函数式编程原理

Lecture 6

1. 分析以下函数或表达式的类型**(**不要用**smlnj)**：

```
fun all (your, base) =
        case your of
             0 => base
           | _ => "are belong to us" :: all(your - 1, base)
```

Int * string list -> string list

```
fun funny (f, []) = 0
| funny (f, x::xs) = f(x, funny(f, xs))
```

(‘a * int -> int) * ‘a list -> int

```
(fn x => (fn y => x)) "Hello, World!“
```

string -> ‘a -> string

7. 编写函数reverse和reverse’，要求：

①函数类型均为：int list->int list，功能均为实现输出表参数的逆序输出；

②函数reverse不能借助任何帮助函数；函数reverse’可以借助帮助函数，时间复杂度为O(n)。

1. 函数类型均为： int list->int list， 功能均为实现输出表参数的逆序输出

```
fun reverse (L : int list) : int list =
  case L of
      [] => []
    | x::R => (reverse R) @ [x]
```

2. 函数reverse不能借助任何帮助函数；函数reverse’可以借助帮助函数，时间复杂度为*O*(n)。

```
fun reverse' (L : int list) : int list =
   let    fun reverse'' (L : int list, A : int list) : int list =
      case L of
          [ ] => A
        | x::R => reverse'' (R, x :: A)
   in     reverse'' (L, [ ])
   end
```

8. 给定一个数组A[1..n]，前缀和数组PrefixSum[1..n]定义为：
PrefixSum[i] = A[0]+A[1]+...+A[i-1]；
例如：PrefixSum [ ] = [ ]
PrefixSum [5,4,2] = [5, 9, 11]
PrefixSum [5,6,7,8] = [5,11,18,26]

试编写：
(1)函数PrefixSum: int list -> int list，
要求：$W_{PrefixSum}(n) = O(n^2)$。(n为输入int list的长度)
(2) 函数fastPrefixSum: int list -> int list，
要求： $W_{fastPrefixSum}(n) = O(n)$.
（提示：可借助帮助函数PrefixSumHelp）

(1) 函数PrefixSum: int list -> int list　要求：$W_{PrefixSum}(n) = O(n^2)$。

```
fun addList (x, [ ] ) = [ ]
| addList(x, y::L) = (x+y) :: addList(x, L)
fun PrefixSum [ ] = [ ]
| PrefixSum x::L = x :: addList(x, PrefixSum L)
```

(2) 函数fastPrefixSum: int list -> int list，要求：　$W_{fastPrefixSum}(n) = O(n)$.

```
fun PrefixSumHelp(x, [ ]) = []
    | PrefixSumHelp (x,y::L) = (x+y)::PrefixSumHelp(x+y,L)
fun fastPrefixSum L =PrefixSumHelp(0,L)

fun PrefixSum’ [ ] = [ ]
    | PrefixSum’ [x] = [x]
    | PrefixSum’(x::y::L) = x::PrefixSum’((x+y)::L)
```

树

数据类型变化：list -> tree

- 新的数据类型：tree
  - datatype tree = Empty | Node of tree * int * tree;
  - 基本函数操作
- 用tree类型设计排序算法
- tree类型排序算法的并行性能分析

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, _)

Empty *matches* t 当且仅当(iff) t is Empty

Node(p_1, p, p_2) *matches* t 当且仅当(iff)
t is Node(t_1, v, t_2) such that
p_1 *matches* t_1, p *matches* v, p_2 *matches* t_2
(and *combines* all the bindings
when the match succeeds)

树的结构归纳法推导过程

证明：对所有树，P(t)成立。

结构归纳法：

- Base case: 当t = Empty时，证明P(Empty)成立

- Inductive case: 当 t = Node(t_1, v, t_2)时,

  假设：P(t_1)和P(t_2)成立

  证明：P(Node(t_1, v, t_2))成立。

树的大小—— size

size：树中的结点个数

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

证明：对所有树t，size(t)为非负整数(t中的结点个数)

树的深度(高度)—— depth

depth：组成该树各结点的最大层次

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

证明：对所有树t，depth(t)为非负整数(从根到叶子结点的最长路径)

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。树的遍历分为前序遍历、中序遍历和后序遍历。

二叉树的遍历(N:访问结点本身，L:遍历该结点的左子树，R:遍历该结点的右子树)：

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

树的遍历函数

trav: tree -> int list

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

对所有树t，trav(t)执行的结果为树t中所有整数的列表(中序遍历的结果)

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

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)

树的拆分

split : int list -> int list * int list

?

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

SplitAt : int * tree -> tree * tree

```
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.

```
(* REQUIRES t is a sorted tree                    *)
(* ENSURES SplitAt(y, t) = a pair (t1, t2)
        such that
            every item in t1 is ≤ y,
            every item in t2 is ≥ y,
        and t1,t2 consist of the items in t *)
```

SplitAt : int * tree -> tree * tree

树的拆分

(* 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 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
```

- x>y, so x和t2应该归在新的右子树，同时拆分原左子树t1
  - t1递归SplitAt,返回l1<=y和r1>y
  - l1放在左子树， r1放在右子树

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

split后l2<=x和r2>x,
l1和l2都在x的左边，r1和r2都在x的右边

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))

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
```

- 对于一棵树将其root拿出，左右子树分别排序后merge，再插入root
- 循环直到某叶子节点的empty结束，拿到有序树
- 从叶子节点的empty开始merge回去

程序的并行执行

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

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
- 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
- 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 balanced tree of depth d>0,

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

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

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,

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

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

Msort函数的span

Msort : tree -> tree

For balanced trees of same depth d>0,

$S_{Msort}(d)$ is ?

