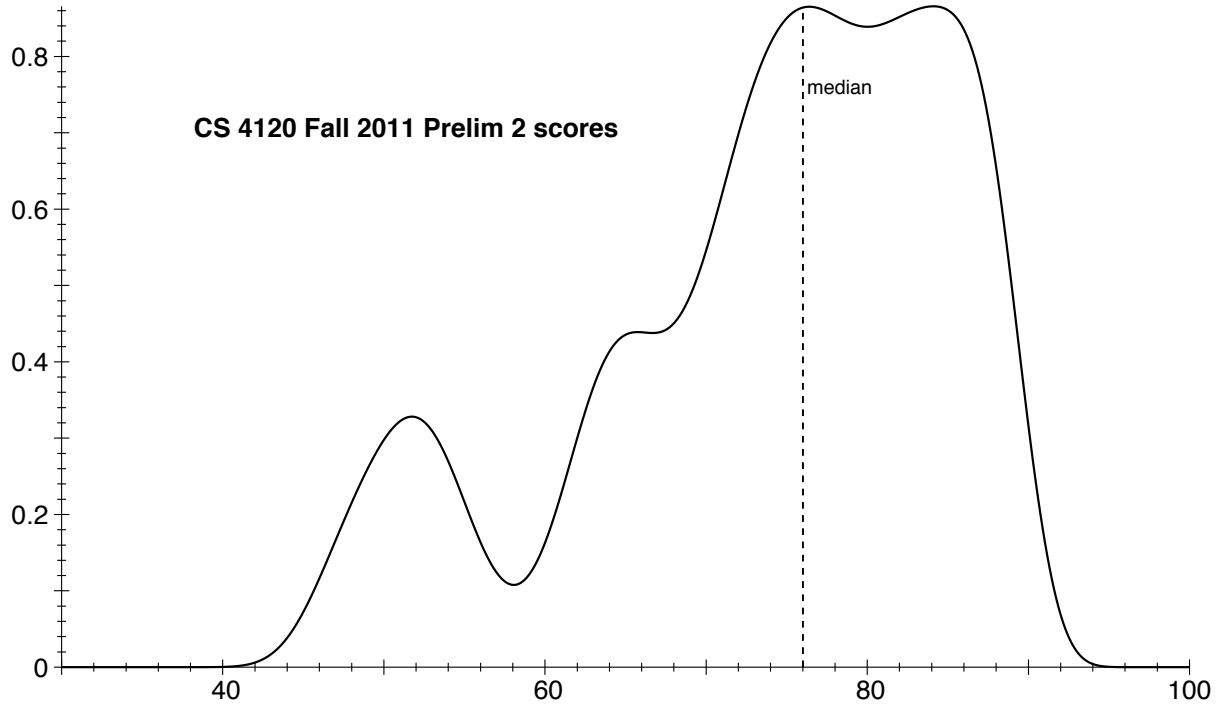
Preliminary Examination II
Computer Science 4120, Cornell University 1 December 2011

Solutions

In the following smoothed histogram of exam scores, the vertical axis measures the number of students per unit score.



1. **True/False** [14 pts] (parts a–g)

   Two points per question. Only one point off for questions left blank.

   (a) In a language such as Pascal with nested lexically scoped functions that cannot be returned as results or stored into data structures, activation records can always be allocated safely on the stack.

   true

   (b) An LALR parser generator will report a shift–shift conflict on some input grammars.

   false

   (c) To support multiple inheritance, it is necessary to leave holes both in objects and in dispatch tables in order to avoid field and method offset conflicts.

   false – at most dispatch tables need to be sparse, and with the C++ approach, even these can be dense, though there may be multiple dispatch table pointers in the object.

   (d) Dispatch using a decision tree works relatively poorly in the case where there are many possible method implementations and they are all roughly equally likely.

true

(e) In a k-CFA analysis, the same function body may be analyzed a number of times that is exponential in k.

true

(f) Consider an ordering of the integers in which $x \sqsubseteq y$ if and only if $x \leq y$ and x and y have the same parity (are equal modulo 2). The integers with this ordering are a lattice.

false

(g) Reordering basic blocks at the IR level may require inserting new jumps.

true

2. **Must-read dataflow analysis** [23 pts] (parts a–f)

Baker's algorithm for copying garbage collection involves running the mutator in to-space while the garbage collector is working. It relies on a *read barrier*.

(a) [2 pts] Explain in 2–3 sentences why the read barrier is necessary in Baker's algorithm.

