CS 434 Meeting 5 — 2/16/06

Announcements

1. Debugging lab?

Dealing with Nested Scopes at Run-time

- Suppose we are compiling a program with static nesting like that shown in Figure 1. As discussed in a previous lecture, we can represent the static nesting of scopes using a tree diagram like that shown in Figure 2.
- Now, consider the dynamic behavior of such a program. In particular, suppose that while it is running the program constructs one object of each of the classes declared, and then constructs an additional object of class Three and an additional object of class In3.
- When executing the code of mIn3, the variable y is referenced. This is a reference to the y declared in Three. There are, however, two allocated objects of class Three (and, of course, there could be hundreds). How do we determine at run-time which copy of Three to use?
- Rather than really answer this question, we will simply provide a mechanism to solve it. Just as the tree in Figure 1 can represent the static structure, a tree like that shown in Figure 3 could be used to represent the dynamic nesting. That is, for each object of a class like In3, we could store a pointer to the object of class Three that should be used when a non-local reference to a variable like x is encountered.
- We will assume in our compiler that every object in the heap will contain such a pointer to the correct object of any surrounding class. We will call these pointers *static links*. Each static link must be stored at a fixed, known location within its object. Therefore,

```
public class NestingExample {
    int x; int y;
        class One {
          Two y;  int z;

          class In1 {
              One y;
            }

          void m1() { int y;  y = 1;  x = y;  }
        }

     class Two {
        Three x;
        void m2()  x.m3();
        }

      class Three {
          Two x; One y;

          class In3 extends One {
              void mIn3() {
                  x.m2();
                  m1();
              }
          }

          void m3() { /* ... */ }
        }
}
```

