中国科学技术大学
University of Science and Technology of China

Data Flow Analysis

《程序语言设计和程序分析》

张昱

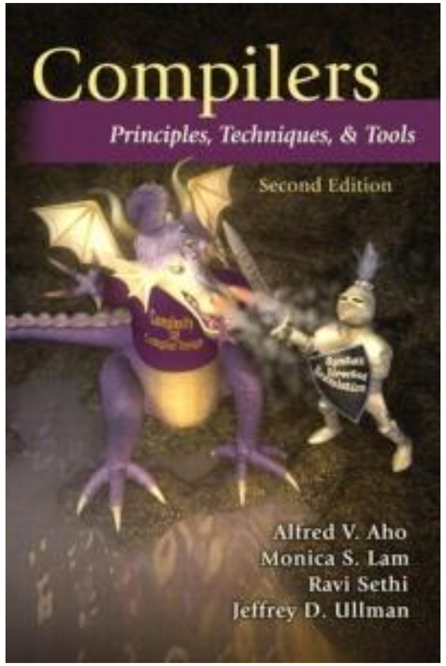
yuzhang@ustc.edu.cn

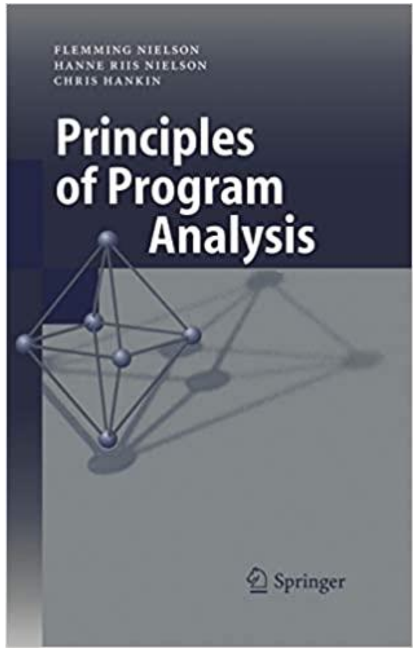
中国科学技术大学

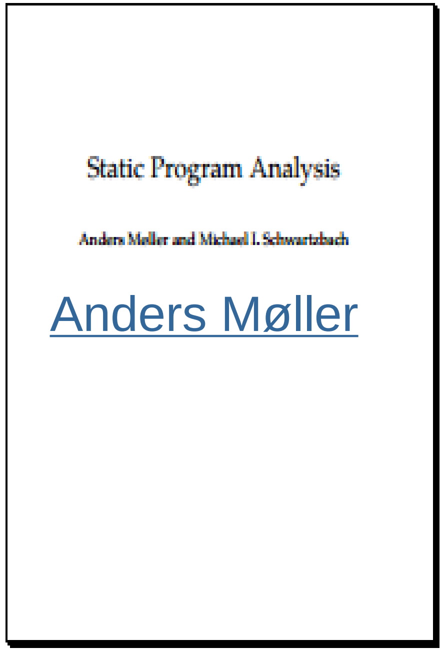
计算机科学与技术学院

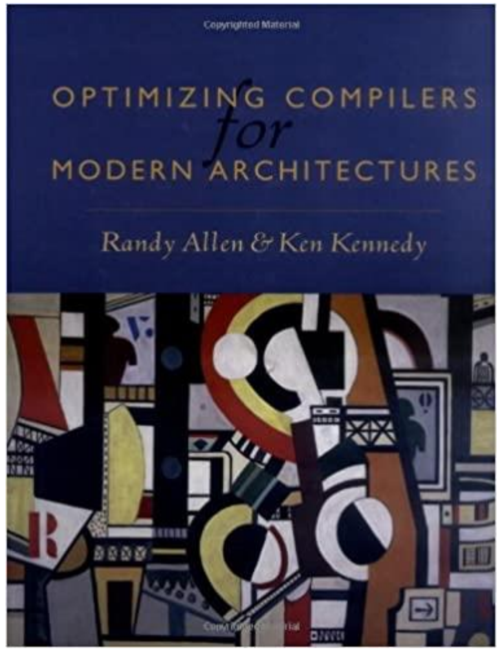
Resources

- Principles of Programming Analysis
- Dragon book: Compilers
- Optimizing Compilers for Modern Architectures
- Static Program Analysis



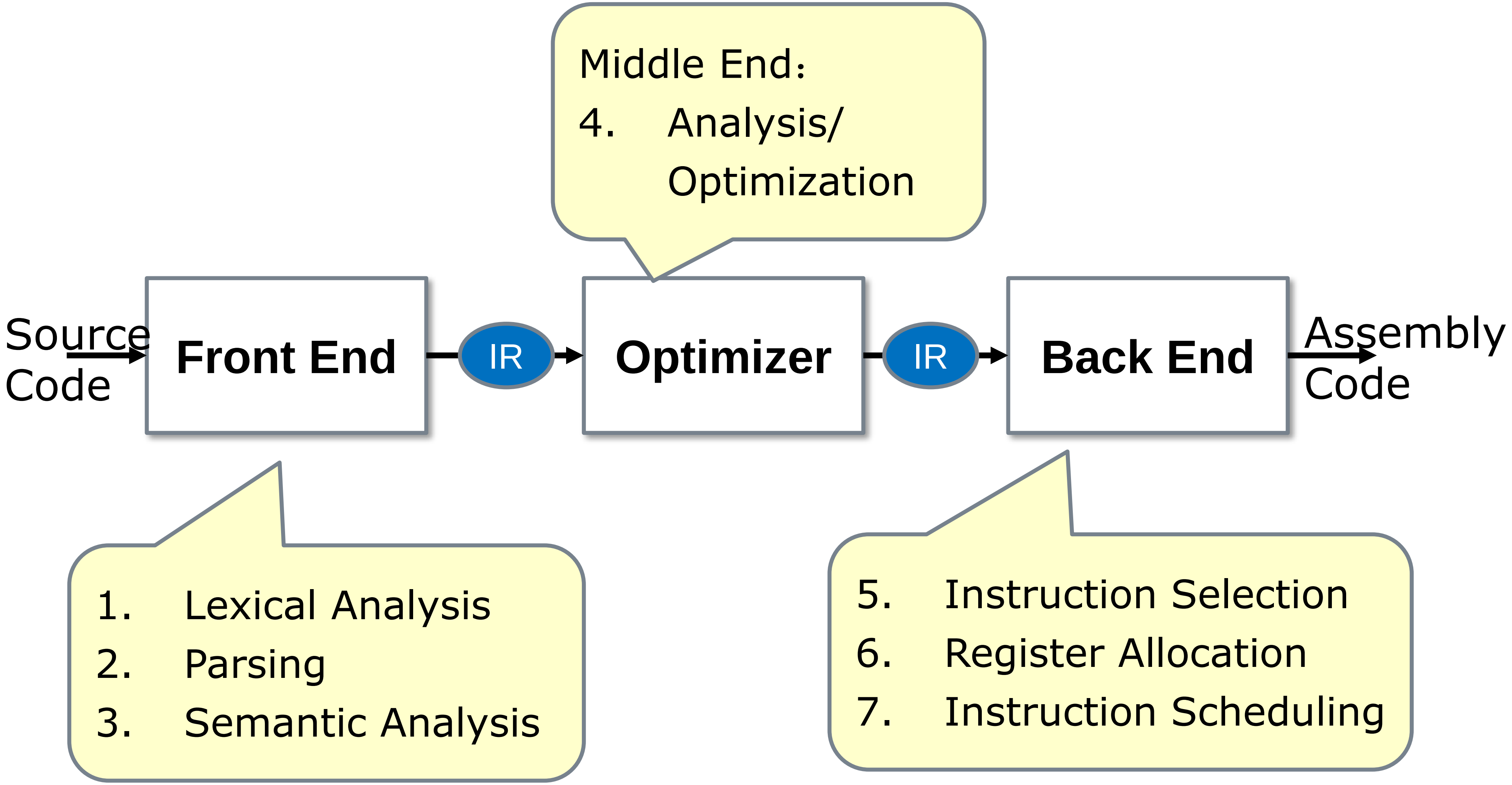






https://github.com/amilajack/reading/tree/master/Type_Systems
https://suif.stanford.edu/papers/