Answer:

It is needed for safety. The mutator needs to avoid storing references to from-space objects into to-space objects, because they will become dangling references once the garbage collector finishes. The read barrier in the mutator detects attempts to read from-space pointers and triggers copying of the referenced object to to-space if necessary.

Read barriers are expensive. Let us make two assumptions: first, that the run-time system prevents the garbage collector from starting during the execution of the function. And second, that when one field in an object has been detected as a from-space pointer, all references in the object are fixed to point to to-space. With these assumptions, once an object reference has been used to read a field, the object can be assumed to pass the read-barrier check. The obvious optimization is to remove the read barrier from further reads from such an object.

Let's design a dataflow analysis to figure out where read barriers are not needed. We want to compute for each CFG node a set of variables that are guaranteed to point to an object that has already been read from.

(b) [1 pt] Is this a forward or backward analysis?

Answer:

Forward.

(c) [4 pts] What is the meet operation? And the top element of the lattice of dataflow values? Justify briefly.

Answer:

Set intersection, because an object is definitely read at entry to a node only if it is definitely read on all *incoming edges. Therefore the top element is the universal set of all variables, just like other forward must-analyses such as available expressions.*

(d) [4 pts] Define the dataflow equations for $in(n)$ and $out(n)$ in terms of $gen(n)$ and $kill(n)$ functions.

Answer:

$$in(n) = \bigcap_{n' \prec n} out(n')$$

$$out(n) = in(n) \cup gen(n) - kill(n)$$

(e) [8 pts] Complete the following table to obtain appropriate definitions of the $gen(n)$ and $kill(n)$ functions you used in part 2(d). Note that we are using a slightly higher-level CFG in which accesses to fields of objects are made explicit as expressions $x.f$, which you can think of as equivalent to $[x + \mathit{offset}_f]$ in our standard notation for CFG nodes. For simplicity we only allow simple expressions on the right-hand side of assignments.

Answer:

n	$gen(n)$	$kill(n)$
$x = y.f$	$\{y\}$	$\{x\}$
$x.f = y$	$\emptyset$	$\emptyset$
$x = f(y)$	$\emptyset$	$\{x\}$
START	$\emptyset$	$\top$
return x	$\emptyset$	$\emptyset$
if x	$\emptyset$	$\emptyset$

(f) [4 pts] To make the analysis more effective, we might want to fold in some amount of copy propagation: if y is known to have been read already, then x is known to have been read after an assignment $x = y$. Show how to define the transfer function for this kind of node and identify any other changes that need to be made in the analysis you have defined.

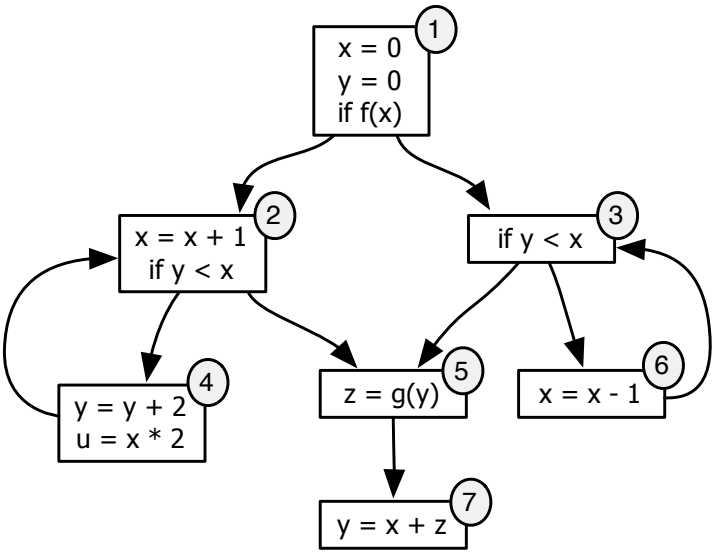
Answer:

We want to change the transfer function for $x = y$ so that it can generate the element x if y is in the must-read set. The function $gen(n)$ becomes a function $gen(n, \ell)$ and similarly with $kill(n)$. Then we modify the following definitions:

$$\begin{aligned} out(n) &= in(n) \cup gen(n, in(n)) - kill(n, in(n)) \\ gen(x = y, \ell) &= \{x\} && \text{if } y \in \ell \\ kill(x = y, \ell) &= \emptyset && \text{if } y \in \ell \\ kill(x = y, \ell) &= \{x\} && \text{if } y \notin \ell \end{aligned}$$

3. **Control flow analysis and optimization** [29 pts] (parts a–f)

This problem uses the following CFG, in which each node is given a unique number as an identifier. Note that `f`, `g`, and `h` are names of functions.



