



华中科技大学

函数式编程原理课程报告

姓 名: 高兴

班 级: CS2102

学 号: U202115357

指导教师: 郑然

分数	
教师签名	

2023 年 10 月 19 日

目录

函数式编程原理课程报告	1
一、Heapify 求解	3
1.1 问题需求	3
1.2 解题思路与代码	3
1.3 遇到的问题及运行结果	4
1.4 性能分析（请用树的深度进行分析）	5
二、课程总结和建议	7
三、八皇后问题求解	8
3.1 问题需求	8
3.2 解题思路与代码	8
3.3 运行结果	10

一、Heapify 求解

1.1 问题需求

(1) 针对 minheap 树编写函数 `treecompare: tree * tree -> order`, `SwapDown: tree -> tree` 和 `heapify : tree -> tree`。

`treecompare` 参数为两棵树，输出为表示两棵树的根节点的值的比较结果的order值；

`SwapDown` 参数为一棵树且该树的两棵子树都是最小堆，输出为这棵树的最小堆；

`heapify` 的参数为一棵树，输出是由该树元素组成的最小堆。

(2) 在作业中分析 `SwapDown` 和 `heapify` 两个函数的 work 和 span。

1.2 解题思路与代码

依据题干中给出的最小堆的描述,可以得到如下的数据结构:

```
datatype tree = Empty | Br of tree * int * tree
```

针对`heapify` 函数由一棵随机输入的树生成最小堆的需求,只需要递归地将根节点的两个子树转化为最小堆后,就可以使用 `SwapDown` 函数获得输入树的最小堆,因此`heapify`的实现可以在`SwapDown`的实现基础上完成。

针对`SwapDown` 函数将一个数字和两个最小堆整合的需求,可以得到如下的递归逻辑:比较根节点的左右子树的大小,并交换根节点和较小子树的根节点。在交换之后,函数递归调用 `SwapDown` 函数来维护较小子树的堆属性,如此一来就可以保证整个树的最小堆属性;并且在不断的根节点与较小子树的交换中问题规模逐渐减小,使得`SwapDown`的递归必然结束,进而使得`heapify`的递归也必然结束。

在实现 `SwapDown`函数中两棵子树的比较时,需要涉及`treecompare` 函数的实现。只需要考虑待比较的两棵树是否为空,均为空时是为大小相同,一方为空则视为该方较小,如果都不为空则取出其中的两个子树各自的根节点,将两个根节点比较的结果作为 `treecompare` 函数的返回值即可。

其实现如下:

```

fun treecompare (Empty, Empty) = EQUAL
  | treecompare (Empty, Br (l, x, r)) = LESS
  | treecompare (Br (l, x, r), Empty) = GREATER
  | treecompare (Br (l1, x, r1), Br (l2, y, r2)) = Int.compare (x, y);

```

```

fun SwapDown(T as Br (a, x, b)) =
  let
    val (st, bt) = case treecompare(a, b) of
      GREATER => (b, a)
      | _      => (a, b)
    in
      case st of
        Empty => Br(st, x, bt)
      | Br(c, y, d) =>
          case treecompare(T, st) of
            GREATER => Br(SwapDown(Br(c, x, d)), y, bt)
            | _      => T
          end
    end
  end

```

```

fun heapify(Empty : tree) = Empty
  | heapify(Br(t1, n, t2)) = SwapDown(Br(heapify t1, n, heapify
t2));

```

1.3 遇到的问题及运行结果

本次实验中先后遇到了两个较为关键的问题其一，是测试运行时无法正确加载出 SwapDown 函数。错误截屏如图 1。

```
问题  输出  调试控制台  端口  终端

val getIntList = fn : int -> int list
val split = fn : 'a list -> 'a list * 'a list
datatype tree = Br of tree * int * tree | Empty
val trav = fn : tree -> int list
val listToTree = fn : int list -> tree
val treecompare = fn : tree * tree -> order
heapify.sml:42.5-54.8 Warning: match nonexhaustive
      T as Br (a,x,b) => ...

val SwapDown = fn : tree -> tree
val heapify = fn : tree -> tree
```

