Global Optimization (2): Dead Code Elimination

Martin Kellogg

Course Announcements

- PA3 deadline is now Monday, April 14 AoE
  - I have also moved the PA4c1 deadline to 4/28

Course Announcements

- PA3 deadline is now Monday, April 14 AoE
  - I have also moved the PA4c1 deadline to 4/28
- PA4 leaderboard will go live (with your PA3 submissions so far) Sometime Soon™
  - recall that PA4 is due on the day of the final exam
    - which means I cannot grant extensions on it!

Course Announcements

- PA3 deadline is now Monday, April 14 AoE
  - I have also moved the PA4c1 deadline to 4/28
- PA4 leaderboard will go live (with your PA3 submissions so far) Sometime Soon™
  - recall that PA4 is due on the day of the final exam
    - which means I cannot grant extensions on it!
- There was a bug in the reference compiler's implementation of the `in_string()` builtin
  - new version of Cool (v1.40) released
  - I will not test the difference between the two, so it's okay to continue to use v1.39

Agenda

- Global constant folding
- Global liveness analysis
  - this analysis enables dead code elimination
- Interprocedural optimizations (and analysis)

Review: Global Constant Folding

Review: Global Constant Folding

- We want to apply **the same kinds of optimizations** at the global level that we do at the local and regional levels
  - constant folding, DCE, etc.

Review: Global Constant Folding

- We want to apply **the same kinds of optimizations** at the global level that we do at the local and regional levels
  - constant folding, DCE, etc.
  - “global” = an analysis of the **entire control-flow graph** for one method body

Review: Global Constant Folding

- We want to apply **the same kinds of optimizations** at the global level that we do at the local and regional levels
  - constant folding, DCE, etc.
  - "global" = an analysis of the **entire control-flow graph** for one method body
- To replace a use of *x* by a constant *k* we must know this *correctness condition*:

  On every path to the use of x, the last assignment to x is x := k

Review: Global Constant Folding

- We want to apply **the same kinds of optimizations** at the global level that we do at the local and regional levels
  - constant folding, DCE, etc.
  - "global" = an analysis of the **entire control-flow graph** for one method body
- To replace a use of *x* by a constant *k* we must know this *correctness condition*:

  On every path to the use of x, the last assignment to x is x := k
- This correctness condition is **not trivial** to check
  - "**All paths**" includes paths around loops and through branches of conditionals

Review: Dataflow Analysis

- We use a **dataflow analysis** to check the correctness condition

Review: Dataflow Analysis

- We use a **dataflow analysis** to check the correctness condition
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph

Review: Dataflow Analysis

- We use a **dataflow analysis** to check the correctness condition
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph
  - alternate view: it’s just another **abstract interpretation**

Review: Dataflow Analysis

- We use a **dataflow analysis** to check the correctness condition
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph
  - alternate view: it’s just another **abstract interpretation**
- Our dataflow analyses for enabling optimizations will be **sound and conservative**

Review: Dataflow Analysis

- We use a **dataflow analysis** to check the correctness condition
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph
  - alternate view: it’s just another **abstract interpretation**
- Our dataflow analyses for enabling optimizations will be **sound and conservative**
  - i.e., we will **not optimize** when they say “I don’t know”

Review: Dataflow Analysis

- We use a **dataflow analysis** to check the correctness condition
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph
  - alternate view: it’s just another **abstract interpretation**
- Our dataflow analyses for enabling optimizations will be **sound and conservative**
  - i.e., we will **not optimize** when they say “I don’t know”
  - we can’t check the correctness condition directly because it is **undecidable** (as a direct corollary of Rice’s Theorem)

Review: Dataflow Analysis

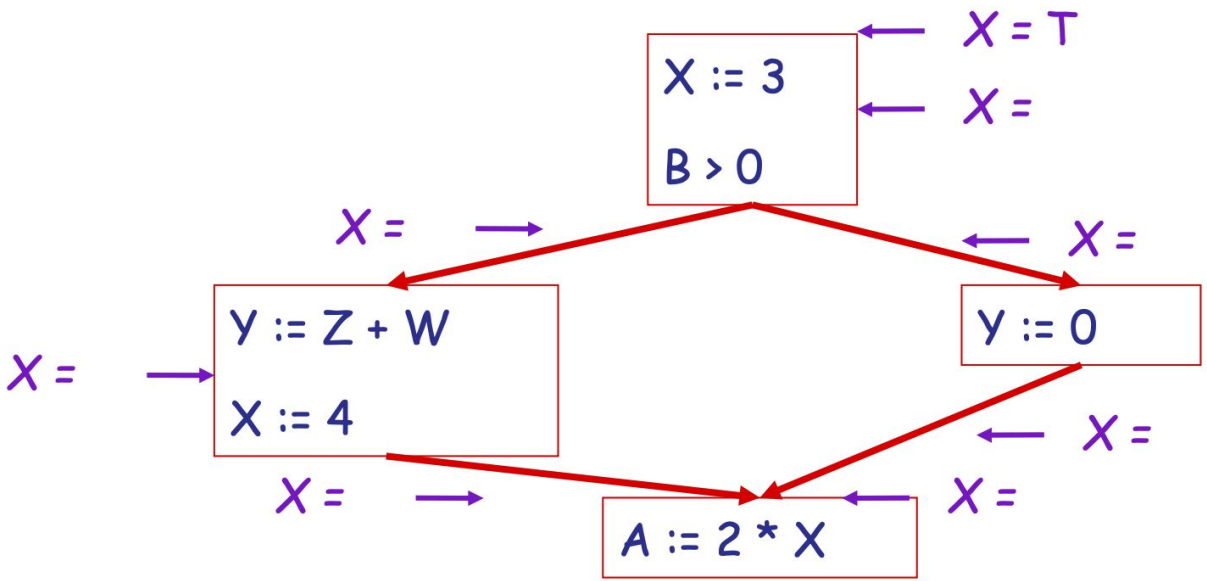
- We use a **dataflow analysis** to check the correctness condition
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph
  - alternate view: it’s just another **abstract interpretation**
- Our dataflow analyses for enabling optimizations will be **sound and conservative**
  - i.e., we will **not optimize** when they say “I don’t know”
  - we can’t check the correctness condition directly because it is **undecidable** (as a direct corollary of Rice’s Theorem)
- Given global constant information, it is **easy to decide** whether or not to perform the optimization

Review: Global Constant Folding Abstraction

- To make the problem precise, we associate one of the following **abstract values** with X at every program point:
  - T (“top”) = “don’t know if X is a constant”
  - constant *c* = “the last assignment to X was X = c”
  - ⊥ (“bottom”) = “X has no value here”

