Lecture 3:

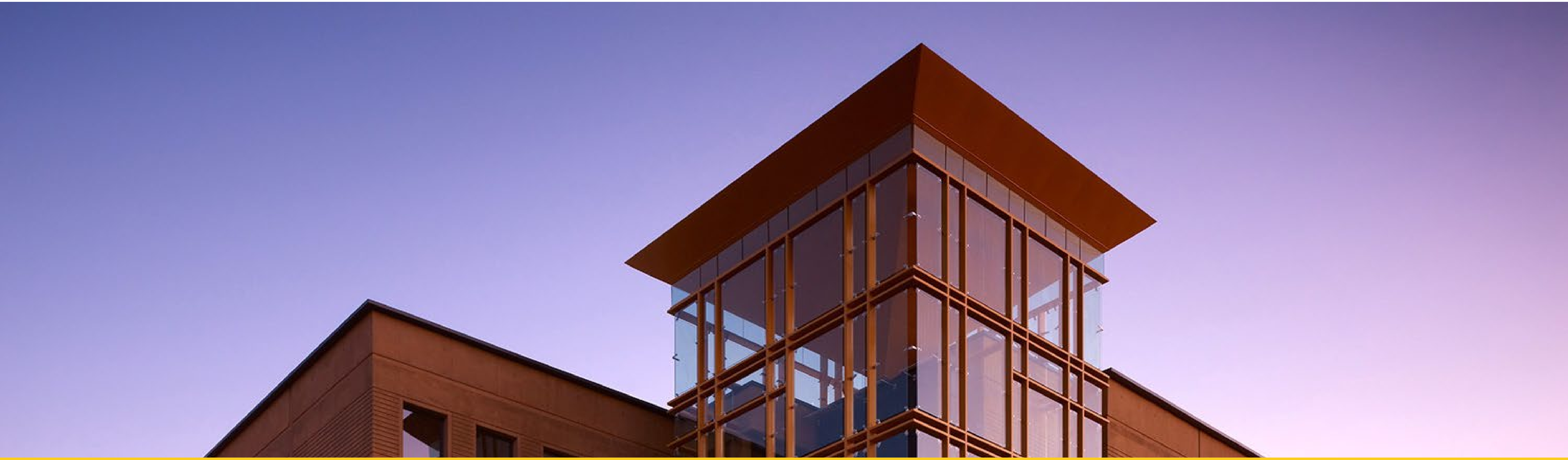
Language and Compiler Basics (II)

Sitao Huang

sitaoh@uci.edu

January 25, 2022

Slide courtesy of Prof. Vikram Adve, UIUC, CS 426: Compiler Construction



Logistics

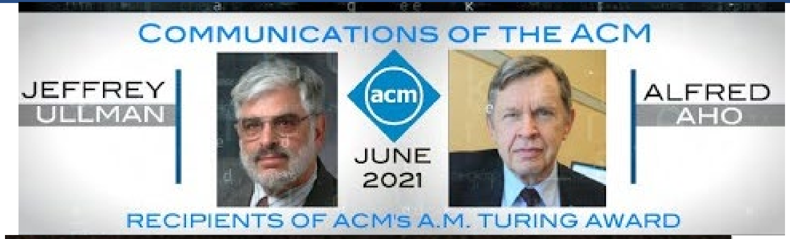
- **Homework 1** released, due: ***January 31***, 11:59 PM on Canvas
- Homework 2 expected to release later this week, due February 7