Figure 1: A class definition skeleton illustrating nested declarations

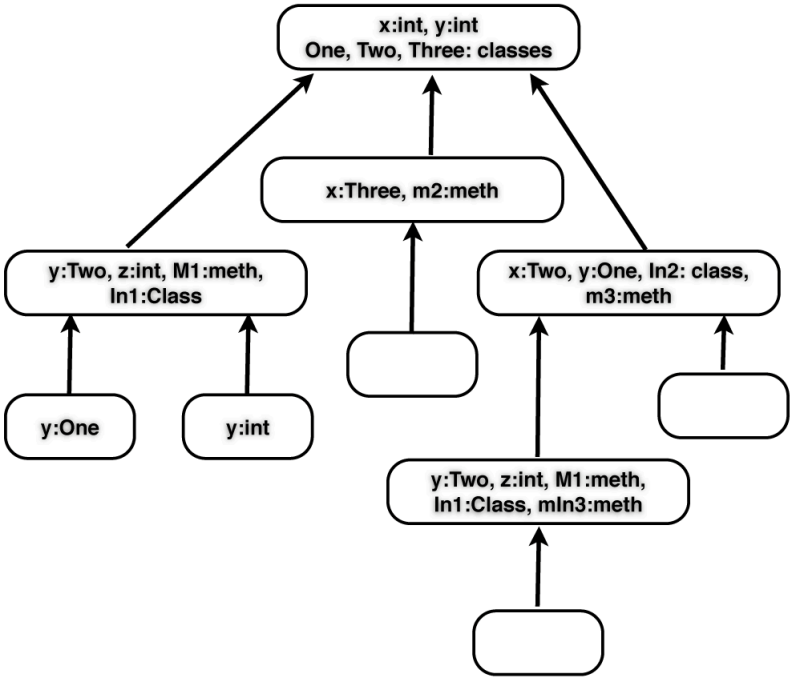

Figure 2: Tree of scopes corresponding to Figure 1

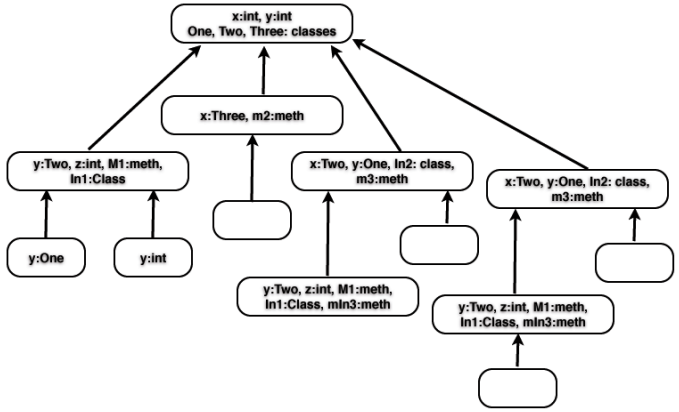

Figure 3: Tree of run-time objects corresponding to Figure 1

we will store it before all the variables (before or after the dispatch table pointer).

- Given such static links, we can follow a path to access any variable accessible from the current object, as long as we have a pointer for the current object! We will assume that such a pointer is stored in a register. In fact, to ensure that we can efficiently represent local variables and the instance variables of the active object we will assume that one register holds a pointer to the current method frame and another to the current object.
- We won't worry about how the static links gets set for now. We will just assume they are there when considering how to generate code for references to non-local variables.
- We will have to consider how to set these pointers when implementing the new operation.

More on Order of Declaration Processing

1. Last week (and in the handout for the first phase), I explained that it is necessary to make (at least) two complete passes over

the abstract syntax tree to correctly process the declarations in a Woolite program.

- The goal of the first pass is to build correct declaration descriptors for all methods including information about return types and formal parameter types and to insert appropriate bindings in the hash table that maps class/name pairs to method declaration descriptors.
- Even though the main goal is to create declaration descriptors for methods and enter them in the hash table for resolving qualified method invocations, you will also need to build declaration descriptors for class and methods and maintain scope information:
  - The return types and parameter types of methods may involve classes, so we clearly need to have created declaration descriptors for classes so that we can use them to describe these types.
  - In order to interpret the names used as return types and formal types, we need to follow normal scope rules.
  - According to normal scope rules, variable and method declarations in an inner scope can hide class declarations from outer scopes. As a result, to ensure we interpret method return and parameter types correctly, we must include such declarations in our system for representing the names accessible in a scope.
  - For similar reasons, your processing during the first pass will have to account for the inheritance of methods (since such method names might hide class names defined in surrounding scopes).

  During this first pass, you can ignore method bodies and local variables.
- The goal of the second pass will be to use process the code and local variable declarations in method bodies.
  - The hash table built during the first phase will provide all the information needed to handle qualified method invocations
  - In order to handle unqualified references to method, type, and variable names you will need to (painfully) reconstruct the lists and stacks used to keep track of the meanings of names and you revisit the various scopes in the program.

2. Even within the first pass, processing order is critical.

3. During the first pass of semantic analysis of a program, for each class you need to:

   (a) Create a declaration descriptor for each declaration found in the class (i.e. the top level declaration, not those nested in nested class declarations).

   (b) Put the declaration "in scope" by adding it to the correct scope's list of bindings and the correct identifier's stack of bindings.

   (c) Set (many of) the fields of each declaration descriptor that are independent of the correct meanings of other names in its scope:

   - The type of declaration,
   - Its declaration level,
   - The line number it occurred on and the identifier involved,
   - etc.

   (d) Put bindings for the methods declared in the hash table that maps class/name pairs to method declarations

   (e) Add the declarations of inherited methods to the scope for the class.

(f) Add entries for inherited methods paired with the enclosing class to the method hash table.

(g) Set (many of) the fields of each declaration descriptor that depend its type and on the correct meanings of other names in its scope (primarily type names):

- Method return types,
- Method formal parameter types,
- Instance variable types,

(h) Check to see whether method declarations that attempt to override inherited methods have matching return and parameter types.

4. These tasks can be completed in many different orders, but there are many constraints that require certain tasks be performed before others. To understand the constraints, consider the class "Nasty" shown in Figure 4 below.

