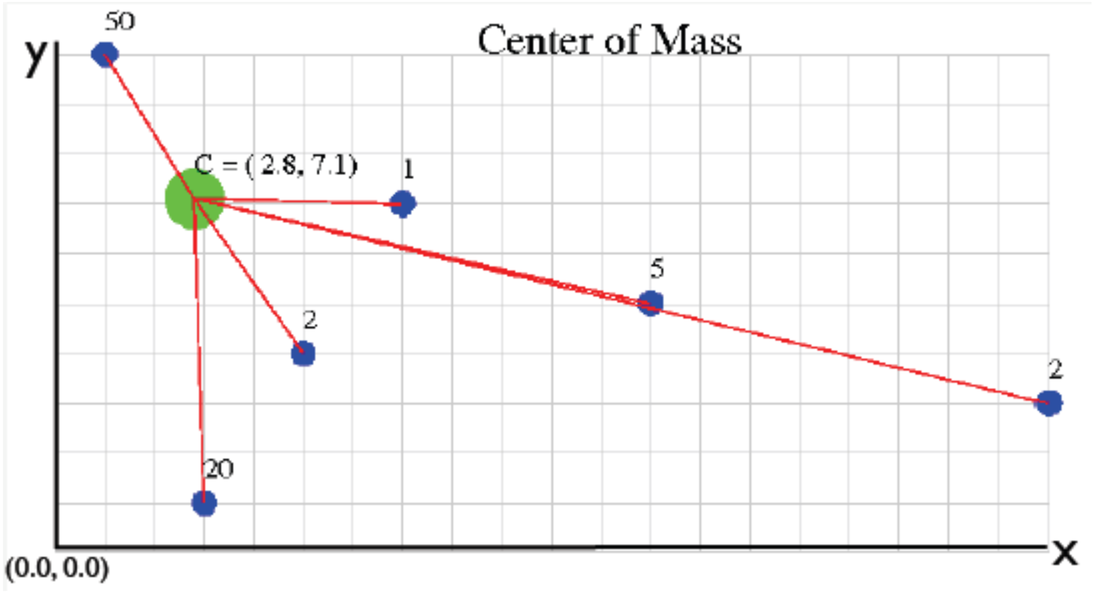
- 点集的表示： $[(m_1,(x_1,y_1)),\ldots,(m_n,(x_n,y_n))]$

```
type point = real * real
type body = real * point
```

- 辅助函数：

```
fun add((x1,y1), (x2,y2)):point = (x1+x2, y1+y2)
fun mass (m, (x, y)) = m
fun scale r (m, (x, y)) = (r * m * x, r * m * y)
fun center (L : body list) : point = let
        val M = foldr (op +) 0.0 (map mass L)
    In foldr add (0.0, 0.0) (map (scale (1.0/M)) L)
    end
```

点集中心的求解




```
- center [(50.0,(1.0,10.0)),(20.0,(3.0,1.0)),(2.0,(5.0,4.0)),
        (1.0,(7.0,7.0)),(5.0,(12.0,5.0)),(2.0,(20.0,3.0))];

val it = (2.8375,7.075) : real * real
```

高阶函数的更多应用

字符串的相关操作：

["all ","your "," base "] → "all your base are belong to us "

foldr (op ^) "are belong to us": string list -> string

[“all ”,“your ”,“base ”] → ["All ","Your ","Base "]

map *capitalize* : string list -> string list

explode : string -> char list
implode : char list -> string
Char.toUpper : char -> char

```
fun capitalize (s:string) : string =
        let val (x::L) = explode s
        In implode(Char.toUpper x :: L)
        end;
```

通用排序(general sorting)

- Isort, msort : int list -> int list
- Msort : tree -> tree

能否扩展为其他各种数据类型(如int, int*int，string等)的排序?

对公式/表达式进行抽象(An abstract formulation)：

1.对任意类型的数据，都能够进行比较(A type of data, with a comparison function)

2.对表和树等结构数据进行排序(Sorting lists and trees of data)

数据的预处理

对任意类型的数据，都能够进行比较

- 该类型需要配备比较函数
- 比较函数使数据有序，一般为数字顺序或词典/字典顺序

一般类型实例包括：

type	***comparison***	***ML***
int	*usual*	*compare*
int*int	*lexicographic*	*lex (compare, compare)*
string	*dictionary*	*String.compare*

