Data Flow Analysis

Lecture 6
ECS 240

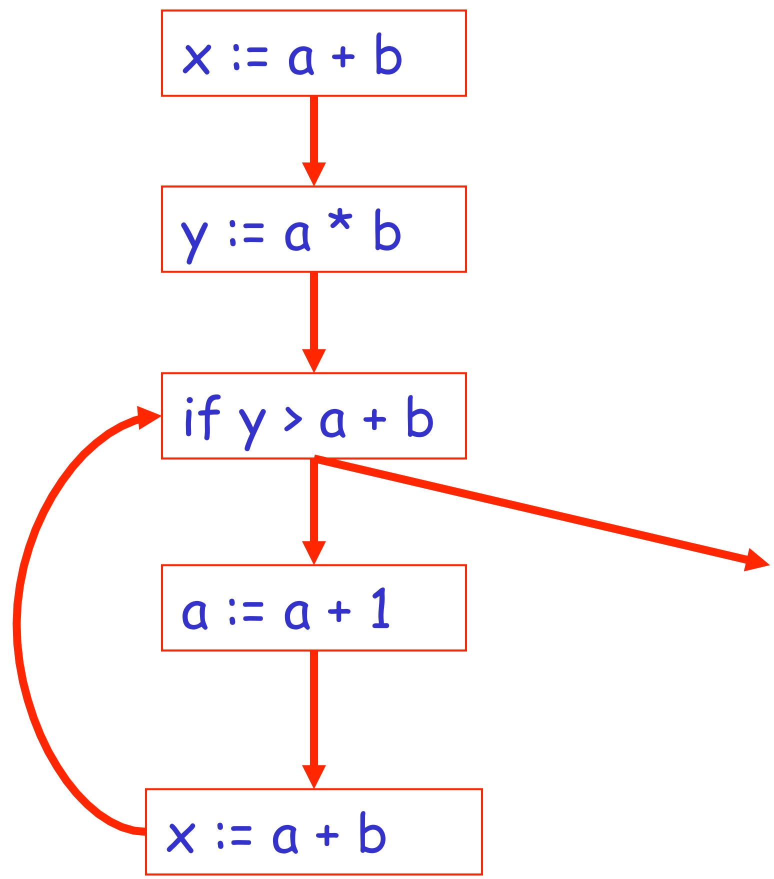
The Plan

- Introduce a few example analyses
- Generalize to see the underlying theory
- Discuss some more advanced issues

Control-Flow Graphs

```
x := a + b;
y := a * b;
while y > a + b {
    a := a + 1;
    x := a + b;
}
```

Control-flow graphs are state-transition systems.



