CS614: Advanced Compilers

Just-In-Time Compilation

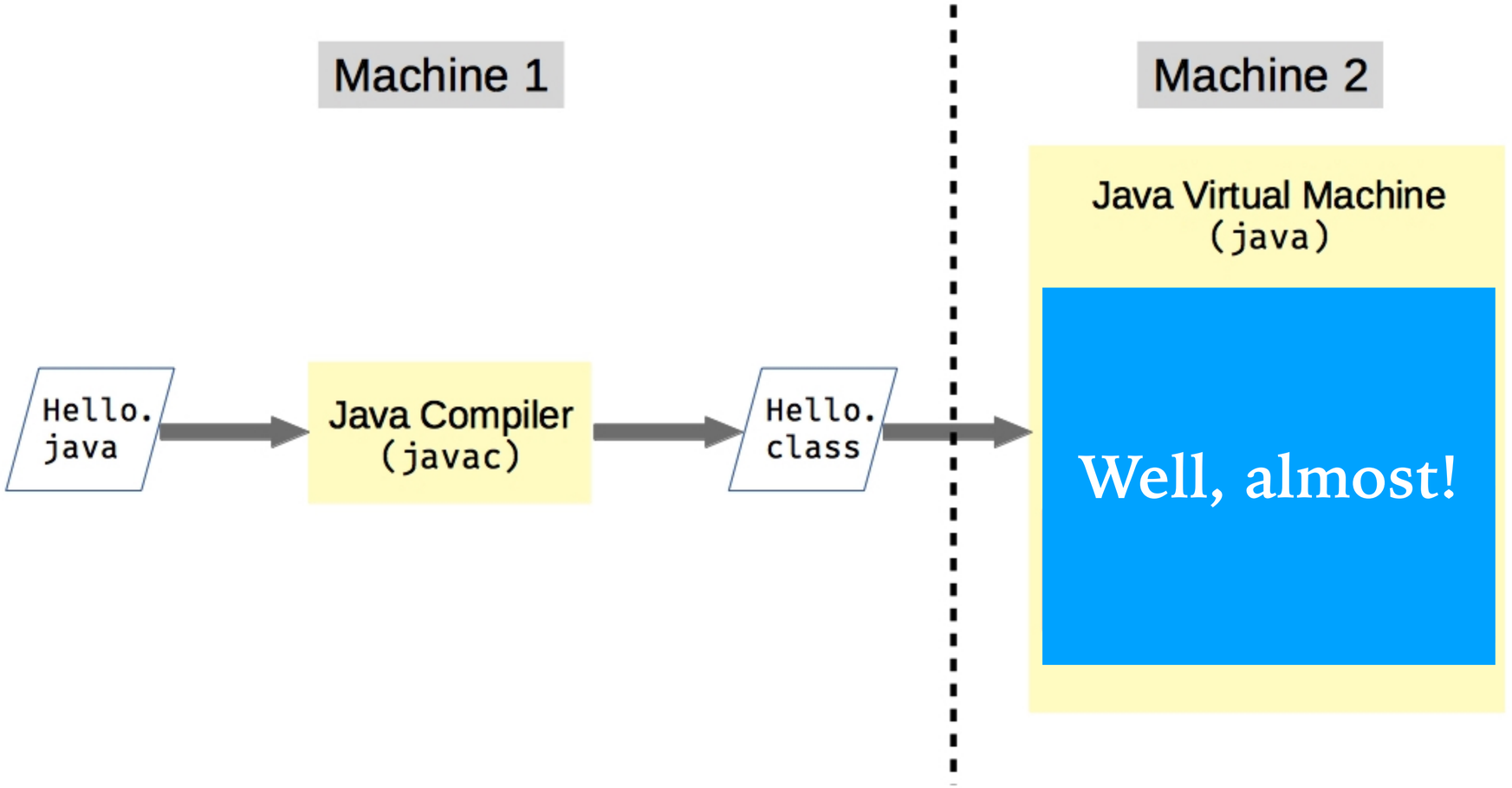
Manas Thakur
CSE, IIT Bombay

Spring 2025

So we have learnt advanced material about compilers

- IR generation
- Dataflow analysis
- SSA-based optimizations
- Method inlining
- Register allocation
- Instruction scheduling
- Cache optimizations
- Parallelization
- Pointer analysis
- *And we learnt all of them additionally for OO languages, with programming assignments in **Java**.*

