函数式编程原理

Lecture 5

参考CMU 15-150 Lecture07, 08, 09-slides.pdf

上节课内容回顾

- List类型的两种排序算法
  - 插入排序
  - 归并排序
- List排序性能分析

数据类型变化：list -> tree

插入排序程序性能分析

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

for int list L (length L =n)

$W_{ins}(n)$ is $O(n)$

```
fun isort [ ] = [ ]
  |   isort (x::L) = ins (x, isort L)
```

for int list L (length L =n)

$W_{isort}(n)$ is $O(n^2)$

归并排序程序性能分析

```
fun split [ ]  = ([ ],  [ ])
    | split [x] = ([x], [ ])
    | split (x::y::L) =
            let val (A, B) =split L
            in (x::A, y::B)
            end
```

$W_{split}(n)$ is $O(n)$ (length(L)=n)

```
merge (A, [ ]) = A
   | merge ([ ], B) = B
   | merge (x::A, y::B) = case compare(x, y) of
                 LESS => x :: merge(A, y::B)
               | EQUAL => x::y::merge(A, B)
               | GREATER => y :: merge(x::A, B)
```

$W_{merge}(n)$ is $O(n)$ (length(A)+length(B)=n)

```
fun msort [ ] = [ ]
   | msort [x] = [x]
   | msort L =  let
                    val (A, B) = split L
                    in merge (msort A, msort B)
           end
```

$W_{msort}(n)$ is $O(n \log n)$
(length(L)=n)

- 有没有新的数据类型能并发执行?
  - Tree结构
  - 用树结构进行排序

msort(L)优于is
吗？

- 从程序算法上考虑：……
- 从数据结构上考虑：List结构为线性（顺序）结构，很难提升性能

本节课主要内容

- 新的数据类型：tree
- tree类型排序算法
- Tree排序并行性能分析 (Span)
- 类型检测
- 多态特点
- 类型推导

新的类型——tree

- tree：非线性数据结构
  - 由n(n>0)个元素组成的有限集合。每个元素称为结点(node)，一个特定的结点称为根结点(root)，且除根结点外，其余结点被分成m(m>0)个互不相交的有限集合，而每个子集又都是一棵树(子树)。

```
datatype tree = Empty | Node of tree * int * tree;
```

树的递归表述：Node(t_1, x, t_2) (t_1, t_2: tree, x: integer)

树的基本术语

- 度：结点的分支数。
- 叶子：度为0的结点称为叶子或终端结点。（度不为0的结点称为分支结点或非终端结点）
- 树的度：树中各结点的度的最大值。
- 双亲和孩子：结点的子树的根称为该结点的孩子，该结点称为孩子的双亲。
- 兄弟：同一双亲的孩子之间互称兄弟。
- 结点的层次：从根开始定义，根为第一层，其它结点的层次等于它的父结点的层次数加1。
- 深度：树的结点的最大层次称为树的深度。
- 路径：树中任意两个不同的结点，如果从一个结点出发，按层次自上而下沿着一个个树枝能到达另一结点，称它们之间存在着一条路径。可用路径所经过的结点序列表示路径，路径的长度等于路径上的结点个数减1。

树类型的模式表示与模式匹配

- Empty
- Node(_, _, _)
- Node(Empty, _, Empty)
- Node(_, 42, _)

树的基本函数

树的大小　size：树中的结点个数

```
fun size Empty = 0
  | size (Node(t1, _, t2)) =  size t1 + size t2 + 1;
```

树的深度(高度)　depth：组成该树各结点的最大层次（非负整数，从根到叶子节点的最长路径）

```
fun max(x,y): int = if x>y then x else y;
```

```
fun depth Empty = 0
  | depth (Node(t1, _, t2)) = max(depth t1, depth t2) + 1;
```

树的遍历

对树中所有结点信息的访问，即依次对树中每个结点访问一次且仅访问一次，分为前序遍历、中序遍历和后序遍历。

二叉树的遍历：

- 前序/前根遍历(Pre-order Traversal)
- 中序/中根遍历(In-order Traversal)
- 后序/后根遍历(Post-order Traversal)

```
trav: tree -> int list

fun trav Empty = [ ]
  | trav (Node(t1, x, t2)) =  trav t1 @ (x :: trav t2);
```

有序树(sorted trees)

若将树中每个结点的各子树看成是从左到右有次序的(即不能互换)，则称该树为有序树(Ordered Tree)；否则称为无序树(Unodered Tree)。

- 空树(Empty)为有序树
- Node(t1, x, t2)为有序树 当且仅当

  t1中的任意整数 ≤ x 且

  t2中的任意整数 ≥ x 且

  t1和t2均为有序树

$\Longleftrightarrow$ trav(t)为有序表

树的归并排序

Msort : tree -> tree

```
(* REQUIRES true                                         *)
(* ENSURES Msort(t) = a sorted tree                      *)
(*                  consisting of the items of t         *)

fun Msort Empty = Empty
  | Msort (Node(t1,x,t2)) =  Ins(x, Merge(Msort t1, Msort t2))
```

