Foundations of Dataflow Analysis

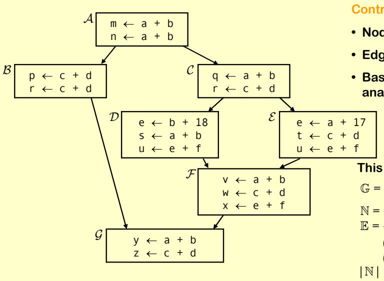
This lecture is primarily based on Konstantinos Sagonas set of slides
(**Advanced Compiler Techniques**, (2AD518)
at Uppsala University, January-February 2004).
Used with kind permission.

Terminology: Program Representation

Control Flow Graph (CFG):

- Nodes *N* – statements of program
- Edges *E* – flow of control
  - *pred(n)* = set of all immediate predecessors of *n*
  - *succ(n)* = set of all immediate successors of *n*
- Start node n_0
- Set of final nodes N_{final}

Terminology: Control-Flow Graph



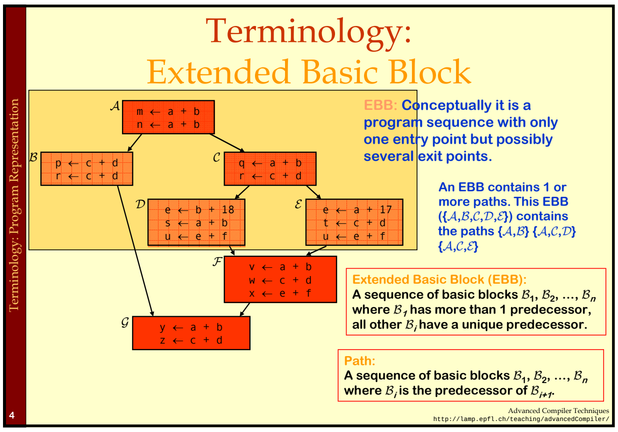
Control-flow graph (CFG)

- **Nodes for basic blocks**
- **Edges for branches**
- **Basis for much of program analysis & transformation**

This CFG,

$\mathbb{G} = (\mathbb{N},\mathbb{E})$

$\mathbb{N} = \{\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}\}$
$\mathbb{E} = \{(\mathcal{A}, \mathcal{B}), (\mathcal{A}, \mathcal{C}), (\mathcal{B}, \mathcal{G}), (\mathcal{C}, \mathcal{D}), (\mathcal{C}, \mathcal{E}), (\mathcal{D}, \mathcal{F}), (\mathcal{E}, \mathcal{F}),(\mathcal{F}, \mathcal{G})\}$
$|\mathbb{N}| = 7$
$|\mathbb{E}| = 8$



Terminology: Program Representation

Terminology:
Extended Basic Block

EBB: Conceptually it is a program sequence with only one entry point but possibly several exit points.

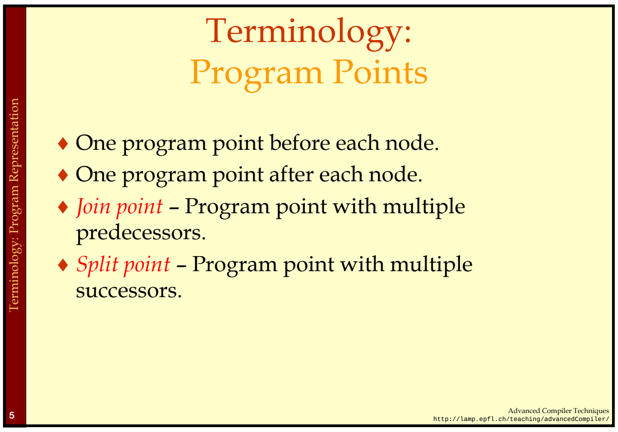
An EBB contains 1 or more paths. This EBB ({$\mathcal{A}$,$\mathcal{B}$,$\mathcal{C}$,$\mathcal{D}$,$\mathcal{E}$}) contains the paths {$\mathcal{A}$,$\mathcal{B}$} {$\mathcal{A}$,$\mathcal{C}$,$\mathcal{D}$} {$\mathcal{A}$,$\mathcal{C}$,$\mathcal{E}$}

Extended Basic Block (EBB):
A sequence of basic blocks $\mathcal{B}_1$, $\mathcal{B}_2$, …, $\mathcal{B}_n$ where $\mathcal{B}_1$ has more than 1 predecessor, all other $\mathcal{B}_i$ have a unique predecessor.

Path:
A sequence of basic blocks $\mathcal{B}_1$, $\mathcal{B}_2$, …, $\mathcal{B}_n$ where $\mathcal{B}_i$ is the predecessor of $\mathcal{B}_{i+1}$.

4

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



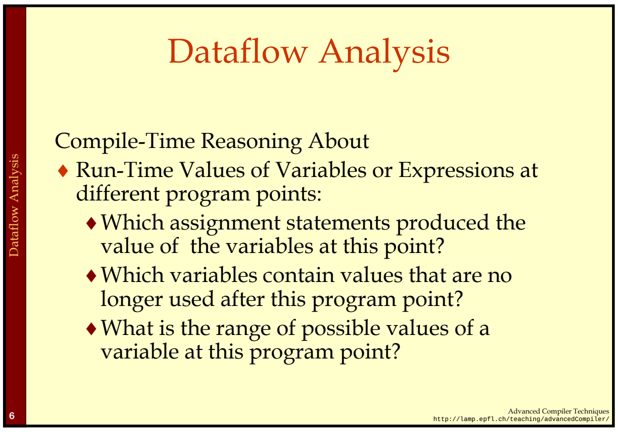
Terminology: Program Representation

Terminology:
Program Points

- One program point before each node.
- One program point after each node.
- *Join point* – Program point with multiple predecessors.
- *Split point* – Program point with multiple successors.

5

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



Dataflow Analysis

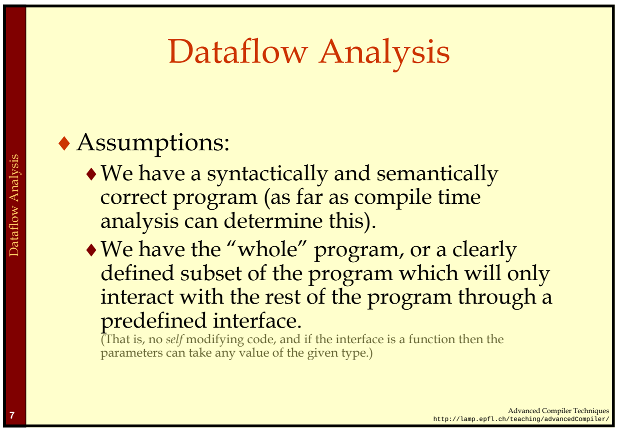
Dataflow Analysis

Compile-Time Reasoning About

- Run-Time Values of Variables or Expressions at different program points:
  - Which assignment statements produced the value of the variables at this point?
  - Which variables contain values that are no longer used after this program point?
  - What is the range of possible values of a variable at this program point?

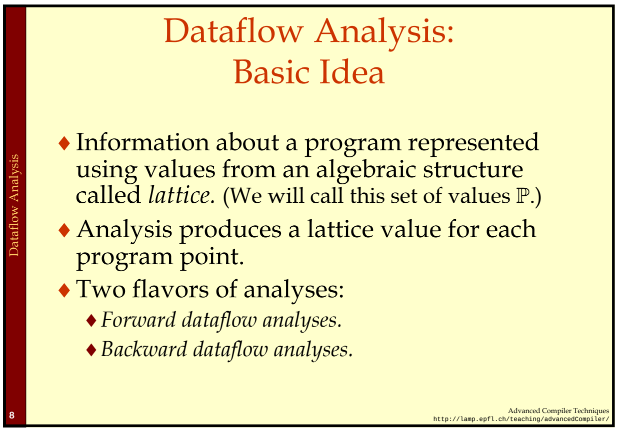
6

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



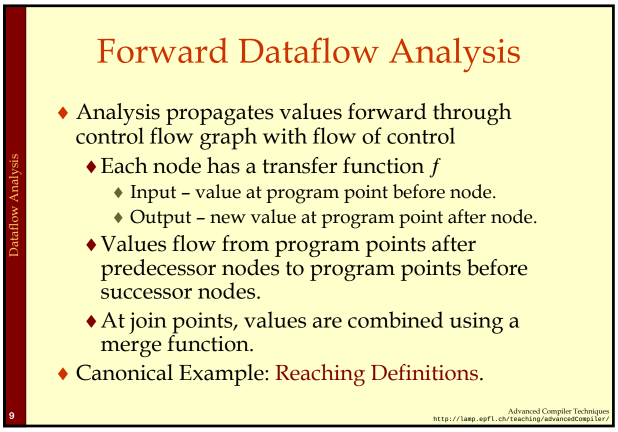
Dataflow Analysis

- Assumptions:
  - We have a syntactically and semantically correct program (as far as compile time analysis can determine this).
  - We have the “whole” program, or a clearly defined subset of the program which will only interact with the rest of the program through a predefined interface.
(That is, no *self* modifying code, and if the interface is a function then the parameters can take any value of the given type.)



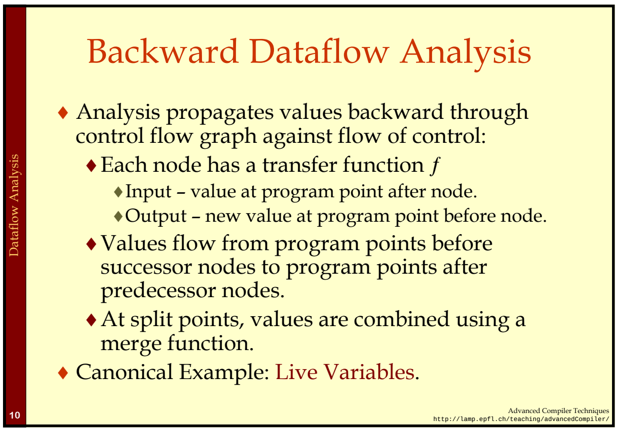
Dataflow Analysis: Basic Idea

- Information about a program represented using values from an algebraic structure called *lattice*. (We will call this set of values $\mathbb{P}$.)
- Analysis produces a lattice value for each program point.
- Two flavors of analyses:
  - *Forward dataflow analyses.*
  - *Backward dataflow analyses.*



Forward Dataflow Analysis

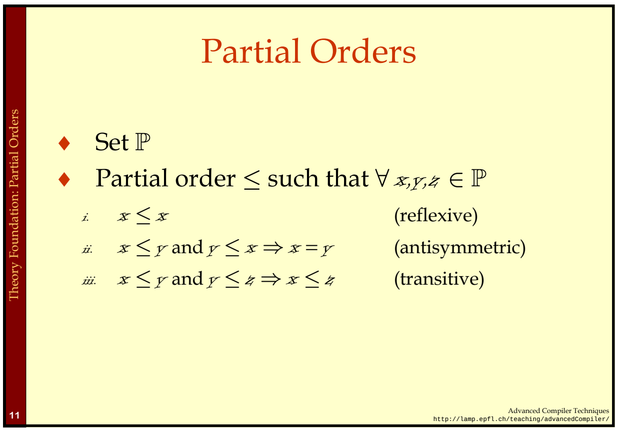
- Analysis propagates values forward through control flow graph with flow of control
  - Each node has a transfer function f
    - Input – value at program point before node.
    - Output – new value at program point after node.
  - Values flow from program points after predecessor nodes to program points before successor nodes.
  - At join points, values are combined using a merge function.
- Canonical Example: Reaching Definitions.



