CSE 505

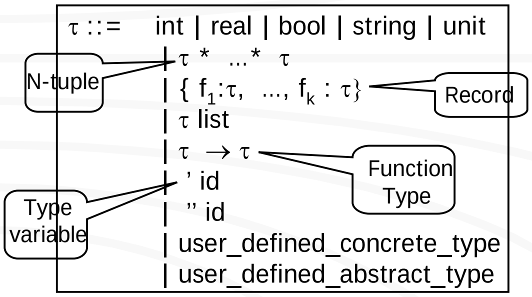
Lecture #10

October 3, 2012

Characteristics of ML

- Strongly typed language (with type inference):
  - (parametric) polymorphic types
  - concrete type
  - abstract types
- Higher-order functional language:
  - expression-oriented (like Lisp)
  - rule-based definitions, with pattern matching
  - higher-order functions (output can also be function)
  - static scoping, with nested function definitions
- Modular language:
  - signatures, structures, and functors

ML Type System



τ ::= int | real | bool | string | unit

| τ * ...* τ

| { f_1:τ, ..., f_k : τ}

| τ list

| $\tau \rightarrow \tau$

| ' id

| '' id

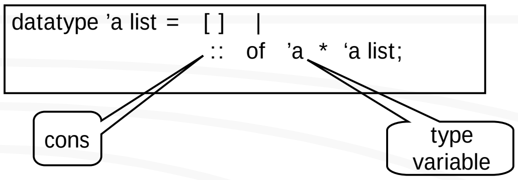
| user_defined_concrete_type

| user_defined_abstract_type

Type Inference - Remarks

In the absence of overloaded operators, types for all identifiers in an ML program can be unambiguously determined without any type declarations (assuming no type errors).

Polymorphic Datatypes



```
datatype ’a list =   [ ]   |
                      ::  of  ’a  *  ‘a list;
```

Constructors:

[ ] : ’a list

:: : ’a * ’a list → ’a list

List Constants (literals)

[1, 2, 3] ≡ 1 :: 2 :: 3 :: [ ]

[“abc”, “def”] ≡ “abc” :: “de” :: [ ]

[[1, 2],[3]] ≡ ((1::2::[]) :: (3::[]) :: [])

[1, “apple”, 3.14] → badly typed list

[1, [2], [1,2,3]] → badly typed list

Polymorphic Functions

```
fun length([ ])   = 0
   | length(h::t) = 1 + length(t);
```

What is the type of length?

- Output Type: int, since 0:int, and this agrees with the second case 1 + length(t);

- Input Type: the terms [ ] and h::t have types ‘a list and ‘b list, but a=b since they must be compatible.

Hence the type of length: ‘a list → int.

Polymorphic Functions (cont’d)

```
fun member(x, [])      = false
     | member(x, h::t) =  if x=h
                          then true
                          else member(x,t)
```

“a * “a list → bool

Note: The following is incorrect:

Nonlinear Pattern Disallowed!

```
fun member(x, [])      = false
   | member(x, x::t) =  true
   | member(x, y::t) =  false
```

More examples

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

‘a list * ‘a list ->‘a list

Mystery type analysis:

- input and output types must be lists;

- h @ mystery(t) indicates that h must be a list;

mystery: ‘a list list -> ‘a list

A simple tree datatype

```
datatype ’a tree=
           leaf of  ’a
         | node of ’a tree  * ’a tree
```

Sample tree literals:

leaf(true)

node(leaf(1), leaf(2))

node(node(leaf(3.5), leaf(4.5)), leaf(1.5))

node(node(leaf("Ada"), leaf(“C")),
      node(leaf(“Java"), leaf(“Prolog")))

‘a tree → int

anonymous variable

```
fun depth(leaf(_)) = 0
   | depth(node(t1, t2)) =
         let val d1 = depth(t1);
             val d2 = depth(t2)
         in  if d1>d2
             then 1+d1
             else 1+d2
         end;
```

Pattern Matching Examples

match([1, 2, 3], h::t) ➔ true
    h = 1,
    t = [2,3]

match([[1,2], [3,4], [5]], (h1::t1) ::t2) ➔ true
    h1 = 1
    t1 = [2]
    t2 = [[3,4], [5]]

match(node(leaf(1), leaf(2)), node(leaf(x), t2)) ➔ true
    x = 1
    t2 = leaf(2)

Pattern Matching in ML

match(c1, c2) ➔ true, if c1 and c2 are constants and c1 = c2

match(t, v) ➔ true, if v is a var; also bind v ← t

match(h1::t1, h2::t2) ➔ match(h1, h2) and match(t1, t2)

match(c(t1,..., tn), c(u1, ..., un) ➔ match(t1, u1) and ... match(tn, un)