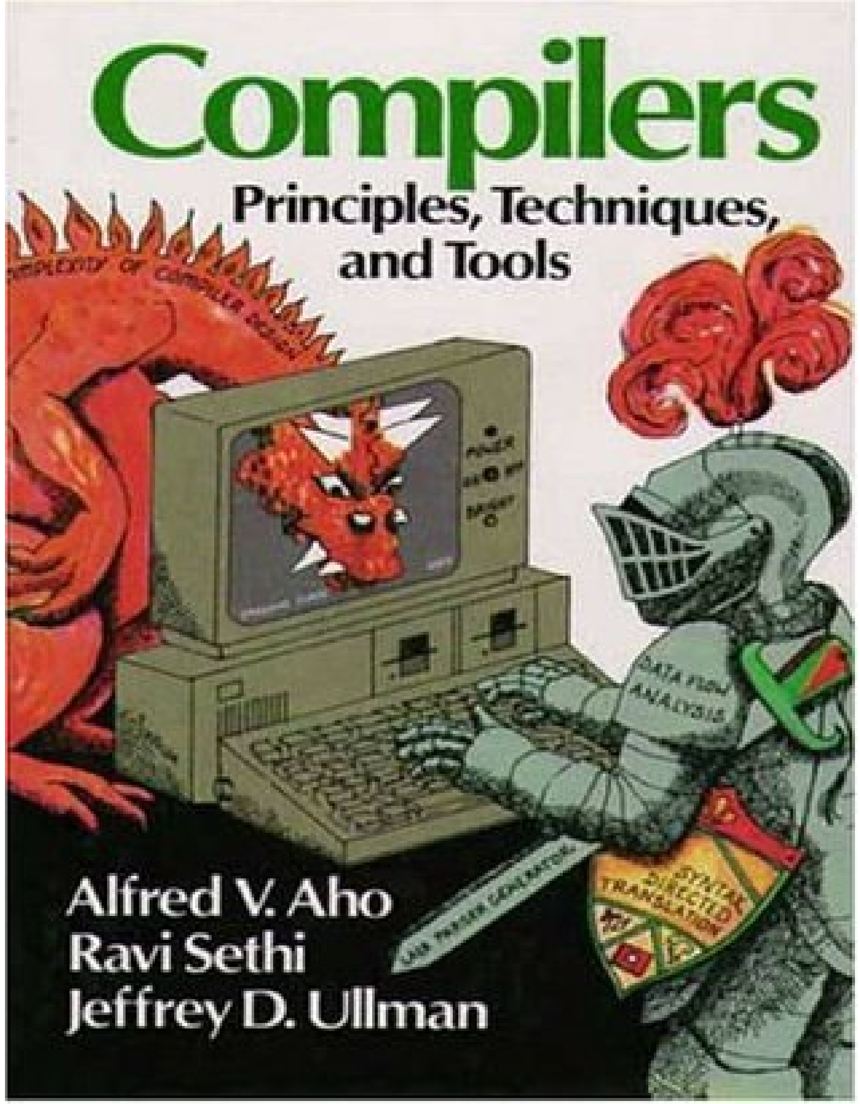
- **Midterm**: **February 10** (Thursday), 8:00-9:20 AM (in class)
- Midterm review session / Q&A: February 8 (Tuesday)

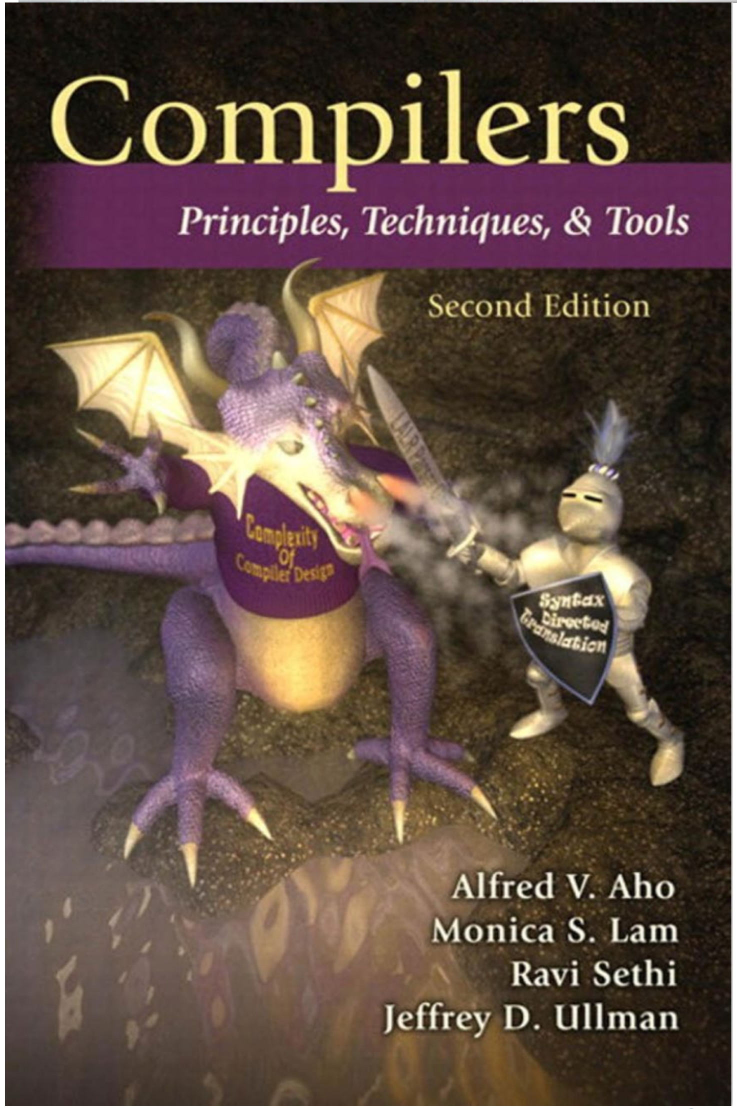
- Project proposal due: February 14
  - Options: (a) literature review paper or (b) compiler + accelerator project

