CSE 505

Lecture #11
October 8, 2012

ML Type System

```
τ ::=   int | real | bool | string | unit
        | τ *  ...*  τ
        | { f1:τ,  ..., fk : τ}
        | τ list
        | τ  → τ
        |  ' id
        |  '' id
        | user_defined_concrete_type
        | user_defined_abstract_type
```


The ML Abstract Type

```
abstype [parameters]  type-id  = representation-type
 with  [exception1   ... exceptionk ]
       < implementation of operation1 >
        ...
       < implementation of operationn >
end
```

Abstract Types are defined behaviorally, i.e., in terms of relevant operations of the type.

How to implement an ML abstype

1. Choose a representation, e.g. stack → list
2. Implement operations.

```
abstype 'a stack = rep of 'a list
with
        val emptystack = rep([]);
        fun push(x, rep(list)) = rep(x::list);
        ...
end;
```


Internally, it is rep([])

```
- emptystack;
val it = - : 'a stack
```

Representation not visible

```
- val stk2 = push("apple", push("fig", emptystack));
val stk2 = - : string stack
```

Internally, it is rep(["apple", "fig"])

Implementing ML abstype (cont'd)

1. Choose a representation
2. Implement operations.
3. Declare exceptions

```
abstype 'a stack = rep of 'a list
with   exception poperror;
       exception toperror;
              ... define push and pop ...
       fun pop(rep([]))    = raise poperror
          | pop(rep(_::t)) = rep(t);
       fun top(rep([]))  = raise toperror
          | top(rep(h::_)) = h;
      end;
```

Need for Exception Handling

Why is the following code not OK from the standpoint of types?

```
fun top(rep([]))     = “stack is empty!”
   | top(rep(h::_)) = h;
```

Answer: The type of the stack is forced to become “string stack”, i.e., it can work on only on strings!

Exception Handling

raise exception_name [(arguments)]

handle exception-name$_1$ (pattern$_1$) = expr$_1$
| exception-name$_2$ (pattern$_2$) = expr$_2$
...
| exception-name$_n$(pattern$_n$) = expr$_n$

expr handle handler

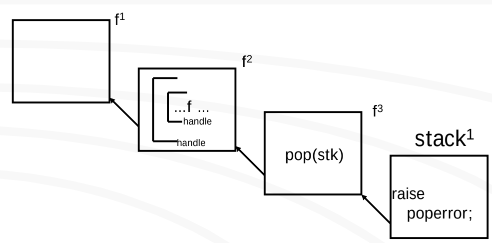
Exception Handling - Remarks

- Strong Typing: Given
  expr handle handler
  The type of result returned by handler must be the same as that returned by expr

- Dynamic-cum-Static Scoping: When an exception is raised, the handler is searched by first looking in the immediate lexical context, and if none is found proceeding up the dynamic-link chain, etc.

Exception Handling

Uses both static and dynamic scoping:



Exception Handling for Stacks

```
- val stk2 = push("apple", push(“fig", emptystack));
val stk2 = - : string stack
- top(pop(pop(stk2)));
        uncaught exception toperror
-top(pop(pop(stk2)))
        handle toperror => "I caught toperror!";
val it = "I caught toperror!" : string
-top(pop(pop(pop(stk2))))
       handle poperror => 0
```

Type error!


```
abstype ’a stack = rep of ’a list
with
        exception poperror;
        exception toperror;
        val emptystack = rep([]);
        fun push(x, rep(list)) = rep(x::list);
        fun pop(rep([]))    = raise poperror
           | pop(rep(_::t)) = rep(t);
        fun isempty(rep([]))  = true
           | isempty(rep(_::_)) = false;
        fun top(rep([]))    = raise toperror
           | top(rep(h::_)) = h;
        fun show(rep(list)) = list;
end;
```

Stack Interface (inferred by ML)

```
type 'a stack
exception poperror
exception toperror
val emptystack = - : 'a stack
val push = fn: 'a * 'a stack  → 'a stack
val pop = fn: 'a stack →  'a stack
val isempty = fn: 'a stack →  bool
val top = fn: 'a stack →  'a
val show = fn: 'a stack →  'a list
```


Polymorphic Ordered Structures

Example:

```
datatype 'a olist = onil | ocons of 'a * 'a list
```

While this does not fully capture the requirements for an ordered list, it provides the starting point for an abstract data type definition.