Notation

s is a statement

succ(s) = { *successor statements of s* }

pred(s) = { *predecessor statements of s* }

write(s) = { *variables written by s* }

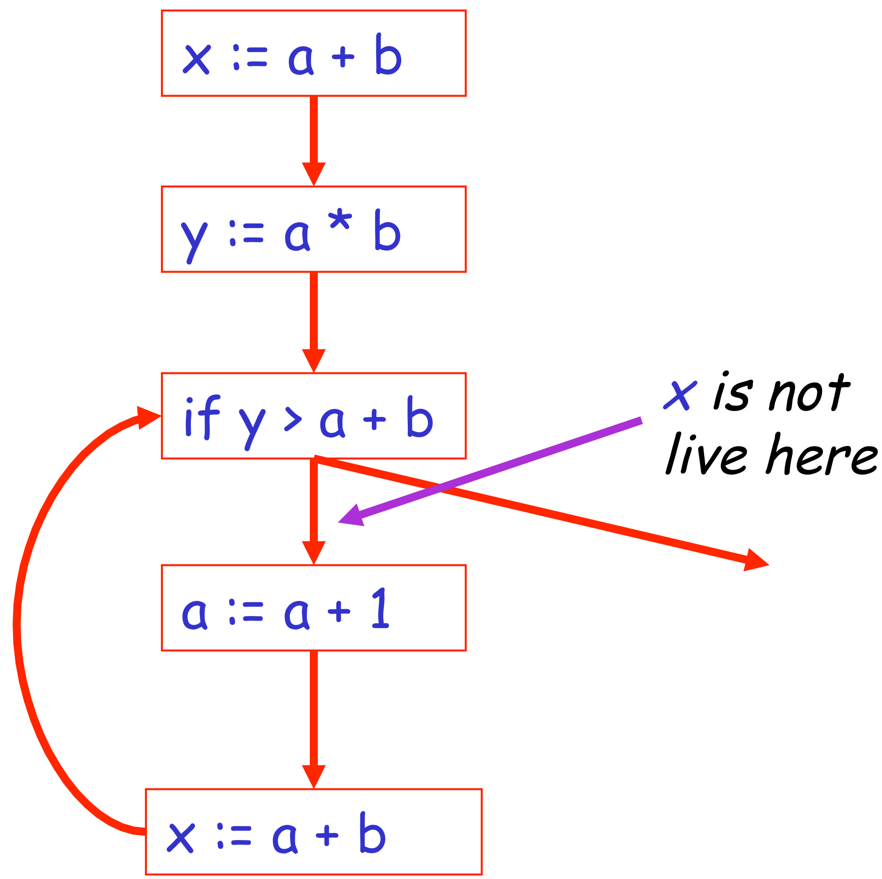
read(s) = { *variables read by s* }

Kill(s) = facts killed by statement s

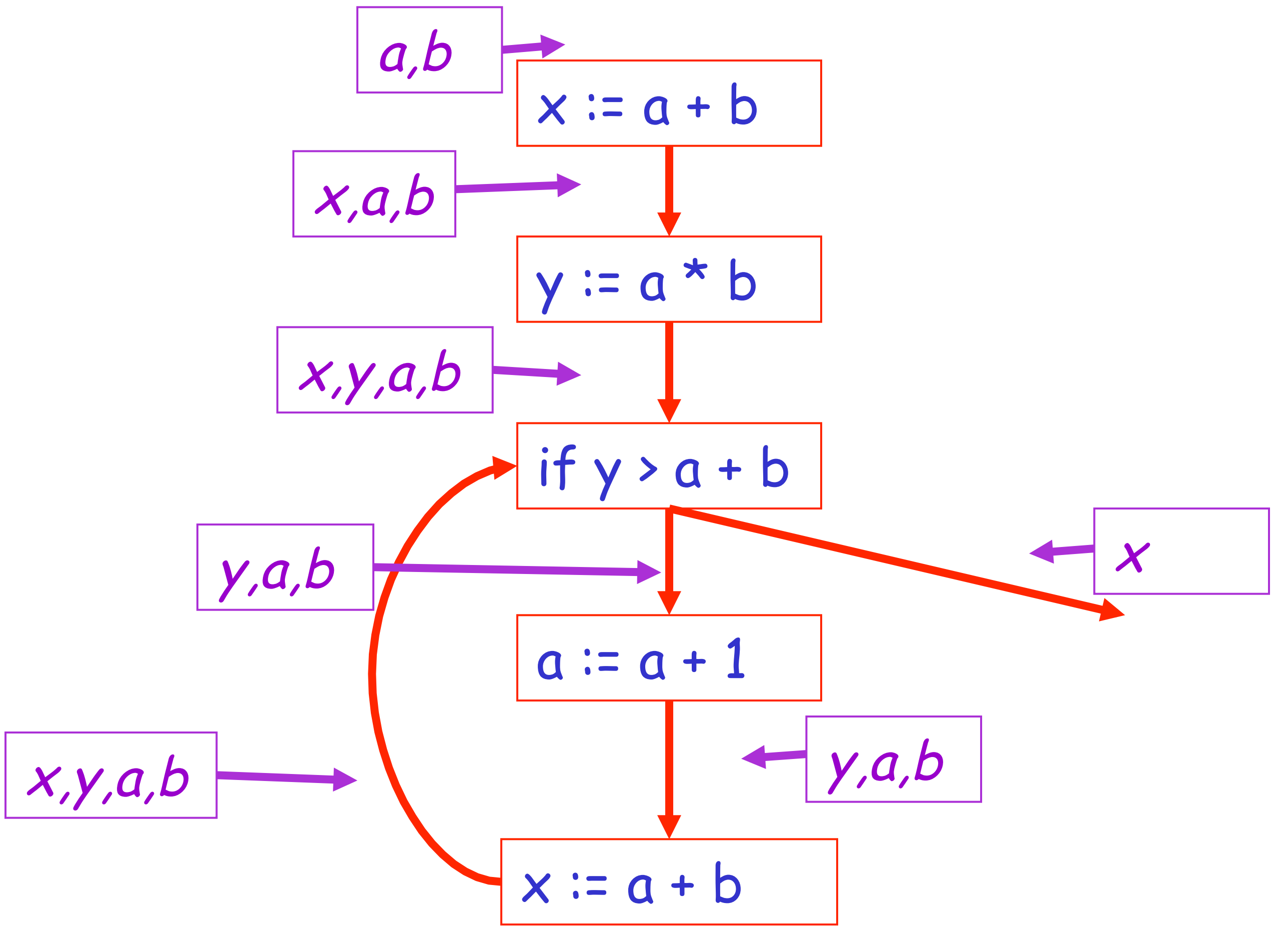
Gen(s) = facts generated by statement s

Liveness Analysis

- For each program point p, which of the variables defined at that point are used on some execution path?
- Optimization: If a variable is not live, no need to keep it in a register.



