Generalizing Data-flow Analysis

Last Time

– Introduction to data-flow analysis

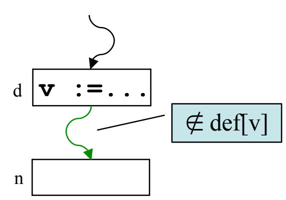
Today

– Other types of data-flow analysis
  – Reaching definitions, available expressions, reaching constants
– Abstracting data-flow analysis
  What's common among the different analyses?

Reaching Definitions

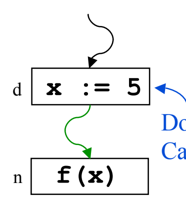
Definition

– A definition (statement) d of a variable **v** **reaches** node n if there is a path from d to n such that **v** is not redefined along that path



Uses of reaching definitions

– Build use/def chains
– Constant propagation
– Loop invariant code motion



Does this def of **x** reach n?
Can we replace n with **f(5)**?

```
1   a = . . .;
2   b = . . .;
3   for (. . .) {
4         x = a + b;
5         . . .
6   }
```

Reaching definitions of **a** and **b**

To determine whether it's legal to move statement 4 out of the loop, we need to ensure that there are no reaching definitions of **a** or **b** inside the loop

Computing Reaching Definitions

Assumption

- At most one definition per node
- We can refer to definitions by their node "number"

Gen[n]: Definitions that are generated by node n (at most one)

Kill[n]: Definitions that are killed by node n

Defining Gen and Kill for various statement types

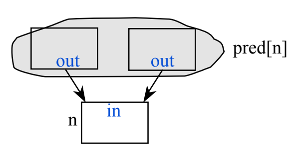
statement	Gen[s]	Kill[s]
s: t = b op c	{s}	def[t]
s: t = M[b]	{s}	def[t]
s: M[a] = b	{?}	{}
s: if a op b goto L	{}	{}

statement	Gen[s]	Kill[s]
s: goto L	{}	{}
s: L:	{}	{}
s: f(a,…)	{}	{}
s: t=f(a, …)	{s}	def[t]

Data-flow Equations for Reaching Definitions

The in set

- A definition reaches the beginning of a node if it reaches the end of **any** of the predecessors of that node



The out set

- A definition reaches the end of a node if (1) the node itself **generates** the definition **or** if (2) the definition reaches the beginning of the node and the node does **not kill** it

(1) (2)

$$in[n] = \bigcup_{p \in pred[n]} out[p]$$

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

Recall Liveness Analysis

Data-flow equations for liveness

$in[n] = \textbf{use}[n] \cup (out[n] - \textbf{def}[n])$

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

Liveness equations in terms of Gen and Kill

$$\left.\begin{array}{l} in[n] = \textbf{gen}[n] \cup (out[n] - \textbf{kill}[n]) \\ out[n] = \bigcup_{s \in succ[n]} in[s] \end{array}\right\}$$

A use of a variable generates liveness
A def of a variable kills liveness

Gen: New information that's added at a node

Kill: Old information that's removed at a node

Can define almost any data-flow analysis in terms of Gen and Kill

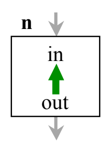
Direction of Flow

Backward data-flow analysis

– Information at a node is based on what happens later in the flow graph
i.e., in[] is defined in terms of out[]

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

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



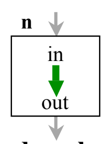
liveness

Forward data-flow analysis

– Information at a node is based on what happens earlier in the flow graph
i.e., out[] is defined in terms of in[]

$in[n] = \bigcup_{p \in pred[n]} out[p]$

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



reaching definitions

Some problems need both forward and backward analysis

– *e.g.,* Partial redundancy elimination (uncommon)