Dataflow Analysis

Backward Dataflow Analysis

- Analysis propagates values backward through control flow graph against flow of control:
  - Each node has a transfer function *f*
    - Input – value at program point after node.
    - Output – new value at program point before node.
  - Values flow from program points before successor nodes to program points after predecessor nodes.
  - At split points, values are combined using a merge function.
- Canonical Example: Live Variables.



Theory Foundation: Partial Orders

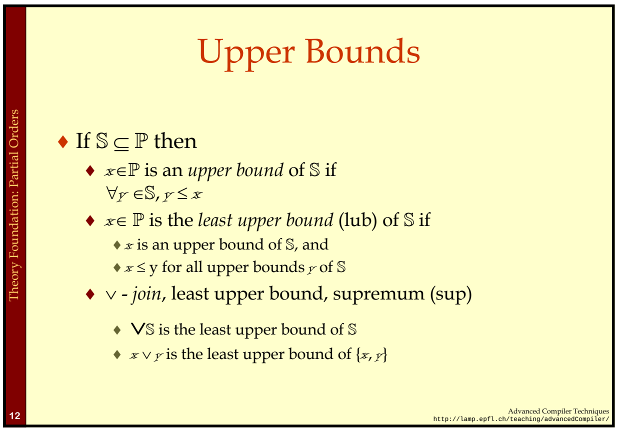
Partial Orders

- Set $\mathbb{P}$
- Partial order $\leq$ such that $\forall\, x,y,z \in \mathbb{P}$

i. $x \leq x$ (reflexive)

ii. $x \leq y$ and $y \leq x \Rightarrow x = y$ (antisymmetric)

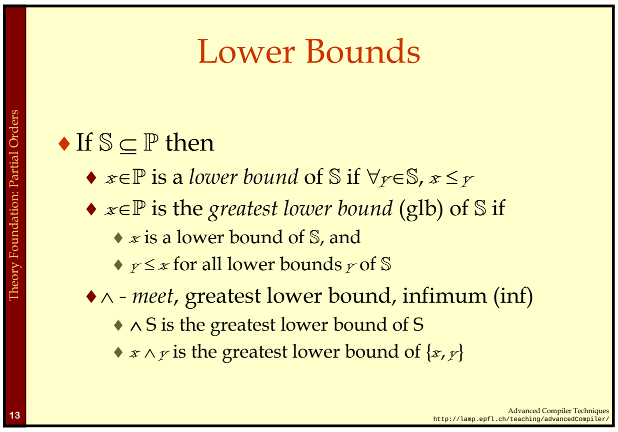
iii. $x \leq y$ and $y \leq z \Rightarrow x \leq z$ (transitive)



Theory Foundation: Partial Orders

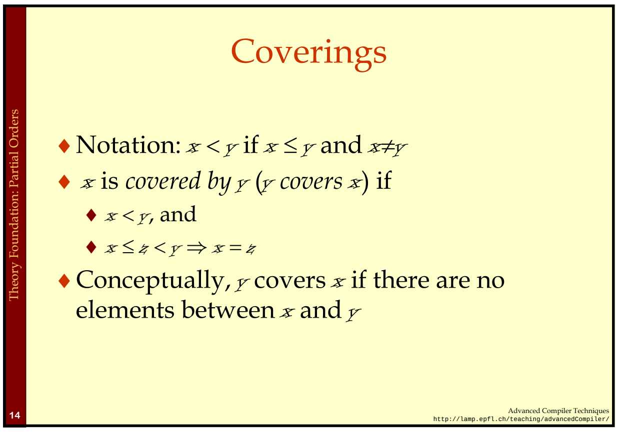
Upper Bounds

- If $\mathbb{S} \subseteq \mathbb{P}$ then
  - $x \in \mathbb{P}$ is an *upper bound* of $\mathbb{S}$ if $\forall y \in \mathbb{S}, y \leq x$
  - $x \in \mathbb{P}$ is the *least upper bound* (lub) of $\mathbb{S}$ if
    - x is an upper bound of $\mathbb{S}$, and
    - $x \leq$ y for all upper bounds y of $\mathbb{S}$
  - $\vee$ - *join*, least upper bound, supremum (sup)
    - $\bigvee\mathbb{S}$ is the least upper bound of $\mathbb{S}$
    - $x \vee y$ is the least upper bound of $\{x, y\}$



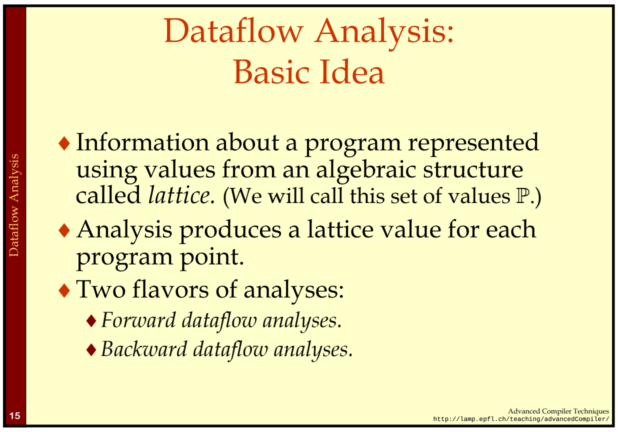
Lower Bounds

- If $\mathbb{S} \subseteq \mathbb{P}$ then
  - $x \in \mathbb{P}$ is a *lower bound* of $\mathbb{S}$ if $\forall y \in \mathbb{S}$, $x \leq y$
  - $x \in \mathbb{P}$ is the *greatest lower bound* (glb) of $\mathbb{S}$ if
    - x is a lower bound of $\mathbb{S}$, and
    - $y \leq x$ for all lower bounds y of $\mathbb{S}$
  - $\wedge$ - *meet*, greatest lower bound, infimum (inf)
    - $\wedge$ S is the greatest lower bound of S
    - $x \wedge y$ is the greatest lower bound of $\{x, y\}$



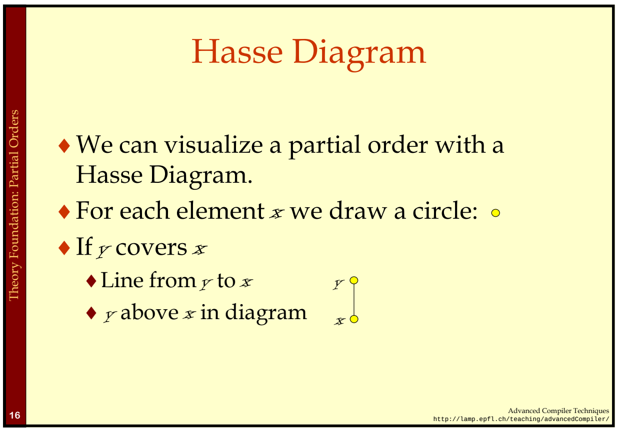
Coverings

- Notation: $x < y$ if $x \leq y$ and $x \neq y$
- x is *covered by* y (y *covers* x) if
  - $x < y$, and
  - $x \leq z < y \Rightarrow x = z$
- Conceptually, y covers x if there are no elements between x and y



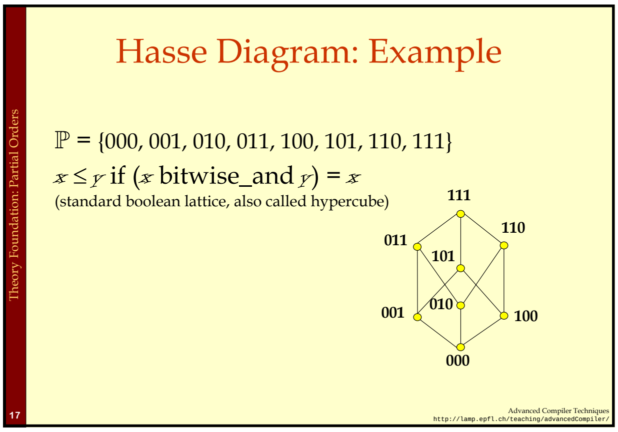
Dataflow Analysis: Basic Idea

- Information about a program represented using values from an algebraic structure called *lattice.* (We will call this set of values $\mathbb{P}$.)
- Analysis produces a lattice value for each program point.
- Two flavors of analyses:
  - *Forward dataflow analyses.*
  - *Backward dataflow analyses.*



Hasse Diagram

- We can visualize a partial order with a Hasse Diagram.
- For each element x we draw a circle: ○
- If y covers x
  - Line from y to x
  - y above x in diagram

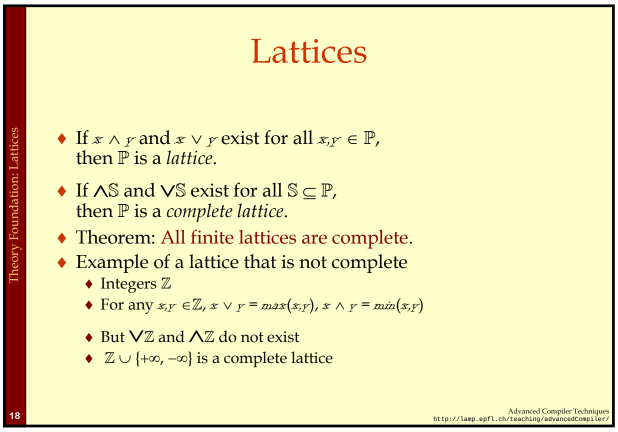


Hasse Diagram: Example

$\mathbb{P}$ = {000, 001, 010, 011, 100, 101, 110, 111}

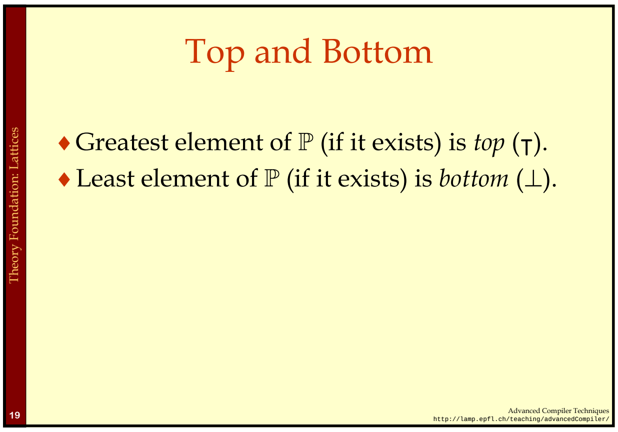
$x \leq y$ if (x bitwise_and y) = x

(standard boolean lattice, also called hypercube)