数据的比较

- 类型t的比较函数： cmp : t * t -> order

- 比较函数的特点：
  - 逆反性：

    cmp(x,y)=LESS iff cmp(y,x)=GREATER

    cmp(x,y)=EQUAL iff cmp(y,x)=EQUAL

  - 传递性：

    cmp(x,y)=LESS & cmp(y,z) <> GREATER implies cmp(x,z)=LESS

    cmp(x,y)=GREATER & cmp(y,z) <> LESS implies cmp(x,z)=GREATER

    cmp(x,y)=EQUAL & cmp(y,z)=EQUAL implies cmp(x,z)=EQUAL

比较函数的实现

- for int:

```
compare : int * int -> order
fun compare(x:int, y:int):order =
if x<y then LESS else
if y<x then GREATER else EQUAL
```

compare(2,3) = LESS
compare(2,2) = EQUAL

- for other type data?

- for int*int:

```
leftcompare : (int * int) * (int * int) -> order
fun leftcompare((x1, y1), (x2, y2)) =
compare(x1, x2)
```

```
lexcompare : (int * int) * (int * int) -> order
fun lexcompare((x1, y1), (x2, y2)) =
case compare(x1,x2) of
      LESS => LESS
    | GREATER => GREATER
    | EQUAL => compare(y1, y2)
```

lexcompare((2,3),(3,2)) = LESS
lexcompare((2,3),(2,0)) = GREATER

二元组数据的通用比较函数lex

- 对任意类型、且异构的二元组数据，如何按数字何字典序进行比较?

  如(3, “Jack”) 与 (6, “Rose”), (4.5, (1.0,2.4))与(3.7, (2.6,5.1))……

lex : ('a * 'a -> order) * ('b * 'b -> order) -> ('a * 'b) * ('a * 'b) -> order

fun lex (cmp1, cmp2) ((x_1, y_1), (x_2, y_2)) =
case cmp1(x_1, x_2) **of**
LESS => LESS
| GREATER => GREATER
| EQUAL => cmp2(y_1, y_2)

lexcompare = lex(compare, compare)
: (int * int) * (int * int) => order

If cmp1为类型t1的比较函数
and cmp2为类型t2的比较函数

list数据的通用比较函数listlex

listlex : (’a * ’a -> order) -> ’a list * ’a list -> order

- 当cmp为类型t的比较函数时, listlex cmp实例化为类型t list的比较函数
- 比较规则：

```
listlex cmp ([ ], [ ]) = EQUAL
listlex cmp ([ ], y::R) = LESS
listlex cmp (x::L, [ ]) = GREATER
listlex cmp (x::L, y::R) = cmp(x,y)              if cmp(x,y)<>EQUAL
listlex cmp (x::L, y::R) = listlex cmp (L, R)  if cmp(x,y)=EQUAL
```

函数less与lesseq

```
less : (’a * ’a -> order) -> (’a * ’a -> bool)
lesseq : (’a * ’a -> order) -> (’a * ’a -> bool)
```

```
fun less cmp (x, y) = (cmp(x, y) = LESS)
```

```
fun lesseq cmp (x, y) = (cmp(x, y) <> GREATER)
```

函数sorted

sorted : (’a * ’a -> order) -> ’a list -> bool

```
fun sorted cmp [ ] = true
  | sorted cmp [x] = true
  | sorted cmp (x::y::L) =
      case cmp(x, y) of
          GREATER => false
        |    _       => sorted cmp (y::L)
```

L is cmp-sorted iff
sorted cmp L = true

函数insertion

ins : ('a * 'a -> order) -> ('a * 'a list) -> 'a list

```
fun ins cmp (x, [ ]) = [x]
  | ins cmp (x, y::L) =
      case cmp(x, y) of
           GREATER => y::ins cmp (x, L)
         |    _    => x::y::L
```

If cmp is a comparison and L is cmp-sorted,
ins cmp (x, L) = a cmp-sorted permutation of x::L

柯里函数(currying functions)

- 维基百科：
  - 在计算机科学中，柯里化（Currying）是把接受多个参数的函数变换成接受一个单一参数(最初函数的第一个参数)的函数，并且返回接受余下的参数且返回结果的新函数的技术。
  - 在直觉上，柯里化声称“如果你固定某些参数，你将得到接受余下参数的一个函数”。所以对于有两个变量的函数y^x，如果固定了 $y = 2$，则得到有一个变量的函数2^x。
- 柯里函数F : t1 -> t2 -> t可以部分应用类型t1的一个参数，从而产生新的函数(类型为t2 -> t)

  例如： ins : ('a * 'a -> order) -> ('a * 'a list) -> 'a list

  ins compare : int * int list -> int list

  ins String.compare : string * string list -> string list
- 函数F的”Uncurried”版本： 函数G:t1 * t2 -> t, 满足for all x:t1, y:t2, G(x,y) = (F x) y

  —— 一种使用匿名单参数函数来实现多参数函数的方法。

函数柯里化的技巧

ins: ('a * 'a -> order) -> ('a * 'a list) -> 'a list

为什么不是:

ins : ('a * 'a list) -> ('a * 'a -> order) -> 'a list ?

