

CIS 341: Compilers

Lecture 21

The Plan

- Today:
 - Midterm exam review
 - Objects & (simple) compilation strategies
- Project 4: Code generation & typechecking for the “AT” (arrays and types) language
 - Due: *tonight* at 11:59 pm
- Project 5: Full OAT compiler: objects and extended typechecking
 - Available online soon
 - Due: November 7th at 11:59 pm

CIS 341: Compilers
2

Midterm Statistics

- Max: 50 Avg.: 35 Std.Dev: ~10

CIS 341: Compilers
3

Modules & Abstract Types

- Last time: we saw that modules with abstract types are like records with type components
- Stripped down ML-like syntax:
- Signatures:


```
sig
  type I1 ... type In
  val v1 : T1 ... val vm : Tm
end
```
- Modules:


```
struct
  type I1 = S1 ... type In = Sn
  let v1 : T1 = e1 ... let vk : Tk = ek
end
```

CIS 341: Compilers
4

Type Checking A Module

- Module definitions must agree with the interface in the signature
- Inside the module the concrete types are known
 - Extend the context with the definition (or substitute S_i for I_i)
- This rule also provides width subtyping

Module $E' = E, I_1 = S_1, I_2 = S_2, \dots, I_n = S_n$

$E' \vdash e_1 : T_1 \quad E' \vdash e_2 : T_2 \quad \dots \quad E' \vdash e_m : T_m \quad E' \vdash e_{m+1} : T_{m+1} \dots E' \vdash e_k : T_k$

$E \vdash$ **struct**

type I₁ = S₁ :

... :

type I_n = S_n :

let v₁ : T₁ = e₁ :

... :

let v_k : T_k = e_k :

end

sig

type I₁ :

... :

type I_n :

val v₁ : T₁ :

... :

val v_m : T_m :

end

CIS 341: Compilers
5

Objects as Abstract Data Types (ADTs)

- Objects: another way of extending records to ADTs
- Source code for the class defines the concrete types and implementation
- Interface defined either implicitly (via public members) or explicitly via interface ascription

```
class IntSet1 implements IntSet {
  private List<Integer> rep;
  public IntSet1() {
    rep = new LinkedList<Integer>();
  }
  public IntSet1 insert(int i) {
    rep.add(new Integer(i));
    return this;
  }
  public boolean has(int i) {
    return rep.contains(new Integer(i));
  }
  public int size() { return rep.size(); }
}
```

```
interface IntSet {
  public IntSet insert(int i);
  public boolean has(int i);
  public int size();
}
```

CIS 341: Compilers
6

Classes

- Fields or instance variables:
 - Values may differ from object to object (not shared)
 - Usually mutable
 - Presence inherited from the superclass
- Methods:
 - (Function) values shared among all instances of a class
 - Code inherited from the superclass
 - Immutable (usually)
 - Usually take an implicit argument that refers to the object itself (**this** or **self**)
- All components have visibility modifiers
 - public/private/protected (subclass visible)

CIS 341: Compilers 7

Code Generation for Objects

- Methods:
 - Generating method body code is similar to functions/closures
 - Generating method calls requires *dispatch*
- Fields:
 - Issues are the same as for records
 - Memory layout
 - Packing & alignment
 - Generating access code

CIS 341: Compilers 8

Multiple Implementations

- The same interface can be implemented by multiple classes:

```

interface IntSet {
    public IntSet insert(int i);
    public boolean has(int i);
    public int size();
}
    
```

```

class IntSet1 implements IntSet {
    private List<Integer> rep;
    public IntSet1() {
        rep = new LinkedList<Integer>();
    }
    public IntSet1 insert(int i) {
        rep.add(new Integer(i));
        return this;
    }
    public boolean has(int i) {
        return rep.contains(new Integer(i));
    }
    public int size() {return rep.size();}
}
        
```

```

class IntSet2 implements IntSet {
    private Tree rep;
    private int size;
    public IntSet2() {
        rep = new Leaf(); size = 0;
    }
    public IntSet2 insert(int i) {
        Tree nrep = rep.insert(i);
        if (nrep != rep) {
            rep = nrep; size += 1;
        }
        return this;
    }
    public boolean has(int i) {
        return rep.find(i);
    }
    public int size() {return size;}
}
        
```

CIS 341: Compilers 10

The Dispatch Problem

- Consider a client program that uses the IntSet interface:


```

IntSet set = ...;
int x = set.size();
            
```
- Which code to call?
 - IntSet1.size ?
 - IntSet2.size ?
- Client code doesn't know the answer.
 - So objects must "know" which code to call.
 - Invocation of a method must indirect through the object.

CIS 341: Compilers 10

Compiling Objects

- Objects are implemented by adding an extra pointer to a *dispatch vector* (also called a *virtual table* or *vtable*) with pointers to method code.

IntSet1
 rep:List

Dispatch Vector
 IntSet1.insert
 IntSet1.has
 IntSet1.size

IntSet2
 rep:Tree
 size:int

Dispatch Vector
 IntSet2.insert
 IntSet2.has
 IntSet2.size

set IntSet

?

Dispatch Vector
 ?.insert
 ?.has
 ?.size

CIS 341: Compilers 11

Method Dispatch

- Idea: every method has its own small integer index.
- Index is used to look up the method in the dispatch vector.

```

interface A {
    void foo();
}
            
```

Index
 0

```

interface B extends A {
    void bar(int x);
    void baz();
}
            
```

1
 2

Inheritance / Subtyping:
 A <: B <: C

```

class C implements B {
    void foo() {...}
    void bar(int x) {...}
    void baz() {...}
    void quux() {...}
}
            
```

0
 1
 2
 3

CIS 341: Compilers 12

Dispatch Vector Layouts

- Each interface and class gives rise to a dispatch vector layout.
- Note that inherited methods have identical dispatch indices in the subclass.

CIS 341: Compilers 13

Method Arguments

- Method bodies are compiled just like top-level procedures...
- ... except that they have an implicit extra argument: **this** or **self**
 - Historically (Smalltalk), these were called the "receiver object"
 - Method calls were thought of as sending "messages" to "receivers"

A method in a class...

```
class IntSet1 implements IntSet {
  IntSet1 insert(int i) { <body> }
}
```

... is compiled like this (top-level) procedure:

```
IntSet1 insert(IntSet1 this, int i) { <body> }
```

- Note 1: the type of "this" is the class containing the method.
- Note 2: references to fields inside <body> compiled like **this.field**

CIS 341: Compilers 14

Method Invocation Compilation

- Consider method invocation: `[[e.f(e1, ..., en)]]` ctxt
- First, compile `[[e]]` ctxt to get a (reference to) an object value.
 - Call this value obj
- Push the method arguments on the stack (right-to-left).
- Push the **this** argument (it's just obj) on to the stack.
- Compute dispatch vector address into a temporary
 - `dv = [obj]` (just dereference obj)
- Execute: Call `[dv + 4*i]`
 - Where i is method f's dispatch vector index i

CIS 341: Compilers 15

X86 Code For Dynamic Dispatch

- Suppose `b : B`
- What code for `b.bar(3)`?
 - `bar` has index 1
 - Offset = $4 * 1$

```

Mov eax, [[b]]
Push 3
Push eax
Mov ebx, [eax]
Mov ecx, [ebx + 4]
Call ecx
```

CIS 341: Compilers 16