We also know Java's translation mechanism



And yet `javac` doesn't perform (m)any optimizations! **Why?**

Dynamism prohibits useful static optimization [1]

- Dynamic classloading — Could it be *useful* for a JIT analysis?
  - Ability to load classes dynamically, on demand.

```
class A {
   void foo() {
      return 10;
   }
}
class Test {
   int bar(A x) {
      int y = 10 + x.foo();
      return y;
   }
   void m() {
      if (this.bar() == 0)
         launch();
   }
}
```

Not present statically:

```
class B extends A {
   void foo() {
      return -10;
   } }
```

If class `B` gets loaded dynamically, statically inlining `foo` into `bar` (and possibly that `bar` into `m`) may go **wrong**.

Dynamism prohibits useful static optimization [2]

- Reflection
  - Ability to read class/method names and create instances, based on runtime values.

```
interface Game { void play(); }
class Cricket implements Game {
   void play() {
      ...
   }
}
class Soccer implements Game {
   void play() {
      ...
   }
}
```

The static compiler may know very little (and sometimes nothing) about the call graph rooted at the call site `mtd.invoke(obj);`.

```
class M {
   void m(String c) throws Exception {
      Class<?> cls = Class.forName(c);
      Method mtd = cls.getDeclaredMethod("play", null);
      Object obj = cls.newInstance();
      mtd.invoke(obj); // obj is the receiver
   } }
```

Dynamism prohibits useful static optimization [3]

- Hot-code replacement
  - Ability to modify code (usually in a debugging session), without suspending and/or restarting program execution.

```
class A {
   void foo() {
      return 10;
   } }
class Test {
   int bar(A x) {
      int y = 10 + x.foo();
      return y;
   }
   void m() {
      if (this.bar() == 0)
         launch();
   } }
```

Change using HCR:

```
class A {
   void foo() {
      return -10;
   } }
```

If the user replaces method `foo`'s body, statically inlining `foo` into `bar` (and possibly that `bar` into `m`) may go wrong.

Are we doomed to Bytecode interpretation?

- Let's compare performances:
  - Python vs Java Bytecode Interpreter
  - Python vs Java
  - C/C++ vs Java

- Without "`-Xint`", Java Bytecodes are not doomed for interpretation; they are also compiled to native code, just-in-time!

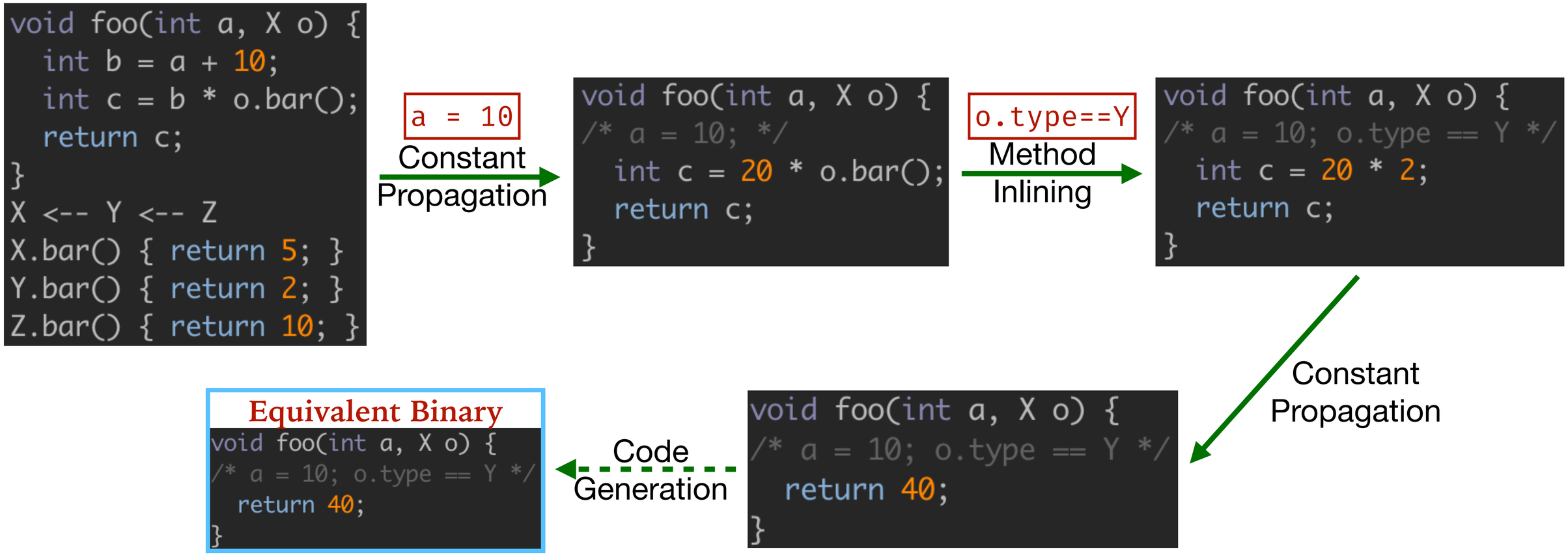
What is basic compilation anyway?

