Implementation

Edward Z. Yang
(intro)
Input → Program → Output
Source Program

Input → Interpreter → Output

Interpreted Languages
Python, Ruby, PHP, Perl, MATLAB...

Source Program

Ahead-Of-Time Compiler

Compiled Languages
C, C++, Haskell, Go, ML, Rust

Input -> Program -> Output
Source Program

JIT-Interpreted Languages

Javascript (V8, TraceMonkey)

Interpreter

Lexer/Parser

Semantic Analyzer

Just-in-time Compiler

Input → Output

Languages often support multiple implementation
- Garbage Collection

- Dynamic Dispatch in a JIT
  compare with C++ vtables
I COULD BE BOUNDED IN A NUTSHELL
AND CALL MYSELF THE KING OF INFINITE SPACE

HAMLET ACT II SCENE II
Lambda Calculus

Activation-record Model

x86 machine architecture

Memory Hierarchy
INFINITE MEMORY

FINITE MEMORY
INFINITE MEMORY

GARBAGE COLLECTION

FINITE MEMORY

& reuse space which probably will never be used again
Managed Memory

Memory Management
<table>
<thead>
<tr>
<th>allocate</th>
<th>Interface for allocating objects. Pointers are opaque.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Garbage collection / Reference counting</td>
<td></td>
</tr>
<tr>
<td>malloc free</td>
<td>Interface for explicitly allocating/deallocating finite memory</td>
</tr>
</tbody>
</table>

Operating System / Hardware
Garbage Collection

- Reference Counting
  ARC, Perl, PHP, Python
  The Cycle Problem

- Tracing Collection
  Java, Haskell, ML, Lisp, Go, JavaScript
  Mark and Sweep
  Copying Collection
When I am done using an object, free its memory
When I am done using an object, free its memory.
IDEAL
Object has no causal influence on future program execution

When I am done using an object, free its memory
The Model

- Stack Pointer
- Registers
- Static Data

Root Set

Heap
The Model

Live Data
- stack pointer
- registers
- static data

Root Set

Garbage

Heap

Why must pointer arithmetic be disallowed?
Reference counting

Count the number of incoming references

Root Set

- A
- B
- C
- D
- E
Reference counting

Count the number of incoming references

Root Set

A

B

C

D

E

1

2

∅
Reference counting

Count the number of incoming references

Root Set

A -> B
A -> C

∅

1

1

1

D

E
Reference counting

Count the number of incoming references

Root Set

A

B

C
Reference counting

Count the number of incoming references

Root Set

Diagram:

- B
- C
- A
- E (empty set)
Reference counting

Count the number of incoming references

Root Set
Reference counting

Count the number of incoming references

Root Set
Reference counting

Count the number of incoming references

Root Set
Reference counting

- Very easy to implement
- Objects immediately deallocated
- Cycles never die! (cycle-breaking)
- Storing & updating counts is costly
- Synchronizing updates
Reference counting

- Count the number of incoming references

- Very easy to implement
- Objects immediately deallocated

- Cycles never die! (cycle-breaking)
- Storing & updating counts is costly
- Synchronizing updates
Mark and Sweep

Traverse object graph for live objects

root set

root set
Mark and Sweep

Traverse object graph for live objects

root set

root set

Todo: B
Mark and Sweep

Traverse object graph for live objects

Todo: (nothing)

root set

root set
Mark and Sweep

Sweep memory for dead objects

root set
Mark and Sweep

Sweep memory for dead objects

root set
Mark and Sweep

Sweep memory for dead objects

root set
Mark and Sweep

Sweep memory for dead objects

root set
Mark and Sweep

Sweep memory for dead objects

root set
Mark and Sweep

Sweep memory for dead objects

free list

root set
Mark and Sweep

- Cycles are handled
- No extra bookkeeping

- Naively needs to traverse entire heap
- Naively leads to fragmentation (can compact)

- Needs to store a mark bit
- Needs to maintain TODO list

- Stop-the-world GC (could refcounting pause?)

