Compiling Object Oriented Languages

Last time

- Undergraduate compiler review

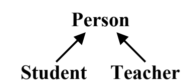
Today

- Introduction to compiling object oriented languages
- What are the issues?

What is an Object-Oriented Programming Language?

Objects

- Encapsulate code and data



Inheritance

- Supports code reuse and software evolution

```
PrintName(Person p);
```

Subtype polymorphism

- Can use a subclass wherever a parent class is expected

```
Person p = new Person;
Student s = new Student;

PrintName(p);
PrintName(s);
```

Dynamic binding (*message sends*)

- Binding of method name to code is done dynamically based on the dynamic type of the (receiver) object

```
p.reprimand();
```


Implementation: Inheritance of Instance Variables

Goal

- Lay out object for type-independent instance variable access

Solution (single inheritance)

- Prefixing: super-class fields are at beginning of object

Example

Person
Name

Student
Name
ID

Teacher
Name
Salary

Implementation: Dynamic Binding

Problem

- The appropriate method depends on the dynamic type of the object e.g., `p.reprimand()`

Solution

- Create descriptor for each class (*not* object) encoding available methods
- Encode pointer to class descriptor in each object
- Lay out methods in class descriptor just like instance variables

Person
getName

Student
getName
reprimand
workhard

Teacher
getName
reprimand
party

Usage summary

- Load class descriptor pointer from object
- Load method address from descriptor
- Jump to method

Why are Object-Oriented Languages Slow?

Dynamism
- Code
- Data

Style
- Granularity (lots of small objects)
- Exploit dynamism

Other reasons
- High-level (modern) features such as safety
- Garbage collection

Dynamism: Code

Dynamic binding
- What code gets executed at a particular static message send?
- It depends, and it may change

Example

? ?

```
class rectangle extends shape {
    int length() { ... }
    int width() { ... }
    int area() { return (length() * width()); }
}

class square extends rectangle {
    int size;
    int length() { return(size); }
    int width() { return(size); }
}
```

```
rect.area();
sq.area();
```


Cost of Dynamic Binding

Direct cost
- Overhead of performing dynamic method invocation

Indirect cost
- Inhibits static analysis of the code

Example

Want to inline and assign to registers, *etc.*

```
class rectangle:shape {
      int length() { ... }
      int width() { ... }
      int area() { return (length() * width()); }
}
```

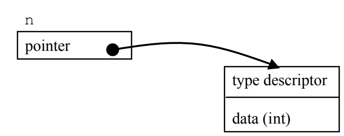
Driessen and Holzle (OOPSLA 96), in C++ programs median of 5.2% total execution time spent on dynamic dispatch

Dynamism: Data

Object instance types are not statically apparent
- Need to be able to manipulate uniformly
- Boxing: wrap up all data and reference it with a pointer

Example

```
Integer n = new Integer(33);
```



Cost of Dynamism: Data

Direct cost

- Overhead of actually extracting data
- *e.g.,* 2 loads versus 0 (if data already in a register)

Indirect cost

- More difficult to statically reason about data

Style

Sometimes programmers write C-style code in OO languages

- Easy: just "optimize" it away

Sometimes programmers actually exploit dynamism

- Hard: it can't just be "optimized away"

Programmers create many small objects

- Thwarts local analysis
- Exacerbates dynamism problem
- Huge problem for pure OO languages

Programmers create many small methods

- Methods to encapsulate data
- *e.g.* Methods to get and set member fields

A Concrete Example: Java

High-level and modern

- Object-oriented
- Mobile/portable (standard bytecode IL)
- Multithreaded (great for structuring distributed and UI programs)
- Garbage collected
- Dynamic class loading
- Reasonable exception system
- Rich standard libraries

Approaches to Implementing Java

Interpretation

- Extremely portable
  - Simple stack machine
- Performance suffers
  - Interpretation overhead
  - Stack machine (no registers)

Direct compilation

- Compile the source or bytecodes to native code
- Sacrifices portability
- Can have very good performance

Approaches to Implementing Java (cont)

JIT compilation

- Still supports mobile code (with more effort)
- Can have very good performance
  - Compilation time is critical
- Compiler can exploit dynamic information

JIT/Dynamic compilation

- Compiler gets several chances on the same code
- Compiler can exploit changes in dynamic information

Why is Java Slow?

Bytecode interpretation?

- Not a good answer

Why is Java Slow?