Compiler Structure

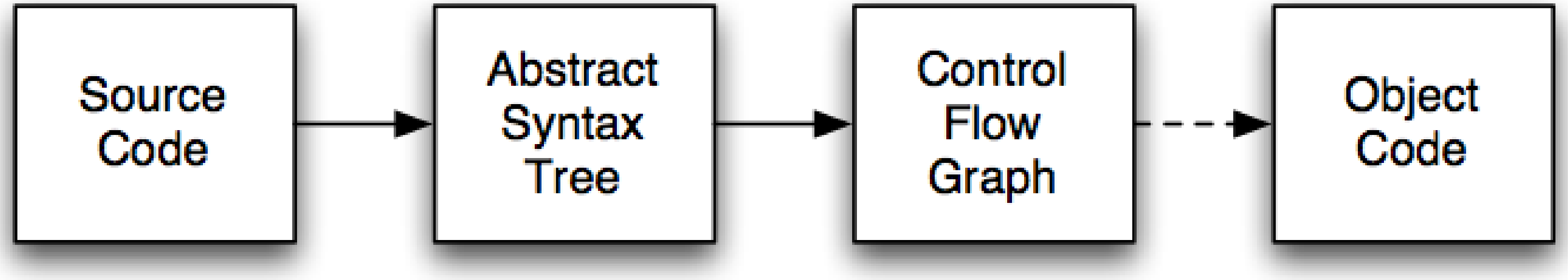


Compiler Structure

- Source code parsed to produce AST

- AST transformed to CFG

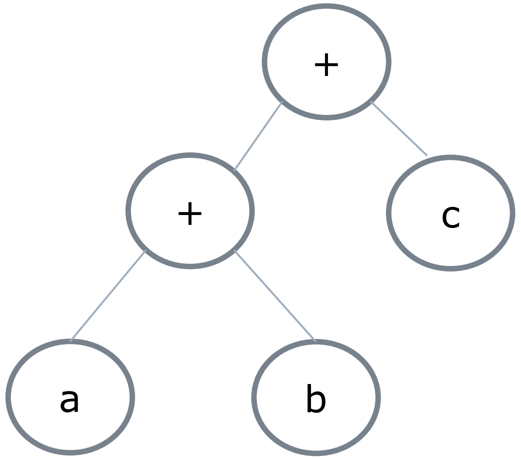
- Data flow analysis operates on control flow graph (and other intermediate representations)



ASTs

ASTs are *abstract*

- They don’t contain all information in the program
  - e.g., spacing, comments, brackets, parentheses
- Any ambiguity has been resolved
  - e.g., a + b + c produces the same AST as (a + b) + c



https://astexplorer.net/

```
import ast
…
source = open(sourcefile, "r").read()
root = ast.parse(source)
```

https://github.com/s4plus/pyscan

Disadvantages of ASTs

- AST has many similar forms
  - e.g., for, while, repeat...until
  - e.g., if, ?:, switch
- Expressions in AST may be *complex*, *nested*
  - (42 * y) + (z > 5 ? 12 * z : z + 20)
- Want simpler representation for analysis
  - ...at least, for dataflow analysis

Control-Flow Graph (CFG)

- **A directed graph where**
  - Each node represents a statement
  - Edges represent control flow

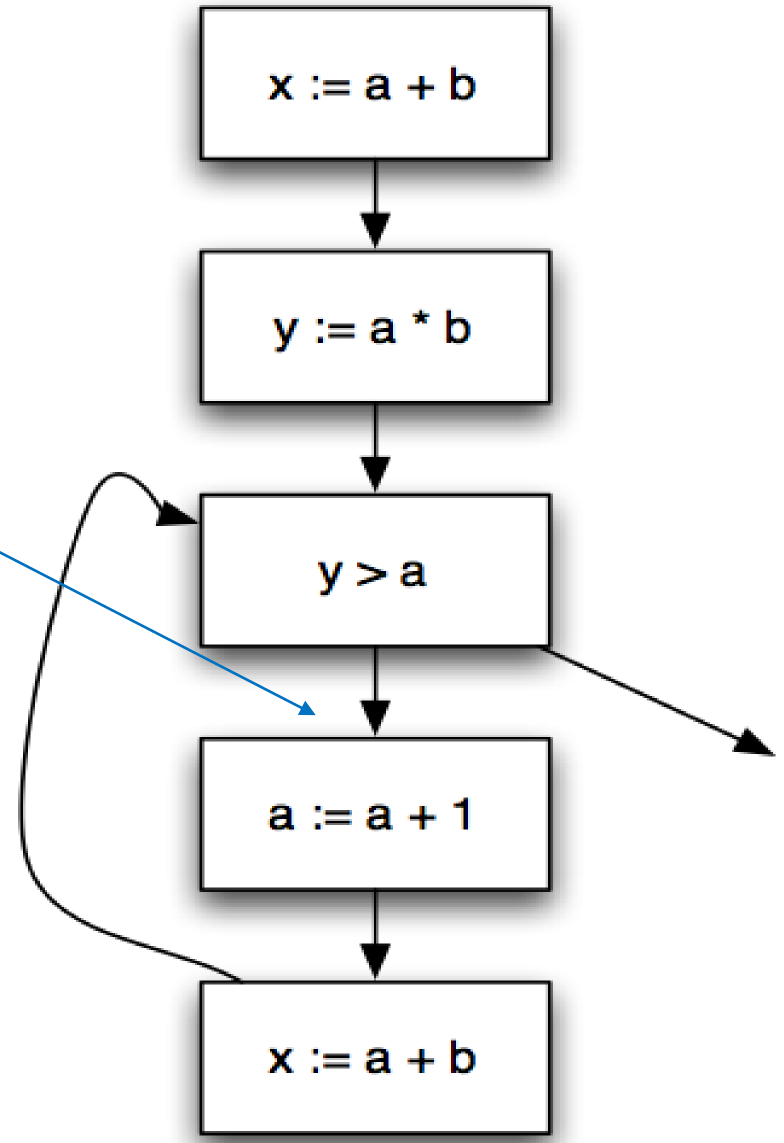
- **Statements may be**
  - Assignments x := y op z or x := op z
  - Copy statements x := y
  - Branches goto L or if x relop y goto L
  - etc.

Control-Flow Graph Example

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

Program point

y > a holds



Invariants (不变式)：

A property holds at a program point if it holds in any such state for any execution with any input

y > a at above program point

Variations on CFGs

中国科学技术大学
University of Science and Technology of China

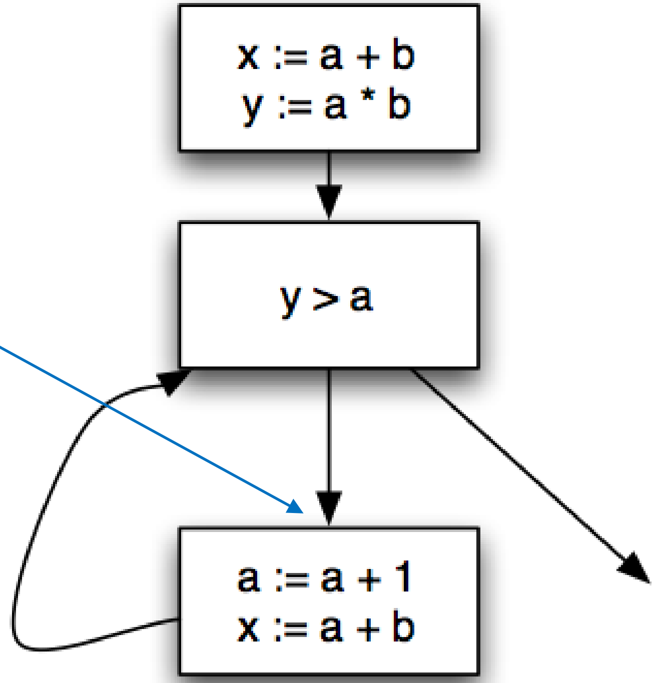
- **We usually don’t include declarations (e.g., int x;)**
  - But there’s usually something in the implementation
- **May want a unique entry and exit node**
  - Won’t matter for the examples we give
- **May group statements into basic blocks**
  - A sequence of instructions with no branches into or out of the block

Control-Flow Graph w/Basic Blocks

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

Program point

y > a holds



- **Can lead to more efficient implementations**
- **But more complicated to explain, so...**
  - We'll use single-statement blocks in lecture today

CFG vs. AST

- CFGs are much simpler than ASTs
  - Fewer forms, less redundancy, only simple expressions
- But...AST is a more faithful representation
  - CFGs introduce temporaries
  - Lose block structure of program
- So for AST,
  - Easier to report error + other messages
  - Easier to explain to programmer
  - Easier to unparse to produce readable code

Data flow analysis: Examples

Data Flow Analysis

- A framework for proving facts about programs
- Reasons about lots of little facts
- Little or no interaction between facts
  - Works best on properties about *how* program computes
- Based on all paths through program
  - Including infeasible paths