Global Constant Folding: Formalized

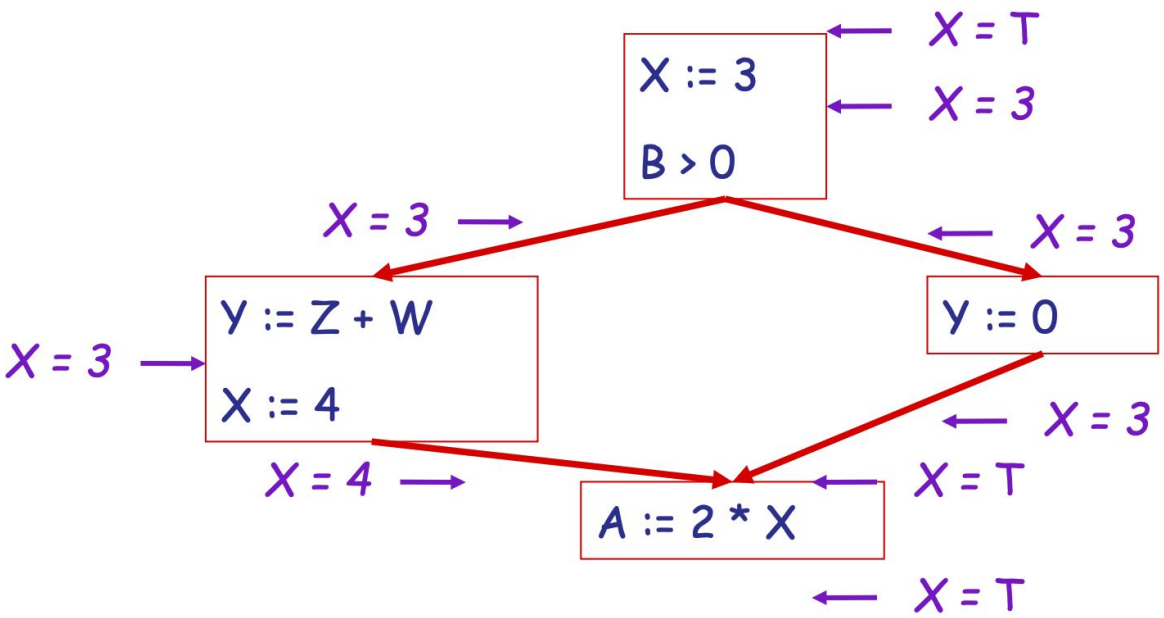
Get out a piece of paper. Fill in these blanks:



Recall:
T = “don’t know”
c = constant
⊥ = unreachable

Global Constant Folding: Formalized

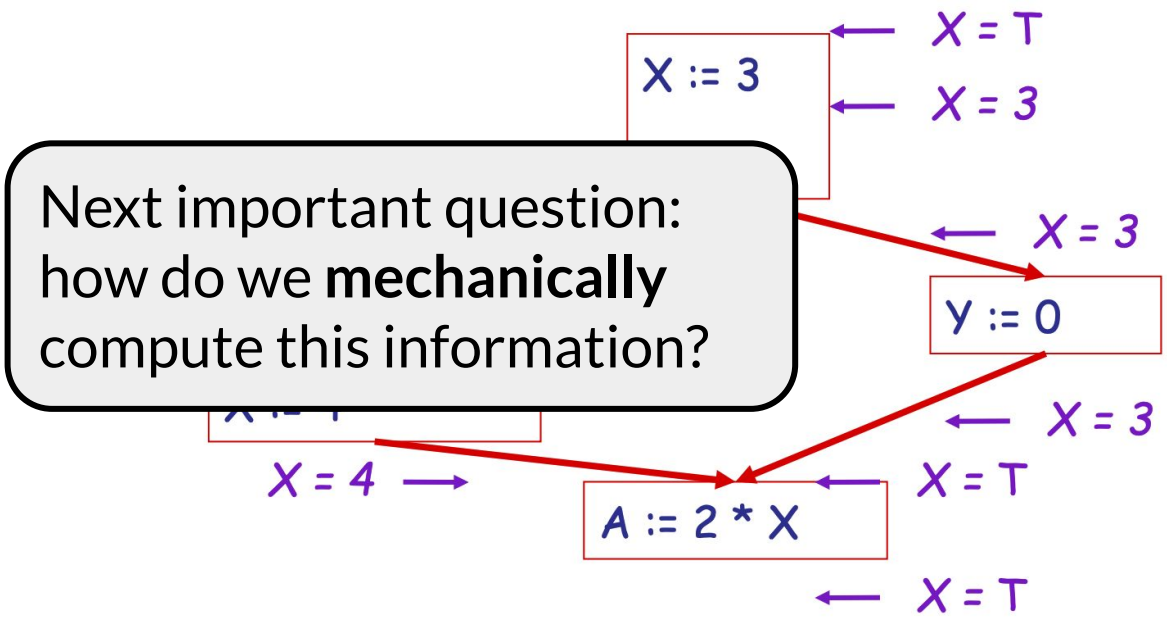
Get out a piece of paper. Fill in these blanks:



Recall:
T = “don’t know”
c = constant
⊥ = unreachable

Global Constant Folding: Formalized

Get out a piece of paper. Fill in these blanks:



Next important question: how do we **mechanically** compute this information?

Recall:
T = “don’t know”
c = constant
⊥ = unreachable

Key Idea of Dataflow Analysis

The analysis of a complicated program can be expressed as a combination of simple rules relating the change in information between adjacent statements

Key Idea of Dataflow Analysis

Explanation:

Key Idea of Dataflow Analysis

Explanation:

- The idea is to “push” or “***transfer***” information from one statement to the next

Key Idea of Dataflow Analysis

Explanation:

- The idea is to “push” or “***transfer***” information from one statement to the next
- For each statement s, we compute information about the value of x immediately before and after s:

Key Idea of Dataflow Analysis

Explanation:

- The idea is to “push” or “***transfer***” information from one statement to the next
- For each statement s, we compute information about the value of x immediately before and after s:
  - $C_{in}(x,s)$ = value of x before s
  - $C_{out}(x,s)$ = value of x after s

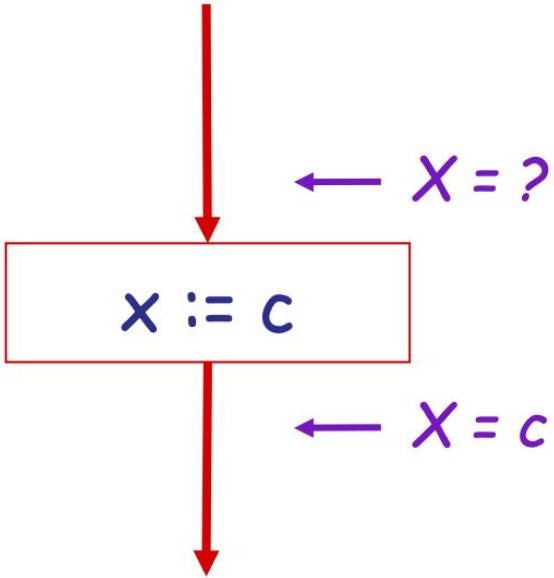
Key Idea of Dataflow Analysis

Explanation:

- The idea is to "push" or "***transfer***" information from one statement to the next
- For each statement s, we compute information about the value of x immediately before and after s:
  - $C_{in}(x,s)$ = value of x before s
  - $C_{out}(x,s)$ = value of x after s

Definition: a ***transfer function*** expresses the relationship between $C_{in}(x, s)$ and $C_{out}(x, s)$