Data-flow Equations for Reaching Definitions

Symmetry between reaching definitions and liveness

– Swap in[] and out[] and swap the directions of the arcs

Reaching Definitions

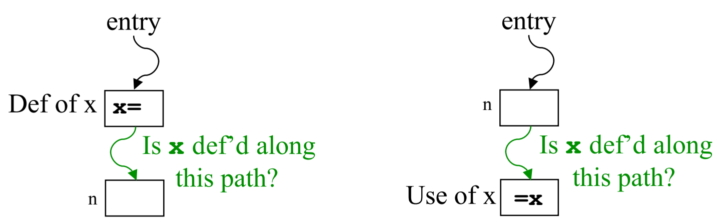
$$\text{in}[n] = \bigcup_{p \in \text{pred}[n]} \text{out}[s]$$

$$\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$$

Live Variables

$$\text{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{in}[s]$$

$$\text{in}[n] = \text{gen}[n] \cup (\text{out}[n] - \text{kill}[n])$$



A Better Formulation of Reaching Definitions

Problem

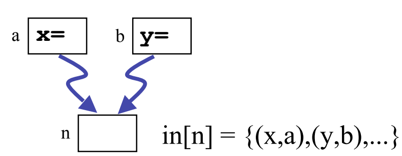
– Reaching definitions gives you a set of definitions (nodes)

– Doesn't tell you what variable is defined

– Expensive to find definitions of variable v

Solution

– Reformulate to include variable
e.g., Use a set of (var, def) pairs



Merging Flow Values

Live variables and reaching definitions

– Merge **flow values** via set union

Reaching Definitions

$$in[n] = \bigcup_{p \in pred[n]} out[s]$$

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

Live Variables

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

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

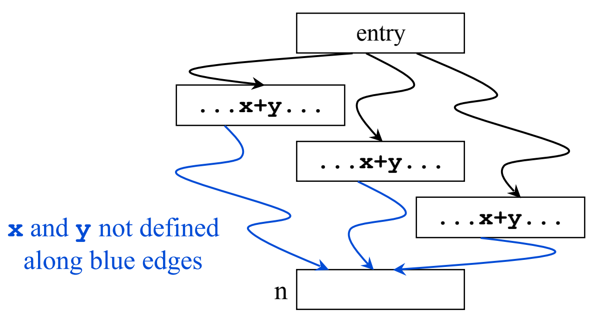
Why?

When might this be inappropriate?

Available Expressions

Definition

– An expression, **`x+y`**, is **available** at node n if every path from the entry node to n evaluates **`x+y`**, and there are no definitions of **`x`** or **`y`** after the last evaluation



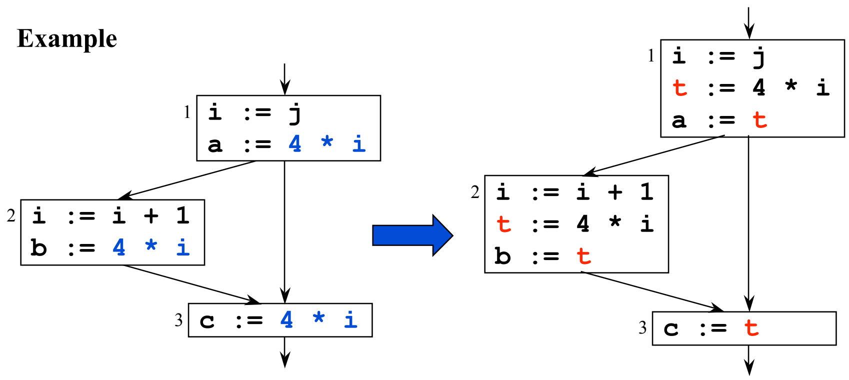
Available Expressions for CSE

How is this information useful?

Common Subexpression Elimination (CSE)

– If an expression is available at a point where it is evaluated, it need not be recomputed

Example



Must vs. May Information

May information

– Identifies possibilities

Must information

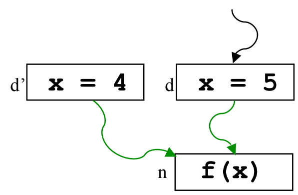
– Implies a guarantee

Liveness? Available expressions?

	May
safe	overly large set
desired information	small set
Gen	add everything that might be true
Kill	remove only facts that are guaranteed to be fals
merge	union
initial guess	empty set

Reaching Definitions: Must or May Analysis?

