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;
(* lots of output, indicating enq et al. defined *)
- 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