CSc 553
Principles of Compilation

10. Dataflow Analysis Frameworks

Saumya Debray

The University of Arizona

Tucson, AZ 85721

Dataflow analysis: commonalities

merge operator

$\cup$ $\cap$

	$\exists$	$\forall$
forward	reaching defns.	available exprs.
backward	liveness	?

boundary value

$\emptyset$ **all**

dataflow equations

out[B] = f_B(in[B])

in[B] = f_B(out[B])

Dataflow analysis: commonalities

- The analyses compute sets of "dataflow facts"
  - for each basic block B: in[B], out[B]
  - computed iteratively to convergence ("fixpoint")

- intra-block analysis: uses a "transfer function" f_B that captures the effects of B
  - forward analyses: out[B] = f_B(in[B])
  - backward analyses: in[B] = f_B(out[B])

- inter-block analysis: uses a "merge operator"
  - "for some path" ($\exists$) analyses: $\cup$
  - "for all paths" ($\forall$) analyses: $\cap$

Dataflow analysis: questions

Given some dataflow analysis A:

- Is it sound?
  - do the results account for all possible runtime scenarios?
  - under what assumptions?
- Is it precise?
  - how good are the results?
- Is it efficient?
  - how fast does it run?

Dataflow analysis frameworks

- Provides a unifying mathematical structure underlying these analyses
  - helps explain why the analyses are the way they are
  - helps us understand commonalities between different analyses
- Makes it easier to figure out the details of new analyses
- Helps answer questions about soundness, precision, efficiency

Mathematical preliminaries

Partial order

Definition: A binary relation $\sqsubseteq$ over a set S is a *partial order* if it satisfies:

- $\forall x \in S: x \sqsubseteq x$ (reflexive)
- $\forall x, y \in S : x \sqsubseteq y$ and $y \sqsubseteq x$ implies $x = y$ (anti-symmetric)
- $\forall x, y, z \in S : x \sqsubseteq y$ and $y \sqsubseteq z$ implies $x \sqsubseteq z$ (transitive)

Notation:

- $(S, \sqsubseteq)$ denotes a set S with a relation $\sqsubseteq$
- if $\sqsubseteq$ is a partial order on a set S, then $(S, \sqsubseteq)$ is called a *partially ordered set* (poset)

EXERCISE

Which of these are partial orders? Why or why not?

- $(\mathbb{Z}, \leq)$ where $\mathbb{Z}$ is the set of integers
- $(\mathbb{Z}, <)$
- (set of all finite ASCII strings; lexicographic ordering)
- (S, R) where:
  - S = the set of all UA students, and
  - $\forall x, y \in S$: x R y iff x and y have the same last name
- (S, R) where:
  - S = the set of all UA students, and
  - $\forall x, y \in S$: x R y iff x and y are friends

Monotonicity

Given a poset $(S, \sqsubseteq)$, a function $f : S \rightarrow S$ is said to be *monotone* iff

$$\forall x, y \in S: x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

Intuition: If f is monotone, then a bigger input yields a bigger (or same) output

Meets and joins

Given a poset (S, ⊑) and a, b ∈ S:

- c ∈ S is a *join* of a and b (denoted a ⊔ b) iff:
  - a ⊑ c and b ⊑ c; and
  - there is no other x ∈ S : a ⊑ x ⊑ c and b ⊑ x ⊑ c

  c is also called the *least upper bound* (LUB) of a and b

- d ∈ S is a *meet* of a and b (denoted a ⊓ b) iff:
  - d ⊑ a and d ⊑ b; and
  - there is no other x ∈ S : d ⊑ x ⊑ a and d ⊑ x ⊑ b

  d is also called the *greatest lower bound* (GLB) of a and b

Lattices

Definition [lattice]:

- A poset (S, ⊑) is a *lattice* iff every pair of elements x, y ∈ S has a unique meet and a unique join

Note: this implies that every non-empty finite subset of S has a unique meet and join

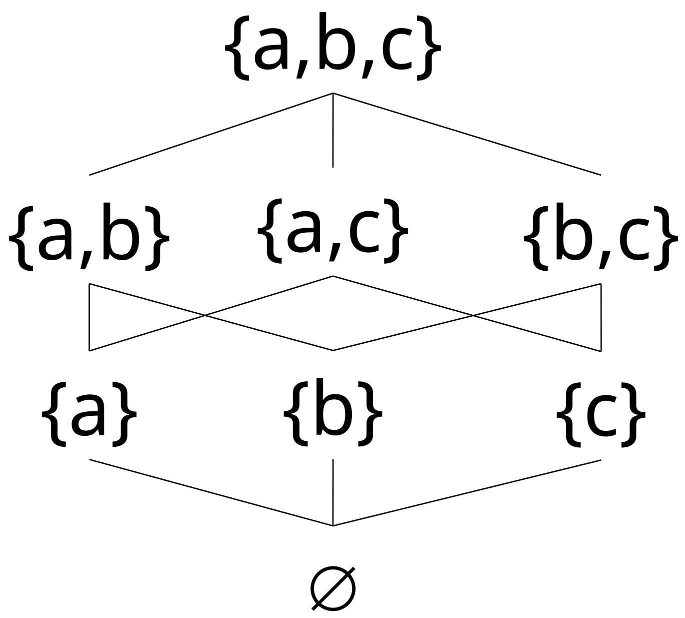
Definition [complete lattice]:

- A *complete lattice* is a lattice (S, ⊑) where every subset X ⊆ S has a unique meet and join