Consider constant propagation



We need to know if d' might reach node n

Defining Available Expressions Analysis

Must or may Information?

Direction?

Flow values?

Initial guess?

Kill?

Gen?

Merge?

Available Expressions (cont)

Data-Flow Equations

$\text{in}[n] = \bigcap_{p \in \text{pred}[n]} \text{out}[p]$

$\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$

Plug it in to our general DFA algorithm

for each node n
  in[n] = υ; out[n] = υ
repeat
  for each n
    $\text{in}'[n] = \text{in}[n]$
    $\text{out}'[n] = \text{out}[n]$
    $\text{in}[n] = \bigcap_{p \in \text{pred}[n]} \text{out}[p]$
    $\text{out}[n] = \text{gen}[n] \cup (\text{in}[n] - \text{kill}[n])$
until $\text{in}'[n]=\text{in}[n]$ and $\text{out}'[n]=\text{out}[n]$ for all n

Reaching Constants

Goal

– Compute value of each variable at each program point (if possible)

Flow values

Merge function

Data-flow equations

– Effect of node n `x = c`
  – $\text{kill}[n] = \{(x,d) | \forall d\}$
  – $\text{gen}[n] = \{(x,c)\}$
– Effect of node n `x = y + z`
  – $\text{kill}[n] = \{(x,c) | \forall c\}$
  – $\text{gen}[n] = \{(x,c) \mid c=valy+valz, (y, valy) \in \text{in}[n], (z, valz) \in \text{in}[n]\}$

Improving Iterative DFA Algorithm

How can we do better?

Problem

- If any node's in[] or out[] set changes after an iteration, our algorithm computes all of the equations again, even though many of the equations may not be affected by the change.

Solution

- A **work-list** algorithm keeps track of only those nodes whose out[] sets must be recalculated
- If node n is recomputed **and** its out[] set is found to change, all successors of n are added to the work list
- (For a backwards problem, substitute in[] for out[] and predecessor for successor.)

Work-List Algorithm for IDFA

Algorithm

for each node n
    in[n] = υ; out[n] = υ
worklist = {entry node}
while worklist not empty
    Remove some node n from worklist
    out' = out[n]
    in[n] = $\bigcap_{p \in pred[n]}$ out[p]
    out[n] = gen[n] $\cup$ (in[n] – kill[n])
    if out[n] $\neq$ out'
        for each s $\in$ succ[n]
            if s $\notin$ worklist, add s to worklist

Forward or Backward? May or Must?

Improving Iterative DFA Algorithm (cont)

Problem

– CFG is bloated when each statement is represented by a node

Solution

– Perform IDFA on CFG of basic blocks

Approach

(1) Build CFG of basic blocks

(2) Perform local data-flow analysis within each basic block to summarize Gen and Kill information for each node

(3) Perform global analysis on the smaller CFG

(4) Propagate global information inside of basic block: push information throughout the basic block from the entrance to the exit (or from the exit to the entrance if it's a backwards problem)

Example

Liveness

❷ ❶ ❸ ❹

Gen: {c,d}
Kill: {x,y}

```
x := c * d;
y := x / 2;
```

Gen: {b,x,y}
Kill: {a}

```
a := x + y;
x := a + b;
```

Reality Check!

Some definitions and uses are ambiguous

- We can't tell whether or what variable is involved
  e.g., `*p = x; /* what variable are we assigning?! */`
- Unambiguous assignments are called **strong updates**
- Ambiguous assignments are called **weak updates**

Solutions

- Be conservative
  - Sometimes we assume that everything is updated
    e.g., Defining `*p` (generating reaching definitions)
  - Sometimes we assume that nothing is updated
    e.g., Defining `*p` (killing reaching definitions)
- Compute a more precise answer:
  - Pointer analysis (more in a few weeks)

Concepts

Many data-flow analyses have the same character

Computed in the same way

Distinguished by

- Flow values (initial guess, type)
- May/must
- Direction
- Gen
- Kill
- Merge

Complication

- Ambiguous references (strong/weak updates)

Next Time

Lecture

– Lattice theoretic foundation for data-flow analysis