Example



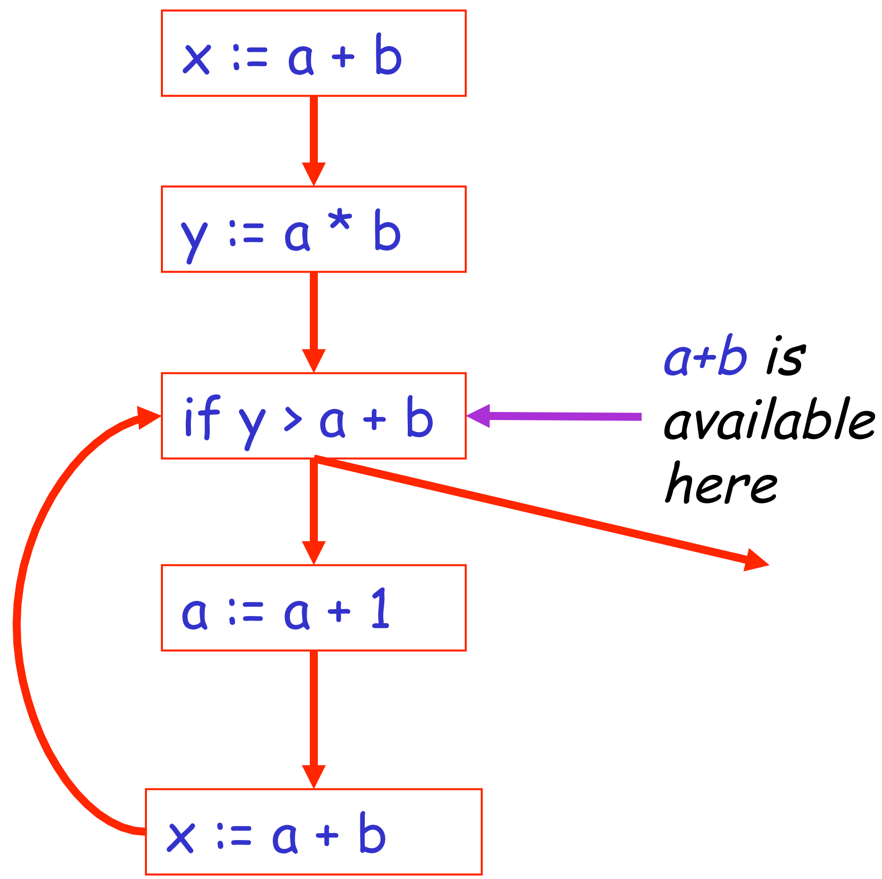
Dataflow Equations

$$L_{in}(s) = (L_{out}(s) - write(s)) \cup read(s)$$

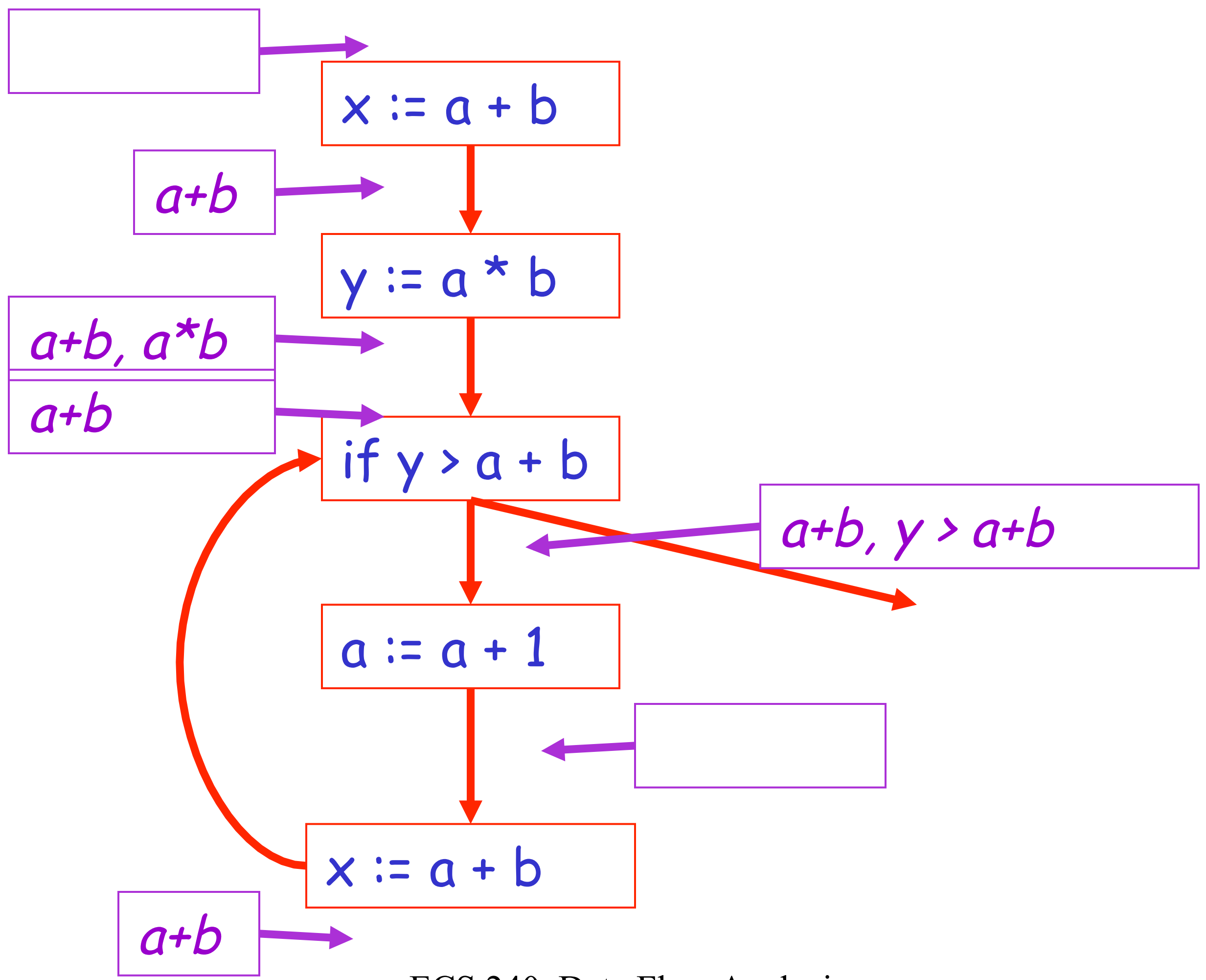
$$L_{out}(s) = \begin{cases} \emptyset & \text{if } succ(s) = \emptyset \\ \bigcup_{s' \in succ(s)} L_{in}(s') & \text{otherwise} \end{cases}$$