Fact: A complete lattice (S, ⊑) has a least element ⊥ ("bottom") and a greatest element ⊤ ("top")

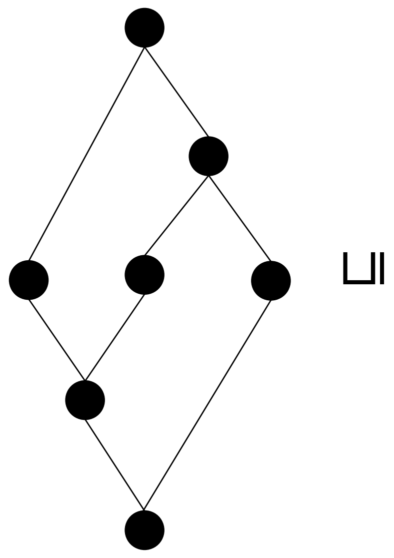
Example

The set of all subsets of {a, b, c} ordered by ⊆:

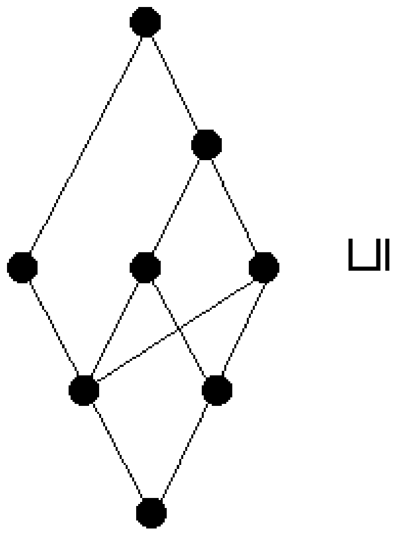


lattice ordering: ⊆
meet: ∩
join: ∪

Example



a poset that is a lattice



a poset that is not a lattice (why?)

Semilattices

Definition [semilattice]:

- A *join-semilattice* is a poset (S, ⊑) where every pair of elements has a unique join
- A *meet-semilattice* is a poset (S, ⊑) where every pair of elements has a unique meet

Fact: If (S, ⊑) is a lattice then (S, ⊑) is a join-semilattice and a meet-semilattice

EXERCISE

Consider $(B^3, \preceq)$ where:

- B^3 is the set of length-3 bit-vectors, i.e.,

  $B^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$
- $\forall\, x, y \in B^3$: $x \preceq y$ iff $\#1s(x) \leq \#1s(y)$

#1s(u) = the number of 1s in u
e.g., #1s(011) = 2

Questions:

- is $\preceq$ a partial order?
- is $(B^3, \preceq)$ a lattice?
  - what is the meet operation?
  - what is the join operation?
  - is it a complete lattice?

EXERCISE

Consider (B^*, $\preceq$) where:

- B^* is the set of all finite-length bit-vectors, i.e.,

  $B^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 010, 011, 100, \ldots\}$
- $\forall\, x, y \in B^*$: $x \preceq y$ iff #1s(x) $\leq$ #1s (y)

#1s(u) = the number of 1s in u
e.g., #1s(011) = 2

Questions:

- is $\preceq$ a partial order?
- is (B^*, $\preceq$) a lattice?

EXERCISE

Consider $(\subseteq(\mathbb{Z}), \square)$:

- is this a poset?
- is it a join-semilattice?
  - what is the join operation?
- is it a meet-semilattice?
  - what is the meet operation?
- is it a lattice?
  - is it a complete lattice?

Dataflow analysis frameworks

Transfer functions for basic blocks

Dataflow equations:

- Reaching definitions:

  $in[B] = \cup \{out[p] \mid p \in preds[B]\}$

  $out[B] = \boxed{gen[B] \cup (in[B] - kill[B])}$

- Available expressions:

  $in[B] = \cap \{out[p] \mid p \in preds[B]\}$

  $out[B] = \boxed{gen[B] \cup (in[B] - kill[B])}$

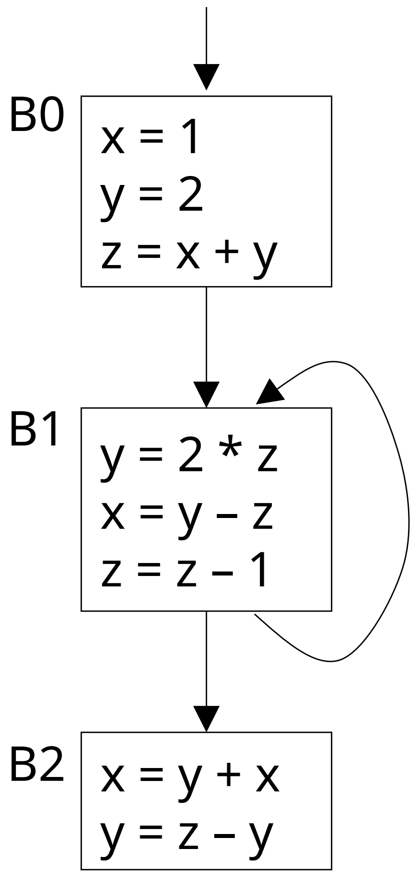
transfer function for B

- captures how the code in B affects the relationship between in[B] and out[B]
- gen[B], kill[B] depend only on B
  - *can be considered to be fixed for any given B*

EXERCISE

Analysis: reaching definitions

- What are the transfer functions for each of the blocks B0, B1, and B2?
- How are these transfer functions affected if we add an edge B0 $\rightarrow$ B2 ?



```
x = 1
y = 2
z = x + y
```