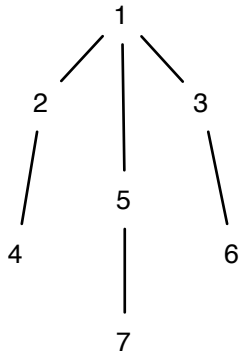
(a) [2 pts]
The nodes are basic blocks. Which nodes, if any, can still be combined to form a basic block?

Answer:

Combining 5 and 7 yields a larger basic block.

(b) [5 pts] Using the original CFG, draw a Hasse diagram of the dominator relation. Use the numeric identifiers to represent the nodes in the diagram.

Answer:



(c) [4 pts] Circle each loop in the control flow graph.

Answer:

One loop is nodes 2 and 4.
The other is nodes 3 and 6.

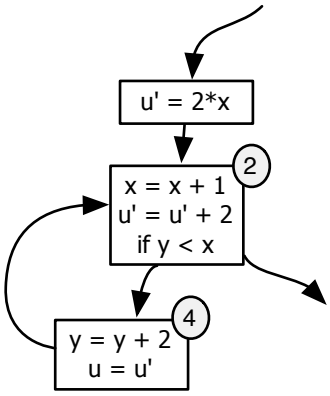
(d) [5 pts] What are the induction variables in each loop? For each induction variable, classify it as basic or derived, and indicate whether it is linear.

Answer:

In loop {2,4}, there are linear basic induction variables `x` *and* `y`*, and linear derived induction variable* `u` $= \langle \mathtt{x}, 2, 0 \rangle$.
In loop {3,6}, there is only the linear basic induction variable `x`.

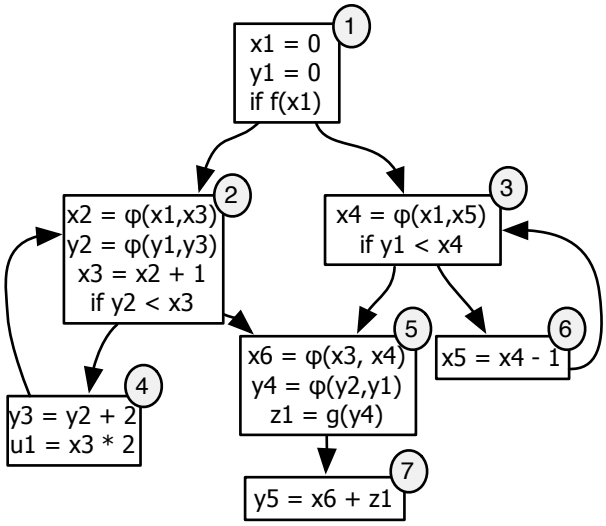
(e) [6 pts]
Show how to do a strength reduction optimization based on the induction variables in this example. Redraw only the part of the CFG that is affected by the optimization.

Answer:



(f) [7 pts]
Convert the original CFG, without the various modifications you have been making, to SSA form. Use the ϕ function as little as possible. Give the variables in SSA form the same names as the original variables, but with appropriate integer subscripts. Draw the resulting CFG in full.

Answer:



4. **Escaping functions and objects** [34 pts] (parts a–e)

Suppose that we extend Xi with both objects (with simple single inheritance, as in OO Xi, or Java sans interfaces) and lexically scoped nested functions (similar to ML). Consider the following code which uses both objects and functions:

```
f(y: int) {
  z: int = ...
  w: int = ...
  g(x:int):int { return z + y }
  h(x:int):int { return w - x }
  Cell a = new Cell // allocation without initializing fields
  a.gg = g
  return a.doit() + h(y)
}
class Cell {
  doit():int { ... do something complex here ... } // method
  gg: int->int
}
```

(a) [7 pts]

Assuming that we are not able to analyze the method `doit` and must treat its behavior conservatively, which of the nested functions in this example may escape, and which variables are escaping variables? Explain briefly.