5. First, it should be clear that we cannot perform step (g) until step (a) has been completed since we obviously cannot use a classes declaration descriptor to represent a variable's type or a return type if the declaration descriptor hasn't been created.

   - Consider the declaration of the variable `var`. If we try to process declarations in order, and try to store var's type in its descriptor before we have created a descriptor fo A, we cannot possible succeed!

6. Similarly, we cannot perform step (g) until step (b) has been completed, since without putting the declarations "in scope" we will have no way of finding the correct declaration descriptor for a class name while processing a type description.

   - Again, we would not be able to process the class name A in the declaration of `var`.

```
class Nasty {

    A var;

    class A {
        int meth() {
            Bad x;
            x = new Bad;
            return x.meth().meth();
        }
         void C() { . . . }
     }

    class Bad {
        C meth () {   return new C;    }

        int C;
    }

    class C extends A {
          C meth3() {  return var;  }
     }

     int meth2( C p ) {
        A y;
        y = new A;
        return y.meth();
     }
}
```

Figure 4:

- The class Bad illustrates a more subtle reason for performing step (b) on all of a class' declarations before worrying about step (g). The first declaration in this class looks fine at first. It defines a class that returns an object of class C. The second declaration in Bad, however, associated a new meaning with C as a variable rather than a class. A variable name cannot be used as a return type. The declaration of meth should therefore be detected as an error. This can only be done if the return type of meth is processed after all the declarations made in Bad have been added to the scope.

7. This suggest that "pass 1" must include at least two subpasses.

   - During the first we will create declaration descriptors for every declaration in a class and put all the descriptors "in scope" by creating appropriate bindings.

     While we are doing this, we might as well do as much of step (c) as possible (or as much as we feel like).
   - During the second subpass, we will fill in the details described in step (g).

8. Performing step (d) is the real goal of the first pass. Since all it requires is that the method declaration descriptors exist (and not that they be completely filled in), we might as well do it on the first subpass.

9. Because it may change the meanings of names within the scope of a class, it is also necessary to perform step (e) before step (g).

   - This time, consider class C which extends A. Because it extends A, class C effectively contains a declaration of the method C which hides the name of the class itself within the scope which is the class body. As a result, the definition of meth3 within C is invalid. In this context the "C" used to specify its return type must be interpreted as the name of the method inherited from class A.

10. Since we will have to scan through all the superclasses to add inherited methods to the scope (i.e. perform step (e)), we might as well add the methods to the hash table while we are at it (i.e. perform step (f).

11. Finally, step (h) cannot be performed before step (g) for a given method since to determine whether an inherited method and a method defined within a class represent method overriding or a name conflict we have to know the return type and parameter types of both method declarations.

Type Checking

1. Type checking is basically the enforcement of a set of rules designed to ensure that only valid operands are provided to the various operations that can be invoked within a program.

   - Type systems are a big topic in programming language design but typically are given little attention in compiler courses and textbooks.
   - The specification of a type system requires:
     - a language/notation for describing types.
     - a set of rules specifying how to associated type descriptions with language constructs (primarily expressions).
     - a set of rules specifying restrictions on the types associated with expressions appearing in various contexts.

2. The description of types in "conventional" languages typically includes:

   - A set of names for primitive types (int, bool, etc.),
   - Constructors for composite types including:
     - pointer(t)
     - array(i, t)

  - record constructor ($id_1 : t_1, id_2 : t_2, ...$)
  - product constructor ($t_1 x t_2 x ... t_n$)
  - function constructor ($t_1 \rightarrow t_2$)
- The ability to bind and use type names.
  - Essential for the description of list, trees and other recursive types.

3. The rules for associating types with expression are typically inextricably intermixed with the rules specifying type constraints. Basically, if a expression violates the constraints, none of the rules for assigning types will apply.

   - For example....
     Given an expression, e, of the form:

     $$x + y$$

     where x and y have type "int" then the type of 'e' is "int".
     Given an expression, e, of the form:

     $$f(x)$$

     where f has type $t_1 rightarrow t_2$ and x has type t_1 then e has type t_2.