CS 434 Meeting 17 — 4/13/06

Anouncements

1. Phase 3? (Building a little parser with Yacc) should be finished by the end of the week.

Generating Code for Methods

1. It seems like it would be a good idea to review/summarize the suggestions for generating code for methods and method invocations.

   - First, at this point, the idea is that the actual code generated for a method's body would look like:

     ```
     methLabel     LINK    A6,#-sizeOfLocalsAndTemps
                   MOVE    A5,objPtrSaveDisp(A6)
                          ...
                     code for the statements in the body
                          ...
                   UNLK    A6
                   RTD     #-numberOfParams
     ```

     Note: The correct value for "sizeOfLocalsAndTemps" won't be known until after the code for the body is generated. You can use a symbolic label in the LINK instruction and then generate an assembler EQU directive after the RTD.

   - Second, while processing of each class, you should generate a method jump table of the form:

     ```
     jmpTabLabel     DC      superClassTableLabel
                     JMP    method 1
                     JMP    method 2
                         ....
     ```

     Note: We haven't actually talked about the initial DC yet, but I will shortly.

   - Next, during semantic processing, you will restructure the tree so that any invocation of the form:

     ```
     methname( ... )
     ```

     in which "methname" refers to a method from a surrounding class will be rewritten as a tree for an invocation of the form

     ```
     expr.methname( ... )
     ```

     where the tree for "expr" is basically a chain of ref-var nodes designed to generate code that will take the correct number of steps up the chain of static links to get to the object with which "methname" is associated.

   - Next (to last), when you encounter an invocation of the form:

     ```
     methname( param1, param2, ... )
     ```

     you will generate code for the form:

     ```
     SUB    #1,A7      // space to save A5?
       code for param1
     MOVE   p1-op,-(A7)
       code for param2
     MOVE   p2-op,-A7)
        ...
     MOVE   (A5),Ai      // get addr of method tab
     JSR    1+methNum*2(Ai)
     ADD    #1,A7      // pop space for saved A5
                 or
     MOVE   (SP)+,some-temp // access return value
     ```

   - Finally, when you encounter an invocation of the form:

     ```
     expr.methname( param1, param2, ... )
     ```

     you will generate code for the form:

     ```
     SUB    #1,A7      // space to save A5?
       code for param1
     MOVE   p1-op,-(A7)
       code for param2
     MOVE   p2-op,-A7)
        ...
     ```

```
   code for expr
MOVE   expr-op,A5
MOVE   (A5),Ai      // get addr of method tab
JSR    1+methNum*2(Ai)
MOVE   objPtrSaveDisp(A6),A5  // restore A5
ADD    #1,A7      // pop space for saved A5
             or
MOVE   (SP)+,some-temp // access return value
```

2. As mentioned above, to make all this work, you should restructure the trees for invocations of non-local methods so that they look like trees for qualified invocations. To do this you will add a subtree that describes an expression that evaluates to the object on which the method should be invoked.

   - The root of the expression subtree will be a refvar node.
   - The displacement in the root ref-var node will be the displacement to the static link within an object (probably 1).
   - The base address pointer for the refvar node will either be another tree rooted at a similar refvar node or Nthis.
   - The number of refvar nodes chained together before you get to an Nthis should equal the difference between the level of the class in which the invocation occurs and the level of the class with which the method being invoked is associated (which is not necessarily the same as the class in which the method was declared thanks to inheritance).

3. Finally, since all of this depends upon the idea that objects will contain pointers to method tables and also contain static links, we should talk about implementing the "new" operation for a moment.

   - Eventually, your code for "new" will have to interact with some sort of library routine that talks to the garbage collector. For now, however, let's just assume that you never run out of memory.
   - At the end of your code, generate instructions like:

     ```
     freePtr        DC   freeAreaStart
     freeAreaStart  DS   1
     ```

     This will ensure that the word labeled "freePtr" will contain the address of the first unused wordin memory.
   - When you have to generate code for a new operation output something like:

     ```
     MOVE   freePtr,Ai        // Get addr of new obj
     ADD    #objSize,freePtr  // Increment free ptr
     LEA    classMethTab,(Ai) // Set ptr to method tab
     MOVE   1(A5),Aj          // Get
     MOVE   1(Aj),Aj          //     static
         ...                  //        link
     MOVE   Aj,1(Ai)          // Set static link
     ```

   - Later, we will replace the first two instruction with a JSR to a library routine that will allocate a given amount of space (calling the garbage collector if necessary).

Saving Registers during Calls

1. When generating calls within expression and code for non-void methods, you must include code to ensure that register values are preserved during all calls.