The Quicksort Algorithm

```
fun qsort([])   =  []
   | qsort(h::t) =
                     let val (l, r) = partition(h, t)
                      in qsort(l) @ [h] @ qsort(r)
                   end;
```

```
fun partition(pivot, [ ]) = ([ ], [ ])
   | partition(pivot, h::t) =
         let val (l, r) = partition(pivot, t)
          in  if h < pivot
                  then (h::l,  r)
                  else (l,  h::r)
         end;
```

append

unresolved overloaded op'r

```
fun partition(pivot, [ ]) = ([ ], [ ])
   | partition(pivot:int, h::t) =
        let val (l, r) = partition(pivot, t)
         in  if h < pivot
              then (h::l,  r)
              else (l,  h::r)
        end;

fun qsort([])   =  []
   | qsort(h::t) = let val (l, r) = partition(h, t)
                    in qsort(l) @ [h] @ qsort(r)
                   end;
```

must specify type

int list → int list

Type Inference by solving Type Equations – Example 1

```
fun f(x, y) = (x+2, y*3.5);
```

Type Equations:

$T_f = T_x * T_y \rightarrow T_r$

$T_r = T_{r1} * T_{r2}$

$T_x * T_2 \rightarrow T_{r1} = int * int \text{ -> } int$

$T_y * T_{3.5} \rightarrow T_{r2} = real * real \rightarrow real$

Type Equations – Example 2

```
fun h(x, y) = if x then y else h(y,x)
```

Type Equations:

$T_h = T_x * T_y \rightarrow T_r$

$bool * T_r * T_r \rightarrow T_r = T_x * T_y * T_r \rightarrow T_r$

$T_h = T_y * T_x \rightarrow T_r$

Type Equations – Example 3

```
fun app([], x) = x
   | app(h::t, y) = h::app(t, y)
```

Type Equations:

$T_{app} = T_1 * T_2 \rightarrow T_r$

$T_1 * T_2 = A \text{ list} * T_x = B \text{ list} * T_y$

$T_h * T_t = B * B \text{ list}$

$T_r = T_x = C \text{ list} = D \text{ list}$

$T_h * T_r = D * D \text{ list}$

⇒ A = B = C = D

Solving Type Equations

Type Equations in ML are solved by an algorithm called “unification,” a generalized form of pattern matching.

Unification will give the “most general polymorphic type” if there are no type clashes.

From Concrete to Abstract Types

Concrete types are defined structurally whereas abstract types are defined behaviorally, i.e., in terms of relevant operations of the type.

Why Abstract Data Types?

1. All uses of constructors not meaningful.
2. All uses of pattern-matching not meaningful.
3. Equality is not definable as structural identity.

Criteria #1 - Constructors

Ordered List:

```
datatype ’a  olist = onil  |  ocons  of ’a * ’a  olist;
```

In order:

```
      ocons ("apple",  ocons("orange", onil))
ocons (5, ocons(10, ocons(30, ocons(200,onil))))
```

Out of order:

```
             ocons(10, ocons(5, onil))
ocons(5, ocons(40, ocons(3, ocons(2000, onil))))
```

Ordered data structures
should NOT
be defined as concrete types.
They should be defined
as abstract types.

Criteria #2 – Pattern Matching

Stacks

```
datatype ’a  stack = empty | push  of ’a * ’a  stack;
```

All uses of constructors are meaningful:

```
       push("apple", push("orange", empty))
push(10, push (4, push(30, push(200,empty))))
```

However, we can access any element of stack:

```
fun top(push(x, _)) = x;
fun next_to_top(push (x1, push (x2, s))) = x2;
```

Data with restricted access should NOT be defined as concrete types. They should be defined as abstract types.

Criteria #3 – Equality

Sets

```
datatype 'a set=empty | single of 'a
                      | union of 'a set *'a  set;
```

All uses of constructors are meaningful:

```
     union(single( [10]), single( [10,20] ))
union(single(3),union(single(3), single(20) ))
```

However, equality is not 'structural identity':

```
union (single(3),union(single(3), single(20)))
<> union(single(3), single(20)).
```

Concrete types should not be used to define data for which equality is not expressible as structural identity. Such data types should be defined as abstract types.

The ML Abstract Type

abstype [parameters] type-id = representation-type
with [exception$_1$ ... exception$_k$]
< implementation of operation$_1$ >
...
< implementation of operation$_n$ >
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$_n$

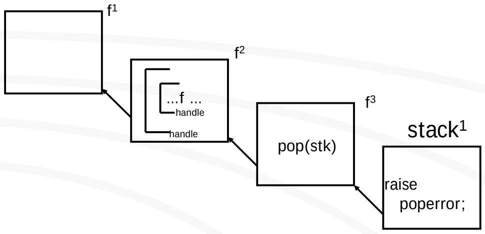
expr handle handler

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
needs 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`**