CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 19: Modules and Abstract Data Types

12 Mar 01

Administration

- Programming Assignment 3 due today
- Programming Assignment 4 is online

Further topics

- Generating better code: optimization
  - register allocation
  - optimization (high- and low-level)
  - dataflow analysis
- Supporting language features
  - modules
  - objects
  - first-class functions
  - exceptions
  - parametric polymorphism
  - advanced GC techniques
  - dynamic linking, loading, & PIC
  - dynamic types and reflection

High-level languages

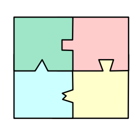
- So far: how to compile simple languages
  - Data types: primitive types, strings, arrays
  - **No** user-defined abstractions: objects
  - **No** first-class function values
- Next 3 lectures: modules, abstract data types, and objects (functions later)
  - semantic checking
  - code generation (IR and assembly)
  - Iota already has (simple) modules
  - Iota$^+$ (Programming Assignment 5) has abstract types, objects

Outline

- Goals of a module mechanism
  - Encapsulation
  - Abstraction
  - Separate compilation
- Related mechanisms
  - Records
  - ADTs
  - Abstract types

What is a module?

- A collection of named, related values and types
- Definitions partially hidden from the outside
- Program composed of separate modules
- Why have a module mechanism?
  - separate compilation: scalable
  - code reuse
  - namespace management
  - encapsulation
  - security
  - abstraction, abstract data types



- Java: classes, packages; C++: classes; Modula-[23], Iota, Iota+: modules; C: source files; ML: structures; CLU: clusters; standard Pascal: nothing

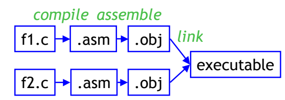
Separate Compilation

- Program is made up of several *compilation units* : independent inputs to compiler
- C: .c files; Java: .java files; Iota: .im
- Avoids recompiling whole program at every change
- Code more reusable
- Type safety: need interfaces

  C, C++: .h files; Java: .class file (!); Iota: .ii

Implementation: Linking

f1.c:
```
extern int x;
int f( ) {
  return x;
}
```
f2.c:
```
int x;
```



- **Problem:** can't generate code to access global x because its address is not known
- **Solution:** EXTRN x in f1.asm, PUBLIC x in f2.asm
  - f1.obj file contains 0 for address of x
  - linker glues together f1.obj, f2.obj,
  - fills in all EXTRN uses (0's) with actual addresses

Namespaces

- C, FORTRAN: all global identifiers visible everywhere
- **Problem:** can't have two global variables, functions with same name (Also: linker doesn't type-check)
- **Solutions:**
  - C++, Java: qualified identifiers (C.x where C is a class name or $P_1.P_2.P_3$.C.x)
  - Object code formats have flat namespace (usually): need way to *mangle* qualified identifiers

    C++: int C::f(int x) ⇒ f__1Ci
  - Modula-3, Iota: qualified identifiers + renaming
  - Java, Modula-3: link-time type checking

Encapsulation

- Don't want everything inside a module/compilation unit to be visible outside: encapsulation/information hiding

```
names: array[string]
passwords: array[string]
bool check_password(n, p: string) = (
   j: int = 0;
   while (j < length names) (
       if (names[j] == n & passwords[j] = p)
       return true else j=j+1); false))
```

- Can have security implications: internal data (names, passwords) protected by encapsulation; Java security based on encapsulation

Encapsulation mechanisms

- Need way to indicate which identifiers should be *exported* from a module
- Modula-3, Iota: separate module interface (.i3, .ii file)

```
names: array[string]
passwords: array[string]
bool check_password(n,p: string)
```

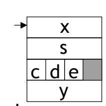
- C++, Java: public/private in module
- C, C++: "static" globals
- Assembly: PUBLIC declarations

Namespaces: records

- Records (C structs, Pascal records)
  - provide named fields of various types
  - usu. implemented as a block of memory

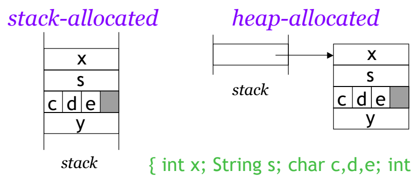
  type: {x:int, s: String, c,d,e: char, y: int }

  expr: {x = 2, s = "hi", c = 'x', ... y = 10 }
  - **efficient**: accesses to data members compiled to loads/stores indexed from start of record; compiler converts name of field to an offset.



