

CS558 Programming Languages

Winter 2008

Lecture 14

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.

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** an **abstraction barrier**, and perhaps **separate compilation**.

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.)

```
structure Machine =  
struct  
  open Stack (* avoid dot notation *)  
  type prog = ...  
  fun progToString instrs = ...  
  fun exec instrs = ...  
end
```

MODULE INTERFACES IN ML

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.

```
signature MACH =  
sig  
  type prog (* details hidden *)  
  val exec : prog -> 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.

```
structure LimitedMachine : MACH = Machine
```

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.

CLASSES AS MODULES

In many OO languages, the **class** mechanism is used to get the effect of modules.

This can happen in two ways:

- A class might correspond to a **single** (abstract) type, with fields holding the type representation and methods implementing operations on the type.
- A class might correspond to a **collection** of types, defined by **nested** classes, and operations, defined by (typically **static**) methods.

This dual use of the class mechanism is confusing. A separate module-level mechanism would be better. (Java does have **packages** to help.)

POLYMORPHISM REVISITED

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. Example: polymorphic functions in ML.
- Harder case is **ad-hoc polymorphism**, where 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

```
void qsort (void *base, int nmemb, int size,  
           int (*compar) (const void *,const void *));
```

EXAMPLE

```
static int intcompare(int *i,int *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 (lt : '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 lt(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/class level. Here we always need to parameterize polymorphic algorithms by type, and maybe operators too.

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

ML FUNCTORS EXAMPLE

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

```
functor Sort(SA:SortArg) : sig  
  type t  
  val sort : t list -> t list  
end =
```

```
struct  
  type t = SA.t  
  fun sort nil = nil  
    | sort (h::t) = insert h (sort t)  
  and insert x nil = [x]  
    | insert x (h::t) =  
      if SA.lt(h,x) then h::(insert x t) else x::h::t  
end
```

```
structure SortInt = Sort(type t = int val lt = 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!

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

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

Later on...

```
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 definition itself).

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

A different 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 **boxed** (stored in the heap and represented by a pointer).
- Identical machine code can work on any instance of a polymorphic type.
- 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.