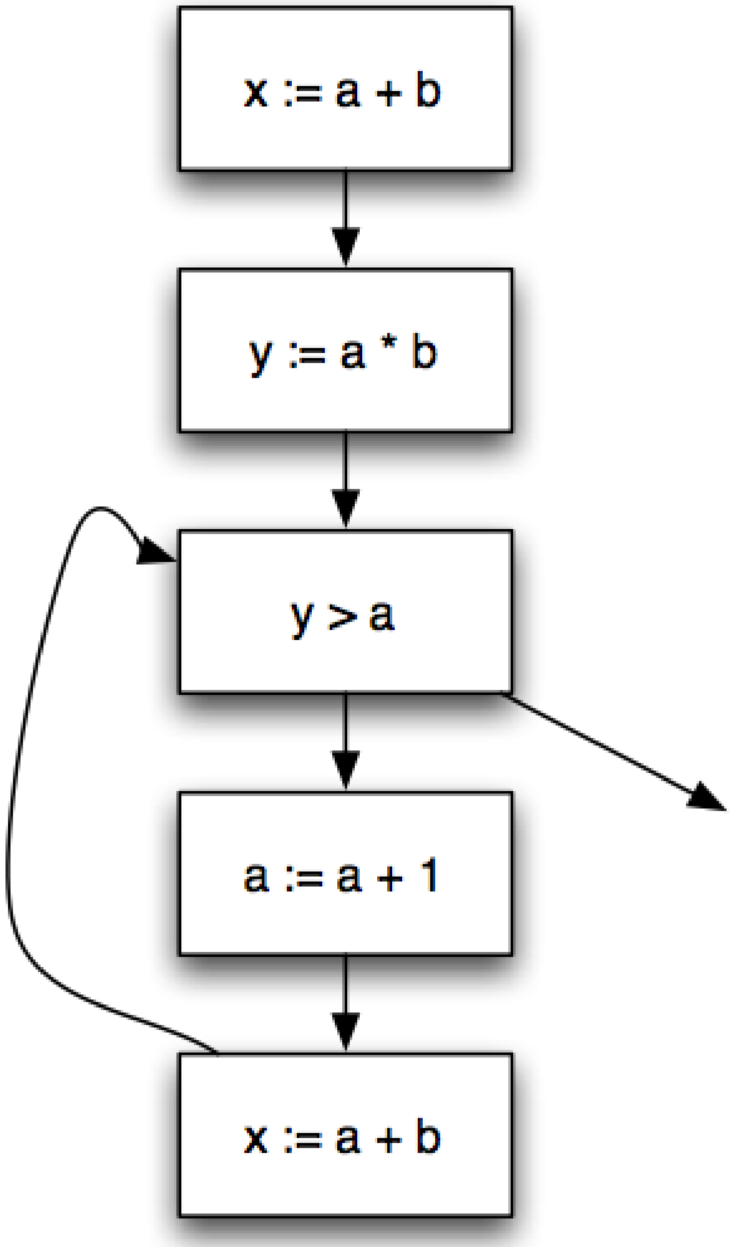
Available Expressions

- **An expression e is available at program point p if**
  - e is computed on every path to p, and
  - the value of e has not changed since the last time e is computed on p

- **Optimization**
  - If an expression is available, need not be recomputed
    - (At least, if it's still in a register somewhere)

Data Flow Facts

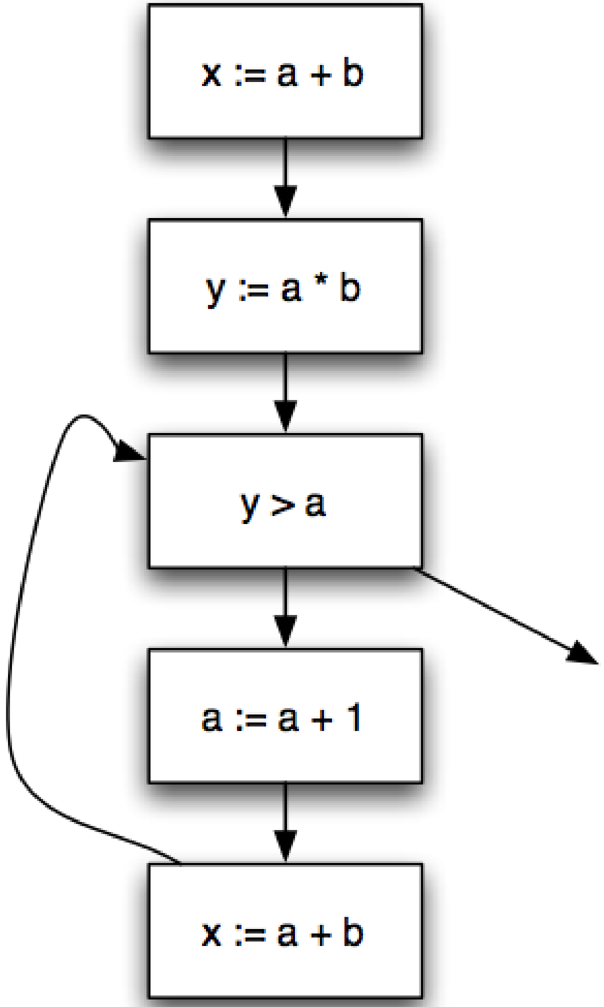
- Is expression e available?
- Facts:
  - a + b is available
  - a * b is available
  - a + 1 is available



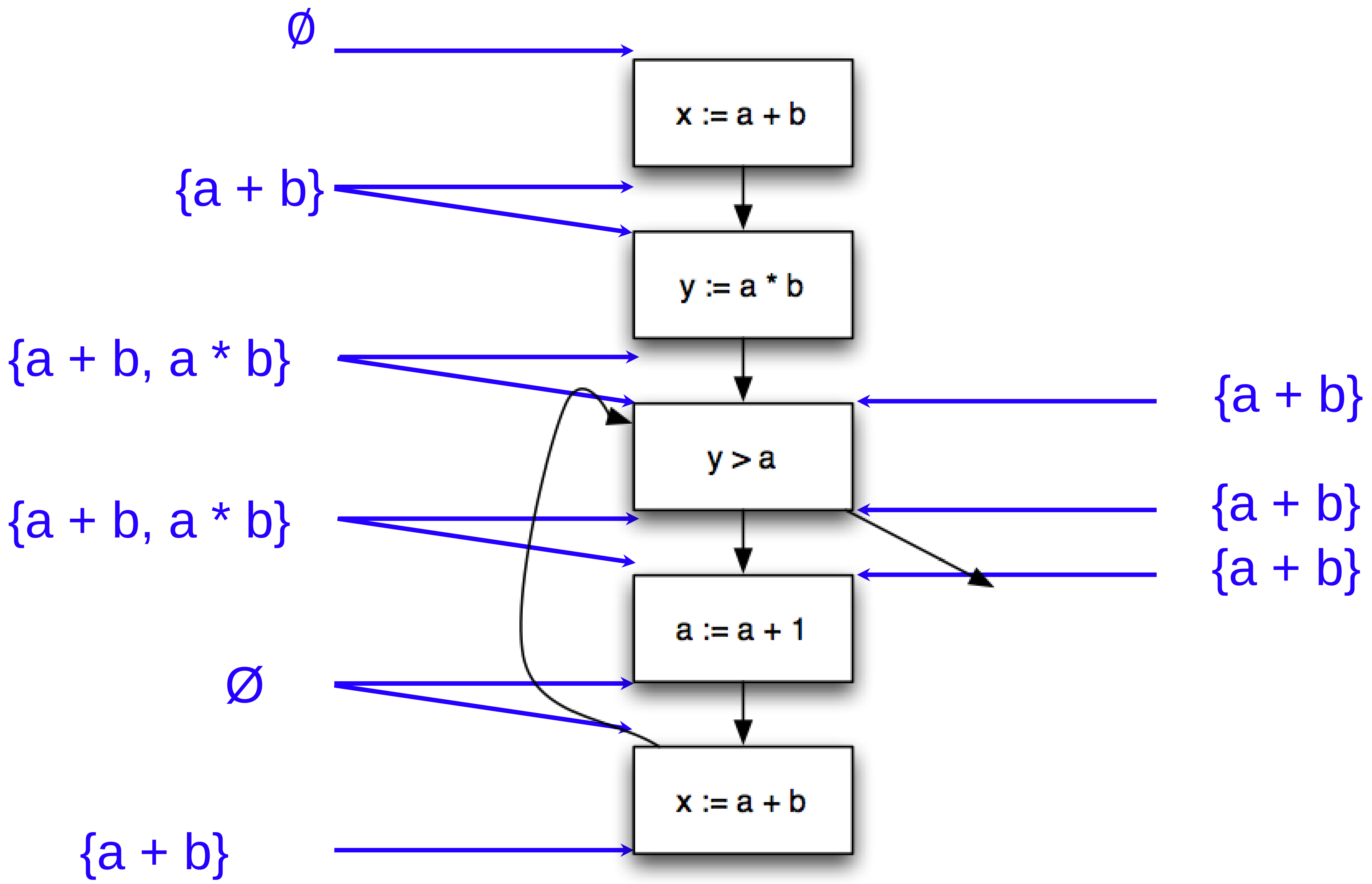
Gen and Kill

- What is the effect of each statement on the set of facts?

Stmt	Gen	Kill
x := a + b	a + b	
y := a * b	a * b	
a := a + 1		a + 1, a + b, a * b



Computing Available Expressions at Each Program Point



Terminology

- A *joint point* is a program point where two branches meet

- Available expressions is a ***forward must*** problem
  - Forward = Data flow from in to out
  - Must = At join point, property must hold on all paths that are joined

Data Flow Equations

中国科学技术大学 University of Science and Technology of China

- **Let s be a statement**
  - succ(s) = { immediate successor statements of s }
  - pred(s) = { immediate predecessor statements of s}
  - In(s) = program point just before executing s
  - Out(s) = program point just after executing s

- **$In(s) = \cap_{s' \in pred(s)} Out(s')$**
- **$Out(s) = Gen(s) \cup (In(s) - Kill(s))$**
  - Note: These are also called *transfer functions*

Liveness Analysis

- **A variable v is *live* at program point p if**
  - v will be used on some execution path originating from p...
  - before v is overwritten

- **Optimization**
  - If a variable is not live, no need to keep it in a register
  - If variable is dead at assignment, can eliminate assignment

