函数式编程原理

Lecture 7

多态类型的推导(typability)

- t is a type for e

  iff (e has type t) is ***provable***
- In the scope of d, x has type t

  iff (d declares x:t) is ***provable***

If e has type t, and t' is an instance of t, then e also has type t'

list的反转函数： rev： 'a list -> 'a list

int list -> int list	is a type for	rev
real list -> real list	is a type for	rev
string list -> string list	is a type for	rev

Options类型

datatype ’a option = NONE | SOME of ’a

option：将空值和一般值包装成同一种类型。

- NONE：空值option
- SOME e：把表达式e的值包装成对应的option类型数据
- isSome t：查看t是否为SOME，如果t为NONE，则返回false
  如果t为SOME，则返回true
- valOf t：得到SOME包装的值。如valOf (SOME 5) = 5

Options类型

datatype ’a option = NONE | SOME of ’a

```
fun try (f, [ ]) = NONE
    | try (f, x::L) = case (f x) of
                          NONE => try (f, L)
                        |   y     => y
```

try : (’a -> ’b option) * ’a list -> ’b option

相等性 (equality)

- 等式类型：该类型的值能够进行相等性测试
  用“=”进行相等性判断
  - int类型
  - 用等式类型构建的元组或表
  - real和函数类型不是等式类型
- 等式类型表示为''a, ''b, ''c
- 使用时必须实例化

等式类型举例

```
fun mem (x, [ ]) = false
   | mem (x, y::L) = (x=y) orelse mem (x, L)
```

Declares mem : ''a * ''a list -> bool

实例化： int * int list -> bool

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

real * real list -> bool

所以…

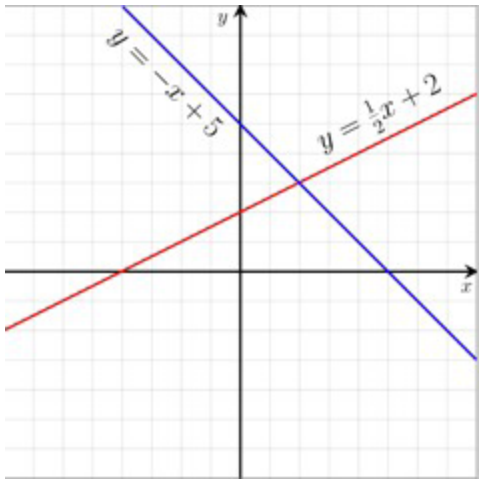
- 函数作为值 (Function as values)
- “多态”的力量(The power of polymorphism)
- 还有新的问题？？
  - list数据的单独求解问题——map
  - list数据的联合求解问题——foldr，foldl

新的需求/问题

数据标准化(归一化)：原始数据经过数据标准化处理，使各指标处于同一数量级，以便消除指标之间的量纲影响，进行综合对比评价。

对实数a,b (a<b)，存在线性函数f : real -> real，使f(a) = ~1.0，f(b) = 1.0

线性函数(一次函数)：在某一个变化过程中，设有两个变量x和y，如果可以写成y=α*x+β (α,β为实数)，就说y是x的一次函数



求解思路

min-max标准化：对原始数据进行线性变换，使结果值映射到[-1,1]之间

norm : real * real -> (real -> real)

对求解区间的实数a (min), b(max)，满足

norm(a, b) =>* 线性函数f满足：

f(a) = ~1.0 and f(b) = 1.0

函数norm

fun norm(a, b) = **fn** x => (2.0 * x - a - b) / (b - a)

- val norm = fn : real * real -> real -> real

norm : real * real -> (real -> real)

将[a,b]间的x进行变换，进行归一化处理，使结果值映射到[-1,1]之间，即

~1.0 ≤ norm(a, b) (x) ≤ 1.0

norm(a, b) a = ~1.0

norm(a, b) b = 1.0

函数norm执行后返回一个函数

例如：The *type* of norm(~2.0, 2.0) is real -> real

The *value* of norm(~2.0, 2.0) is

fn x => (2.0 * x - (~2.0) - 2.0) / (2.0 - (~2.0))

This value is *equal* to **fn** x => x / 2.0

函数norm的扩展使用

```
fun norm(-2.0,2.0) = fn x => x / 2.0
```

```
fun normpair(a, b) =
   fn (x, y) => let
          val f = norm(a,b)
        in
          (f x, f y)
        end
```

- 对实数对(实数二元组)进行归一化处理：

  利用norm(~2.0, 2.0)，将(1.0,1.5) 处理为 (0.5, 0.75)

```
fun normpair(a, b) =
      fn (x,y) => (norm(a,b) x, norm(a,b) y)
```

- 对实数表中的每个元素进行归一化处理：

  利用norm(~2.0, 2.0)，将[1.0,1.5,1.8]处理为[0.5, 0.75, 0.9]

需求分析

- 需求：如何将一个函数应用于某种数据结构中的所有元素
  - 批处理：对每个元素执行相同的操作(调用相同的函数)

——数据结构与函数无关

```
fun normpair(a, b) = fn (x,y) => (norm(a,b) x, norm(a,b) y)
fun fibpair(a, b) = (fib a, fib b)
fun factpair(x, y) = (fact x, fact y)
```

对表结构(list) 如何设计?