---

Traverse object graph for live objects

Sweep memory for dead objects
Mark and Sweep

- Cycles are handled
- No extra bookkeeping
- Naively needs to traverse entire heap
- Naively leads to fragmentation (can compact)
- Needs to store a mark bit
- Needs to maintain TODO list
- Stop-the-world GC (could re-counting pause?)

Traverse object graph for live objects
Sweep memory for dead objects
Copying Collection

TO-SPACE

unscanned

FROM-SPACE

root set
Copying Collection

FROM-SPACE

A

unscanned

root set

TO-SPACE
Copying Collection

TO-SPACE

FROM-SPACE

unscanned

root set
Copying Collection

FROM-SPACE

To-SPACE

unscanned

root set
Copying Collection

FROM-SPACE

E | D | B | A | C

root set

unscanned

TO-SPACE
Copying Collection

FROM-SPACE

root set

TO-SPACE

unscanned
Copying Collection

✓ Compacts data (better locality)
✓ Constant space bookkeeping
× Needs x2 available space
× (Still) Stop-the-world GC
Summary: Garbage Collection

- Provide the ILLUSION of infinite memory
- Liveness based on reachability
- Generational GC (it’s hard!)
Dynamic Dispatch
Recap: C++ Virtual Tables
Recap: C++ Virtual Tables

Virtual Method Call

obj -> vtable [2]

virtual offset

<table>
<thead>
<tr>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
</tr>
<tr>
<td>h</td>
</tr>
</tbody>
</table>

vtable

C++ goal: Make virtual dispatch as efficient as possible
Recap: C++ Virtual Tables

class A : B, C

Consequence: Multiple inheritance, but no interfaces
Recap: C++ Virtual Tables

Motivating Problem

class A {
    virtual void f();
    virtual void g();
}  
class B {
    virtual void g();
    virtual void f();
}

Naive solution: Do a dictionary lookup
Recap: C++ Virtual Tables

Motivating Problem

```
class A {
    virtual void f();
    virtual void g();
}
class B {
    virtual void g();
    virtual void f();
}
```

C++ says: Do both layouts
Recap: C++ Virtual Tables

Motivating Problem

class A {
    virtual void f();
    virtual void g();
}
class B {
    virtual void g();
    virtual void f();
}

Today: Do a dictionary lookup and cache it
Briefly:

**JVM**

- **Loader**
  - On-demand class loading
  - Search FS for object
  - Can override default class loader

- **Verifier**
  - Check if bytecode is valid
  - Valid opcode
  - Valid jump targets
  - Well-typed

- **Linker**
  - Add class/interface to runtime
  - Initialize static fields
  - Resolve names

- **Interpreter/JIT**
  - Runtime checks (e.g., bounds checks)
Briefly: JVM

Bytecode is for a stack machine

class A {
    int i;
    void f(int val) {
        i = val + 1;
    }
}

aload 0; object ref this
iload 1; int val
iconst 1
iadd ; add val + 1
putfield #4 <Field int i>
return
Dynamic Dispatch in the JVM

1. invokevirtual

2. invokevirtual

3. invokevirtual

(byted Code rewriting)

(inline caches)

(polymorphic inline caches)

(or Smalltalk or Self)
invokevirtual

A x;
...
x→foo();
invokevirtual

class A {
  virtual void foo();
  virtual void bar();
}

A x;
...
x->foo();

obj->vtable[0]

foo
bar

dependency

in C++
invokevirtual

class A {
  virtual void bar();
  virtual void foo();
}

A x;
...
x->foo();

 obj->vtable[0]

update

bar

foo

in C++

dependency
invokevirtual

```
A x;
...
x->foo();
```

class A {
    virtual void bar();
    virtual void foo();
}

obj->vtable[1]

```
bar
foo
```

in C++

dependency

recompile
invokevirtual

A x;
...
x→foo();

class A

void bar() { ... }
void foo() { ... }

invokevirtual "A.foo"

in Java

A.class

typechecked against

verified against
invokevirtual

A x;
...
x→foo();

class A {
    void bar() {
        ...
    }
    void foo() {
        ...
    }
}

invokevirtual “A.foo”

