CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 38: Dynamic Types
4 May 01

Static vs. dynamic typing

- This lecture: how to handle incomplete information about run-time type
- Arises even in statically-typed OO languages because only *supertype* is known (e.g. casts and instanceof)

Type safety

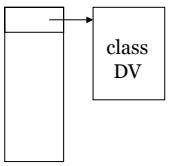
	Strongly typed	Not strongly typed
Statically typed	ML, Pascal, Java, Modula-3	C, Iota, Iota+, C++
Not statically typed	Scheme, PostScript, Smalltalk, SELF, CLOS	FORTH, assembly code

Dynamically typed languages

- Scheme, CLOS, Dylan, PostScript: Variables do not have a declared type – can contain any kind of value
- Operations can be invoked without knowing type of value
- Strong typing: must check value to make sure it has a type supporting the operation
- Must be able to figure out the run-time type of every value!

Unsupported object operations

- Object operations=method invocations
- Need to check for unsupported methods
- Option 1: give every method unique index
- Option 2: Hash table implementation of DV automatically handles unsupported methods
- Option 3: Use standard DV but check method identity
- Field accesses: not a problem for this, treat as methods for other variables



Primitive types

```
x = 48463751374;
x = new Foo;
```

- If variables are untyped, how to know x is actually an int (or not)?
- Must change representation of integers! (booleans, characters, floats, etc.)
  - Box everything into an object?
  - Use two words per value?

Tag bits

- Another approach: reserve 1-3 bits in each word to identify primitive values (handy for GC too)
- *Advantage*: variable in a single word
- *Disadvantage*: extra overhead, smaller range of representable values, pointers

12 = 00001100 → 001100 00
'\f' = 00001100 → 001100 01
new Foo = 00110000 → 001100 11

Tag bit tricks

- **Integers**: use zero bit pattern so integer *n* represented by number 4*n*
  - Adding two integers a + b: just add tagged representation!
  - Multiply: a * b → a*(b shr 2)
- **Pointers**: represent a pointer to an object at address p by p′ = p+3 (don't need to be able to address every byte!)
  [p+k] → [p′+k-3]

new Foo = 00110000 → 001100 11

Dynamic type discrimination

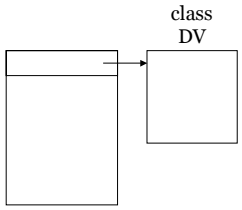
- Even statically typed languages need to find type of object at run time

```
class Number {
  boolean equals(Object x) {
      if (x instanceof Number) {
          return equals((Number)x);
      } else return false;
  }
}
```

- How to implement dynamic type discrimination: instanceof, dynamic cast?

Using DV

- All objects of a class share same DV
- DV identifies which class it comes from
- Idea: implement instanceof as comparison of DV pointer
- x instanceof C ⇒ x.dv == C__DV
- Complete?

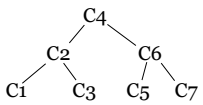


Hashing DV pointer

- Problem to solve: given DV pointer, type T, determine if class(DV) ≤ T
- T may be a class or an interface; consider class with DV_2
- Use pre-initialized global hash table to look up type relationships: Hash DV, DV_2 to look up either true or false
- Construct pseudo-DV's for interfaces so they can be entered in hash table too
- Can update table dynamically (for caching or dynamic loading)

Class indices

- If only single inheritance, can implement instanceof as range check
- Traverse class hierarchy depth-first, number classes
- All classes that are subclasses of C_n have indices in a contiguous range



Class indices

- Class index is stored in the DV

x instanceof C

⇒ x.dv.class ≤ C__index_max && x.dv.class ≥ C__index_min

⇒ (x.dv.class − C__index_min) $\leq_u$ (C__index_max − C__index_min)

- *Limitation*: can't add new classes to system without rewriting code

Run-time type information

- Run-time representation of classes discussed so far: dispatch vectors and method code
- Other useful information: types of fields, layout in memory, supertype relationships
- Useful for: GC, persistence, dynamic code generation (*e.g.*, RPC stubs, Java Beans), dynamic type discrimination

Meta-objects

- How to store dynamic type information? Idea (Smalltalk): use ordinary objects—*meta-objects*
- For every class, introduce an object to represent it
- *Class object* contains information about class: methods, fields, list of supertypes
- Can use class object to do object dispatch (slowly)
- Class DV contains pointer to class object; can find any object's class object

```
Object o; Class c = o.getClass();
```

