CSE 413 Autumn 2008

Implementing Dynamic Dispatch

Dynamic Dispatch

- Recall: In an object-oriented language, a subclass can override (redefine) a method
- When a message is sent to an object, the actual method called depends on the type of the *object*, not the type of the variable that references it
- How?

Conceptual Model

- An object consists of
  - State (instance variables, …)
  - Behavior (methods, messages)
- So we can implement an object as something that contains data and procedures
- But… Not good engineering – multiple copies of method code in each object

Attempt #2

- Instead of replicating the methods in each object, include a set of pointers to the applicable methods
- But… Lots of duplicate pointers per object

Attempt #3

- Instead of having method pointers in each object, have one set of method pointers per class
  - Each object contains a pointer to a “class object”
  - Method calls are indirect to the actual methods in the class object
- A little bit of time overhead per method call
- Need some tweaks for something as dynamic as Ruby

Dynamic Dispatch in Ruby

- Complicatons
  - Modules (mixins) as well as classes
  - Can add or change methods dynamically as the program runs
  - Can include per-object methods as well as per-class methods

Ruby Data Structures

- Every object has a pointer to its class
- A class is represented by a “class object”
  - Every class object contains a hash table with method names and code
- Every class object has a pointer to its superclass
- Search for applicable methods starting in the object and moving up
  - If you hit the top without finding it, “message not understood”

Complications

- Mixins
  - One object per mixin, searched after the class object and before the superclass
- Per-object methods
  - Define a “virtual class” of methods for that object that is searched first
- What is the class of a class object?
  - Interesting question… left as an exercise

Types for O-O Languages

- Java, C++, and others are *strongly typed*
- Purpose of the type system: prevent certain kinds of runtime errors by compile-time checks (i.e., static analysis)

O-O Type Systems

- “Usual” guarantees
  - Program execution won’t
    - Send a message that the receiver doesn’t understand
    - Send a message with the wrong number of arguments
- “Usual” loophole
  - Type system doesn’t try to guarantee that a reference is not null

Typing and Dynamic Dispatch

- The type system allows us to know in advance what methods exist in each class, and the potential type(s) of each object
  - Declared (static) type
  - Supertypes
  - Possible dynamic type(s) because of downcasts
- Use this to engineer fast dynamic type lookup

Object Layout

- Whenever we execute "new Thing(…)"
  - We know the class of Thing
  - We know what fields it contains (everything declared in Thing plus everything inherited)
- We can guarantee that the initial part of subclass objects matches the layout of ones in the superclass
  - So when we up- or down-cast, offsets of inherited fields don't change

Per-Class Data Structures

- As in Ruby, an object contains a pointer to a per-class data structure
  - (But this need not be a first-class object in the language)
- Per-class data structure contains a table of pointers to appropriate methods
  - Often called “virtual function table” or vtable
  - Method calls are indirect through the object’s class’s vtable

Vtables and Inheritance

- Key to making overriding work
  - Initial part of vtable for a subclass has the same layout as its superclass
    - So we can call a method indirectly through the vtable using a known offset fixed at compile-time *regardless of the actual dynamic type of the object*
  - Key point: offset of a method pointer is the same, but it can refer to a different method in the subclass, not the inherited one