Assignment 3
Pointers and Program Analysis

15-411: Compiler Design
Rob Arnold (`rdarnold@andrew`) and Eugene Marinelli (`emarinel@andrew`)

Due: Thursday, October 9, 2008 (1:30 pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own.

You may hand in a handwritten solution or a printout of a typeset solution at the beginning of lecture on Thursday, October 9. Please read the late policy for written assignments on the course web page. If you decide not to typeset your answers, make sure the text and pictures are legible and clear.

Problem 1

[15 points]

As discussed in class, constant propagation can be performed by first computing reaching definitions. However, it is also possible to perform constant propagation directly as a forward dataflow analysis. In this problem, we will explore a sort of constant propagation for pointer values, keeping track of some information about the value that a pointer variable x may have.

We write $\mathsf{val}(l, x, V)$ if the value of the pointer variable x at line l in the programs may be V, where V is either NULL or !NULL. This means we don't track the precise value (which would be impossible), but only a sound approximation of the possible values of x at line l. As in Lectures 4 and 5, we mean the value of x *before* the statement at line l is executed.

The val predicate is seeded at assignments according to the following rules.

$$\frac{l : x \leftarrow \mathsf{NULL} \quad \mathsf{succ}(l, l')}{\mathsf{val}(l', x, \mathsf{NULL})} \qquad \frac{l : x \leftarrow e \quad e \neq \mathsf{NULL} \quad \mathsf{succ}(l, l')}{\mathsf{val}(l', x, \mathsf{NULL}) \quad \mathsf{val}(l', x, !\mathsf{NULL})} \qquad \frac{l : x \leftarrow \mathsf{alloc}(k) \quad \mathsf{succ}(l, l')}{\mathsf{val}(l', x, !\mathsf{NULL})}$$

In the middle rule we have two conclusions, because we cannot predict the value of e in this simple analysis and so x may be null or it may not be null. In the right rule, we exploit that if alloc returns it gives us a non-null pointer; failure to allocate would be signaled as an exception.

a. Complete the specification of the analysis by writing rules to propagate value information throughout the program. You may assume the language and predicates are as in notes to Lecture 5 on dataflow analysis.

b. If x is a function parameter, how do we define val of x at l_0, the beginning of the function body?

c. Suppose $\{V \mid \mathsf{val}(l, x, V)\} = \{\,\}$. What does this mean about a use of x at l?

We can use the approximate value information for optimization in the following ways:

- If $\{V \mid \mathsf{val}(l, x, V)\} = \{\mathsf{NULL}\}$ we can replace uses of x at line x by NULL, because NULL is the only value x can have at l.
- We can evaluate some pointer comparisons if both sides are either constant or known to be only null or non-null. For example, if $\{V \mid \mathsf{val}(l, x, V)\} = \{\mathsf{NULL}\}$ and $\{V \mid \mathsf{val}(l, y, V)\} = \{!\mathsf{NULL}\}$ then the expression $x == y$ can be replaced by 0.

Now consider the program

l_0 : $z \leftarrow \mathsf{NULL}$
l_1 : $y \leftarrow \mathsf{alloc}(4)$
l_2 : $\mathsf{if}\ (y == z)\ \mathsf{goto}\ l_4$
l_3 : $y \leftarrow \mathsf{NULL}$
l_4 : $b \leftarrow (y == z)$
l_5 : $\mathsf{return}\ b$

d. Compute the possible values V of y and z at each line.

e. Using these values, perform the constant propagation and constant folding optimizations described above.