Available Expressions

- For each program point p, which expressions must have already been computed, and not later modified, on all paths to p.
- Optimization: Where available, expressions need not be recomputed.



Example



Dataflow Equations

$$A_{in}(s) = \begin{cases} \emptyset & \text{if } pred(s) = \emptyset \\ \bigcap_{s' \in pred(s)} A_{out}(s') & \text{otherwise} \end{cases}$$

$$A_{out}(s) = (A_{in}(s) - \{a \in S \mid write(s) \cap V(a) \neq \emptyset\})$$
$$\cup \{s \mid \text{if } write(s) \cap read(s) = \emptyset\}$$

Available Expressions: Schematic

$$A_{in}(s) = \bigcap_{s' \in pred(s)} A_{out}(s')$$

Transfer function:

$$A_{out}(s) = A_{in}(s) - C_1 \cup C_2$$

Must analysis: property holds on all paths

Forwards analysis: from inputs to outputs

Live Variables Again

$$L_{in}(s) = (L_{out}(s) - write(s)) \cup read(s)$$

$$L_{out}(s) = \left\{ \begin{array}{ll} \emptyset & \text{if } succ(s) = \emptyset \\ \bigcup_{s' \in succ(s)} L_{in}(s') & \text{otherwise} \end{array} \right\}$$

Live Variables: Schematic

Transfer function:

$$L_{in}(s) = L_{out}(s) - C_1 \cup C_2$$

$$L_{out}(s) = \bigcup_{s' \in succ(s)} L_{in}(s')$$

May analysis: property holds on some path

Backwards analysis: from outputs to inputs

Very Busy Expressions

- An expression *e* is very busy at program point p if every path from p must evaluate e before any variable in e is redefined

- Optimization: hoisting expressions

- A must-analysis
- A backwards analysis

Reaching Definitions

- For a program point p, which assignments made on paths reaching p have not been overwritten

- Connects definitions with uses (use-def chains)

- A may-anlaysis
- A forwards analysis

One Cut at the Dataflow Design Space

	May	*Must*
Forwards	Reaching definitions	Available expressions
Backwards	Live variables	Very busy expressions

The Literature

- Vast literature of dataflow analyses

- 90+% can be described by
  - Forwards or backwards
  - May or must

- Some oddballs, but not many
  - Bidirectional analyses

Another Cut at Dataflow Design

- What theory are we dealing with?
- Review our schemas:

$$A_{in}(s) = \bigcap_{s' \in pred(s)} A_{out}(s')$$

$$L_{in}(s) = L_{out}(s) - C_1 \cup C_2$$

$$A_{out}(s) = A_{in}(s) - C_1 \cup C_2$$

$$L_{out}(s) = \bigcup_{s' \in succ(s)} L_{in}(s')$$

Essential Features

- Set variables $L_{in}(s)$, $L_{out}(S)$
- Set operations: union, intersection
  - Restricted complement (- constant)
- Domain of atoms
  - E.g., variable names
- Equations with single variable on lhs

Dataflow Problems

- Many dataflow equations are described by the grammar:

$$EQS \rightarrow v = E ; EQS \mid \varepsilon$$

$$E \rightarrow E \cap E \mid E \cup E \mid v \mid a$$

- v is a variable
- a is an atom
- Note: More general than most problems . . .

Solving Dataflow Equations

- Simple worklist algorithm:
  - Initially let $S(v) = 0$ for all v
  - Repeat until $S(v) = S(E)$ for all equations
    - Pick any $v = E$ such that $S(v) \neq S(E)$
    - Set $S := S[v/S(E)]$

Termination

- How do we know the algorithm terminates?
- Because
  - operations are *monotonic*
  - the domain is finite

Monotonicity

- Operation f is monotonic if

$$x \leq y \Rightarrow f(x) \leq f(y)$$

- We require that all operations be monotonic
  - Easy to check for the set operations
  - Easy to check for all transfer functions; recall:

$$L_{in}(s) = L_{out}(s) - C_1 \cup C_2$$

Termination again

- To see the algorithm terminates
  - All variables start empty
  - Variables and rhs’s only increase with each update
    - By induction on # of updates, using monotonicity
  - Sets can only grow to a max finite size
- Together, these imply termination

The Rest of the Lecture

- Distributive Problems
- Flow Sensitivity
- Context Sensitivity
  - Or interprocedural analysis

- What are the limits of dataflow analysis?