- In-person instruction starting next week (week 5)!
- First in-person class: ***February 1***

Compilers







“Complexity of Compiler Design”, “LALR parser generator”, “Syntax Directed Translation”, “Data Flow Analysis”

Flow Graphs

- **Flow Graph**: A triple $G = (N, A, s)$ where (N, A) is a (finite) directed graph, $s \in N$ is a designated entry node, and there is a path from node s to every node $n \in N$ (s dominates every node $n \in N$).
- **Entry node**: A node with no predecessors
- **Exit node**: A node with no successors

Properties

- There is a unique entry node, which must be s (reachability assumption)
- Conservative: some branches may never be taken
- Control flow graphs are usually sparse. That is, $|A| = O(|N|)$. In fact, if only binary branching is allowed $|A| \leq 2|N|$.

Dominance in Flow Graphs

Definitions

- d dominates n ("$d\ dom\ n$") *iff* every path in G from s to n contains d.
- d properly dominates n if d dominates n and $d \neq n$.
- d is the immediate dominator of n ("$d\ idom\ n$") if d is the last dominator on any path from initial node to n, $d \neq n$.

Properties

- $s\ dom\ d$, $\forall$ node d in G.
- Partial Ordering: The dominance relation of a flow graph G is a partial ordering:
  - Reflective: $n\ dom\ n$ is true $\forall n$.
  - Antisymmetric: if $d\ dom\ n$, then $n\ dom\ d$ *cannot* hold.
  - Transitive: $(d_1\ dom\ d_2) \wedge (d_2\ dom\ d_3) \Longrightarrow d_1\ dom\ d_3$

Dominator Tree

- The dominators of a node form a chain:
- If $d_1\ dom\ n$ and $d_2\ dom\ n$, and $d_1 \neq d_2$, then $d_1\ dom\ d_2$ ***or*** $d_2\ dom\ d_1$.

  $\Longrightarrow$ Every node $n \neq s$ has a ***unique*** immediate dominator.

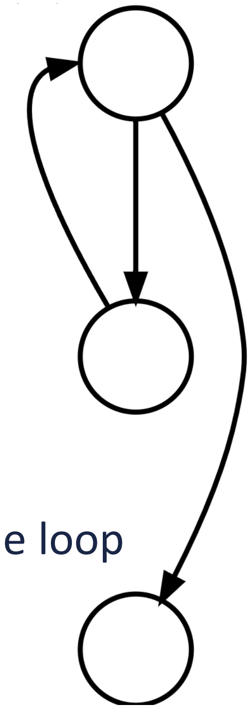
Dominator Tree

The *Dominator Tree* of a flow graph G is a graph with the same nodes as G, and an edge $n_1 \rightarrow n_2$ *iff* $n_1\ dom\ n_2$.

Defining Loops in Flow Graphs

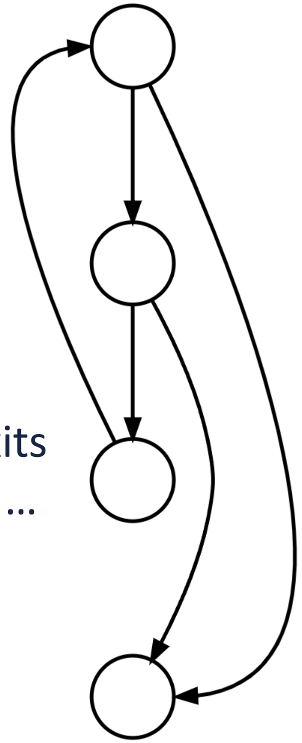
Why defining loops is challenging?