Problem 2

[10 points]

In the previous problem, we are not fully exploiting our knowledge about the values of the variables because the conditional branch remains. More generally, for common tests such as

$$l : \mathsf{if}\ (x == \mathsf{NULL})\ \mathsf{goto}\ l'$$

we should be able to conclude that x is NULL at l' and x is !NULL at $l + 1$. An analysis that captures this phenomenon is called *flow sensitive*.

g. Extend and/or modify your set of rules from Problem 1 so that the analysis is now flow sensitive.

h. Compute the possible approximate values V of x in the following program with your analysis. It should be strong enough to show that the dereference of x cannot yield a null pointer exception. Assume g is an unknown function.

l_0 : $x \leftarrow g(3429)$
l_1 : $\mathsf{if}\ (x == \mathsf{NULL})\ \mathsf{goto}\ l_4$
l_2 : $y \leftarrow M[x]$
l_3 : $\mathsf{goto}\ l_5$
l_4 : $y \leftarrow 0$
l_5 : $\mathsf{return}\ y$

Problem 3

[10 points]

Return to the (flow-insensitive) analysis in Problem 1. The basic predicate here is $\mathsf{val}(l, x, V)$, which talks about the possible values for x at location l. If the program were written in static single-assignment (SSA) form, then each variable would be defined at a unique line of the program. Therefore, it is not necessary

to track both line l and variable x and we can instead write rules to reason about a two-place predicate $\mathsf{val}(x, V)$.

Rewrite the rules from your solution to Problem 1 to specify a $\mathsf{val}(x, V)$, assuming the program is in SSA form.

Hint: While there is some simplification you must now handle the case $l : x \leftarrow \phi(x_1, \ldots, x_n)$.

Problem 4

[10 points]

Assume that you are working with C. Consider the function `alloca(int nbytes)` which allocates dynamic memory on the stack.

The implementation of `alloca()` on x86-32 is easy, since it maintains both a stack pointer and base pointer in dedicated registers. Whenever a function returns to its caller, it automatically frees the memory allocated by `alloca()` with the standard function epilogue.

On other architectures, such as the PowerPC, there is no dedicated base pointer, and only a stack pointer. On x86-64 the base pointer is optional. On function entry, the stack pointer is decremented by a fixed amount (determined at compile-time), to hold local variables. Variables are referenced by offsets relative to the stack, and the function returns by incrementing the stack pointer the same amount it was decremented.

Clearly, this makes implementing `alloca()` more difficult. Changing the stack pointer means that the code can no longer access local variables correctly, since the amount of change to the stack pointer is not known at compile time.

Despite these problems, you want to still support `alloca()` on PowerPC mostly because your favorite version control software, Subversion, depends on it.

There are two ways to implement `alloca()` in this situation.

a. You can change your compiler to generate special code for functions in which `alloca()` is called. Describe in detail how to implement this option.

b. You can implement `alloca()` as a library function which approximates `alloca()` using `malloc()` and `free()`. Here, you may not rely on garbage collection. "Approximating" `alloca()` means that you should try to structure your solution so that the allocated space is freed as soon as you can safely do so from within the `alloca()` function. Describe in detail how to implement this option.

Problem 5

[15 points]

In Java, all allocations take place on a garbage collected heap. As you know, stack allocation can be much faster that a garbage-collected allocation. If the lifetime of an allocation can be determined by the compiler, then it can go on the stack rather than the heap. In Java, each object has synchronization methods so it is also useful to determine if an object escapes to another thread. This is called *escape analysis*.

a. Starting with the IL introduced in the dataflow analysis notes, add a new statement $x \leftarrow \mathsf{alloc}(y)$ which allocates a block of y bytes and stores a pointer to that block into x. Design formulas and rules for tracking which variables hold pointers to a stack-allocated block. For simplicity, you may pretend there is no pointer arithmetic allowed. We recommend that you use SSA form, as in Problem 3, but you are not required to do so.

b. Present formulas and rules for tracking if a given allocation escapes the current function. Remember that there are memory stores and your function may have parameters. Assume any pointer parameters have "escaped" since they are not local to the function's stack frame.

c. Write two functions with allocation in this IL. They should include a conditional. Write them such that some allocation(s) escape and others do not and illustrate your analysis by annotating lines with the information you can infer.