Answer:

`g` *escapes because it is assigned into a field of* `a`*, which in turn may escape via the call to* `doit`*, where it is an (implicit) argument to the call.*

`h` *does not escape because it is never stored anywhere or returned from the function.*

Therefore the variables `y` *and* `z` *are escaping variables; all other variables are non-escaping, including* `g` *itself. Many people were confused about* `g` *and* `a`*, because the values they refer to may escape, but the* variables `g` *and* `a` *do not escape because they are not referred to by any escaping function.*

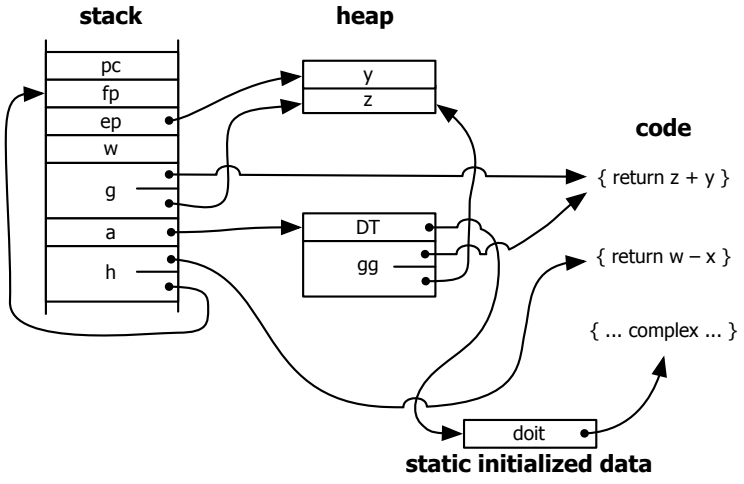
(b) [12 pts]

Draw the layout of the stack and heap during a call to the function `f`, just at the point where the `return` statement is about to start executing. Assume that all variables are allocated on the stack (if this can be done safely based on your answer to the previous part) rather than in registers.

Also include in your diagram the layout of the object `a`, not omitting any dispatch information it needs.

Label each part of the diagram to indicate whether it represents part of the stack, the heap, static initialized data, or code.

Answer:



Let's build an analysis to determine which objects escape. For simplicity we consider only the following syntax for CFG nodes:

$x = y$	
$x =$ `new` C /*obj*$_i$	(allocating an object of class C with abstract object identity *obj*$_i$)
$x =$ `function`$(\vec{x})\{\dots\}$ /*fun*$_j$	(creating a new nested function with abstract object identity *fun*$_j$)
$x.f = y$	(a write to memory at offset for field f)
$x = f(y)$	(call to function f)
$x = y.f$	(a read from memory at offset for field f)
if x	
return x	

The new node type $x =$ `function`$(\dots)\dots$ is used to represent statements like the definitions of `g` and `h` in the example code, e.g.:

```
g = function(x:int) { return z + y } / fun1
h = function(x:int) { return w - x } / fun2
```

We are not doing an interprocedural analysis, so the body of the nested function does not matter for this analysis.

We assume that we already have done an inclusion-based pointer analysis that tells us for any given variable v which abstract objects of the form *obj*$_i$ or *fun*$_j$ (for some index i or j) that it may point to at the entry to node n. This set of abstract objects is denoted $Ptr_n(v)$. Recall that a variable v may be either an actual program variable x or a field of an abstract object, $o.f$.

Our goal is to find a set of objects o that may escape so that we can then identify escaping variables. We define a single variable E to compute the set of escaping objects, with a dataflow equation of the form $E = E \cup gen(n, E)$ for each node n.

(c) [1 pt] Is this analysis flow-sensitive or flow-insensitive? Explain why briefly.

Answer:

Flow-insensitive, because we don't need to know the escaping objects at each program point.

(d) [2 pts] Given these dataflow equations, what is the top element of the lattice of dataflow values?

Answer:

The empty set.

(e) [12 pts] Complete the following table to define $gen(n, E)$. (*Hint*: the case for $x.f = y$ is the trickiest and worth getting right.)

Answer:

n	$gen(n, E)$
$x = y$	$\emptyset$
$x = \texttt{new}\ C\ /obj_i$	$\emptyset$
$x = \texttt{function}(\vec{x})\{\ldots\}\ /fun_j$	$\emptyset$
$x.f = y$	$Ptr_n(y)$ *if* $Ptr_n(x) \cap E \neq \emptyset$, *otherwise* $\emptyset$
$x = f(y)$	$Ptr_n(y)$
$x = y.f$	$\emptyset$
if x	$\emptyset$
return x	$Ptr_n(x)$
START	$\bigcup\{Ptr_n(y) \mid y$ *is a pointer argument*$\}$

Note that in the **START** *case, all input arguments must be treated as escaping because they may be used in the caller after returning.*