

CIS 341: Compilers

Lecture 28

The Plan

- Today:
 - Liveness analysis continued
 - Other dataflow analyses

- Project 6: Simple optimizations
 - Due: Monday, November 17th at 11:59 pm

Dataflow Analysis

- *Idea*: compute liveness information for all variables simultaneously.
- Approach (three steps):
- Step 1: define sets of interesting information about each node
- Step 2: define *equations* that must be satisfied by any liveness determination.
 - Equations based on “obvious” constraints among the sets from step 1:
 - $in[n] \supseteq use[n]$
 - $in[n] \supseteq out[n] - def[n]$
 - $out[n] \supseteq in[n']$ if $n' \in succ[n]$
- Step 3: Solve the equations by iteratively converging on a solution.
 - Start with a “rough” approximation to the answer
 - Refine the answer at each iteration
 - Keep going until no more refinement is possible: a *fixpoint* has been reached

Liveness Analysis Algorithm

for all n , $in[n] := \emptyset$, $out[n] := \emptyset$

repeat until no change

for all n

$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

end

end

- Finds a *fixpoint* of the **in** and **out** equations.
- Claim: The $in[n]$ and $out[n]$ sets are *monotonically* increasing.
 - i.e. let $in[n]_i$ be the set at iteration i , then $in[n]_i \subseteq in[n]_{i+1}$
 - and similarly: $out[n]_i \subseteq out[n]_{i+1}$

Iterating the Equations

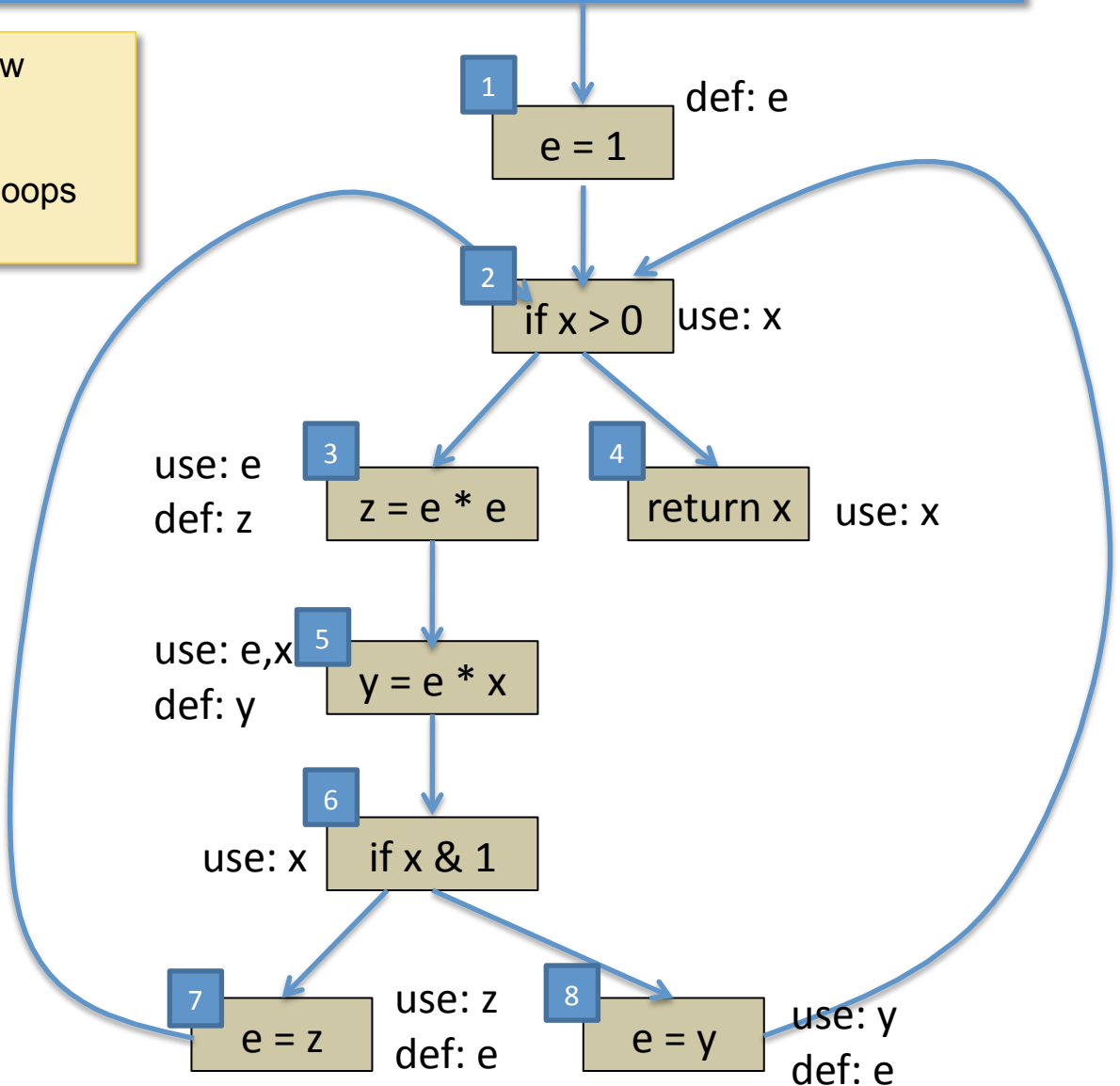
- 2: in = {x}
- 3: in = {e}
- 4: in = {x}
- 5: in = {e,x}

