Polymorphic Types

Polymorphic functions are achieved using type variables.

- A type variable starts with a prime ‘.’
  - ’a read $\alpha$
  - ’b read $\beta$

- Examples:

  - fun swap(x,y)=(y,x);
    val swap = fn : ’a * ’b -> ’b * ’a

  - fun map f [] = []
    | map f (head::tail) =
    | (f head)::(map f tail);
    val map =
    fn : (’a -> ’b) -> ’a list -> ’b list

  - map swap [(3,"c"),(5,"f"),(17,"z")];
    val it = [("c",3),("f",5),("z",17)] : (string * int) list
Composition

- infix o;
infix o
- fun (f o g) x = f (g x);
val o =
   fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- fun plus3 x = x + 3; fun times4 y = y * 4;
val plus3 = fn : int -> int
val times4 = fn : int -> int

- fun times4plus3 z = (plus3 o times4) z;
val times4plus3 = fn : int -> int

- times4plus3 5;
val it = 23 : int
Equality Types

- Type variables for data types that must support equality begin with 'a'.
  - 'a read α=β
  - 'b read β=β

- Example — list membership:

  - fun member (e,[]) = false
  =  | member(e,(head::tail)) =
  =   (e=head) orelse (member (e,tail));
  val member = fn : 'a * 'a list -> bool

  - member ("t",["a","b","c"]);
  val it = false : bool

  - member (false,[true,true,false,true]);
  val it = true : bool
Heterogeneous Types

The datatype declaration handles heterogeneous data:

- type point = real * real;
  type point = real * real
- datatype shape =
  = Circle of point * real
  = (* (center, radius) *)
  = | Triangle of point * point * point
  = (* 3 points *)
  = | Square of point * point * point * point
  = point (* 4 points *);

datatype shape
  = Circle of (real * real) * real
  | Square of (real * real) *
  | (real * real) * (real * real) *
  | (real * real)
  | Triangle of (real * real) *
  | (real * real) * (real * real)
Heterogeneous Types

- Circle;
  val it = fn : point * real -> shape

- Square;
  val it = fn : point * point *
          point * point -> shape

- Triangle;
  val it = fn : point * point *
          point -> shape

- Circle is a function that takes a point and a radius and returns an object of type shape.

- Circle ( (1.0,2.5), 0.75);
  val it = Circle ((1.0,2.5),0.75) : shape
Enumeration Types

An enumeration type consists of a finite number of constants.

- datatype bool = true | false;
datatype bool = false | true

- false;
val it = false : bool

- datatype color = Red | Yellow | Blue |
  = Green;
datatype color = Blue | Green | Red | Yellow

- Green;
val it = Green : color
Function on Shapes

Use pattern matching on datatypes.

- fun center (Circle(c,r)) = c
  =  | center (Triangle((x1,y1),(x2,y2),
  =          (x3,y3))) =
  =          ((x1+x2+x3)/3.0,(y1+y2+y3)/3.0)
  =  | center (Square((x1,y1),(x2,y2),
  =          (x3,y3),(x4,y4))) =
  =          ((x1+x2+x3+x4)/4.0,
  =          (y1+y2+y3+y4)/4.0);
val center = fn : shape -> point

- center (Circle((2.5,3.78),4.97));
val it = (2.5,3.78) : point

- center (Square((0.0,0.0),(1.0,0.0),
  =          (0.0,1.0),(1.0,1.0)));
val it = (0.5,0.5) : point
Binary Trees

- datatype 'a tree = Lf
  =  | Br of 'a * 'a tree * 'a tree;
 datatype 'a tree = Br of 'a * 'a tree * 
  'a tree | Lf

- fun size Lf = 0
  =  | size (Br(v,t1,t2)) =
  1 + size t1 + size t2;
 val size = fn : 'a tree -> int

- fun depth Lf = 0
  =  | depth (Br(v,t1,t2)) =
  1 + Int.max (depth t1, depth t2);
 val depth = fn : 'a tree -> int

- fun preord (Lf, vs) = vs
  =  | preord (Br(v,t1,t2), vs) =
  = v :: preord (t1, preord (t2, vs));
 val preord =
  fn : 'a tree * 'a list -> 'a list
Binary Trees Continued

- \texttt{val tree2} = \texttt{Br(2, Br(1,Lf,Lf),}
  \texttt{Br(3,Lf,Lf))};
\texttt{val tree2} = \texttt{Br (2,Br (1,Lf,Lf),}
  \texttt{Br (3,Lf,Lf)) : int tree}

- \texttt{val tree5} = \texttt{Br(5, Br(6,Lf,Lf), Lf)};
\texttt{val tree5} = \texttt{Br (5,Br (6,Lf,Lf),Lf) :}
  \texttt{int tree}

- \texttt{val tree4} = \texttt{Br(4, tree2, tree5)};
\texttt{val tree4} = \texttt{Br (4,Br (2,Br #,Br #),}
  \texttt{Br (5,Br #,Lf)) : int tree}
Binary Trees Concluded

- size tree4;
val it = 6 : int

- depth tree4;
val it = 3 : int

- preord (tree4,[]);
val it = [4,2,1,3,5,6] : int list
Graphs

A directed graph can be represented as a list of ordered pairs.