- Easy case: structured nested loops: FOR loop or WHILE loop
- Harder case: arbitrary flow and exits in loop body, but unique loop “entry”
- Hardest case: no unique loop “entry” (“irreducible loops”)

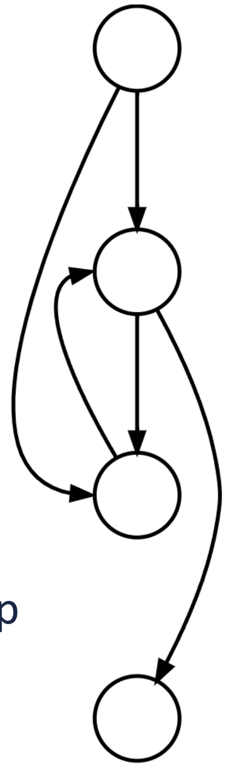


A while loop

A natural loop with two exits (e.g., while loop with an if … break in the middle), non-structured but reducible

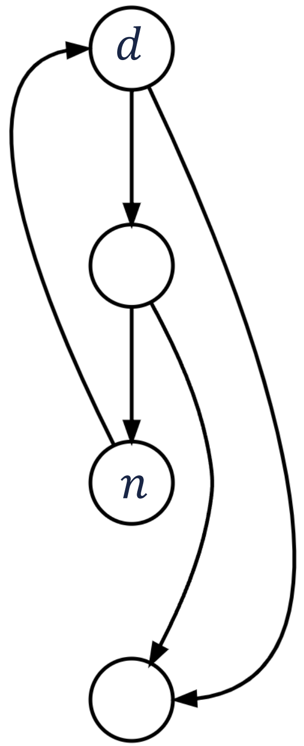


An irreducible CFG, a loop with two entry points, e.g., goto into a while or for loop



Defining Loops in Flow Graphs

- **Back Edge**: An edge $n \rightarrow d$ where $d\ dom\ n$
- **Natural Loop**: Given a back edge, $n \rightarrow d$, the *natural loop* corresponding to $n \rightarrow d$ is the set of nodes: $\{d$ and all nodes that can reach n without going through $d\}$



- **Loop Header**: A node d that dominates all nodes in the loop
- Header is unique for each natural loop

  $\Longrightarrow d$ is the unique entry point into the loop
- Uniqueness is very useful for many optimizations

Preheader: An Optimization Convenience

The Idea:

- If a loop has multiple incoming edges to the header, moving code out of the loop safely is complicated
- Preheader gives a safe place to move code before a loop

The Transformation:

- Introduce a pre-header p for each loop (let loop header be d):
  - Insert node p with one out edge: $p \rightarrow d$
  - All edges that previously entered d should now enter p

Reducible and Irreducible Flow Graphs

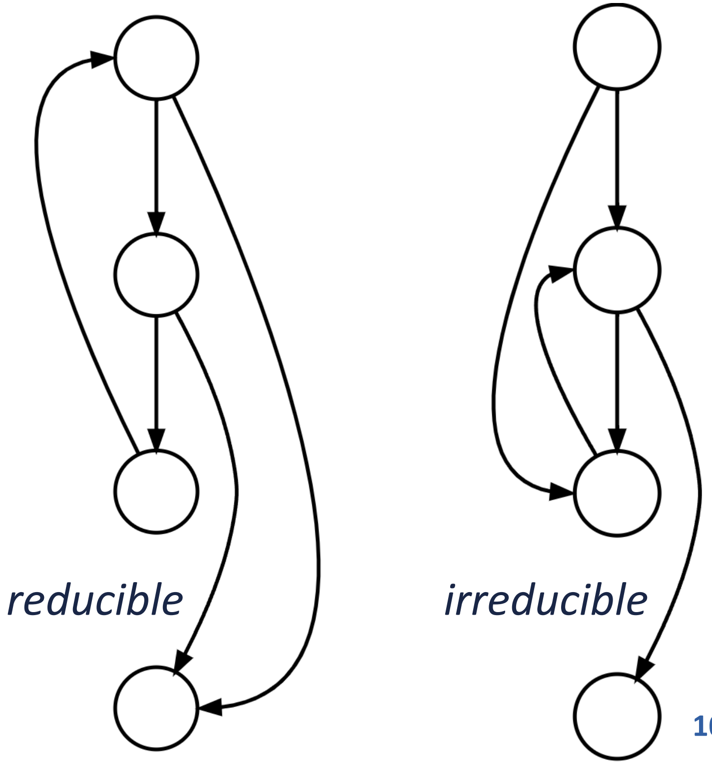
Reducible Flow Graph:

A flow graph G is called reducible *iff* we can partition the edges into two sets:

- *Forward edges*: should form a DAG in which every node is reachable from initial node
- Other edges must be *back edges*: i.e., only these edges $n \rightarrow d$ where $d \; dom \; n$
- Otherwise, graph is called *irreducible*
- *Idea:*
- Every "cycle" has at least one back edge

$\Longrightarrow$ All "cycles" in a reducible graph are natural loops

(NOT true in an irreducible graph!)



Dataflow Analysis

- A technique for collecting information about the flow of values and other program properties over control-flow paths

- Examples
  - What definitions of *x* reach a given use of *x* (and vice versa)?
  - Are any uses research by a particular definition of *x*?
  - What <ptr, target> pairs are possible at each statement?
  - Is variable *x* defined on every path to a use of *x*?
  - Is a pointer to a local variable live on exit from a procedure?
- Applications
  - Pointer Analysis
  - Type inference
  - Common Subexpression Elimination (CSE)
  - Loop-invariant code motion
  - …

Definitions

- ***Alias or alias pair***: two different names for the same storage location
- ***Reference***: An occurrence of a name in a program statement
- ***Use of a variable***: A reference that *may* read the value of the variable
- ***Definition of a variable***: A reference that *may* store a value into the storage
    - Unambiguous definition: guaranteed to store to *X*
    - Ambiguous definition: may store to *X*

Ambiguity comes from aliases, unpredictable side effects of procedure calls, arrays

Dataflow Analysis Basics

- ***Point***: A location in a basic block just before or after some statement
- ***Path***: A path from p_1 to p_n is a sequence of points $p_1, p_2, \ldots, p_n$ such that (intuitively) some execution can visit these points in order

- ***Kill of a definition***: A definition d of variable V is killed on a path if there is an unambiguous definition of V on that path
- ***Kill of an expression***: An expression e is killed on a path if there is a possible definition of any of the variables of e on that path

An Example Dataflow Problem: Reaching Definitions

- **Reaching Definitions**
  - $\forall p$, Compute REACH(p): the set of defs that reach point p.
  - Definition d reaches points p if there is a path from the point after d to p such that d is not killed along that path

- **Dataflow Variables** (for each basic block `B`)
  - `Gen(B)`: the set of defs in B that are not killed in `B`
  - `Kill(B)`: the set of all defs that are killed in `B`
  - `In(B)`: the set of defs that reach the point before first statement in `B`
  - `Out(B)`: the set of defs that reach the point after last statement in `B`

Local properties of *B*

Global dataflow properties

Dataflow Analysis for Reaching Definitions

- Dataflow Equations

Confluence operator

$$In(B) = \bigcup_{p:p\rightarrow B} Out[p]$$

$$Out[B] = Gen[B] \bigcup (In[B] - Kill[B])$$

- Dataflow Algorithms

Goal: solve these $2n$ simultaneous dataflow equations (n is # of basic blocks)

  - Block-structured graph (no GOTO; no BREAK from loops):
    - bottom-up evaluation, one scope at a time
  - General flow graphs:
    - Iterative solution

Iterative Algorithm for Reaching Definitions

1. Initialize:

```
/* If there are globals or formals, in[s] ≠ ϕ */
in[B]  = ϕ      ∀B
out[B] = gen[B] ∀B
```

2. Iterate until Out[B] does not change:

```
do
    change = false
    for each block B do
        In(B) = ∪_{p:p→B} Out[p]
        oldout = Out[B]
        Out[B] = Gen[B] ∪ (In(B) − Kill[B])
        if (oldout ≠ Out[B]) change = true
    end
 while (change == true)
```

$$In(B) = \bigcup_{p:p\rightarrow B} Out[p]$$

$$oldout = Out[B]$$

$$Out[B] = Gen[B] \cup (In(B) - Kill[B])$$

Convergence of the Algorithm

Out[B] converges in a finite number of iterations. Why?

