Typed compilation of objects

Christopher League

6 November 2003
Stevens Institute of Technology

LONG ISLAND UNIVERSITY

Why typed compilation?

- We must ensure safe, efficient execution of untrusted code
  - Digital signature confirms identity, not safety
  - Reference monitor too expensive for fine-grained properties

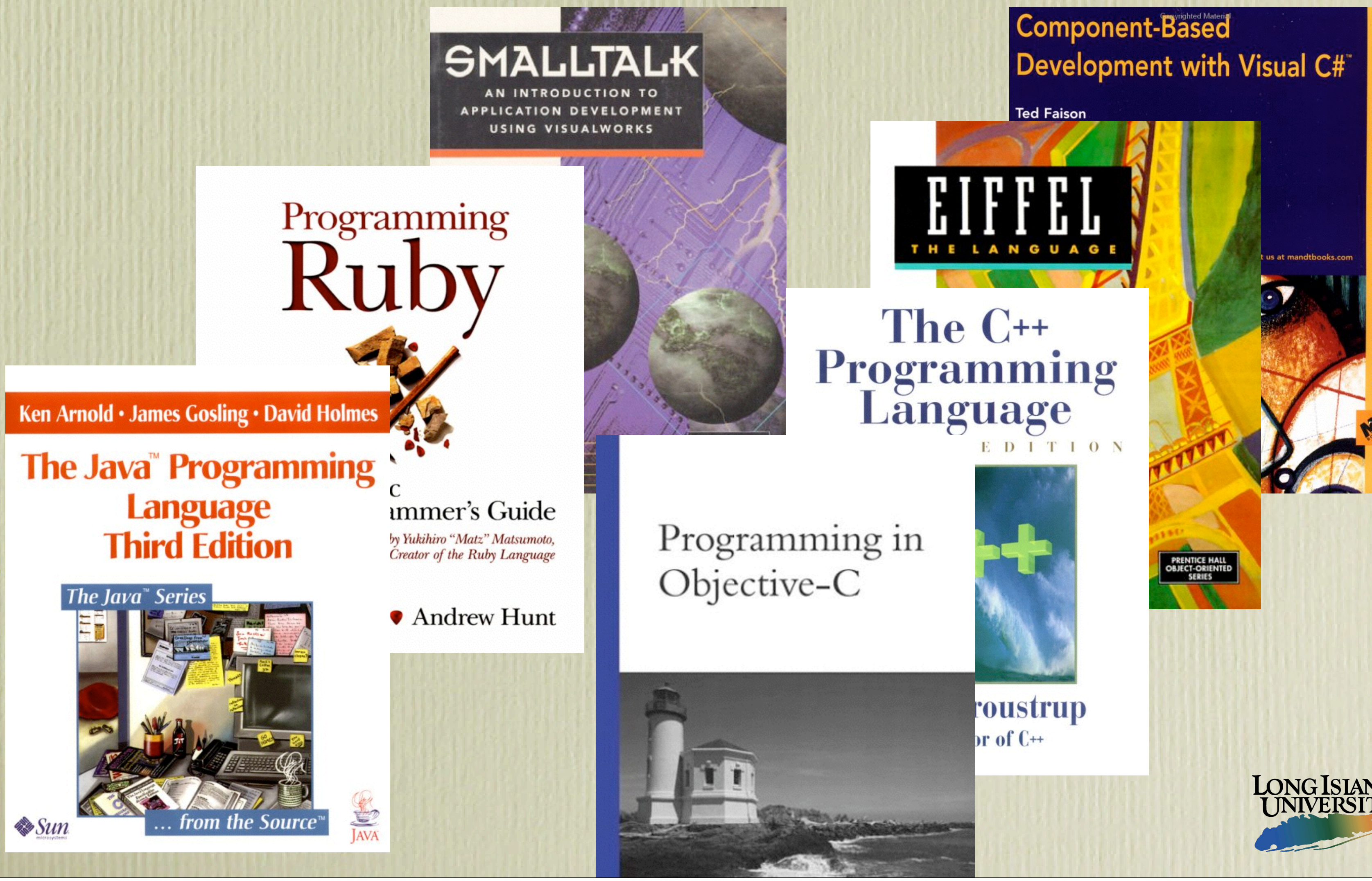
LONG ISLAND UNIVERSITY

How does it work?

- Develop sound & decidable type system for intermediate and object languages
- Transform source-level type information
- Emit object code + typing derivation

Why objects?

- Cannot deny that OO technology remains popular...



Why objects?

