

VMs, Interpreters, JIT & Co

A First Introduction

Marcus Denker



Overview

- Virtual Machines: How and Why
 - Bytecode
 - Garbage Collection (GC)
- Code Execution
 - Interpretation
 - Just-In-Time
 - Optimizing method lookup
 - Dynamic optimization



Caveat

- Virtual Machines are not trivial!
- This lecture can only give a very high-level overview
- You will not be a VM Hacker afterwards ;-)
- Many slides of this talk should be complete lectures (or even courses)



Virtual Machine: What's that?

- Software implementation of a machine
- Process VMs
 - Processor emulation (e.g. run PowerPC on Intel)
 - FX32!, MacOS 68k, powerPC
 - Optimization: HP Dynamo
 - High level language VMs
- System Virtual Machines
 - IBM z/OS
 - Virtual PC



High Level Language VM

- We focus on HLL VMs
- Examples: Java, Smalltalk, Python, Perl....
- Three parts:
 - The Processor
 - Simulated Instruction Set Architecture (ISA)
 - The Memory: Garbage Collection
 - Abstraction for other OS / Hardware (e.g, I/O)
- Very near to the implemented language
- Provides abstraction from OS



The Processor

- VM has an instruction set. (virtual ISA)
 - Stack machine
 - Bytecode
 - High level: models the language quite directly
 - e.g, “Send” bytecode in Smalltalk
- VM needs to run this Code
- Many designs possible:
 - Interpreter (simple)
 - Threaded code
 - Simple translation to machine code
 - Dynamic optimization (“HotSpot”)



Garbage Collection

- Provides a high level model for memory
- No need to explicitly free memory
- Different implementations:
 - Reference counting
 - Mark and sweep
 - Generational GC

- A modern GC is **very** efficient
- It's hard to do better



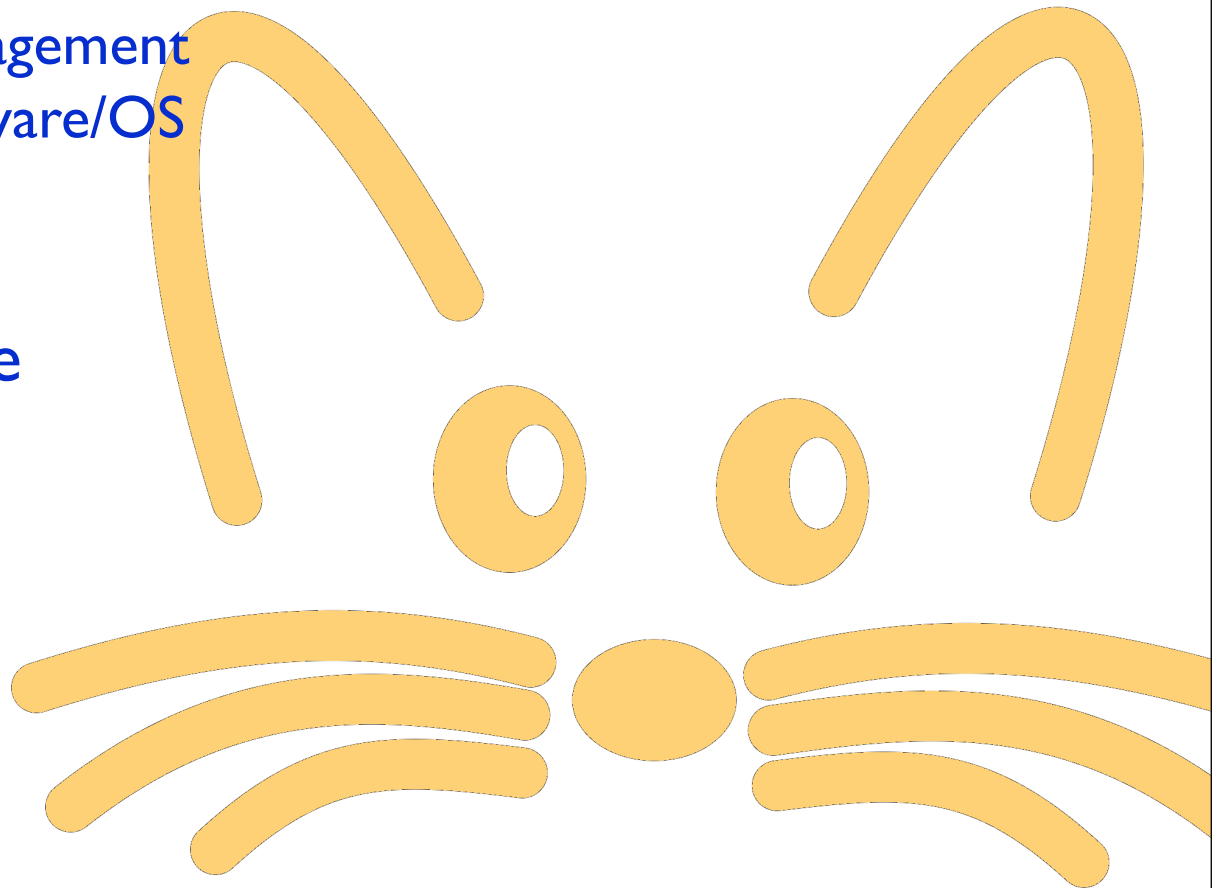
Other Hardware Abstraction

- We need a way to access Operating System APIs
 - Graphics
 - Networking (TCP/IP)
 - Disc / Keyboard....
- Simple solution:
 - Define an API for this Hardware
 - Implement this as part of the VM (“Primitives”)
 - Call OS library functions directly



Virtual Machine: Lessons learned

- VM: Simulated Machine
- Consists of
 - Virtual ISA
 - Memory Management
 - API for Hardware/OS
- Next: Bytecode