Data Flow Equations

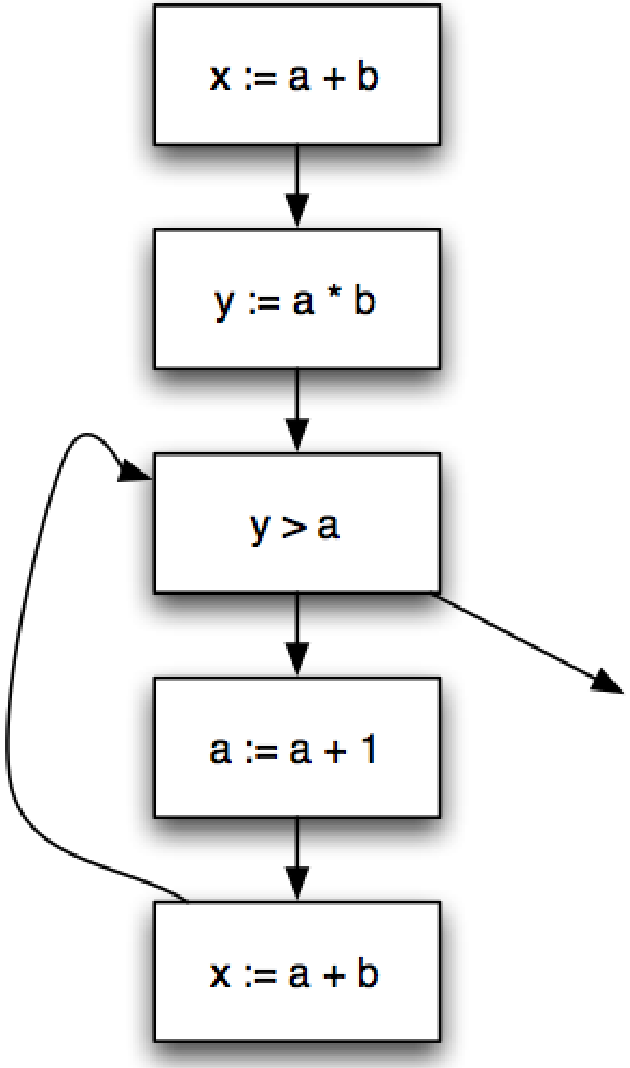
- **Available expressions is a forward must analysis**
  - Data flow propagate in same dir as CFG edges
  - Expr is available only if available on all paths
- **Liveness is a *backward* *may* problem**
  - To know if variable live, need to look at future uses
  - Variable is live if used on some path

- $\mathbf{Out(s)} = \bigcup_{s' \in succ(s)} \mathbf{In(s')}$
- $\mathbf{In(s)} = \mathbf{Gen(s)} \cup (\mathbf{Out(s)} - \mathbf{Kill(s)})$

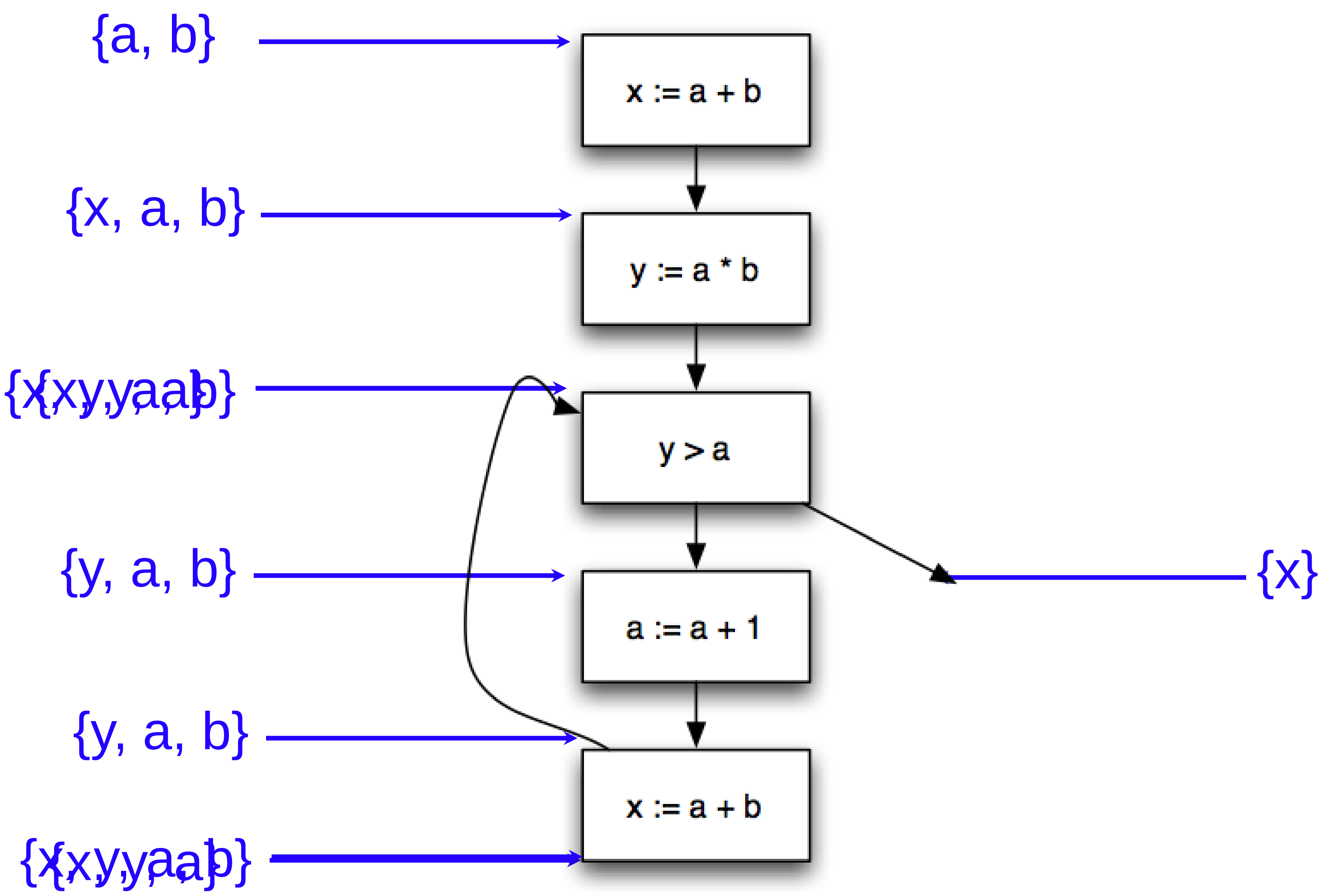
Gen and Kill

- What is the effect of each statement on the set of facts?

Stmt	Gen	Kill
x := a + b	a, b	x
y := a * b	a, b	y
y > a	a, y	
a := a + 1	a	a



Computing Live Variables



Very Busy Expressions

- **An expression e is *very busy* at point p if**
  - On every path from p, expression e is evaluated before the value of e is changed

- **Optimization**
  - Can hoist very busy expression computation

- **What kind of problem?**
  - Forward or backward? backward
  - May or must? must

Reaching Definitions

中国科学技术大学 University of Science and Technology of China

- A *definition* of a variable v is an assignment to v
- A definition of variable v reaches point p if
  - There is no intervening assignment to v
- Also called def-use information
- What kind of problem?
  - Forward or backward?
  - May or must?

forward

may

Space of Data Flow Analyses

	May	Must
Forward	Reaching definitions	Available expressions
Backward	Live variables	Very busy expressions

- **Most data flow analyses can be classified this way**
  - A few don’t fit: bidirectional analysis
- **Lots of literature on data flow analysis**

Generalization

Generalization

Dataflow analysis

- A common framework for such analysis
- Computes information at each program point
- Conservative: characterizes all possible program behaviors

Methodology

- Describe the information (e.g., live variable sets) using a structure called a *lattice*
- Build a system of equations based on:
  - How each statement affects information
  - How information flows between basic blocks
- Solve the system of constraints

Parts of Live Vars Analysis

Live variable sets

- Called *flow values* 流值
- Associated with program points
- Start "empty", eventually contain solution

Effects of instructions

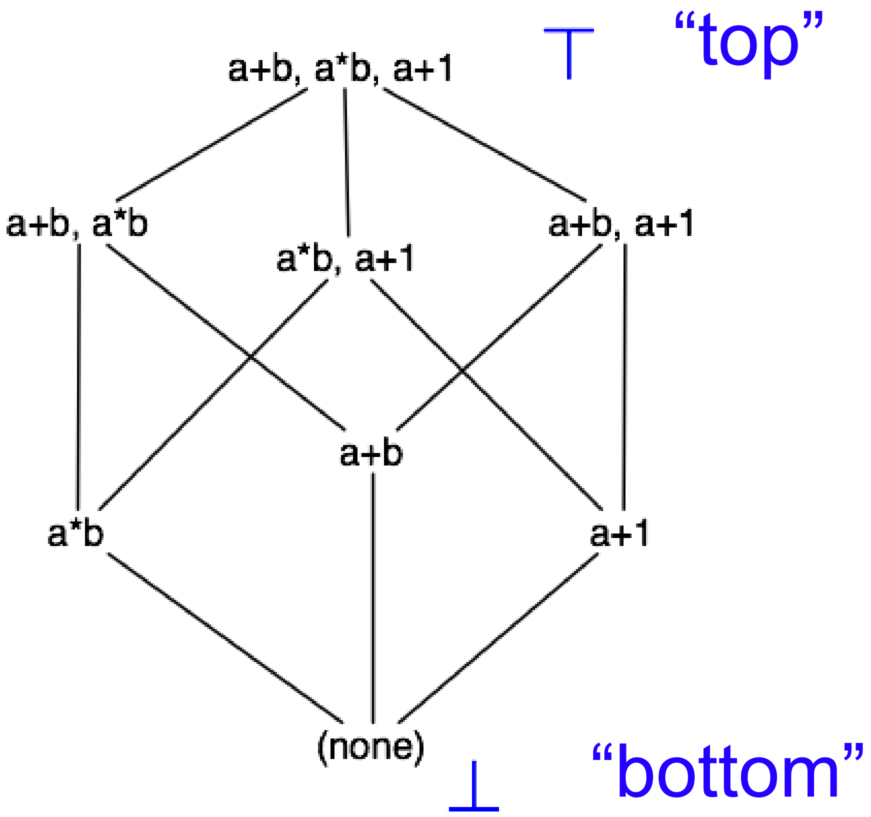
- Called *transfer functions* 迁移函数
- Take a flow value, compute a new flow value that captures the effects
- One for each instruction – often a schema