- Cannot deny that OO technology remains popular...
- Thus, for certifying compilation to be viable, we must support OO!

What is an object?

- C++ compiler hacker:

> "An object is just a struct with a pointer to a struct containing function pointers."

- True, but that fails to capture the subtle invariants that make it work...

What is an object?

- Functional programming advocate:

> "An object is just a closure
> with multiple entry points."

- Maybe, but that fails to account for dynamic binding.

LONG ISLAND UNIVERSITY

Dynamic binding is essential

- Inheritance without polymorphism is possible, but certainly not very useful.
- One can declare derived types, but the actual operation being called is always known at compile time.

[Booch 1994]

Outline

- Object layout — efficient dynamic dispatch
- Object encoding — capturing the invariants
- Additional issues in compiling Java
- Compiling non-manifest base classes

Object layout

What is object layout?

- von Neumann architecture has no notion of methods, objects, classes, inheritance, or dynamic binding
- We must map these features onto load/store operations and sequential memory
- Procedural abstractions (records and functions) are “closer to the metal”

Learn from compiler hackers

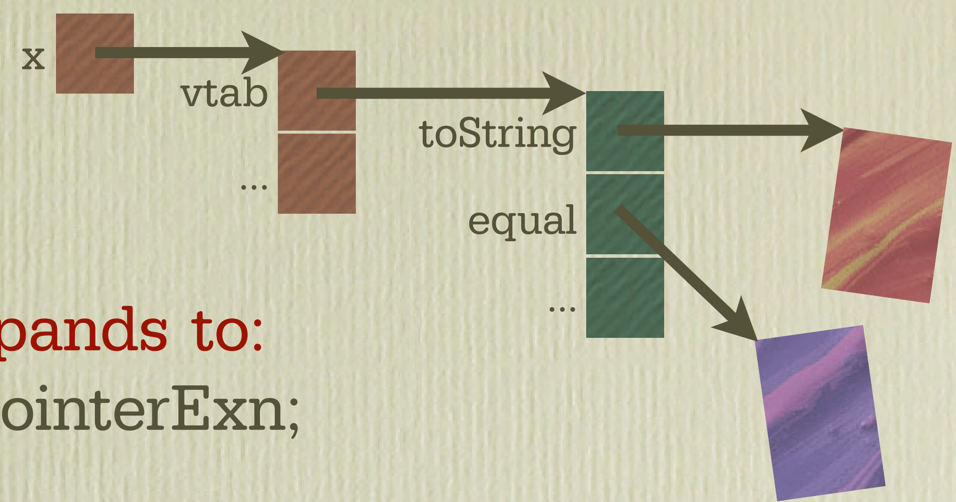
- Whether or not they know type theory, they certainly understand invariants
- The efficient layout used in C++ works for a reason
- Can we understand and capture that reason?

Implementing dynamic dispatch

```
Object x = ...;
x.toString();          // invokevirtual – a primitive
                       // of the Java Virtual Machine
```

Implementing dynamic dispatch

```
Object x = ...;
x.toString();
```



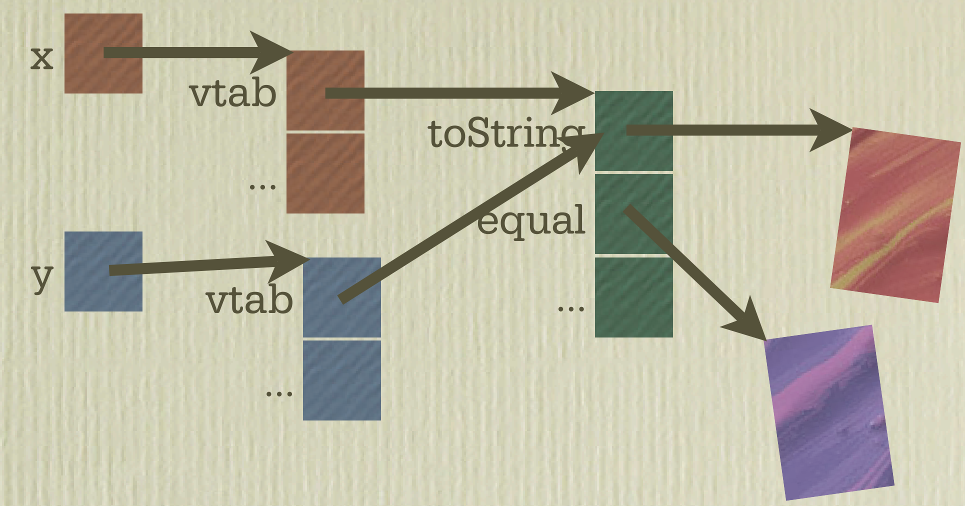

```
// virtual method call expands to:
if (x == null) throw NullPointerExn;
r1 = x.vtab;
r2 = r1.toString;
call r2 (x);
```

self argument

Multiple objects share vtable

Object x = ...;
x.toString();