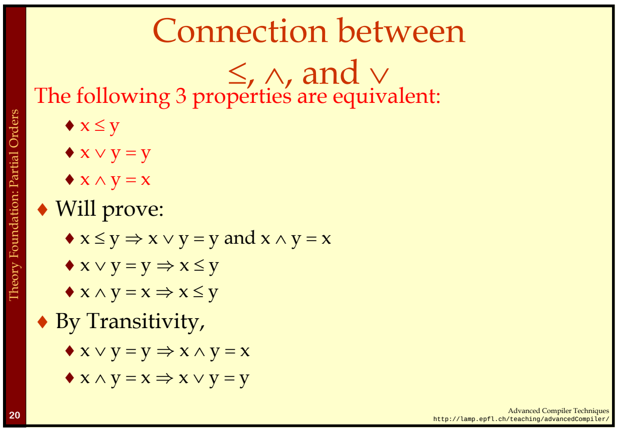
Lattices

- If $x \wedge y$ and $x \vee y$ exist for all $x,y \in \mathbb{P}$, then $\mathbb{P}$ is a *lattice*.
- If $\bigwedge\mathbb{S}$ and $\bigvee\mathbb{S}$ exist for all $\mathbb{S} \subseteq \mathbb{P}$, then $\mathbb{P}$ is a *complete lattice*.
- Theorem: All finite lattices are complete.
- Example of a lattice that is not complete
  - Integers $\mathbb{Z}$
  - For any $x,y \in \mathbb{Z}$, $x \vee y = max(x,y)$, $x \wedge y = min(x,y)$
  - But $\bigvee\mathbb{Z}$ and $\bigwedge\mathbb{Z}$ do not exist
  - $\mathbb{Z} \cup \{+\infty, -\infty\}$ is a complete lattice



Top and Bottom

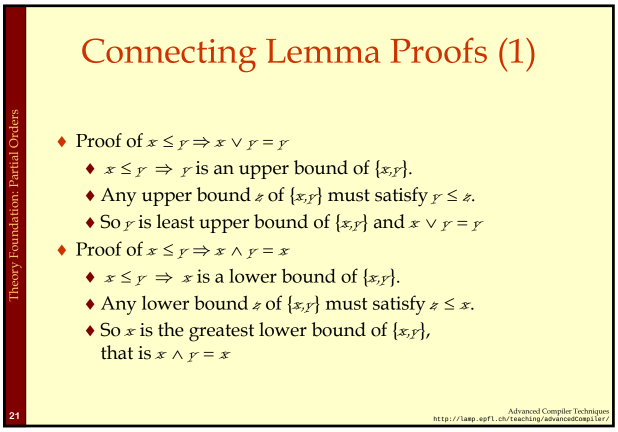
- Greatest element of $\mathbb{P}$ (if it exists) is *top* ($\top$).
- Least element of $\mathbb{P}$ (if it exists) is *bottom* ($\bot$).



Connection between $\leq$, $\wedge$, and $\vee$

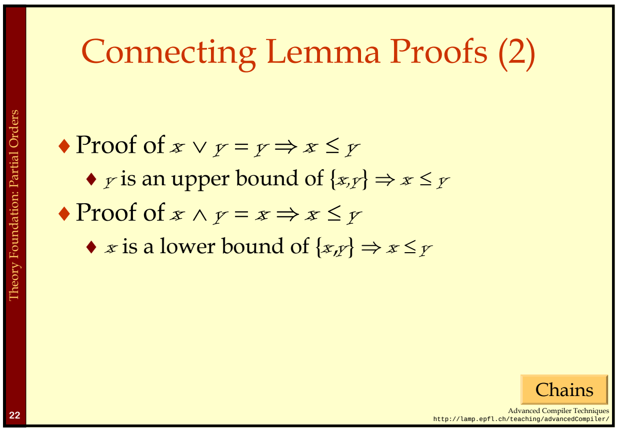
The following 3 properties are equivalent:

  - $x \leq y$
  - $x \vee y = y$
  - $x \wedge y = x$
- Will prove:
  - $x \leq y \Rightarrow x \vee y = y$ and $x \wedge y = x$
  - $x \vee y = y \Rightarrow x \leq y$
  - $x \wedge y = x \Rightarrow x \leq y$
- By Transitivity,
  - $x \vee y = y \Rightarrow x \wedge y = x$
  - $x \wedge y = x \Rightarrow x \vee y = y$



Connecting Lemma Proofs (1)

- Proof of $x \leq y \Rightarrow x \vee y = y$
  - $x \leq y \Rightarrow y$ is an upper bound of $\{x,y\}$.
  - Any upper bound z of $\{x,y\}$ must satisfy $y \leq z$.
  - So y is least upper bound of $\{x,y\}$ and $x \vee y = y$
- Proof of $x \leq y \Rightarrow x \wedge y = x$
  - $x \leq y \Rightarrow x$ is a lower bound of $\{x,y\}$.
  - Any lower bound z of $\{x,y\}$ must satisfy $z \leq x$.
  - So x is the greatest lower bound of $\{x,y\}$, that is $x \wedge y = x$

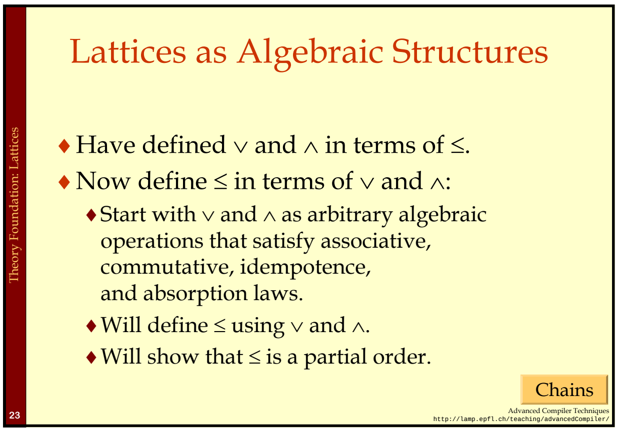


Theory Foundation: Partial Orders

Connecting Lemma Proofs (2)

- Proof of $x \vee y = y \Rightarrow x \leq y$
  - y is an upper bound of $\{x,y\} \Rightarrow x \leq y$
- Proof of $x \wedge y = x \Rightarrow x \leq y$
  - x is a lower bound of $\{x,y\} \Rightarrow x \leq y$

Chains

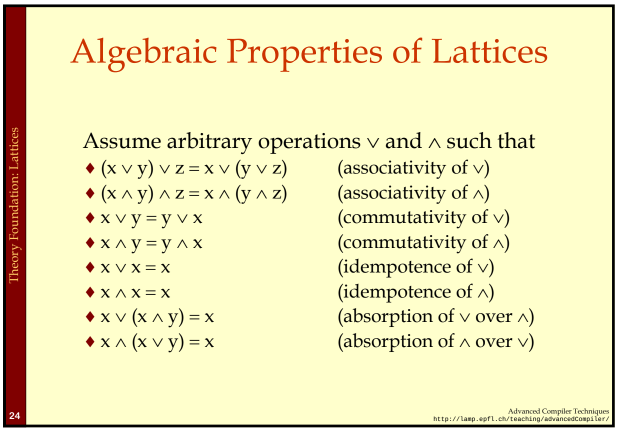


Theory Foundation: Lattices

Lattices as Algebraic Structures

- Have defined $\vee$ and $\wedge$ in terms of $\leq$.
- Now define $\leq$ in terms of $\vee$ and $\wedge$:
  - Start with $\vee$ and $\wedge$ as arbitrary algebraic operations that satisfy associative, commutative, idempotence, and absorption laws.
  - Will define $\leq$ using $\vee$ and $\wedge$.
  - Will show that $\leq$ is a partial order.

Chains

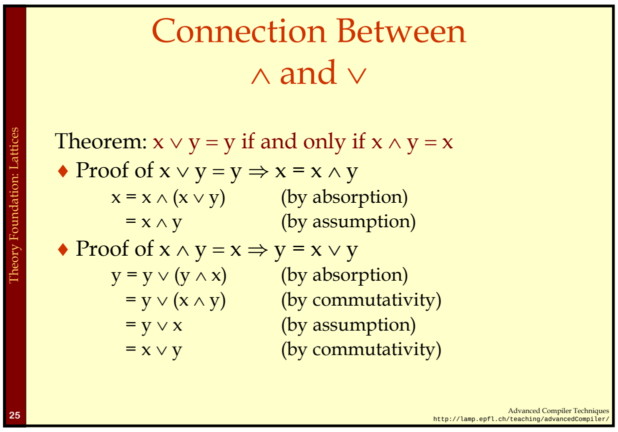


Theory Foundation: Lattices

Algebraic Properties of Lattices

Assume arbitrary operations $\vee$ and $\wedge$ such that

♦ $(x \vee y) \vee z = x \vee (y \vee z)$	(associativity of $\vee$)
♦ $(x \wedge y) \wedge z = x \wedge (y \wedge z)$	(associativity of $\wedge$)
♦ $x \vee y = y \vee x$	(commutativity of $\vee$)
♦ $x \wedge y = y \wedge x$	(commutativity of $\wedge$)
♦ $x \vee x = x$	(idempotence of $\vee$)
♦ $x \wedge x = x$	(idempotence of $\wedge$)
♦ $x \vee (x \wedge y) = x$	(absorption of $\vee$ over $\wedge$)
♦ $x \wedge (x \vee y) = x$	(absorption of $\wedge$ over $\vee$)