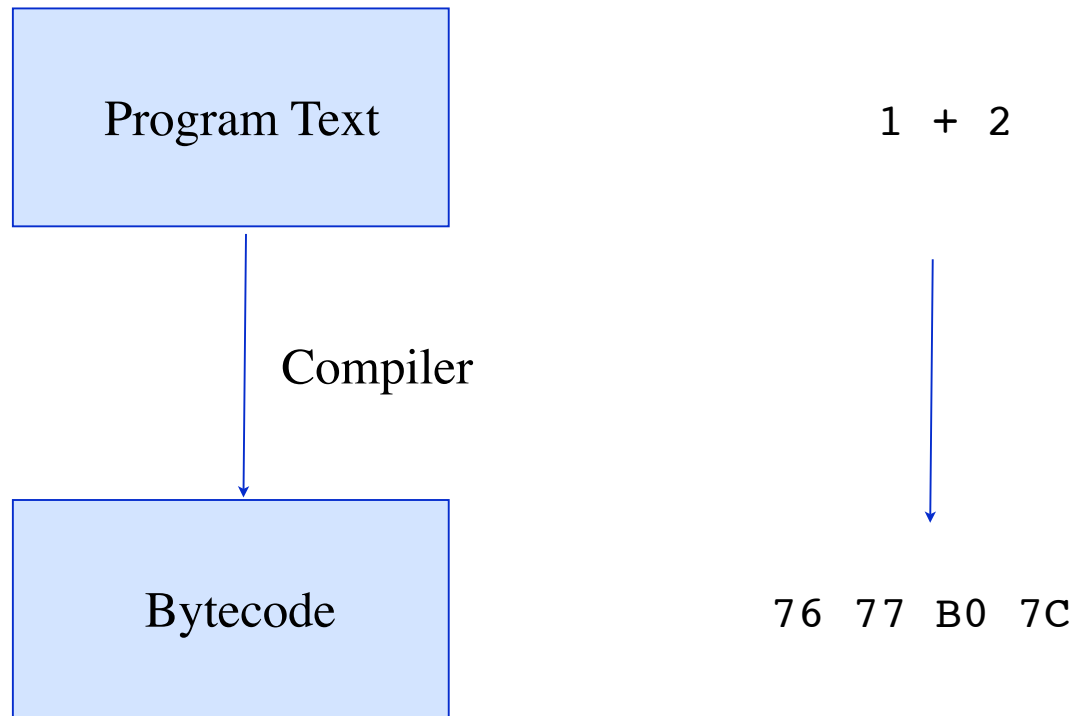


Bytecode

- Byte-encoded instruction set
- This means: 256 main instructions
- Stack based
- Positive:
 - Very compact
 - Easy to interpret
- Important: The ISA of a VM does not need to be Bytecode.



Compiling to Bytecode



Example: Number>>asInteger

- Smalltalk code:

```
^1 + 2
```

- Symbolic Bytecode

```
<76> pushConstant: 1  
<77> pushConstant: 2  
<B0> send: +  
<7C> returnTop
```



Example: Java Bytecode

- From class Rectangle

```
public int perimeter()
```

```
0: iconst_2
```

```
1: aload_0           "push this"
```

```
2: getfield#2        "value of sides"
```

```
5: iconst_0
```

```
6: iaload
```

```
7: aload_0           2*(sides[0]+sides[1])
```

```
8: getfield#2
```

```
11: iconst_1
```

```
12: iaload
```

```
13: iadd
```

```
14: imul
```

```
15: ireturn
```



Difference Java and Squeak

- Instruction for arithmetic
 - Just method calls in Smalltalk
- Typed: special bytecode for int, long, float
- Bytecode for array access

- Not shown:
 - Jumps for loops and control structures
 - Exceptions (java)
 -



Systems Using Bytecode

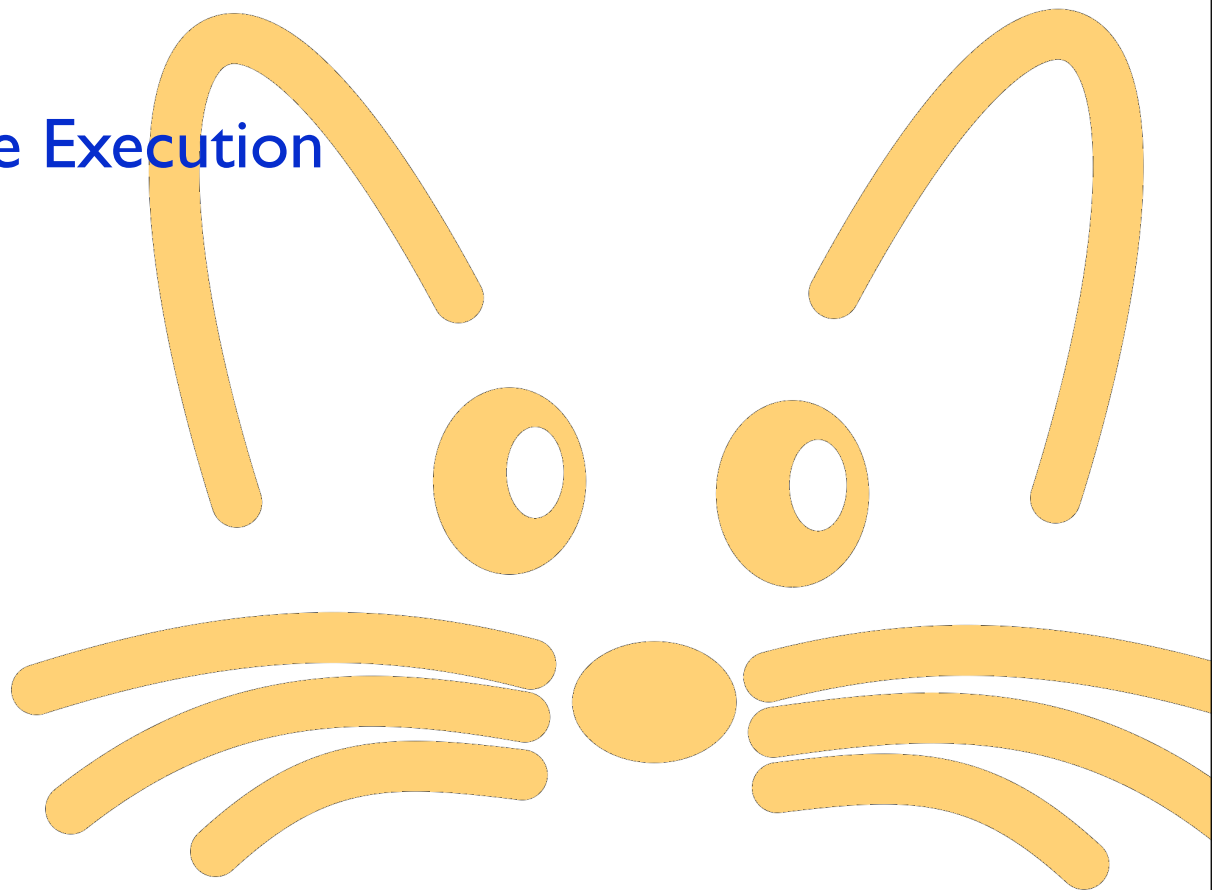
- USCD Pascal (first, ~1970)
- Smalltalk
- Java
- PHP
- Python

- No bytecode:
 - Ruby: Interprets the AST



Bytecode: Lessons Learned

- Instruction set of a VM
- Stack based
- Fairly simple
- Next: Bytecode Execution



Running Bytecode

- Invented for Interpretation
- But there are faster ways to do it
- We will see
 - Interpreter
 - Threaded code
 - Translation to machine code (JIT, Hotspot)



Interpreter

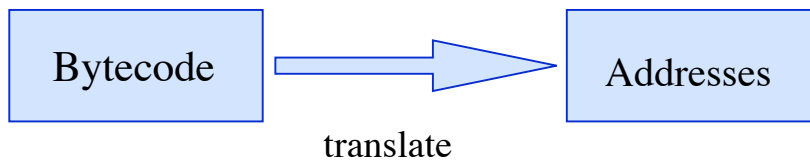
- Just a big loop with a case statement
- Positive:
 - Memory efficient
 - Very simple
 - Easy to port to new machines
- Negative:
 - Slooooooow...

```
while (1) {  
    bc = *ip++;  
    switch (bc) {  
        ...  
        case 0x76:  
            *++sp = ConstOne;  
            break;  
        ...  
    }  
}
```