或

ins : ('a * 'a -> order) -> 'a -> 'a list -> 'a list ?

源于函数定义:

if cmp(x,y)=EQUAL, then

ins cmp (x, y::L) = x::y::L

排序函数isortl与isortr

isortl, isortr : ('a * 'a -> order) -> 'a list -> 'a list

```
fun isortl cmp L = foldl (ins cmp) [ ] L;
fun isortr cmp L = foldr (ins cmp) [ ] L;
```

isortl compare [3,1,2,1] = [1,1,2,3]
isortr lexcompare [(1,2),(2,2),(1,1),

If cmp is a comparison, then
for all lists L,
isortl cmp L = a cmp-sorted permutation of L
& isortr cmp L = a cmp-sorted permutation of L

代数规则(“algebraic” specs)

If g is total, then for all z and $[x_1, ..., x_n]$,

foldl g z $[x_1, ..., x_n] = g(x_n, g(x_{n-1}, ... g(x_1,z)...))$

foldr g z $[x_1, ..., x_n] = g(x_1, g(x_2, ... g(x_n,z)...))$

Let i = ins cmp.

isortl cmp $[x_1, ..., x_n] = i(x_n, i(x_{n-1}, ... i(x_1,[\ ])...))$

inserts “equal” items in the opposite order

isortr cmp $[x_1, ..., x_n] = i(x_1, i(x_2, ... i(x_n,[\ ])...))$

inserts “equal” items in the same order

```
fun isortl cmp L = foldl (ins cmp) [ ] L;
fun isortr cmp L = foldr (ins cmp) [ ] L;
```

isortl compare [3,1,2,1] = ?

isortr compare [3,1,2,1] = ?

算法稳定性：经过排序后，具有相同关键字的记录的相对次序保持不变，则称这种排序算法是稳定的；否则称为不稳定的。

isortr cmp稳定

isortl cmp不稳定

(rev L’)

继续扩展应用

- generalize mergesort and quicksort
- generalize from lists to trees

msort : (’a * ’a -> order) -> (’a list -> ’a list)

Msort : (’a * ’a -> order) -> (’a tree -> ’a tree)

应用高阶函数的好处

- One ***polymorphic*** *sorting* function s
- Can be used with different *types* and *comparisons*

```
s compare
        : int list -> int list
```

```
s (lex(compare,compare))
        : (int*int) list -> (int*int) list
```

```
s (listlex compare)
        : int list list -> int list list
```

应用多态类型的好处

- ***One*** type, ***many*** instances
- ***One*** specification, ***many*** special cases
- ***One*** function definition, ***many*** uses
- ***One*** correctness proof, ***many*** consequences

柯里化的好处

- 提高适用性
- 延迟执行：不断currying，累积传入的参数，最后执行
- 固定易变因素：提前把易变因素传参固定下来，生成一个更明确的应用函数

找零问题

- 给定整数n、一批硬币L、和某个限制条件p，是否能找出总值为n、且满足条件p的硬币子集?
    - 穷举法/枚举法：
        - 列举硬币组合的所有可能情况
        - 对所有可能情况逐一进行验证，直到全部情况验证完毕

        若某个情况验证符合条件，则为一个解；若全部情况验证后都不符合，则无解

    - 递归法：
        - 把找零分为两类：使用不包含第一枚硬币的所有零钱进行找零

          使用包含第一枚硬币的所有零钱进行找零
        - 两者方案之和即为问题求解结果

找零问题—穷举法

列举硬币组合的所有可能情况
对所有可能情况逐一进行验证，直到全部情况验证完毕
若某个情况验证符合条件，则为一个解；若全部情况验证后都不符合，则无解

- 需要解决几个子问题：
  - 穷举所有硬币L的所有子集

```
fun sublists [ ] = [ [ ] ]
  | sublists (x::R) = let val S = sublists R
                      in S @ map (fn L => x::L) S
                      end
```