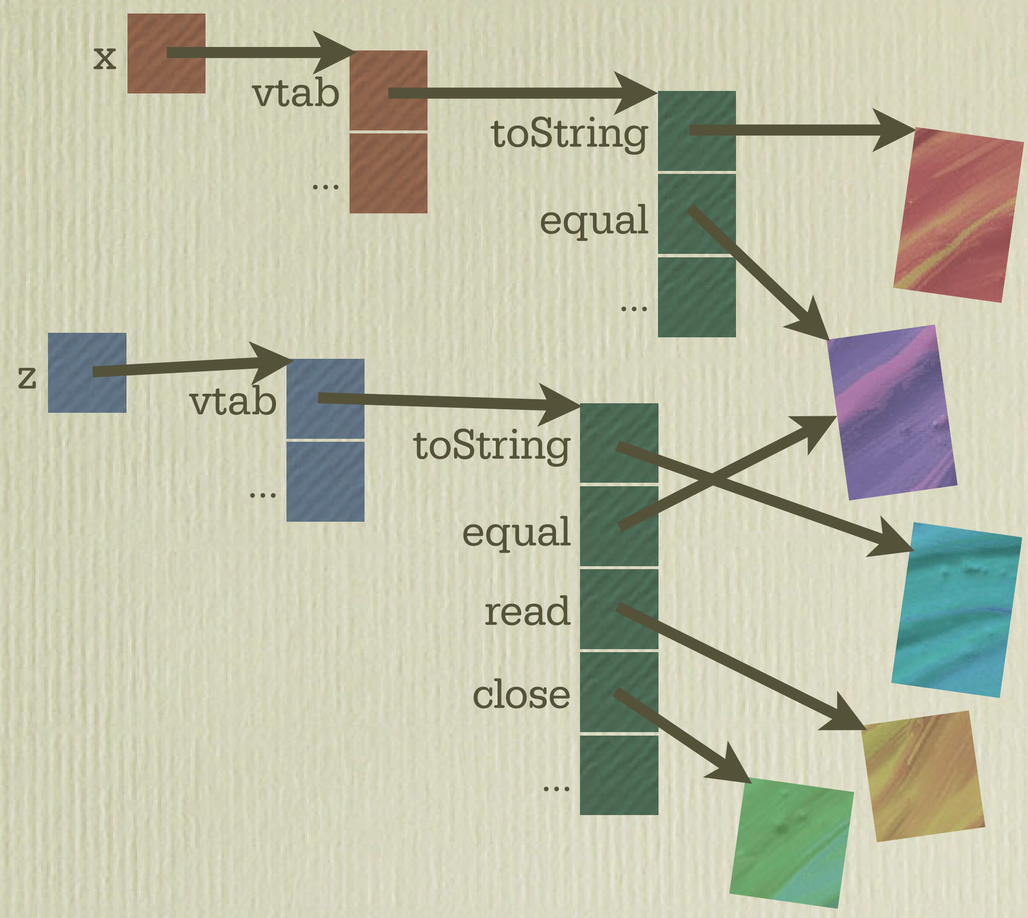
Object y = ...;



Subclasses share method code

```
Object x = ...;
x.toString();

InputStream z = ...;
```