Transfer functions: rule 1

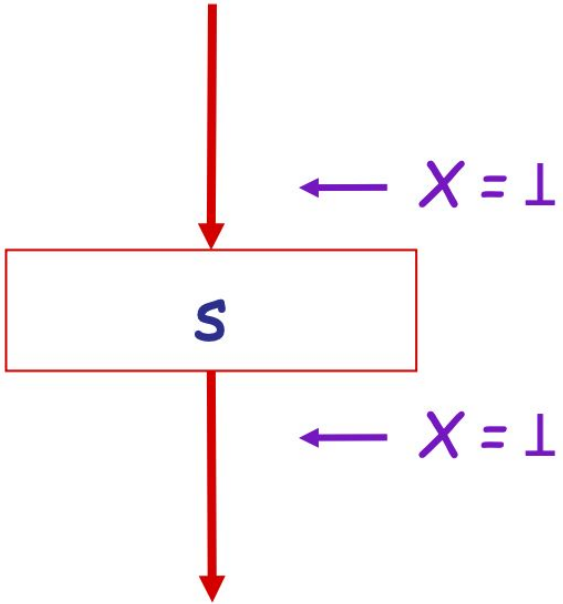


$C_{out}(x, x := c) = c$ if c is a constant

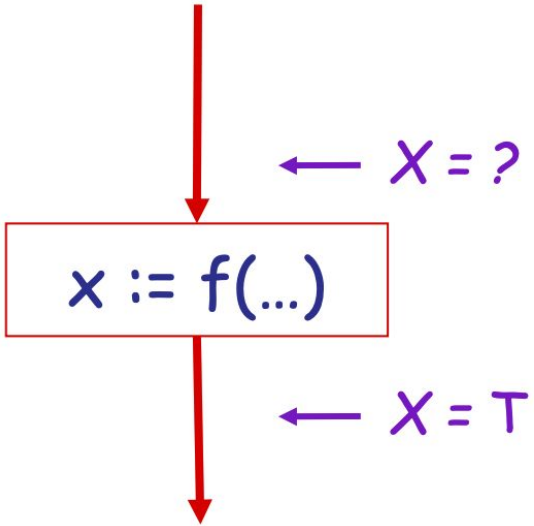
Transfer functions: rule 2

$C_{out}(x, s) = \text{bottom if } C_{in}(x, s) = \text{bottom}$

Recall bottom = “unreachable code”

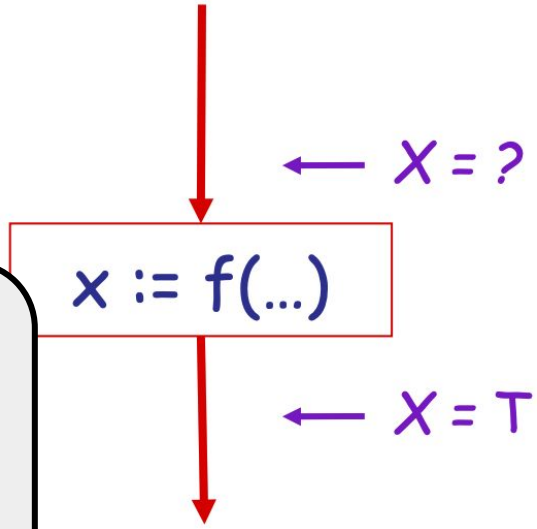


Transfer functions: rule 3



$C_{out}(x, x := f(...)) = T$

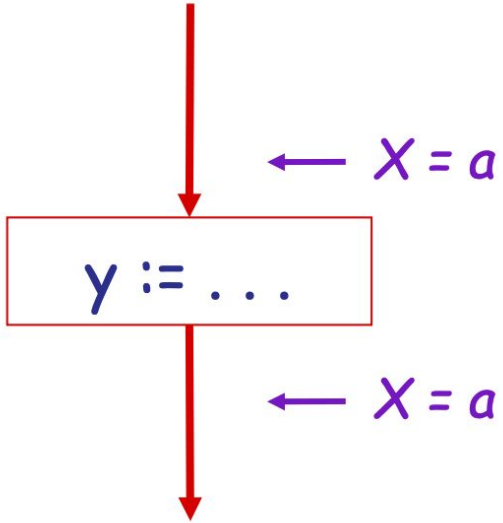
Transfer functions: rule 3



This is a **conservative approximation**! f(...) might always return a constant, but we don't even try!

$C_{out}(x, x := f(...)) = T$

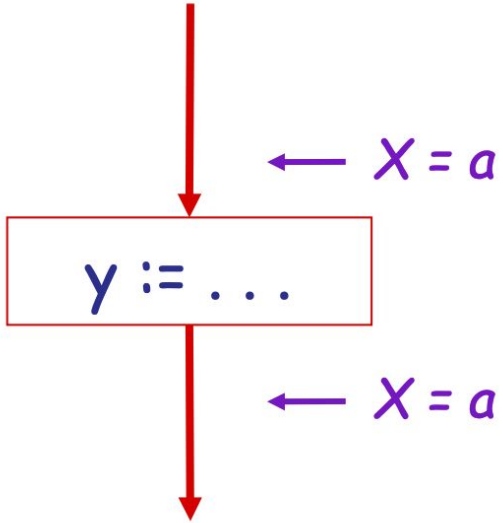
Transfer functions: rule 4



$C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ if $x \neq y$

Transfer functions: rule 4

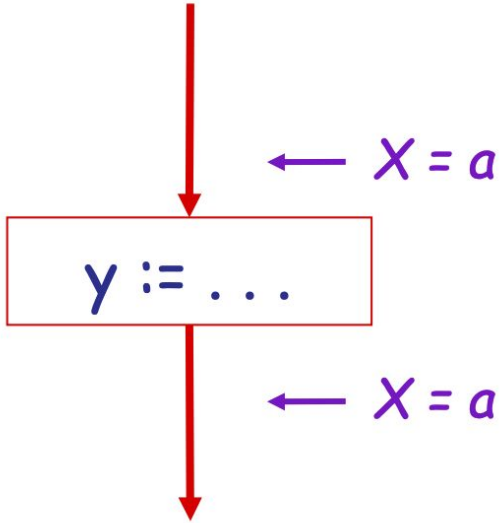
How hard is it to check if x ≠ y on all executions?



$C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ if $x \neq y$

Transfer functions: rule 4

How hard is it to check if x ≠ y on all executions? (oh no)



$C_{out}(x, y := \ldots) = C_{in}(x, y := \ldots)$ if $x \neq y$

Propagation between Statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement

Propagation between Statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements

Propagation between Statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements
- We also need rules relating the *out* of one statement to the *in* of the successor statement

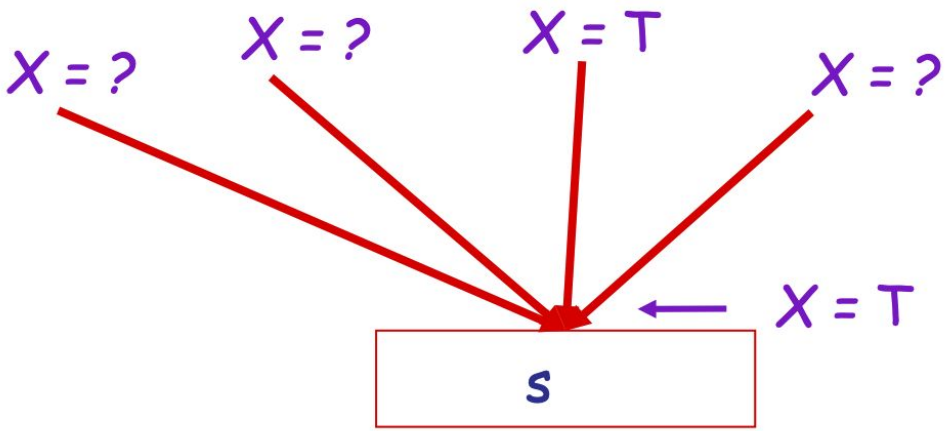
Propagation between Statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements
- We also need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information **forward** along paths