Handling control flow

- Called *confluence operator* 合流算子
- Combines flow values from different paths

Data Flow Facts and Lattices

- **Typically, data flow facts form a lattice**
  - Example: Available expressions



Mathematical model

- **Flow values**
  - Elements of a lattice $L = (P, \leq)$
  - Flow value $v \in P$
- **Transfer functions**
  - Set of functions (one for each instruction)
  - $F_i\colon P \rightarrow P$
- **Confluence operator**
  - Merges lattice values
  - $C\colon P \times P \rightarrow P$
- **How does this help us?**

Partial Orders

- A partial order is a pair $(P, \leq)$ such that
  - $\leq \subseteq P \times P$
  - $\leq$ is reflexive: $x \leq x$
  - $\leq$ is anti-symmetric: $x \leq y$ and $y \leq x \Rightarrow x = y$
  - $\leq$ is transitive: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

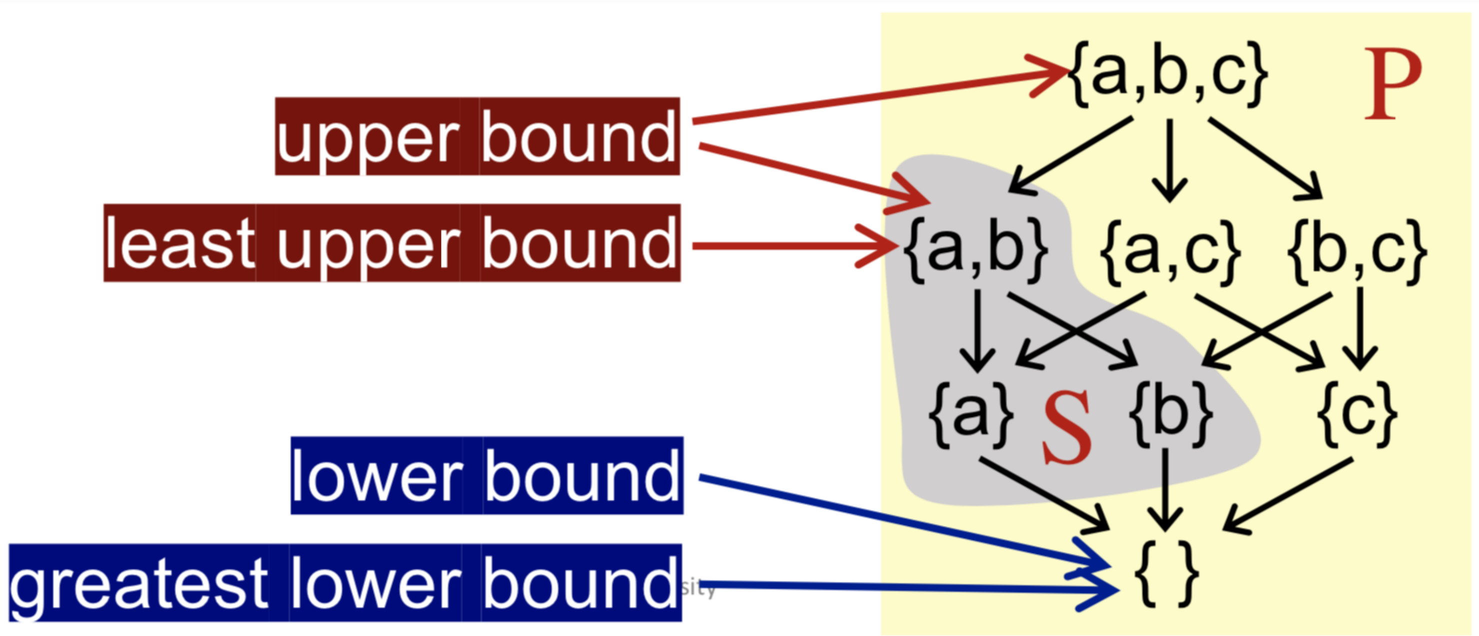
Lattices

中国科学技术大学 University of Science and Technology of China

- **A partial order is a lattice if $\sqcap$ and $\sqcup$ are defined on any set:**
  - $\sqcap$ is the *meet* or *greatest lower bound* operation:
    - $x \sqcap y \leq x$ and $x \sqcap y \leq y$ 交、最大下界
    - if $z \leq x$ and $z \leq y$, then $z \leq x \sqcap y$ 下确界
  - $\sqcup$ is the *join* or *least upper bound* operation:
    - $x \leq x \sqcup y$ and $y \leq x \sqcup y$ 并、最小上界
    - if $x \leq z$ and $y \leq z$, then $x \sqcup y \leq z$ 上确界

Lattices

- A partial order is a lattice if ⊓ and ⊔ are defined on any set:
  - ⊓ is the *meet* or *greatest lower bound* operation:



交、最大下界
下确界

并、最小上界
上确界

Lattices (cont'd)

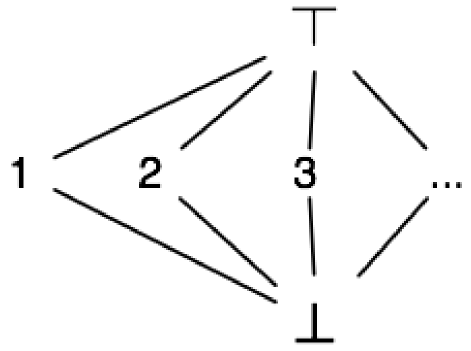
- A finite partial order is a lattice if meet and join exist for every pair of elements
- A lattice has unique elements $\perp$ and $\top$ such that
  - $x \sqcap \perp = \perp$ $\quad x \sqcup \perp = x$ 底元、 顶元
  - $x \sqcap \top = x$ $\quad x \sqcup \top = \top$
- In a lattice,

$$x \leq y \text{ iff } x \sqcap y = x$$
$$x \leq y \text{ iff } x \sqcup y = y$$

Useful Lattices

- **$(2^S, \subseteq)$ forms a lattice for any set S**
  - 2^S is the powerset of S (set of all subsets)

    幂集
- **If $(S, \leq)$ is a lattice, so is $(S, \geq)$**
  - i.e., lattices can be flipped
- **The lattice for constant propagation**



Confluence Operator

☐ Combine flow values

- "Merge" values on different control-flow paths
- Result should be a safe over-approximation
- We use the lattice ⊆ to denote "more safe"

☐ Example: live variables

- v1 = {x, y, z} and v2 = {y, w}
- How do we combine these values?
- v = v1 ∪ v2 = {w, x, y, z}
- What is the "⊆" operator?
- Superset

Meet and Join

- **Goal:** ***Combine two values to produce the "best" approximation***
  - Intuition:
    - Given v1 = {x, y, z} and v2 = {y, w}
    - A safe over-approximation is "all variables live"
    - We want the smallest set
- **Greatest lower bound**
  - Given x,y $\in$P
  - GLB(x,y) = z such that
    - $z \subseteq x$ and $z \subseteq y$ and
    - $\forall w. w \subseteq x$ and $w \subseteq y \Rightarrow w \subseteq z$
  - *Meet* operator: $x \wedge y = GLB(x, y)$
- **Natural "opposite": Least upper bound, *join* operator**

Forward Must Data Flow Algorithm

中国科学技术大学
University of Science and Technology of China

Out(s) = Top for all statements s

// Slight acceleration: Could set Out(s) = Gen(s) ∪(Top - Kill(s))

W := { all statements } (worklist)

repeat

Take s from W

In(s) := $\bigcap_{s' \in pred(s)}$ Out(s′)

temp := Gen(s) ∪ (In(s) - Kill(s))

if (temp != Out(s)) {

Out(s) := temp

W := W ∪ succ(s)

}

until W = ∅

Monotonicity单调性

- A function f on a partial order is *monotonic* if

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

- Easy to check that operations to compute In and Out are monotonic
  - $\text{In}(s) := \bigcap_{s' \in \text{pred}(s)} \text{Out}(s')$
  - temp := Gen(s) ∪ (In(s) - Kill(s))

- Putting these two together,
  - temp := $f_s(\sqcap_{s' \in \mathrm{pred}(s)} Out(s'))$

Termination终止性

- We know the algorithm terminates because
  - The lattice has finite height
  - The operations to compute In and Out are monotonic
  - On every iteration, we remove a statement from the worklist and/or move down the lattice

Forward Data Flow, Again

中国科学技术大学
University of Science and Technology of China

Out(s) = Top   for all statements s

W := { all statements }   (worklist)

repeat

    Take s from W

    temp := $f_s(\sqcap_{s' \in pred(s)} Out(s'))$ (f_s monotonic *transfer fn*)

    if (temp != Out(s)) {

        Out(s) := temp

        W := W ∪ succ(s)

