CS558 Programming Languages

Fall 2015

Lecture 9

"PROGRAMMING IN THE LARGE"

Language features for controlling types and bindings are crucial for writing **large programs**.

• **Multiple Namespaces** As programs grow large, we need a way to avoid conflicts in the choice of top-level names.

• **Encapsulation** Related definitions should be grouped together in a unit ("**module**", "package", "namespace", "class", etc.), and it should be possible to export just a subset of definitions from the module, as described by an **interface** specification.

• **Independent Modification** It should be possible to change the implementation of a module without requiring its **clients** to be rewritten (or, ideally, even recompiled).

These features enable us to work on a small part of a large system without needing to understand every detail of the whole.

SEPARATE PRIVATE NAMESPACES

The most basic aspect of a module is the ability to group together a set of definitions under an overall name that is then used to control their visibility in the rest of the program.

Example: Java **packages**

• `package foo;` declaration at top of a file says the classes in this file belong to a package named "`foo`". (Several files can contribute to the same package.)

• Only classes, methods, and fields declared `public` are visible from outside the package

• From outside the package, its elements can be referenced using "dot" notation, e.g. `foo.C`

• Or, clients can include the line `import foo;` after which the elements can be accessed directly, e.g. as just `C`

In other languages, modules can incorporate other kinds of top-level definitions, including functions, types, variables, constants, exceptions, etc.

HIDING AND ABSTRACTION

A module typically encapsulates a set of services or facilities for use by other parts of the program.

- Sometimes we just want a sane naming scheme, e.g. putting all the trigonometic functions in a library called `Math`.

Usually we also want to **abstract** over the services provided by the module, by hiding some implementation information (internal functions, type definitions, etc.) behind an **interface**.

Benefits of abstraction:

- Implementation and client can be developed **independently**.
- Implementation can be **changed** without affecting client's code.
- Improves clarity, maintainability, etc. of the code base.

INTERFACES

In many languages, the interface to a module is **implicitly** given using privacy modifiers on individual pieces of the implementation.

But some languages allow module interfaces to be specified **explicitly**, and (perhaps) in a separate file from the implementation.

These explicit interfaces usually consist of a set of top-level identifiers with their **type** signatures. This typically makes it possible to write, type-check, and compile client code based solely on the interface description.

(In some sense, the whole interface can be viewed as the type signature of the whole module.)

Of course, there must also be an (at least informal) specification of what the module's facilities **do**, but specifying this formally is a hard problem, and few languages provide any support for making sure that the implementations adhere to more than a type specification.

INTERFACE FILES

For example, in the OCaml language, each module `foo` typically lives in a file `foo.ml` and its interface in a separate file `foo.mli`.

`env.mli:`

```
type env
val empty: env
val extend: env -> string -> int -> env
val lookup: env -> string -> int option
```

`env.ml:`

```
type env = (string * int) list
let empty = []
let extend e k v = (k,v)::e
let rec lookup e k =
   match e with
   | (k',v')::e' when k = k' -> Some v'
   | _::e' -> lookup e' k
   | _ -> None
```

Clients of `env` can be compiled just using the information in `env.mli`, even if `env.ml` doesn't exist yet! And recompilation of `env.ml` at any point does not affect the client, so long as the interface is not changed: only relinking is required.

Here the interface exposes the existence of a type `env` but **not** its implementation as a list. This means that clients can only operate on values of type `env` using the three operators given in the interface. Hence `env` is an **abstract type**...

CLASSES AS MODULES

In class-based OO like C++ and Java, **classes** themselves can also play the role of modules. For example, in Java it is common to group together `static` methods and variables having a common theme into a class. (Indeed, they must go in **some** class.)

- e.g., the `Integer` contains static methods for for string-integer conversion and the `MAX_VALUE` constant.

- the same "dot" notation is used to reference these (e.g. `Integer.MAX_VALUE`).