- Out[B] is finite $\forall B$
- Out[B] never decrease $\forall B$
  - Only KILL sets (constants) are ever subtracted from OUT sets
  - IN sets never decrease (if OUT sets never decrease)

Acyclic Property

- Definitions need propagate only over acyclic paths
  - Each block only adds Gen[B], subtracts Kill[B]
  - $\cup$, $-$: only need to add, remove once
  - Must visit each block exactly once
- Also need one final iteration to check convergence

Assume reducible graphs → cycles formed by back edges

- No back edges: 2 iterations
- 1 back edge (on any acyclic path): 3 iterations
- K back edges on any acyclic path: k + 2 iterations

Another Example: Available Expressions

- Available Expressions: $x + y$ is available at point p if:
  - Every path to p evaluates $x + y$
  - Between the last such evaluation and p on each path, neither x nor y is modified

Kill: Block B kills $x + y$ if it may assign to x or y, and it does not subsequently recompute $x + y$

Generate: Block B generates $x + y$ if it definitely evaluates $x + y$, and it does not subsequently modify x or y

Dataflow variables:

Let U = universal set of expressions in the program. Then:

$$in[B] = \{\epsilon \in U | \epsilon \text{ is available at entry to } B\}$$

$$out[B] = \{\epsilon \in U | \epsilon \text{ is available at exit to } B\}$$

$$e\_kill[B] = \{\epsilon \in U | \epsilon \text{ is generated by } B\}$$

$$e\_kill[B] = \{\epsilon \in U | \epsilon \text{ is killed by } B\}$$

Naming Expressions

Examples

```
a = x * y;                      // eval e_1: x * y
b = x * y;                      // eval e_1: x * y: redundant
x = 2;                          // "kills" e_1
c = x * y;                      // eval e_1: x * y

if (...) { x=5; t= x+y; }       // eval e_2: x+y
else     { x=9; t= x+y; }       // eval e_2: x+y
x = x+y;                        // eval e_2: x+y: redundant!

p = cond? &X : &Z;
... = *p + 1;            // e_3: X+1, e_4: Y+1 may not be eval
... = X + 1;             // eval e_3: X+1 may not be redundant
```

Dataflow Analysis for Available Expressions

- Dataflow Equations:

$$In(B) = \bigcap_{p:p \to B} Out[p]$$

$$Out[B] = e\_gen[B] \bigcap (In[B] - e\_kill[B])$$

- Iterative Algorithm: algorithm is identical to *Reaching Definitions* except:
  - *Confluence* operator is ∩ instead of ∪
  - Algorithm must initialize sets as follows:

$$In[s] = \emptyset$$
$$Out[s] = e\_gen[s]$$
$$Out[B] = U - e\_kill[B], \forall B \neq s$$

General Approach to Dataflow Analysis

- Step 1: Choose dataflow variables for problems of interest:
  - `Gen[B]`: "information" generated in block B
  - `Kill[B]`: "information" killed in block B
  - `In[B]`, `Out[B]`
- Step 2: Set up dataflow equations
  - What is the transfer function for each block?

    e.g., $Out[B] = Gen[B] \cup (In[B] - Kill[B])$
  - Is it a forward or backward problem?

    e.g., $In(B) = \bigcup_{p:p \to B} Out[p]$ or $In(B) = \bigcap_{p:p \to B} Out[p]$
  - What is the "confluence" operator?

    e.g., ∪, ∩, or other
- Step 3: Solve iteratively until convergence
  - Postorder (PO) or Reverse Postorder (RPO)

Def-Use and Use-Def Chains

- Use-Def chain or ud-chain: For each use u of a variable v, $\textsc{Defs}(u)$ is the set of instructions that may have defined v last prior to u
- Def-Use chain or du-chain: For each use d of a variable v, USES(d) is the set of instructions that may use the value of v computed at d

Note: $d \in \textsc{Defs}(u)$ iff $u \in \textsc{Uses}(d)$

Note: du-chains (or ud-chains) form a graph

Comparing with SSA:

- Multiple defs reach each use, unlike SSA
- More edges in def-use graph than in SSA graph
- Fewer variable names, no ϕ functions

Constructing and Using du-Chains and ud-Chains

Constrution:

- Construct DEFS(u) from the results of Reaching Definitions
- Invert DEFS to compute USES

$\Longrightarrow$ we can build chains very efficiently!

Some applications of chains:

- Building live ranges for graph-coloring register allocation
- Constant propagation
- Dead-code elimination
- Loop-invariant code motion

EECS 221:
Languages and Compilers for Hardware Accelerators

(Winter 2022)

Sitao Huang

sitaoh@uci.edu