Propagation between Statements

- Rules 1-4 relate the *in* of a statement to the *out* of the same statement
  - they propagate information **across** statements
- We also need rules relating the *out* of one statement to the *in* of the successor statement
  - to propagate information **forward** along paths
- In the following rules, let statement s have immediate predecessor statements p_1 , …, p_n

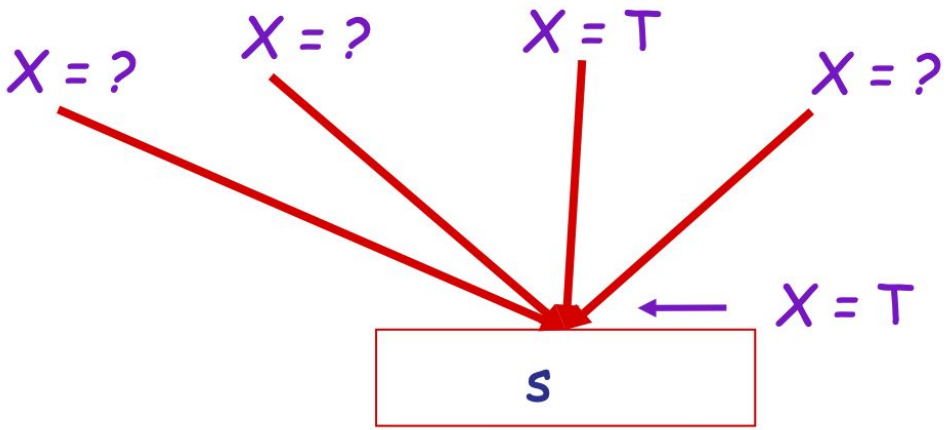
Transfer functions: rule 5



if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

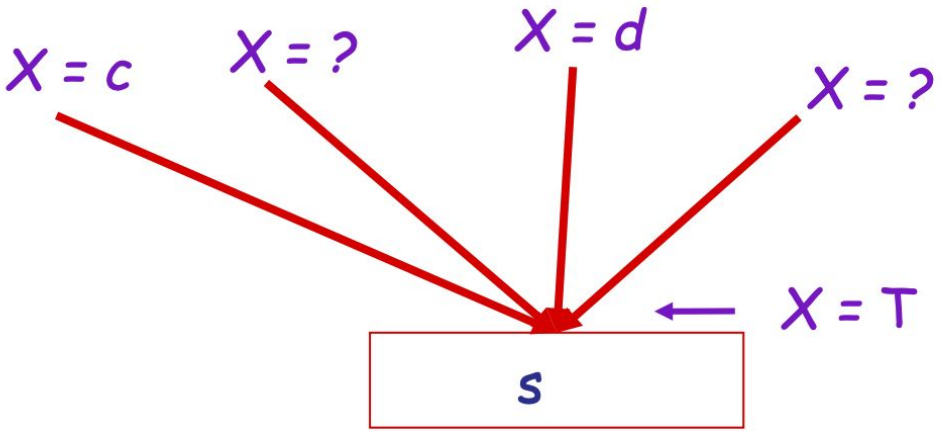
Transfer functions: rule 5

If there's **any** path on which we don't know, then we don't know at all



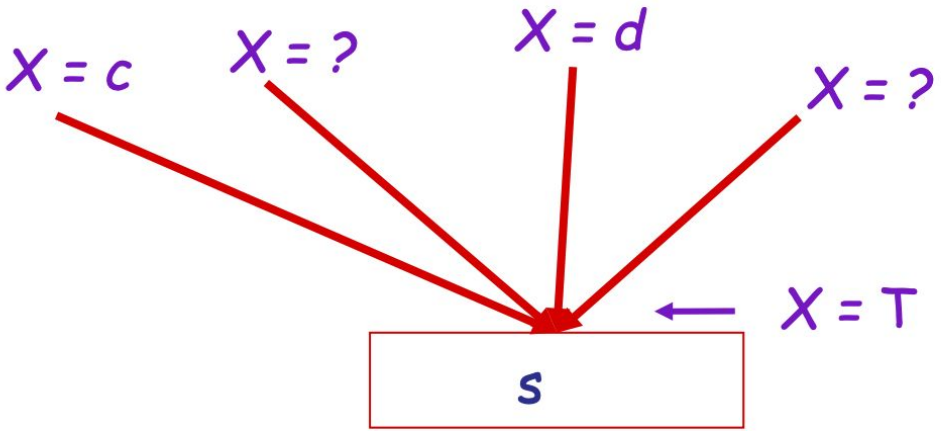
if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

Transfer functions: rule 6



if $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$ then $C_{in}(x, s) = T$

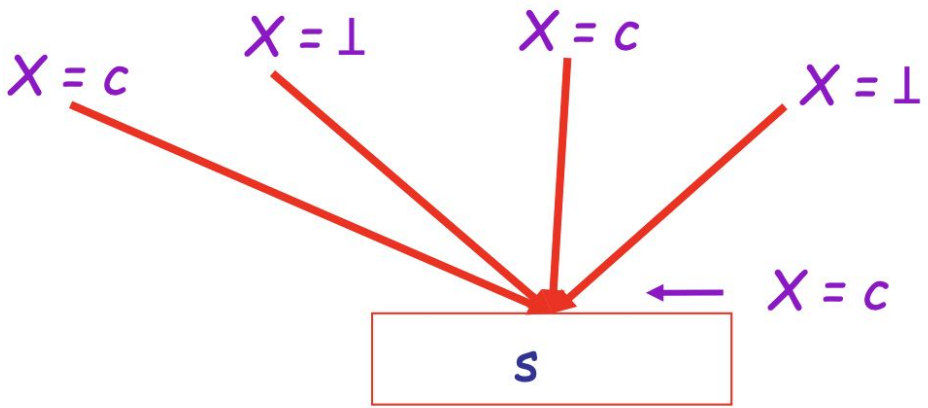
Transfer functions: rule 6



We **don't know** which of the paths a given execution will take (so assume T)

if $C_{out}(x, p_i) = c$ and $C_{out}(x, p_j) = d$ and $d \neq c$ then $C_{in}(x, s) = T$

