CSE 505

Lecture #12
October 10, 2012

ML Demo

Recursion is like the "goto" of Functional Programming

The newer languages (notably Java) have abandoned the "goto" statement in favor of control structures (for, while) and 'exit' statements.

Similarly, the newer functional languages are leaning towards constructs that avoid explicit recursive programming. Two notable approaches:

- functional operators
- list comprehensions

Functional Operators

The functions map(f) and reduce(f, b) encapsulate common patterns of recursion over lists. They are sometimes called "operators" since they take a function as input and return a function as output:

map: ('a → 'b) → ('a list → 'b list)

reduce: ('a → 'b) * 'b → ('a list → 'b)

Programming with operators minimizes recursion, and enhances program readability – sometimes at the expense of efficiency.

Programming with Operators

comp(f,g), denoted as 'f ∘ g':

```
fun comp(f,g) = let fun h(x) = f(g(x)) in h end;
```

pair(f1, f2), denoted as <f1, f2>:

```
fun pair(f1, f2) = let fun h(x) = (f1(x), f2(x)) in h end;
```

transp(l1, l2), denoted as transp:

```
fun transp(l1, l2) = let fun h([],[]) = []
                       | fun h(h1::t1, h2::t2) =
                             (h1,h2) :: trans(t1,t2)
                      in h
                     end;
```

Simple Example

```
fun innerprod([], []) = []
  | innerprod(h1::t1, h2::t2) = h1 * h2 +
                                innnerprod(t1,t2)
```

Using functional operators:

innerpod = reduce(plus,0) ∘ map(prod) ∘ transp

where plus(x,y):int = x+y;
prod(x,y):int = x*y;

Relating comp, map, reduce (algebra of programs)

<f, g> ∘ h   = <f ∘ h,   g ∘ h>

map(f1) ∘ map(f2)   = map(f1 ∘ f2)

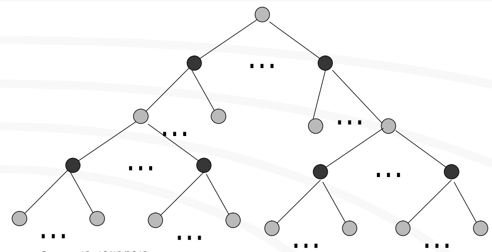
map(f) = reduce(h, []), where h(x, y) = f(x) :: y

etc.

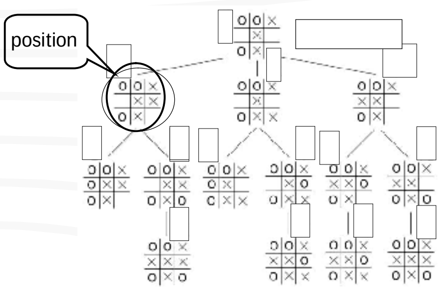
Helps compiler perform optimizations.

Game Trees

Game Trees are typically infinite structures for nontrivial games such as a chess.



Game Tree
(fragment of Tic Tac Toe)



Strength Assessment in Infinite Game Trees

datatype ‘a gametree   = node of ‘a * ‘a gametree list;

fun moves: position → position list

fun strength: position → int

fun assess(position) = minimax ∘
treemap(strength) ∘
prune(n) ∘
game(moves, position)

Map each node to a number

Keep only top n levels of tree

May produce an infinite game tree

“Why Functional Programming Matters” by John Hughes*

This easy-to-read article explains through a number of different types of example the benefits of higher-order functions and lazy evaluation for achieving great program modularity. Numeric and non-numeric examples are given. One separate section devoted to game trees.

* From “Research Topics in Functional Programming” ed. D. Turner, Addison-Wesley, 1990, pp 17–42

List Comprehensions

Originally introduced in Miranda (early 1980’s), and more recently also found in Python.
Helps avoid unnecessary recursive definitions.

‘[‘ expr : <generators>’]’

‘[‘ expr : <generators> ; <boolexpr> ’]’

<generators> ::= <generator> {; <generator> }

<generator> ::= <var> ← <list_expr>
| (<var>, var>) ← <list_expr>

List Comprehension Examples

fun map(f, l) = [f(x) : x ← l]

Contrast: fun map(f, []) = []
| map(f, h::t) = f(h) :: map(f, t)

fun allpairs(l1, l2) = [(x, y) : x ← l1 ; y ← l2]

Contrast: fun allpairs([], l2) = []
| allpairs(h::t, l2) = pairup(h,l2) @ allpairs(t,l2)

List Comprehension Examples

Builtin generator:
[m .. n] – integers from m to n

fun divisors(n) = [d : d ← [2..n] ; n mod d == 0]