- *Specialization of an interpreter with the program to be interpreted.*
- **Explanation:**
  - An interpreter `I` written in language `T` for a language `S` takes a program `P` in `S` along with `P`'s input and gives `P`'s output as its output.
  - A compiled version of `P` in target language `T` takes `P`'s input and generates `P`'s output.
  - If we evaluate a program `P` with *part* of its input, we get a specialized version of `P` that can take the remaining input and generate `P`'s output (e.g. `pow(x,2)` is like `sqr(x)`).
  - What if we specialize `I` (written in T) with `P` (written in S)?
    - We would get a specialized program in language `T` that can take `P`'s input and generate `P`'s output!! **Q.E.D.**

Just-In-Time Compilation

- Execution-time *specialization* of the program with respect to the execution instance
  - Has access to run-time profile
  - Can specialize the program to specific inputs
- Generates native code that can execute directly on the machine
  - Much faster than interpretation
- Speeds up performance for languages that statically cannot be optimized much
  - Java, Scala, Clojure, Kotlin, Groovy, … HotSpot, OpenJ9, GraalVM, TornadoVM
  - C# family: .NET framework
  - Python (PyPy); R (Ř), JavaScript (V8), Lua (LuaJIT), …

Speculate, Optimize, Generate



What to do when assumptions go wrong?

- A binary compiled under assumptions that go wrong can no longer be used.
- What is the safest code to execute in the next execution?
  - Interpretation.

```
void foo(int a, X o) {
  int b = a + 10;
  int c = b * o.bar();
  return c;
}
X <-- Y <-- Z
X.bar() { return 5; }
Y.bar() { return 2; }
Z.bar() { return 10; }
```

Equivalent Binary

```
void foo(int a, X o) {
/* a = 10; o.type == Y */
  return 40;
}
```

Deoptimize

- The process of switching down from optimized code to unoptimized code (with respect to the violated assumption) is called **deoptimization**.
  - It's often a wasted effort; happens fewer times the better.
- Should we throw away the binary immediately?
- Wait for some time to see if the assumption starts holding again?
- Reprofile and find newer assumptions if the code is still hot?
- Maintain multiple versions (subject to the dispatch cost and runtime memory)?

JIT seems expensive; should we do it for the whole program?

Binary B1

```
void foo(int a, X o) {
  int b = a + 10;
  int c = b * o.bar();
  return c;
}
X <-- Y <-- Z
X.bar() { return 5; }
Y.bar() { return 2; }
Z.bar() { return 10; }
```

Binary B2

```
void foo(int a, X o) {
/* a = 10; */
  int c = 20 * o.bar();
  return c;
}
```

Binary B3

```
void foo(int a, X o) {
/* a = 10; o.type == Y */
  return 40;
}
```