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: allow functions to have **polymorphic types**

Polymorphic types

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 the same type

When calling a polymorphic function, caller knows real type

- replace type variable with regular type

```
val length = fn : 'a list -> int
```

```
length([3,4,5])
```

replaces `'a` with `int`

```
length([(3.0,"hi"), (2.0,"there")])
```

replaces `'a` with `int * string`

Examples

```
- fun map(f, nil)    = nil
    | map(f, x::xs) = f x :: map(f, xs);
val map = fn : ('a -> 'b) * 'a list -> 'b list


val square = fn : int -> int

- map square [3,4,5];  (* 'a→int, 'b→int *)
[9,16,25] : int list


val length = fn : 'a list -> int

- map length [[3,4], [5,6,7], []]
    (* length: 'a → int;
       map: 'a → int list, 'b → int *)
[2,3,0] : int list
```


Currying

Simple definition:

```
- fun map f =
    fn lst =>
      if null lst then nil
      else f (hd lst) :: (map f) (tl lst);
val map = fn : ('a->'b) -> 'a list -> 'b list
```

Simple application:

```
- (map square) [3,4,5];
[9,16,25] : int list
```

Exploiting left-to-right associativity of “juxtaposition” fn calling:

```
- map square [3,4,5];
[9,16,25] : int list
```

Pattern-matching-based definition:

```
- fun map f =
    fn nil => nil
     | x::xs => f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list
```

Clean syntactic sugar for currying

Allow multiple formal argument patterns ⇒ curried function

```
- fun map f  nil    = nil
    | map f (x::xs) = f x :: map f xs;
val map = fn : ('a->'b) -> 'a list -> 'b list

- map square [3,4,5];
[9,16,25] : int list

- val squarer = map square;
val squarer = fn : 'int list -> 'int list

- map (map square) [[3,4,5], [6,7,8], [1,2]];
[[9,16,25], [36,49,64], [1,4]] : int list
```

Curried is the normal way to define functions

- syntactically cleaner
- semantically more flexible

Polymorphism versus overloading

Polymorphic function:
same function usable for many different argument types

```
- fun append  nil    lst = lst
    | append (x::xs) lst = x::append xs lst;
val append = fn : 'a list -> 'a list -> 'a list
```

Overloaded function:
several different functions all given name

Resolve overloading to particular function based on:

- static argument types (in ML)
- dynamic argument classes (in object-oriented languages)

```
- 3 + 4;
7 : int
- 3.0 + 4.5;
7.5 : real
- op +;  (* which +? default to int version *)
fn : int*int -> int
- (op +):real*real->real;
fn : real*real -> real
```


Equality types

The = built-in function is polymorphic over all types that "admit equality"

- any type except those containing functions or reals

Use `''a`, `''b`, etc. to stand for these equality types

```
- fun member x  nil    = false
    | member x (y::ys) =
        x = y orelse member x ys;
val member = fn : ''a -> ''a list -> bool

- member 3 [4,5,6,3,4,5];
true : bool
- member 4.5 [3.4,5.6,4.5];
... type error ...
- member ("hi",3,square) [("there",6,double)];
... type error ...
```


Type inference

System infers types of expressions automatically, 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
  - each reference to a let-bound polymorphic identifier (~ named function) gets separate type variables
- each subexpression in construct places constraints on types of its operands
- solve constraints

Constraint solving can partially refine types, e.g.:

- `'a` ⇒ `'b list`
- `'a` ⇒ `''b`

If overloaded operator is unresolved after constraint solving, default to `int` version

Overconstrained (unsatisfiable constraints) ⇒ type error

Underconstrained ⇒
OK for let-bound functions (gives polymorphic function),
type error ("nongeneralizable type variable") otherwise

Properties of ML's type system

Hindley-Milner type system/inference

- universal parametric polymorphism,
  no constrained polymorphism (other than equality types)
- only let-bound polymorphism
  - cannot pass polymorphic value that's polymorphic inside callee

```
- fun id x = x;
val id = fn : 'a -> 'a
- fun g f = (f 3, f "hi");
(* type error in ML, but in SuperML++: *)
val g = fn : (∀'a.'a->'a) -> int*string
(* not the similar ML type:
val g = fn : ∀'a.(('a->'a) -> int*string) *)
- g id;
(* I wish *)
(3, "hi") : int * string
```