  - 求解每个硬币子集的总值

```
fun sum L = foldr (op +) 0 L
```

  - 判断硬币子集的总值是否为n

```
sum A = n
```

  - 判断硬币子集是否满足条件p

```
fun exists p [ ] = false
  | exists p (x::R) = p(x) orelse exists p R
```

找零问题—穷举法(slowchange)

```
(* REQUIRES p is total                                   *)
(* ENSURES slowchange (n, L) p = true                     *)
(*              iff there is a sublist A of L with        *)
(*              sum A = n and p A = true                  *)
```

slowchange : int * int list -> (int list -> bool) -> bool

```
fun slowchange (n, L) p =
        exists (fn A => (sum A = n andalso p A)) (sublists L)
```

slowchange (210, [1,2,3,...,20]) (fn _ => true)

- 计算量大，性能极差
  - 没有递归
  - 暴力求解

把找零分为两类：

使用不包含第一枚硬币的所有零钱进行找零

使用包含第一枚硬币的所有零钱进行找零

两者方案之和即为问题求解结果

找零问题—递归法

避免穷举所有硬币子集——挑选合适的硬币子集

- 需要解决：
  - 首先确定基本情况(边界条件)：
    - n=0
    - n > 0, L = [ ]
  - 当n > 0, L = x::R时，递归调用

```
change : int * int list -> (int list -> bool) -> bool
(* REQUIRES p is total, n ≥0, L a list of positive integers     *)
(* ENSURES change (n, L) p = true                                *)
(*                 if there is a sublist A of L with             *)
(*                 sum A = n and p A = true                      *)
(*           change (n, L) p = false, otherwise                  *)
```

找零问题—递归法(change)

change (10, [5,2,5]) (**fn** _ => **true**)
= **true**
change (210, [1,2,3,...,20]) (**fn** _ => **true**)
=>* **true**
change (10, [10,5,2,5]) (**fn** A => length(A)>1)
= **true**
change (10, [10,5,2]) (**fn** A => length(A)>1)
= **false**

```
fun change (0, L) p =      p [ ]
  | change (n, [ ]) p =    false
  | change (n, x::R) p =
        if x <= n
        then (change (n-x, R) (fn A => p(x::A))
                orelse change (n, R) p)
        else change (n, R) p
```

程序的局限性

- 程序执行后返回布尔值，只能获取能否找零的结果(true能，false不能)
- 能否在判断过程中获取更多的信息：如果能找零，怎么找?

change : int * int list -> (int list -> bool) -> bool

mkchange : int * int list -> (int list -> bool) -> int list option

```
(* REQUIRES p is total, n ≥0, L a list of positive integers          *)
(* ENSURES mkchange (n, L) p = SOME A,                              *)
(*                  where A is a sublist A of L with                 *)
(*                  sum A = n and p A = true                         *)
(*                    if there is such a sublist;                    *)
(*              mkchange (n, L) p = NONE, otherwise          *)
```

mkchange

```
fun change (0, L) p = p [ ]
  | change (n, [ ]) p = false
  | change (n, x::R) p =
        if x <= n
        then (change (n-x, R) (fn A => p(x::A))
                orelse change (n, R) p)
        else change (n, R) p
```

```
fun mkchange (0, L) p =
        if p [ ] then SOME [ ] else NONE
  | mkchange (n, [ ]) p = NONE
  | mkchange (n, x::R) p =
        if x <= n
        then
            case mkchange (n-x, R) (fn A => p(x::A)) of
                SOME A => SOME (x::A)
              | NONE => mkchange (n, R) p
        else mkchange (n, R) p
```

表达式的计算

表达式

↓ Well-typed

类型推导

↓

表达式求值

- 结果为某个值
- or
- 永远循环

除逻辑错误外，是否会出现其他错误?

过程严谨，不会出错，但可能导致运行时错误。如fact 100、42 div 0

- 基于语法的类型检测：不对表达式进行求解，故不能避免运行时错误

如何确保程序安全运行?

