CS 434 Meeting 4 — 2/14/06

Introduction

1. Shall we do a debugging lab?

Outline

1. Structure of semantic processing code
2. Handling targeted method invocations
3. Handling inheritance (as a symbol table issue)
4. Run-time Memory Organization
   - Allocation Units
     - Method frames
     - Objects
     - Dispatch Tables
     - Arrays
   - Frame layouts
   - Static Links
   - Garbage collection issues

Structure of Semantic Processing Code

1. The code you write to traverse the syntax tree should **not** be composed of a single procedure containing one big case statement. Instead, have distinct processing routines (with little case statements) for each major class of nodes (i.e. statements, expressions, . . .).

Handling Qualified Method Invocations

1. The scheme I have described does not handle the processing of invocations of the form:

   obj . M (arguments)

   - The interpretation of '*M*' in such an invocation should not change when we enter a new scope unless the type of value produced by the expression *obj* changes.. In particular, it should not matter if the same symbol *M* is redeclared as the name of a new variable, class or method in the new scope.

2. When the semantic processor encounters a subtree corresponding to a selection like:

   obj . M

   it needs some way to locate a binding that associates the name M with the appropriate declaration descriptor for the method (It can't just look on the top of the stack hanging off of M's identifier descriptor).

3. The key to qualified method component names is to realize that a method name alone is potentially ambiguous. Only when paired with some particular class can a component name be interpreted unambiguously.

   - Consider how one would interpret the name 'M' in a context where there where 2 distinct classes that included a method named 'M'.

     ```
     C1 x;
     C2 y;
     C2 z;
     . . . x.M() . . .     y.M() . . .
     z.M() . . .
     ```

   - It is not the identity of the variable from which field is selected but the identity of the type of the variable that matters.

   Thus, there should be one declaration descriptor (and one binding?) for each 'structure type/component name' pair.

4. There are many ways to solve this problem.

   Approach 1 We will already have a linked list of method declaration descriptors pointed to by each class' declaration descriptor. To

process `x.M` do a linear search for "M" in the linked list attached to the declaration descriptor for x's class.

This is simple, but linear lookup may not be efficient enough.

Approach 2 Hang a hash table of field declaration descriptors off of each class' declaration descriptor (or an AVL tree or a ...).

This will tend to be complicated and space inefficient.

Compromise Approach Employ a single dictionary structure (hash table or binary tree) in which the search keys are class/ method name pairs.

- The keys used to look up items in this table will actually be pairs composed of a pointer to the class' declaration descriptor and a pointer to the method name's identifier descriptor.
- The semantic processor will have access to both parts of the needed pair when it processes a selection like obj.M.
- We will use a hash table for this purpose
- Note that we will have to add a distinct entry to this hash table for every class by which a method is inherited.

5. Overview of required processing:

   - When processing a class' body one must:
     (a) Create a declaration descriptor for the class itself.
     (b) For each method specified (or inherited), create a binding between the method name and its declaration descriptor and enter it into the method hash table using as the class-descriptor/ identifier-descriptor pair as key.
   - When processing a reference to a component one must:
     (a) Determine the type of the object from which the component is being selected.
     (b) Look up the pair composed of the variable's type and the field name in the hash table. (Note: the type here must be a class and we will use the class descriptor rather than the type descriptor).

Dealing with Inheritance

1. When one class inherits a declaration from another, we won't make a new declaration descriptor. We will just make a new binding to the existing declaration descriptor and place this binding in the current scope.

   - This means that when semantic processing is done, all references in the syntax tree to the same declaration will refer to the same descriptor regardless of which subclass the reference involved.

2. In Woolite, to make things a bit simpler, only method names are inherited (variable and class declarations are considered private). Also, to simplify handling inheritance, forward references are not allowed in extends clauses.

3. As a result of Woolite's design, when we process a class, the declaration descriptor of any superclass will already contain:

   - a pointer to a list of all methods defined within the body of the superclass definitions, and
   - a pointer to any superclasses of the superclass.

4. Your compiler will run up the chain of superclasses and through each superclass' list of methods creating bindings and adding them to the new class' scope.

Run-time Storage Management

1. Our goal at this point is to understand enough about run-time organization to understand the information the compiler must collect while processing declarations.

2. The key idea is that at run-time all memory will be allocated in blocks. There will be frames (i.e. activation records) for method invocations and objects allocated on the heap in response to the construction of **new** objects and arrays.

3. Ultimately, our understanding of run-time storage management will have to include knowledge of how to generate code to determine the address of any block of storage that contains a variable we want to access.

- That is, we will have to figure out where we can find pointers to the current method's frame, the current object, and objects corresponding to all the scopes (i.e. classes) that textually surround the code of the executing method.

4. For now, however, we will just assume we can find the necessary pointers to blocks of memory and think about the organization within the blocks to make sure that we collect enough information while processing declarations to determine the offset to each variable/method within the blocks of memory we allocate.

5. As a start, we can identify four types of structures that will be allocated for Woolite programs. (Good news, these will also be sufficient for many other languages).

   Method Activation Records Each time a method is invoked, we will push a block of storage to hold the method's parameters and local variables onto a stack. This block of memory will also hold things like the return program counter and saved values of some critical pointers (like the pointer to the caller's frame and the previously active object).

   Object Records Each time a program constructs an object (not an array), we will allocate space on the heap to hold all of the instance variables declared in the object's class (including inherited variables — even though we can't access them).

   Dispatch tables When a method is invoked, we need a way to determine how to interpret the name of the method given the actual type of the object involved. To make this possible, each object structure allocated on the heap will contain a pointer to a table of pointers to the code for the method's associated with the type of the object.