Contrast: fun divisors(n) = filter(gen(2,n), n);
fun gen(m,m) = [m]
| gen(n,m) = n :: gen(n+1, n)
fun filter([],n) = []
| filter(h::t,n) = if (n mod h== 0)
then h :: filter(t,n)
else filter(t,n)

Python Syntax: List Comprehensions

range(10) ➔ [1,...,10]

range(10,20) ➔ [10, 11, ... 20]

def divisors(n): return
[i for i in range(2,n)
if n%i == 0]

Quick Sort using List Comprehension

fun sort([]) = []
| sort(h::t) = sort([x : x ← t ; x <= h])
@ [h] @
sort([x : x ← t ; x > h])

Note: [x : x ← t ; x <= h] and [x : x ← t ; x > h] require two traversals over the list t.

Set vs List Comprehensions

Set Theory: (i) { x | x ∊ S /\ property(x) }
(ii) { x | property(x) }

Logic Programming (list and set comprehensions):
(i) |?- member(X,L1), property(X).
(ii) |?- setof(X, anc(bob, X), L).

Functional Programming (only list comprehensions):
(i) [x | x ← L1 ; property(x)]
(ii) not expressible directly

List Comprehensions combine well with with Lazy Evaluation

fun primes = sieve_all([2..])

fun sieve_all(h::t) = h :: sieve_all([n : n ← t ;
n mod h > 0]
)

Listlessness better than Laziness ☺

Sometimes, modularity is achieved by generating unnecessary intermediate lists. E.g.,

map(square) o map(succ) [1,2,3,4]
➔* map(square) [2,3,4,5]
➔* [4,9,16,25]

Faster execution results if we define:

```
fun g(x) = square(succ(x));
```

And perform: (map g) [1,2,3,4]

ML Types vs Modules

Types	~	**Signatures**
Values	~	**Structures**
Functions	~	**Functors**

ML has two distinct concepts: types and signatures.
In Java, a signature is a specification of a type.

Signature for BST

```
signature BST =
sig
  type item
  type bstree
  val make: item → bstree
  val insert: item * bstree → bstree
  val max: bstree →  item
  val min: bstree →  item
end;
```


Signature of BST elements

```
signature ELEMENT =
sig
  type element
  val eq: element * element →  bool
  val lt: element * element →  bool
end;
```


Encapsulating Element Implementation Details

```
structure Int: ELEMENT =
struct
  type element = int
  fun eq (x, y: element)  = x = y
  fun lt (x, y: element) = x < y
end;
```

```
structure String: ELEMENT =
struct
  type element = string
  fun eq (x, y: element)  = x = y
  fun lt (x, y: element) = x < y
end;
```


Specifying Type Constraints and BST Implementation

```
functor BSTree(Elem:ELEMENT): BST =
struct
type item = Elem.element
val eq      = Elem.eq
val lt       = Elem.lt
datatype bstree=leaf |
                        node of item * bstree * bstree;
. . . see next slide ...
end;
```

```
fun make(n) = node(n,leaf,leaf);

fun insert(x, leaf) = node(x,leaf,leaf)
  | insert(x, tr as node(n, t1, t2)) =
         if eq(x,n) then tr else
         if lt(x,n) then
                  node(n,insert(x,t1),t2)
                  else node(n,t1,insert(x,t2))

fun min(node(n,leaf,_)) = n
 |  min(node(n,t,  _))   = min(t);

fun max(node(n,_,leaf)) = n
 |  max(node(n,_,t  )) = max(t);
```


Using Functors

structure IntBSTree = BSTree (Int);
structure StringBSTree = BSTree (String);

```
fun test1() =
     let open IntBSTree;
         val h1 = make(21);
         val h2 = insert(39, h1);
        ...
         val h5 = insert(47, h4)
      in
         (min(h5), max(h5))
     end;
```


Data Specification

Two important components of a datatype specification:

1.Signature (interface)
2.Axioms (meaning)

PLs normally support only signatures, but axioms are necessary for a complete specification of the type.

Stack and Queue

(signatures are isomporphic)

signature STACK {
type stack
exception toperr, poperr
emptystack: stack
push: int x stack → stack
top: stack → int
pop: stack → stack
isempty: stack → bool
}
	signature QUEUE{
	type queue
	exception fsterr, remerr
	emptyqueue: queue
	ins: int x queue →queue
	front: queue → int
	remove: queue → queue
	isempty: queue → bool
	}

Need for Axioms

Consider stack push(10, push(20, emptystack)). The value 10 is at the top of the stack.

Consider queue ins(10, ins(20, emptyqueue)). The value 20 is at the front of the queue.