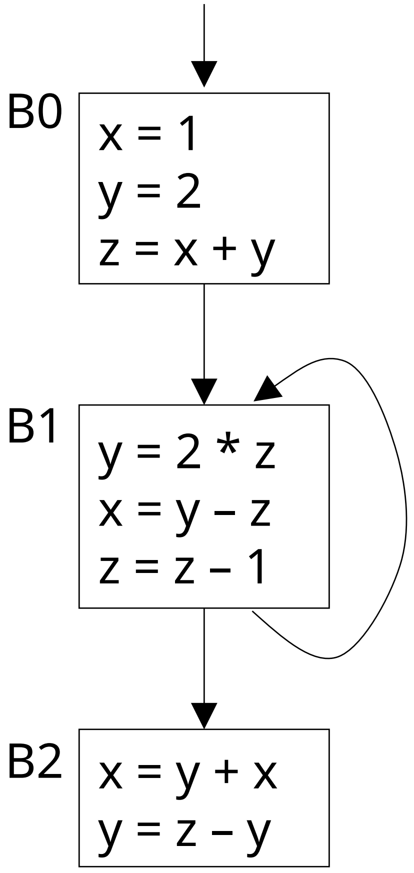
```
y = 2 * z
x = y – z
z = z – 1
```

```
x = y + x
y = z – y
```

EXERCISE

Analysis: reaching definitions

- What are the transfer functions for each of the blocks B0, B1, and B2?
- How are these transfer functions affected if we add an edge B0 → B2 ?

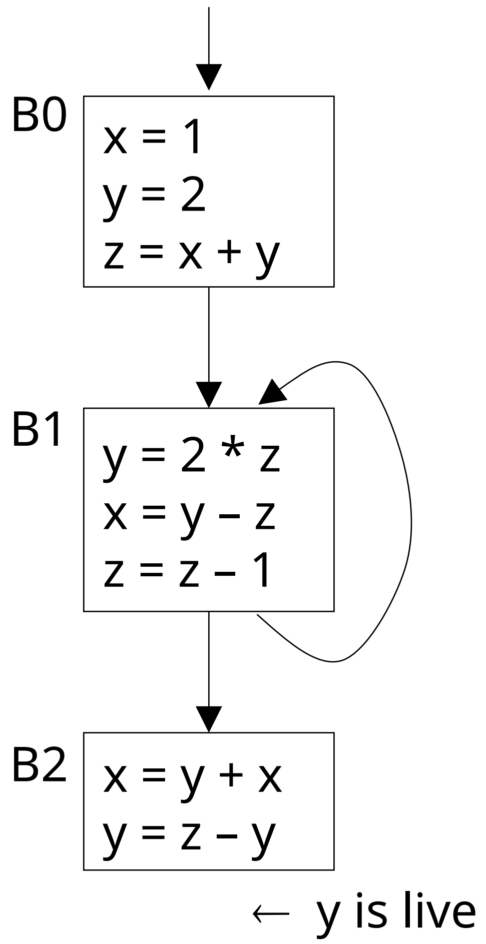


B0: out[B0] = (in[B] - #4, #5, #6, #7, #8}) U {#1, #2, #3}

EXERCISE

Analysis: variable liveness

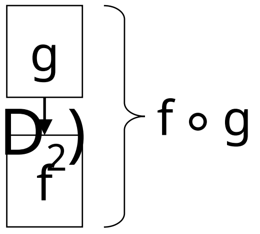
- What are the transfer functions for each of the blocks B0, B1, and B2?
- How are these transfer functions affected if x is also live at the end of B2?



Transfer functions: properties

$$f(x) = C_1 \cup (x - C_2)$$

- Monotone:
  - $x_1 \subseteq x_2 \Rightarrow f(x_1) \subseteq f(x_2)$
- Closed under composition:
  - suppose $f(x) = C_1 \cup (x - C_2)$, $g(x) = D_1 \cup (x - D_2)$
  - $(f \circ g)(x) = f(g(x)) = (C_1 \cup D_1) \cup (x - (C_2 \cup D_2))$



- Can be identity:
  - $C_1 = \emptyset, C_2 = \emptyset \Rightarrow f(x) = x$

Axioms for transfer functions

The set of transfer functions **F** satisfies the axioms:

- $\forall f \in$ **F** : f is monotone
- id $\in$ **F**
- **F** is closed under composition

Dataflow analysis frameworks

A dataflow analysis framework consists of:

- a control flow graph G = (V, E)
- a complete lattice **L** with meet operation $\sqcap$
  - the *domain* of dataflow facts
- a transfer function F that associates each node b $\in$ V with a monotone function

$$f_b : \mathbf{L} \to \mathbf{L}$$

- an initial value v_{ENTRY} (or v_{EXIT}) that gives the lattice value for the entry (or exit) blocks

EXERCISE

- Suppose that:
  - **L** is a complete lattice with ordering $\sqsubseteq$ and meet $\sqcap$
  - f : **L** $\rightarrow$ **L** is monotone w.r.t $\sqsubseteq$
  - $x, y \in$ **L**
- What is the relationship (in terms of $\sqsubseteq$) between:

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

Example 1: “gen-kill analyses”