图 1warning

根据输出观察，发现程序在编译 SwapDown 函数时报出warning；在实际运行中则会有不匹配错误爆出。

```
问题  输出  调试控制台  端口  终端

      (Br (l1,x,r1),Br (l2,y,r2)) => ...

val treecompare = fn : tree * tree -> order
val SwapDown = fn : tree -> tree
val heapify = fn : tree -> tree
1 2 3 4 5 6 7
val L = [1,2,3,4,5,6,7] : int list
4 2 6 1 5 3 7 val it = () : unit
D:\Environment\smlnj\bin\run\run.x86-win32.exe: Fatal error -- Uncaught exception Match with 0
  raised at heapify.sml:39.59
```

图 2不匹配报错

经检查发现， SwapDown 函数没有考虑到输入参数是空树的情况，导致程序在递归执行中无法对叶子节点的子树进行正确的模式匹配，并导致程序没有实际的终止条件。添加空树特判后，程序能够正常加载。

1.4 性能分析（请用树的深度进行分析）

假设树的深度为 h 。

首先，易知treecompare的work和span都是 $O(1)$ 。

SwapDown 的 work 和 span 都是 $O(h)$ 。

分析如下：

首先是work：

$$\begin{aligned}\text{SwapDown}(h) &= \text{SwapDown}(h-1) + 2 * \text{treecompare}(a, b) + \text{Br}() \\ &= \text{SwapDown}(h-1) + C\end{aligned}$$

由此可以看出 $W_{(\text{SwapDown})} = O(h)$

接下来是span：

$$\begin{aligned}\text{SwapDown}(h) &= \max(\text{SwapDown}(h-1) + 2 * \text{treecompare}() + \text{Br}(), C) \\ &= \text{SwapDown}(h-1) + C\end{aligned}$$

因此SwapDown的Span也是 $O(h)$ 。

heapify 函数的 work 是 $O(2^h * h^2)$ ，span 是 $O(h^2)$ 。

分析如下：

heapify 函数对每个节点都调用了一次 SwapDown，又对每一个子节点使用 heapify 递归调用，而节点总数量级为 $O(2^n)$ ，故

$$\text{work}_{(\text{heapify})} = O(h * 2^h);$$

并发运行时, 对同层节点的递归调用可以并行，故

$$\text{span}_{(\text{heapify})} = O(h * h) = O(h^2)$$

二、 课程总结和建议

经过《函数式编程原理》这门课程的学习，我体会到了典型的命令式语言变成中的致命缺陷以及函数式编程的区别所在。函数式编程，程序的封闭性良好，函数的概念非常贴近数学中的“函数”概念：在函数式编程中，函数被视为一等公民，在可以匹配的情况下可以完全作为值来使用。函数式充分使用了递归的概念，在练习中我对于递归的理解也得到了充分的提升。

对于这门课程，我的建议是增加课后习题的提供，并使用理论课程中所提到的一些经典例题引导学生实践，以便于更好地让学生理解函数式编程的思想。

三、八皇后问题求解

3.1 问题需求

针对八皇后问题进行求解：在 8×8 格的国际象棋上摆放 8 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。

3.2 解题思路与代码

针对八皇后问题，采用回溯法的思想进行求解。首先将第一个皇后先放第一行第一列，然后将第二个皇后放在第二行第一列并判断是否冲突，如果冲突，继续继续依次穷举直到找到一个合适位置；继续第三个皇后，依旧穷举……直到8个皇后全都放在不冲突的位置，就是算是找到了一个正确解。当得到一个正确解时，函数开始回溯，倒序向前继续前面没有完成的穷举，直到每一种可能都成功或者提前因为冲突continue。经过这样的穷举，理论上可以找到所有的解。

在上述的过程中，首先要完成的是判断冲突的函数。

在solve函数实现中需要对棋形chbd进行冲突判断，因此编写 conserted(chbd, row, col)函数进行实现。conserted 函数表示皇后落在 chbd 棋盘上 (row, col) 处是否造成冲突，其输入是当前棋盘 chbd、目的位置的行数row和列数col。输出是chbd 棋形下 (row, col) 处落子是否协调，协调则返回true，否则返回false。

函数内部定义了一个名为 consertedHelper 的辅助函数，它接受四个参数：chbd, row, col 和 n。该函数用于递归地检查前 n 行中是否有皇后与当前位置 (row, col) 冲突。如果存在冲突，则返回 false，否则返回 true。