For all trees t,
Msort(t) = a sorted permutation of t

插入函数的移植

ins : int * int list -> int list

Ins : int * tree -> tree

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

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins (x, Node(t1, y, t2)) =
      case compare(x, y) of
          GREATER => Node(t1, y, Ins(x, t2))
      |       _         => Node(Ins(x, t1), y, t2);
```

For all sorted integer lists L,
ins(x, L) = a sorted permutation of x::L.

For all sorted integer tree t,
Ins(x, t) = a sorted tree t' such that
trav(t') is a perm of x::trav(t)

树的合并

Merge : tree * tree -> tree

```
(* REQUIRES t1 and t2 are sorted trees                    *)
(* ENSURES Merge(t1, t2) = a sorted tree t                *)
(*            consisting of the items of t1 and t2        *)

fun Merge (Empty, t2) = t2
  | Merge (Node(l1,x,r1), t2) = let
                                   val (l2, r2) = SplitAt(x, t2)
                                in
                                   Node(Merge(l1, l2), x, Merge(r1, r2))
                                end
```

For all sorted trees t1 and t2
Merge(t1, t2) = a sorted tree
consisting of the items of t1 and t2

树的拆分

Split : tree ~~->~~ tree * tree

split : int list -> int list * int list

SplitAt : int * tree -> tree * tree

(* REQUIRES t is a sorted tree *)

(* ENSURES SplitAt(y, t) = a pair (t_1, t_2)
such that every item in t_1 is ≤ y,
every item in t_2 is ≥ y,
and t_1,t_2 consist of the items in t *)

```
fun split [ ]  = ([ ],  [ ])
  | split [x] = ([x], [ ])
  | split (x::y::L) =
        let val (A, B) =split L
        in (x::A, y::B)
        end
```

For all L:int list,
split(L) = a pair of lists (A, B) such that length(A) ≈ length(B) and A@B is a perm of L.

```
fun SplitAt(y, Empty) = (Empty, Empty)
  | SplitAt(y, Node(t1, x, t2)) =
        case compare(x, y) of
          GREATER => let   val (l1, r1) = SplitAt(y, t1)
                     in   (l1, Node(r1, x, t2))
                          end
        |      _  => let   val (l2, r2) = SplitAt(y, t2)
                     in   (Node(t1, x, l2), r2)
                     end
```

程序的并行执行

```
fun Merge (Empty, t2) = t2
  | Merge (Node(l1,x,r1), t2) = let val (l2, r2) = SplitAt(x, t2)
                                in Node(Merge(l1, l2), x, Merge(r1, r2))
                                end
```

Merge(Msort t1, Msort t2)中的两个递归调用可以并行执行

- Merge串行执行的时间开销为分别对t1和t2执行Msort的开销之和+c
- Merge并行执行的时间开销为分别对t1和t2执行Msort开销的最大值+c
- 用”***span***”表示程序在足够多的并行处理器上的时间开销

*Span*和*work*的关系？

Ins函数的span

Ins : int * tree -> tree

```
fun Ins (x, Empty) = Node(Empty, x, Empty)
  | Ins (x, Node(t1, y, t2)) =
      case compare(x, y) of
          GREATER => Node(t1, y, Ins(x, t2))
        |    _        => Node(Ins(x, t1), y, t2);
```

For a tree of depth d>0,

$$S_{Ins}(d) = c + S_{Ins}(d-1)$$

$S_{Ins}(d)$ is $O(d)$

平衡二叉树：它是一 棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。

SplitAt函数的span

SplitAt : int * tree -> tree * tree

```
fun SplitAt(y, Empty) = (Empty, Empty)
  | SplitAt(y, Node(t1, x, t2)) =
      case compare(x, y) of
        GREATER => let
                     val (l1, r1) = SplitAt(y, t1)
                   in   (l1, Node(r1, x, t2))
                   end
      |    _    => let
                     val (l2, r2) = SplitAt(y, t2)
                   in   (Node(t1, x, l2), r2)
                   end
```

For a balanced tree of depth d>0,

$$S_{SplitAt}(d) = k + S_{SplitAt}(d-1)$$

$S_{SplitAt}(d)$ is $O(d)$

Merge函数的span

Merge : tree * tree -> tree

```
fun Merge (Empty, t2) = t2
  | Merge (Node(l1,x,r1), t2) = let
                                   val (l2, r2) = SplitAt(x, t2)
                                 in
                                   Node(Merge(l1, l2), x, Merge(r1, r2))
                                 end
```

For balanced trees of same depth d>0,

$$S_{Merge}(d) = S_{SplitAt}(d-1) + \max(S_{Merge}(d-1), S_{Merge}(d-1))$$

$S_{Merge}(d)$ is $O(d^2)$

Msort函数的span

Msort : tree -> tree

For balanced trees of same depth d>0,

$S_{Msort}(d)$ is $O(d^3)$

```
fun Msort Empty = Empty
  | Msort (Node(t1,x,t2)) =
      Ins(x, Merge(Msort t1, Msort t2))
```