	Available expressions	**Live variables**
Domain	sets of expressions	sets of variables
Direction	forward: $OUT[b] = f_b(IN[b])$ $IN[b] = \sqcap \{OUT[x] \mid x \in pred(b)\}$	backward: $IN[b] = f_b(OUT[b])$ $OUT[b] = \sqcap \{IN[x] \mid x \in succ(b)\}$
Transfer function	$f_b(x) = (x - kill_b) \cup gen_b$	$f_b(x) = (x - def_b) \cup use_b$
Meet operation	$\cap$	$\cup$
Boundary condition	$IN[entry] = \varnothing$	$IN[exit] = \varnothing$
Initialization values (interior	$IN[b]$ = set of all variables	$IN[b] = \varnothing$

Example 2: Constant propagation

1. **Domain of analysis**

- The analysis propagates sets of mappings from variables in the CFG to their values. E.g.:

  $[\, x \mapsto 2,\ y \mapsto \text{undef},\ z \mapsto \text{NAC} \,]$

  - undef : “we don’t yet know anything about its value”
  - NAC : “(maybe) not a constant”

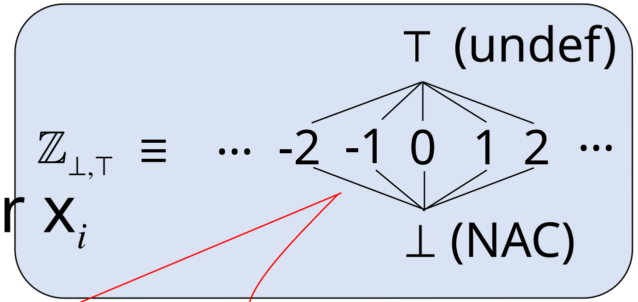
Example 2: Constant propagation

2. The lattice ordering

- We use a *product lattice* with one component for each variable in the program
  - for a program with variables $x_1, x_2, \ldots, x_n$ the analysis lattice L is:

    $$L \equiv L_1 \times L_2 \times \ldots \times L_n$$

    where $L_i \in \mathbb{Z}_{\perp,\top}$ is the mapping for x_i



  - the lattice ordering ⊑ on L is:

    $[u_1, \ldots, u_n] \sqsubseteq [v_1, \ldots, v_n]$ iff $u_1 \sqsubseteq v_1 \wedge \ldots \wedge u_n \sqsubseteq v_n$

    (aka “pointwise ordering”)
  - meet operation is similarly computed pointwise

Example 2: Constant propagation

3. Transfer functions

- Transfer functions map lattice elements (i.e., tuples) to lattice elements
- For s : x = y+z: the transfer function $f_s(p) = q$, where:
  - q(x) is defined as:

    if $p(y) = \top$ **or** $p(z) = \top$ **then** $q(x) = \top$

    else if $p(y) = \perp$ **or** $p(z) = \perp$ **then** $q(x) = \perp$

    else if $p(y) = c_y$ **and** $p(z) = c_z$ **then** $q(x) = c_y + c_z$
  - $q(w) = p(w)$ for $w \neq x$
- For a basic block: compose transfer functions for the individual statements

Iterative dataflow analysis

Iterative algorithm (forward)

- Initialization:

  OUT[ENTRY] = v_{ENTRY}

  for all other blocks B: OUT[B] = $\top$ (top element of lattice)

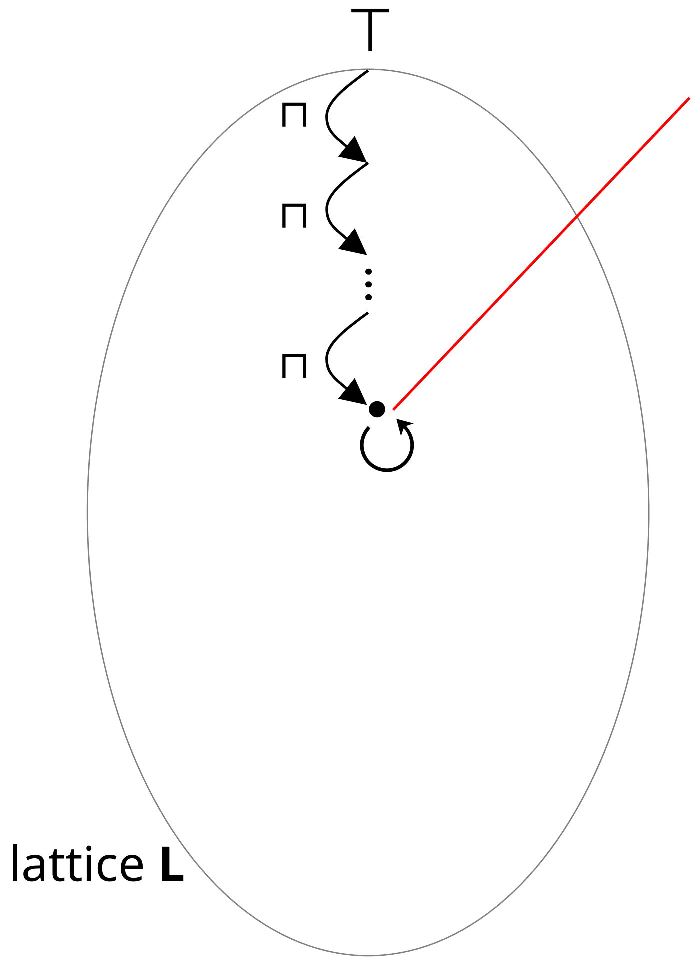
- Iteration:

```
while there is a change to any OUT set:
    for each block B:
        IN[B] = $\sqcap$ {OUT[p] | p $\in$ predecessors(B)}
        OUT[B] = $f_B$(IN[B])
```

Iterative algorithm (backward)

- Similar to iterative algorithm for forward analyses:
  - swap IN and OUT everywhere
  - replace ENTRY by EXIT

