Implementing Higher-Level Languages

Quick tour of programming language implementation techniques.
From the Java level to the C level.
Ahead-of-time compiler

compile time

C source code → C compiler → x86 assembly code → x86 assembler → x86 machine code

run time

x86 machine code → x86 computer → Output

Data

Figures for compilers/runtime systems adapted from slides by Steve Freund.
Interpreter

Source Program

Data

Interpreter = virtual machine

Output
Compilers... that target interpreters

Java source code \rightarrow \text{Java Compiler} \rightarrow \text{Java bytecode}

Java bytecode \rightarrow \text{Data} \rightarrow \text{Java Virtual Machine} \rightarrow \text{Output}
Interpreters... that use compilers.
JIT Compilers and Optimization

Java source code → javac → Java bytecode → Data → JVM

Just-in-time compiler

Performance Monitor

Bytecode interpreter

Output

- HotSpot JVM
- Jikes RVM
- SpiderMonkey
- v8
- Transmeta
- ...
Data in Java

Arrays

Every element initialized to 0 or null
Immutable length field

Since it has this info, what can it do?

int array[5]:

C

Java

Data Representation in Java
Data in Java

Arrays

Every element initialized to 0 or null
Immutable length field

Bounds-check every access.

Bounds-checking sounds slow, but:
1. Length is likely in cache.
2. Compiler may store length in register for loops.
3. Compiler may prove that some checks are redundant.

int array[5]:

C

0 4 20 24

Java

5 00 00 00 00 00
Data in Java

Characters and strings

16-bit Unicode
Explicit length, no null terminator

the string ‘CS 240’:

C: ASCII

Java: Unicode
Data structures (objects) in Java

C: programmer controls layout, inline vs. pointer.
Java: objects always stored by reference, never stored inline.

C
```c
struct rec {
    int i;
    int a[3];
    struct rec *p;
};
struct rec *r = malloc(...);
struct rec r2;
r->i = val;
r->a[2] = val;
r->p = &r2;
```

Java
```java
class Rec {
    int i;
    int[] a = new int[3];
    Rec p;
    ...
}
```
```java
r = new Rec();
r2 = new Rec();
r.i = val;
r.a[2] = val;
r.p = r2;
```

Data Representation in Java
Pointers/References

Pointers in C can point to any memory address

References in Java can only point to [the starts of] objects

And can only be dereferenced to access a field or element of that object
class Point {
    int x;
    int y;

    Point() {
        x = 0;
        y = 0;
    }

    boolean samePlace(Point p) {
        return (x == p.x) && (y == p.y);
    }

    String toString() {
        return "(" + x + "," + y + ")";
    }
}
Java objects

For each class, compiler maps: field signature $\rightarrow$ offset (index)

**vtable pointer**: points to per-class **virtual method table (vtable)**

For each class, compiler maps: method signature $\rightarrow$ index
- samePlace: 0
- toString: 1
Implementing dynamic dispatch

Java:

Point\ p = new Point();
Point\* p = calloc(1,sizeof(Point));
p->header = ...;
p->vtable = Point_vtable;
Point_constructor(p);
return p.samePlace(q);

what happens (pseudo code):

Point* p = calloc(1,sizeof(Point));
p->header = ...;
p->vtable = Point_vtable;
Point_constructor(p);
return p->vtable[0](this=p, q);
Subclassing

```java
class ColorPoint extends Point{
    String color;
    boolean getColor() {
        return color;
    }
    String toString() {
        return super.toString() + "[" + color + "]";
    }
}
```

How do we access superclass pieces?
- fields
- inherited methods

Where do we put extensions?
- new field
- new method
- overriding method
dynamic (method) dispatch

Java:

Point p = ???;
return p.toString();

what happens (pseudo code):
return p.vtable[1](p);

![Diagram showing dynamic method dispatch]

- **Point** object
  - vtable
  - x
  - y

- **ColorPoint** object
  - vtable
  - x
  - y
  - color

- **Point vtable**
  - code:Point.samePlace()
  - code:Point.toString()

- **ColorPoint vtable**
  - code:ColorPoint.toString()
  - code:ColorPoint.getColor()