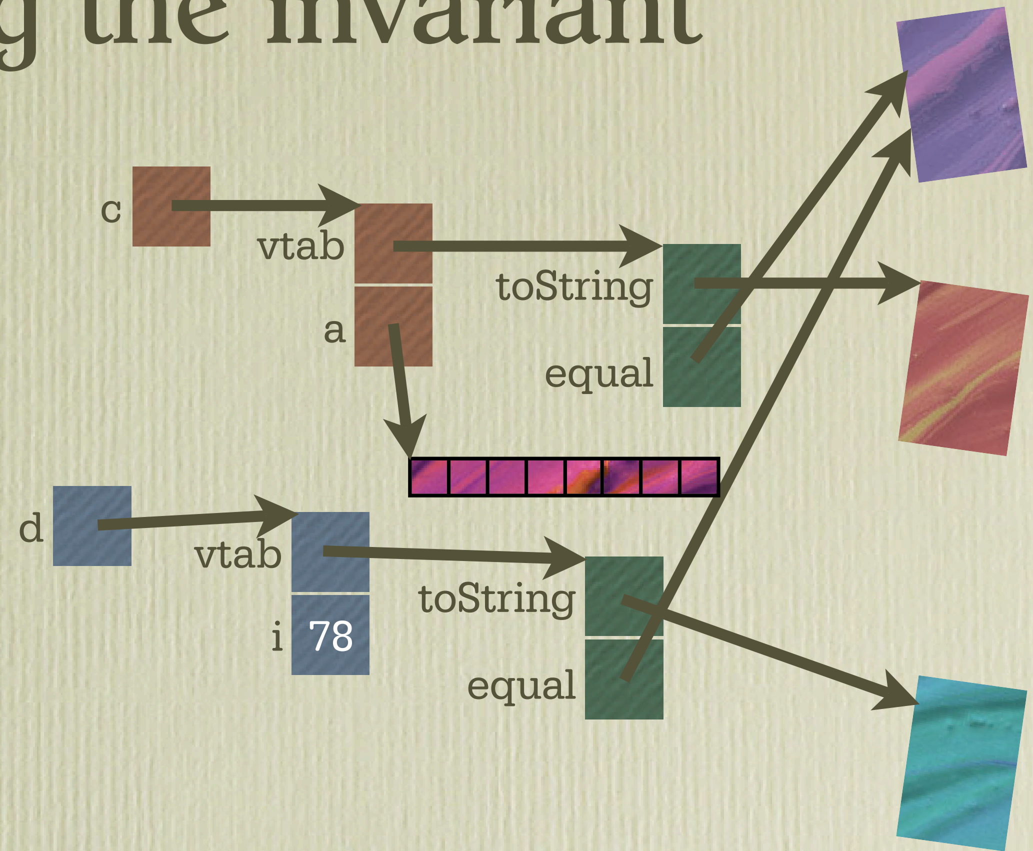


Breaking the invariant

Object c = new C();
c.toString();

Object d = new D();

// virtual method call
(null check)
r1 = c.vtab;
r2 = r1.toString;
call r2 (c);
call r2 (d);



What happened?

- The object passed as the self argument must be the same object from which the method was selected
  - (Actually, it must belong to same dynamic class)
- Goal:
  - Encode that invariant using a type system, but
  - Do not interfere with efficient layout

LONG ISLAND UNIVERSITY

Object encoding

Objects are tuples of functions

- With single inheritance, all offsets should be known at compile time
- Therefore we just need tuples with fixed offsets
  - No record extension or concatenation
  - No first-class labels

Long Island University

Tuples

$$\frac{\Delta; \Gamma \vdash e_i \colon \tau_i \qquad \forall i \in 1 \ldots n}{\Delta; \Gamma \vdash \langle e_1, \ldots, e_n \rangle \colon \langle \tau_1, \ldots, \tau_n \rangle}$$

$$\frac{\Delta; \Gamma \vdash e \colon \langle \tau_1, \ldots, \tau_n \rangle \qquad 1 \leq i \leq n}{\Delta; \Gamma \vdash e.i \colon \tau_i}$$

Functions

$$\frac{\Delta; \Gamma, x : \tau \vdash e : \tau'}{\Delta; \Gamma \vdash \lambda x{:}\tau.e : \tau \to \tau'}$$

$$\frac{\Delta; \Gamma \vdash e : \tau \to \tau' \qquad \Delta; \Gamma \vdash e' : \tau}{\Delta; \Gamma \vdash e\ e' : \tau'}$$

Typing self application

- Suppose x is an object, with an integer method in slot 1 of its vtable

$$\Delta; \Gamma \vdash (x.1.1)\ x : \mathrm{int}$$

- But what is the type of x ?

LONG ISLAND UNIVERSITY

Typing self application

$$\cfrac{\cfrac{\Delta;\Gamma \vdash x : \langle\langle\tau_x \rightarrow \mathrm{int}\rangle, \mathrm{int}\rangle}{\Delta;\Gamma \vdash x.1 : \langle\tau_x \rightarrow \mathrm{int}\rangle}}{\Delta;\Gamma \vdash x.1.1 : \tau_x \rightarrow \mathrm{int}} \quad \Delta;\Gamma \vdash x : \tau_x$$

$$\frac{\Delta;\Gamma \vdash x.1.1 : \tau_x \rightarrow \mathrm{int} \qquad \Delta;\Gamma \vdash x : \tau_x}{\Delta;\Gamma \vdash (x.1.1)\ x : \mathrm{int}}$$

Recursive definition?