Distributive Dataflow Problems

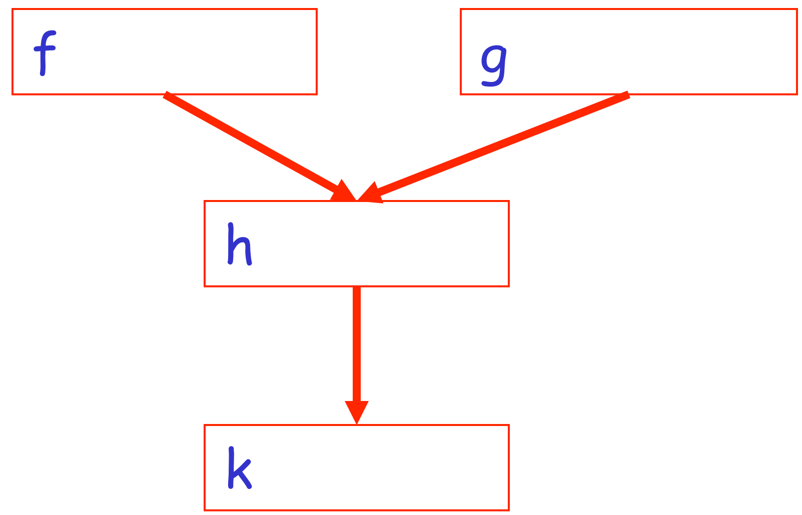
- Monotonicity implies for a transfer function f:

$$f(x \cup y) \geq f(x) \cup f(y)$$

- Distributive dataflow problems satisfy a stronger property:

$$f(x \cup y) = f(x) \cup f(y)$$

Distributivity Example



$k(h(f(0) \cup g(0))) =$

$k(h(f(0)) \cup h(g(0))) =$

$k(h(f(0))) \cup k(h(g(0)))$

The analysis of the graph is equivalent to combining the analysis of each path!

Meet Over All Paths

- If a dataflow problem is distributive, then the (least) solution of the dataflow equations is equivalent to the analyzing every path (including infinite ones) and combining the results