- 6: in = {x}
- 7: out = {x}, in = {x,z}
- 8: out = {x}, in = {x,y}

- 1: out = {x}, in = {x}
- 2: out = {e,x}, in = {e,x}
- 3: out = {e,x}, in = {e,x}
- 5: out = {x}, in = {e,x}
- 6: out = {x,y,z}, in = {x,y,z}
- 7: out = {e,x}, in = {x,z}
- 8: out = {e,x}, in = {x,y}

- 1: out = {e,x}, in = {x}
- 5: out = {x,y,z}, in = {e,x,z}
- 3: out = {e,x,z}, in = {e,x}
- done!

Steps at left show which sets have changed. Brackets group loops over "for all n".



Improving the Algorithm

- Can we do better?
- Observe: the only way information propagates from one node to another is using: $out[n] := \bigcup_{n' \in succ[n]} in[n']$
 - This is the only rule that involves more than one node
- If a node's successors haven't changed, then the node itself won't change.
- Idea for an improved version of the algorithm:
 - Keep track of which node's successors have changed

A Worklist Algorithm

- Use a FIFO queue of nodes that might need to be updated.

for all n , $in[n] := \emptyset$, $out[n] := \emptyset$

w = new queue with all nodes

repeat until w is empty

let $n = w.pop()$

// pull a node off the queue

old_in = in[n]

// remember old in[n]

$out[n] := \bigcup_{n' \in succ[n]} in[n']$

$in[n] := use[n] \cup (out[n] - def[n])$

if (old_in \neq in[n]),

// if in[n] has changed

for all m in pred[n], $w.push(m)$ *// add to worklist*

end

Generalizing Dataflow Analyses

- The kind of iterative constraint solving used for liveness analysis applies to other kinds of analyses as well.
 - Reaching definitions analysis
 - Available expressions analysis
- To see these as an instance of the same kind of algorithm, it is useful to work over a canonical intermediate instruction representation called *quadruples*
 - Allows easy definition of $def[n]$ and $use[n]$
 - Slightly more abstract than X86, closer to a RISC assembly
 - Easier for dataflow analyses
- These analyses follow the same 3-step approach as for liveness.

Quadruple Format

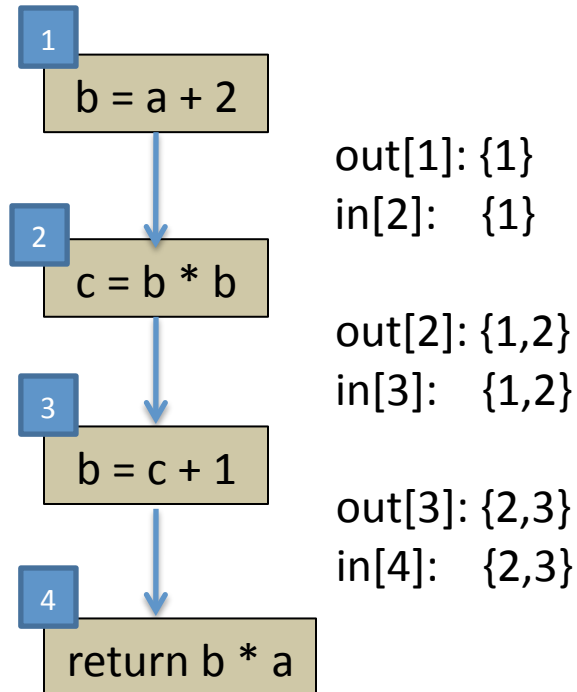
- A Quadruple sequence is just a control-flow graph (flowgraph) where each node is a quadruple:
- Quadruple forms n: def[n] use[n] description
a = b op c {a} {b,c} arithmetic
a = [b] {a} {b} load
[a] = b \emptyset {b} store
jump L \emptyset \emptyset jump
if a goto L1 else L2 \emptyset {a} branch
L: \emptyset \emptyset label
a = f(b₁,...,b_n) {a} {b₁,...,b_n} call w/return
f(b₁,...,b_n) \emptyset {b₁,...,b_n} call no return
return a \emptyset {a} return
- It's easy to convert the IR we've been using to this format...

