Modules in General

An ADT is one particular kind of **module**, containing:

- a single abstract type, with its representation;
- a collection of operators, with their implementations.

**Instances** of the ADT are typically created dynamically, and contain space for the components of the representation; all the instances share the same operator code.

More generally, modules might contain:

- multiple type definitions;
- arbitrary collections of functions (not necessarily abstract operators on the type);
- variables;
- constants;
- exceptions; etc.

Primary purpose is to **divide** large programs into (somewhat) independent sections, offering **separate namespaces** and perhaps **separate compilation**.
Interfaces

Even when a module does not represent a particular abstract data type, it usually represents a kind of abstraction over some set of facilities, in which some implementation information will be hidden behind an interface.

Clients of a module want to know what module does, not how it does it. Of course, specifying “what” is a hard problem! A key goal is that it should be possible to change the implementation without rewriting (or ideally, even recompiling) the client code that depends on the interface.

Most languages use type information to give a partial characterization of what a module does. An interface definition is then a collection of identifiers with their types.

In many languages we can write and compile client code based solely on type interfaces.

Of course, there must also be an (at least informal) specification of what the module’s facilities do, but few languages provide any support for making sure that the implementations adhere to more than a type specification.
Modules in ML

ML modules are called **structures**. By default, a structure exports all its components, and does not need a specified interface (since its component types can be inferred.)

```ml
structure Eval =
struct
  open Ast (* avoid dot notation *)
  fun evalexp env exp = ...
  val eval = evalexp Env.empty
end
```

ML module interfaces are called **signatures**. Signatures can be attached to structures, but can also be separately named and manipulated, without reference to any particular structure.

```ml
signature EVAL =
sig
  val eval : Ast.exp -> int
end
```

The same structure can be **viewed** through multiple signatures. For example, a structure can be defined without an explicit signature but later be **thinned** by a signature to form a more private structure.

```ml
structure LimitedEval : EVAL = Eval
```
Aside: Partial Transparency in ML

In our usual presentation of the Env structure, the signature completely hides the fact that environments can be compared for equality.

```ml
structure Env :>
   sig
     type 'a env
     ...
   end =
struct
  type 'a env = (string * 'a) list
  ...
end
```

There are two approaches to solving this problem.

(1) We can make the fact that env is an equality type explicit in the signature:

```ml
structure Env :>
   sig
     eqtype 'a env
     ...
   end = ...
```
Partial Transparency (2)

(2) We can expose the underlying implementation by using **non-opaque** signature matching (\(\_\) instead of \(\_\rightarrow\)), but maintain abstraction by using a datatype with no exported constructors.

```
structure Env :
  sig
    type 'a env
    ...
  end =
struct
  datatype 'a env = ENV of (string * 'a) list
  ...
end
```

Now the implementation of type `env` is public, so equality comparison works, but it is still impossible for clients to create or inspect values, because the constructor `ENV` is not public.
Modules in C?

Even C provides a (primitive) form of (unnamed) modules, i.e., files.

- The top-level declarations in a file are its components.
- By default, all components are exported, but they can be hidden using the `static` specifier.
- The `.h` file serves as a rough kind of interface specification. Manual methods must be used to ensure that such files are accurate and complete, and that they are used where needed.

The major defect of C’s approach is that all the names exported from all the files linked into a program occupy one `global` name space, and hence must be unique. There is no “dot” notation.
Polymorphism

Often need to perform same or similar actions on data of different types.

Goal: Avoid writing the same code twice (while maintaining type safety and efficiency).

Simplest case is **parametric polymorphism**, where behavior of the code is essentially the same regardless of the types being manipulated. Particularly useful for **container types**.

Example: algorithm to reverse a list is the same, no matter what the contents of the list.

Languages in the Lisp tradition support this kind of polymorphism at the function level. Easy in Lisp because it is untyped, but even Standard ML can do this for **homogeneous** lists.

```ml
datatype 'a list = nil | :: of 'a * 'a list
fun ('a) reverse (l: 'a list) : 'a list =
  case l of
    nil => nil
  | (h::t) => (reverse t)@[h]
val a : int list = reverse [1,2,3]
val i = hd a
val b : bool list = reverse [true,false,true]
```
More Complex Parameterization

We cannot do this directly in most conventional typed languages (without modules), although Java can do this (somewhat awkwardly).

Sometimes the behavior of the code **differs** significantly depending on the types being manipulated.

Classic example: sorting. It makes sense to use the same sort algorithm on many different types of data (e.g., integers, reals, strings, etc.), provided they have a defined ordering.

