CS 412/413

Introduction to
Compilers and Translators

Andrew Myers
Cornell University

Lecture 37: Dynamic Types

1 May 00

Static vs. dynamic typing

- Have looked mainly at compiling statically-typed languages
- 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

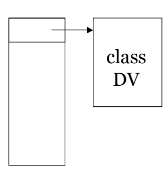
	Strongly typed	Not strongly typed
Statically typed	ML Pascal Iota Iota+ Java Modula-3	C C++
Not statically typed	Scheme PostScript Smalltalk SELF CLOS	FORTH assembly code

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

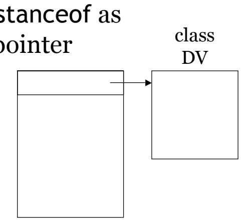
- Even statically typed languages need to find type of object at run time

```
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?

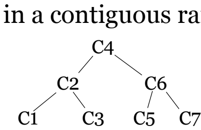


Hashing DV pointer

- 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 have indices in a contiguous range



Class indices

- Class index is stored in the class 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

Primitive types: subtyping?

- Java, Iota+: primitive types have no subtype relation to any other type

  x: object = 10 // NO
- Limits subtype polymorphism: routines written in terms of object not applicable to primitives (work-around: x = new Integer(10))
- Can we allow int <: object?
- x: object declares x as untyped; dynamically typed approaches work

Subtyping for primitives

- *Solution 1: objects.*
- *Solution 2: tagging.*
- *Solution 3: automatic boxing.*
  - Only works in statically-typed language
  - Allow multiple representations of primitive values: boxed and unboxed
  - Primitives are represented in efficient way when type is known; as objects when type is unknown

Automatic boxing

- Use static type to decide when to box
- Conversion from primitive to type object: compiler boxes the primitive

  Object x = 10 ⇒ x = new Integer(10);
- Cast from object to primitive: unbox if cast succeeds

  y: int = (int)x; ⇒

  if (x instanceof Integer) y = ((Integer)x).value;
  else throw ClassCastError;

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
- Class DV contains pointer to class object; can find any object's class object

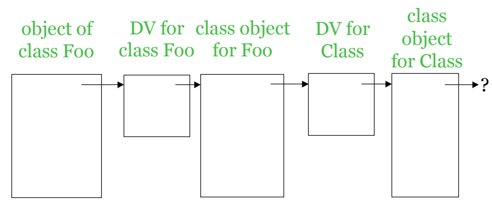
Class Class

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

```
class Class {
    Method methods[ ];
    Field fields [ ];
    Class superclasses[ ];
    Type interfaces_implemented[ ];
}
```

- Set of methods supported by Class: *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 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