函数式编程原理

Lecture 7

参考CMU 15-150 Lecture10, 11…-slides.pdf

上节课内容回顾

- 函数作为值 (Function as values)
- 高阶函数 (Higher-order functions)
- “多态”的力量(The power of polymorphism)
  - list数据的单独求解问题——map
  - list数据的联合求解问题——foldr, foldl

本节课主要内容

- 以排序算法为例
  - 设计通用排序函数解决排序类问题
  - 分析多态类型和高阶函数的好处
  - 柯里化和高阶函数的部分求解
- 大规模程序设计思路和方法

——以找零问题为例

高阶函数的更多应用—通用排序

- lsort, 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：

1. 对任意类型的数据，都能够进行比较
   - 配备比较函数，作为参数代入

2. 对表和树等结构数据进行排序

数据的比较

- 类型t的比较函数： cmp : t * t -> order

- 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

- 其他类型的数据如何比较?

- 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

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)

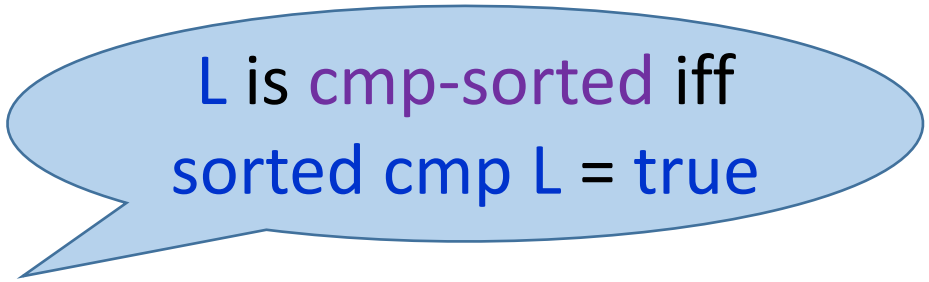
```
fun less cmp (x, y) = (cmp(x, y) = LESS)

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)
```



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
```

Why did we choose the type
('a * 'a -> order) -> ('a * 'a list) -> 'a list
Instead of
('a * 'a -> order) * ('a * 'a list) -> 'a list ?

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”。

  例如：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;

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。

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

isortr cmp稳定
isortl cmp不稳定
(rev L')

继续扩展应用

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

找零问题—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

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)

```
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
```

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

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                *)
```

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

mkchange: 提供找零方案

```
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
```

```
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
```

功能进一步扩充

- 能否将功能进行扩充，设计更为灵活的函数？

  如根据找零方案(***option***类型的值)生成给顾客的通知(string):
  - 成　功：告知顾客可以找零
  - 不成功：告知顾客需要更换钱币
- 结果具有不确定性，如何改进？

——利用多态类型

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

mkchange2 : int * int list -> (int list -> bool)
-> (int list -> 'a) -> (unit -> 'a) -> 'a

进一步推广功能和应用范围

```
mkchange2 : int * int list -> (int list -> bool)
                           -> (int list -> 'a) -> (unit -> 'a) -> 'a


(* REQUIRES p is total, n ≥0, L a list of positive integers          *)
(* ENSURES mkchange2 (n, L) p s k = s A,                             *)
(*                    where A is a sublist A of L such that          *)
(*                      sum A = n and p A = true                     *)
(*                       if there is one;                            *)
(*            mkchange2 (n, L) p s k= k ( ), otherwise               *)
```

mkchange2: 找零功能的扩充

```
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
```

```
fun mkchange2 (0, L) p s k =
        if p [ ] then s [ ] else k( )
  | mkchange2 (n, [ ]) p s k = k( )
  | mkchange2 (n, x::R) p s k =
        if x <= n
        then
              mkchange2 (n-x, R)
                (fn A => p(x::A))
                  (fn A => s(x::A))
                    (fn ( ) => mkchange2 (n, R) p s k)
        else
              mkchange2 (n, R) p s k
```

mkchange2的功效

- 函数change和mkchange2的关系？

```
fun change (n, L) p =
        mkchange2 (n, L) p (fn _ => true) (fn ( ) => false)
```

- 函数mkchange和mkchange2的关系？

```
fun mkchange (n, L) p =
        mkchange2 (n, L) p SOME (fn ( ) => NONE)
```

能否继续拓展

mkchange2 (n, L) p s k

mkchange2: int * int list -> (int list -> bool) -> (int list -> 'a) -> (unit -> 'a) -> 'a
n L p s k

- s is a *success continuation*
- k is a *failure continuation*

能否用这种continuation-style的程序/思想解决需要回滚(*backtracking*)的搜索问题？

回溯法

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

```
exception Unimplemented

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

- 抛出的异常视为任意类型

```
raise Foo
42 + raise Foo                  = raise Foo
(fn x:int => 0) (raise Foo)     = raise Foo
```

找零问题

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

```
fun change' (0, L) = [ ]
  | change' (n, [ ]) = raise Impossible
  | change' (n, x::R) =
      if x <= n
      then x :: change' (n-x, x::R)
           handle Impossible => change(n, R)
      else change' (n, R)
```

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

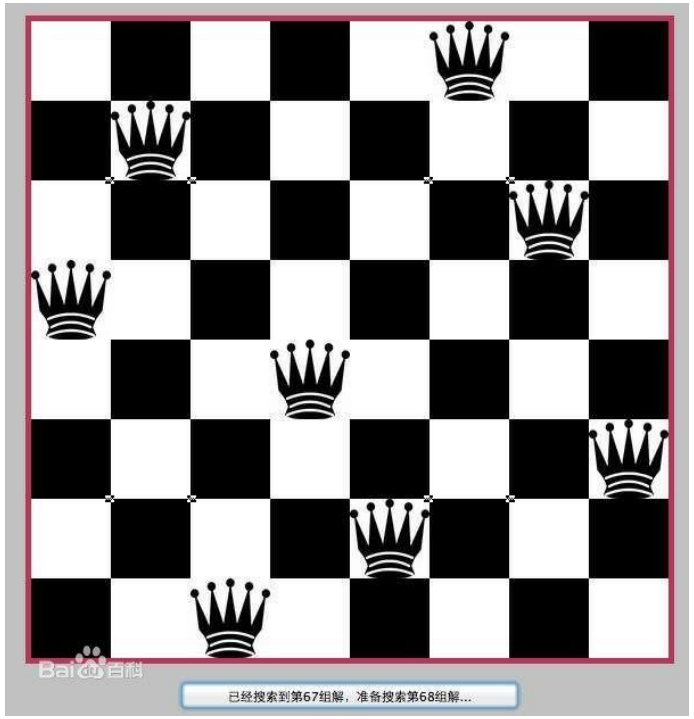
```
fun mkchange' (n, L) =
   SOME change' (n, L)
      handle Impossible => NONE
```

八皇后问题

- 在8x8格的国际象棋上摆放八个皇后，使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法？

在8!=40320种不同行/列/斜线的排列中共有92种解决方案：

从第0列开始，逐列进行搜索，找到一个不受任何现有皇后攻击的位置。如果找不到安全位置，则后退一步(改变前一个皇后的位置)重新查找，直到找到安全位置……



课程总结报告

纸质，2023年11月15日之前交至东五楼214：

一、实验报告

Heapify一题的实验报告，包括思路、代码、运行结果、性能分析、遇到的问题及如何解决等；

二、课程总结和建议

课程知识梳理总结或课程建议，包括头歌平台实验部署的改进意见和方案。