   These don't need to be dynamically allocated. There will be one per class and they can be statically allocated before execution begins.

   Arrays We will also need to allocate blocks of storage to hold the elements of arrays created by constructions. We distinguish these from the allocation of single objects because they won't need to contain dispatch tables.

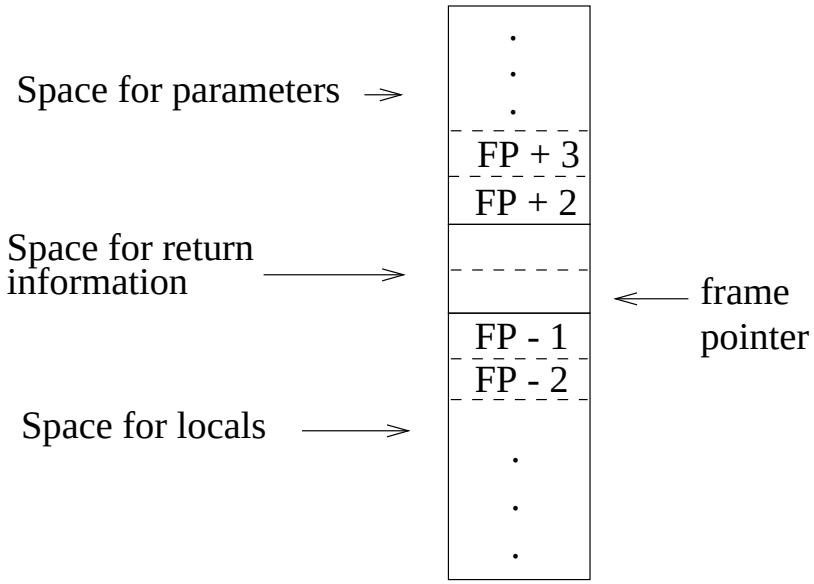
6. For variables (i.e. instance variables, local variables, and formal parameters) stored within activation records and objects, the semantic processor's task it to determine the offset to each variable within the block of memory allocated.

   - Variables are often allocated in order within the block to which they belong as their declarations are processed.
   - To know each variable's offset, all the compiler has to do is keep track of the total amount of space used by all of the previously processed variables.
   - In general, this will involve being able to determine the amount of memory required for each variable, i.e. required for the type associated with each variable.
   - In Java-like OO languages, this is made somewhat easy by the fact that all values of non-primitive types are represented by pointers to their own block of memory on the heap and therefore all variables of all such types require the same amount of space, the space required to store a pointer.
   - In Woolite and on the 34000, there is only one primitive type, int, and it takes the same amount of space as a pointer, 1 word. Therefore, all variables take one word, and the compiler does not have to do any work to determine the type of a variable. The amount of space consumed by n variables is just n.

7. The actual layout of activation records for methods is strongly influenced by the hardware support for function/method calls on the target system.

   - Most machine architectures have special instructions to support calling procedures and allocating memory for their variables. These instructions require the code to follow the architecture's conventions about where variables are located relative to the register that points to the activation record.

8. To get a feel for the impact of a machine architecture on variable allocations, recall the stack frame layout conventions of the MC68000/WC34000:



9. A compiler for such a machine needs to keep track of offsets for local variables and parameters separately since they "grow" in different directions.

10. For Woolite, it is sufficient to simply count the number of parameters and local variables. It is then easy to compute the offset to the nth variable or nth parameter.

11. The layout of blocks of memory on the heap is slightly complicated by the ability to extend classes.

    - Although Woolite does not allow code within the methods of a subclass to access the variables declared in a superclass, if a method associated with the superclass is invoked on an object of the subclass, the method can reference the superclass variables, so each object of a given class must include space for all variables declared in all superclasses.
    - If class P is a superclass of B and P declares a variable x, then the offset to x must be the same in P objects as in B objects, because the code we generate for methods declared in P will use P's offset to x when x is referenced even if such a method is invoked on an object of class B.
    - This simply means that when we start counting the number of variables in a subclass, the count should start out equal to the total number of variables declared in all superclasses.

12. In an OO language the supports inheritance, we cannot in general implement method invocations by branching directly to the code for the correct method. Instead, we must build tables of pointers to the code for each classes methods and dynamically select the correct methods to invoke at run-time.

    - Again, assume the P is a superclass of B and that P defines a method, m, which is overridden by B.
    - If the program includes a variable, o, of type P, then we can assign o to refer to either an object of type P or B (or any other subclass of P).
    - As a result, when processing the invocation

      ```
      o.m( ... )
      ```

      we cannot tell at compile time which version of m should be executed.
    - The method executed will depend on the class of the object assigned to o when the invocation is executed, but not the precise object. Therefore we can solve this problem by building a table containing pointers to the method code to use for each method of each class. We will call this a dispatch table.
    - To make this work, we have to be able to put the pointer to the code for a method like m at the same offset in the table of pointers for each subclass of P (including those that override m).
    - Therefore, as with variables, we will basically assign offsets to methods by counting the total number of methods declared in a class and its superclasses, but only counting methods that do not

override inherited methods. Instead, each method that overrides another will be assigned the same displacement as the method it overrides.

13. In addition to building these tables, we have to be able to find the correct table quickly when executing an invocation like

    ```
    o.m( ... )
    ```

14. As a result, each object allocated on the heap will have to include a pointer to the dispatch method for its class.

15. Arrays don't require dispatch table pointers. Each array will be a collection of integers (which require no dispatch tables) or pointer to objects that each contain their own dispatch table pointers.