```
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)))
```

tree的总结

- 新的数据类型：tree
  - datatype tree = Empty | Node of tree * int * tree;
  - 基本函数：size, depth, trav
- 用tree类型设计排序算法
  - Ins : int * tree -> tree
  - SplitAt : int * tree -> tree * tree
  - Merge : tree * tree -> tree
  - Msort : tree -> tree
- tree类型排序算法的并行性能分析 (Span)

复杂类型推导和规则应用

- 静态类型检测能提供运行保障 (a static check provides a *runtime* guarantee)
- 完善的推导规则，包含函数，分支，运算等值和操作
- 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 (ML only evaluates *well-typed* expressions)

  If e has type t and e =>* v, then v is a value of type t.

- ML只处理*well-typed*的声明 (ML only evaluates *well-typed* declarations)

  If d declares x of type t, then d binds x to a value of type t

- ML只处理*well-typed*的模式匹配 (ML only performs *well-typed* pattern matches)

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 (The type of an expression depends on the types of its sub-expressions and the types of its free varables)

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?

类型分析的时机

- 编译时进行类型分析和确定(type analysis can be done at compile time)
  - 语法导向(***syntax-directed***)规则用于表达式的类型判定：表达式e的类型t依赖于表达式中自由变量的类型
  - 表达式e和类型t的语法规范是规则的基础

类型规则(Typing rules)

- 基于某种假设，有如下语法导向规则(syntax-directed rules):

  e *has type* t

  d *declares* x : t

  p *fits type* t and *binds* x : t

1. 数学运算(Arithmetic):
   - 0, 1, 2, ~1, ... have type int
   - 0.0, 1.1, ~2.0, ... have type real
   - e1 + e2 has type int if e1 and e2 have type int
   - e1 + e2 has type real if e1 and e2 have type real

   e1 + e2 is not well-typed, otherwise

similarly
e1 - e2
e1 * e2

类型规则(Typing rules)

2. 表达式比较(Comparison)

- $e_1 < e_2$ has type bool   if e_1 and e_2 have type int
- $e_1 < e_2$ has type bool   if e_1 and e_2 have type real

  $e_1 < e_2$ is not well-typed, otherwise

3. 分支语句 (Conditional for all types t)

- **if** e **then** e_1 **else** e_2 has type t

  if e has type bool and e_1, e_2 have type t

  if e **then** e_1 **else** e_2 is not well-typed, otherwise

both branches must have the same type

类型规则(Typing rules)

4. 元组 (Tuples for all type t_1 and t_2)

- (e_1, e_2) has type $t_1 * t_2$ if e_1 has type t_1 and e_2 has type t_2

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

  ( ) has type unit

5. 表(List for all type t)

- $[e_1, ..., e_n]$ has type t list if for each i, e_i has type t $n \geq 0$
- $e_1::e_2$ has type t list if e_1 has type t and e_2 has type t list
- $e_1@e_2$ has type t list if e_1 and e_2 have type t list

类型规则(Typing rules)

6. 函数 (Functions)

- **fn** x => e has type t_1 -> t_2 if, assuming $x : t_1$, e has type t_2

  fn x => e is not well-typed, if no such t_1 and t_2 exist

7. 应用(Application)

- $e_1\ e_2$ has type t_2 if e_1 has type t_1 -> t_2 and e_2 has type t_1

  $e_1\ e_2$ is not well-typed, otherwise
  if e_1 does not have a function type,
  or e_1 has type t_1 -> t_2 but e_2 doesn't have type t_1

类型规则(Typing rules)

8. 声明 (Declarations)

- **val** x = e declares x : t   if e has type t
- **fun** f x = e declares $f : t_1 \to t_2$

  if, assuming $x : t_1$ and $f : t_1 \to t_2$, e has type t_2

  (also rules for *combining* declarations)

```
val x = 42
fun f(y) = x+y
```

declares：

x : int and f : int -> int

类型规则(Typing rules)

9. let表达式 (let expressions)

- **let** d **in** e **end** has type t_2

  if d declares x : t_1, ..., e has type t_2

```
let
   val x = 21
in
   x + x
end
```

```
let
   val x = 21
   fun f(y) = x+y
in
   x + (f x)
end
```

has type int

类型规则(Typing rules)

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

规则的应用： 函数

- **fn** p_1 => e_1 | … | p_k => e_k has type t_1 -> t_2

  if for each i,fitting p_i to t_1 succeeds,

  with type bindings for which e_i has type t_2

fn 0 => 0 | n => n - 1

has type int -> int

规则的应用：递归函数

- **fun** f p_1 = e_1 | … | f p_k = e_k declares f : t_1 -> t_2

  if for each i,

  matching p_i to t1 succeeds,

  with type bindings for which,

  assuming f : t_1 -> t_2, e_i has type t_2

```
fun f 0 = 0 | f n = f (n - 1)

fun f n = if n=0 then 1 else n + f (n - 1)
```

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，
- 用某个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