Let's see how an ordered list ADT can be defined ...


```
abstype 'a olist = olistrep of  'a list
with  exception empty_olist;
       val  onil = olistrep([]);
       fun ocons(e, olistrep(list)) =
              let  fun ins(x, [])   = [x]
                     | ins(x, y::t) = if x=y orelse x<y
                                      then x::y::t
                                      else y::ins(x,t)
              in   olistrep(ins(e,list))
              end;
       fun min(olistrep([]) = raise empty_olist
          | min(olistrep(h::_)) = h;
       ...
end;
```

Unresolved Overloaded Operator


```
abstype 'a olist = olistrep of
                       ('a list * {eq: 'a * 'a -> bool,
                                   lt: 'a * 'a -> bool })
with
exception empty_olist;
fun onil(ops) = olistrep([], ops);
fun ocons(e, olistrep(list, ops as {eq=feq, lt=flt})) =
      let  fun ins(x, [])   = [x]
             | ins(x, y::t) = if feq(x,y) orelse flt(x,y)
                              then x::y::t
                              else y::ins(x,t)
        in   olistrep(ins(e,list), ops)
      end;    ...
end;
```

pattern matching

Ordered List Interface (inferred by ML)

```
type 'a olist
exception empty_olist
val onil    = fn : { eq: 'a * 'a -> bool,
                     lt: 'a * 'a -> bool }   -> 'a olist
val ocons = fn : 'a * 'a olist -> 'a olist
val min    = fn : 'a olist -> 'a
...
```



```
-  fun f1(x,y:int) =  x=y;
       val f1 = fn : int * int -> bool

- fun f2(x,y:int) =   x<y;
       val f2 = fn : int * int -> bool

- val o1 = onil({eq = f1, lt = f2});
       val o1 = - : int olist
- val o2 = ocons(10, ocons(30, ocons(20,
                                   ocons(40, o1))));
       val o2 = - : int olist
- min(o2);
       val it = 40 : int
```

Critique of ML Abstype

1. The abstype defines an implementation, not an interface.
2. The implementation details of an abstype cannot* entirely be encapsulated in the abstype.
3. The constraints on type parameters of the abstype are not explicitly given.

* Auxiliary type definitions cannot be written inside the abstype.

ML Solution

Interface	→ **`signature`**
Encapsulation	→ **`structure`**
Type Constraints	→ **`functor`**

We will return to this topic later.

Higher Order Functions in ML

```
map(f, []) = []
map(f, h::t) = f(h) :: map(f, t)
```

What is the inferred type for 'map'?

* The input list is type: 'a list

* The type of f is: 'a → 'b

* The type of map: ('a → 'b) * 'a list → 'b list

The 'reduce' function

```
reduce(f, b, []) = b
reduce(f, b, h::t) = f(h, reduce(f, b, t))
```

What is the inferred type for 'reduce'?

* The input list is type: 'a list

* The type of f is: 'a * 'b → 'b

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

Using Map and Reduce

```
- map(sqrt, [1.0, 4.0, 9.0, 16.0]);
  val it = [1.0, 2.0, 3.0, 4.0] : real list

- map(length, [[1,2,3], [], [1,2]]);
  val it = [3,0,2] : int list

- reduce(sum, 0, [10, 20, 30, 40]);
  val it = 100 : int
```


Returning Functions as Results

Another way to define map:

```
map(f) =  let fun g([]) = []
                | g(h::t) = f(h) :: g(t)
          in g
          end;
```

Type of map: ('a → 'b) → ('a list → 'b list)

This allows us to give the arguments of map one at a time, hence called "partial parameterization".

Partial Parameterization (syntax)

Some languages allow:

```
map  f  []     =  []
map  f  h::t  =  f(h) ::  (map  f  t)
```

as short-hand for:

```
map(f) =  let fun g([]) = []
                  | g(h::t) = f(h) :: g(t)
          in g
          end;
```

Another way to define reduce

```
reduce(f, b) =  let fun g([]) = b
                      | g(h::t) = f(h, g(t))
                in g
                end;
```

Type of reduce: (‘a → ‘b) * ‘b → (‘a list → ‘b)

This allows “partial parameterization” of reduce.

Using Map and Reduce

- map(sqrt);

  val it = - : real list -> real list

- map(length);

  val it = - : ‘a list list → int list

- reduce(sum, 0);

  val it = - : int list → int

Lazy Evaluation and Higher Order Functions

Lazy Evaluation is similar to call-by-name in that we do not evaluate a function’s arguments before entering the function body.

Benefits: Enhances modularity, supports conceptually infinite data structures, e.g. game trees

Two types of laziness:

1. Constructor laziness
2. Full laziness

Generating Infinite Lists

```
fun numsfrom(n) = n :: numsfrom(n+1);
```

Above is nonterminating under ML evaluation:

The call numsfrom(0) will result in an infinite loop in ML, although it stands for the infinite list [0, 1, 2, ...]

numsfrom(0) → 0 :: numsfrom(1)

→ 0 :: 1 :: numsfrom(2)

→ 0 :: 1 :: 2 :: numsfrom(3)

→ ... nonterminating ...

