

函数式编程原理

Lecture 4

上节课内容回顾

- 程序正确性验证：归纳法
 - 简单归纳法
 - 完全归纳法
- 程序有效性验证：基于归纳法的work分析

本节课主要内容

- List类型的应用（以排序为例）
 - 插入排序
 - 归并排序
- 排序程序的性能分析
 - 数据类型变化：list -> tree
- 新的数据类型：tree

整数的比较——compare

compare: int * int -> order

datatype order = LESS | EQUAL | GREATER;

fun compare(x:int, y:int):order =

if x<y **then** LESS **else**

if y<x **then** GREATER **else** EQUAL;

compare(x,y) = LESS	if x<y
compare(x,y) = EQUAL	if x=y
compare(x,y) = GREATER	if x>y

排序结果的判断——sorted

- `sorted : int list -> bool`
- 函数功能：线性表中的元素按照升序（允许相邻元素相同）的方式排列，则该整数表为有序表（增序）。A list of integers is \leq -sorted: if each item in the list is \leq all items that occur later.

——用于排序算法的测试

- 函数代码：

```
fun sorted [ ] = true
  | sorted [x] = true
  | sorted (x::y::L) =
    (compare(x,y) <> GREATER) andalso sorted(y::L);
```

For all L : int list,
sorted(L) = true if L is \leq -sorted
 = false otherwise

插入排序

- 基本思想：

- 每次将一个待排数据按大小插入到已排序数据序列中的适当位置，直到数据全部插入完毕。

- 操作步骤：

1. 从有序数列和无序数列 $\{a_2, a_3, \dots, a_n\}$ 开始进行排序；
2. 处理第 i 个元素($i=2, 3, \dots, n$)时，数列 $\{a_1, a_2, \dots, a_{i-1}\}$ 是有序的，而数列 $\{a_i, a_{i+1}, \dots, a_n\}$ 是无序的。用 a_i 与有序数列进行**比较**，找出合适的位置将 a_i 插入；
3. 重复第二步，共进行 $n-i$ 次**插入**处理，数列全部有序。

如何用递归程序实现？

整数的插入——ins

ins : int * int list -> int list

(* REQUIRES L is a sorted list *)

(* ENSURES ins(x, L) = a sorted perm of x::L *)

fun ins (x, []) = [x]

| ins (x, y::L) = **case** compare(x, y) **of**

GREATER => y::ins(x, L)

| _ => x::y::L

如何证明 ?

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

用归纳法证明Ins函数的正确性

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

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

• 根据L的长度进行归纳证明

1. L长度为0时，证明ins(x, []) 为有序表.
2. 假设对所有长度小于k的有序表A，ins(x, A) 为 x::A的有序表.
证明：ins(x, L) 为x::L的有序表,其中L的长度为k且为有序表

插入排序——isort

isort : int list -> int list

(* REQUIRES true *)

(* ENSURES isort(L) = a sorted perm of L *)

fun isort [] = []

| isort (x::L) = ins (x, isort L)

For all integer lists L,
isort L = a <-sorted permutation of L.

如何证明？

另一个插入排序——isort'

- isort' : int list -> int list

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

isort and isort' are extensionally equivalent.

For all L : int list, $\text{isort } L = \text{isort}' L$.

归并排序

`msort: int list -> int list`

- 基本思想：采用分治法（*Divide and Conquer*）将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。

`split : int list -> int list * int list`

- 操作步骤：
 1. 将 n 个元素**分成**两个含 $n/2$ 元素的子序列
 2. 将两个子序列递归排序
 3. **合并**两个已排序好的序列

`merge : int list * int list -> int list`

表的分割——split

split : int list -> int list * int list

(* REQUIRES true *)
(* ENSURES split(L) = a pair of lists (A, B) *)
(* such that length(A) and length(B) differ by at most 1, *)
(* and A@B is a permutation of L. *)

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

能否去掉?

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

如何证明 ?

用归纳法证明split函数的正确性

根据L的长度用完全归纳法进行证明

1. $L = [], [x]$

① $\text{split } [] = \text{a pair } (A, B) \text{ such that } \text{length}(A) \approx \text{length}(B) \text{ \& } A @ B \text{ is a perm of } []$.