Connection Between $\wedge$ and $\vee$

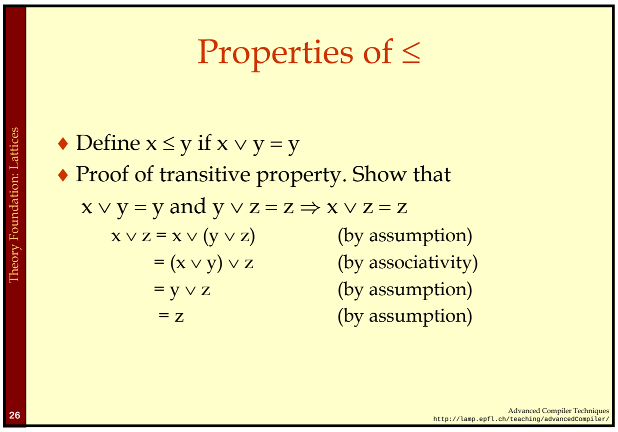
Theorem: $x \vee y = y$ if and only if $x \wedge y = x$

- Proof of $x \vee y = y \Rightarrow x = x \wedge y$

$x = x \wedge (x \vee y)$ (by absorption)
$= x \wedge y$ (by assumption)

- Proof of $x \wedge y = x \Rightarrow y = x \vee y$

$y = y \vee (y \wedge x)$ (by absorption)
$= y \vee (x \wedge y)$ (by commutativity)
$= y \vee x$ (by assumption)
$= x \vee y$ (by commutativity)

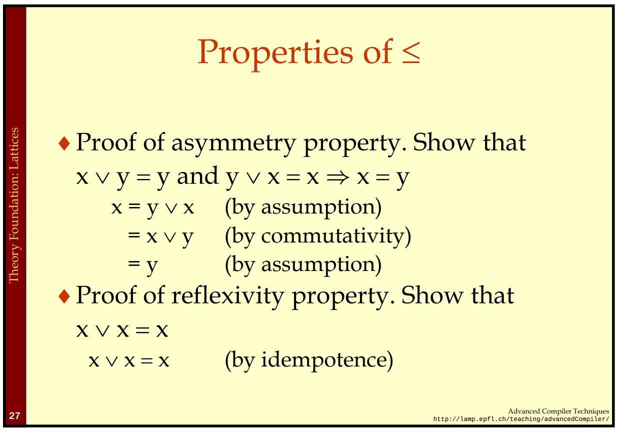


Properties of $\leq$

- Define $x \leq y$ if $x \vee y = y$
- Proof of transitive property. Show that

$x \vee y = y$ and $y \vee z = z \Rightarrow x \vee z = z$

$x \vee z = x \vee (y \vee z)$ (by assumption)
$= (x \vee y) \vee z$ (by associativity)
$= y \vee z$ (by assumption)
$= z$ (by assumption)



Properties of $\leq$

- Proof of asymmetry property. Show that

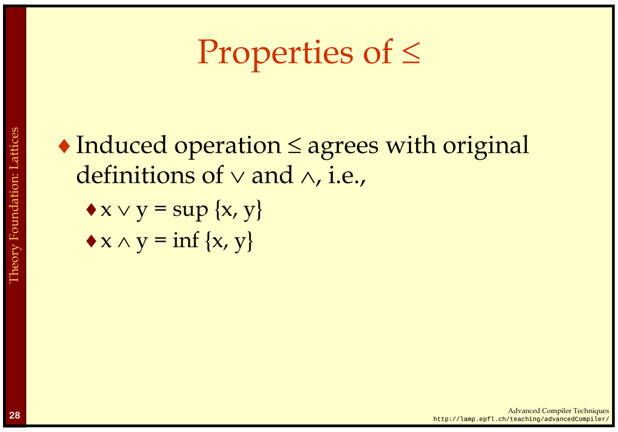
$x \vee y = y$ and $y \vee x = x \Rightarrow x = y$

$x = y \vee x$ (by assumption)
$= x \vee y$ (by commutativity)
$= y$ (by assumption)

- Proof of reflexivity property. Show that

$x \vee x = x$

$x \vee x = x$ (by idempotence)



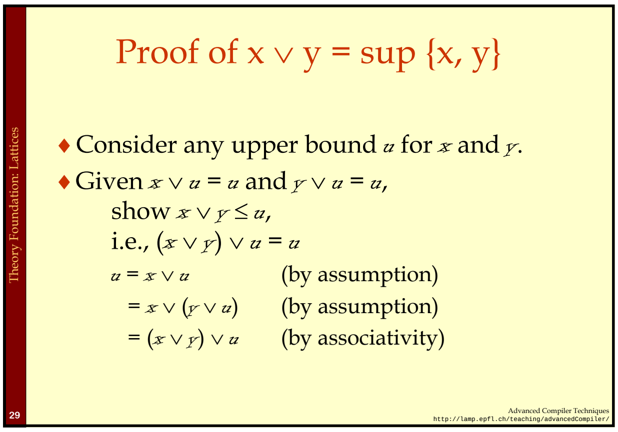
Theory Foundation: Lattices

Properties of ≤

- Induced operation ≤ agrees with original definitions of ∨ and ∧, i.e.,
  - $x \vee y = \sup \{x, y\}$
  - $x \wedge y = \inf \{x, y\}$

28

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



Theory Foundation: Lattices

Proof of x ∨ y = sup {x, y}

- Consider any upper bound u for x and y.
- Given $x \vee u = u$ and $y \vee u = u$,

  show $x \vee y \leq u$,

  i.e., $(x \vee y) \vee u = u$

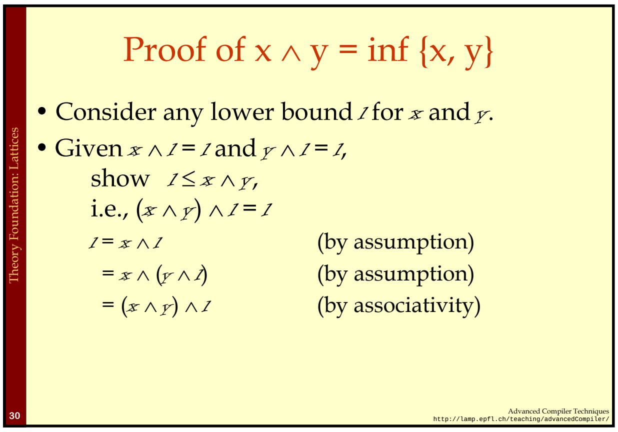
  $u = x \vee u$ (by assumption)

  $= x \vee (y \vee u)$ (by assumption)

  $= (x \vee y) \vee u$ (by associativity)

29

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



Theory Foundation: Lattices

Proof of x ∧ y = inf {x, y}

- Consider any lower bound l for x and y.
- Given $x \wedge l = l$ and $y \wedge l = l$,

  show $l \leq x \wedge y$,

  i.e., $(x \wedge y) \wedge l = l$

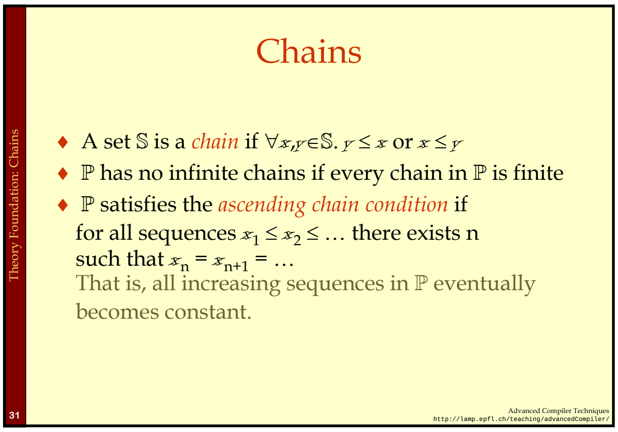
  $l = x \wedge l$ (by assumption)

  $= x \wedge (y \wedge l)$ (by assumption)

  $= (x \wedge y) \wedge l$ (by associativity)

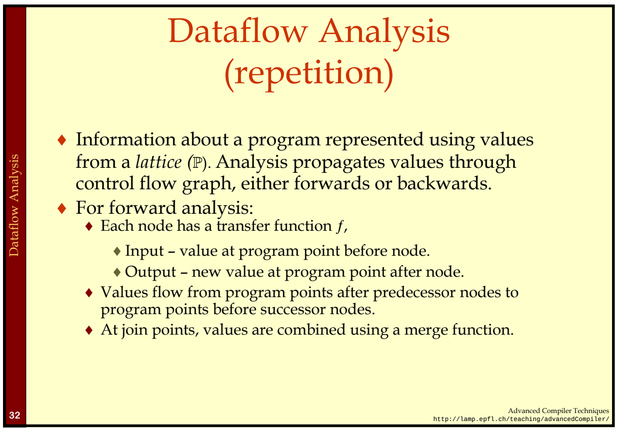
30

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



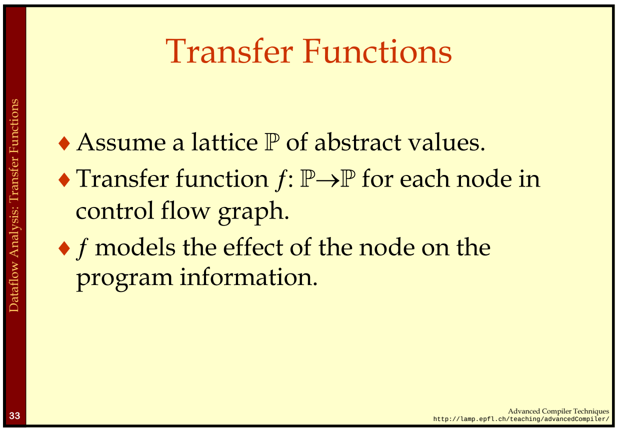
Chains

- A set $\mathbb{S}$ is a *chain* if $\forall x,y \in \mathbb{S}.\ y \le x$ or $x \le y$
- $\mathbb{P}$ has no infinite chains if every chain in $\mathbb{P}$ is finite
- $\mathbb{P}$ satisfies the *ascending chain condition* if for all sequences $x_1 \le x_2 \le \ldots$ there exists n such that $x_n = x_{n+1} = \ldots$
  That is, all increasing sequences in $\mathbb{P}$ eventually becomes constant.