- again, only class contents not labeled `private` can be accessed from outside the class

The role of interfaces can be played by **abstract classes**, i.e., class definitions that give the names and types of members, but defer their concrete implementation to subclasses. In Java, an **interface** is essentially a kind of abstract class.

PRIMITIVE VS. CONSTRUCTED TYPES REVISITED

Primitive types like `float`, `bool` and `->` (functions) are usually **abstract**:

- the internals of values cannot be inspected
- values can only be created and manipulated using a particular set of operators

For example, we test the sign of a floating point number using a relational operator like this:

```
if (a < 0) print "a is negative"
```

rather than trying to inspect the sign bit in the IEEE 754 representation directly:

```
if (a & 0x80000000 != 0) print "a is negative"
```

This helps makes programs **portable** across language implementations and hardware.

The same kind of abstraction can be very useful for **constructed** types too: it makes client code independent of type representation and implementation.

Abstract Data Types (ADT's)

Ideally, to mimic the behavior of built-in primitive 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.

Being able to define ADTs is an important test for a language's module facilities.

ADT Example: Environments

Consider an ADT for mutable environments mapping strings to values of some arbitrary type. Here's a signature (in a made-up abstract language):

```
type env_V
operator empty() returns env_V
operator extend(env_V,string,V) returns void
operator lookup(env_V,string) returns (Found(V) + NotFound)
```

Here `env_V` means an environment carrying values of type `V`, and `Found(V) + NotFound` is a disjoint union type.

One way to give a formal description of the desired interface behavior is to state some laws that we want the operators to obey, where $\{$ `stmts` $\}$ $\Longrightarrow$ `P` means "`P` is true after execution of `stmts`."

```
{ e = empty(); v = lookup(e,k) } ⟹ v = NotFound
{ v0 = lookup(e,k'); extend(e,k,v); v1 = lookup(e,k') } ⟹
    v1 = if k = k' then Found(v) else v0
```

We look at possible implementations of the ADT in C, Java, and OCaml.

C VERSION: INTERFACE

C doesn't have explicit module constructs, but separate compilation and header files can be used to simulate them (unsafely)

Here's an `env.h` file, to be `#include`d in both implementation and client:

```
struct envrep;
typedef struct envrep* Env;

Env empty(void);
void extend(Env e, char* k, void* v);
void* lookup(Env e, char* k);
```

- The `typedef` defines `Env` as a synonym for a **pointer** to the **incomplete** `envrep` structure type. This lets client compile without knowing details of `envrep`.
- We support polymorphism over values as long as they are pointers; C permits (unsafe!) casting of any pointer to/from `void*`.
- We use `NULL` (unreliably) to represent `NotFound`.

C VERSION: IMPLEMENTATION

A possible `env.c` file, using an array to represent environments:

```
#include "env.h"
#define SIZE 100

struct envrep {
  int count;
  char *keys[SIZE];
  char *values[SIZE];
};

Env empty(void) {
  Env e = malloc(sizeof(struct envrep));
  if (e == NULL) exit(EXIT_FAILURE);
  e->count = 0;
  return e;
}

static int find(Env e, char* k) {
  int i;
  for (i = 0; i < e->count; i++)
    if (e->keys[i] == k)
      return i;
  return -1;
}
```

```
void extend(Env e, char* k, void* v) {
  int i = find(e,k);
  if (i >= 0)
    e->values[i] = v;
  else {
    if (e->count >= SIZE)
      exit(EXIT_FAILURE);
    e->keys[e->count] = k;
    e->values[e->count] = v;
    e->count += 1;
  }
}

void* lookup(Env e, char* k) {
  int i = find(e,k);
  if (i >= 0)
    return e->values[i];
  else
    return NULL;
}
```

C VERSION: IMPLEMENTATION(2)

- This implementation can run out of room (a point not considered in our abstract definition of the ADT).
- It can also leak storage (we have not provided a way to delete an environment).
- C enforces that the `static` function `find` is not visible outside this file.
- All other names are visible externally, and must be unique across the entire program.
- The `envrep` structure is private only by convention.

