

Compiling Object Oriented Languages

Last time

- Program slicing

Today

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

What is an Object-Oriented Programming Language?

Objects

- Encapsulate code and data

Inheritance

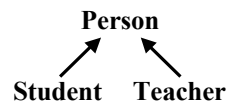
- Supports code reuse and software evolution (kind of)

Subtype polymorphism

- Can use a subclass wherever a parent class is expected

Dynamic binding (*message sends*)

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



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

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

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

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

Implementation: Inheritance of Instance Variables

Goal

- Lay out object for type-independent instance variable access

Solution

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

Example

Person
Name

Student
Name
ID

Teacher
Name
Salary

Multiple inheritance?

- May need to leave blanks
- Use graph coloring (one node for each distinct field, edge between coexistent fields, color indicates layout position)

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
- Store 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

What is a Pure Object-Oriented Programming Language?

Everything is an object

- Even numbers, strings, constants, *etc.*

All work achieved by sending messages to objects

- Even simple arithmetic and control flow

Example

```
if ( &x.eq(3) ,  
    &a.set(a.plus(1)) ,  
    &a.set(a.minus(1))  
);
```

Invoke *x*'s equal method

Pass closures to the *if* method
to create a control flow construct

Pass *then* computation

Pass *else* computation

Very clean and simple

- But very inefficient if naively implemented

Why are Object-Oriented Languages Slow?

Dynamism

- Code
- Data

Style

- Granularity (lots of small objects)
- Exploit dynamism

High-level (modern) features

- Closures & non-LIFO activation records
- Safety, *etc.*

Garbage collection

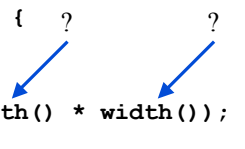
Dynamism: Code

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); }  
}
```

What happens with the following?

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

Cost of Dynamic Binding

Direct cost

- Overhead of performing dynamic method invocation

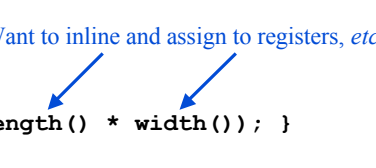
Indirect cost

- Inhibits static analysis of the code

Example

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

Want to inline and assign to registers, etc.



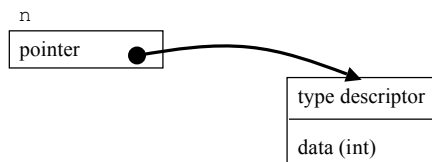
Dynamism: Data

Object instance types are not statically apparent

- Need to be able to manipulate all objects uniformly
- Boxing: wrap 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: optimize in usual ways

Sometimes programmers actually exploit dynamism

- Hard: new optimization problems

Programmers create many small objects

- Thwarts global 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

Modern High-level Features

Closures and non-LIFO activation records

- Leads to much heap allocation of data

Example