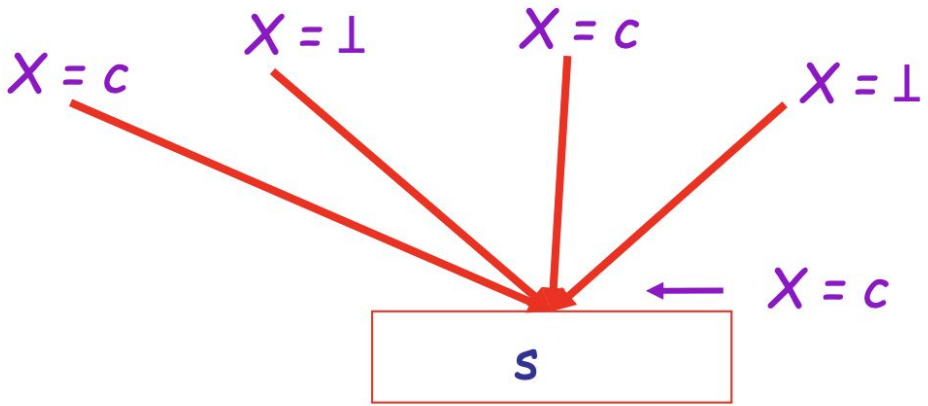
Transfer functions: rule 7



if $C_{out}(x, p_i) = c$ or bottom for all i, then $C_{in}(x, s) = c$

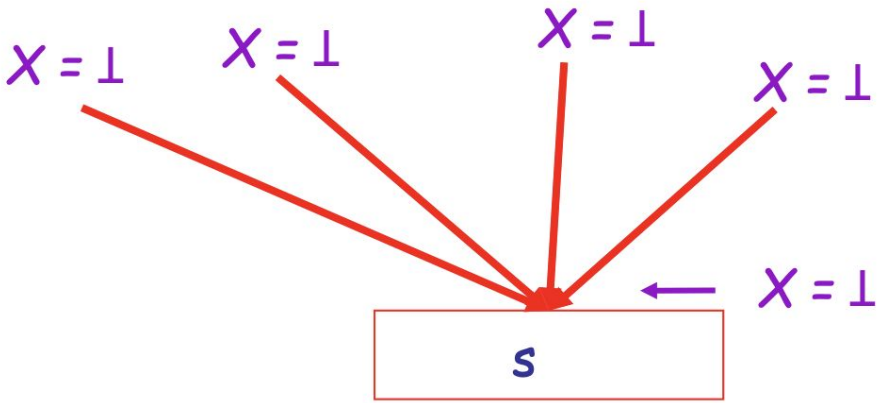
Transfer functions: rule 7

If x has the **same** value (or bottom) on all input edges, it has that value in s



if $C_{out}(x, p_i) = c$ or bottom for all i, then $C_{in}(x, s) = c$

Transfer functions: rule 8



if $C_{out}(x, p_i) = \text{bottom}$ for all i, then $C_{in}(x, s) = \text{bottom}$

A Dataflow Analysis Algorithm

A Dataflow Analysis Algorithm

- For every **entry point** *e* to the procedure, set $C_{in}(x, e) = T$

A Dataflow Analysis Algorithm

- For every **entry point** *e* to the procedure, set $C_{in}(x, e) = T$
  - why top? Top models “we don’t know”, and we don’t know the inputs to the procedure.

A Dataflow Analysis Algorithm

- For every **entry point** *e* to the procedure, set $C_{in}(x, e) = T$
  - why top? Top models “we don’t know”, and we don’t know the inputs to the procedure.
- Set $C_{in}(x, s) = C_{out}(x, s) =$ bottom everywhere else

A Dataflow Analysis Algorithm

- For every **entry point** *e* to the procedure, set $C_{in}(x, e) = T$
  - why top? Top models "we don't know", and we don't know the inputs to the procedure.
- Set $C_{in}(x, s) = C_{out}(x, s) =$ bottom everywhere else
- **Repeat** until all points satisfy rules 1-8:
  - Pick s not satisfying rules 1-8 and update using the appropriate rule

A Dataflow Analysis

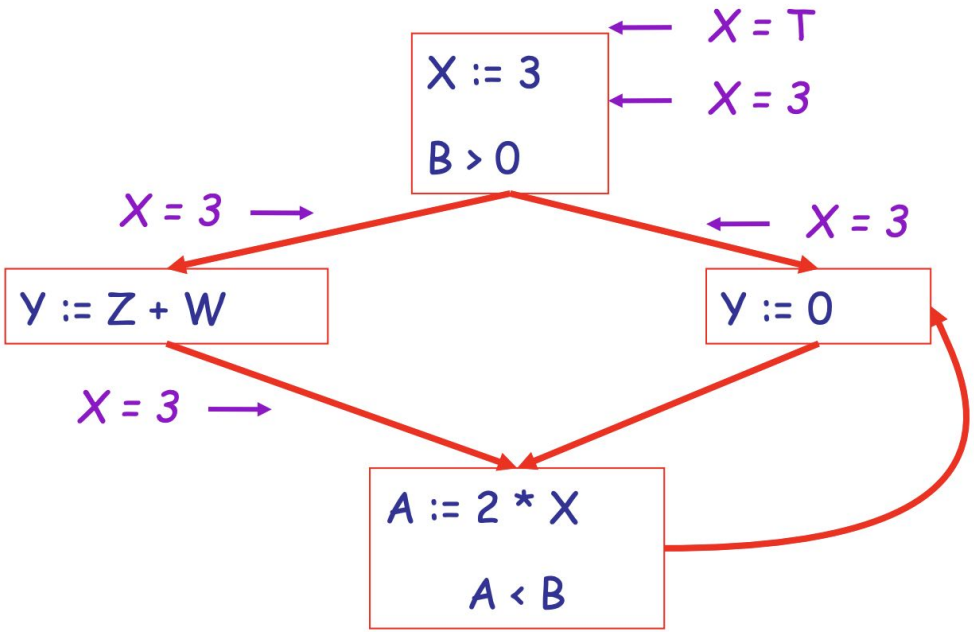
- For every **entry point** *e* to
  - why top? Top models
    inputs to the procedu
- Set $C_{in}(x, s) = C_{out}(x, s) =$ bo
- **Repeat** until all points satisfy rules 1-8:
  - Pick s not satisfying rules 1-8 and update using the appropriate rule

This is a ***fixpoint*** (or ***fixed point***) ***iteration*** algorithm. Such algorithms are characterized by a finite set of rules, which are applied until they "reach fixpoint", which means that applying any rule produces no change.

Why do we need bottom?

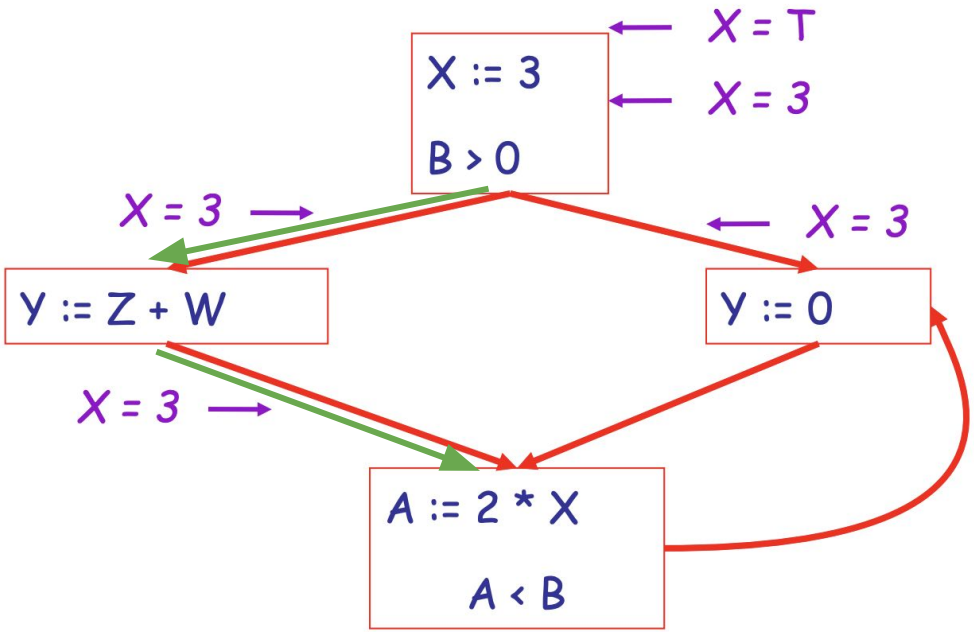
Why do we need bottom?

