CS 434 Meeting 18 — 4/18/06

Anouncements

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

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 method 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. You can divide the available registers into two set — one to be handled callee-saved, the other set handled caller-saved.

3. 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.

4. 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. 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. 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'.

2. We can prove that each item in closure(π) is valid for γ using an argument based on showing that the claim that execution of the loop body preserves the "invariant" that all items in π' are valid for γ.

   basis? All items in π are assumed valid for γ, so the invariant will be true when the loop begins to execute..

   induction? Assume that all item in π' are valid for γ and let $[M \rightarrow . \beta_3]$ be the item added during an execution of the loop body. The addition of this item implies that some item of the form $[N \rightarrow \beta_1 . M \beta_2]$ must have already been in π'. By the assumption that the invariant holds at the beginning of each execution of the loop body, this item must be valid for γ. Accordingly, there must be some derivation:

   $$S' \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N \omega \underset{\text{rm}}{\Longrightarrow} \alpha \beta_1 M \beta_2 \omega$$

   with $\gamma = \alpha \beta_1$. Assuming the grammar has no useless non-terminals, it must be possible to derive some string of terminals, ω' from β_2. Thus, there is a derivation:

   $$S' \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N \omega \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha \beta_1 M \omega' \omega \underset{\text{rm}}{\Longrightarrow} \alpha \beta_1 \beta_3 \omega' \omega$$

   The existence of this derivation implies that the item $[M \rightarrow . \beta_3]$ is valid for γ. Therefore, adding this item to π' preserves the invariant.

3. Now, assuming Lemma 1, we can prove Lemma 2 by induction on the length of γ. The basis step is so simple that we will look at the induction step first:

   induction Assume that we know that the theorem holds for all strings of length n and consider some string γx such that γ is of length n and x is a single symbol.

   Suppose that $[N \rightarrow \beta_1 x . \beta_2]$ is an item in $\Delta(\pi_0, \gamma x)$. The fact that this item is in this set implies that the item $[N \rightarrow \beta_1 . x \beta_2]$ must be in $\Delta(\pi_0, \gamma)$. This, together with our inductive assumption implies that $[N \rightarrow \beta_1 . x \beta_2]$ must be valid for γ. Therefore, there exists a derivation:

   $$S' \underset{\text{rm}}{\overset{*}{\Longrightarrow}} \alpha N \omega \underset{\text{rm}}{\Longrightarrow} \alpha \beta_1 x \beta_2 \omega$$

   with $\alpha \beta_1 = \gamma$. This, however implies that $[N \rightarrow \beta_1 x . \beta_2]$ is indeed valid for γx.

   basis Similarly, when we consider strings of length 0, the only kernel item in $\Delta(\pi_0, \epsilon)$ is $[S' \rightarrow . S]$. The derivation $S' \underset{\text{rm}}{\overset{*}{\Longrightarrow}} S' \underset{\text{rm}}{\Longrightarrow} S$ shows that this item is valid for ϵ.

Garbage Collection

1. In writing the various phases of the compiler project, you will find yourself spending a good bit of time making sure you "free" memory areas that were no longer in use:

   - You should have already realized that you have to think a bit about when to free operand descriptors.
     – This may be interesting because some operand descriptors (those for A5 and A6 for example) are referenced from many places and can't be freed when one reference to them is destroyed.
     – In an optimizing compiler that recognized common subexpressions, any operand descriptor might have multiple references making the decision when to free quite complicated.

2. If we were writing compilers in Java, this just would not be an issue.

   - While Java has a "new" operation that allocates storage much like C's "malloc", there is no "free".
   - The Java compiler and run-time system automatically find and recover dynamically allocated values that are no longer accessible.

3. This approach to dynamic memory management has been standard in applicative languages (LISP, ML, Miranda, Haskell, etc.) for years (forever?). It makes programming simpler, safer, but possibly less efficient.

4. Garbage collection is the process of automatically reclaiming dynamically allocated memory areas.

5. The area in which dynamically allocated memory resides is called the "heap".

6. Before we worry about how to automatically deallocate space from a heap, lets talk about some simple ways to organize a heap and allocate space as needed.

   Free lists Have allocated and available areas of memory intermixed with the unallocated units linked together on a "free list".

   Free/used regions Maintain a register/variable pointing to the moving boundary between the allocated and unallocated regions of memory. Note: In this scheme the "allocated" region will consist of intermixed used and un-used (i.e. garbage) words.

7. The "free list" idea works best if all the units of memory allocated are of the same size. This was true in the first significant language whose implementation relied heavily on garbage collection, LISP.

8. A dynamically allocated data structure in the heap can be safely disposed and its memory space reused only if it can no longer be accessed. This is true when no variable or method parameter (or hidden temporary) or other accessible dynamically allocated object refers to it. Thus, to find the garbage we must find the stuff that can't be found!

   - The trick is that we can find out what can't be found by finding out what can be found.
   - The variables, method parameters and hidden temporaries are referred to as "roots". We must search for findable objects starting at each root.

     So, the "main program" of a garbage collector might look like:

```
for each record in heap do
          record->alreadyfound = false
for each root do findThings( root )
```

   - Once an object is found, we must recursively "find" all the objects it points to. So, the "findThings" pseudo-code looks something like a "careful" tree traversal:

```
findThings( root )
  {
    if ( root != NULL  &&  ! root->alreadyfound )
       then {  root->alreadyfound = TRUE;
               for all ``child''ren of root do
                  findThings(  child )
            }
  }
```

9. The next question is what to do when we find things. Two common answers:

   Mark-scan Just set a bit to mark the items as "found" as suggested in the code above. Then later make a pass through the whole heap and identify anything that isn't marked as "unfindable" stuff that can be reused.

   Copying Move everything found to a new, unused heap space (and update all pointers to refer to the new copies). When this is all done, the new heap will have used and free space but no garbage.

10. Mark-scan garbage collection was developed for the implementation of LISP where all objects were of the same size. So, as the heap was scanned for free areas they could just be added to the free list.

11. 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.

12. 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).
      - Addition and subtraction can ignore the factor of 2. After each multiplication, you will need to divide the result by 2....
      - On the 34000 this would be costly (wastes one out of 16 bits and words on the heap), but on real machines it works very well (1 out of 32 bits isn't bad, and on many machines all records have to be stored on odd or even *byte* addresses anyway).
    - 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?).