ML

- functional programming style
- static typing
- type inference
- pattern matching
- polymorphism
- exceptions
- first-class modules

Basic datatypes

- unit (like void)

```
- ();
() : unit
```

- bool, int, real, string

```
- 7 > 5;
true:bool
- ~24+1;
~23:int
- 3.14159;
3.141519:real
- "hi there"
"hi there":string
```

- lists

```
- 3 :: 4 :: nil;
[3,4] : int list
- ( 3 :: nil) :: ( 4 :: 5 :: nil) :: nil;
[[3],[4,5]]: int list list
- [3, "hi there"];
... error ...
- nil;
...error (ML now doesn't like untyped lists)
```


Basic datatypes (cont)

- tuples

```
- ("wow",6);
("wow",6): string * int
- (1.0, 2, true, "s");
(1.0, 2, true, "s"): real*int*bool*string
```

- records

```
- {name="J",age=1};
{name="J",age=1}:{name:string,age:int}
- {name="J",age=1}={age=3 mod 2,name="J"};
true:bool
```


Bindings

```
- val x = 23;
val x = 17 : int
- local val x = 10
  in
    val u = 230 * x
    val z = (true,22 + x)
  end;
val u = 2300 : int
val z = (true,32) : bool * int
```

Functions

```
- fun succ(x) = x + 1;
val succ = fn : int -> int
- fn x=>x + 1;
fn : int -> int
- succ 8;
9 : int
- (fn x=>x+1)(8);
9 : int
```

Pattern Matching

Means of decomposing compound values

```
- val x=(false,17);
...
- val (a,b) = x;
val a = false: bool
val b = 17:int
...
- val (true,w) = x;
exception: nonexhaustive binding failure

- val s=["1", "2", "3"];
...
- val hd::tl = s;
val hd = "1" : string
val tl = ["2", "3"] : string list
...
- val hd::_ = s;
val hd = "1" : string
```


Pattern Matching with Functions

Means of defining function behavior by cases:

```
- fun fib(0) = 1
  | fib(1)   = 1
  | fib(n)   = fib(n-1) + fib(n-2);
val fib = fn : int -> int

- fun is_empty(nil) = true
  | is_empty(_::_)  = false;
val is_empty = fn : 'a list -> bool

- fun append(nil,l)= l
  |  append(hd::tl,l) = hd :: append(tl,l);
val append = fn : ('a list * 'a list)->'a list
```

Something is special about these last two functions...

Polymorphic Functions

Some functions are general

- can be used on arguments of different types

E.g.

- `length`
- `append`
- `is_empty`
- `first_of_pair`

In Scheme, dynamic typing allows these to be defined easily

How to define such functions in a statically-typed language?

- can't in C (or have to use casts)
- in ML: allows functions to have polymorphic types

Polymorphic types

(Note: this is a programming languages Concept.)

A polymorphic type contains one or more **type variables**

E.g.

```
'a list
'a * 'b
('a * 'b) list * 'a -> 'b
```

To make a regular type, replace type variables with regular types

- each occurrence of a type variable must be replaced with same type

When calling a polymorphic function, caller knows real type

- replace type variable with regular type

`length([3,4,5])`
replaces `'a` with `int`

`length([(3.0,"hi"),(2.0,"there")])`
replaces `'a` with `int * string`

Polymorphism versus Overloading

A polymorphic function is a function with a polymorphic type.

```
- fun append(nil,l)= l
  | append(hd::tl,l) = hd :: append(tl,l);
val append = fn : ('a list * 'a list)->'a list
```

There is just one append function, which can be applied to many (monomorphic) types.

In contrast, an overloaded function (another Concept) has several different definitions, and the compiler or the runtime decides which function is mean using the types of the arguments.

In ML this is done at compile time. Also, only certain builtin functions are overloaded; you can't overload user-defined functions (at least as far as I could figure out).

Example:

+ is overloaded -- there are actually two different plus functions, one for ints and one for reals.

We can also overload other kinds of names.

Type Inference

System infers types of expressions automatically. Works most of the time:

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

For weird reasons known only to the ML designers, in its current version ML assumes + means integer +.

What if we wanted this to work on reals?

```
- f(3.0,4.0);
Error: operator and operand don't agree [tycon
mismatch]
```

Then you have to give it a hint:

```
- fun f(x:real, y) = x + y;
val f = fn : real * real -> real
```


Exceptions

Means of handling unusual cases

```
- exception Head;
exception Head
- fun head(nil)  = raise Head
  | head(x::_)   = x;
val head = fn : 'a list -> 'a
...
- head(nil : int list);
Failure: Head
```

Handling exceptions:

```
- fun head2(l) = head(l) handle Head => 0;
val head2 = fn : int list -> int
- head2(nil);
0 : int
```

Can even have arguments:

```
- exception IOError of int

... raise IOError(-3) ...
... handle IOError(code) ...
```


Polymorphic type inference

Type inference for a construct proceeds as follows:

- assign each declared variable a fresh type variable
  - result of function is implicit variable
  - share argument and result type variables across function cases
  - references to let-bound polymorphic identifiers get distinct type variables
- each subexpression in construct places constraints on types of its operands
- solve constraints

Constraint solving can refine types:

- e.g. `'a` ⇒ `'b list`

For overloading, constraint solving must resolve overloading to particular instance

For = operation, refine `'a` to `''a` type

Overconstrained (unsatisfiable constraints) ⇒ type error

Underconstrained ⇒ OK; polymorphic bindings allowed

Properties of ML's type system

Hindley-Milner type system/inference

- allows code to depend on exact type of value, or have no dependencies on value (universal polymorphism)
- no constrained polymorphism, other than equality types...
- only let-bound polymorphism

Type inference yields **principal type** for expression

- i.e. unicity of typing

Complexity of type inference: DEXPTIME !