$$\tau_x = \langle\langle\tau_x \to \mathrm{int}\rangle, \mathrm{int}\rangle$$

$$\begin{aligned}\tau_x &= \mu\alpha.\langle\langle\alpha \to \mathrm{int}\rangle, \mathrm{int}\rangle \\ &= \langle\langle(\mu\alpha.\langle\langle\alpha \to \mathrm{int}\rangle, \mathrm{int}\rangle) \to \mathrm{int}\rangle, \mathrm{int}\rangle\end{aligned}$$

A simple object type

$$\tau_x = \mu\alpha.\langle\langle\alpha \rightarrow \text{int}\rangle, \text{int}\rangle$$

- But what about subclasses?
- Subtyping doesn't help much,
  due to the recursive type
- Again, take inspiration from the programmer...

Each object has two types

- Programmers distinguish between the static and dynamic classes of an object

```
Object x;
if (rand()%2 == 0) { x = new Cat(); }
else { x = new Dog(); }
x.toString();
```

Each object has two types

- The static class is known at compile time.
  The dynamic class is unknown,
  but it is some subclass of the static class.
- There are several ways to model this idea directly

Embrace the unknown

- ...with an existential quantifier

$$\frac{\Delta, \alpha :: \kappa \vdash \tau :: \mathrm{Type} \qquad \Delta \vdash \tau' :: \kappa \qquad \Delta; \Gamma \vdash e : \tau[\alpha := \tau']}{\Delta; \Gamma \vdash \mathrm{hide}\ \alpha :: \kappa = \tau'\ \mathrm{in}\ e{:}\tau : \exists \alpha :: \kappa.\tau}$$

$$\frac{\Delta; \Gamma \vdash e : \exists \alpha :: \kappa.\tau \qquad \Delta \vdash \tau' :: \mathrm{Type} \qquad \Delta, \alpha :: \kappa; \Gamma, x : \tau \vdash e' : \tau'}{\Delta; \Gamma \vdash \mathrm{open}\ e\ \mathrm{as}\ \alpha :: \kappa, x : \tau\ \mathrm{in}\ e' : \tau'}$$

Quantify over tuple tail

- Each object may have additional fields and methods beyond what is known at compile time.

$$\frac{\Delta \vdash \tau :: \mathrm{Type} \qquad \Delta \vdash \tau' :: \mathrm{R}^{i+1}}{\Delta \vdash \tau;\tau' :: \mathrm{R}^{i}}$$

$$\frac{}{\Delta \vdash \mathrm{End}^{i} :: \mathrm{R}^{i}} \qquad \frac{\Delta \vdash \tau :: \mathrm{R}^{0}}{\Delta \vdash \langle \tau \rangle :: \mathrm{Type}}$$

Efficient object encodings

$$\exists\alpha.\alpha \wedge (\mathrm{I}\ \alpha)$$

$$\exists\alpha \leq (\mathrm{I}\ \alpha).\alpha$$

$$\exists\delta :: \mathrm{Type} \to \mathrm{R}^1.\mu\alpha.(\mathrm{I}'\ \delta\ \alpha)$$

$$\mathrm{I} = \lambda\alpha.\langle\alpha \to \mathrm{int}\rangle$$
$$\mathrm{I}' = \lambda\delta :: \mathrm{Type} \to \mathrm{R}^1.\lambda\alpha.\langle\alpha \to \mathrm{int}; \delta\ \alpha\rangle$$

Non-manifest base classes

Limitations

- Preceding ideas work well for most of Java & C#
  - Single inheritance
  - Method offsets known at compile time
- Some languages are more flexible
  - Moby — base class specified at link time
  - Loom — first-class classes
  - Mixins

Non-manifest base classes

- The common substrate of many advanced OO features
- When compiling a class C, relatively little is known about its super class
- How do we determine C's object layout?
- Method calls are more expensive; how to optimize them?

'Links'

- Fisher, Reppy, and Riecke [ESOP 2000] developed an **untyped** IL to handle non-manifest base classes
  - Method suites are still tuples
  - Dictionaries map method labels to their offsets
  - Offsets may be computed and stored at compile time, link time, or run time.
  - A type system for 'links' seems **very difficult**

LONG ISLAND UNIVERSITY

Type-safe ‘certified binaries’

- Shao, et al. [POPL 2002] showed how to use calculus of constructions as a very sophisticated type language for any computation language
- Example: reason about array indices in the type language, and safely lift & remove bounds checks in the computation language
- Should work for reasoning about offsets in Links

LONG ISLAND UNIVERSITY