Reaching Definition Analysis

- Question: what uses in a program does a given variable definition reach?
- This analysis is used for constant propagation & copy prop.
 - If only one definition reaches a particular use, can replace us by the definition (for constant propagation).
 - Copy propagation additionally requires that the copied value still has its same value – computed using an *available expressions* analysis (next)
- Input: Quadruple CFG
- Output: $in[n]$ (resp. $out[n]$) is the set of nodes defining some variable such that the definition may reach the beginning (resp. end) of node n

Example of Reaching Definitions

- Results of computing reaching definitions on this simple CFG:



Reaching Definitions Step 1

- Define the sets of interest for the analysis
- Let $\text{defs}[a]$ be the set of nodes that define the variable a
- Define $\text{gen}[n]$ and $\text{kill}[n]$ as follows:

Quadruple forms n :	$\text{gen}[n]$	$\text{kill}[n]$
$a = b \text{ op } c$	$\{n\}$	$\text{defs}[a] - \{n\}$
$a = [b]$	$\{n\}$	$\text{defs}[a] - \{n\}$
$[a] = b$	\emptyset	\emptyset
jump L	\emptyset	\emptyset
if a goto $L1$ else $L2$	\emptyset	\emptyset
L :	\emptyset	\emptyset
$a = f(b_1, \dots, b_n)$	$\{n\}$	$\text{defs}[a] - \{n\}$
$f(b_1, \dots, b_n)$	\emptyset	\emptyset
return a	\emptyset	\emptyset

Reaching Definitions Step 2

- Define the constraints that a reaching definitions solution must satisfy.
- $out[n] \supseteq gen[n]$
“The definitions that reach the end of a node at least include the definitions generated by the node”
- $in[n] \supseteq out[n']$ if n' is in $pred[n]$
“The definitions that reach the beginning of a node include those that reach the exit of *any* predecessor”
- $out[n] \cup kill[n] \supseteq in[n]$
“The definitions that come in to a node either reach the end of the node or are killed by it.”
 - Equivalently: $out[n] \supseteq in[n] - kill[n]$

Reaching Definitions Step 3

- Convert constraints to iterated update equations:
- $in[n] := \bigcup_{n' \in \text{pred}[n]} out[n']$
- $out[n] := \text{gen}[n] \cup (in[n] - \text{kill}[n])$
- Algorithm: initialize $in[n]$ and $out[n]$ to \emptyset
 - Iterate the update equations until a fixed point is reached
- The algorithm terminates because $in[n]$ and $out[n]$ increase only *monotonically*
 - At most to a maximum set that includes all variables in the program
- The algorithm is precise because it finds the *smallest* sets that satisfy the constraints.

Available Expressions

- Idea: want to perform common subexpression elimination:
 - $a = x + 1$ $a = x + 1$
 ... \rightarrow ...
 $b = x + 1$ $b = a$
- This transformation is safe if $x+1$ means computes the same value at both places (i.e. x hasn't been assigned).
 - “ $x+1$ ” is an *available expression*
- Dataflow values:
 - $in[n]$ = set of nodes whose values are available on entry to n
 - $out[n]$ = set of nodes whose values are available on exit of n

Available Expressions Step 1

- Define the sets of values
- Define $gen[n]$ and $kill[n]$ as follows:

Quadruple forms n:	$gen[n]$	$kill[n]$
$a = b \text{ op } c$	$\{n\} - kill[n]$	$uses[a]$
$a = [b]$	$\{n\} - kill[n]$	$uses[a]$
$[a] = b$	\emptyset	$uses[[x]]$ (for all x that may equal a)
jump L	\emptyset	\emptyset
if a goto L1 else L2	\emptyset	\emptyset
L:	\emptyset	\emptyset
$a = f(b_1, \dots, b_n)$	\emptyset	$uses[a] \cup uses[[x]]$ (for all x)
$f(b_1, \dots, b_n)$	\emptyset	$uses[[x]]$ (for all x)
return a	\emptyset	\emptyset

Note the need for “may alias” information...

Note that functions are assumed to be impure...

Available Expressions Step 2

- Define the constraints that an available expressions solution must satisfy.
- $out[n] \supseteq gen[n]$
“The expressions made available by n that reach the end of the node”
- $in[n] \subseteq out[n']$ if n' is in $pred[n]$
“The expressions available at the beginning of a node include those that reach the exit of *every* predecessor”
- $out[n] \cup kill[n] \supseteq in[n]$
“The expressions available on entry either reach the end of the node or are killed by it.”
 - Equivalently: $out[n] \supseteq in[n] - kill[n]$

Note similarities and differences with constraints for “reaching definitions”.

Available Expressions Step 3

- Convert constraints to iterated update equations:
- $in[n] := \bigcap_{n' \in pred[n]} out[n']$
- $out[n] := gen[n] \cup (in[n] - kill[n])$
- Algorithm: initialize $in[n]$ and $out[n]$ to {set of all nodes}
 - Iterate the update equations until a fixed point is reached
- The algorithm terminates because $in[n]$ and $out[n]$ decrease only *monotonically*
 - At most to a minimum of the empty set
- The algorithm is precise because it finds the *largest* sets that satisfy the constraints.