CS558 Programming Languages

Winter 2013

Lecture 7

Abstract Data Types

Can the user define genuinely new types with the same status as the built-in types?

Ideally, to mimic the behavior of built-in 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.

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.

Purely **functional** operators yield simpler and more elegant ADTs.

Example: Environments in OCaml

```
module type ENV =
sig
    type env
    val empty : env
    val extend : env -> string -> int -> env
    val lookup : env -> string -> int option
end

module Env : ENV =
struct
    type env = (string * int) list

    let empty = []
    let extend env k v = (k,v)::env
    let rec lookup env k =
       match env with
       | (k0,v0)::rest ->
          if k = k0 then Some v0 else lookup rest k
       | [] -> None
end (* Env *)
```

Algebraic Specification

If clients are to be able to use an ADT without knowing anything about the implementation, they need a full **specification** of the operations' behavior.

Type signatures give only a partial specification.

A standard approach is to add **axioms** describing the behavior of different combinations of axioms. Example:

```
ADT env
Signatures:
  empty : env
  extend : env -> key -> value -> env
  lookup : env -> key -> value option

Axioms:
  lookup empty k0 = None
  lookup (extend e k v) k0 =
    if k = k0 then Some v else lookup e k0
```

Another Example

```
ADT set
Signatures:
  empty : set
  insert : set -> elem -> set
  union : set -> set -> set
  member : set -> elem -> bool

Axioms: ...
```

Choosing Axioms

How many axioms are enough?

We can identify two important subsets of operations:

• **constructors** return new instances of the ADT.

• **observers** (or **inspectors**) take one or more instances of the ADT as arguments and return some other type(s) as result.

Example: for the `Env` ADT, the constructors are `empty` and `extend`; the sole observer is `lookup`.

The only way to create an ADT value is to call a constructor. So every ADT value can be built up inductively by applying constructors.

The only aspect of an ADT value that matters is how it behaves when passed to an observer. (We can't tell anything else about the value!)

So, it suffices if we give enough axioms to define the behavior of every observer on every possible combination of constructors.

Implementations from Axioms

It turns out that we can often use the axioms to build an implementation 'for free." The idea is to represent each value of the ADT by the sequence of constructors used to build it.

The resulting implementation may not be very efficient, but it can be useful for prototyping...

Example

```
module Env : ENV =
struct
    type env =
        Empty
      | Extend of env * string * int

    let empty = Empty
    let extend e k v = Extend(e,k,v)
    let rec lookup e0 k0 =
      match e0 with
      | Empty -> None
      | Extend(e,k,v) ->
          if k = k0 then Some v else lookup e k0
end (* Env *)
```

Observational Equivalence

We can use the axioms to prove the **observational equivalence** of two ADT values, even in cases where the representations of the values are different!

Example: suppose we have

e_1 `= extend (extend empty "a" 1) "b" 2`
e_2 `= extend (extend empty "b" 2) "a" 1`

Using the axioms, we can prove that, for any key k,

`lookup` e_1 k `= lookup` e_2 k

Hence e_1 and e_2 are observationally equivalent, even though they may have different representations (e.g. in the implementations we gave).

In conventional languages, axioms only have the status of **comments**. So reasoning using observational equivalence is dangerous unless we have proved that the actual implementation obeys the axioms; we can imagine systems that checked (or helped us check) this.

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.

For example, OCaml distinguishes **module types** (module specifications, or signatures) from **modules** (module bodies, or structures), and encourages them to be in 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!)

Many languages, particularly in the C/C++ tradition, don't make this separation very cleanly. Java doesn't support it cleanly either, even given the notion of interfaces (constructors are one sticking point).

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 expect to **assign** values or **pass** them as arguments without writing type-specific code for doing so. They may also expect to be able to **compare** them, at least for equality, without writing type-specific code.

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. (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 types, and conflicts with separate compilation.

One common, but slightly inefficient, solution is to **box** all abstract types.

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 OCaml

OCaml module definitions 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.)

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

Module Interfaces in ML

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

```
module type 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.

```
module LimitedMachine : MACH = Machine
```

Modules in Other Languages

Many languages provide one or more mechanisms for name space management. Terminology varies widely, e.g.:

• Python **modules**, which define separate namespaces, are associated with files. Hierarchical names spaces (sub-modules) can be defined using **packages**.

• Java **packages** define namespaces. It is also common to use **classes** to group together related (`static`) definitions.

• C **files** can be used to define a primitive form of namespace: `static` declarations are not visible outside the file. But all names exported from all files in the program occupy one global name space.

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 OCaml

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

```
let mksort (le : 'a -> 'a -> bool) : ('a list -> 'a list) =
  let rec sort = function
        | [] -> []
        | h::t -> insert h (sort t)
  and insert x = function
        | [] -> [x]
        | h::t -> if le h x then h::(insert x t) else x::h::t in
  sort

let le_pair (x:int*int) (y:int*int) : bool =
        fst x * snd x <= fst y * snd y
let sort_pairs : (int*int) list -> (int*int) list = mksort le_pair
let l = sort_pairs [(1,2);(3,0);(8,9)]
```

This extends (but only 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.

Most 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**, OCaml **functors**.

We can often think of a parameterized module as a **client** of some **service** provided by the parameter. Note that this gives us a way to typecheck and maybe even compile the client code before having an implementation of the service at all.

OCaml Functors Example

```
module type SorterArg =
sig
  type t
  val le: t -> t -> bool
end

module Sorter(SA:SorterArg) : sig
  type t = SA.t
  val sort : t list -> t list
end  =
struct
  type t = SA.t
  let rec sort = function
        | [] -> []
        | h::t -> insert h (sort t)
  and insert x = function
        | [] -> [x]
        | h::t -> if SA.le h x then h::(insert x t) else x::h::t
end

module PairsSorter = Sorter(struct
                             type t = int * int
                             let le = le_pair
                            end)
PairsSorter.sort [(1,2);(3,0);(8,9)]
```

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, C++, and .NET, 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 some ML implementations have been moving towards a code specialization approach to improve performance.