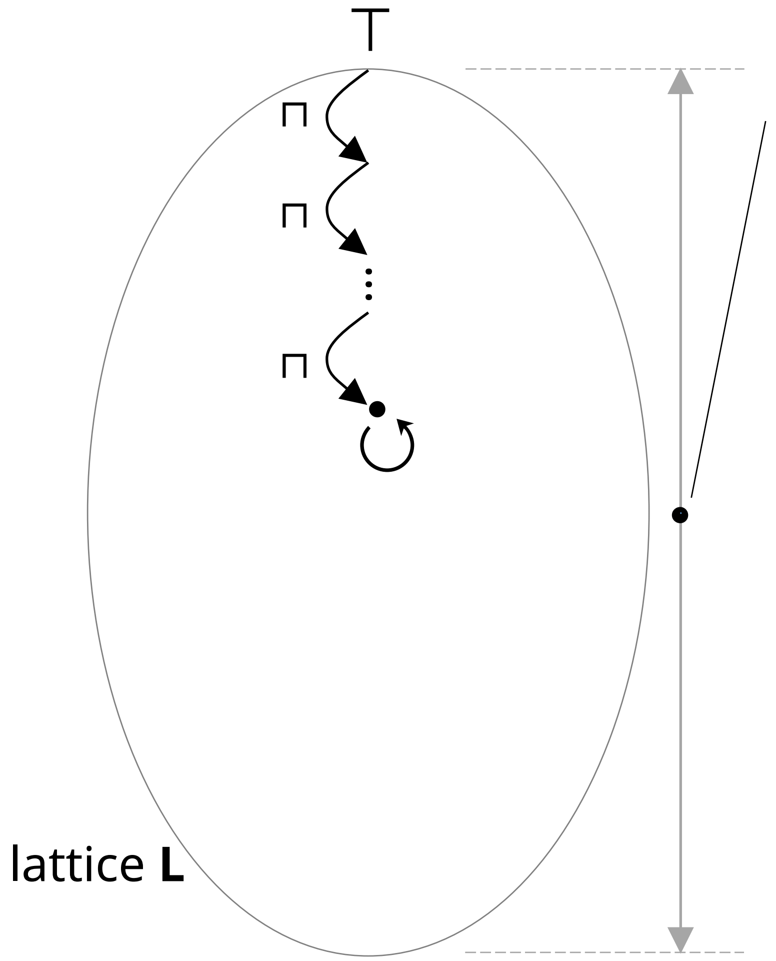
What does the iteration compute?



the value does not change on further iteration
⇒ a "fixpoint" of the transfer function F

computed starting from T by repeated applications of ⊓:
⇒ "greatest (or maximal) fixpoint"

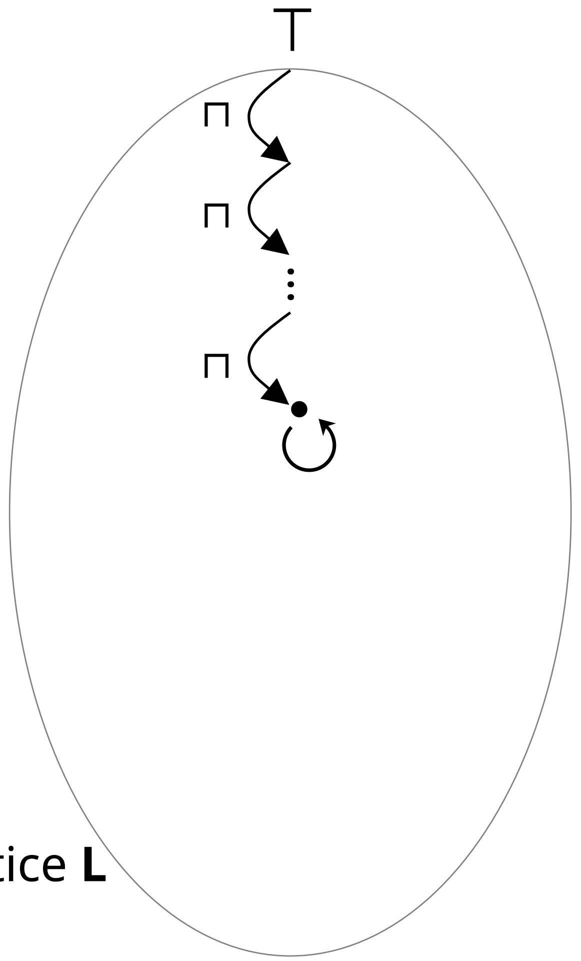
What does the iteration compute?



height of **L**
= length of longest ⊑-chain in **L**

Fact: If **L** has finite height, then:
- the iterative algorithm terminates
- and computes the maximal fixpoint (MFP) of the transfer function F

What does the iteration compute?



lattice **L**

Intuition: Each iteration of the algorithm accounts for more and more of the program's runtime behavior

- ⊑ measures "ignorance"
  - x ⊑ y : "x accounts for more runtime behaviors than y"
  - T : does not account for any runtime behaviors

Soundness

Soundness

- **Required**: The result computed by *any* analysis must be *safe*
  - i.e., must capture all possible executions of the program
- **Fact**: There is no algorithm that *always* captures *exactly* the effects of all possible executions of the program (Rice's Theorem)