   (a) Registers are typically partitioned into:

   Callee saved registers whose values the called method must save and restore (if it uses them).

   Caller saved registers whose value the calling method must save and restore (if it depends on them after the call).

   It is possible (and not uncommon) to put all registers in one group or the other (callee-saved is the favorite).

   (b) Callee saving has the advantage of keeping the total size of your code small (each proc only contains one set of instructions to save registers).

Note: You have to generate the register saving instructions before you know what registers need to be saved. The fact that you are generating assembly code lets you leave this problem to the assembler by using a symbolic name for the mask that determines what registers need to be saved.

(c) Caller saving has the advantage that you only save the registers in use at a particular call rather than all registers ever used in the procedure (although you may end up saving registers that aren't altered by the called procedure).

2. A register allocator can be designed to take advantage of these two classes of registers:

   - Use caller saved registers to hold values whose usefulness begins and ends in a section of code including no calls (entire body of any "leaf" procedure.
   - Use callee saved registers for values produced before and needed after a call.

3. Where are register values saved?

   (a) In the caller's frame — for caller saved registers.

   (b) In the called method's frame — for callee saved.

   In both cases, the stack pointer will be incremented as part of the register saving (rather than treating them as locals counted in determining localsize).

The Correctness of LR(0) parsing

1. The correctness of the LR(0) parsers we have discussed rests on the theorem:

   Theorem $[N \rightarrow \beta_1.\beta_2] \in \Delta(\pi_0, \gamma)\ iff\ [N \rightarrow \beta_1.\beta_2]$ is valid for γ.

2. Rather than completely prove the theorem, I would like to prove it in one direction (the easy one) and let you convince yourselves of the other direction. In particular, we will prove that:

   $[N \rightarrow \beta_1.\beta_2] \in \Delta(\pi_0, \gamma)$ only if $[N \rightarrow \beta_1.\beta_2]$ is valid for γ.

3. The proof of this theorem is simplified by identifying a way to partition the set of LR(0) items associated with a state of the LR(0) machine into two parts.

   Kernel items We say that an LR(0) item is a kernel item if either:

   (a) it is the item $[S' \rightarrow .S]$, or

   (b) it is of the form $[N \rightarrow \beta_1.\beta_2]$ for $\beta_1 \neq \epsilon$.

   All other items are called non-kernel items.

   Note that the set of items associated with a state in the LR(0) machine is just the closure of the set of kernel items associated with the state. In particular, all items in goto(π,x) are kernel items.

4. Given this partition, we can break the proof of this half of the theorem into two lemmas:

   Lemma 1: Given a set π of LR(0) items valid for some $\gamma \in (V_n \cup V_t)^*$, all items in closure($\pi$) are valid for γ.

   Lemma 2: For kernel items, $[N \rightarrow \beta_1.\beta_2] \in \Delta(\pi_0, \gamma)$ only if $[N \rightarrow \beta_1.\beta_2]$ is valid for γ.

About this closure stuff...

1. Recall the definition for the closure of a set of LR(0) items:

   closure Given a set π of LR(0) items for a grammar G with productions P, we define closure(π) to be the smallest set of LR(0) items such that:

   (a) closure(π) $\supseteq \pi$

   (b) if $[N_1 \rightarrow \beta_1.N_2\beta_2] \in$ closure(π) and $N_2 \rightarrow \beta_3 \in P$ then $[N_2 \rightarrow .\beta_3] \in$ closure(π)

2. When we build an LR(0) machine we use the following algorithm to compute closures:

   - An algorithm to compute closure(π)

(a) set π' equal to π.

(b) while there is some $[N \rightarrow \beta_1 . M \beta_2] \in \pi'$ such that $M \rightarrow \beta_3 \in P$ and $[M \rightarrow . \beta_3] \notin \pi'$ add $[M \rightarrow . \beta_3]$ to π'.

3. How can we prove that the sets described by the definition and produced by the algorithm are the same?

   By identifying the invariant of the loop. Namely:

   > At the beginning (and end) of each iteration of step (b), π' is a subset of the closure of π.

   - This is clearly true before the first iteration.
   - If it is true before any subsequent iteration, then the item $[M \rightarrow . \beta_3]$ that the iteration will add must also be a member of $closure(\pi)$ by the definition of *closure*.

   These facts establish that π' will always be a subset of $closure(\pi)$. If the loop terminates, then we know that π' must contain $closure(\pi)$ and therefore $\pi' = closure(\pi)$. Luckily, the loop must terminate since there are only a finite number of LR(0) items that step (b) could add to π'.