- To understand why we need to set non-entry points to bottom initially, consider a program with a loop:



Why do we need bottom?

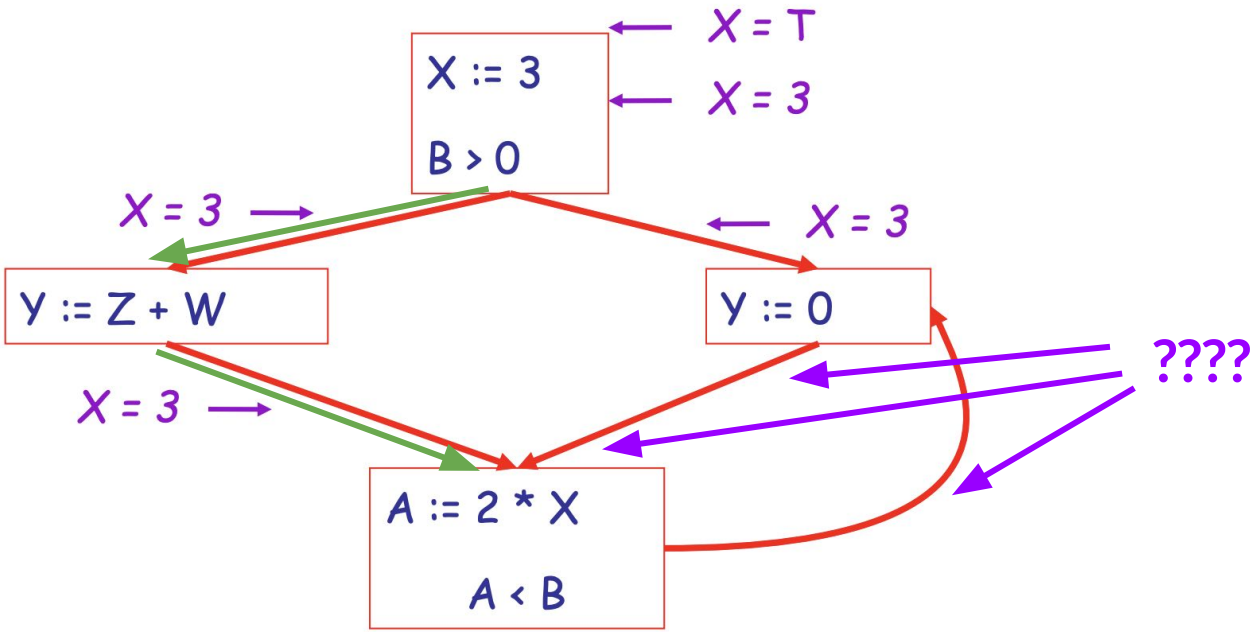
- To understand why we need to set non-entry points to bottom initially, consider a program with a loop:



This way is easy!

Why do we need bottom?

- To understand why we need to set non-entry points to bottom initially, consider a program with a loop:



This way is easy!

Why do we need bottom?

- To understand why we need to set non-entry points to bottom initially, consider a program with a loop.
- Because of **cycles**, all points must have values at all times during the analysis

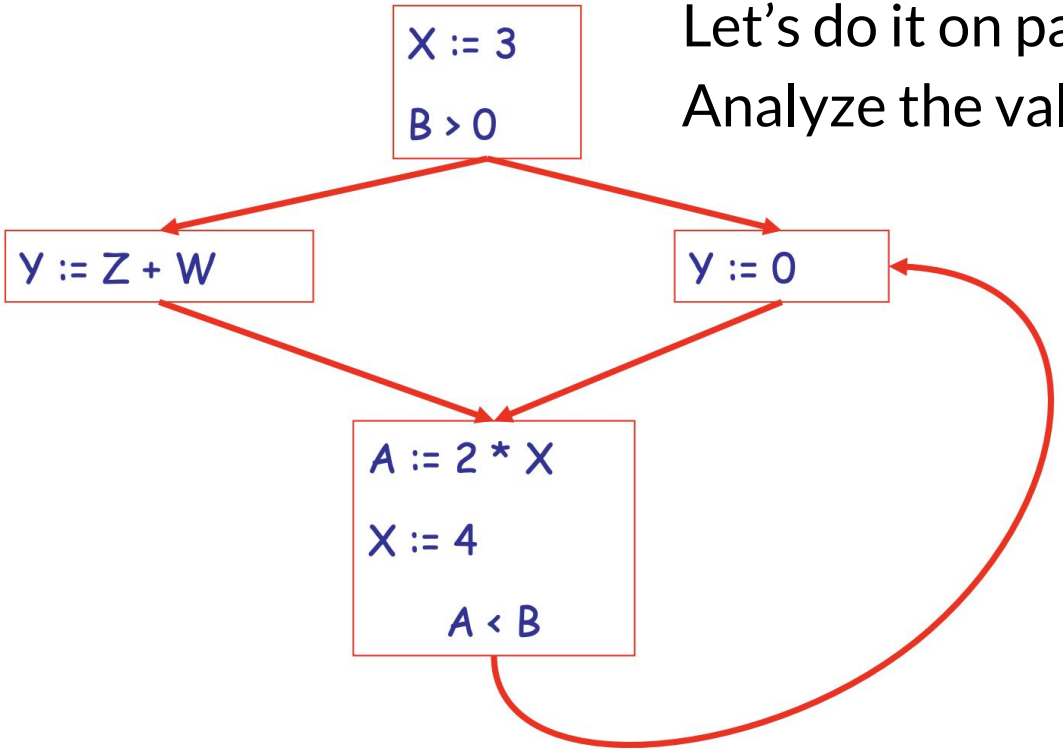
Why do we need bottom?

- To understand why we need to set non-entry points to bottom initially, consider a program with a loop.
- Because of **cycles**, all points must have values at all times during the analysis
- Intuitively, assigning some initial value allows the analysis to **break cycles**

Why do we need bottom?