```
fun Msort Empty = Empty
  | Msort (Node(t1, x, t2)) =
      Rebalance(Ins (x, Merge(Msort t1, Msort t2)))
```

类型分析 (type analysis)

- 静态类型检测能提供运行保障
- ML程序的基本特点：强类型(*well-typed*)

变量有且只有一种类型

——确保程序运行不会出错

  - ML只处理*well-typed*的表达式

    如果e为类型t且e=>*v， 那么v为类型t中的一个值.

  - ML只处理*well-typed*的声明

    如果声明d为类型t的某个值x， 那么d与x绑定，且类型为t

  - ML只处理*well-typed*的模式匹配

类型的引用透明性 (Referential transparency)

- 表达式类型依赖于子表达式的类型，依赖于自由变量的类型

x + x

has type int ？	if x has type int
has type real ？	if x has type real

ML标记出所有常量类型，并且将类型检测规则应用到每种形式的表达式上

什么时候检测和确定类型？

- 编译时进行类型分析和确定
  - 语法导向(*syntax-directed*)规则：表达式e的类型t依赖于表达式中自由变量的类型（规则的基础）

类型规则(Typing rules)

1. 数学运算：
   - 0, 1, 2, ~1, ... 类型：int
   - 0.0, 1.1, ~2.0, ... 类型：real
   - e1 + e2 类型：int/real if e1, e2 同为int/real
2. 表达式比较
   - e1 < e2 类型：bool if e1 , e2 同为 int或real
3. 分支语句
   - **if** e **then** e1 **else** e2 类型：t if e 为类型bool 且 e1, e2 同为类型t
4. 元组
   - (e_1, e_2) 类型：$t_1 * t_2$，if e_1 为类型t_1，且 e_2 为类型t_2

     Similarly for $(e_1, ..., e_k)$ when $k>0$

类型规则(Typing rules)

5. 表

- $[e_1, ..., e_n]$ 类型： t list if 对每个i, e_i 为类型t n≥0
- $e_1::e_2$ 类型： t list if e_1为类型t 且 e_2为类型t list
- $e_1@e_2$ 类型： t list if e_1 和e_2均为类型t list

6. 函数

- **fn** x => e 类型： t_1 -> t_2 if x为类型t_1, e 为类型 t_2

7. 应用

- e_1 e_2 类型： t_2 if e_1 为类型t_1 -> t_2 且e_2为类型 t_1

8. 声明

- **val** x = e 类型： x : t if e : t
- **fun** f x = e 类型： f : t_1 -> t_2 if x : t_1， f : t_1 -> t_2， e : t_2

类型规则(Typing rules)

9. let表达式 (let expressions)

- **let** d **in** e **end**　　　　类型： t_2

  if d declares x : t_1, ...,

  and, assuming x : t_1, ...,

  e has type t_2

10. 模式 (Patterns)

• _ fits t	always
• 42 fits t	iff t is int
• x fits t	always
• (p1, p2) fits t	iff t is t1*t2, p1 fits t1, p2 fits t2
• p1::p2 fits t	iff t is t1 list, p1 fits t1, p2 fits t1 list

多态类型(Polymorphic types)

- 多态：多种形态。

  类型推导后剩下一些无约束的类型，则声明就是多态的。

- ML has ***type variables***

  'a, 'b, 'c
- A type with type variables is ***polymorphic***

  'a list -> 'a list
- A polymorphic type has ***instances***

  int list -> int list

  real list -> real list

  (int * real) list -> (int * real) list

. .. instances of 'a list -> 'a list

多态类型是一个类型模式，用类型替换类型变量就形成一个类型模式的实例(instance)

多态的应用：split

```
fun split [ ] = ([ ], [ ])
  | split [x] = ([x], [ ])
  | split (x::y::L) =
      let val (A,B) = split L in (x::A, y::B) end
```

declares

  split : int list -> int list * int list

declares

  split : ’a list -> ’a list * ’a list

多态的好处：

1. 避免写较多多余的代码
2. 便于维护

多态的应用：sorting

Assuming split : int list -> int list * int list ⇒ split : ‘a list -> ‘a list * ‘a list

merge : int list * int list -> int list

```
fun msort [ ] = [ ]
  | msort [x] = [x]
  | msort L = let
            val (A,B) = split L
        in
            merge (msort A, msort B)
        end
```

declares msort : int list -> int list

declares msort : ’a list -> int list

多态类型

- ML allows ***parameterized*** datatypes

  datatype ’a tree = Empty
  | Node of ’a tree * ’a * ’a tree

  a type constructor tree

  and ***polymorphic*** value constructors

  Empty : ’a tree

  Node : ’a tree * ’a * ’a tree -> ’a tree

```
fun trav Empty = [ ]
  | trav (Node(t1, x, t2)) = (trav t1) @ x :: (trav t2)
```

declares trav : ’a tree -> ’a list

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

```
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

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

声明： mem : ''a * ''a list -> bool

- 等式类型表示为''a, ''b, ''c
- 使用时必须实例化

实例化： int * int list -> bool
(int list) * (int list) list -> bool
real * real list -> bool X