Type inference yields **principal type** for expression

- single most general type that can be inferred

Worst-case complexity of type inference: exponential time
Average case complexity: linear time

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;
val signum = fn : int -> sign

- fun signum_value(Positive) = 1
    | signum_value(Zero)     = 0
    | signum_value(Negative) = ~1;
val signum_value = fn : sign -> int
```


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;
datatype Sexpr

- Nil;
Nil : Sexpr
- Integer;
Integer : int -> Sexpr
- Symbol;
Symbol : string -> Sexpr
- Cons;
Cons : Sexpr * Sexpr -> Sexpr
```


Using datatypes

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

- fun car  Nil         = Nil
    | car  (Cons(x,_)) = x;
val car = fn : Sexpr -> Sexpr
- fun cdr  Nil          = Nil
    | cdr  (Cons(_,xs)) = xs;
val cdr = fn : Sexpr -> Sexpr

- cdr lst;
Cons(Symbol "hi", Nil) : Sexpr

- car lst = Integer 3;
(* Sexpr is an equality type;
   = defined automatically *)
true : bool
```

Recursive user-defined data types

Example: binary search trees of integers

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

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

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

How to write a polymorphic version, over = and <?

`int_tree` is an equality type. Does = do the right thing?

Polymorphic user-defined data types

Polymorphic binary trees, passing = and < functions as args:

```
- datatype 'a tr =
    Empty |
    Node of 'a * 'a tr * 'a tr;
- fun ins eq lt x  Empty = Node(x,Empty,Empty)
    | ins eq lt x (n as Node(y,t1,t2)) =
        if eq(x,y) then n
        else if lt(x,y) then
              Node(y, ins eq lt x t1, t2)
             else
              Node(y, t1, ins eq lt x t2);
val ins = fn :
  ('a*'a->bool)->('a*'a->bool)-> 'a -> 'a tr
   -> 'a tr
- fun memb eq lt x  Empty          = false
    | memb eq lt x (Node(y,t1,t2)) =
        if eq(x,y) then true
        else if lt(x,y) then memb eq lt x t1
                        else memb eq lt x t2;
val memb = fn :
  ('a*'b->bool)->('a*'b->bool)-> 'a -> 'b tr
   -> bool
```


A wrapper

Avoid clients passing `eq` and `lt` procedures by
creating a wrapper data structure storing procedures

```
- datatype 'a tree =
    Tree of {tree: 'a tr,
             eq: 'a * 'a -> bool,
             lt: 'a * 'a -> bool};
- fun make_tree eq_fn lt_fn =
    Tree {tree=Empty, eq=eq_fn, lt=lt_fn};
val make_tree = fn :
  ('a*'a->bool) -> ('a*'a->bool) -> 'a tree

- fun insert x (Tree {tree=tr,
                      eq=eq_fn, lt=lt_fn}) =
    Tree {tree=ins eq_fn lt_fn x tr,
          eq=eq_fn, lt=lt_fn};
val insert = fn : 'a -> 'a tree -> 'a tree

- fun member x (Tree {tree=tr,
                      eq=eq_fn, lt=lt_fn}) =
    memb eq_fn lt_fn x tr;
val member = fn : 'a -> 'a tree -> bool
```


A problem

In Scheme,
can use “distinguished values” to handle exceptional cases

```
(define (find pred lst)
  (cond ((null? lst) #f)
        ((pred (car lst)) (car lst))
        (else (find fn (cdr lst)))))
(find is-positive? '(-3 3 5)) → 3
(find is-positive? '(-3 -5))  → #f
```

In ML, doesn't work well

```
- fun find pred  nil    = false
    | find pred (x::xs) =
        if pred x then x else find pred xs;
val find = fn: (bool->bool) -> bool list -> bool

- find is_positive [~3,3,5];
... type error ...
```

Exceptions

Declare exceptions that can be returned from functions
(without affecting normal return type)
and handled in calling functions

