CS412/413

Introduction to
Compilers and Translators
Cornell University
Andrew Myers

Lecture 18: Modules and Abstract Data Types
8 March '00

Administration

- Programming Assignment 3 due Wednesday
- Prelim 2 date changed to *Thursday*, April 13 evening 7:30-9:30PM

A note about IR simplification

- Can do better job with extra rules
- Earlier rule for simplifying binary expressions:

$$\frac{T[\![e_1]\!] = (s_1,\ldots, s_m) \;;\; e'_1 \qquad T[\![e_2]\!] = (s'_1,\ldots, s'_n) \;;\; e'_2}{T[\![\, \mathrm{OP}(e_1, e_2)\,]\!] = (s_1,.., s_m, \mathrm{MOVE}(\mathrm{TEMP}(t), e'_1), s'_1,\ldots,s'_n) \;;\; \mathrm{OP}(\mathrm{TEMP}(t), e'_2)}$$

- Problem: often rips apart e'_1 and e'_2 into different statements unnecessarily

Optimized rules

- In case where e_2 has no side effects, no reason to split expression up:

$$\frac{T[\![e_1]\!] = (s_1,\ldots, s_m) \;;\; e'_1 \qquad T[\![e_2]\!] = () \;;\; e'_2}{T[\![\, \mathrm{OP}(e_1, e_2)\,]\!] = (s_1,.., s_m) \;;\; \mathrm{OP}(e'_1, e'_2)}$$

$$\approx (s_1,.., s_m, \mathrm{MOVE}(\mathrm{TEMP}(t), e'_1), s'_1,\ldots,s'_n); \mathrm{OP}(\mathrm{TEMP}(t), e'_2)$$

General rule

- If side effects of e_2 *commute* with e_1, transpose:

$$\frac{T[\![e_1]\!] = (s_1,\ldots, s_m) \;;\; e'_1 \qquad T[\![e_2]\!] = (s'_1,.., s'_n);\; e'_2 \qquad commutes(e'_1, s'_i) \;\; (\text{i in 1..n})}{T[\![\, \mathrm{OP}(e_1, e_2)\,]\!] = (s_1,.., s_m, s'_1,.., s'_n) \;;\; \mathrm{OP}(e'_1, e'_2)}$$

$$\approx (s_1,.., s_m, \mathrm{MOVE}(\mathrm{TEMP}(t), e'_1), s'_1,\ldots,s'_n); \mathrm{OP}(\mathrm{TEMP}(t), e'_2)$$

- Similar optimized rules can be produced for simplifying other IR nodes

Outline

- Goals of a module mechanism
  - Encapsulation
  - Abstraction
- Module mechanisms
  - Records
  - ADTs
  - Abstract types

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

Module goals

- Why have a module mechanism?
  - separate compilation: scalable
  - code reuse
  - namespace management
  - encapsulation
  - security
  - abstraction, abstract data types
- Java, C++: classes; Modula-[23], Iota, Iota+: modules; C: source files

Separate Compilation

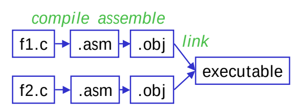
- Program is made up of several *compilation units* : independent inputs to compiler
- C: .c files; Java: .java files; Iota: .mod
- Avoids recompiling whole program at every change
- Code more reusable
- Type safety: need interfaces

  C: .h files; Java: .class file (!); Iota: .int

Implementation: Linking

<u>f1.c:</u>
```
extern int x;
int f( ) {
  return x;
}
```

<u>f2.c:</u>
```
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 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 $P_1.P_2.P_3.C.x$)
  - Need way to *mangle* qualified idents in assembly
  - 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, .int file)

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

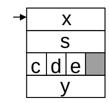
- C++, Java: public/private in module
- C, C++: “static” globals
- Assembly: PUBLIC declarations

Namespaces: records

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

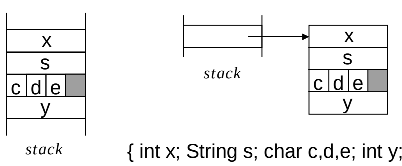
  type: {x:int, s: String, c,d,e: char, y: int }

  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 2nd-class record value computed at load time
- Module interface looks like record *type*

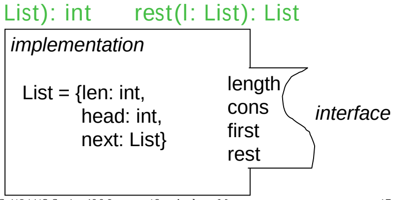
```
names: array[string]    passwords: array[string]
mod = {
    check_password = (function(n,p: string): bool =
        (j: int = 0; while (j < length names) ( if (names[j] == n &
        passwords[j] = p) return true else j=j+1); false)))
    is_name = (function(n: string): bool = (...) )
}
mod : {check_password: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