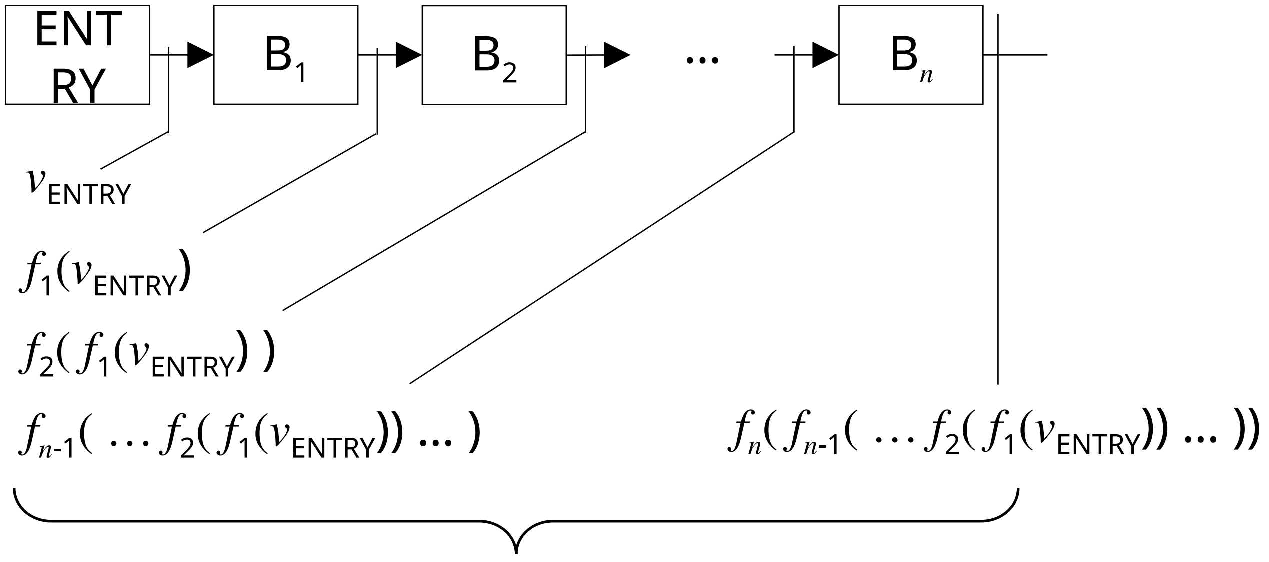
⇒ An analysis can only compute an approximation to the real behavior of the program

  - the safety requirement implies that this has to be a *conservative approximation*

Transfer function of a path

- An execution path in a program is a path in its control flow graph

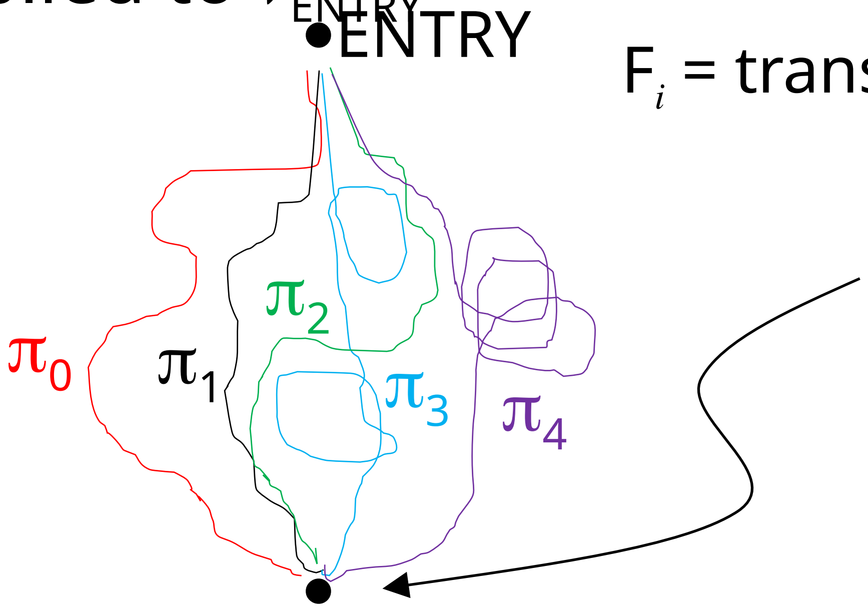
≡ a sequence of blocks: ENTRY → B_1 → B_2 → … → B_n


ENTRY
B_1
B_2
…
B_n
v_{ENTRY}
$f_1(v_{\text{ENTRY}})$
$f_2(f_1(v_{\text{ENTRY}}))$
$f_{n\text{-}1}(\ldots f_2(f_1(v_{\text{ENTRY}}))\ldots)$
$f_n(f_{n\text{-}1}(\ldots f_2(f_1(v_{\text{ENTRY}}))\ldots))$

Transfer function = $f_n \circ f_{n\text{-}1} \circ \ldots \circ f_2 \circ f_1$

Meet over all paths: MOP

MOP: meet, over all paths π_i from ENTRY to a given point, of the transfer function along π_i applied to v_{ENTRY}



F_i = transfer function for path π_i

MOP = $\bigwedge F_i(v_{\text{ENTRY}})$

The ideal solution

- MOP assumes that all paths from ENTRY in the control flow graph can be executed
  - this is safe
  - but may not always be true
- IDEAL: meet over all *executable* paths from ENTRY to a point
  - this is the ideal solution
  - not computable in general