Concrete user-defined datatypes

Users can define their own (polymorphic) data structures

Simple example: ML's version of enumerated types

```
datatype sign = Positive | Zero | Negative;
```

Introduces constants

- can be used in patterns

```
fun signum(x) =
  if x > 0 then Positive
  else if x = 0 then Zero
  else Negative;

fun signum_value(Positive) = 1
|   signum_value(Zero)     = 0
|   signum_value(Negative) = ~1;
```


Variant records/tagged unions in ML

Each component of a datatype declaration can have info

- constructors act as functions to create values with that tag
- can be used in patterns to take apart values of a tag

Example:

```
datatype Sexpr =
  Nil |
  Integer of int |
  Symbol of string |
  Cons of (Sexpr * Sexpr);

(* '(3 hi) *)
val lst =
  Cons(Integer(3), Cons(Symbol("hi"), Nil));

fun first(Nil) = Nil
|   first(Cons(hd,_)) = hd;
fun rest (Nil) = Nil
|   rest (Cons(_,tl)) = tl;
```

Supports bounded, planned heterogeneity in ML

No abstraction or encapsulation, though

Recursive user-defined data types

Example: binary search trees of integers

```
datatype int_tree =
  Empty |
  Node of int * int_tree * int_tree;

(* insert: int_tree * int -> int_tree *)
fun insert(Empty, x) = Node(x,Empty,Empty)
|   insert(n as Node(y,t1,t2), x) =
    if x = y then n
    else if x < y then
      Node(y, insert(t1,x), t2)
    else
      Node(y, t1, insert(t2,x));

(* lookup: int_tree * int -> bool *)
fun lookup(Empty,x) = false
|   lookup(Node(y,t1,t2), x) =
     if x = y then true
     else if x < y then
       lookup(t1, x)
     else
       lookup(t2, x);
```

Cannot be polymorphic, though

Polymorphic user-defined data types

Polymorphic tables:

```
datatype (''Key,'Value) table =
  Table of (''Key * 'Value) list;

exception NotFound;
fun lookup(Table(assocs),key) =
  let fun lookup_help(nil,key) =
            raise NotFound
        | lookup_help((k,v)::assocs,key)=
            if key = k then v
            else lookup_help(assocs,key)
  in lookup_help(assocs,key) end;

fun store(Table(assocs),key,value) =
  Table((key,value)::assocs);
```


Streams in ML

Represent a stream cons cell as a pair of a head value and a function which will return the next in line

```
datatype 'a stream =
  stream of 'a * (unit -> 'a stream);

fun cons_stream(x,f) = stream(x,f);
fun hd_stream(stream(x,f)) = x;
fun tl_stream(stream(x,f)) = f();

fun ints_from(x) =
  cons_stream(x, fn() => ints_from(x+1));
val nats = ints_from(0);

fun map_stream(g,s) =
  cons_stream(
     g(hd_stream(s)),
     fn() => map_stream(g, tl_stream(s)));

val squares = map_stream(fn(x)=>x*x, nats);
```


References

Allow side-effects through explicit reference values (first-class locations):

- `ref: 'a → 'a ref`
- `! : 'a ref → 'a`
- `(op :=) : 'a ref * 'a → unit`

```
- val v = ref 0;
val v = ref 0 : int ref
- v := !v + 1;
- !v;
1 : int
```

Arrays are indexed sequences of refs to things:

- `array: int * 'a → 'a array`
- `(op sub): 'a array * int → 'a`
- `update: 'a array * int * 'a → unit`
- `length: 'a array → int`
- `tabulate: int * (int → 'a) → 'a array`

Problems with polymorphic references

```
- fun I x = x;
val I = fn : 'a → 'a
- val fp = ref I (*NOT LEGAL *)
val fp = ref fn : ('a → 'a) ref
- (!fp true, !fp 5);
(true, 5) : bool * int
- fp := not;
```

hmmmm...

```
- !fp 5
```

CRASH!!!