Faster: Threaded Code

- Idea: Bytecode implementation in memory has Address



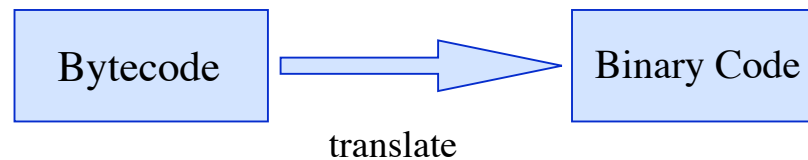
```
push1:
    *++sp = ConstOne
    goto *ip++;
push2:
    *++sp = ConstTwo
    goto *ip++;
```

- Pro:
 - Faster
 - Still fairly simple
 - Still portable
 - Used (and pioneered) in Forth.
 - Threaded code can be the ISA of the VM



Next Step: Translation

- Idea: Translate bytecode to machine code



- Pro:
 - Faster execution
 - Possible to do optimizations
 - Interesting tricks possible: Inline Caching (later)
- Negative
 - Complex
 - Not portable across processors
 - Memory



Simple JIT Compiler

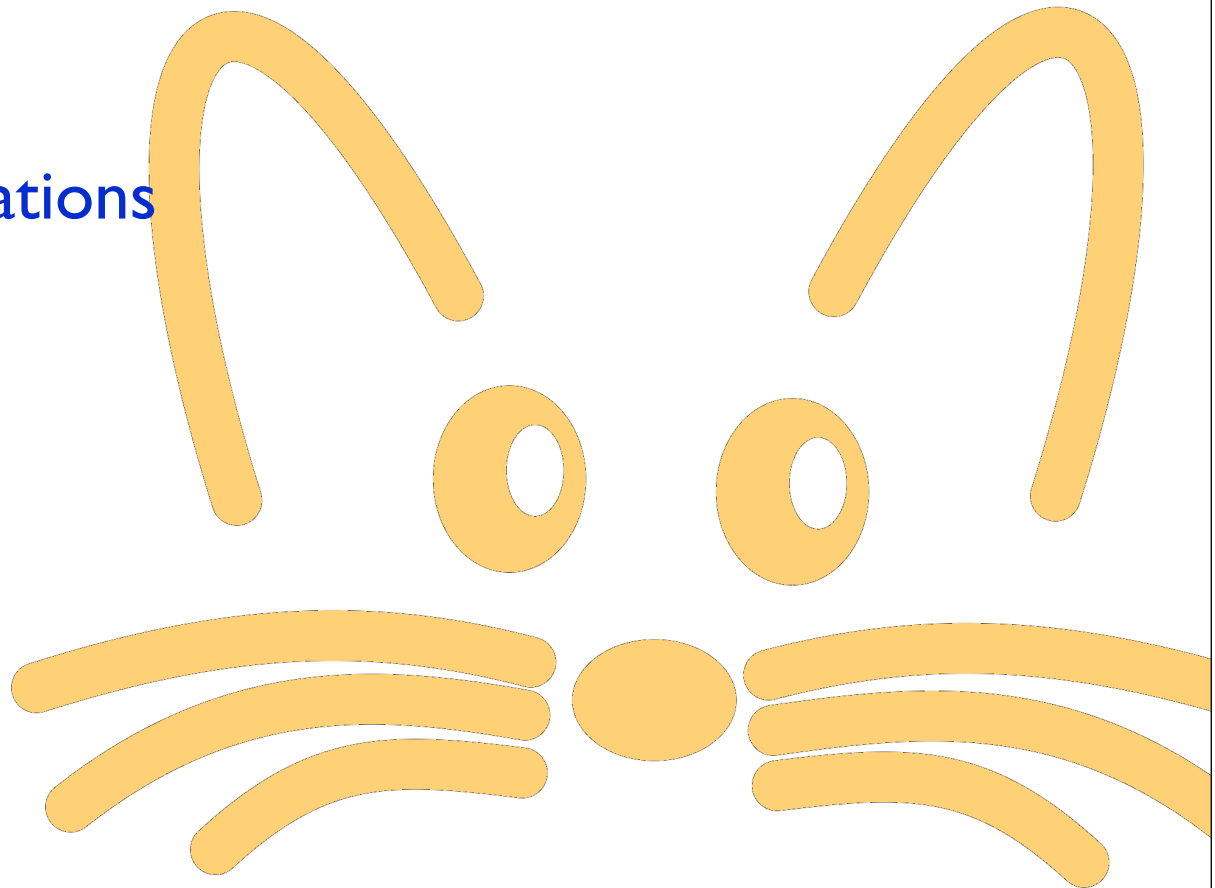
- JIT = “Just in Time”
- On first execution: Generate Code
- Need to be very fast
 - No time for many code optimizations
- Code is cached (LRU)

- Memory:
 - Cache + implementation JIT in RAM.
 - We trade memory for performance.



Bytecode Execution: Lessons Learned

- We have seen
 - Interpreter
 - Threaded Code
 - Simple JIT
- Next: Optimizations



Optimizing Method Lookup

- What is slow in OO virtual machines?
- Overhead of Bytecode Interpretation
 - Solved with JIT
- Method Lookup
 - Java: Polymorph
 - Smalltalk: Polymorph and dynamically typed
 - Method to call can only be looked up at runtime



Example Method Lookup

- A simple example:

```
array := #(0 1 2 2.5).
```

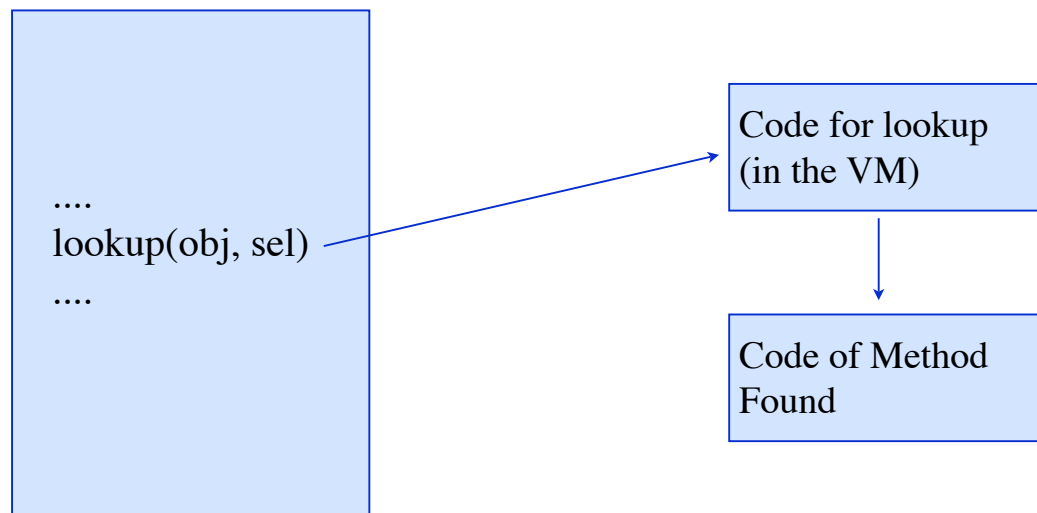
```
array collect: [:each | each + 1]
```