    }

until W = ∅

Lattices (P, ≤)

- **Available expressions**
  - P = sets of expressions
  - S1 ⊓ S2 = S1 ∩ S2
  - Top = set of all expressions
- **Reaching Definitions**
  - P = set of definitions (assignment statements)
  - S1 ⊓ S2 = S1 ∪ S2
  - Top = empty set

Lattices (P, ≤), cont'd

- **Live variables**
  - P = sets of variables
  - S1 ⊓ S2 = S1 ∪ S2
  - Top = empty set
- **Very busy expressions**
  - P = set of expressions
  - S1 ⊓ S2 = S1 ∩ S2
  - Top = set of all expressions

Forward vs. Backward


```
Out(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(⊓_{s′ ∈ pred(s)} Out(s′))
    if (temp != Out(s)) {
        Out(s) := temp
        W := W ∪ succ(s)
    }
until W = ∅
```

```
In(s) = Top  for all s
W := { all statements }
repeat
    Take s from W
    temp := f_s(⊓_{s′ ∈ succ(s)} In(s′))
    if (temp != In(s)) {
        In(s) := temp
        W := W ∪ pred(s)
    }
until W = ∅
```

Dataflow Solution

Question:

- What is the solution we compute?
- Start at lattice top, move down
- Called greatest *fixpoint*
- Where does approximation come from?
- Confluence of control-flow paths

Ideal solution?

- Consider each path to a program point separately
- Combine values at end
- Called *meet-over-all-paths* solution (MOP)
- When is the fixpoint equal to MOP?

Fixpoints不动点

- **We always start with Top**
  - Every expression is available, no definitions reach this point
  - Most optimistic assumption
  - Strongest possible hypothesis
    - = true of fewest number of states
- **Revise as we encounter contradictions**
  - Always move down in the lattice (with meet)
- **Result: A greatest fixpoint**

Termination Revisited

How many times can we apply this step:

$$temp := f_s(\sqcap_{s' \in pred(s)} Out(s'))$$

$$if\ (temp\ != Out(s))\ \{ \ldots \}$$

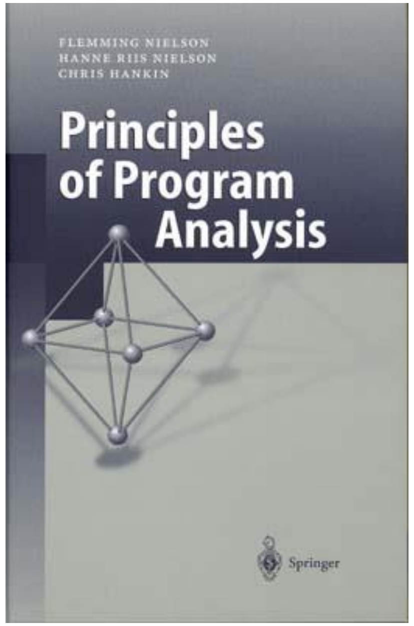
- Claim: Out(s) only shrinks
  - Proof: Out(s) starts out as top
    - So temp must be ≤ than Top after first step
  - Assume Out(s′) shrinks for all predecessors s′ of s
  - Then $\sqcap_{s' \in pred(s)}$ shrinks
  - Since f_s monotonic, $f_s(\sqcap_{s' \in pred(s)} Out(s'))$ shrinks

Termination Revisited (cont’d)

- A *descending chain* in a lattice is a sequence
  - $x0 \sqsupset x1 \sqsupset x2 \sqsupset \ldots$
- The *height* of a lattice is the length of the longest descending chain in the lattice
- Then, dataflow must terminate in O(n k) time
  - n = # of statements in program
  - k = height of lattice
  - assumes meet operation takes O(1) time

Relationship to Section 2.4 of Book (NNH)

- MFP (Maximal Fixed Point) solution – general iterative algorithm for monotone frameworks
  - always terminates
  - always computes the right solution



Flemming Nielson et al. Principles of Program Analysis (2nd Edition). Springer, 2005.

https://github.com/amilajack/reading/tree/master/Type_Systems

Least vs. Greatest Fixpoints

University of Science and Technology of China

- **Dataflow tradition: Start with Top, use meet**
  - To do this, we need a *meet semilattice with top*
  - meet semilattice = meets defined for any set
  - Computes greatest fixpoint

交半格

偏序集且a ⊓ b存在（下确界）

- **Denotational semantics tradition: Start with Bottom, use join**
  - Computes least fixpoint

Distributive Data Flow Problems

- By monotonicity, we also have

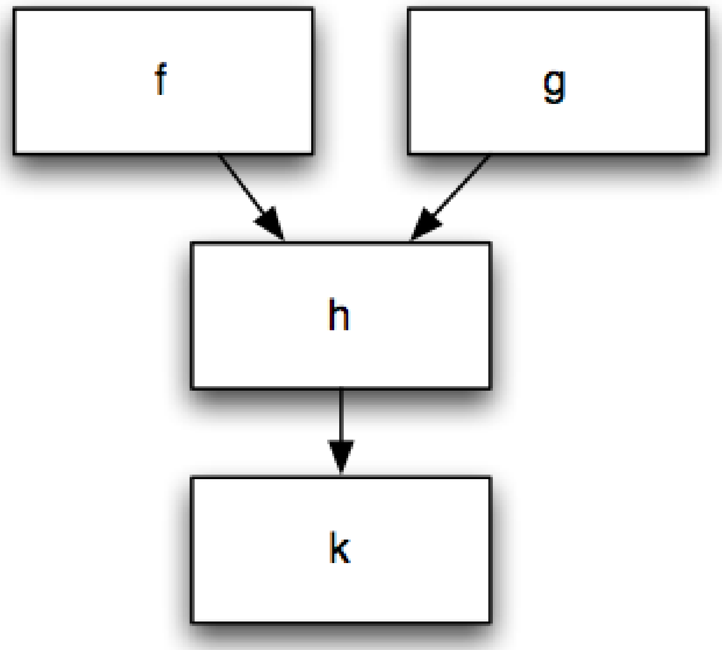
$$f(x \sqcap y) \leq f(x) \sqcap f(y)$$

- A function **f** is distributive (可分配) if

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

Benefit of Distributivity

- Joins lose no information



$$k(h(f(\top) \sqcap g(\top))) =$$
$$k(h(f(\top)) \sqcap h(g(\top))) =$$
$$k(h(f(\top))) \sqcap k(h(g(\top)))$$

Accuracy of Data Flow Analysis

- **Ideally, we would like to compute the meet over all paths (MOP) solution:**

将所有路径都join/meet的方法

  - Let f_s be the transfer function for statement s
  - If p is a path $\{s_1, \ldots, s_n\}$, let $f_p = f_n;\ldots;f_1$
  - Let path(s) be the set of paths from the entry to s

该路径上所有语句的转移函数的复合

$$\mathrm{MOP}(s) = \sqcap_{p \in \mathrm{path}(s)} f_p(\top)$$

- **If a data flow problem is distributive, then solving the data flow equations in the standard way yields the MOP solution, i.e., MFP = MOP**