```
foo (Integer i) {  
    Integer n;  
    ...  
    return (&{n+i});  
}
```

A Concrete Example: Java

High-level and modern

- Object-oriented (not pure, but more pure than C++)
 - Granularity of objects and methods can be large or small
- Mobile (standard bytecode IR)
- Multithreaded (great for structuring distributed and UI programs)
- Garbage collected
- Dynamic class loading
- Reasonable exception system
- Rich standard libraries

Why is Java Slow?

Bytecode interpretation?

- Not a good answer

Approaches to Implementing Java

Interpretation

- Extremely portable
 - Simple stack machine
- Performance suffers
 - Few optimization opportunities
 - 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
- These systems are now quite sophisticated and effective

Approaches to Implementing Java (cont)

Custom processor

- Direct hardware support of Java bytecodes
- This has proven to be an impractical approach
 - See “Retrospective on High-Level Language Computer Architecture” by Ditzel and Patterson (ISCA 1980)
- But maybe some hardware support (*e.g.*, for GC) is a good idea?

Hybrids

- JIT and Interpretation
- Direct compilation and interpretation

Same-context translation

- Source-to-source or bytecode-to-bytecode

Why is Java Slow?

Impediments to performance

- Dynamic class loading thwarts optimization
 - Even the class hierarchy is dynamic
- Flexible array semantics
- Run-time checks (null pointers, array bounds, types)
- Precise exception semantics thwart optimization
- 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

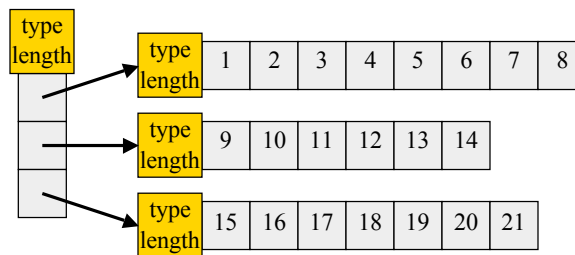
- Their use requires optimization for good performance
- Large body of work on loop transformations makes assumptions
 - Arrays stored in contiguous memory
 - No aliasing among array elements
 - (Arrays are not ragged)

Comparing Arrays

A 2D array in C

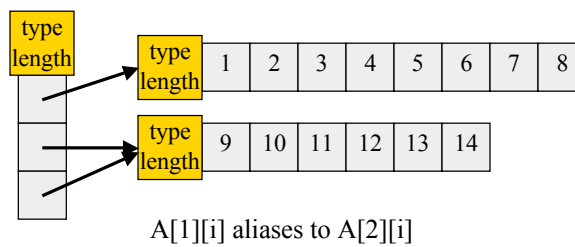
1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24

An array of arrays in Java



Java Arrays

Elements within an array can alias with one another

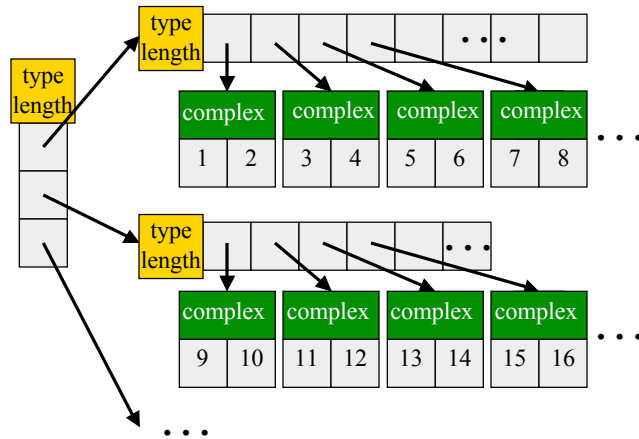


Implications?

- Complicates dependence testing

Java Arrays (cont)

An array of arrays of complex numbers



What are the implications of this structure?

Semantic Expansion [Artigas *et al.* LCPC '99]

Idea

- Introduce a new final array class with simpler semantics
- Treat the new class as a primitive in the compiler

```
doubleArray2D C = new doubleArray2D(m,p);
doubleArray2D A = new doubleArray2D(m,n);
doubleArray2D B = new doubleArray2D(n,p);

for (i=0; i<m; i++)
  for (j=0; j<p; j++)
    for (k=0; k<n; k++)
      C.set(i,j,C.get(i,j)+A.get(i,k)*B.get(k,j));
```

Look at this ugly syntax.

Semantic Expansion (cont)

Pros

- Yields good performance
- Doesn't officially change the language
- Can be used for other pseudo primitive classes (e.g., Complex)

Cons

- Inelegant (ugly syntax)
- Not general
- Does in fact change the language
- Loses syntactic benefits of true primitives
- At odds with the spirit of the language

Concepts

Dynamism

- Direct costs
- Indirect costs

Exception semantics

Array semantics

Object overhead

Next Time

Reading

- [Arnold05]

Lecture

- Addressing OO inefficiencies