引入异常

- ML中的异常：
  - ML自带一些异常处理(如Div，Overflow等)处理运行时错误
  - 由程序员自定义：异常声明(declaring)/抛出(raising)/处理(handling)
  - 机制灵活、作用域规则简单
  - 较好的适应类型规范
- 异常的引入：
  - 代码的调整：求值(Evaluation)/等价(Equality)/引用透明性(Referential transparency)

求值(evaluation)

表达式求值

结果为某个值
or
永远循环
or
处理运行时错误 → 抛出异常

等价(equality)

- 两个整型表达式相等　　　　　　当且仅当
  - 经推导、求值后，得到相同(same)的结果
  - or 都执行失败(无法终止)
  - or 都抛出相同的异常
- 类型为t -> t’ 的两个表达式相等　　　　当且仅当
  - 经表达式求值后，得到相等(equal)的结果
  - or 都执行失败(无法终止)
  - or 都抛出相同的异常

f = g iff
for all x,y : t,
x = y implies f(x) = g(y)

引用透明性(ref trans)

- 相等(equal)的子表达式可以作为参数进行传递：

If $e_1 = e_2$ then $E[e_1] = E[e_2]$

21 + 21 = 42, so (fn x:int => 21 + 21) = (fn x:int => 42)

fact 100 = fact 200, so

(fn x:int => fact 100) = (fn x:int => fact 200)

异常的声明(declaring)

exception Negative

exception Ring-ding-ding-ding-dingeringeding

exception Wa-pa-pa-pa-pa-pa-pow

选择合适的名称

```
exception Unimplemented
fun f (x: int) : int = raise Unimplemented
```

异常的作用域

与其他声明具有相同的作用域特点

```
let
    exception Foo
in
....
end
```

```
local
    exception Foo
in
....
end
```

OK to **raise** and **handle** Foo here

gcd : int * int -> int

```
(* REQUIRES x>0 & y>0 *)
(* ENSURES gcd(x, y) = the g.c.d. of x and y. *)
fun gcd (x, y) =
      case Int.compare(x, y) of
            LESS       => gcd(x, y-x)
          | EQUAL      => x
          | GREATER    => gcd(x-y, y)
```

无限循环

gcd(1, 0) =>* gcd(1-0, 0) =>* gcd(1, 0) =>* ...

gcd(1, ~1) =>* gcd(2, ~1) =>* gcd(3, ~1) =>*...

抛出异常 (溢出)

GCD : int * int ->int

(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* ENSURES GCD(x,y) = raise NotPositive if x ≤ 0 or y ≤ 0. *)

```
exception NotPositive
fun GCD (x, y) =    if (x <= 0 orelse y <= 0) then raise NotPositive else
                      case Int.compare(x,y) of
                           LESS => GCD(x, y-x)
                         | EQUAL => x
                         | GREATER => GCD(x-y x)
```

有问题吗?
递归调用时存在冗余测试!

```
GCD(42, 72)    =>* 6
GCD(1, 0)      =>* raise NotPositive
GCD(1, ~1)     =>* raise NotPositive
```

GCD : int * int ->int

```
(* REQUIRES true *)
(* ENSURES GCD(x,y) = the g.c.d of x and y if x > 0 and y > 0, *)
(* ENSURES GCD(x,y) = raise NotPositive if x ≤ 0 or y ≤ 0. *)
```

```
exception NotPositive
fun gcd (x, y) =
      case Int.compare(x, y) of
            LESS      => gcd(x, y-x)
          | EQUAL     => x
          | GREATER   => gcd(x-y, y)
```

只做一次测试

```
fun GCD(x, y) = if (x >0 andalso y >0) then gcd(x,y) else raise NotPositive
```

GCD‘ : int * int ->int

```
exception NotPositive
local
   fun gcd(x, y) =
         case Int.compare(x, y) of
                 LESS        => gcd(x, y-x)
               | EQUAL       => x
               | GREATER => gcd(x-y, y)
in  fun GCD'(x, y) = if (x > 0 andalso y > 0) then gcd(x,y) else raise NP
end;
```

even better: the dangerous gcd function
is not available outside

GCD = GCD‘

GCD : int * int -> int
GCD’ : int * int -> int

are **extensionally equal**, because:

for all integer values x and y,

EITHER x>0 & y>0, and

GCD(x,y) and GCD’(x,y) both evaluate to the g.c.d of x and y,

OR not(x>0 & y>0), and

GCD(x,y) and GCD’(x,y) both raise NotPositive

异常的处理(handling)

norris : int * int -> int
"For all values n:int, n div 0 = n."

```
fun norris(x:int, y:int) : int =
        (x div y) handle Div => x
```

e1 **handle** Foo => e2

- Has type t   if e1 and e2 have type t
- If e1 =>* v,   so does   e1 **handle** Foo => e2
- If e1 raises Foo,   e1 **handle** Foo => e2 =>* e2
- If e1 raises Bar,   so does   e1 **handle** Foo => e2
- If e1 loops, so does   e1 **handle** Foo => e2