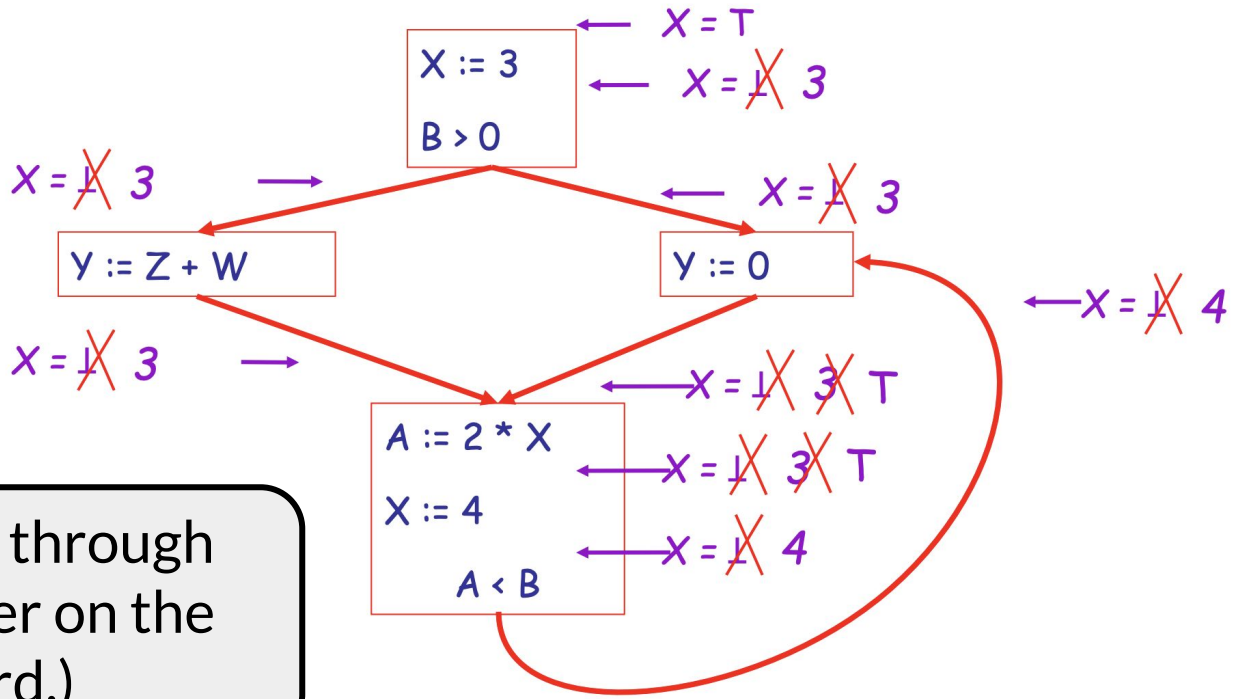
- To understand why we need to set non-entry points to bottom initially, consider a program with a loop.
- Because of **cycles**, all points must have values at all times during the analysis
- Intuitively, assigning some initial value allows the analysis to **break cycles**
- The initial value bottom means "**we have not yet analyzed control reaching this point**"

Another example: dealing with loops



Let's do it on paper!
Analyze the value of X.

Another example: dealing with loops



(We went through this answer on the whiteboard.)

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our dataflow analysis

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our dataflow analysis
  - (Most) locations start as bottom

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our dataflow analysis
  - (Most) locations start as bottom
  - Locations whose current value is bottom might become *c* or T

Lattices & Orderings

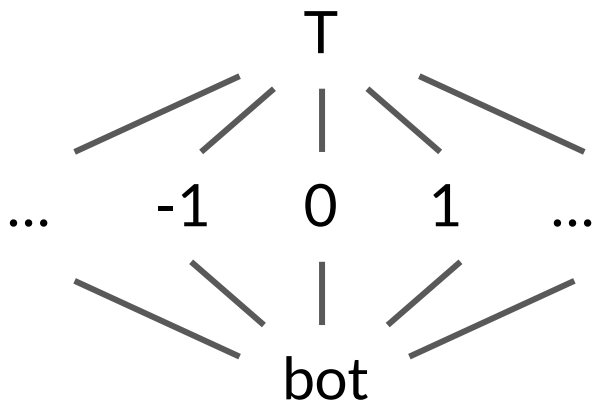
- You may have observed that there is a natural ***order*** to the different abstract values in our dataflow analysis
  - (Most) locations start as bottom
  - Locations whose current value is bottom might become *c* or T
  - Locations whose current value is *c* might become T
    - but never go back to bottom!

Lattices & Orderings

- You may have observed that there is a natural ***order*** to the different abstract values in our dataflow analysis
  - (Most) locations start as bottom
  - Locations whose current value is bottom might become *c* or T
  - Locations whose current value is *c* might become T
    - but never go back to bottom!
  - Locations whose current value is T never change

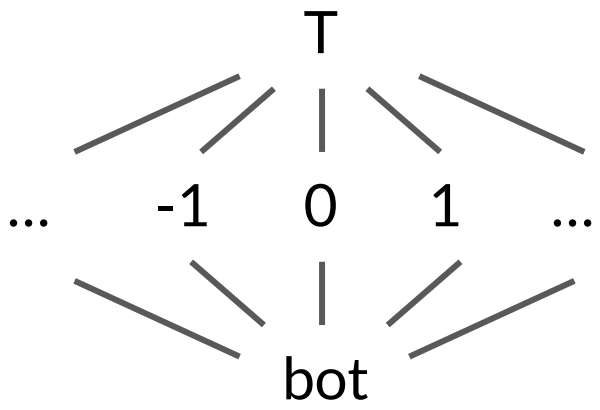
Lattices & Orderings

This structure between values is a ***lattice*** (just like in AI!):



Lattices & Orderings

This structure between values is a ***lattice*** (just like in AI!):



Review of how to read a lattice:

- abstract values **higher** in the lattice are **more general** (e.g., T is true of more things than 0)
- easy to compute ***least upper bound***: it's the lowest common ancestor of two abstract values

Lattices (continued)

- least upper bound (“**lub**”) has useful properties:

Lattices (continued)

- least upper bound (“**lub**”) has useful properties:
  - ***monotonicity*: implicitly captures that values only flow in one direction as the analysis progresses**

Lattices (continued)

- least upper bound ("**lub**") has useful properties:
  - ***monotonicity*: implicitly captures that values only flow in one direction as the analysis progresses
  - we can rewrite rules 5-8 in our dataflow analysis using lub:

    $C_{in}(x, s) = \text{lub}\{ C_{out}(x, p) \mid p \text{ is a predecessor of } s \}$

Lattices (continued)

- least upper bound ("**lub**") has useful properties:
  - ***monotonicity*: implicitly captures that values only flow in one direction as the analysis progresses**
  - we can rewrite rules 5-8 in our dataflow analysis using lub:

$C_{in}(x, s) = \text{lub}\{ C_{out}(x, p) \mid p \text{ i}$

lub is the reason dataflow analysis is an **algorithm**: because lub is monotonic, we only need to analyze each loop **as many times as the lattice is tall**

Termination

- Let’s formalize the argument that our global constant folding analysis terminates
  - saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes, after all

Termination

- Let’s formalize the argument that our global constant folding analysis terminates
  - saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes, after all
- The use of **lub** explains why the algorithm terminates:

Termination

- Let’s formalize the argument that our global constant folding analysis terminates
  - saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes, after all
- The use of **lub** explains why the algorithm terminates:
  - values start as bottom and only increase

Termination

- Let’s formalize the argument that our global constant folding analysis terminates
  - saying “repeat until nothing changes” doesn’t guarantee that eventually nothing changes, after all
- The use of **lub** explains why the algorithm terminates:
  - values start as bottom and only increase
  - bottom can change to a constant, and a constant to T

Termination

- Let's formalize the argument that our global constant folding analysis terminates
  - saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes, after all
- The use of **lub** explains why the algorithm terminates:
  - values start as bottom and only increase
  - bottom can change to a constant, and a constant to T
  - thus, C_(x, s) can change at most **twice** (= lattice height minus one)

Trivia Break: Music

BBC disc jockey John Peel said of this 1969 experimental rock album, "If there has been anything in the history of popular music which could be described as a work of art in a way that people who are involved in other areas of art would understand, then [this album] is probably that work." Its unconventional musical style, which includes polyrhythm, and polytonality, has given the album a reputation as one of the most challenging recordings in the 20th century musical canon. The album was recorded in a single six-hour session after the band had rehearsed it for eight months (!!) straight.

Name either the album or the band that created it.

