CS558 Programming Languages

Fall 2023

Lecture 9a

Andrew Tolmach
Portland State University

Programming "in the large"

- Language features for modularity are crucial for writing large programs
- Enable programmers to work on small part of a large system without understanding every detail of the whole
- Idea: related definitions can be grouped together into a named unit
  - "module", "package", "class", "namespace", etc.
  - defining functions, types, variables, constants, exceptions, sub-modules, etc.
- Often possible to export just a subset of definitions, specified by an interface, for use outside the module
- Uses of definitions outside the module are qualified by module name

Example: Java packages

file1.java:

```
package foo;
public class A {
  A() {…B.y+C.z…}
}
class B {
  public static int x;
  static int y;
}
class C {
  static int z;
}
```

declares that all definitions in this file belong to package foo

makes declaration visible outside the package

other declarations are only visible inside the package

file2.java:

```
a = foo.A()
w = foo.B.x
```

from outside of package, elements are accessed using dot notation

file3.java:

```
import foo;
a = A()
w = B.x
```

or by importing entire package

Hiding and Abstraction

- A module typically encapsulates a set of services or a facility for use by other parts of the program
- Sometimes we just want a sane way to manage global name space
- Usually we also want to abstract over these services, by keeping some implementation info (internal functions, type definitions, etc.) hidden behind an interface
- Allows independent development of service and client
- Allows implementation to be changed without affecting client code
- Improves clarity, maintainability, etc. of the code base

Interfaces

- In many languages, interface to a module is given implicitly by using privacy modifiers on individual definitions
- Some languages allow interfaces to be specified explicitly, maybe in a separate file from the implementation
- An explicit interface is usually a set of top-level identifiers each with its type signature
  - Makes it possible to write, type-check and (maybe) compile client code based just on the interface
  - (In a sense, whole interface is type signature of whole module)
- Would also be nice to specify what interface elements do

but this is hard…

Abstract Data Types (ADT's)

- Can we apply same kind of abstraction to user-defined types to make client code independent of type representation and operator implementation?
- Ideally, user-defined types should have an associated set of operators, and clients should only be able to manipulate values via these operators
  - (and maybe a few generic operators such as assignment or equality testing)
- Clients should not be able to inspect or change the fields of the value representation
- Idea: try to implement ADTs using modules and interfaces

ability to do this is an important test for language's module facilities

ADT Example: Environments

- Consider an ADT for mutable environments mapping strings to values (of some arbitrary type), with this interface (in a made-up language):

environment carrying values of type v

disjoint union

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

- One way to give a formal description of the desired ADT behavior is to state some laws that we want the operators to obey:

```
{ 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
```

`{ stmts } ⇒ P` means "`P` is true after execution of `stmts`"

OCaml version: Interface and Client

env.mli:

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

.mli contains just the interface

env is explicitly polymorphic

encodes disjoint union

example-client.ml:

```
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);
```

clients can be compiled using just the information in .mli even if implementation doesn't exist yet!

.ml contains just the implementation

OCaml version: Implementation

env.ml:

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

type 'a t = 'a tree ref

let empty () = ref Leaf
```

type of boxes

creates a box

representation is a box containing a (pure) binary search tree

```
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 =  ...
```

recompiling .ml does not affect clients as long as .mli is unchanged; only relinking is needed

stores into a box

fetches from a box

Java version: Interface

declares list of methods with their type signatures

could use an `abstract class` instead

Java generics handle the polymorphism straightforwardly

Env.java:

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

We use `null` (unreliably) to represent NotFound

there is no `empty` method; we will need to use a (concrete) constructor to make new environments

Java version: implementation

ListEnv.java:

```
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;
  }
}
```

uses private linked list representation

creates new empty env

can only run out of space if entire Java program heap is full

Java garbage collector takes care of freeing environments when they are no longer accessible

Java version: client

EnvClient.java:

```
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

but is completely isolated from implementation internals

C version: Interface

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

header file to be #included in both implementation and clients

env.h:

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

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

Defines Env as a synonym for a pointer to the incomplete envrep structure type. This lets client compile without knowing details of envrep

We again use NULL (unreliably) to represent NotFound

Allow polymorphism over values as long as they are pointers; C permits (unsafe!) casting of any pointer to/from void*

C Version: Implementation

one possible implementation

env.c:

```c
#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;
}
```

```c
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;
}
```

`envrep` is private to this file only by convention (not enforced by C)

uses a pair of arrays to represent `env`

can run out of room (unlike our abstract ADT description)

C enforces that `static` function is invisible outside this file

all non-static functions are globally visible and must be unique across entire program

can leak storage: ADT has no operator to delete an environment, but C does not have garbage collection

C version: Client

example-client.c:

```
#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 in the C language prevents a bad client from including the concrete definition of `envrep` (or using a different definition altogether) and corrupting the representation arrays.

Universal operations

- Although the idea of defining all operators for an ADT explicitly seems sensible, it can get quite tedious for the ADT author!

- For every types, we will need a way to assign values or pass them as arguments. We may also expect to be able to compare them (at least for equality).

- So many languages that support ADTs have built-in support for these basic operations, defined in an uniform way across all types — and sometimes also mechanisms for ADT authors to customize them.

Too much abstraction?

- It is impossible for a compiler to generate client code for operations that move or compare data without knowing the size and layout of that data.
- But these are characteristics of the type's implementation, not its interface!
- So these "universal" operations break the abstraction barrier around the type and prevent separate compilation
- A common fix (seen in our examples) is to require all abstract values to be boxed, giving a simple universal implementation for assignment and equality comparison