Lecture 27: Modules

CSC 131
Fall, 2014

Kim Bruce

Modules

- Reusable modules:
  - Separate, but not independent compilation
  - Maintain type checking
  - Control over export and import of names

Ada (1980)

```
generic
    length : Natural := 100;   -- generic parameters
    type element is private;
package stack is
    type stack is private;
    procedure make_empty (S : out stack);
    procedure push (S : in out stack; X : in element);
    procedure pop (S : in out stack; X: out element);
    function empty (S : stack)  return boolean;
    stack_error : exception;

private
    type stack is record
        space   : array(1..length) of element;
        top     : integer range 0..length := 0;
    end record;
end stack;
```

Why does specification have "private" part?

Ada (1980)

```
package body stack is
    procedure make_empty (S : out stack);
    begin
        S.top := 0;
    end make_empty ;

    procedure push (S : in out stack; X : element) is
    begin
        if full(S)  then
                raise stack_error;
        else
                S.top := S.top + 1;
                S.space(S.top) := X;
        end if;
    end push;

    ...
end stack;
```

Ada (1980)

```
s: stack(100,int);

begin
  make_empty(s);
  push(s, 47);
  if (empty(s)) then ...
end;
```

Internal representation (object)

```
generic
    length : Natural := 100;      -- generic parameters
    type element is private;      -- only assignment and tests for =
package stack is                  -- specification only
    procedure push (X : in element);
    procedure pop (X: out element);
    function empty return boolean;
    function full return boolean;
    stack_error : exception;
end stack;

package body stack is             -- implementation
    space : array (1..length) of element;
    top : integer range 0..length := 0;

    procedure push (X : in element) is
    begin
        if full()  then
            raise stack_error;
        else
            top := top + 1;
            space(top) := X;
        end if;
    end push;
    ...
```

Using internal representation

```
package stack1 is new stack(20,integer);
package stack2 is new stack(100, character);
    -- Note that this initializes length in both cases to 0
use stack2;
stack1.push(5)
if not stack1.empty() then
    stack1.pop(Z);
endif;
push('z');
```

- Internal rep like an object
- Changing rep requires recompilation but not changing source code of users.

Modula 2

- Similar to Ada except
  - no generics
  - no “private” section
- Require all private types to take same amount of space -- a pointer

```
DEFINITION MODULE stackMod;
IMPORT element FROM elementMod;
    TYPE stack;
    PROCEDURE make_empty (VAR S : stack);
    PROCEDURE push (VAR S : stack; X : element);
    PROCEDURE pop (VAR S : stack; X: element);
    PROCEDURE empty (S : stack): BOOLEAN;

END stackMod.

IMPLEMENTATION MODULE stackMod;
    TYPE stack = POINTER TO RECORD
        space   : array[1..length] of element;
        top     : INTEGER;
    END;

    PROCEDURE make_empty (VAR S : stack);
    BEGIN
        S^.top := 0;
    END make_empty ;

    ...  (* can be start-up code too to initialize *)
END stackMod;
```

Ada vs. Modula 2

- Representations fairly similar
  - can import from other units (modules or packages) and export items to other units.
- For external representations not much difference.
- Private types in Ada vs opaque types in Modula.

Ada vs. Modula 2

- Use of opaque types require Pointer types
- Representation changes
  - in Ada forces recompilation of user programs
  - Not in Modula 2
- Internal reps of ADT's almost identical.
- Ada more flexible via generic routines -
  - can parameterize on types and sizes
  - Create new instances of packages

Easier if Uniform Reps

- LISP, Scheme, ML, Haskell, Clu, Eiffel, and Java have uniform reps for values so can share same code.
- Non-uniform representations:
  - Ada requires different implementation, but still type-checks statically.
  - C++ type-checks only when instantiated
- Automatic boxing and unboxing now helps with primitives in Java and C#.

Modules in Haskell

- Module:
  - Control namespace
  - define abstract data types.
  - No nesting of modules
- Examples: PcfLexer.hs, ParsePCF.hs
  - Name starts with cap (not nec. same as file)
  - list functions and types (including constructors) to be exported.
  - Make more abstract by leaving off constructors
  - if no export list, then all exported

Importing

- Module must be explicitly imported
  - Can narrow more by listing items to be imported
    - import PcfLexer(getTokens, Token(ID,NUM))
  - If name conflicts, can "import qualified" and access:
    - PcfLexer.getTokens
  - "hiding" clause can hide imported items.
  - "as" clause can rename imported features

ML

- datatype not provide information hiding
- abstype similar, but provides information hiding
  - Can't get at representation (no pattern matching)
  - deprecated, so skip.

ML Modules

- SML has two sub-languages:
  - Core -- programming in the small
    - details of types and expressions
  - Modules -- programming in the large -- architecture.
    - group defs of types & expressions into units w/interfaces
- Separate interfaces (signatures) from implementations (structures)
- Can reveal implementations if want.

Signature

```
signature INTSTACKSIG =
  sig
    type intstack;
    exception stackUnderflow;
    val emptyStk: intstack;
    val push: int -> intstack -> intstack;
    val pop: intstack -> intstack;
    val top: intstack -> int;
    val IsEmpty: intstack -> bool;
  end;
```

Structure

optional

```
structure IntStack: INTSTACKSIG =
  struct
    type intstack = int list;
    exception stackUnderflow;

    val emptyStk = [];

    fun push (e:int) (s:intstack) = (e::s);

    fun pop [] = raise stackUnderflow
      | pop (e::s) = s;
     ...

     fun extra ...  (* not visible outside *)
   end;
```

Accessing Structure

- IntStack.push 12 IntStack.emptyStk
- open IntStack;
  push 12 emptyStk;
- Considered bad style to open outside structure.
- Rather than open, rename:
  - val push = IntStack.push;
    val emptyStk = IntStack.emptyStk;

Ascription

- `structure IntStack: INTSTACKSIG = ...`
  - lets type definitions escape - transparent
  - hides extra features
- `structure IntStack:> INTSTACKSIG = ...`
  - also hides type definitions - opaque
- Can further restrict structure to create "views" by giving new name and signature
  - `structure ResStack:> RESSTACKSIG = IntStack`

More ML Modules

- Modules may be nested -- helping modules
- Functors: Modules parameterized by other modules.
- Supports code reuse -- apply to many different structures

Functor Examples

```
signature EQ =
   sig
      type t
      val eq : t * t -> bool
   end;

functor PairEQ(P : EQ) : EQ = struct
  type t = P.t * P.t
  fun eq((x,y),(u,v)) = P.eq(x,u) andalso P.eq(y,v)
end;

structure IntEQ : EQ = struct
      type t = int
      val eq : t*t->bool = (op =)
    end;

structure IntPairEQ : EQ = PairEQ(IntEQ);
```

What is this like in Haskell?

Module Languages

- Signatures like types
  - Describe families of structures
  - Make functor argument specification possible
  - But ... components can themselves be types
- Structures like records
  - But ... can contain types
- Complications:
  - Opacity sometimes requires "sharing constraints"
  - force types in parameters to match up -- omit details

Evaluating ML Modules

- Limitations:
  - Functors are first-order only
    - can't be applied to or return functor
  - Structures & functors are second-class values
    - Compile-time structures, can't be constructed or stored at run-time.

Key Features of ADT's

- Encapsulation of all features in one place
- Information hiding -- explicit control over what imported and exported.
- Generally separate specification and implementation (except in Clu)
  - Separately compiled (except in ML)
- Ada, Clu, and ML provide parameterized modules.