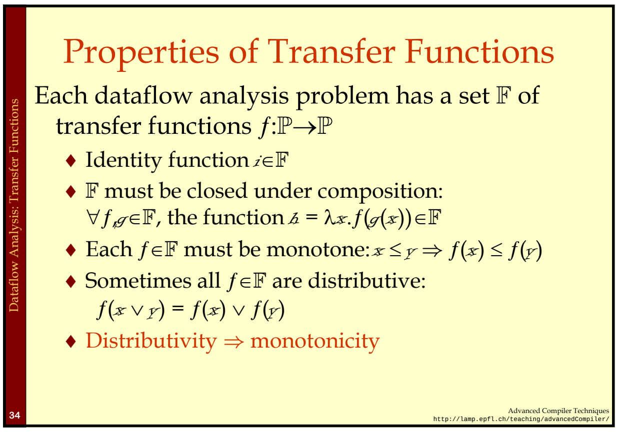
Dataflow Analysis (repetition)

- Information about a program represented using values from a *lattice* ($\mathbb{P}$). Analysis propagates values through control flow graph, either forwards or backwards.
- For forward analysis:
  - Each node has a transfer function f,
    - Input – value at program point before node.
    - Output – new value at program point after node.
  - Values flow from program points after predecessor nodes to program points before successor nodes.
  - At join points, values are combined using a merge function.



Transfer Functions

- Assume a lattice $\mathbb{P}$ of abstract values.
- Transfer function $f: \mathbb{P} \to \mathbb{P}$ for each node in control flow graph.
- f models the effect of the node on the program information.

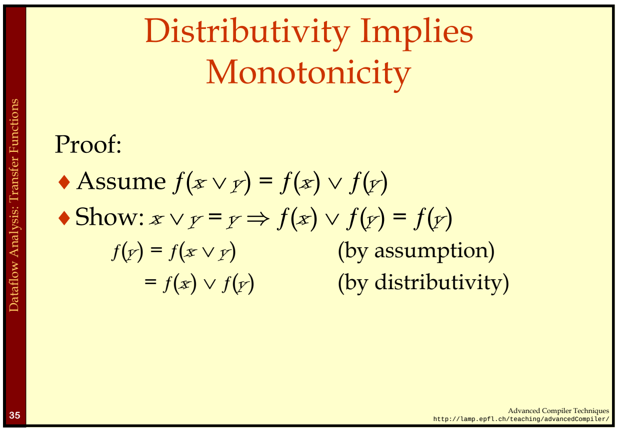


Dataflow Analysis: Transfer Functions

Properties of Transfer Functions

Each dataflow analysis problem has a set $\mathbb{F}$ of transfer functions $f:\mathbb{P}\rightarrow\mathbb{P}$

- Identity function $i\in\mathbb{F}$
- $\mathbb{F}$ must be closed under composition: $\forall f,g\in\mathbb{F}$, the function $h = \lambda x.f(g(x))\in\mathbb{F}$
- Each $f\in\mathbb{F}$ must be monotone: $x \le y \Rightarrow f(x) \le f(y)$
- Sometimes all $f\in\mathbb{F}$ are distributive: $f(x \vee y) = f(x) \vee f(y)$
- Distributivity $\Rightarrow$ monotonicity



Dataflow Analysis: Transfer Functions

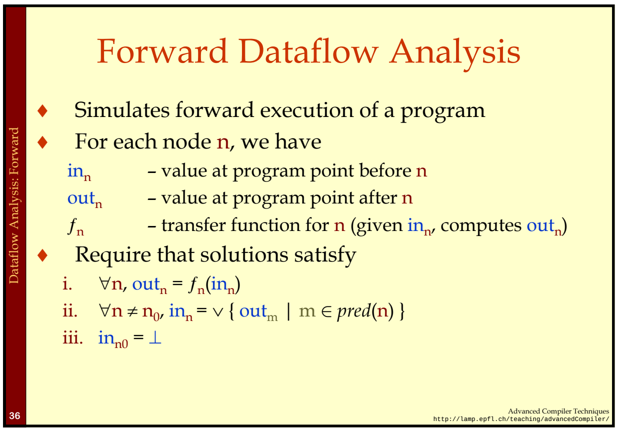
Distributivity Implies Monotonicity

Proof:

- Assume $f(x \vee y) = f(x) \vee f(y)$
- Show: $x \vee y = y \Rightarrow f(x) \vee f(y) = f(y)$

$f(y) = f(x \vee y)$ (by assumption)

$= f(x) \vee f(y)$ (by distributivity)



Dataflow Analysis: Forward

Forward Dataflow Analysis

- Simulates forward execution of a program
- For each node n, we have

in_n – value at program point before n

out_n – value at program point after n

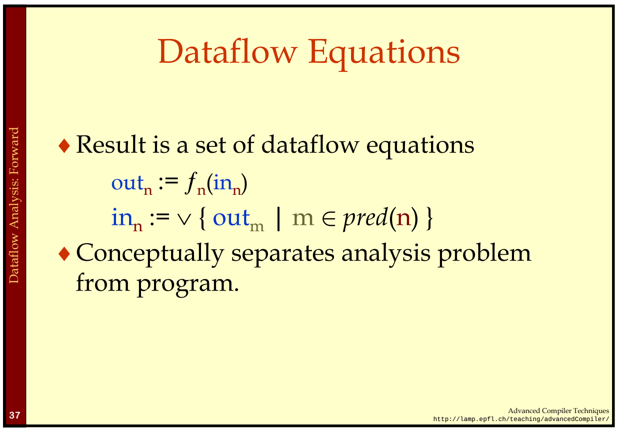
f_n – transfer function for n (given in_n, computes out_n)

- Require that solutions satisfy

i. $\forall n,\ out_n = f_n(in_n)$

ii. $\forall n \neq n_0,\ in_n = \vee \{ out_m \mid m \in pred(n) \}$

iii. $in_{n0} = \perp$



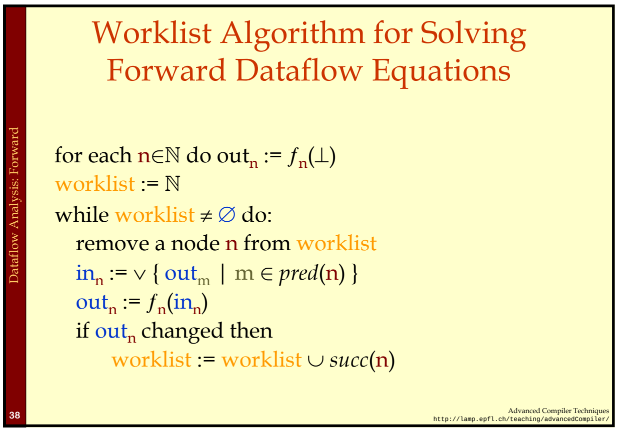
Dataflow Analysis: Forward

Dataflow Equations

- ♦ Result is a set of dataflow equations

  $\text{out}_n := f_n(\text{in}_n)$

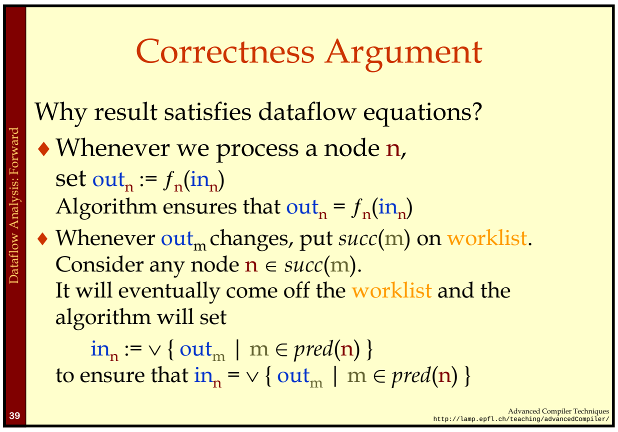
  $\text{in}_n := \vee \{ \text{out}_m \mid m \in \mathit{pred}(n) \}$
- ♦ Conceptually separates analysis problem from program.



Dataflow Analysis: Forward

Worklist Algorithm for Solving Forward Dataflow Equations

```
for each $n \in \mathbb{N}$ do $\text{out}_n := f_n(\bot)$
worklist := $\mathbb{N}$
while worklist $\neq \varnothing$ do:
   remove a node n from worklist
   $\text{in}_n := \vee \{ \text{out}_m \mid m \in \mathit{pred}(n) \}$
   $\text{out}_n := f_n(\text{in}_n)$
   if $\text{out}_n$ changed then
        worklist := worklist $\cup$ $\mathit{succ}(n)$
```



Dataflow Analysis: Forward

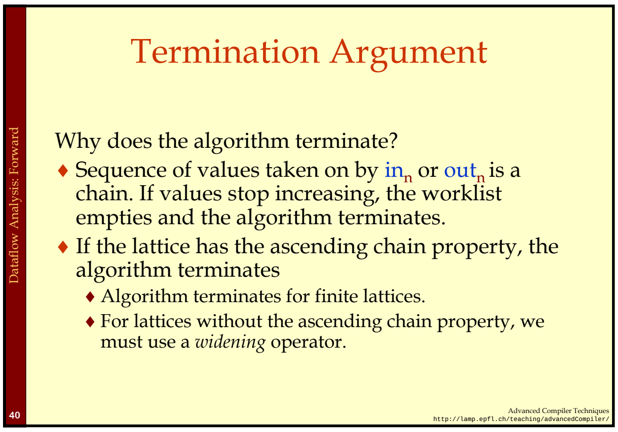
Correctness Argument

Why result satisfies dataflow equations?

- ♦ Whenever we process a node n,
  set $\text{out}_n := f_n(\text{in}_n)$
  Algorithm ensures that $\text{out}_n = f_n(\text{in}_n)$
- ♦ Whenever out_m changes, put $\mathit{succ}(m)$ on worklist.
  Consider any node $n \in \mathit{succ}(m)$.
  It will eventually come off the worklist and the algorithm will set

  $\text{in}_n := \vee \{ \text{out}_m \mid m \in \mathit{pred}(n) \}$

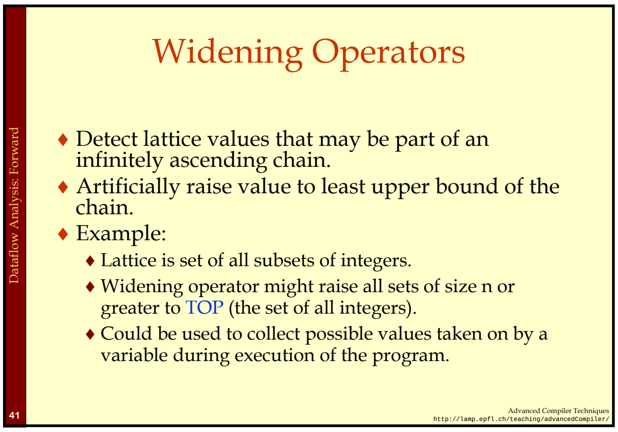
  to ensure that $\text{in}_n = \vee \{ \text{out}_m \mid m \in \mathit{pred}(n) \}$



Termination Argument

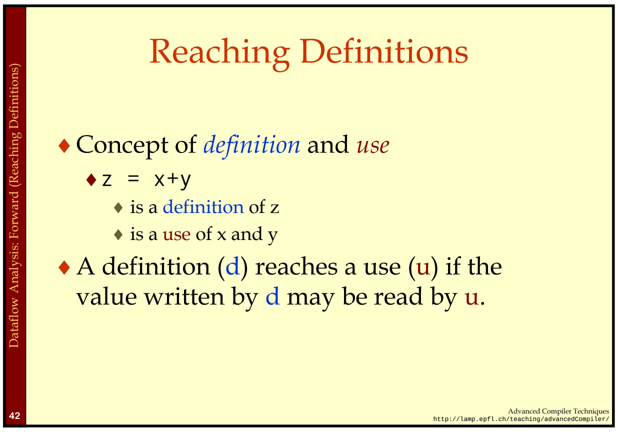
Why does the algorithm terminate?