- The method “+” executed depends on the receiving object



Method Lookup

- Need to look this up at runtime:
 - Get class of the receiver
 - if method not found, look in superclass



Binary Code
(generated by JIT)



Observation: Yesterday's Weather

- Predict the weather of tomorrow: Same as today
- Is right in over 80%
- Similar for method lookup:
 - Look up method on first execution of a send
 - The next lookup would likely get the same method
- True polymorphic sends are seldom



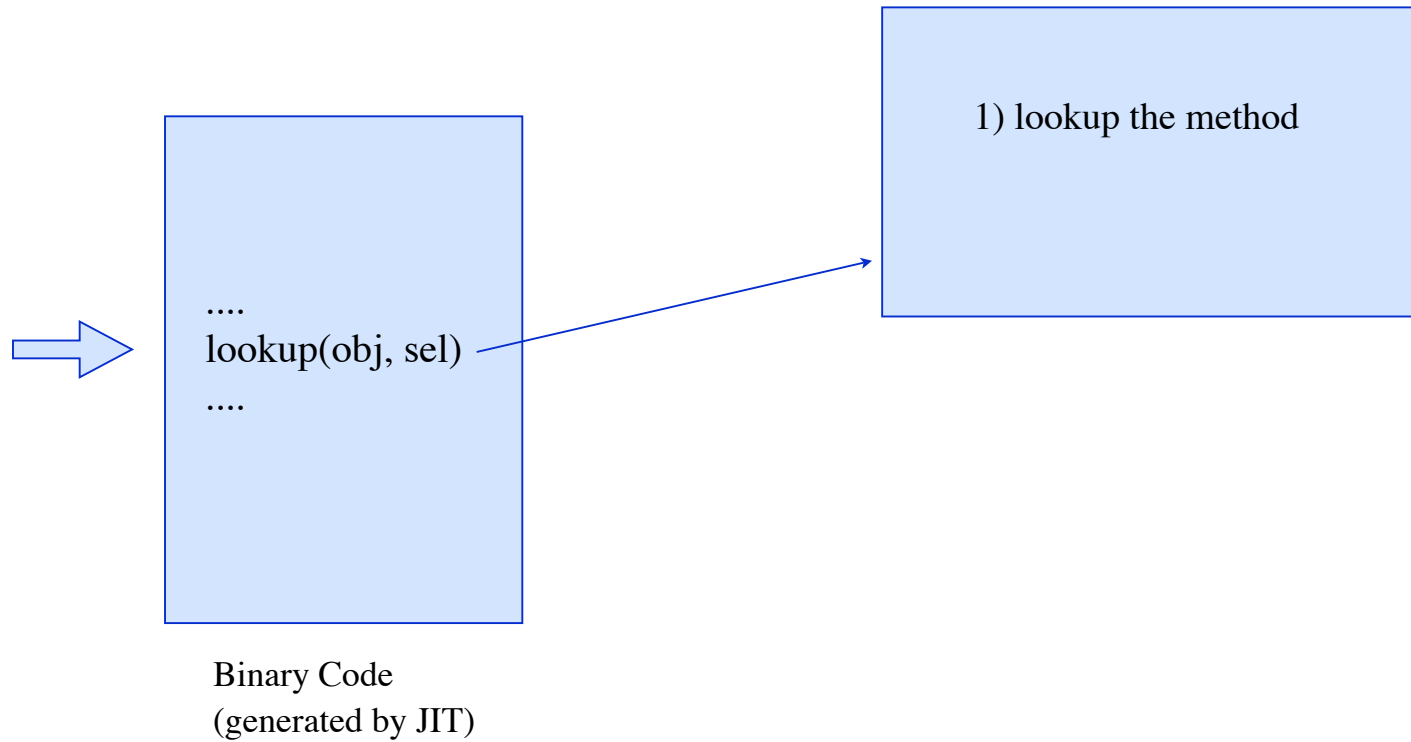
Inline Cache

- Goal: Make sends faster in many cases
- Solution: remember the last lookup
- Trick:
 - Inline cache (IC)
 - In the binary code of the sender



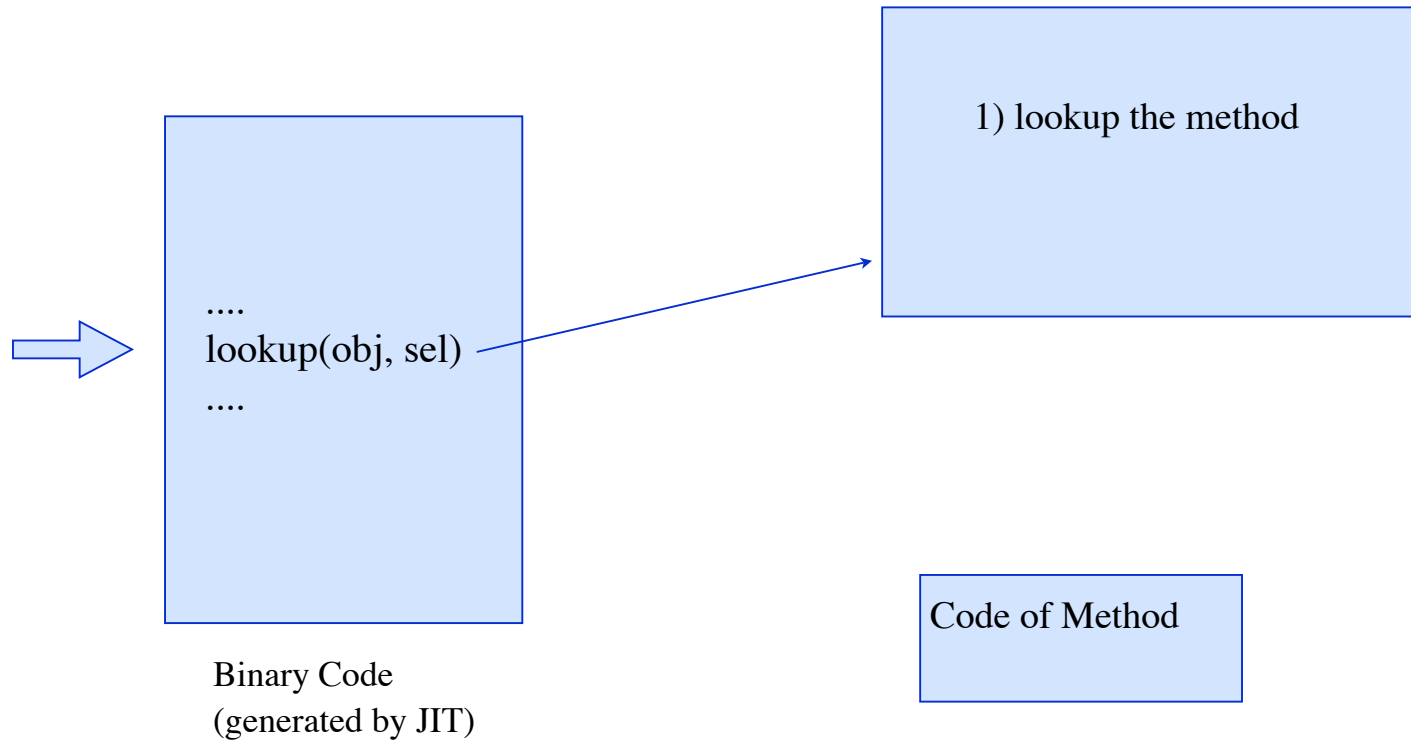
Inline Cache: First Execution

First Call:



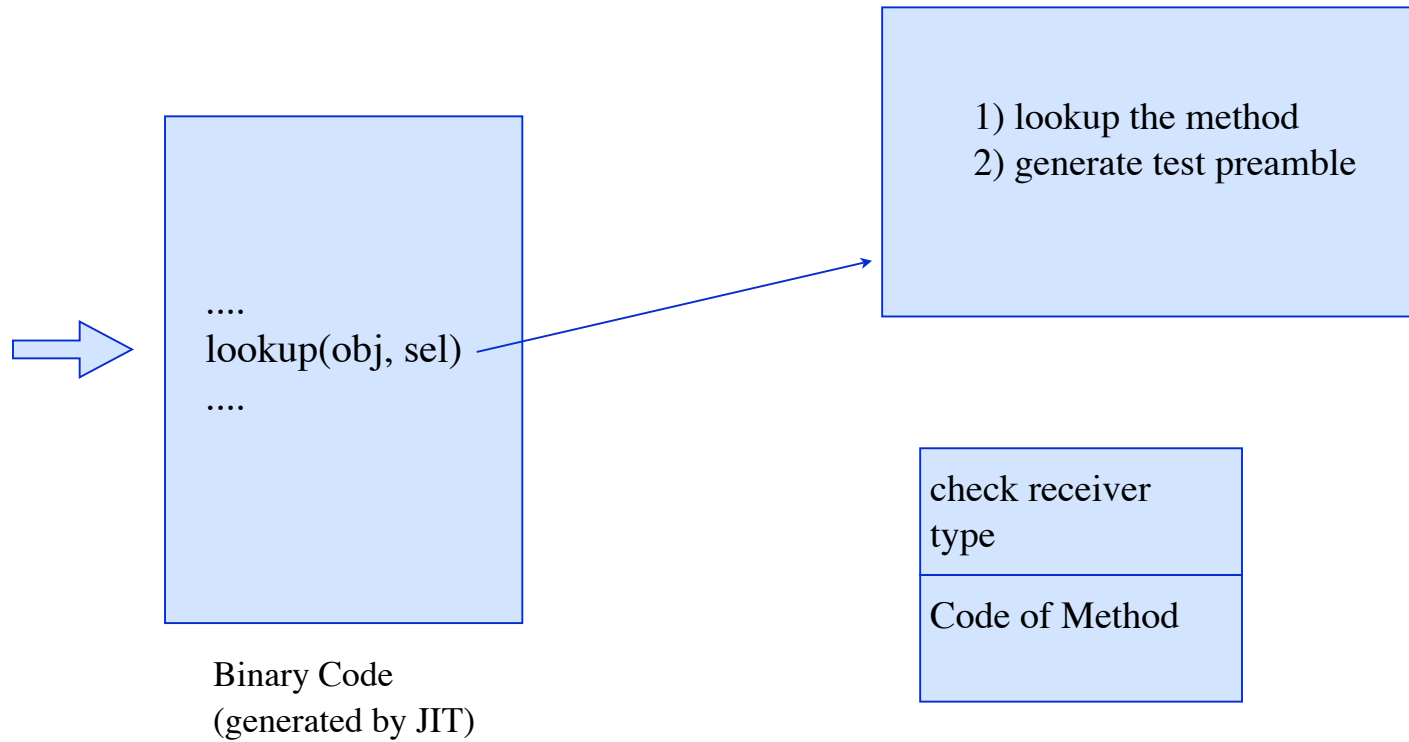
Inline Cache: First Execution

First Call:



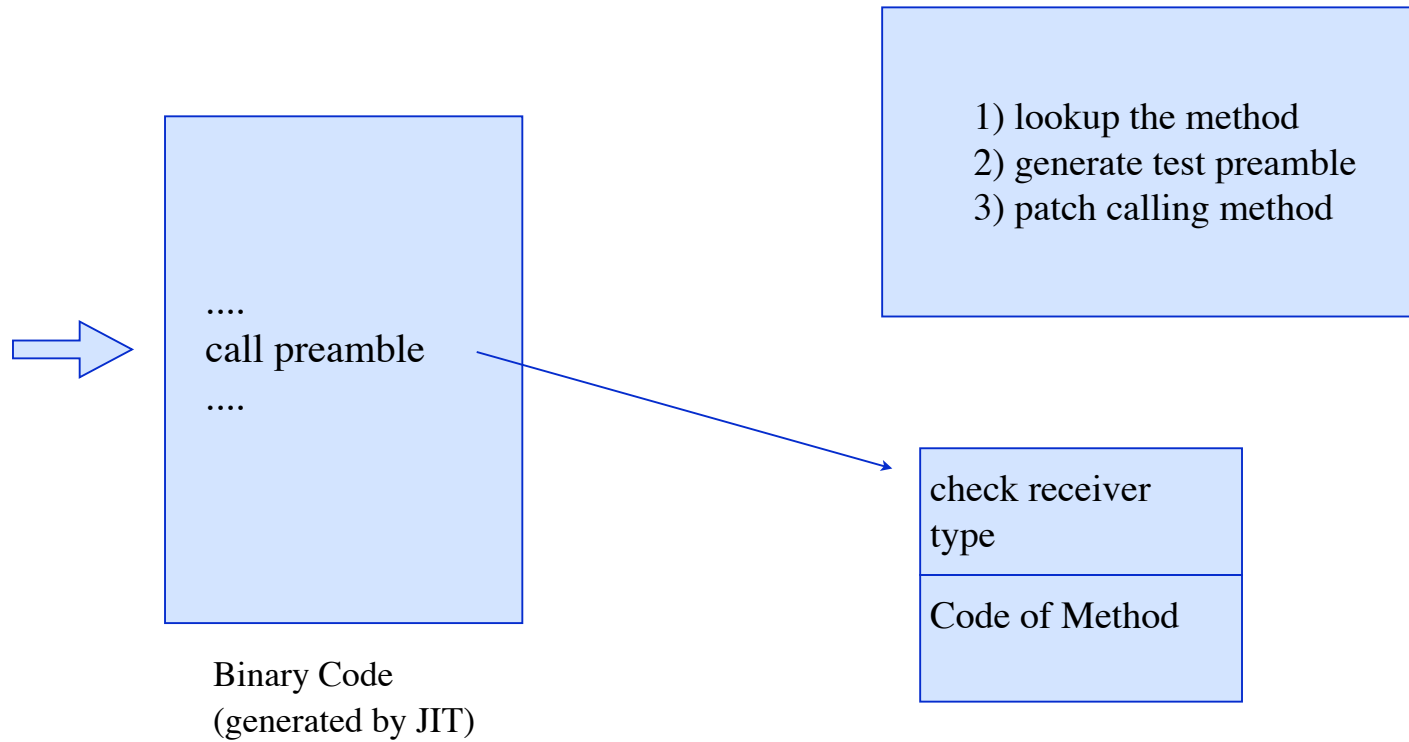
Inline Cache: First Execution

First Call:



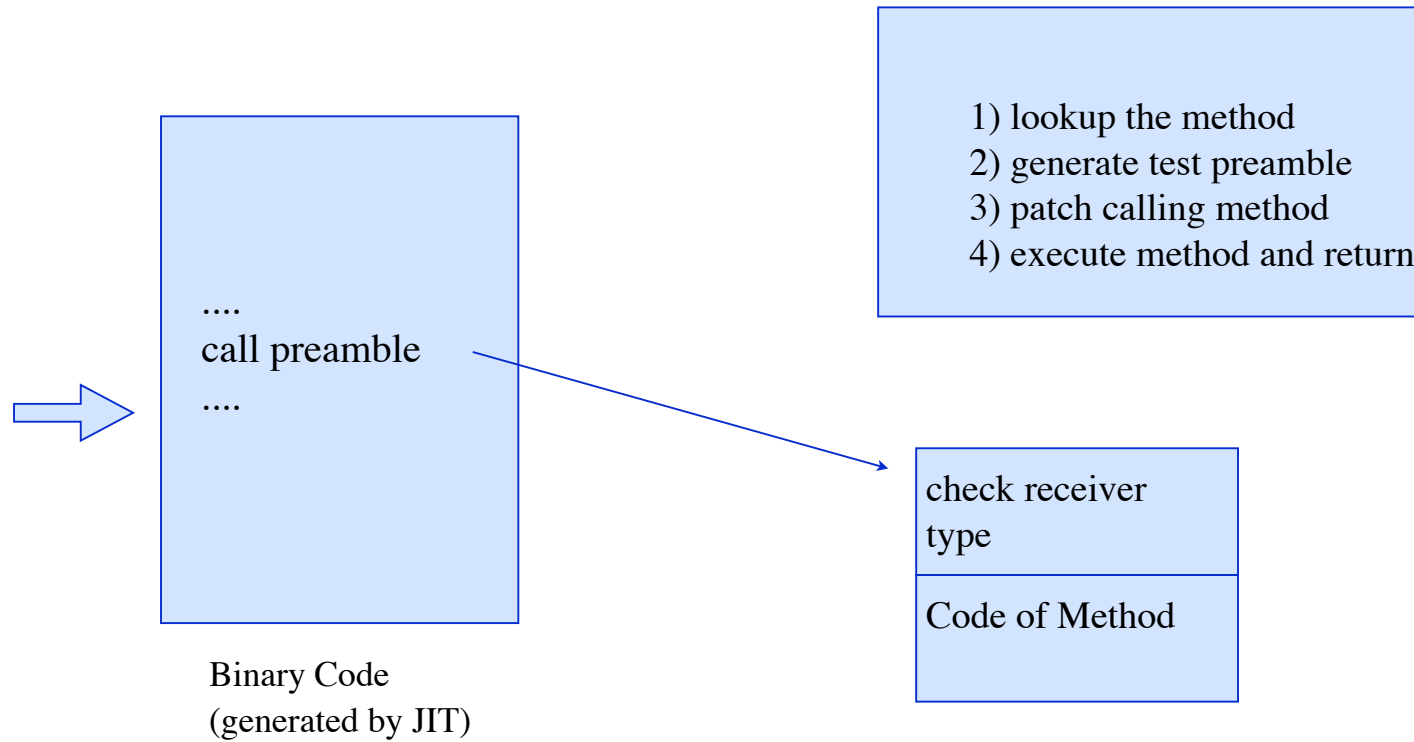
Inline Cache: First Execution

First Call:



Inline Cache: First Execution

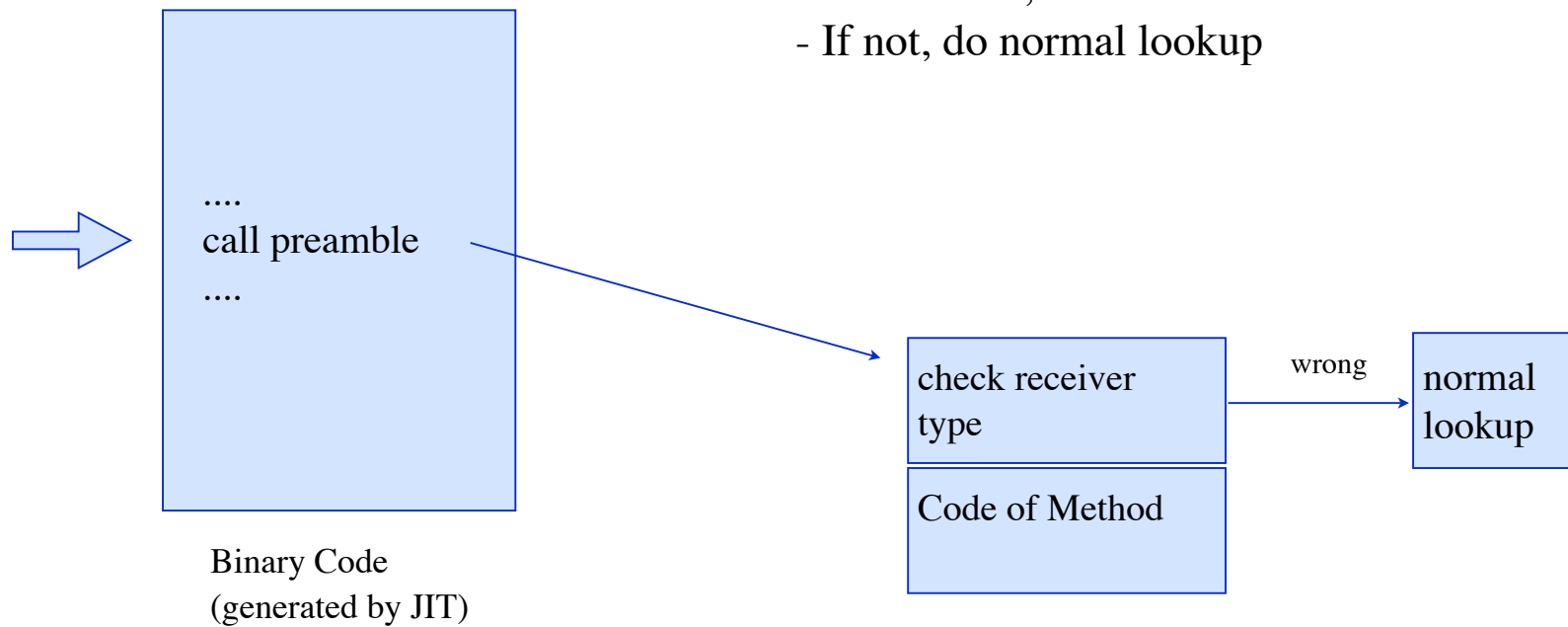
First Call:



Inline Cache: Second Execution

Second Call:

- we jump directly to the test
- If test is ok, execute the method
- If not, do normal lookup



Limitation of Simple Inline Caches

- Works nicely at places where only one method is called. “Monomorphic sends”. >80%
- How to solve it for the rest?
- Polymorphic sends (<15%)
 - <10 different classes
- Megamorphic sends (<5%)
 - >10 different classes



Example Polymorphic Send

- This example is Polymorphic.

```
array := #(1 1.5 2 2.5 3 3.5).
```

```
array collect: [:each | each + 1]
```

- Two classes: Integer and Float
- Inline cache will fail at every send
- It will be slower than doing just a lookup!