Soundness

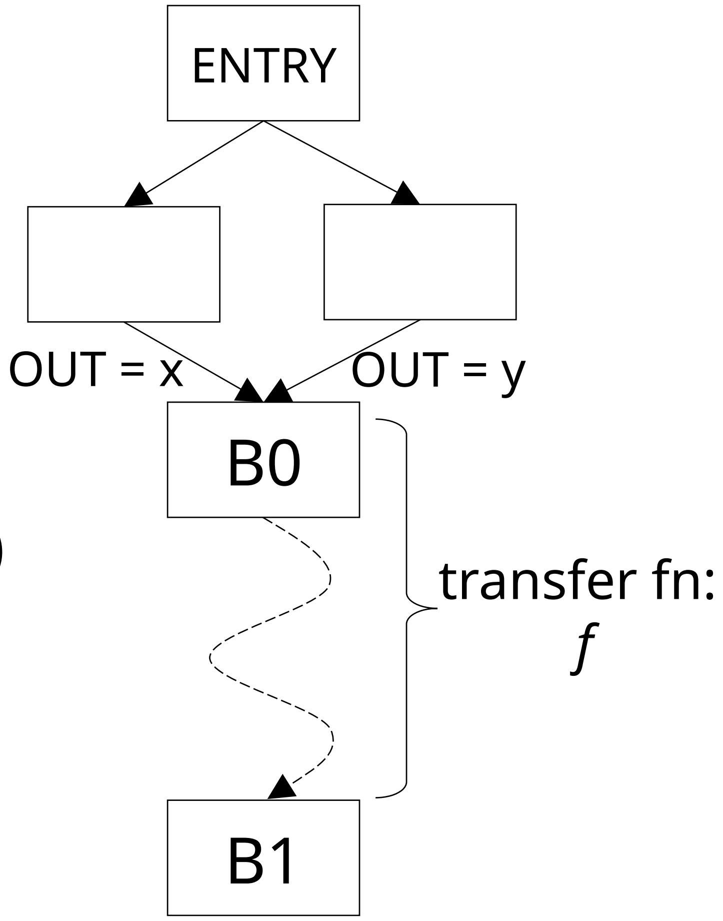
- Need to show:
  - the result computed by the iterative algorithm is a conservative approximation to the program's runtime behavior
  - equivalently: MFP $\sqsubseteq$ IDEAL
- Note: any solution x such that $x \sqsupseteq$ IDEAL is unsafe
  - x does not account for some execution paths

Relationship between MFP and MOP

- MFP vs. MOP:
  - MOP: composes transfer fns along all paths, then takes $\sqcap$ over all of them
  - MFP: alternates transfer fn composition and $\sqcap$
- **Fact:** $f(x \sqcap y) \sqsubseteq f(x) \sqcap f(y)$
  - f is monotone
  - $(x \sqcap y) \sqsubseteq x$ and $(x \sqcap y) \sqsubseteq y$
- This implies: MFP $\sqsubseteq$ MOP

Relationship between MFP and MOP

- MFP: applies ⊓ early
  - IN[B0] = x ⊓ y
  - IN[B1] = *f*(x ⊓ y)
- MOP: applies ⊓ late
  - IN[B1] = *f*(x) ⊓ *f*(y)
- *f* monotone ⇒ *f*(x ⊓ y) ⊑ *f*(x) ⊓ *f*(y)
  - MFP ⊑ MOP
  - i.e., MFP is potentially less precise than MOP, as though it considers additional (nonexistent) paths



Soundness

- MOP ⊑ IDEAL
  - since IDEAL considers fewer execution paths than MOP
- MFP ⊑ MOP
- Since ⊑ is transitive, we have: MFP ⊑ IDEAL

  ⇒ the iterative algorithm is safe
- The soundness argument assumes that:
  - the dataflow lattice **L** has finite height
  - the transfer functions are monotone
    - easy proof for functions in Gen-Kill form

$\left.\vphantom{\begin{matrix}a\\b\\c\end{matrix}}\right\}$ satisfied by all the analyses we've studied