- datatype 'node graph =
    Graph of ('node * 'node) list;
datatype 'a graph = Graph of ('a * 'a) list

- val beats = Graph [["paper","rock"],
    ("rock","scissors"),
    ("scissors","paper")];
val beats = Graph [["paper","rock"],
    ("rock","scissors"),
    ("scissors","paper")]
: string graph

- val divides =
    Graph [(3,6),(3,12),(6,12),
        (3,21),(6,18),(3,18)];
val divides =
    Graph [(3,6),(3,12),(6,12),
        (3,21),(6,18),(3,18)] : int graph
Basic Graph Functions

- **Successor:**

  ```ml
  fun nexts (a, Graph []) = []
  = | nexts (a, Graph ((x,y) :: rest)) =
  = if a=x then
        y :: nexts(a,Graph rest)
  = else nexts(a,Graph rest);
  val nexts =
    fn : ''a * ''a graph -> ''a list

  - nexts (3, divides);
  val it = [6,12,21,18] : int list

- **Nodes:**

  ```ml
  fun nodes (Graph x) =
  = (unique o flatten) x;
  val nodes = fn : ''a graph -> ''a list

  - nodes divides;
  GC #0.0.0.0.3.144: (4 ms)
  val it = [12,21,6,3,18] : int list```
Flatten and Unique

- **Flatten:**
  
  ```ml
  fun flatten [] = []
  | flatten ((x,y)::tail) =
  |     x :: y :: flatten tail;
  val flatten = fn : ('a * 'a) list -> 'a list
  ```

- **Unique:**
  
  ```ml
  fun member (e,[]) = false
  | member (e,(head::tail)) =
  |     (e=head) orelse member (e,tail);
  val member = fn : ''a * ''a list -> bool

  fun unique [] = []
  | unique (head :: tail) =
  |     if member(head,tail)
  |     then unique tail
  |     else head :: unique tail;
  val unique = fn : ''a list -> ''a list
  ```
Depth-First Search

Use depth-first search to determine the nodes reachable from a particular node.

- fun depthf ([], Graph graph, visited) =
  =     rev visited
  =   | depthf (head::tail, Graph graph,
  =   visited) =
  = if member(head,visited)
  = then depthf (tail,Graph graph,visited)
  = else depthf
  =   (nexts(head, Graph graph) @ tail,
  =       Graph graph, head::visited);
val depthf =
  fn : ''a list * ''a graph * ''a list ->
  ''a list

- depthf ([3],divides,[]);
val it = [3,6,12,18,21] : int list

- depthf ([6],divides,[]);
val it = [6,12,18] : int list
Depth-First Search

Faster version. Avoids append.

- fun depth ([], Graph graph, visited) =
  =       rev visited
  =     | depth (head::tail, Graph graph,
  =         visited) =
  =     depth (tail, Graph graph,
  =       if member(head,visited) then visited
  =     else depth (nexts(head,Graph graph),
  =       Graph graph, head::visited));
val depth =
   fn : ’’a list * ’’a graph * ’’a list ->
      ’’a list

- depth ([6],divides,[]);
val it = [12,6,18] : int

- depth ("rock", beats, []);
val it = ["paper","scissors","rock"] : string list
Exceptions

- Declaring exceptions:
  
  exception exid

  or

  exception exid of type

- Raising exceptions:

  raise exid

- Handling exceptions:

  exp handle pat₁ => exp₁ | ⋯ | patₙ => expₙ
Exception Example

- exception OddOne;
exception OddOne

- fun sum_even [] = 0
  =  | sum_even (head::tail) =
  =    if (head mod 2) = 1 then raise OddOne
  =      else head + (sum_even tail);
val sum_even = fn : int list -> int

- val t = [2,4,6,8,12,13,14,17,20];
val t = [2,4,6,8,12,13,14,17,20] : int list

- sum_even t;
GC #0.0.0.0.2.28: (2 ms)

uncaught exception OddOne
   raised at: stdin:39.36-39.42

- sum_even t handle OddOne => ^1
   | Any => ^3;
val it = ^1 : int
Infinite Data

An infinite list (called a sequence or lazy list) is an ordered pair where the second item is a function to generate the rest of the list.