- Says joins cause no loss of information

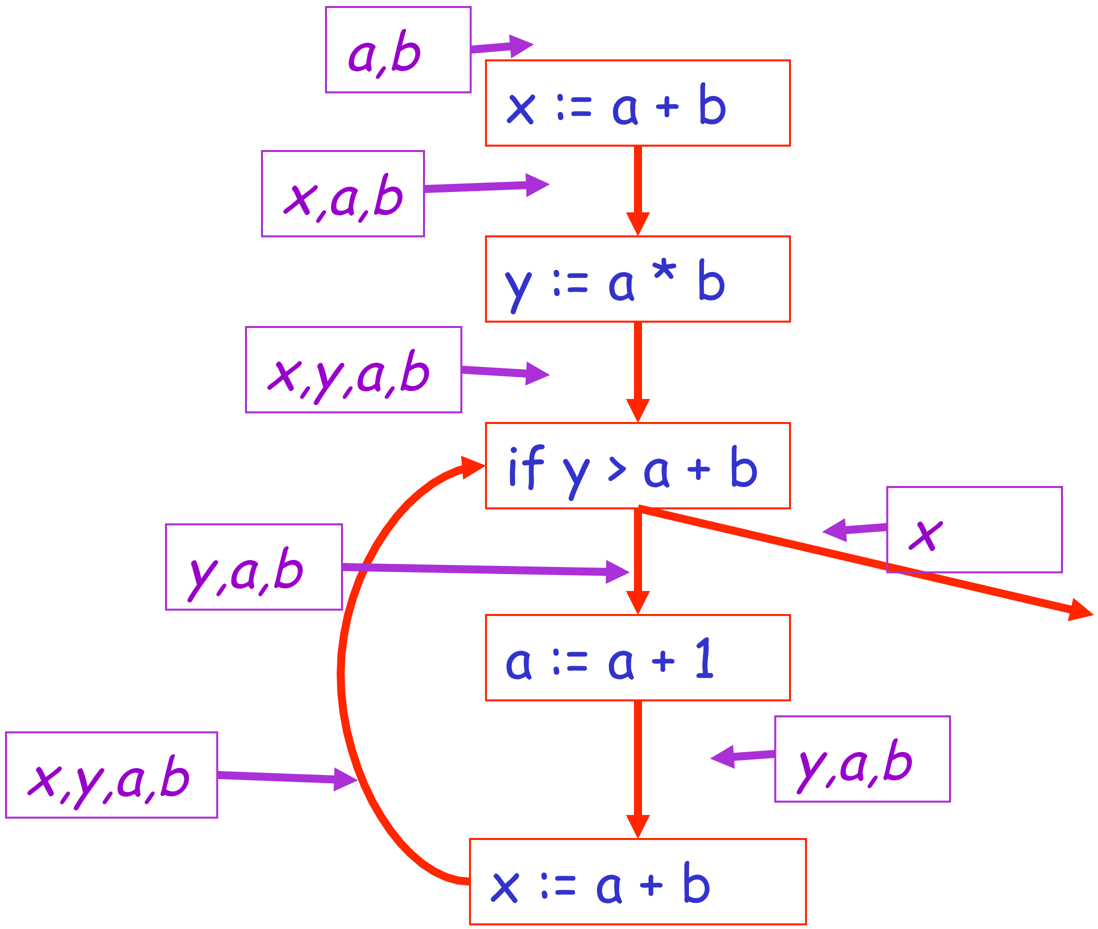
Distributivity Again

- Obtaining the meet over all paths solution is a very powerful guarantee
- Says that dataflow analysis is really as good as you can do for a distributive problem.
- Alternatively, can be viewed as saying distributive problems are very easy indeed . . .

What Problems are Distributive?

- Many analyses of program structure are distributive
  - E.g., live variables, available expressions, reaching definitions, very busy expressions
  - Properties of *how* the program computes

Liveness Example Revisited



Constant Folding

- Ordering $i < \top$ for any integer i
- $j \sqcup k = \top$ if $j \neq k$
- Example transfer function:

$$C(v := e_1 \times e_2)\sigma = \sigma[v \leftarrow C(e_1)\sigma \otimes C(e_2)\sigma]$$

$$\text{where } a \otimes b = \begin{cases} a \times b & \text{if } a,b \text{ constants} \\ \text{ú} & \text{otherwise} \end{cases}$$

- Consider $C(z := y * y)[y = 1] \cup C(z := y * y)[y = -1]$

$$C(z := y * y)([y = 1] \cup [y = -1])$$

What Problems are Not Distributive?

- Analyses of *what* the program computes
  - The output is (a constant, positive, …)

Flow Sensitivity

- Flow sensitive analyses
  - The order of statements matters
  - Need a control flow graph
    - Or transition system, ….

- Flow insensitive analyses
  - The order of statements doesn't matter
  - Analysis is the same regardless of statement order

Example Flow Insensitive Analysis

- What variables does a program fragment modify?

$$G(x := e) = \{x\}$$
$$G(s_1; s_2) = G(s_1) \cup G(s_2)$$

- Note $G(s_1;s_2) = G(s_2;s_1)$

The Advantage

- Flow-sensitive analyses require a model of program state at each program point
  - E.g., liveness analysis, reaching definitions, …

- Flow-insensitive analyses require only a single global state
  - E.g., for *G*, the set of all variables modified

Notes on Flow Sensitivity

- Flow insensitive analyses seem weak, but:
- Flow sensitive analyses are hard to scale to very large programs
  - Additional cost: state size X # of program points
- Beyond 1000's of lines of code, only flow insensitive analyses have been shown to scale

Context-Sensitive Analysis

- What about analyzing across procedure boundaries?

```
Def f(x){…}
Def g(y){…f(a)…}
Def h(z){…f(b)…}
```

- Goal: Specialize analysis of f to take advantage of
  - f is called with a by g
  - f is called with b by h

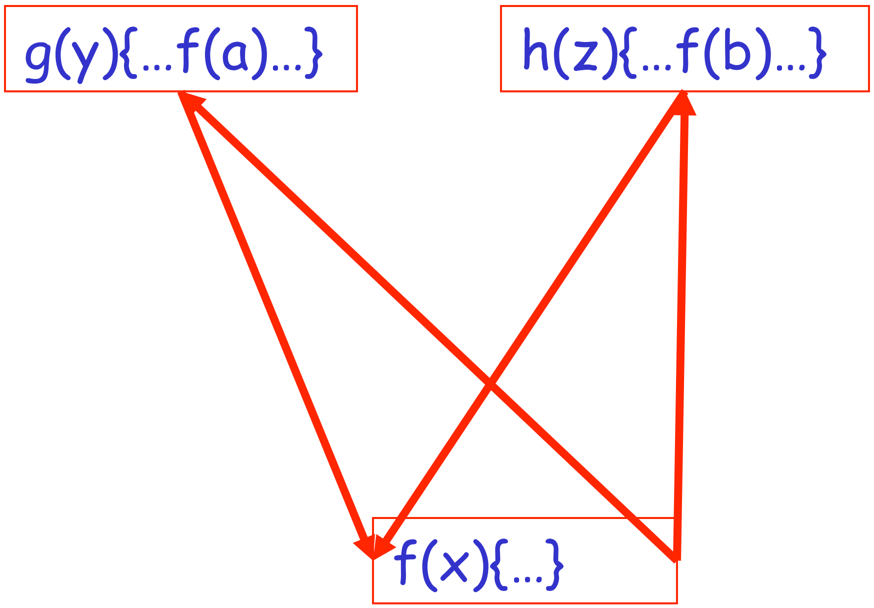
Control-Flow Graphs Again

- How do we extend control-flow graphs to procedures?