MOP vs. MFP

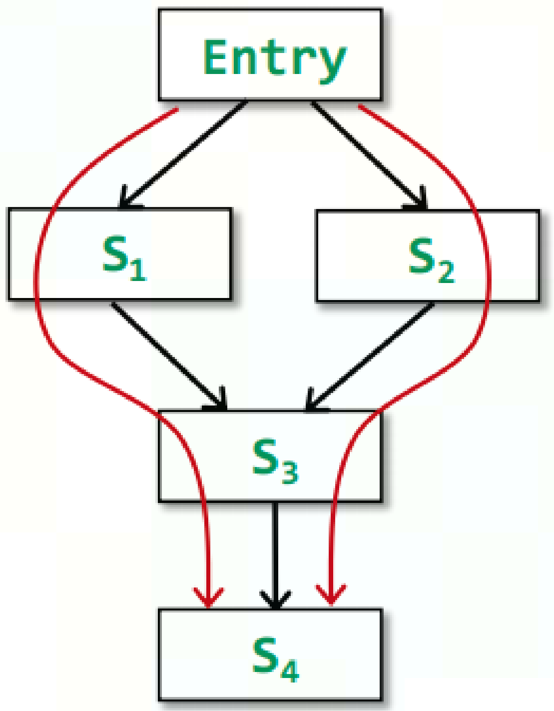
- MOP（Meet Over All paths）

$$f(x) \sqcap f(y)$$

- MFP（Maximal Fixed Point）

$$f(x \sqcap y)$$

- MFP ≤ MOP ≤ PerfectSolution

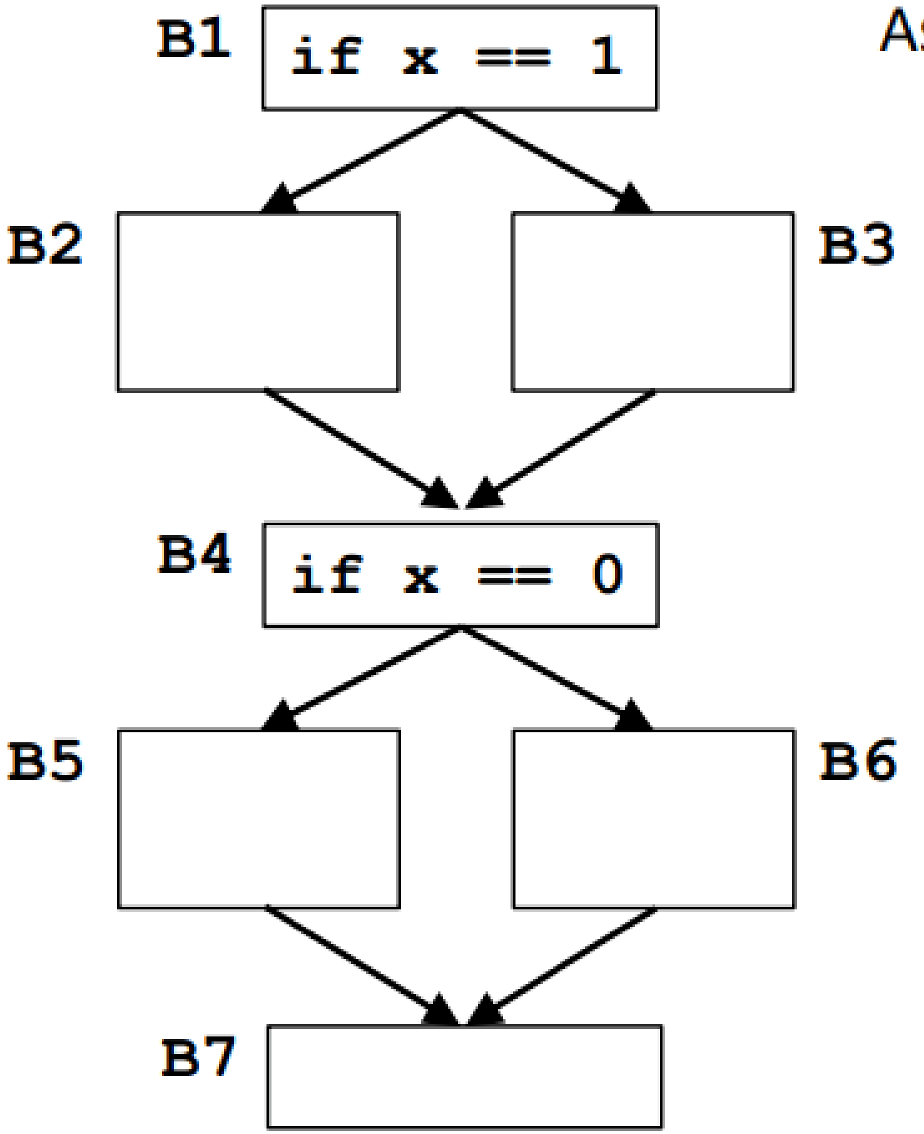


What Problems are Distributive?

- **Analyses of *how* the program computes**
  - Live variables
  - Available expressions
  - Reaching definitions
  - Very busy expressions

- **All Gen/Kill problems are distributive**

MOP considers more paths than Ideal



Assume $x \in \{0,1\}$ and B2 & B3 do not update x

Ideal considers only which 2 paths?

B1-B2-B4-B6-B7 (i.e., x=1)

B1-B3-B4-B5-B7 (i.e., x=0)

MOP: Also considers unexecuted paths

B1-B2-B4-B5-B7

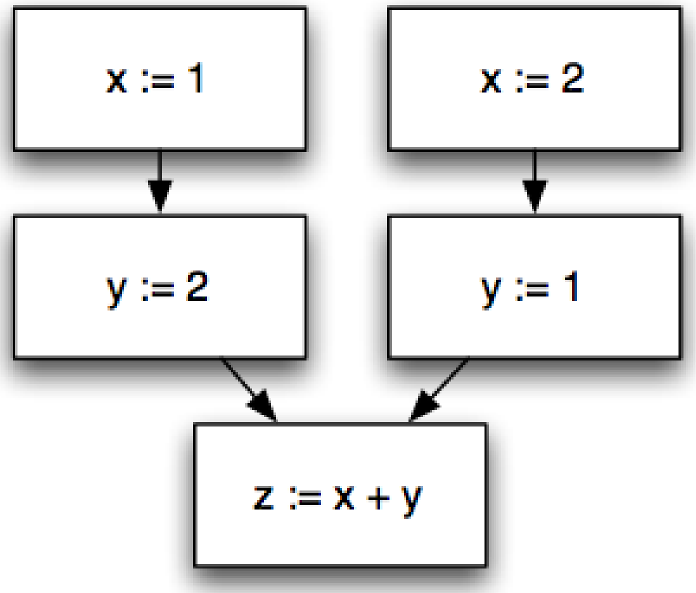
B1-B3-B4-B6-B7

What changes if $x \in \{0,1,2\}$?

B1-B3-B4-B6-B7 is also an Ideal path

A Non-Distributive Example

- Constant propagation



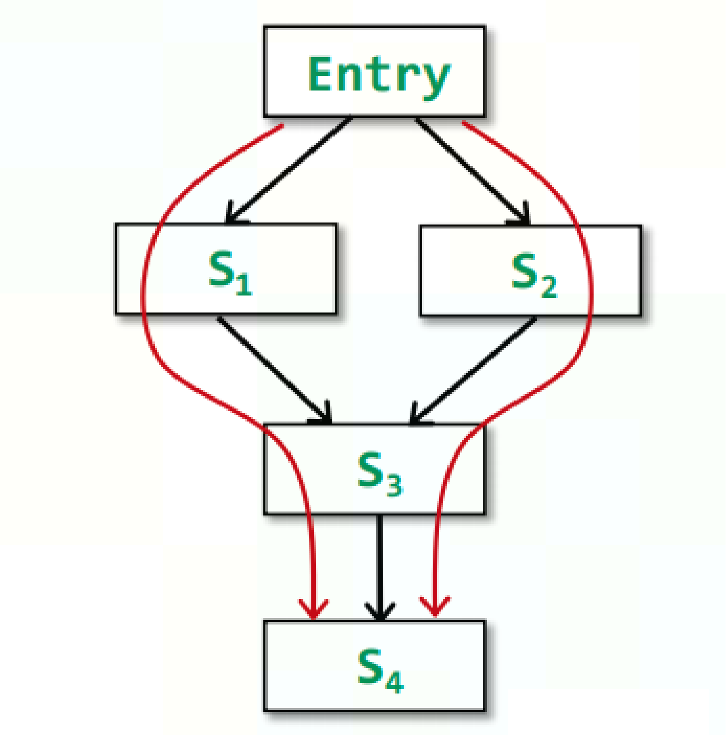
MOP：先考虑所有路径，得到两条路径z的值为3，再聚合得到z的值就是3

MFP：过早地进行交汇运算，最后并不能得到z的值是多少

- In general, analysis of *what* the program computes in not distributive

MOP vs MFP

- Computing **MFP is always safe**: MFP $\sqsubseteq$ MOP
- When distributive: MOP = MFP
- When non-distributive: MOP may not be computable (decidable)
  - e.g., MOP for constant propagation (see Lemma 2.31 of NNH)



中国科学技术大学
University of Science and Technology of China

Practical Implementation

Practical Implementation

- **Data flow facts = assertions that are true or false at a program point**

- **Represent set of facts as bit vector**
  - Fact_i represented by bit i
  - Intersection = bitwise and, union = bitwise or, etc

- **"Only" a constant factor speedup**
  - But very useful in practice

Basic Blocks

- A *basic block* is a sequence of statements s.t.
  - No statement except the last in a branch
  - There are no branches to any statement in the block except the first

- In practical data flow implementations,
  - Compute Gen/Kill for each basic block
    - Compose transfer functions
  - Store only In/Out for each basic block
  - Typical basic block ~5 statements

Order Matters

- **Assume forward data flow problem**
  - Let $G = (V, E)$ be the CFG
  - Let k be the height of the lattice
- **If G acyclic, visit in topological order**
  - Visit head before tail of edge
- **Running time $O(|E|)$**
  - No matter what size the lattice

Order Matters — Cycles

University of Science and Technology of China

- If G has cycles, visit in reverse postorder
  - Order from depth-first search
- Let Q = max # back edges on cycle-free path
  - Nesting depth
  - Back edge is from node to ancestor on DFS tree
- Then if $\forall x. f(x) \leq x$ (sufficient, but not necessary)
  - Running time is $O((Q+1)|E|)$
    - Note direction of req't depends on top vs. bottom

Flow-Sensitivity

- Data flow analysis is *flow-sensitive*
  - The order of statements is taken into account
  - I.e., we keep track of facts per program point

- Alternative: *Flow-insensitive* analysis
  - Analysis the same regardless of statement order
  - Standard example: types
    - /* x : int */ x := ... /* x : int */

Terminology Review

- Must vs. May
  - (Not always followed in literature)
- Forwards vs. Backwards
- Flow-sensitive vs. Flow-insensitive
- Distributive vs. Non-distributive