Hiding implementation

- Problem: can’t write down interface
  length(l: List): int   cons(h: int, l: List):
  List first(l: List): int   rest(l: List): List

unless List is defined.

- 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
- Pascal: ADT idea w/o language support
- Solution: *abstract types*, identifiers representing an unknown type

Abstract types

Iota+abstract types, Modula-3 style:

list.int: *declaration of abstract type*

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

binding to actual type

list.mod:

```
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

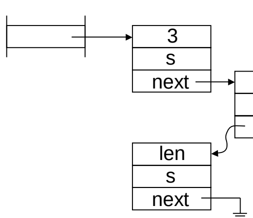
- Classes have private/public visibility modifiers that hide parts of object
- Class is a partially abstract type: some parts of type are known externally

```
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
- Problem: compiler doesn't know representation either... can't stack alloc.

Abstract Types



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

Private/Protected

- Objects in C++ are semi-abstract -- interface declares representation, only method code 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

Modules + abstract types

- Module is no longer a record: interface also contains list of abstract types
- Type: module($I_1..I_n$) { $v_i : T_i$ }

  ≡ type I_1 ... type I_n

  $v_1 : T_1 \ldots v_2 : T_2$
- Stripped-down module syntax:

  type $I_1 = T'_1, \ldots, I_n = T'_n$

  $v_i : T_i = e_i$

Multiple Implementations

- Most (non-OO) languages: only one implementation of (module value for) any interface
- Can have multiple impls of an interface using *first-class module values*
- Can also get multiple implementations via object-oriented subtyping (next time)

First-class module values

- List interface:

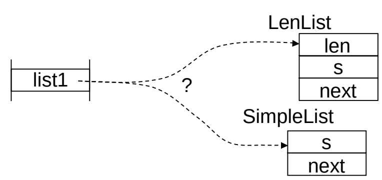
```
ListMod = module (T) {
   length: T→int, cons: int×T→T,
   first: T→int, rest: T→T }
SimpleList: ListMod = {
   type T = {head: int, next: T},
   length = function(l: T) = (/* recurse */), ... }
LenList: ListMod = {
   type T = {len: int, head:int, next: T},
   length = function(l: T) = l.len, ... }
```

- Must name module value *explicitly* rather than using name of interface: SimpleList.length, LenList.T instead of ListMod.length, ListMod.T

Implementing Multiple Implementations

- Problem: from interface, don't know which implementation we are dealing with.

L: ListMod, list1: L.T

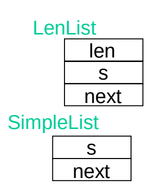


Compiling Multiple Impls

- Can't stack allocate -- need to know the *concrete type* of a reference (as in C++)
- Don't know what code to run when an operation (e.g. length) is invoked

L: ListMod, list1: L.T

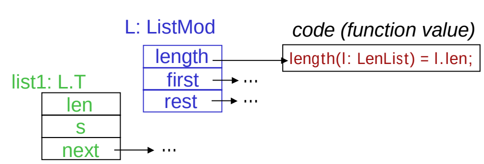
L.length(list1) – calls what?



Dispatch Vectors

- First-class module value is *dispatch vector* pointing to proper code
- For single implementation (2nd-class modules), linker makes module calls direct

L: ListMod, list1: L.T; list1.length()



Summary

- Variety of different mechanisms for providing data abstraction
- Increased abstraction power leads to more expensive implementations -- more indirections
- **Next time**: objects (similar to first-class modules), subtyping, inheritance, other object-oriented features