- Sequence of values taken on by in_n or out_n is a chain. If values stop increasing, the worklist empties and the algorithm terminates.
- If the lattice has the ascending chain property, the algorithm terminates
  - Algorithm terminates for finite lattices.
  - For lattices without the ascending chain property, we must use a *widening* operator.



Widening Operators

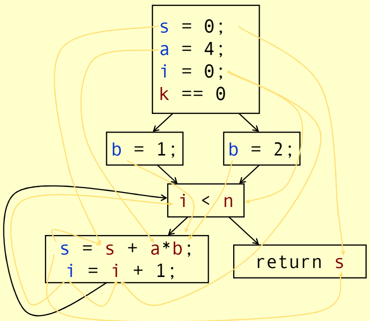
- Detect lattice values that may be part of an infinitely ascending chain.
- Artificially raise value to least upper bound of the chain.
- Example:
  - Lattice is set of all subsets of integers.
  - Widening operator might raise all sets of size n or greater to TOP (the set of all integers).
  - Could be used to collect possible values taken on by a variable during execution of the program.



Reaching Definitions

- Concept of *definition* and *use*
  - `z = x+y`
    - is a definition of z
    - is a use of x and y
- A definition (d) reaches a use (u) if the value written by d may be read by u.

Reaching Definitions



```
s = 0;
a = 4;
i = 0;
k == 0

b = 1;

b = 2;

i < n

s = s + a*b;
i = i + 1;

return s
```

Reaching Definitions Framework

- ℙ = ℘ (the powerset) of the set of definitions in the program (all subsets of the set of definitions).
- ∨ = ∪ (order is ⊆)
- ⊥ = ∅
- 𝔽 = all functions *f* of the form *f*(x) = a ∪ (x-b)
  - b is the set of definitions that the node kills.
  - a is the set of definitions that the node generates.

General pattern for many transfer functions
  - *f*(x) = GEN ∪ (x-KILL)

Does Reaching Definitions Framework Satisfy Properties?

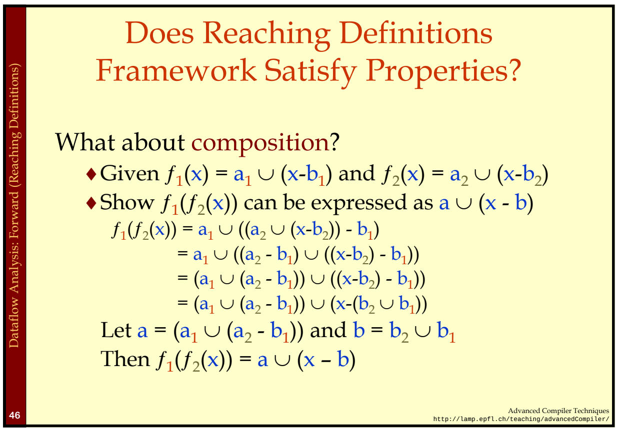
- ⊆ satisfies conditions for ≤

$x \subseteq y$ and $y \subseteq z \Rightarrow x \subseteq z$	(transitivity)
$x \subseteq y$ and $y \subseteq x \Rightarrow y = x$	(asymmetry)
$x \subseteq x$	(reflexivity)

- 𝔽 satisfies transfer function conditions

$\lambda x.\emptyset \cup (x- \emptyset) = \lambda x.x \in \mathbb{F}$	(identity)
Will show $f(x \cup y) = f(x) \cup f(y)$	(distributivity)

$$\begin{aligned} f(x) \cup f(y) &= (a \cup (x - b)) \cup (a \cup (y - b)) \\ &= a \cup (x - b) \cup (y - b) \\ &= a \cup ((x \cup y) - b) \\ &= f(x \cup y) \end{aligned}$$



Does Reaching Definitions Framework Satisfy Properties?

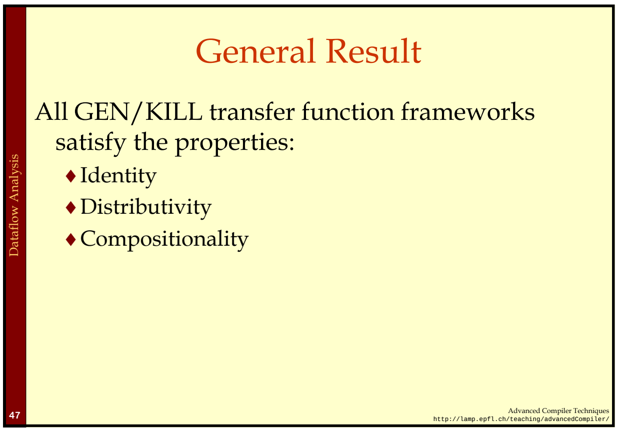
What about composition?

- Given $f_1(x) = a_1 \cup (x-b_1)$ and $f_2(x) = a_2 \cup (x-b_2)$
- Show $f_1(f_2(x))$ can be expressed as $a \cup (x - b)$

$$\begin{aligned} f_1(f_2(x)) &= a_1 \cup ((a_2 \cup (x-b_2)) - b_1) \\ &= a_1 \cup ((a_2 - b_1) \cup ((x-b_2) - b_1)) \\ &= (a_1 \cup (a_2 - b_1)) \cup ((x-b_2) - b_1)) \\ &= (a_1 \cup (a_2 - b_1)) \cup (x-(b_2 \cup b_1)) \end{aligned}$$

Let $a = (a_1 \cup (a_2 - b_1))$ and $b = b_2 \cup b_1$

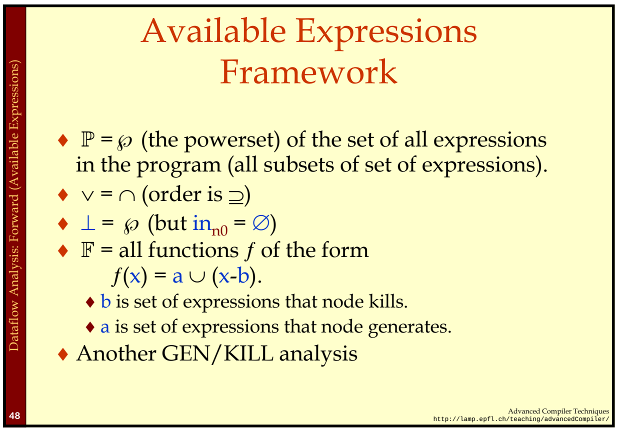
Then $f_1(f_2(x)) = a \cup (x - b)$



General Result

All GEN/KILL transfer function frameworks satisfy the properties:

- Identity
- Distributivity
- Compositionality



Available Expressions Framework

- $\mathbb{P} = \wp$ (the powerset) of the set of all expressions in the program (all subsets of set of expressions).
- $\vee = \cap$ (order is $\supseteq$)
- $\perp = \wp$ (but $in_{n0} = \emptyset$)
- $\mathbb{F}$ = all functions f of the form $f(x) = a \cup (x-b)$.
  - b is set of expressions that node kills.
  - a is set of expressions that node generates.
- Another GEN/KILL analysis

Concept of Conservatism

- Reaching definitions use ∪ as join
  - Optimizations must take into account all definitions that reach along ANY path
- Available expressions use ∩ as join
  - Optimization requires expression to reach along ALL paths
- Optimizations must conservatively take all possible executions into account.
- Structure of analysis varies according to the way the results of the analysis are to be used.

Backward Dataflow Analysis

- Simulates execution of program backward against the flow of control.
- For each node n, we have

  in_n – value at program point before n.

  out_n – value at program point after n.

  f_n – transfer function for n (given out_n, computes in_n).
- Require that solutions satisfy:

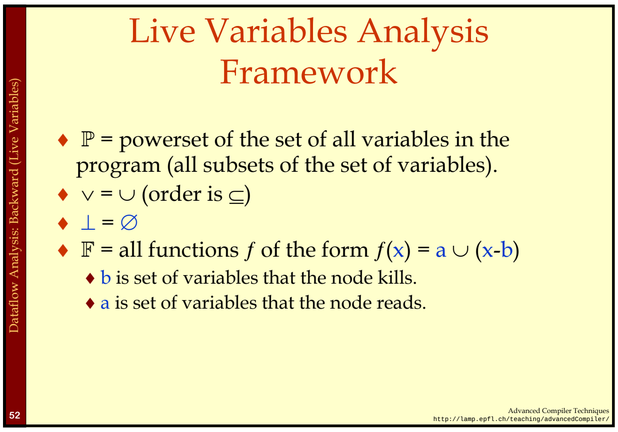
  i. $\forall n.\ in_n = f_n(out_n)$

  ii. $\forall n \notin \mathbb{N}_{final}.\ out_n = \vee \{ in_m \mid m \in succ(n) \}$

  iii. $\forall n \in \mathbb{N}_{final}\ .\ out_n = \bot$

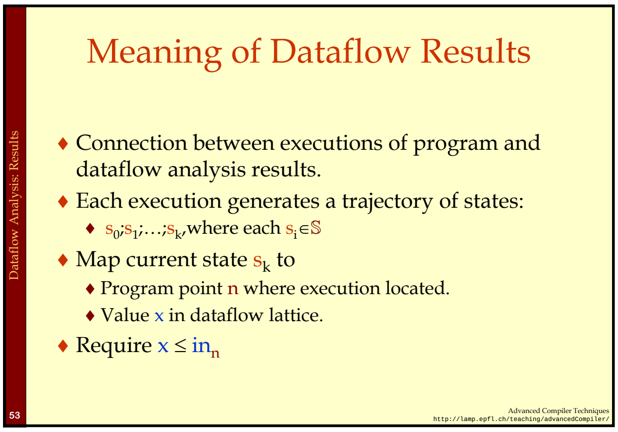
Worklist Algorithm for Solving Backward Dataflow Equations

```
for each $n \in \mathbb{N}$ do $in_n := f_n(\bot)$
worklist := $\mathbb{N}$
while worklist $\neq \emptyset$ do
  remove a node n from worklist
  $out_n := \vee \{ in_m \mid m \in succ(n) \}$
  $in_n := f_n(out_n)$
  if $in_n$ changed then
      worklist := worklist $\cup$ $pred(n)$
```