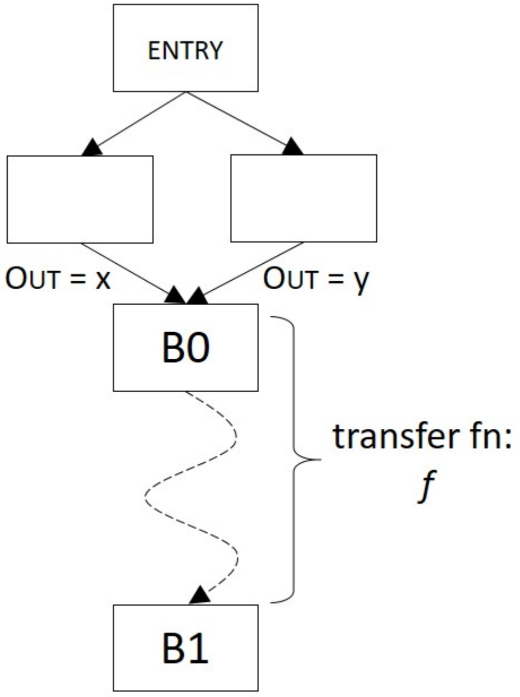
Distributive analyses

MFP vs. MOP revisited

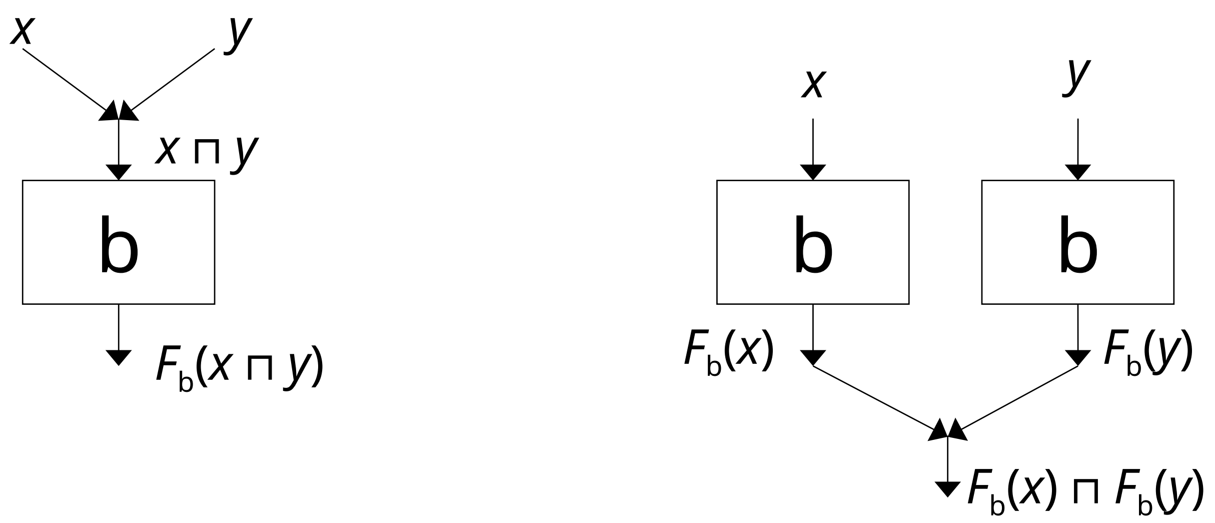
- MFP: applies ⊓ early
  - IN[B0] = x ⊓ y
  - IN[B1] = *f*(x ⊓ y)
- MOP: applies ⊓ late
  - IN[B1] = *f*(x) ⊓ *f*(y)

We already know: MFP ⊑ MOP

QUESTION: When is MFP = MOP?



MFP vs. MOP revisited



The analyses are equivalent if ⊓ does not lose any information, i.e.:

$$F_b(x \sqcap y) \;=\; F_b(x) \sqcap F_b(y)$$

Distributive analyses

- A dataflow analysis over a lattice **L** and transfer function F is said to be *distributive* if

$$\forall\, x, y \in \mathbf{L}: F(x \sqcap y) \;=\; F(x) \sqcap F(y)$$

- This condition is strictly stronger than monotonicity

- Distributivity means combining paths early does not lose precision
  - MFP = MOP

MFP vs. MOP

Liveness analysis:

$$F_b(x \sqcap y) = \text{use}[b] \cup ((x \cup y) - \text{def}[b])$$

$$= \text{use}[b] \cup ((x - \text{def}[b]) \cup (y - \text{def}[b]))$$

$$= (\text{use}[b] \cup ((x - \text{def}[b]))) \cup (\text{use}[b] \cup ((y - \text{def}[b])))$$

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

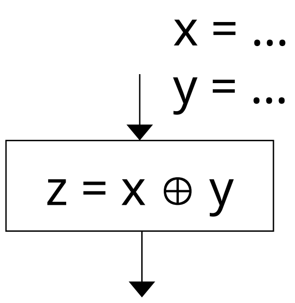
$\Rightarrow$ liveness analysis computes the MOP solution, i.e., it is precise

Fact: All the Gen-Kill analyses are distributive

$\Rightarrow$ they compute precise solutions

Non-distributive analyses

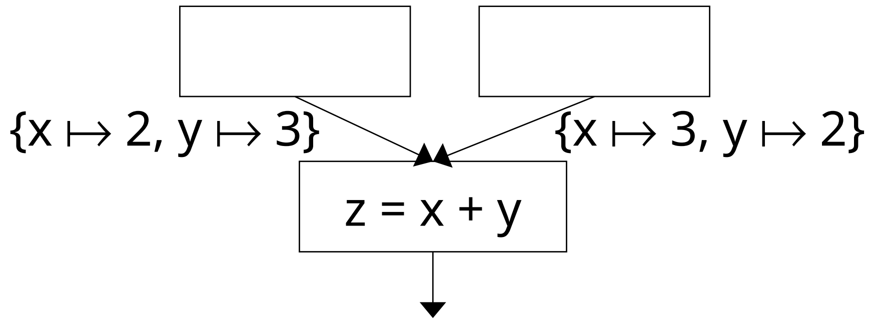
Example: constant propagation



⊤ : no value assigned; ⊥ : don't know

x	y	z = x ⊕ y
c1	c2	c1 ⊕ c2
⊥	any	⊥
any	⊥	⊥
⊤	any	⊤
any	⊤	⊤

Example: constant propagation



⊤ : no value assigned; ⊥ : don't know

x	y	z = x ⊕ y
c1	c2	c1 ⊕ c2
⊥	any	⊥
any	⊥	⊥
⊤	any	⊤
any	⊤	⊤

- Computing meets early:
  - $x \mapsto (\{2\} \sqcap \{3\}) = \bot$; $y \mapsto (\{2\} \sqcap \{3\}) = \bot$; $z \mapsto \bot + \bot = \bot$
- Computing meets late:
  - $z \mapsto 5$

⇒ Constant propagation is not distributive

EXERCISE

Definition: An *interprocedural control flow graph* (ICFG) for a program consists of:

- the CFGs of the individual functions in the program
- the entry and exit node of each function's CFG are connected to their call sites via call and return edges

Problem: Suppose we add function pointers to C--

- What analysis would we need to construct a program's ICFG?
- What can we say about its precision?