进一步思考

- 能否设计一个函数，能分别将不同函数应用于某种数据结构(pairs, tuples, lists ……)中的所有元素?
  - 对pairs，设计“多态”函数：

    pair : ('a -> 'b) -> 'a * 'a -> 'b * 'b

  - 对lists，设计“多态”函数：

    map : ('a -> 'b) -> 'a list -> 'b list

——高阶函数(***higher-order*** functions)

多态 vs. 高阶

- 多态(Polymorphism)类型：简化多类型的相同操作

pair, list, tree……

- 高阶(higher-order)函数：简化多参数的函数操作

简化同类型批量数据的不同函数操作

map, combining…

高阶是函数“多态”应用的一种体现

对pair的处理

pair : ('a -> 'b) -> 'a * 'a -> 'b * 'b

(* REQUIRES true *)
(* ENSURES pair f (x, y) = (f x, f y) *)

For all types t_1 and t_2,
all values $f : t_1 \rightarrow t_2$, and all values $x, y : t_1$,
pair f (x, y) = (f x, f y).

```
fun pair f = fn (x, y) => (f x, f y)
```

pair(norm (~2.0, 2.0)) : real * real -> real * real

pair (norm (~2.0, 2.0)) (1.5, 1.5) =>* ?

对list的处理

map : ('a -> 'b) -> ('a list -> 'b list)

```
fun map f = fn L =>
    case L of
      [ ] => [ ]
    | x::R => (f x) :: (map f R)
```

(* REQUIRES true *)
(* ENSURES For all n ≥0,
map f [x1, ..., xn] = [f x1, ..., f xn] *)

For all n ≥0, all types t_1 and t_2,
all values $f : t_1 \to t_2$, and all values $x_1, \ldots, x_n : t_1$,

$$\text{map } f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n].$$

map (norm(~2.0, 2.0)) : real list -> real list

map (norm(~2.0, 2.0)) [1.0, 1.5, 2.0] =>* [0.5, 0.75, 1.0]

语法分析

- ML对高阶函数采用流线型语法规则 (ML has a streamlined syntax for defining higher-order functions)

```
fun pair f = fn (x, y) => (f x, f y)
fun pair f (x,y) = (f x, f y)
```

```
fun map f = fn L => case L of
                    [ ] => [ ]
                  | x::R => (f x) :: (map f R)
fun map f [ ] = [ ]
  | map f (x::R) = (f x) :: (map f R)
```

list数据的map处理

map : ('a -> 'b) -> 'a list -> 'b list

(* REQUIRES true *)
(* ENSURES For all n ≥0,
map f [x_1, ..., x_n] = [f x_1, ..., f x_n] *)

```
fun map f [ ] = [ ]
  | map f (x::R) = (f x) :: (map f R)
```

Map可用于将某个函数操作同时应用于lists中的所有数据

给定一个list，求解该list的所有子list。

如：sublists [1,2,3] =
[[] ,[3],[2],[2,3],[1],[1,3],[1,2],[1,2,3]]

map函数应用——求解子集

(* sublists : 'a list -> 'a list list *)
(* ENSURES sublists L = a list of all sublists of L *)

算法思想：

(1) 把list分为两部分，第一个元素和剩余元素

(2) 原list的所有子list为：
剩余元素的子list并上
把第一个元素加入所有子list的list

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

另一类问题——批量数据的联合求解

- **real** list求和
- **int** list求乘积
- 寻找**int** list中的最小数
- 寻找**real** list中的最大数

算法设计思路：编写递归函数

(1) 设定一个初值；

(2) 设计相应功能函数递归应用于集合中所有的数

递归函数的设计(combining)：

1. 给定初值：$z : t_2$
2. 设定功能函数：

   $F : t_1 * t_2 \text{ -> } t_2$

   应用于list数据：

   $[x_1,\ldots,x_n] : t_1$ list
3. 求解过程：

   $F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))$

联合函数的设计

多态函数： foldr : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- 函数功能：

  for all types t_1, t_2,

  all $n \geq 0$, and all values $F: t_1 * t_2 \to t_2$, $[x_1,\ldots,x_n] : t_1$ list, $z : t_2$,

  foldr F z $[x_1,\ldots,x_n] = F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))$

```
fun foldr F z [ ] = z
    | foldr F z (x::L) = F(x, foldr F z L)
```

联合函数的应用——int list求和

sum : int list -> int

(* ENSURES sum L = the sum of the items in L *)

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

val sum = foldr (op +) 0
```

foldr (op +) 0 $[x_1,\ldots,x_n] = x_1 + (x_2 + \ldots (x_n + 0)\ldots)$

$= x_1 + x_2 + \ldots + x_n$

联合函数的应用——real list求最大值

