CS301 W'99 Lecture Notes
Lecture 15

Need for abstraction

Is a new type name a genuinely new type, equivalent to the built-in types?

E.g., let's implement a **stack** using an array:

```
TYPE stack = ARRAY 100 OF INTEGER;
VAR s : stack := ARRAY(100,0);
VAR top : INTEGER := 0;
PROCEDURE push(i:INTEGER; s: STACK) IS
BEGIN
  s[top] := i;
  top := top + 1;
END;
...
```

• User of `stack` can **abuse** stack discipline, e.g.,

```
s[random] := 42;
```

• `stack`, `s`, `t`, `push`, etc. aren't grouped together. • Intended use of stack isn't explicit.

On other hand, **machine** datatypes usually are presented abstractly. We don't write

```
if (x & 0x80000000) printf ("x is negative");
```

We'd like to "extend language" by making user-defined types "act like" built-in hardware types.

Abstract Data Types (ADT's)

Ideally, to mimic the behavior of built-in hardware-based types, user-defined types should have an associated set of **operators**, and it should only be possible to manipulate types via their operators (and maybe a few generic operators such as assignment or equality testing).

In particular, when new types are given a **representation** in terms of existing types, it shouldn't be possible for programs to inspect or change the fields of the representation.

Such a type is called an **abstract** data type (**ADT**), because to clients (users) of the type, its implementation is hidden; only its **interface** is known.

We can implement an ADT by combining a type definition together with a set of function operating on the type into a **module** (or **package**, **cluster**, **class**, etc.) Additional **hiding** features are needed to make the type's representation more-or-less invisible outside the module.

Abstraction Compare to **procedural** abstraction: procedure can be **called** if its **type** is known, even if its implementation is not.

Benefits of abstraction:

• Implementation and client can be developed **independently**.

• Implementation can be **changed** without affecting client's code.

• Improves clarity, maintainability, etc.

Example: ADT for Environments (pseudo-PCAT)

```
SIGNATURE env IS
  TYPE env;
  VAR empty : env;
  PROCEDURE extend (e:env,s:STRING,i:INTEGER):env;
  PROCEDURE lookup (e:env,s:STRING) : INTEGER;
END;

MODULE env : env IS
  TYPE env = RECORD
               id: STRING;
               val: INTEGER;
               next : env;
             END;
  VAR empty : env := NIL;
  PROCEDURE extend (e:env;s:STRING;i:INTEGER):env IS
    BEGIN
      RETURN env {id := s; val := i; next := e };
    END;
  PROCEDURE lookup (e:env,s:STRING) : INTEGER IS
    BEGIN
      WHILE e <> NIL DO
        IF e.id = s THEN
          RETURN e.val;
      END;
      RETURN -1;
    END;
END;
```


Client code is restricted:

```
(* client *)
VAR x := env.empty;
x := env.extend(x,"abc",99);
env.lookup(x,"def");

print (x.next.val);  (* NONO! *)
```

Thus, bodies can be changed without affecting clients:

(Note following implementation is not actually as general as the first one, but still matches signature.)


```
MODULE env : env IS
  TYPE envr = RECORD
               id: STRING;
               val: INTEGER;
              END;
  TYPE env = ARRAY 100 OF envr;
  VAR empty := array(100,NIL);
  PROCEDURE extend (e:env;s:STRING;i:INTEGER) : env
    BEGIN
      VAR c : INTEGER := 0;
      WHILE (e[c] <> NIL) DO
        c := c + 1;
      END;
      e[c] := envr { id = s; val = i: };
      RETURN e;
    END;
  PROCEDURE lookup (e:env,s:STRING) : INTEGER IS
    BEGIN
      VAR c : INTEGER := 0;
      VAR a : INTEGER := -1;
      WHILE (e[c] <> NIL) DO
        IF (e[c].id = s)
          a := e[c].val;
        c := c + 1;
      END;
      RETURN a;
    END;
END;
```


Interface vs. Implementation

Ideally, the client of an ADT is not supposed to know or care about its internal **implementation** details – only about its exported **interface**. Thus, it makes sense to separate the **textual** description of the interface from that of the implementation, e.g., into separate files.

• Specifications give the names of types, and the names and types of functions in the package.

• Bodies give the definitions of the types and functions mentioned in the specification, and possibly additional private definitions.

One advantage of this separation is that clients of module `X` can be **compiled** on the basis of the information in the specification of `X`, without needing access to the the body of `X` (which might not even exist yet!)

But many languages, particularly in the C/C++ tradition, don't make this separation very cleanly.

Is abstraction always desirable?

Although the idea of defining explicitly all the operators for a type makes good logical sense, it can get quite inconvenient.

Programmers are used to **assigning** values or passing them as **arguments** without worrying about their types. They may also expect to be able to **compare them**, at least for equality, without regard to type.

So most languages that support ADT's have built-in support for these basic operations, defined in a uniform way across all types.

They also usually have facilities for overriding the built-in definitions with type-specific versions. For example, built-in equality on `intset` is unlikely to work on **contents** of set, so probably want a type-specific equality operator. (Some of the complexity of C++ derives from this.)

Unfortunately, it is impossible to generate code for operations that move or compare data without knowing things like the **size** and **layout** of the data. But these are characteristics of the type's **implementation**, not its interface. So these "universal" operations break the abstraction barrier around type.

Thus, supporting these operations conflicts with separate compilation, often weakening support for the latter. The problem can also be solved, at some cost in efficiency, by treating **all** abstract values as fixed-size pointers to heap-allocated values.

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 it is possible to 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**, and few languages provide any support for making sure that the implementations adhere to more than a type specification.

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.

Parameterization

Often want to write a module definition that is **parameterized** over another type, e.g., sets of `elements`:

```
SIGNATURE set(element) IS
  TYPE set(element);
  PROCEDURE INSERT(s: set(element); e: element);
  PROCEDURE MEMBER(s: set(element); e:element)
                : BOOLEAN;
  PROCEDURE UNION(s,t: set(element)):set(element);
END;

MODULE set(element) : set(element) IS ... END;

MODULE intset = set(INTEGER); (* INSTANTIATE *)
```

Key question: does each instantiation require recompilation of module code? Yes, in Ada or C++; No, in ML.

Note that we can only parameterize sensibly over certain classes of types, e.g., `set(element)` only makes sense if there is an equality operator on `element`. (Haskell type classes.)