But need to parameterize on **type** of elements **and** on comparison **function** to use on elements.
Parameterization in C

One approach is to make the comparison function an argument to sort, as with the C library quicksort function:

SYNOPSIS
qsort(base, nel, width, compar)
char *base;
int (*compar)();

EXAMPLE
static int intcompare(i,j)
  int *i, *j;
  {
    return(*i - *j);
  }

main()
{
  int a[10];
  ...
  qsort(a,10,sizeof(int),intcompare);
  ...
}

Note: Not type safe!
Parameterization in ML

If our language supports first-class functions, a better approach is to write a function that takes the comparison test as an argument and returns a specialized sorting function:

```
fun ('a) mksort (lessthan : 'a * 'a -> bool) : ('a list -> 'a list) =
  let fun sort nil = nil
      | sort (h::t) = insert h (sort t)
  and insert x nil = [x]
  | insert x (h::t) =
      if lessthan(h,x) then
        h::(insert x t)
      else x::h::t
  in sort
  end

val sortint = mksort (Int.<)
val l = sortint [3,1,2]
```

This extends (awkwardly) to situations where we want to generate several functions based on the same functional parameter (e.g., operations on sets with a certain notion of equality).
Parameterized Modules

Really want a way to have parameterized modules over types and operators.

Those conventional typed languages that support polymorphism at all, do so only at the module level. Here we always need to parameterize polymorphic algorithms by type, and maybe operators too.

Examples: Ada generic packages, C++ templates, ML functors.
Functors Example

signature SortArg = 
sig
  type t
  val lessthan: t * t -> bool
end

functor Sort(SA:SortArg) : sig
  type t
  val sort : t list -> t list
end =
structure SortInt =
  Sort(type t = int val lessthan = Int.<)

SortInt.sort [1,2,3];
Top-down Development with Functors

With functors, we can write and compile client code without having an implementation at all!

```haskell
signature ENV =
sig
  type env
  val empty : env
  val extend : env -> (string * int) -> env
  val lookup : env -> string -> int option
end

functor EvalF(structure Env:ENV) =
struct
  fun evalexp (env:Env.env) e =
    case e of
      Var v => Env.lookup env v
    | ...  
  ...  
end

------------------- --- ---- --- --
structure MyEnv : ENV = struct ... end
structure Eval = EvalF(structure Env = MyEnv)
fun main () = ... Eval.evalexp (e) ...
```
Compilation models for Polymorphism

Generic behavior can’t come for free!

Example: How can a generic sort function deal with an array whose entries are of arbitrary size?

In C, programmer must pass the size explicitly!

- Inefficient; doesn’t generalize.

There are two general approaches to compiler-generated generics.

In Ada and C++, completely separate code is generated for each instance of a generic (no code is generated for the generic definition itself).

- Each separate instance “knows” the size and layout of all the type parameters, and can be compiled just like ordinary code.

- Code runs as efficiently as ordinary code.

- But if generics are used heavily, there may be a “code explosion.”

With Ada generics, programmers must explicitly instantiate a generic at the specific instances of interest; with C++ templates, instantiation is supposed to be done automatically by the compiler.
Compilation Models (cont.)

The other approach, often used in ML, is to have just one copy of polymorphic or functorized code.

- Represent all data objects by a single word; if the object is larger than a word, it is stored in the heap and represented by a pointer.

- Identical machine code can work on any container object of a given shape.

- Approach extends to functors: one copy of the code can be generated for a functor definition; no code is generated when the functor is instantiated.

- Supports genuine separate compilation in the top-down-development example.

- Polymorphic and functorized code still runs as efficiently as ordinary code, and there’s no fear of code explosion.

- But ordinary code may run more slowly than in Ada or C++ because of more indirect pointers. So recent ML implementations have been moving towards a code specialization approach to improve performance.
Overloading

Most languages provide some form of overloading, where the same symbol means different things depending on the types to which it is applied.

E.g., overloading of arithmetic operators to work on either integers or reals is very common.

Aim is to do “what we expect;” rules can get quite complicated (especially when coercions are considered)!

Some languages (e.g., Ada, C++) allow user-defined overloading, normally for user-defined types (e.g., complex numbers).

In conventional languages, overloading is resolved statically; that is, the compiler selects the appropriate version of the operator once and for all at compiler time. (Different from object-oriented dynamic overriding.)

Overloading combined with generic modules gives us a way to refer to different instances of a generic by the same name; the compiler figures out which instance we meant.

Overloading is sometimes called “ad-hoc polymorphism”. It is fundamentally different from parametric polymorphism, because the implementation of the overloaded operator changes according to the underlying types.