```
- exception NotFound;
exception NotFound
- fun find pred  nil    = raise NotFound
    | find pred (x::xs) =
        if pred x then x else find pred xs;
val find = fn : ('a->bool) -> 'a list -> 'a

- find is_positive [~3,3,5];
3 : int

- find is_positive [~3,~5];
uncaught exception NotFound
```


Handling exceptions

Add handler clause to expressions to handle (some) exceptions raised in that expression

- must return same type as handled expression

Syntax:

```
expr handle exn_name => expr
        { | exn_name => expr }
        [ | _ => expr ]
```

```
- (find is_positive [~3,~5])
     handle NotFound => 0
0 : int
```


Exceptions with arguments

Can have exceptions with arguments

```
- exception IOError of int;
exception IOError of int;

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


An application

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

- exception NotFound;
- fun lookup key (Table(assocs)) =
    let fun lookup_helper key nil            =
              raise NotFound;
          | lookup_helper key ((k,v)::asscs) =
              if key = k then v
              else lookup_helper key asscs
    in lookup_helper key assocs end;
val lookup = fn :
  ''key -> (''key,'value) table -> 'value

- fun store key value (Table(assocs)) =
    Table((key,value)::assocs);
val store = fn :
  ''key -> 'value -> (''key,'value) table
    -> (''key,'value) table
```

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:

- `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;
() : unit
- !v;
1 : int
```

Must say which things are mutable, unlike `set!` et al.

Modules

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

Modules support **encapsulated** abstract data types, hidden operations, values, and types, and some kinds of polymorphism

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 name *)
  (* constructors: *)
  val empty = nil;
  fun enq x q = q @ [x]; (* @ is append *)
  (* accessors: *)
  exception empty_queue;
  fun head (x::q) = x
    | head  nil    = raise empty_queue;
  fun deq (x::q) = (x,q)
    | deq  nil   = raise empty_queue
end;
```

Access members of structure using dot notation:

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

Importing

To avoid long qualified names, use `open`

```
- open Queue1;
(* actually, this is broken in our version of ML,
   due to uses of nil (via empty) at top-level *)
- val q = enq 4 (enq 3 empty);
val q = [3,4] : int list
- let val (x,q2) = deq(q) in head q2 end;
4 : int
```

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

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

- QueueExtension.enq2 3 4 Queue1.empty;
[3,4] : int list
- QueueExtension.enq 3 QueueExtension.empty;
... error: enq not in QueueExtension ...
```


Signatures

Previous declarations exposed representation of queue as list, in particular order

To achieve encapsulation, define a public external interface

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

Apply signature to restrict interface to structure:

```
structure Queue2:QUEUE = struct ... end;
```

If there were operations in `struct` that weren't in `sig`, then these would become inaccessible

Holes in encapsulation

Signatures don't completely hide module implementation

Types defined using `type` are not hidden

```
- Queue2.empty = nil;
true : bool;
```

Types created with `datatype` *are* hidden, though:

```
structure Queue3:QUEUE = struct
  datatype 'a T = Queue of 'a list;
  val empty = Queue nil;
  fun enq x (Queue q) = Queue (q @ [x]);
  exception empty_queue;
  fun head (Queue (x::q)) = x
    | head (Queue  nil  ) = raise empty_queue;
  fun deq (Queue (x::q)) = (x,q)
    | deq (Queue  nil  ) = raise empty_queue
end;
- Queue3.enq 4 (Queue3.enq 3 Queue3.empty);
[3,4] : int Queue3.T
- Queue3.empty = nil;
... type error ...
```


Another hole

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(Queue3);
```


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 IntOrder = struct
  type T = int;
  val lt = (op <);
  val eq = (op =);
end;
structure IntSort = Sort(IntOrder);
IntSort.sort([3,5,~2,...]);
```

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

Cannot instantiate functors at run-time to create “objects”
$\Rightarrow$ cannot simulate classes and object-oriented programming

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

- multiple types can share private data & operations

Functors similar to parameterized ADTs

Modules more general than ADTs,
but clumsier for common case