

1 Featherweight Java

We continue our discussion today on Featherweight Java. Please refer to the text, *Types and Programming Languages*, on pages 257 and 258, to follow along with the discussion. The rules for method body lookup cover two cases: when the method to lookup is found in the current class, and when it is not and must subsequently be looked up in the superclass. These two rules are presented in Figure 19-2 on page 257 of the text. The function `mbody` is implemented using these two rules. `mbody` will return a pair consisting of the arguments of the requested method and its body. Figure 1 illustrates how inheritance works: if the method we are looking for is found in class C5, then `mbody` does not recur to its superclass. If is not found in C5, then `mbody` recurs to C2 then C1, and so on, until the method is found or the top of the hierarchy is found (in which case the method was not found).

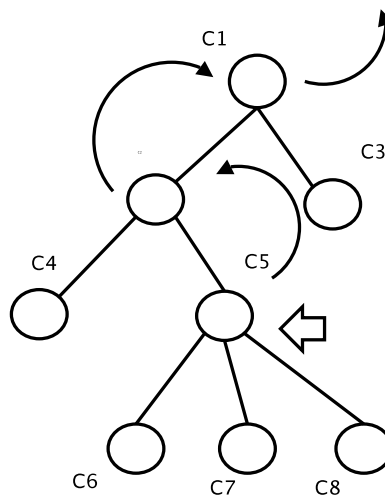


Figure 1: Inheritance graph

Method overriding is implemented in this system as a predicate that

checks to see if a given method can be overridden. For instance, given a method name, a superclass, and a signature, `override` will return either true or false indicating whether it is alright to override the specified method. At first glance, it isn't clear why the override rule uses implication to decide if the method can be overridden. This is because implication captures two important qualities that must hold in order to override: first, a method can be overridden if it already exists in the superclass. Second, a method can be overridden if its signature matches the one declared in the superclass.

The evaluation rules on page 258 of the text are presented in a small-step semantics. The reason why objects are represented as `new C(v)` is because this is our only notion of an object in this syntax. Thus, field access, method invocation, and the rest of the operations will be performed upon the evaluation of an object represented in this fashion.

When evaluating a method invocation, two substitutions are made in the resulting expression. One of which is for the argument list, the other is for the `this` variable in the new object. This style is very similar to the beta-reduction rule of the lambda-calculus. Substitution is very important because nothing interesting can be done without it in this system. If we removed the ability to perform substitution, we could use assignment with references, using cyclic structures to implement recursion. It would be interesting to see substitution implemented using other means, like assignment, for instance. The main difference between the two schemes is scope. Substitution has local scope, while references usually have global scope.

The rule for object casting presented on page 258 is rather interesting. In order to cast an object to another type, this rule simply checks to make sure the casting operation is an upcast, and not a downcast. In other words, if the type we are casting to is a superclass of the type the object already has, then the cast operation is simply "thrown away".

The following problems due Monday, March 21: 19.4.3,4,6.