Another Approach: Elimination

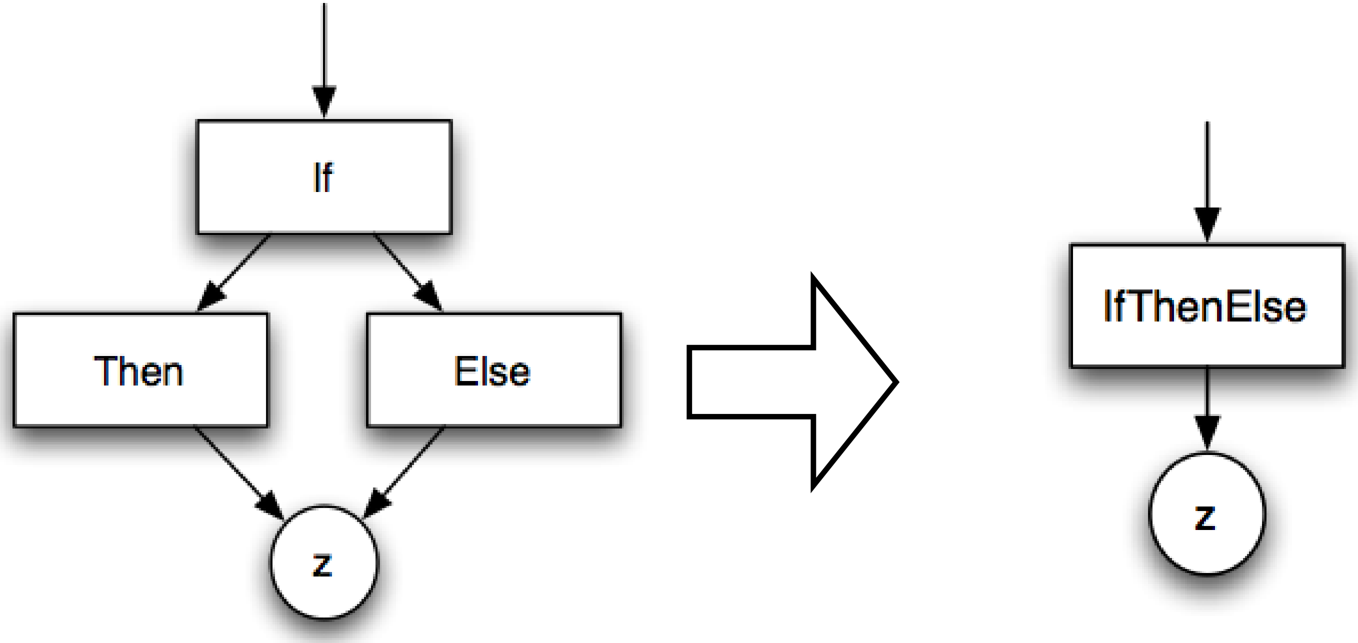
- Recall in practice, one transfer function per basic block

- Why not generalize this idea beyond a basic block?
  - "Collapse" larger constructs into smaller ones, combining data flow equations
  - Eventually program collapsed into a single node!
  - "Expand out" back to original constructs, rebuilding information

Lattices of Functions

- Let $(P, \leq)$ be a lattice
- Let M be the set of monotonic functions on P
- Define $f \leq_f g$ if for all x, $f(x) \leq g(x)$
- Define the function $f \sqcap g$ as
  - $(f \sqcap g)(x) = f(x) \sqcap g(x)$
- Claim: $(M, \leq_f)$ forms a lattice

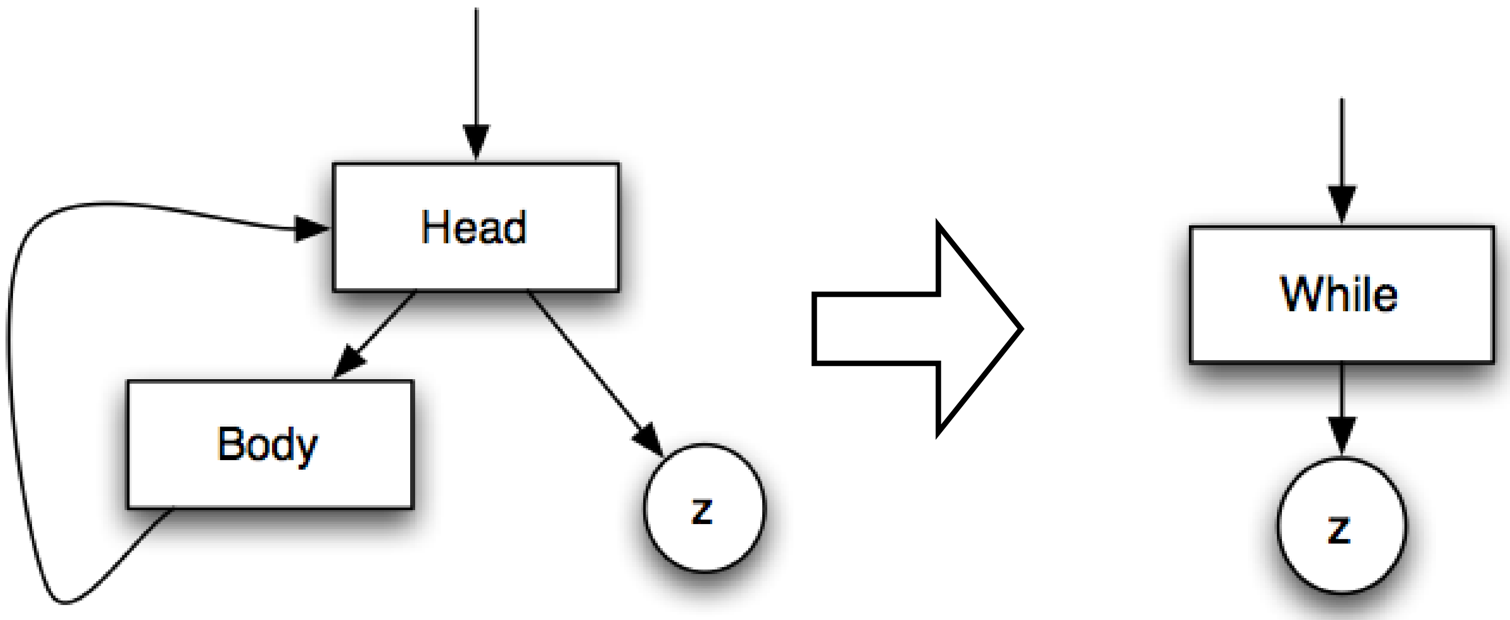
Conditionals



$$f_{\text{ite}} = (f_{\text{then}} \circ f_{\text{if}}) \sqcap (f_{\text{else}} \circ f_{\text{if}})$$

$$\begin{aligned}
\text{Out(if)} &= f_{\text{if}}(\text{In(ite)}) \\
\text{Out(then)} &= (f_{\text{then}} \circ f_{\text{if}})(\text{In(ite)}) \\
\text{Out(else)} &= (f_{\text{else}} \circ f_{\text{if}})(\text{In(ite)})
\end{aligned}$$

Elimination Methods: Loops



$$
\begin{aligned}
f_{\text{while}} \;=\; & f_{\text{head}} \sqcap \\
& f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \\
& f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \circ f_{\text{body}} \circ f_{\text{head}} \sqcap \cdots
\end{aligned}
$$

Elimination Methods: Loops (cont'd)

- Let $f^i = f \circ f \circ \cdots \circ f$ (i times)
  - $f^0 = \text{id}$
- Let
$$g(j) = \sqcap_{i \in [0..j]} (f_{\text{head}} \circ f_{\text{body}})^i \circ f_{\text{head}}$$
- **Need to compute limit as j goes to infinity**
  - Does such a thing exist?
- Observe: $g(j+1) \leq g(j)$

Height of Function Lattice

- **Assume underlying lattice $(P, \leq)$ has finite height**
  - What is height of lattice of monotonic functions?
  - Claim: finite
- Therefore, $g(j)$ converges

Non-Reducible Flow Graphs

- **Elimination methods usually only applied to *reducible* flow graphs**
  - Ones that can be collapsed
  - Standard constructs yield only reducible flow graphs

- **Unrestricted goto can yield non-reducible graphs**

Comments

- **Can also do backwards elimination**
  - Not quite as nice (regions are usually single *entry* but often not single *exit*)
- **For bit-vector problems, elimination efficient**
  - Easy to compose functions, compute meet, etc.
- **Elimination originally seemed like it might be faster than iteration**
  - Not really the case

Data Flow Analysis and Functions

- **What happens at a function call?**
  - Lots of proposed solutions in data flow analysis literature

- **In practice, only analyze one procedure at a time**

- **Consequences**
  - Call to function kills all data flow facts
  - May be able to improve depending on language, e.g., function call may not affect locals

More Terminology

- An analysis that models only a single function at a time is *intraprocedural*
- An analysis that takes multiple functions into account is *interprocedural*
- An analysis that takes the whole program into account is...guess?

- Note: *global* analysis means “more than one basic block,” but still within a function

Data Flow Analysis and The Heap

- **Data Flow is good at analyzing local variables**
  - But what about values stored in the heap?
  - Not modeled in traditional data flow
- **In practice: *x := e**
  - Assume all data flow facts killed (!)
  - Or, assume write through x may affect any variable whose address has been taken
- **In general, hard to analyze pointers**

- **Moore's Law: Hardware advances double computing power every 18 months.**

- **Proebsting's Law: Compiler advances double computing power every 18 *years*.**
  编译器优化每18年提高一倍的计算能力
  - https://proebsting.cs.arizona.edu/law.html

虽然硬件计算能力每年增长约60%，
但编译器优化仅贡献4%。基本上，编译器优化工作仅做出很小的贡献。
也许这意味着编程语言研究应该专注于优化以外的事情。
也许**程序员的生产力**是一个更加富有成效的舞台。

THANKS