CSE 341: Programming Languages

Spring 2006

Lecture 24 — Defining and Implementing Dynamic-Dispatch

Today

Smalltalk look-up rules, a lower-level view of dynamic dispatch

Look-up rules

How we *resolve* various "symbols" is a key part of language definition.

- In many ways, FP boils down to first-class functions, lexical scope, and immutability.

In Smalltalk, we *syntactically distinguish* variables (which resolve to objects), messages (which determine what method is called), and a few special names (true, false, nil, self, super)

- Java makes the same distinction
- Messages are *second-class*

Name Resolution

To resolve a variable (e.g., `x`):

- Like in ML or Scheme, if a use of `x` is in the lexical scope of code-block variable (`[:x | ...]`) or local method variable or parameter, we resolve `x` using the environment in which the code-block or method-body was defined.
  - Smalltalk implementation must build closures (code/environment pairs as in ML/Scheme)
- Else if a use of `x` is in a method `m` of class `A` (because `A` or a transitive-superclass of `A` defines `m`) and `x` is an instance or class variable of `A` (because `A` or a transitive-superclass of `A` defines `x`), then `x` resolves to a field of the object `self` resolves to.
- Else if `x` is a global (e.g., a class object), then `x` resolves to that.

Note: Ignoring pool dictionaries.

Now messages

To resolve a message (e.g., `m`):

- A message is sent to an object (e.g., `expr m`), so first evalaute `expr` to an object `obj`.
- Get the class `a` of `obj` (every object has a class).
- If `m` is defined in `A`, invoke that method, else (recursively) look at superclass(es) of `A`.

What about `self`?

As always, evaluation takes place in an environment.

In every environment, `self` is always bound to some object. (This determines message resolution for `self` and `super`.)

Key principles of OOP:

- Inheritance and override (last slide)
- Private fields (just abstraction)
- *The semantics of message send*

"send `m(x,y+2,...)` to `obj`" means evaluate the body of the method to which `m` resolves for `obj` in an environment with argument names mapped to actual arguments *and `self` bound to `obj`*.

That last phrase is exactly what "late-binding", "dynamic dispatch", and "virtual function call" mean. It is why code defined in superclasses can invoke code defined in subclasses.

Some Perspective on Late-Binding

Later we will discuss design considerations for when late-binding is a good or bad thing. For now, here are some opinions:

- Late-binding makes a more complicated semantics
  - Smalltalk without `self` is easier to define and reason about
  - It takes months in 142/143 to get to where we can explain it
  - It makes it harder to reason about programs
- But late-binding is often an elegant pattern for reuse
  - Smalltalk without `self` is not Smalltalk
  - Late-binding fits well with the "object analogy"
  - Late-binding can make it easier to localize specialized code even when other code wasn't expecting specialization

A Lower-Level View

Smalltalk clearly encourages late-binding with its message-send semantics.

But a definition in one language is often a pattern in another...

We can simulate late-binding in Scheme easily enough

And sketch how compilers/interpreters implement objects

- A naive but *accurate* view of implementation can give an alternate way to reason about programs

The Key Idea

The key to implementing late-binding is extending all the methods to take an extra argument (for `self`).

So an object is implemented as a record holding methods and fields, where methods are passed self explicitly.

And message-resolution always uses `self`.

What about classes and performance?

This approach, while a fine pattern, has some problems:

- It doesn't model Smalltalk, where methods can be added/removed from classes dynamically and an object's class determines behavior.
- It is space-inefficient: all objects of a class have the same methods.
- It is time-inefficient: message-send should be constant-time, not list traversals.

We fix the first two by adding a level of indirection: put a single class field in an object and have a global class-table.

We fix the third with better data structures and various tricks.

Nonetheless: Without dynamic class changes, the "method slot" approach and "class field" approach are equivalent.

Really Implementing Late-Binding

- We have seen late-binding as a Scheme pattern
- In reality, we have learned roughly how OO implementations do it, without appealing to assembly code (where it really happens)
- Using ML instead of Scheme would have been a pain:
  - The ML type system is "unfriendly" for `self`.
  - We would have roughly taken the "embed Scheme in ML" approach, giving every object the same ML type.
  - But to be fair, most OO languages are "unfriendly" to ML datatypes, first-class functions, and parametric polymorphism.