Impediments to performance

- Flexible array semantics
- Run-time checks (null pointers, array bounds, types)
- Precise exception semantics thwart optimization
- Dynamic class loading thwarts optimization
  - Even the class hierarchy is dynamic
- Multithreading introduces synchronization overhead
- Lots of memory references (poor cache performance)

  . . . and all the usual OO challenges

Analysis with a Dynamic Class Hierarchy

Approaches

- Ignore it (*i.e.,* disable dynamic class loading)
- Exploit final classes & methods
- Conservative optimization (*e.g.,* guarded devirtualization)
- Track validity of current code fragments and rebuild as necessary
  - *e.g.,* Resolution dependence graph
  - Necessitates JIT/dynamic compilation

Scientific Programming and Java

Consider matrix multiplication

```
for (i=0; i<m; i++)
    for (j=0; j<p; j++)
        for (k=0; k<n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Costs

- 6 null pointer checks (with just 2 floating point operations!)
- 6 index checks

Can we optimize this code?

- Precise exception model
  - Exception semantics inhibit removal or reordering
- No multidimensional arrays
  - Rows may alias

More on Matrix Multiplication

Why can't we just do this. . . ?

```
if (m <= C.size(0) && p <= C.size(1) &&
    m <= A.size(0) && n <= A.size(1) &&
    n <= B.size(0) && p <= B.size(1)) {
    for (i=0; i<m; i++)
        for (j=0; j<p; j++)
            for (k=0; k<n; k++)
                C[i][j] += A[i][k] * B[k][j];
} else {
    raise exception
}
```

No out-of-bounds checks, right?

Exceptions in Java

Exceptions in Java are precise

- The effects of all statements and expressions before a thrown exception must appear to have taken place, and
- The effects of all statements or expressions after a thrown exception must appear not to have taken place

Implications

- Must be very careful or clever when
  - Eliminating checks or
  - Reordering statements

Safe Regions [Moreira *et al.* TOPLAS 2000]

Idea

- Create two versions of a block of code
- One is guaranteed not to except and is optimized accordingly
- The other is used when the code might except

```
if (m <= C.size(0) && p <= C.size(1) &&
    m <= A.size(0) && n <= A.size(1) &&
    n <= B.size(0) && p <= B.size(1)) {
    for (i=0; i<m; i++)                    // safe region
        for (j=0; j<p; j++)
            for (k=0; k<n; k++)
                C[i][j] += A[i][k] * B[k][j];
} else {
    for (i=0; i<m; i++)                    // unsafe region
        for (j=0; j<p; j++)
            for (k=0; k<n; k++)
                C[i][j] += A[i][k] * B[k][j];
}
```

Java Arrays and Loop Transformations

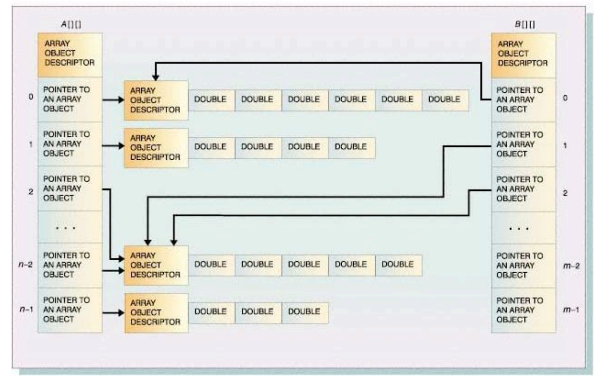
Java arrays

- No multidimensional arrays
  - Instead use arrays of arrays (can be ragged)
  - Requires one memory reference for each array dimension
- Rows may alias with one another

Arrays are common in scientific applications

- Their use requires optimization for good performance
- Large body of work on loop transformations makes assumptions
  - Arrays stored in contiguous memory
  - No aliasing among array elements
  - (Arrays are not ragged)

Java Arrays



IBM Systems Journal 39(1) © 2000 IBM

Summary

Implementing OOP requires handling ...

- member variables and inheritance
- dynamic binding due to polymorphism

Some OOP features that lead to inefficient code

- dynamic code and data
- programming style (ie. use of dynamism and small function bodies)
- safety (ie. array bounds checks and precise exceptions)
- garbage collection

Many sources of inefficiency in Java

- The cost of a cleaner object-oriented language

Progress in improving Java efficiency

- Greatest performance boost comes from eliminating interpretation overhead
- Scientific application performance (ie. extending language to include multi-dim arrays)

Next Time

Assignments

- Project 1 due Friday

Reading

- Garbage collection readings on web and in book

Lecture

- Garbage collection