A.class

in Java
invokevirtual

A x;
...
x->foo();

class A {
  void bar() {
  }
  void foo() {
  }
}

invokevirtual "A.foo"

A.class

re-verify

in Java
invokevirtual

A x;
...
x->foo();

class A {
    void bar() {
    }
    void foo() {
    }
}

invokevirtual "A.foo"

in Java

but no recompilation!

re-verify
invokevirtual

```
A x;
...
x->foo();
```
invokevirtual

A x;
... x->foo();

class A {
    void bar() { ... }
    void foo() { ... }
}

invokevirtual "A.foo"

A.class

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>foo</td>
</tr>
</tbody>
</table>

in Java

manual lookup
invokevirtual

A x;
...  
x→foo();

class A {
    void bar() {
    }
    void foo() {
    }
}

inv_virt_quick 1

in Java
Big Idea #1: Rewrite code to make it more efficient

invokevirtual "A.foo" → inv_virt_quick 1

;; fast, C++-like machine code
What about Interfaces?
invokeinterface

A x;
...
x->foo();

interface A {?
    void bar();
    void foo(); ?
}

class B implements A {?
    ...
}

B.class  C.class

invokeinterface "A.foo"

Rewrite me...
invoke interface

A x;
...  
\( x \rightarrow \text{foo}() \);

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

class B implements A {
  ...
}

B.class

<table>
<thead>
<tr>
<th></th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
</tr>
</tbody>
</table>

C.class

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>foo</td>
</tr>
</tbody>
</table>

???
to what?!
invoke interface

```
A x;
...
x -> foo();
```

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

class B implements A {
  ...
}

B.class

<table>
<thead>
<tr>
<th></th>
<th>foo</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>bar</td>
</tr>
</tbody>
</table>

C.class

<table>
<thead>
<tr>
<th></th>
<th>bar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Ø</td>
<td>foo</td>
</tr>
</tbody>
</table>
invokeinterface

```
inv_int_quick "A.foo" <B.foo addr>
```

if (this.class == B) {
    fastpath: directly invoke <B.foo addr>
} else {
    slowpath: invoke interface "A.foo"
}
Big Idea #2: A cache lookup can be built into the rewritten code, an inline cache.
What if this is the ONLY class loaded which implements A?

(Singleton class)
invoke interface

inv_int_quicker \langle B.\text{foo addr} \rangle

\Rightarrow

fastpath: directly invoke \langle B.\text{foo addr} \rangle

call on A will always be B, omit conditional
Corollary: Rewritten code does not have to be fully general, if you invalidate it when necessary.

→ Class Hierarchy Analysis
invokedynamic

In dynamic languages, usually have <10 distinct underlying types

Big Idea #3: Cache them all!
invokedynamic: Polymorphic Inline Cache

slow: invoke dynamic lookup handler
invokedynamic : Polymorphic Inline Cache

```java
if (this.class == A) {
    directly invoke <A handler>
} else {
    slow: invoke dynamic lookup handler
}
```
invokedynamic : Polymorphic Inline Cache

if (this.class == A) {
    directly invoke <A handler>
if (this.class == B) {
    directly invoke <B handler>
} else {
slow: invoke dynamic lookup handler
}
invokedynamic: Polymorphic Inline Cache

if (this.class == A) {
    directly invoke \( <A \) handler \)
}
if (this.class == B) {
    directly invoke \( <B \) handler \)
}
if (this.class == C) {
    directly invoke \( <C \) handler \)
} else {

slow: invoke dynamic lookup handler
}
invokedynamic: Polymorphic Inline Cache

if (this.class == A) {
    directly invoke <A handler>
}
if (this.class == B) {
    directly invoke <B handler>
}
if (this.class == C) {
    directly invoke <C handler>
} else {
    slow: invoke dynamic lookup handler
}
invokedynamic: Polymorphic Inline Cache

originated in Self
dynamically typed OOP language

PICs $\rightarrow 37\%$ perf improvement
Summary:

JIT codegen = Flexibility

allows Java/JavaScript engines to avoid paying too much for indirection
(end)