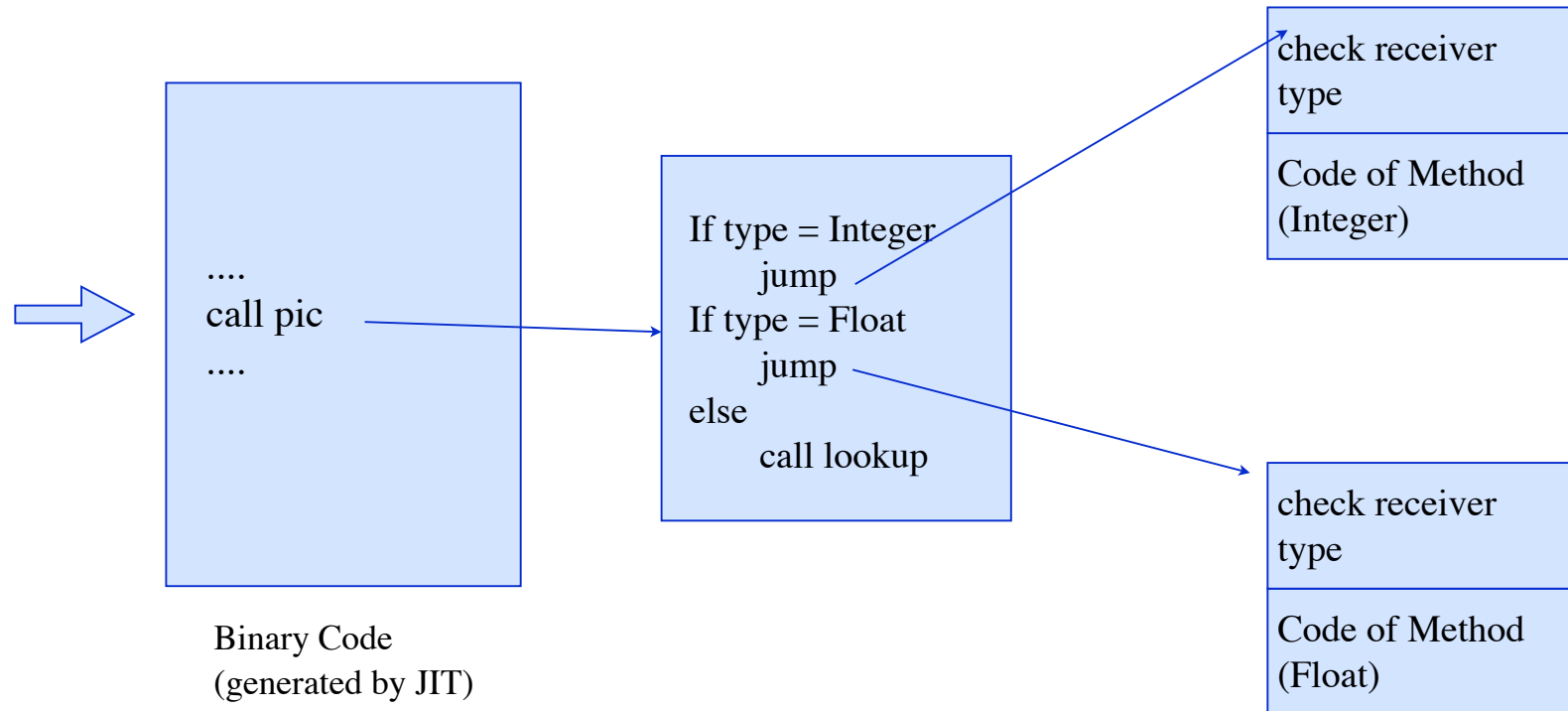


Polymorphic Inline Caches

- Solution: When inline cache fails, build up a PIC
- Basic idea:
 - For each receiver, remember the method found
 - Generate stub to call the correct method



PIC



PIC

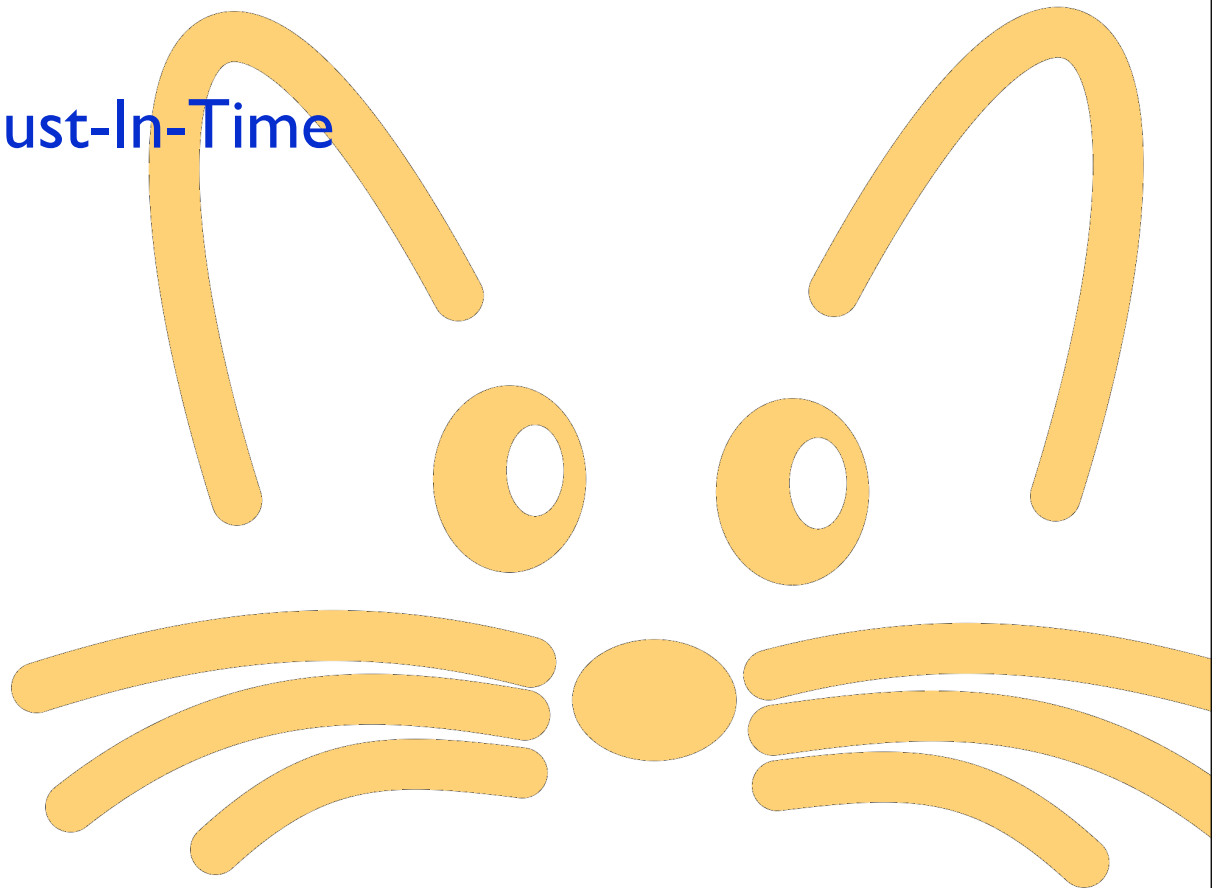
- PICs solve the problem of Polymorphic sends
- Three steps:
 - Normal Lookup / generate IC
 - IC lookup, if fail: build PIC
 - PIC lookup
- PICs grow to a fixed size (~10)
- After that: replace entries