- datatype 'a seq = Nil
  =                  | Cons of 'a * (unit -> 'a seq);
datatype 'a seq =
  Cons of 'a * (unit -> 'a seq) | Nil

- fun from n = Cons(n, fn() => from (n+1));
val from = fn : int -> int seq

- from 0;
val it = Cons (0,fn) : int seq
Sequences Continued

- fun power_2 n =
  =  Cons(n, fn () => power_2 (2*n));
val power_2 = fn : int -> int seq

- val p = power_2 1;
val p = Cons (1,fn) : int seq

- fun takeq (0,sequence) = []
  =  | takeq (n, Nil) = []
  =  | takeq (n, Cons(head,tail)) =
  =    head :: takeq(n-1, tail ());
val takeq = fn : int * 'a seq -> 'a list

- takeq (10,p);
val it = [1,2,4,8,16,32,64,128,256,512] : int list
Using a Sequence

- fun squares Nil : int seq = Nil
  =     | squares (Cons(head,tail)) =
  =       Cons(head*head,
  =               fn() => squares (tail()));
val squares = fn : int seq -> int seq

- val sq = squares (from 1);
val sq = Cons (1,fn) : int seq

- takeq (12,sq);
val it =

  [1,4,9,16,25,36,49,64,81,100,121,144] :
  int list
The Sequence of Primes

Start with a function to filter a sequence according to a predicate.

- fun filterq pred Nil = Nil
  =  | filterq pred (Cons(head,tail)) =
  =    if pred head then
  =    Cons(head,
  =    fn() => filterq pred (tail()))
  =    else filterq pred (tail());
val filterq =
  fn : ('a -> bool) -> 'a seq -> 'a seq

Now build a predicate to sift a sequence of integers by division by a prime.

- fun sift p =
  =  filterq (fn n => n mod p <> 0);
val sift = fn : int -> int seq -> int seq
The Sequence of Primes
Continued

Finally construct a function that sifts by all primes.

- fun sieve Nil = Nil
  = | sieve (Cons(p,rest)) =
  = Cons(p,
  = fn () => sieve(sift p (rest())));
val sieve = fn : int seq -> int seq

Create the sequence of primes.

- val primes = sieve (from 2);
  val primes = Cons (2,fn) : int seq

- takeq(15,primes);
  val it =
  2,3,5,7,11,13,17,19,23,29,31,37,...] :
  int list
Reference Semantics

• A reference to an object in the heap:

  - `val l = ref primes;`
  - `val l = ref (Cons (2, fn)) : int seq ref`

• The object that is referenced:

  - `!l;`
  - `val it = Cons (2, fn) : int seq`

• An assignment:

  - `l := sq;`
  - `val it = () : unit`
  - `!l;`
  - `val it = Cons (1, fn) : int seq`

• Assignment requires type match.
Structures

• Abstract data types or modules in ML are called **structures**.

• A structure encapsulates types, values, and functions.

• External access to these can be controlled by **signatures**.
Stack Structure

signature STACK =
    sig
        type element
        val pop : unit -> element
        val push : element -> unit
        val empty : unit -> bool
    end

structure Stack : STACK =
    struct
        type element = int
        val stack = ref [] : element list ref
        fun pop () =
            case !stack of
                [] => ~1
                | (top::bottom) =>
                    (stack := bottom; top)
        fun push x =
            (stack := x::(!stack))
        fun empty () = (!stack = [])
    end;
Use of Stack

- use "Stack.sml";
[opening Stack.sml]
signature STACK =
  sig
    type element
    val pop : unit -> element
    val push : element -> unit
    val empty : unit -> bool
  end
structure Stack : STACK
val it = () : unit

- Stack.empty();
val it = true : bool
Use of Stack Continued

- Stack.push 5; Stack.push 12; Stack.push 9;
  val it = (): unit
  val it = (): unit
  val it = (): unit

- Stack.empty();
  val it = false : bool

- Stack.pop(); Stack.pop();
  = Stack.pop(); Stack.pop();
  val it = 9 : Stack.element
  val it = 12 : Stack.element
  val it = 5 : Stack.element
  val it = ~1 : Stack.element

- Stack.stack;
  stdIn:18.1-18.12
    Error: unbound variable or constructor:
    stack in path Stack.stack
Opening the Stack

- open Stack;

opening Stack
  type element = int
  val pop : unit -> element
  val push : element -> unit
  val empty : unit -> bool

- empty();

val it = true : bool

- stack;

stdIn:16.1-16.6
  Error: unbound variable or constructor: stack
Opening the Stack
Continued

- push 44; push 112; push 97;
  val it = () : unit
  val it = () : unit
  val it = () : unit

- empty();
  val it = false : bool

- pop(); pop();
  val it = 97 : element
  val it = 112 : element

- Stack.empty();
  val it = false : bool

- fun pair (x: element) = (x,x);
  val pair = fn : element -> element * element
ML Summary

- A strongly-typed, functional programming language with type inferencing

- Interpreted or compiled

- Functions are first-class objects. Function application is the essence of computation.

- Cases are handled by pattern matching.

- Supports polymorphic types

- Supports infinite data structures, but has eager evaluation

- Supports exceptions, type-safe imperative operations, and abstract data structures