```
(* REQUIRES L is a non-empty list                 *)
(* ENSURES maxlist L = the largest item in L  *)

fun maxlist (x::R) = foldr Real.max x R

Real.max : real * real -> real
```

foldr 与 foldl

```
fun foldl F z [ ] = z
   | foldl F z (x::L) = foldl F (F(x, z)) L
```

foldl : (’a * ’b -> ’b) -> ’b -> ’a list -> ’b

$\text{foldl F z } [x_1,\ldots,x_n] = F(x_n, F(x_{n-1},\ldots, F(x_1,z)\ldots))$

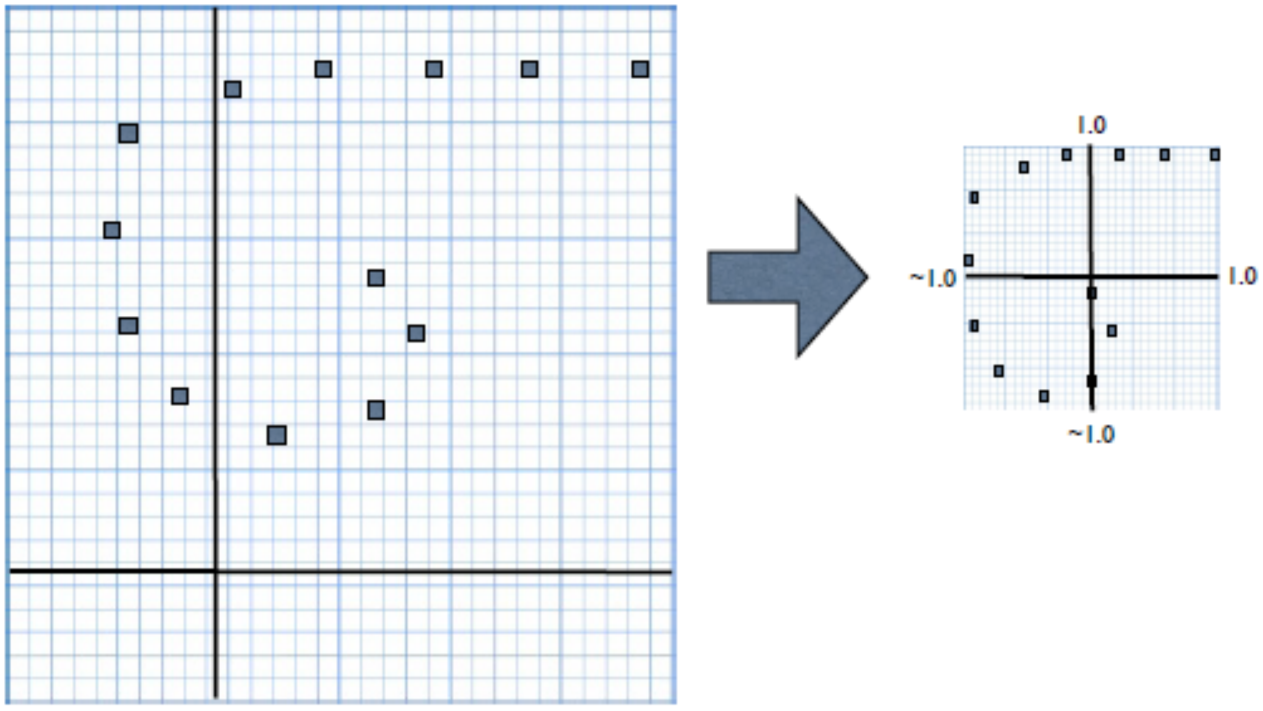
$\text{foldr F z } [x_1,\ldots,x_n] = F(x_1, F(x_2, \ldots, F(x_n, z)\ldots))$

vs.

foldr (op @) [ ] [[1,2], [ ], [3,4]]

foldl (op @) [ ] [[1,2], [ ], [3,4]]

高阶函数应用1——点集数据标准化



将非空、离散的点集标准化至空间： [~1.0 ... 1.0] X [~1.0 ... 1.0]

求解思路：

1.分离出x,y;

2.x,y分别norm标准化

1. 求解x，y的最大/最小值
2. 每个值norm标准化

```
fun norm(a, b) = fn x => (2.0 * x - a - b) / (b - a)
```

点集数据标准化——normalize

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)

(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  In    map (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y)) L
  end
```

ys)
val (xhi, yhi) = pair maxlist

点集数据标准化——normalize

(* normalize : (real * real) list -> (real * real) list *)

(* REQUIRES L is non-empty *)

(* ENSURES normalize L = a list of points in [~1.0...1.0] X [~1.0...1.0] *)

```
fun normalize (L : (real * real) list) : (real * real) list =
  let
    val xs = map (fn (x,y) => x) L
    val ys = map (fn (x,y) => y) L
    val (xlo, xhi) = (minlist xs, maxlist xs)
    val (ylo, yhi) = (minlist ys, maxlist ys)
  In    map (fn (x,y) => (norm(xlo, xhi) x, norm(ylo, yhi) y)) L
  end
```

```
val zs = unzip L
val (xlo, ylo) = pair minlist zs
val (xhi, yhi) = pair maxlist zs
```