Stack vs. heap

- Records have known size; can be allocated either on stack (e.g. C, Pascal) or heap
- Accesses to stack records are fp-relative -- don't need to compute address of record
- Stack allocation ⇒ cache coherence



{ int x; String s; char c,d,e; int y; }

Modules as records

- Record bundles values together, is a mapping from names to values
- Module looks like a 2^{nd}-class record value computed at load time
- Module interface looks like record *type*

```
mod = {
  names: array[string],
  passwords: array[string]
  check_password = (function(n,p: string): bool =
     (j: int = 0; while (j < length names) ( if (names[j] == n &
     passwords[j] = p) return true else j=j+1); false)))
  is_name = (function(n: string): bool = (...) )
}
mod : {check_password: string × string → bool,
       is_name: string → bool }
```


Abstract Data Types

- Not to be confused with Java "abstract"!
- Example: linked list type List
- Abstract operations:

```
length(l: List): int    cons(h: int, l: List): List
first(l: List): int     rest(l: List): List
```

implementation

```
List = {len: int,
        head: int,
        next: List}
length(l: List) = l.len
first(l: List) = l.head
```

interface

```
length(l: List): int
cons(h: int, l: List): List
first(l: List): int
rest(l: List): List
```


Hiding implementation

- Problem: can't write down interface length(l: List): int... unless List is defined as a type.
- Define List = {len: int, head: int, next: List}?
  - No: representation invariant that l.len = length(l) can be broken by any code that overwrites len
  - Want encapsulation for *values* exported by module, not just components inside module
- *Abstract type:*
  identifier representing an unknown type
  (*e.g.*, List)
- ADT = abstract type + function declarations + concrete type definitions + function implementations

Abstract types

Iota+abstract types, Modula-3 style:

list.ii: *declaration of abstract type*

```
type List;
length(l: List): int
cons(h: int, l: List): List
first(l: List): int
rest(l: List): List
```

list.im: *binding to actual type*

```
type List = {len: int, head: int, next: List}
length(l: List): int = l.len
cons(h: int, l: List): List = List{len=l.len+1,head=h,l=l}
...
```


Abstract types in C/C++:

list.h:

```
struct List;
int length(struct List *l);
List *cons(int h, struct List *t);
...
```

list.c:

```
struct List { int len, head; struct List *next; };
int length(struct List *l) { return l->len; }
struct List *cons(int h, struct List *t) {
   struct List *ret = new List; ret->head=h;
   ret->next = t; return ret; }
...
```

Classes in C++/Java

- Classes have private/public visibility modifiers that hide parts of object
- Class is a partially abstract type: some parts of type are known externally

```
class List {
  public static length(l: List): int
  public static cons(h: int, l: List):List
  public first(l: List): int
  public rest(l: List): List
  private int head, len;
  private List next; }
```

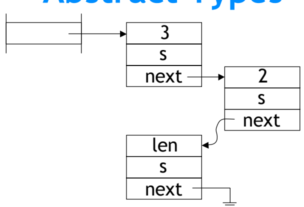

Implementing abstract types

- Representation is hidden from code other than the implementation of the type itself (CLU, Ada, ML, Modula-3)
- External code does not know representation, can't violate the abstraction boundary (e.g. break rep invariants)

+ Same interface can be reimplemented

– Compiler doesn't know representation either... can't stack-allocate w/o fancy linking

Abstract Types



- Implement just like heap-allocated records so representation always takes same size
- C++ objects are abstract types; can be stack-allocated. How does it work?

Private/Protected

- Objects in C++ are semi-abstract – interface file declares representation; method code is hidden from outside (mostly)

```
class List {
  private:  int len, String *s, List *l;
  public: int length( ); List *tail( ); ...
}
```

+ Allows outside code to know how much space List objects take, but not to access fields -- allows allocation on stack

– Downside: change to implementation can require complete recompilation