- Megamorphic sends:
 - Will be slow
 - Some systems detect them and disable caching



Optimizations: Lessons Learned

- We have seen
 - Inline Caches
 - Polymorphic Inline Caches
- Next: Beyond Just-In-Time



Beyond JIT: Dynamic Optimization

- Just-In-Time == Not-Enough-Time
 - No complex optimization possible
 - No whole-program-optimization
- We want to do real optimizations!



Excursion: Optimizations

- Or: Why is a modern compiler so slow?
- There is a lot to do to generate good code!
 - Transformation in a good intermediate form (SSA)
 - Many different optimization passes
 - Constant subexpression elimination (CSE)
 - Dead code elimination
 - Inlining
 - Register allocation
 - Instruction selection



State of the Art: HotSpot et. al.

- Pioneered in Self
- Use multiple compilers
 - Fast but bad code
 - Slow but good code
- Only invest time were it really pays off
- Here we can invest some more
- Problem: Very complicated, huge warmup, lots of Memory
- Examples: Self, Java Hotspot, Jikes



PIC Data for Inlining

- Use type information from PIC for specialisation
- Example: Class Point

```
...  
lookup(rec, sel)  
...
```

Binary Code
(generated by JIT)

```
CartesianPoint>>x  
  ^x
```

```
PolarPoint>>x  
  "compute x from rho and theta"
```



Example Inlining

- The PIC will be build

```
...  
call pic  
...
```

Binary Code
(generated by JIT)

```
If type = CarthesianPoint  
    jump  
If type = PolarPoint  
    jump  
else  
    call lookup
```

The PIC contains
type Information!



Example Inlining

- We can inline code for all known cases

```
...  
if type = cartesian point  
  result ← receiver.x  
else if type = polar point  
  result ← receiver.rho * cos(receiver.theta)  
else call lookup  
...
```

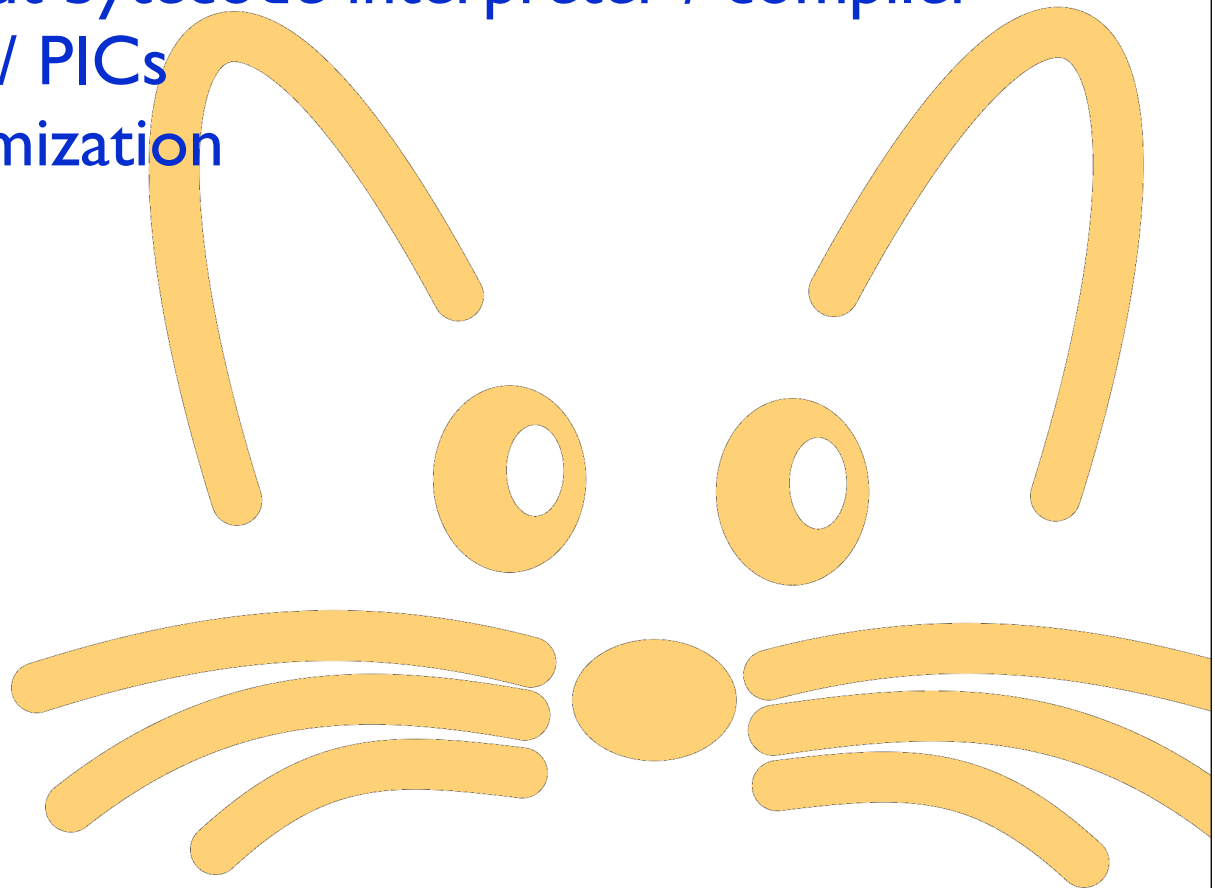
Binary Code
(generated by JIT)



End

- What is a VM
- Overview about bytecode interpreter / compiler
- Inline Caching / PICs
- Dynamic Optimization

- Questions?



Literature

- Smith/Nair: Virtual Machines (Morgan Kaufman August 2005). Looks quite good!
- For PICs:
 - Urs Hölzle, Craig Chambers, David Ungar: Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches
 - Urs Hoelzle. "Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming." Ph.D. thesis, Computer Science Department, Stanford University, August 1994.