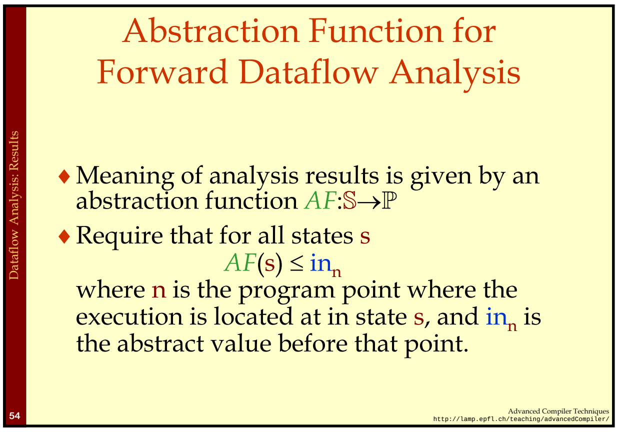
Live Variables Analysis Framework

- $\mathbb{P}$ = powerset of the set of all variables in the program (all subsets of the set of variables).
- $\vee = \cup$ (order is $\subseteq$)
- $\bot = \emptyset$
- $\mathbb{F}$ = all functions f of the form $f(x) = a \cup (x-b)$
  - b is set of variables that the node kills.
  - a is set of variables that the node reads.



Meaning of Dataflow Results

- Connection between executions of program and dataflow analysis results.
- Each execution generates a trajectory of states:
  - $s_0;s_1;\ldots;s_k$, where each $s_i \in \mathbb{S}$
- Map current state s_k to
  - Program point n where execution located.
  - Value x in dataflow lattice.
- Require $x \leq in_n$



Abstraction Function for Forward Dataflow Analysis

- Meaning of analysis results is given by an abstraction function $AF{:}\mathbb{S} \rightarrow \mathbb{P}$
- Require that for all states s

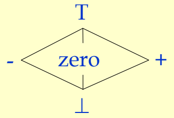
$$AF(s) \leq in_n$$

  where n is the program point where the execution is located at in state s, and in_n is the abstract value before that point.

Sign Analysis Example

Sign analysis - compute sign of each variable v

- Base Lattice: flat lattice on {-,zero,+}



- Actual lattice records a value for each variable
  - Example element: [a→+, b→zero, c→-]

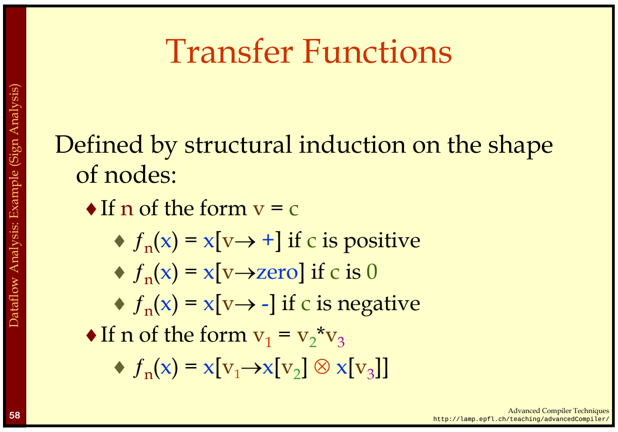
Interpretation of Lattice Values

If value of v in lattice is:

- ⊥: no information about the sign of v.
- -: variable v is negative.
- zero: variable v is 0 .
- +: variable v is positive.
- T: v may be positive or negative or 0.

Operation ⊗ on Lattice

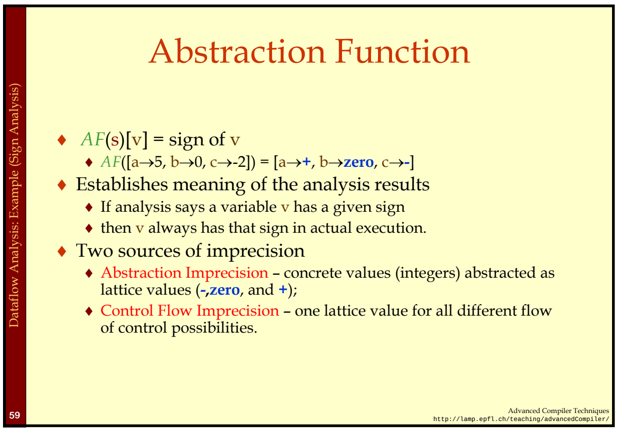
⊗	⊥	-	zero	+	T
⊥	⊥	-	zero	+	T
-	-	+	zero	-	T
zero	zero	zero	zero	zero	zero
+	+	-	zero	+	T
T	T	T	zero	T	T



Transfer Functions

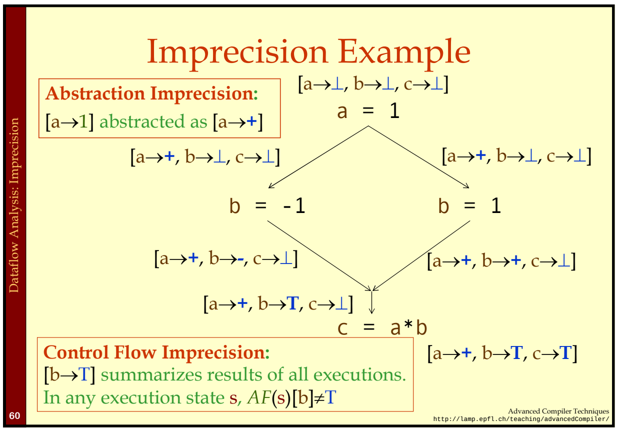
Defined by structural induction on the shape of nodes:

- If n of the form $v = c$
  - $f_n(x) = x[v \rightarrow +]$ if c is positive
  - $f_n(x) = x[v \rightarrow \text{zero}]$ if c is 0
  - $f_n(x) = x[v \rightarrow -]$ if c is negative
- If n of the form $v_1 = v_2 * v_3$
  - $f_n(x) = x[v_1 \rightarrow x[v_2] \otimes x[v_3]]$



Abstraction Function

- $AF(s)[v]$ = sign of v
  - $AF([a \rightarrow 5, b \rightarrow 0, c \rightarrow -2]) = [a \rightarrow +, b \rightarrow \text{zero}, c \rightarrow -]$
- Establishes meaning of the analysis results
  - If analysis says a variable v has a given sign
  - then v always has that sign in actual execution.
- Two sources of imprecision
  - Abstraction Imprecision – concrete values (integers) abstracted as lattice values (-, **zero**, and **+**);
  - Control Flow Imprecision – one lattice value for all different flow of control possibilities.



Imprecision Example

Abstraction Imprecision:
$[a \rightarrow 1]$ abstracted as $[a \rightarrow +]$

$[a \rightarrow \bot, b \rightarrow \bot, c \rightarrow \bot]$

`a = 1`

$[a \rightarrow +, b \rightarrow \bot, c \rightarrow \bot]$ $[a \rightarrow +, b \rightarrow \bot, c \rightarrow \bot]$

`b = -1` `b = 1`

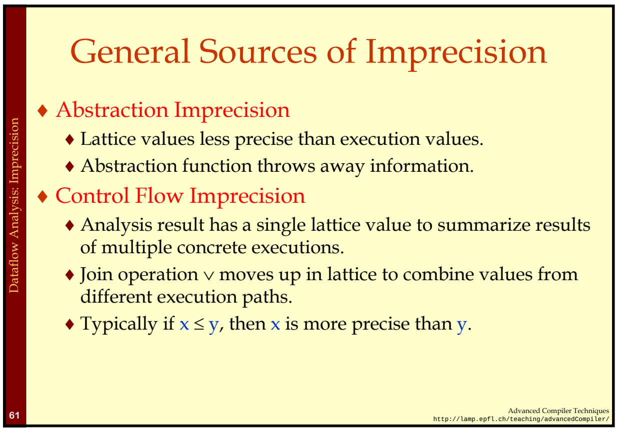
$[a \rightarrow +, b \rightarrow -, c \rightarrow \bot]$ $[a \rightarrow +, b \rightarrow +, c \rightarrow \bot]$

$[a \rightarrow +, b \rightarrow T, c \rightarrow \bot]$

`c = a*b`

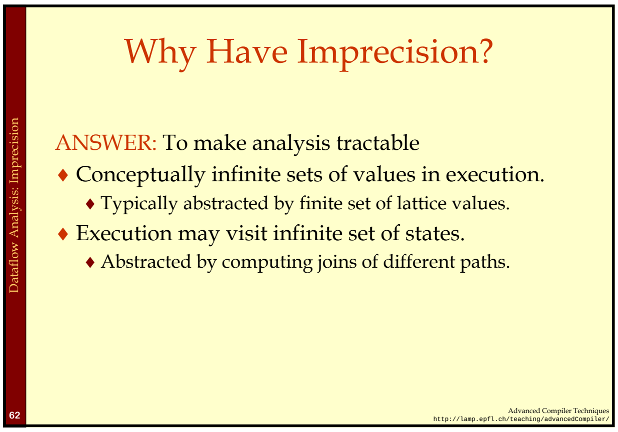
$[a \rightarrow +, b \rightarrow T, c \rightarrow T]$

Control Flow Imprecision:
$[b \rightarrow T]$ summarizes results of all executions.
In any execution state s, $AF(s)[b] \neq T$



General Sources of Imprecision

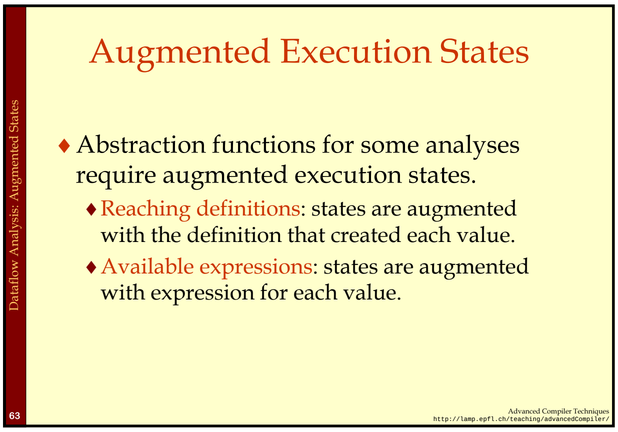
- Abstraction Imprecision
  - Lattice values less precise than execution values.
  - Abstraction function throws away information.
- Control Flow Imprecision
  - Analysis result has a single lattice value to summarize results of multiple concrete executions.
  - Join operation $\vee$ moves up in lattice to combine values from different execution paths.
  - Typically if $x \leq y$, then x is more precise than y.