Thus, the LIFO vs FIFO difference is nowhere captured in the definition of the signatures! This is why datatype axioms are a necessary addition to the signatures.

Stack Axioms

top(emptystack) = $\perp$ undefined
top(push(s, x)) = x

pop(emptystack) = $\perp$
pop(push(s, x)) = s

isempty(emptystack) = true
isempty(push(s, x)) =false

Queue Axioms

if

front(emptyqueue) = ⊥
front(ins(x,emptyqueue)) = x
front(ins(x,q)) = front(q) ← not isempty (q)

remove(emptyqueue) = ⊥
remove(ins(x,emptyqueue)) = emptyqueue
remove(ins(q, x)) = ins(remove(q), x)
← not isempty(q)

isempty(emptyqueue) = true
isempty(ins(s, x)) =false

The Set datatype

```
signature SET =
sig
    type item, set
    val empty      : set
    val single     : item → set
    val union      : set * set → set
    val member     : item * set → bool
    val intersect  : set * set → set
    val subset     : set * set → bool
    val equal      : set * set → bool
end;
```


Set datatype axioms

member(x, empty) = false
member(x, single(y)) = equal(x, y)
member(x, union(s1,s2)) = member(x,s1) or
member(x,s2)

subset(empty, s) = true
subset(single(x), s) = member(x,s)
subset(union(s1,s2), s) = subset(s1,s) and
subset(s2,s)

equal(s1, s2) = subset(s1,s2) and subset(s2,s1)

Datatype Correctness

We can establish the correctness of a datatype implementation with respect to a set of axioms by demonstrating two properties:

1. Representation Invariant
2. Inherent Invariant

(1)➔ every abstract value has a concrete representation

(1) ➔ datatype axioms are satisfied by the implementation

ML Module for Set Datatype

```
functor Set(Item : ITEM) : SET =
struct
    type item = Item.item;
    val eq    = Item.equal;
    datatype set = rep of item list;

    val empty = rep([]);
    fun single(e) = rep([e]);
    fun union(rep(l1), rep(l2)) = rep(l1@l2);

    fun member(e, rep([]))   = false
      | member(e, rep(h::t)) = eq(e,h)  orelse
                                   member(e, rep(t));
    ...
end;
```

Representation Correctness: Example

Datatype constructors: empty, single, union

Operation implementations:

```
val empty = rep([]);
fun single(e) = rep([e]);
fun union(rep(l1), rep(l2)) = rep(l1@l2);
```

The representation invariance follows from the fact that the append (@) of any two lists exists.

Inherent Correctness: example

The member function axioms:

member(x, empty) = false
member(x, single(y)) = equal(x, y)
member(x, union(s1,s2)) = member(x,s1) or member(x,s2)

The member function implementation (ML):

```
fun member(e, rep([]))    = false
  | member(e, rep(h::t)) = eq(e,h)  orelse
                           member(e, rep(t));
```

The member constructor implementation (ML):

```
val empty = rep([]);
fun single(e) = rep([e]);
fun union(rep(l1), rep(l2)) = rep(l1@l2);
```

Inherent Correctness (cont'd)

Substitute constructor definitions in member axioms, we must show that:

1. member(x, rep([])) = false
2. member(x, rep([y])) = eq(x, y)
3. member(x, rep(l1@l2)) = member(x, rep(l1)) orelse member(x, rep(l2))

Based upon the implementation, we can assume:

member(e, rep([])) = false
member(e, rep(h::t)) = eq(e,h) orelse member(e, rep(t));

Properties (1) and (2) are immediate, hence we focus on (3).

Inherent Correctness (cont'd)

We must show that:

member(x, rep(l1@l2)) = member(x, rep(l1)) orelse member(x, rep(l2))

Proof by induction on l1:

Base Case, l1 = []: Easy to see that LHS = RHS

Induction Hypothesis: Assume the equality holds for

member(x, rep(t@l2)) = member(x, rep(t)) orelse member(x, rep(l2))

Induction Step: Show the equality holds for

member(x, rep(h::t @ l2)) = member(x, rep(h::t)) orelse member(x, rep(l2))

Inherent Correctness (cont'd)

To show that:

member(x, rep(h::t @ l2)) = member(x, rep(h::t)) orelse member(x, rep(l2))

Note: h::t @ l2 = h :: (t@ l2) – from definition of @

LHS = eq(x, h) orelse member(t, rep(t@l2))
RHS = eq(x, h) orelse member(x, rep(t)) orelse member(x, rep(l2))

Now LHS = RHS from the induction hypothesis, since:

member(x, rep(t@l2)) = member(x, rep(t)) orelse member(x, rep(l2))