Typed compilation of objects

Christopher League

6 November 2003
Stevens Institute of Technology
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
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.
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

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

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

self argument
Multiple objects share vtable

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

Object y = ...

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

y = ...;
```
Subclasses share method code

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

InputStream \( z = \ldots; \)
Breaking the invariant

```java
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
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
Tuples

\[ \Delta; \Gamma \vdash e_i : \tau_i \quad \forall i \in 1 \ldots n \]
\[ \therefore \Delta; \Gamma \vdash \langle e_1, \ldots, e_n \rangle : \langle \tau_1, \ldots, \tau_n \rangle \]

\[ \Delta; \Gamma \vdash e : \langle \tau_1, \ldots, \tau_n \rangle \quad 1 \leq i \leq n \]
\[ \therefore \Delta; \Gamma \vdash e.i : \tau_i \]
Functions

\[ \Delta; \Gamma, x : \tau \vdash e : \tau' \]

\[ \Delta; \Gamma \vdash \lambda x : \tau . e : \tau \rightarrow \tau' \]

\[ \Delta; \Gamma \vdash e : \tau \rightarrow \tau' \]

\[ \Delta; \Gamma \vdash e' : \tau \]

\[ \Delta; \Gamma \vdash e \; e' : \tau' \]
Typing self application

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

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

- But what is the type of $x$?
Typing self application

\[ \Delta; \Gamma \vdash x : \langle \langle \tau_x \rightarrow \text{int} \rangle, \text{int} \rangle \]

\[ \Delta; \Gamma \vdash x.1 : \langle \tau_x \rightarrow \text{int} \rangle \]

\[ \Delta; \Gamma \vdash x.1.1 : \tau_x \rightarrow \text{int} \quad \Delta; \Gamma \vdash x : \tau_x \]

\[ \Delta; \Gamma \vdash (x.1.1) x : \text{int} \]
Recursive definition?

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

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

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

```java
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

\[ \Delta, \alpha :: \kappa \vdash \tau :: \text{Type} \quad \Delta \vdash \tau' :: \kappa \]
\[ \Delta; \Gamma \vdash e : \tau[\alpha := \tau'] \]
\[ \Delta; \Gamma \vdash \text{hide } \alpha :: \kappa = \tau' \text{ in } e : \tau : \exists \alpha :: \kappa.\tau \]

\[ \Delta; \Gamma \vdash e : \exists \alpha :: \kappa.\tau \quad \Delta \vdash \tau' :: \text{Type} \]
\[ \Delta, \alpha :: \kappa; \Gamma, x : \tau \vdash e' : \tau' \]
\[ \Delta; \Gamma \vdash \text{open } e \text{ as } \alpha :: \kappa, x : \tau \text{ in } e' : \tau' \]
Quantify over tuple tail

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

\[
\begin{align*}
\Delta \vdash \tau :: \text{Type} \quad &\quad \Delta \vdash \tau' :: R^{i+1} \\
\hline
\Delta \vdash \tau; \tau' :: R^i \\
\hline
\Delta \vdash \text{End}^i :: R^i
\end{align*}
\]

\[
\begin{align*}
\Delta \vdash \tau :: R^0 \\
\hline
\Delta \vdash \langle \tau \rangle :: \text{Type}
\end{align*}
\]
Efficient object encodings

\[ \exists \alpha. \alpha \land (I \ \alpha) \]

\[ \exists \alpha \leq (I \ \alpha). \alpha \]

\[ \exists \delta :: Type \rightarrow R^1. \mu \alpha.(I' \ \delta \ \alpha) \]

\[ I = \lambda \alpha.\langle \alpha \rightarrow \text{int}\rangle \]

\[ I' = \lambda \delta :: Type \rightarrow R^1. \lambda \alpha.\langle \alpha \rightarrow \text{int}; \delta \ \alpha\rangle \]
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
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