CS 434 Meeting 19 — 4/20/06

Garbage Collection (cont.)

1. Mark-scan collectors don't work very well when objects of a wide variety of sizes need to be allocated.

2. An alternative approach to garbage collection is based on copying. To implemnent this approach we divide the heap into two areas and switch back and forth between them by garbage collecting when the active area fills up. This seems wasteful (we only use half the memory available at any time). It has several advantages, including that the work performed depends only on the amount of useful data around, not on the amount of garbage.

3. The pseudo-code for this version of "findthings" (which is better called "movethings") looks like:

```
movethings( heapobject ** root )
{
    if ( *root != NULL) {
        if (  !(*root)->alreadymoved ) {
                  move the record **root to new heap
                  (*root)->forwardingaddress = newposition
                  (*root)->alreadymoved = TRUE
                  for all ‘‘child’’ren of copied root do
                              movethings( & child )
         }
         *root = pointer to new copy of record
      }
}
```

4. So, to make this all work, we have to plan the layout of data in memory in a way that will let us:

   - Find all the roots.
   - Find all the pointers within a record once we find the record in the heap.
   - Avoid copying things twice.
   - Know where each object has been moved to
   - Update all pointers during copying.

5. The simplest approach to most of these requirement is to "tag" each object and/or pointer with a "descriptor" (sometimes one bit is enough) that will tell the garbage collector what it is.

   - To find all the pointers in a record, we at least need to know how big the record is. So, the first "descriptor" we might use is a length field at the start of each record in the heap.
   - Because a record may contain both pointers and non-pointer values, we either need a descriptor for the whole record or descriptors for each element.
     - This can be done by multiplying all integer values by 2 (so their low order bits will be 0) and making sure that all records are allocated on odd addresses (so that pointers will have 1's in their low order bits).
   - We also need some way to tag a record that has already been moved. Since its contents are not important after it has been moved, there are plenty of ways to handle this (set its length to 0?).

6. To update pointers that point to objects that have already been moved, we need to store the new address of the object somewhere in the space that held the original object (the word after the length?). This is called a *forwarding pointer*.

7. SO, all that's left is to find all the roots. Most of them (all of them if we do it right) are on the stack of activation records. To find them, we write a loop to look at all the words in the stack and call movethings with any value that we are sure is a heap pointer.

8. The stack contains parameters and locals (which are all tagged pointers and integers) and return points and return frame pointers (stored by JSR and LINK instructions).

   We can easily tell if the "tagged" values are pointers (or not). The return information may look like pointers (i.e. may be odd), but they

won't point to the heap, so a simple test against the heap boundaries will detect them.

9. To make sure that roots aren't hiding in registers, we have to make sure our code stores all registers that hold pointers onto the stack before any call (including to the garbage collector).

10. Finally, if we really are garbage collecting because we ran out of memory we don't want to run a recursive algorithm that might require lots of stack space. This can be fixed by switching from a depth first traversal of the findable data in the heap to a breadth first version using the new copy of the heap as our worklist/queue.

    - The tail of the contents of the new heap will be viewed as a queue with:
      - The head pointer equal to the first moved structure whose internal pointers have not been processed.
      - The tail pointer equal to the last structure moved.

SLR(1) parsing

1. Consider the grammar

   $$< \text{S} > \ \rightarrow \ \text{a} < \text{S} > \text{b} < \text{S} > \mid \epsilon$$

2. If we build the LR(0) machine for this grammar, we discover that it is not an LR(0) grammar because several states contain shift/reduce conflicts.

   - The initial state is composed of the items:

     $[< \text{S'} > \ \rightarrow \ . < \text{S} > \$]$
     $[< \text{S} > \ \rightarrow \ . \ \text{a} < \text{S} > \text{b} < \text{S} >]$
     $[< \text{S} > \ \rightarrow \ . \ \epsilon]$

   - Starting from the initial state on input "a" we reach the the state:

     $[< \text{S} > \ \rightarrow \ \text{a} \ . < \text{S} > \text{b} < \text{S} >]$
     $[< \text{S} > \ \rightarrow \ . \ \text{a} < \text{S} > \text{b} < \text{S} >]$
     $[< \text{S} > \ \rightarrow \ \epsilon \ .]$

   We can use this machine anyway, if we are willing to look ahead a bit.

   - In all the sentential forms you can generate from the grammar an "a" will never directly follows and S.
   - As a result, in either of the states shown, choosing to reduce when the next input is an "a" would definitely lead to a dead end.

3. In general, suppose that we find that after reading some prefix ω_1 of an input $\omega_1 x \omega_2$ we end up in a state that contains a reduce item $[N \rightarrow \beta .]$ which conflicts with some other item.

   - If we decide to reduce using the production in this item, we are basically assuming that

     $$S \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N x \omega_2 \underset{\text{rm}}{\Longrightarrow} \alpha \beta x \omega_2 \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \omega_1 x \omega_2$$

   - That is, we are assuming that there is some sentential form in which an x can follow an N.

4. We can give the set of symbols that might appear after a non-terminal a name:

   $$\text{Follow}(N) = \{x \in V_t \mid A \overset{*}{\Longrightarrow} \alpha N x \beta\} \cup \{\epsilon \text{ if } S \overset{*}{\Longrightarrow} \alpha N\}$$

5. Given the notion of the "follow" set, we can illustrate the use of look-ahead in LR parsing, by considering the simplest form of look-ahead LR parsing — SLR(1) parsing (that's S for simple).

6. For the grammar considered above, a is not in Follow(S). So, we should never reduce using the production $< \text{S} > \ \rightarrow \ \epsilon$ if the next "input" symbol is a. Therefore, the three states with LR(0) conflicts really don't have conflicts at all.

7. In general, we will say that a set of LR(0) items contains an SLR(1) conflict if either:

   (a) It contains two reduce items $[N \rightarrow \beta_1 .]$ and $[M \rightarrow \beta_2 .]$ such that the intersection of Follow(N) and Follow(M) is non-empty, or

(b) It contains a reduce item $[N \rightarrow \beta_1 .]$ and a shift item $[M \rightarrow \beta_2 . x \beta_2]$ such that $x \in \text{Follow}(N)$.

8. If the LR(0) machine for a grammar G contains no states with SLR(1) conflicts we say that G is an SLR(1) grammar.

9. An SLR(1) parser for an SLR(1) grammar G behaves as follows in state π when the next input symbol is "x":

   - reduce using production $N \rightarrow \beta$ if $[\ N \rightarrow \beta\ .\ ]\ \in \pi$, and $x \in \text{Follow}(N)$.
   - shift in next input if π contains one or more items of the form $[\ N \rightarrow \alpha . x \beta\ ]$.
   - error otherwise.