- Idea: Model procedure call f(a) by:
  - Edge from point before call to entry of f
  - Edge from exit(s) of f to point after call

Example

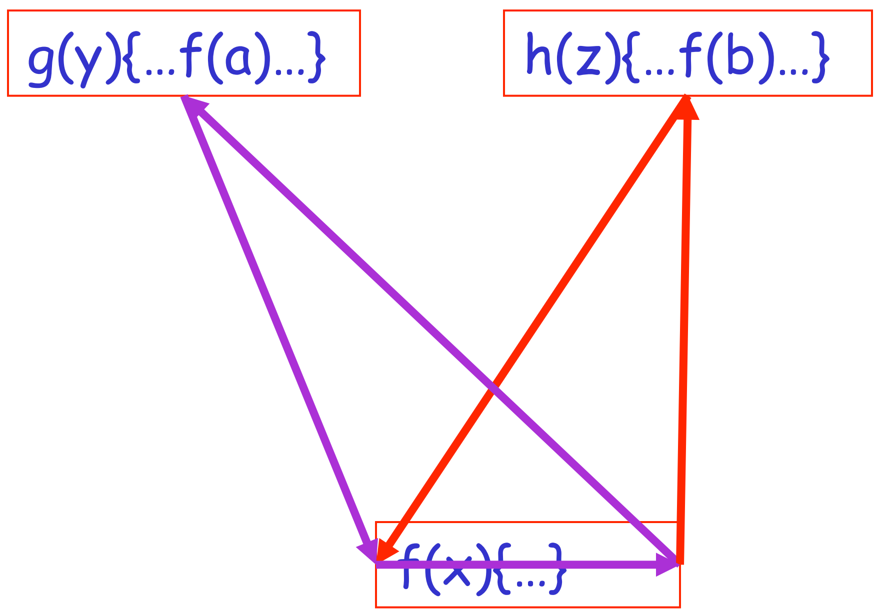
- Edges from
  - before f(a) to entry of f
  - Exit of f to after f(a)
  - Before f(b) to entry of f
  - Exit of f to after f(b)



Example

- Edges from
  - before f(a) to entry of f
  - Exit of f to after f(a)
  - Before f(b) to entry of f
  - Exit of f to after f(b)

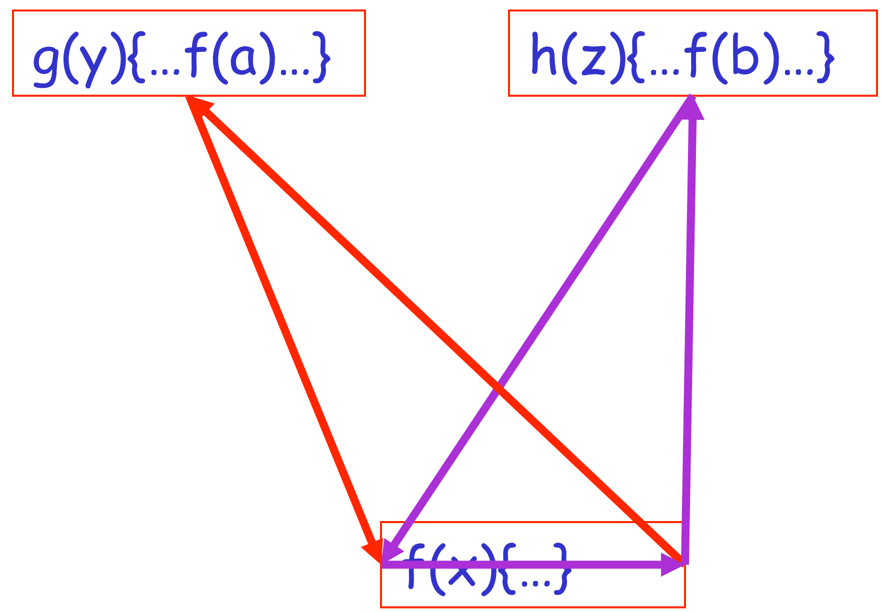
- Has the correct flows for g



Example

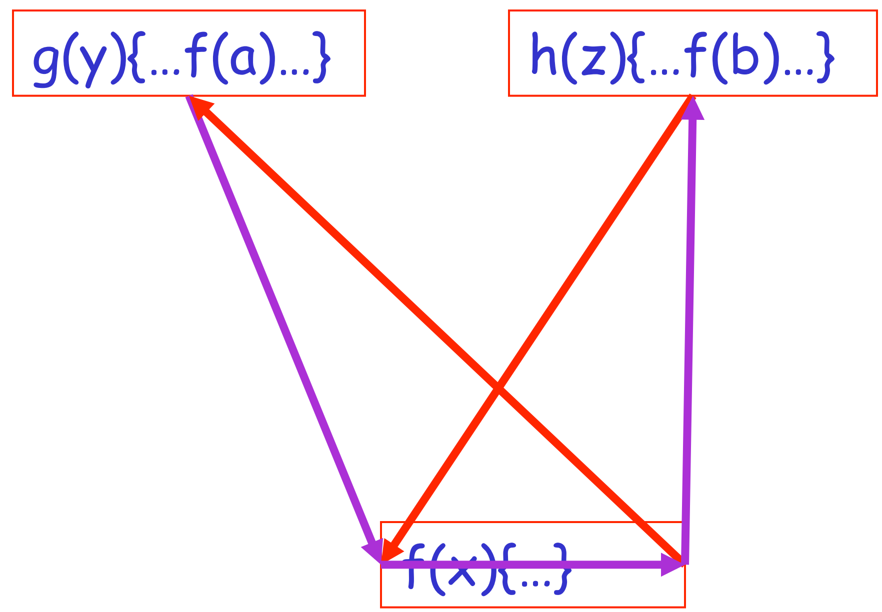
- Edges from
  - before f(a) to entry of f
  - Exit of f to after f(a)
  - Before f(b) to entry of f
  - Exit of f to after f(b)