Constructor Laziness Principle

Do not evaluate the arguments of a constructor until the associated structure is passed as argument to some function that selects (e.g., by pattern-matching) the components of the structure.

Lazy Evaluation - illustration

```
fun numsfrom(n) = n :: numsfrom(n+1);

fun printlist(0, ilist) = ()
  | printlist(n, h :: ilist) =   let val _ = print(h)
                                  in printlist(n-1, ilist)
                                 end;
```

print 0

- printlist(10, numsfrom(0)));

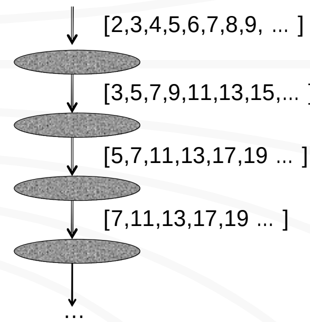
➔ printlist(10, 0 :: numsfrom(1));

➔ printlist(9, numsfrom(1));

➔ printlist(9, 1 :: numsfrom(2));

➔…

Sieve of Eratosthenes

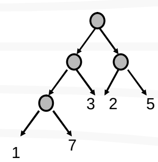


Sieve Program

fun primes = sieve_all(numsfrom(2))

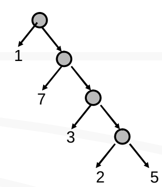
```
fun primes =  sieve_all(numsfrom(2))

fun sieve_all(p::t) =   p :: sieve_all(sieve(t, p))

fun sieve(h::t, p) =  if h mod p = 0
                      then sieve(t, p)
                      else h :: sieve(t, p);
```


The "Same Fringe" Problem



Tree1

Tree1 and Tree2 have the same "fringe," [1,7,3,2,5]



Tree2

Same Fringe – Modular Def'n

```
datatype 'a tree   =  leaf of 'a  |  node of 'a tree * 'a tree;

fun tree2list(leaf(x)) = [x]
   | tree2list(node(t1,t2)) = tree2list(t1) @  tree2list(t2) ;

fun samefringe(t1, t2) =  tree2list(t1) = tree2list(t2);
```

Note: tree2list is a good approach only with lazy evaluation.

```
Note:  fun  []     @ l2 = l2
            | h::t @ l2 = h :: (t @ l2)
```

Illustration of samefringe execution on next page.

```
samefringe(node(leaf(1),leaf(2))), node(leaf(2),leaf(1)))

=> tree2list(node(leaf(1),leaf(2))) =
     tree2list(node(leaf(2),leaf(1)))

=> tree2list(leaf(1)) @ tree2list(leaf(2)) =
                                   tree2list(node(leaf(2),leaf(1)))

=> [leaf(1)] @ tree2list(leaf(2)) =
                                   tree2list(node(leaf(2),leaf(1)))
=> leaf(1) :: ([] @ tree2list(leaf(2))) =
                                   tree2list(node(leaf(2),leaf(1)))

=>* leaf(1) :: ([] @ tree2list(leaf(2))) =
      leaf(2) :: ([] @ tree2list(leaf(1)))

=> false  -- without evaluating remaining expressions
```

Benefits of lazy evaluation

Lazy evaluation enhances program modularity: generation of tree elements is decoupled from the test for equality.

(Contrast solution with coroutine approach.)

Constructor-laziness avoids unnecessary list generation, and allows termination as soon as a disagreement is found in the two lists.

Simulating lazy evaluation using higher-order functions

```
fun numsfrom(n) = let fun thk1() = n
                      fun thk2() = numsfrom(n+1)
                  in (thk1, thk2)
                  end

fun printlist(0, _) = []
  | printlist(n,  (thk1, thk2)) = let val _ = print(thk1())
                                  in printlist(n-1, thk2())
                                  end;
```

Note: numsfrom is returning a pair of functions as result!

Is numsfrom well-typed?

```
fun numsfrom(n) = let fun thk1() = n
                      fun thk2() = numsfrom(n+1)
                  in (thk1, thk2)
                  end;
```

Circular dependency between numsfrom and thk2:

numsfrom: int → (unit → int) * (unit → ??)
thk2: unit → ??

Let us ignore this issue for the time-being.

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

front(emptyqueue) = $\perp$
front(ins(x,emptyqueue)) = x
front(ins(x,q)) = front(q) ← not isempty (q)

if

remove(emptyqueue) = $\perp$
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))