C VERSION: CLIENT

```
#include <assert.h>
#include "env.h"

int main(void) {
  Env e = empty();
  extend(e,"a","alpha");
  extend(e,"b","beta");
  extend(e,"a","gamma");
  char* ax = (char*) lookup(e,"a");
  assert (strcmp(ax,"gamma") == 0);
  char* cx = (char*) lookup(e,"c");
  assert (cx == NULL);
}
```

- We must "downcast" the `void*` values returned by `lookup`.
- Nothing stops the client from including the concrete definition of `envrep` (or using a different definition altogether) and corrupting the array.

JAVA VERSION: INTERFACE

```
interface Env<V> {
  void extend(String k, V v);
  V lookup(String k);
}
```

- A Java `interface` is just a variant on an abstract class that can only contain instance methods (no variables or static members).
- Note that there is no `empty` method; we will need to use a (concrete) constructor to make new environments.
- Java generics handle the polymorphism straightforwardly.
- We again use `null` (unreliably) to represent `NotFound`

JAVA VERSION: IMPLEMENTATION

```
class ListEnv<V> implements Env<V> {
  private class Node {
    String key;
    V value;
    Node next;   // terminate with null
    Node(String key,V value,Node next) {
      this.key = key; this.value = value; this.next = next;
    }
  }

  private Node e;

  public ListEnv() {
    this.e = null;
  }

  public void extend(String k, V v) {
    e = new Node(k,v,e);
  }

  public V lookup(String k) {
    for (Node u = e; u != null; u = u.next)
      if (k.equals(u.key))
        return u.value;
    return null;
  }
}
```

JAVA VERSION: CLIENT

```
class EnvClient {
  public static void main(String argv[]) {
    Env<String> e = new ListEnv<String>();
    e.extend("a","alpha");
    e.extend("b","beta");
    e.extend("a","gamma");
    String ax = e.lookup("a");
    assert (ax.equals("gamma"));
    String cx = e.lookup("c");
    assert (cx == null);
  }
}
```

- Client must commit to a particular implementation (here `ListEnv`)
- But otherwise is completely isolated from implementation details.

OCaml Version: Interface and Implementation

env.mli:

```
type 'a t
val empty : unit -> 'a t
val extend : 'a t -> string -> 'a -> unit
val lookup : 'a t -> string -> 'a option
```

env.ml:

```
type 'a tree =
| Node of 'a tree * string * 'a * 'a tree
| Leaf

type 'a t = 'a tree ref

let empty () = ref Leaf
```

env.ml (continued):

```
let extend e k v =
    let rec ext e =
        match e with
        | Leaf -> Node (Leaf,k,v,Leaf)
        | Node (l,k0,v0,r) ->
            if k < k0 then
              Node(ext l,k0,v0,r)
            else if k > k0 then
              Node(l,k0,v0,ext r)
            else (* k = k0 *)
              Node(l,k,v,r) in
    e := ext (!e)

let lookup e k =  ...
```

- Compared to environment type in the previous OCaml example, this one is polymorphic (`'a` is a type variable) and mutable.
- The `ref` constructor creates a storage location (a box). The `!` operator fetches the contents of the location and the `:=` operator overwrites the contents.
- The environment is represented by a binary search tree.

OCaml Version: Client

For completeness:

```
let main =
  let e = Env.empty() in
  Env.extend e "a" "alpha";
  Env.extend e "b" "beta";
  Env.extend e "a" "gamma";
  assert (Env.lookup e "a" = Some "gamma");
  assert (Env.lookup e "c" = None);
```

Is abstraction always desirable?

Although the idea of defining explicitly all the operators for a type seems sensible and consistent, 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 – and sometimes also mechanisms for programmers to customize these.

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

A common fix is to treat **all** abstract values as fixed-size pointers to heap-allocated values, as seen in our examples.