- Has the correct flows for h



Example

- But also has flows we don't want
  - One path captures a call to g returning at h!
- So-called "infeasible paths"



What to do?

- Must distinguish calls to f in different contexts

- Three techniques
  - Assumptions
    - later
  - Context-free reachability
    - Later
  - Call strings
    - Today

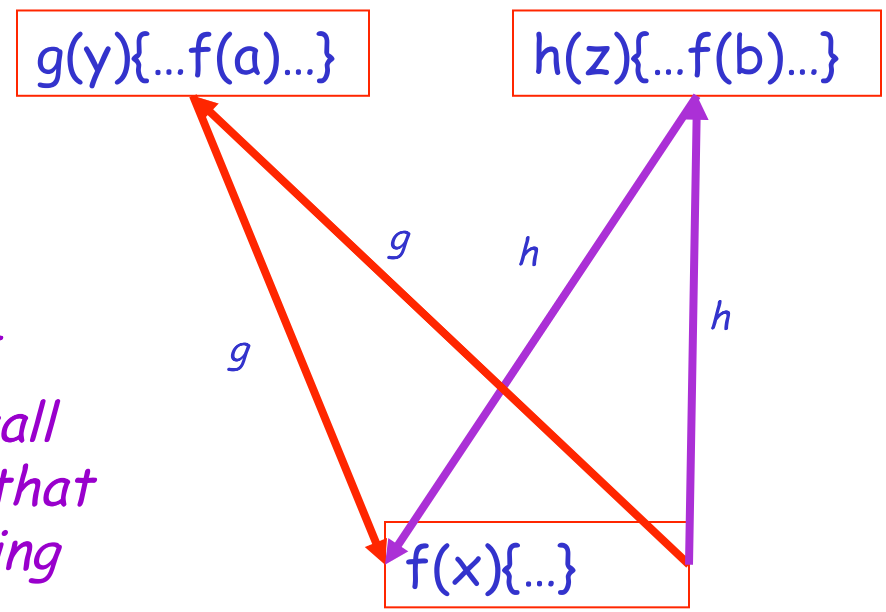
Call Strings

- Observation:
  - At run time, different calls to f are distinguished by the call stack
- Problem:
  - The stack is unbounded
- Idea:
  - Use the last k calls on the stack to distinguish context
  - Represent a call by the name of the calling procedure

<

Example Revisited

- Use call strings of length 1
- Context is name of calling procedure



Note: labels on edges are part of the state: tag a call with "g" on call of f() from g(), filter out all but that portion of the state with call string "g" on return from g() to f()

Experience with Call Strings

- Very expensive
  - Multiplies # of abstract values by (# of procedures ** length of call string)
  - Hard to contemplate call strings > 1
- Fragile
  - Very sensitive to organization of procedures
- Well-studied, but not much used in practice

Review of Terminology

- Must vs. May
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Context-sensitive vs. Context-insensitive
- Distributive vs. non-Distributive

Where is Dataflow Analysis Useful?

- Best for flow-sensitive, context-insensitive, distributive problems on small pieces of code
  - E.g., the examples we've seen and many others
- Extremely efficient algorithms are known
  - Use different representation than control-flow graph, but not fundamentally different
  - More on this in a minute . . .

Where is Dataflow Analysis Weak?

- Lots of places

Data Structures

- Not good at analyzing data structures
- Works well for atomic values
  - Labels, constants, variable names
- Not easily extended to arrays, lists, trees, etc.
  - Work on shape analysis

The Heap

- Good at analyzing flow of values in local variables
- No notion of the heap in traditional dataflow applications
- In general, very hard to model anonymous values accurately
  - Aliasing
  - The "strong update" problem

Context Sensitivity

- Standard dataflow techniques for handling context sensitivity don't scale well
- Brittle under common program edits
- E.g., call strings

Flow Sensitivity (Beyond Procedures)

- Flow sensitive analyses are standard for analyzing single procedures

- Not used (or not aware of uses) for whole programs
  - Too expensive

The Call Graph

- Dataflow analysis requires a call graph
  - Or something close
- Inadequate for higher-order programs
  - First class functions
  - Object-oriented languages with dynamic dispatch
- Call-graph hinders algorithmic efficiency
  - Desire to keep executable specification is limiting

Forwards vs. Backwards

- Restriction to forwards/backwards reachability
  - Very constraining
  - Many important problems not easy to fit into this mold