② $\text{split } [x] = \text{a pair } (A, B) \text{ such that } \text{length}(A) \approx \text{length}(B) \text{ \& } A @ B \text{ is a perm of } [x]$.

2. 假设 $\text{split}(R) = \text{a pair } (A', B') \text{ such that } \text{length}(A') \approx \text{length}(B') \text{ \& } A' @ B' \text{ is a perm of } R$.

证明: $\text{split}(L) = \text{a pair } (A, B) \text{ such that } \text{length}(A) \approx \text{length}(B) \text{ \& } A @ B \text{ is a perm of } x :: y :: R$.
($L = x :: y :: R$)

表的合并——merge

merge : int list * int list -> int list

(* REQUIRES A and B are <-sorted lists *)
(* ENSURES merge(A, B) is a permutation of A@B *)

fun 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)

能否写成:

merge ([], []) = []

| merge ([x], []) = [x]

| merge ([], [y]) = [y]

如何证明 ?

归并排序——mergesort

msort : int list -> int list

(* REQUIRES true *)
(* ENSURES msort(L) = a <-sorted perm of L *)

如何证明？

fun msort [] = []
| msort [x] = [x]
| msort L = **let**

能否去掉？

val (A, B) = split L
 in
 merge (msort A, msort B)
 end

fun msort [] = []
| msort [x] = [x]
| msort L = **let**

val (A, B) = split L
 val A' = msort A
 val B' = msort B
 in merge (A', B')
 end

归并排序

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

    in (x::A, y::B)
    end
```

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

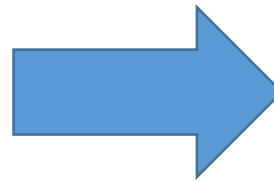
```
fun merge (A, [ ]) = A
  | merge ([ ], B) = B
  | merge (x::A, y::B) = case compare(x, y) of
    LESS => x :: merge(A, y::B)
  | EQUAL => x::y::merge(A, B)
  | GREATER => y :: merge(x::A, B)
```


msort的正确性验证

For all L :int list, if $\text{length}(L) > 1$
then $\text{split}(L) = (A, B)$
where A and B have *shorter length than* L
and $A@B$ is a permutation of L

For all **sorted** lists A and B ,
 $\text{merge}(A, B)$ = a sorted permutation of
 $A@B$

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



For all L :int list,
 $\text{msort}(L)$ = a \leq -sorted
permutation of L .

ML编程原则(principles)

- 每个函数都对应一个功能描述说明 (Every function needs a spec)
- 需要验证程序符合功能描述说明 (Every spec needs a proof)
- 用归纳法进行递归函数的正确性验证 (Recursive functions need inductive proofs)
 - 选取合适的归纳法 (Learn to pick an appropriate method...)
 - 设计恰当的辅助函数 (Choose helper functions wisely)

*m*sort的证明非常简单，源于
函数*split and merge*的使用

帮助(helper)函数

- 满足调用函数的功能需求
- 扩展应用到其他函数中，实现更广泛的功能

`merge : int list * int list -> int list`

在归并排序中：

For all **sorted lists** A and B,
`merge(A, B)` = a **sorted permutation** of
A@B

通常情况下：

For all **integer lists** A and B,
`merge(A, B)` = a **permutation** of
A@B

- 考虑性能调优

功能说明的作用 (the joy of specs)

- 函数的证明有时依赖于某个被调用函数的证明结果(符合spec要求)

The **proof** for `mmerge` relied only on the *specification proven for `split`* (and the specification proven for `merge`)

- 被调用函数可以由具有相同功能说明的其他函数替换，而且证明过程仍然有效

In the definition of `mmerge` we can *replace* `split` by *any function that satisfies this specification*, and *the proof will still be valid*, for the new version of `mmerge`

函数替换举例

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

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

尽管`split`和`split'`函数不相同，但他们都满足整数表分割功能，在正确性证明过程中没有区别，所以函数`msort`和`msort'`都是正确的。

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)

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

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

for int list L (length L = n)

$W_{\text{isort}}(n)$ 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
```

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

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

$W_{\text{msort}}(n)$ is $O(n \log n)$
(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)

$W_{\text{merge}}(n)$ is $O(n)$ (length(A)+length(B)=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isort(L)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