具体来说，函数会检查当前位置 (row, col) 是否与前 n-1 行中的任意一个皇后在同一列或同一对角线上。如果存在冲突，则返回 false，否则递归地检查前 n-1 行中的皇后是否与当前位置 (row, col) 冲突。

conserted 函数实现如下：

```
fun conserted (chbd, row, col) =  
  let  
    fun consertedHelper (chbd, row, col, 0) = true  
      | consertedHelper (chbd, row, col, n) =  
          if List.nth(chbd,n-1) = col orelse  
             abs(List.nth(chbd, n-1) - col) = abs(n - row)  
          then false  
          else consertedHelper(chbd, row, col,n-1)  
  in  
    consertedHelper(chbd, row, col, row-1)  
  end;
```


接下来编写回溯函数 solve。函数接受两个参数：chbd 和 row，分别表示已经放置的皇后的列号列表和当前行号。在8个皇后都安置好之前，函数会递归地尝试在当前行的每个位置放置皇后，并计算放置皇后后续行的解的总数。具体来说，函数内部定义了一个名为 placeQueen 的辅助函数，用于递归地尝试在当前行的每个位置放置皇后。如果当前位置合法，则递归地计算后续行的解的总数，并将其加到当前位置放置皇后的情况下的解的总数中。否则，递归地尝试在下一个位置放置皇后。最后，函数返回当前行的所有情况下的解的总数 solve 函数的实现代码如下：

```
fun solve (chbd, row) =
  if row > 8 then 1
  else
    let
      fun placeQueen (chbd, col) =
        if col > 8 then 0
        else
          let
            val chbd = chbd @ [col]
          in
            if conserted(chbd, row, col)
            then solve(chbd, row+1) + placeQueen(chbd,
col+1)
            else placeQueen(chbd, col+1)
          end
        in
          placeQueen(chbd, 1)
        end;
    end;
```

代码实现如下：

```
fun conserted (chbd, row, col) =
  let
    fun consertedHelper (chbd, row, col, 0) = true
    | consertedHelper (chbd, row, col, n) =
      if List.nth(chbd, n-1) = col orelse
        abs(List.nth(chbd, n-1) - col) = abs(n - row)
      then false
      else consertedHelper(chbd, row, col, n-1)
    in
      consertedHelper(chbd, row, col, row-1)
    end;
  fun solve (chbd, row) =
    if row > 8 then 1
    else
      let
        fun placeQueen (chbd, col) =
          if col > 8 then 0
```

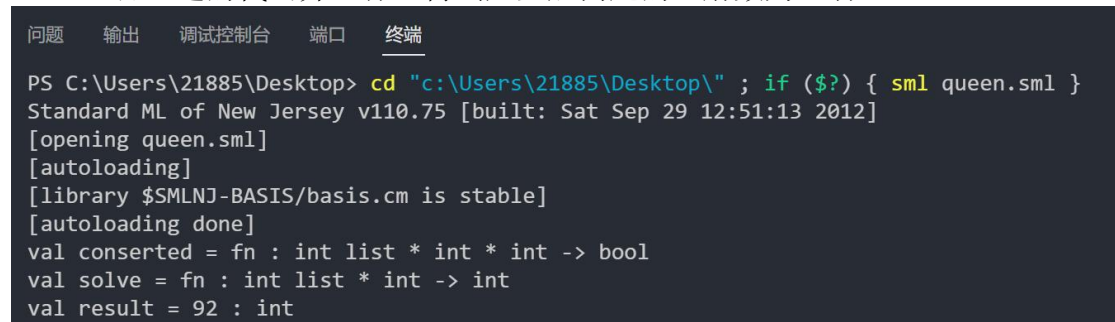
```

        else
            let
                val chbd = chbd @ [col]
            in
                if conserted(chbd, row, col)
                then solve(chbd, row+1) + placeQueen(chbd, col+1) else
                placeQueen(chbd, col+1)
            end
        in
            placeQueen(chbd, 1)
        end;
    val result = solve([], 1)

```

3.3 运行结果

组合上述的代码并运行，得到八皇后问题的总解数为92种。



```

问题  输出  调试控制台  端口  终端
PS C:\Users\21885\Desktop> cd "c:\Users\21885\Desktop\" ; if ($?) { sml queen.sml }
Standard ML of New Jersey v110.75 [built: Sat Sep 29 12:51:13 2012]
[opening queen.sml]
[autoloading]
[library $SMLNJ-BASIS/basis.cm is stable]
[autoloading done]
val conserted = fn : int list * int * int -> bool
val solve = fn : int list * int -> int
val result = 92 : int

```

图 3运行结果