Why Have Imprecision?

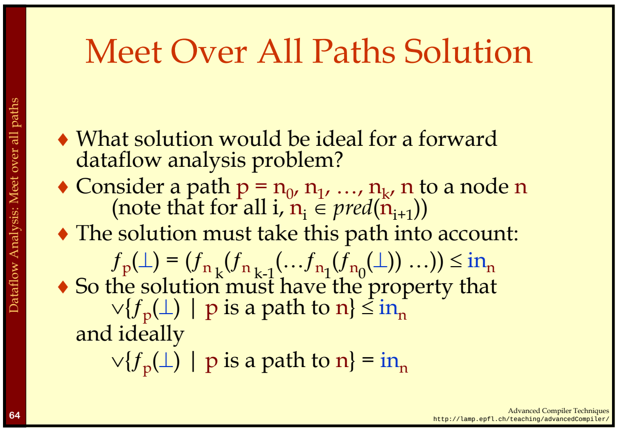
ANSWER: To make analysis tractable

- Conceptually infinite sets of values in execution.
  - Typically abstracted by finite set of lattice values.
- Execution may visit infinite set of states.
  - Abstracted by computing joins of different paths.



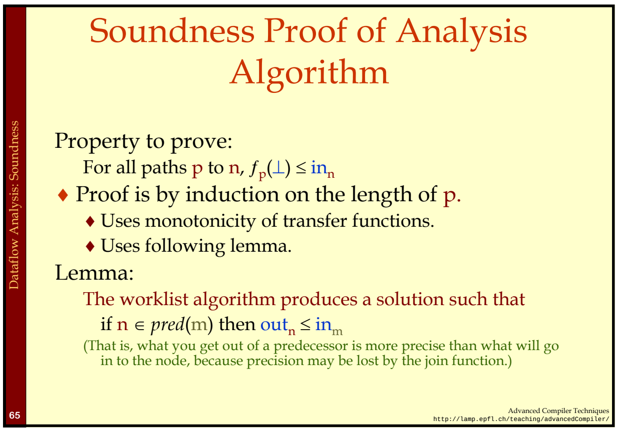
Augmented Execution States

- Abstraction functions for some analyses require augmented execution states.
  - Reaching definitions: states are augmented with the definition that created each value.
  - Available expressions: states are augmented with expression for each value.



Meet Over All Paths Solution

- What solution would be ideal for a forward dataflow analysis problem?
- Consider a path $p = n_0, n_1, \dots, n_k, n$ to a node n (note that for all i, $n_i \in pred(n_{i+1})$)
- The solution must take this path into account:
  $f_p(\bot) = (f_{n_k}(f_{n_{k-1}}(\dots f_{n_1}(f_{n_0}(\bot)) \dots)) \le in_n$
- So the solution must have the property that
  $\vee\{f_p(\bot) \mid p \text{ is a path to } n\} \le in_n$
  and ideally
  $\vee\{f_p(\bot) \mid p \text{ is a path to } n\} = in_n$



Soundness Proof of Analysis Algorithm

Property to prove:

For all paths p to n, $f_p(\bot) \le in_n$

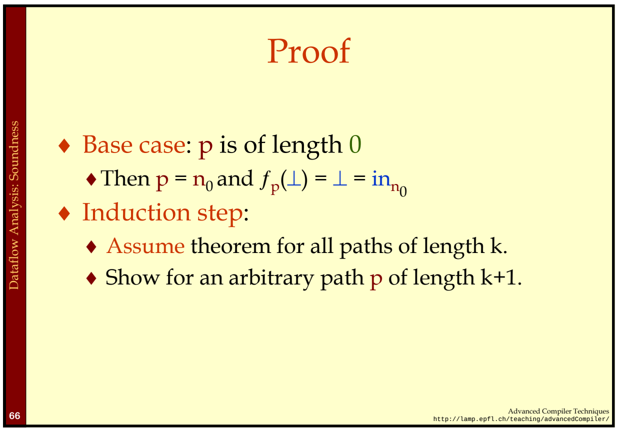
- Proof is by induction on the length of p.
  - Uses monotonicity of transfer functions.
  - Uses following lemma.

Lemma:

The worklist algorithm produces a solution such that

if $n \in pred(m)$ then $out_n \le in_m$

(That is, what you get out of a predecessor is more precise than what will go in to the node, because precision may be lost by the join function.)



Proof

- Base case: p is of length 0
  - Then $p = n_0$ and $f_p(\bot) = \bot = in_{n_0}$
- Induction step:
  - Assume theorem for all paths of length k.
  - Show for an arbitrary path p of length k+1.

Induction Step Proof

- Given a path $p = n_0, \ldots, n_k, n$ show $(f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(f_{n_0}(\perp)) \ldots)) \leq in_n$

  By induction assumption:

  $(f_{n_{k-1}}(\ldots f_{n1}(f_{n0}(\perp)) \ldots)) \leq in_{n_k}$

  Apply f_{n_k} to both sides:

  $f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(f_{n_0}(\perp)) \ldots)$? $f_{n_k}(in_{n_k})$

  By monotonicity:

  $(f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(f_{n_0}(\perp)) \ldots)) \leq f_{n_k}(in_{n_k})$

  By definition of f_{n_k}: $f_{n_k}(in_{n_k}) = out_{n_k}$

  $(f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(f_{n_0}(\perp)) \ldots)) \leq out_{n_k}$

  By lemma: $out_{n_k} \leq in_n$

  By transitivity:

  $(f_{n_k}(f_{n_{k-1}}(\ldots f_{n_1}(f_{n_0}(\perp)) \ldots)) \leq in_n$

Distributivity

- Distributivity preserves precision.
- If framework is distributive, then the worklist algorithm produces the meet over paths solution:

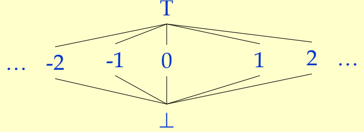
  For all n:

  $\vee\{f_p(\perp) \mid p \text{ is a path to } n\} = in_n$

Lack of Distributivity Example

Integer Constant Propagation (ICP)

- Flat lattice on integers

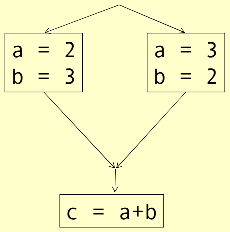


- Actual lattice records a value for each variable
  - Example element: [a→3, b→2, c→5]

Transfer Functions

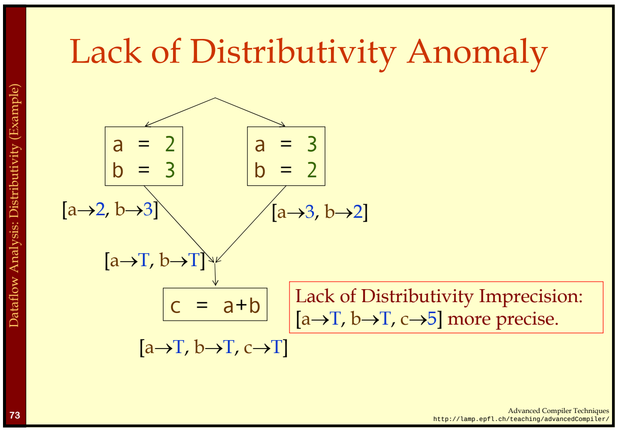
- If n of the form v = c
  - $f_n(x) = x[v \to c]$
- If n of the form $v_1 = v_2 + v_3$
  - $f_n(x) = x[v_1 \to x[v_2] + x[v_3]]$

Lack of Distributivity Anomaly



Lack of distributivity of ICP

- Consider transfer function *f* for c = a + b (*f*(x) = x[c→x[a] + x[b]])
- *f*([a→3, b→2]) ∨ *f*([a→2, b→3]) =
  [a→3, b→2] [c→ [a→3, b→2][a] + [a→3, b→2][b]] ∨
  [a→2, b→3] [c→ [a→2, b→3][a] + [a→2, b→3][b]] =
  [a→3, b→2] [c→ 3 + 2] ∨ [a→2, b→3] [c→ 2 + 3] =
  [a→3, b→2] [c→5] ∨ [a→2, b→3] [c→5] =
  [a→T, b→T, c→5]
- *f*([a→3, b→2]∨[a→2, b→3]) =
  f([a→T, b→T]) =
  [a→T, b→T] [c→ [a→T, b→T][a] + [a→T, b→T][b]] =
  [a→T, b→T, c→T]



Dataflow Analysis: Distributivity (Example)

Lack of Distributivity Anomaly

```
a = 2
b = 3
```

```
a = 3
b = 2
```

[a→2, b→3]

[a→3, b→2]

[a→T, b→T]

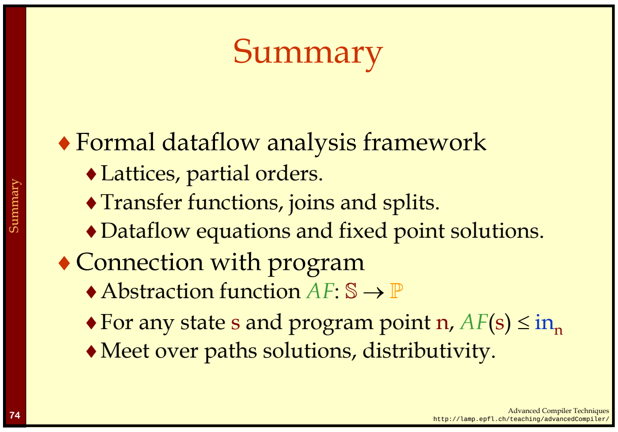
```
c = a+b
```

[a→T, b→T, c→T]

Lack of Distributivity Imprecision:
[a→T, b→T, c→5] more precise.

73

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/



Summary

Summary

- Formal dataflow analysis framework
  - Lattices, partial orders.
  - Transfer functions, joins and splits.
  - Dataflow equations and fixed point solutions.
- Connection with program
  - Abstraction function $AF: \mathbb{S} \rightarrow \mathbb{P}$
  - For any state s and program point n, $AF(s) \le in_n$
  - Meet over paths solutions, distributivity.

74

Advanced Compiler Techniques
http://lamp.epfl.ch/teaching/advancedCompiler/