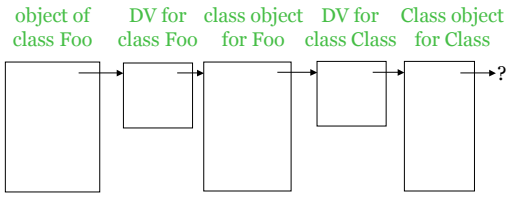

Class Class

- If class objects are ordinary objects, what is the class of a class object?

```
class Class {
    Method[] getMethods();
    Field[] getFields ();
    Class getSuperclass();
    Class[] getInterfaces();
}
class Method {
    Class returnType();
    Class[] getParameterTypes();
    Object invoke(Object receiver, Object[] args);
}
```

- Set of methods supported by meta-objects: *meta-object protocol (MOP)*

Infinite regression?



Dynamic code generation

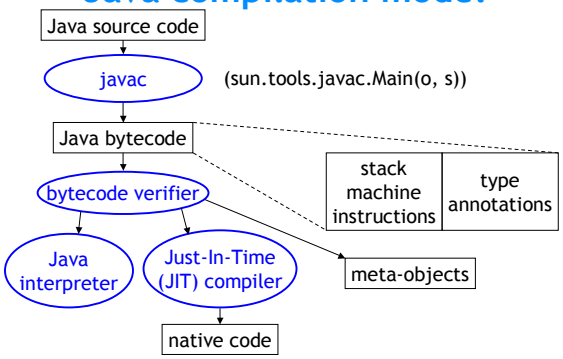
- All information (meta-objects) compiler needs is in running application – can use compiler in the application!
- Application can use compiler as library to generate type-safe code on the fly
  - from source code
  - from partially compiled code (AST, abstract assembly)
- Example: function plotting program
- Convenient if compiler is written in the language it compiles (*e.g.*, Java)

Escaping static limitations

Compiler techniques can be applied to very dynamic systems as well as to statically-typed languages

- untyped languages
- run-time type discrimination
- primitive values treated as objects
- meta-objects expose information about type system as first-class values
- dynamic code generation

Java compilation model



Verification

- Java security depends on
  - access only through public/protected methods
  - hidden private variables
  - unforgeable references to objects (capabilities)
- If Java program is not strongly typed, security of machine can be compromised!
- Java *bytecode verifier* checks Java bytecode to ensure strong typing: *typed intermediate language*
- Java Virtual Machine interpreter runs verified bytecode quickly, avoids run-time checks

JVM bytecode

- stack-machine intermediate code
  - add, sub, mul, rem, div, … : arithmetic
  - dup, swap, pop, … : stack ops (untyped)
- also has local registers/temporaries
  - load, store (untyped)
- built-in object operations
  - invokevirtual, invokestatic, getfield, putfield, …
  - types of methods, fields are declared
- control flow
  - ifeq, goto, ifne, … : conditional branch
- How to show that code is type-safe? (efficiently!)

Type inference

- Type-checking bytecode: need to know
  - type of every stack entry
  - type of every local at every instruction
  - Not present in bytecode file: inferred
- Inference: Start from
  - known argument, return types to method
  - typed object calls inside method
- Use forward data-flow analysis to propagate types to all bytecode instructions
- Data-flow value is type of every stack entry, type of every local
- Meet is pointwise join in type hierarchy

Example

Data-flow value = $(T_1, T_2, \ldots), [0: T'_0, 1: T'_1, \ldots]$

stack types *local types*

$(T_1, T_2, \ldots), [0: T'_0, 1: T'_1, \ldots]$ $(T_1, T_2, \ldots), [0: T'_0, \ldots, i: T'_i, \ldots]$

swap *flow functions* load *i*

$(T_2, T_1, \ldots), [0: T'_0, 1: T'_1, \ldots]$ $(T'_i, T_1, T_2, \ldots), [0: T'_0, \ldots, i: T'_i, \ldots]$

$(\ldots), [\ldots, i: T_i, \ldots]$ $(\ldots), [\ldots, i: T'_i, \ldots]$

combining operator $\sqcap$

$(\ldots), [\ldots, i: T_i \sqcup T'_i, \ldots]$

Object $\sqcup$ int = ?

JIT compilers

- Particularly widely available back end(s) with IR = JVM bytecode
- Code generation by converting stack machine code into quadruples
- Inferred types ⇒ better code
- Compilation *must* be done lazily (on-the-fly): not allowed to load .class files until used
- Generating code quickly is essential → hard to generate good code (but new JITs do it)
- HotSpot: Sun JIT. High-quality profile-driven optimization (*esp.* inlining and specialization), applied to hot code