Weakly polymorphic types

Restrict ref to disallow polymorphic instantiations:

- `ref: 'a1 → 'a1 ref`
- `array: int * 'a1 → 'a1 array`

`'a1` is a weak type variable:
it cannot be instantiated with any type
that contains a strong type variable (e.g. `'a` or `''a`)

```
ref I   (* illegal! *)
ref 3
ref (ref 3)
ref (I:bool→bool)
fun foo(r:'a1 ref) = ref r; foo (ref 3)
```

Exceptions must have monomorphic types, for similar reasons

Modules

`datatype` provides concrete user-defined data types,
with representation exposed (e.g. for pattern matching)

Modules support encapsulated data types and some kinds of polymorphism

- replace old `abstype` declaration

Overview:

- `structure` defines module implementation
- `signature` defines module interface,
hides other aspects of underlying `structure`
- `open` supports importing a module, for naming convenience
- `functor` supports parameterized module implementations

Structures

Package up any set of declarations into a structure:

```
structure Queue1 = struct
  type 'a T = 'a list; (* T is conventional type name *)
  (* constructors: *)
  val empty = nil;
  fun enq(q,x) = q @ [x]; (* @ is append *)
  (* accessors: *)
  exception empty_queue;
  fun head(x::q) = x
    | head(nil) = raise empty_queue;
  fun deq(x::q) = q
    | deq(nil) = raise empty_queue
end;
```

Access members of structure using dot notation:

```
- val q = Queue1.enq(Queue1.empty,3);
val q = [3] : int list
- val q2 = Queue1.enq(q,4);
val q2 = [3,4] : int list
- Queue1.head(q2);
val it = 3 : int
```


Importing

To avoid long qualified names, use `open`

```
- open Queue1;
(* should have lots of output, indicating enq et al. defined ... but
it looks like there is a compiler bug in this version of ML ...*)
- val q = enq(enq(empty,3),4);
val q = [3,4] : int list
- head(deq(q));
val it = 4 : int
```

Often, limit scope of `open` using `let` or `local`:

```
structure QueueExtension = struct
  local open Queue1 in
    fun enq2(q,x,y) = enq(enq(q,x),y);
    ...
  end;
end;


- QueueExtension.enq2(Queue1.empty,3,4);
val it = [3,4] : int list
```

Signatures

To achieve encapsulation, define a public external interface

```
signature QUEUE = sig
  type 'a T;
  val empty: 'a T;
  val enq: 'a T * 'a -> 'a T;
  exception empty_queue;
  val head: 'a T -> 'a;
  val deq: 'a T -> 'a T;
end;
```

Apply signature to restrict interface to structure:

```
structure Queue1:QUEUE = struct ... end;
```

Now `Queue1`'s can only be manipulated abstractly:

```
- Queue1.empty;
val it = [] : 'a Queue1.T
```


Holes in encapsulation

Signatures don't completely hide module implementation

Types defined using `type` are not hidden

```
- Queue1.empty = nil;
val it = true : bool;
```

- types created with `datatype` are hidden, though

Built-in = equality function operates over representation

- two values that abstractly are the same can be revealed to be different by using =

Various proposals to try to fix these in ML community

Functors

Can parameterize structures by other structures

```
functor QueueUser(Q:QUEUE) = struct
  ... Q.enq ... Q.deq ...
end;
```

- only know aspects of `Q` that are defined by `QUEUE`

Instantiate functors to build regular structures:

```
structure QU1 = QueueUser(Queue1);
structure QU2 = QueueUser(Queue2);
```


Functors for bounded quantification

Define a signature representing the operations needed

```
signature ORDERED = sig
  type T;
  val eq: T * T -> bool;
  val lt: T * T -> bool;
end;
```

Define quantified algorithms as elements of functors parameterized by required signature

```
functor Sort(O:ORDERED) = struct
  fun min(x,y) =
    if O.lt(x,y) then x else y;
  fun sort(lst) =
    ... O.lt(x, y) ...;
end;
```

Create specialized sorter by instantiating functor with appropriate operations

```
structure IntSort = Sort(struct
  type T = int;
  val lt = (op <):T * T -> bool; (* resolve overload *)
  val eq = (op =);
end);
IntSort.sort(tabulate(10, fn _ => random()));
```

Create nested, multiply parameterized functors:

```
functor PairOrder(
    structure First:ORDERED;
    structure Second:ORDERED):ORDERED = struct
  type T = First.T * Second.T;
  fun lt((x1,x2),(y1,y2)) =
    First.lt(x1,y1) andalso Second.lt(x2,y2);
  fun eq((x1,x2),(y1,y2)) = ...;
end;

structure IntStringSort = Sort(
  PairOrder(
    structure First = IntOrder;
    structure Second = StringOrder));
```


Signature "subtyping"

Signature specifies a particular interface

Any structure that satisfies that interface can be used where that interface is expected

- e.g. in functor application

Structure can:

- have more operations
- have more general/polymorphic operations
- have more details of implementation of types

than required by signature

Limitations

Structures and signatures are not first-class values

- must be named
- must be declared at top-level or nested inside another structure or signature

Structures can be parameterized only by other structures

- but this isn't really a limitation, since any value can be in the structure

Modules vs. ADTs

Abstract data types implicitly define a single type, with associated constructors, observers, and mutators

Modules can define 0, 1, or many types in same module, with associated operations over several types

ADTs are instantiated, but modules aren't

- functors can simulate instantiable modules

Modules more general than ADTs, but clumsier for common case