Trivia Break: Art History

This early-20th-century avant-garde art movement that began in Paris revolutionized painting and the visual arts, and influenced artistic innovations in music, ballet, literature, and architecture. Its subjects are analyzed, broken up, and reassembled in an abstract form: instead of depicting objects from a single perspective, the artist depicts the subject from multiple perspectives to represent the subject in a greater context. Important artists in the movement include Pablo Picasso, Georges Braque, Jean Metzinger, Albert Gleizes, Robert Delaunay, Henri Le Fauconnier, Juan Gris, Fernand Léger, and others.

Picasso's *Girl with a Mandolin (Fanny Tellier)*

Liveness Analysis

Liveness Analysis

- Next, we want to do **global dead code elimination**

Liveness Analysis

- Next, we want to do **global dead code elimination**
  - To prove that removing a statement is safe, we use a ***liveness analysis*** that computes the set of variables that are live at each program point

Liveness Analysis

- Next, we want to do **global dead code elimination**
  - To prove that removing a statement is safe, we use a ***liveness analysis*** that computes the set of variables that are live at each program point
- Formally, we say that a variable *x* is ***live*** at statement *s* if:

Liveness Analysis

- Next, we want to do **global dead code elimination**
  - To prove that removing a statement is safe, we use a ***liveness analysis*** that computes the set of variables that are live at each program point
- Formally, we say that a variable *x* is ***live*** at statement *s* if:
  - There exists a statement *s'* that uses *x*

Liveness Analysis

- Next, we want to do **global dead code elimination**
  - To prove that removing a statement is safe, we use a ***liveness analysis*** that computes the set of variables that are live at each program point
- Formally, we say that a variable *x* is ***live*** at statement *s* if:
  - There exists a statement *s'* that uses *x*
  - There is a path from *s* to *s'*

Liveness Analysis

- Next, we want to do **global dead code elimination**
  - To prove that removing a statement is safe, we use a ***liveness analysis*** that computes the set of variables that are live at each program point
- Formally, we say that a variable *x* is ***live*** at statement *s* if:
  - There exists a statement *s'* that uses *x*
  - There is a path from *s* to *s'*
  - That path has no intervening assignment to *x*

Liveness Analysis

- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation

Liveness Analysis

- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation
  - Liveness is simpler than constant propagation, since it is a boolean property (true or false)

Liveness Analysis

- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation
  - Liveness is simpler than constant propagation, since it is a boolean property (true or false)
- I will show two **different formalisms** for liveness in these slides:

Liveness Analysis

- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation
  - Liveness is simpler than constant propagation, since it is a boolean property (true or false)
- I will show two **different formalisms** for liveness in these slides:
  - An “abstract interpretation”-like one (transfer functions)

Liveness Analysis

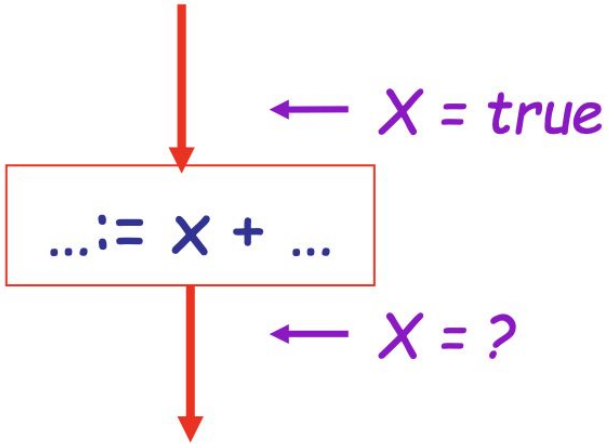
- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation
  - Liveness is simpler than constant propagation, since it is a boolean property (true or false)
- I will show two **different formalisms** for liveness in these slides:
  - An “abstract interpretation”-like one (transfer functions)
  - A “gen-kill”-style system of equations

Liveness Analysis

- We can express liveness in terms of information transferred between adjacent statements, just as in constant propagation
  - Liveness is simpler than constant propagation, since it is a boolean property (true or false)
- I will show two **different formalisms** for liveness in these slides:
  - An “abstract interpretation”-like one (transfer functions)
  - A “gen-kill”-style system of equations
- These formalisms are **equally valid**. You can use which makes the most sense to you when you’re implementing PA4c1/on exams/etc

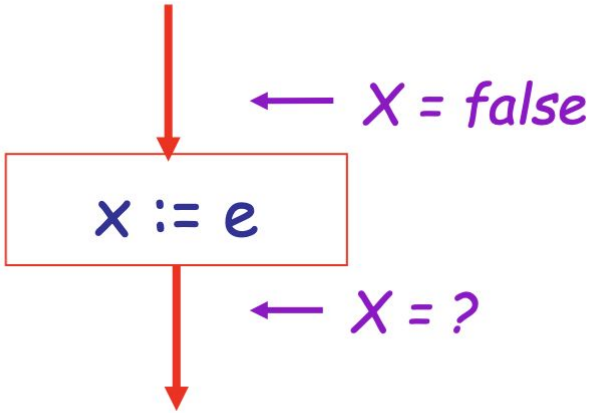
Liveness Analysis: As Rules

Liveness Analysis: Rule 1



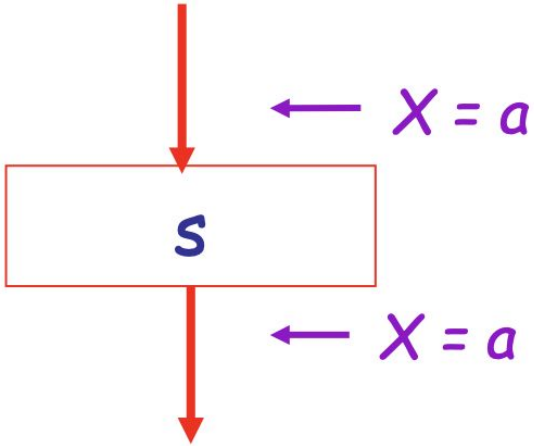
$L_{in}(x,s)$ = true if s refers to x on the rhs

Liveness Analysis: Rule 2



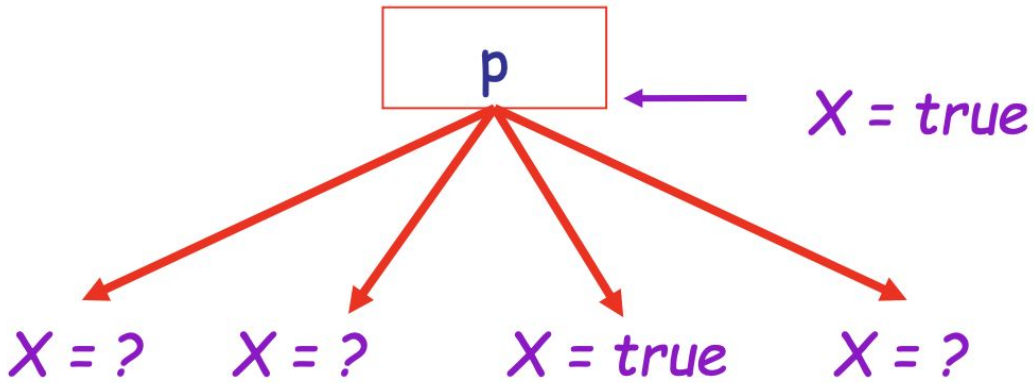
$L_{in}(x, x := e) = false$ if e does not refer to x

Liveness Analysis: Rule 3



$L_{in}(x, s) = L_{out}(x, s)$ if s does not refer to x

Liveness Analysis: Rule 4



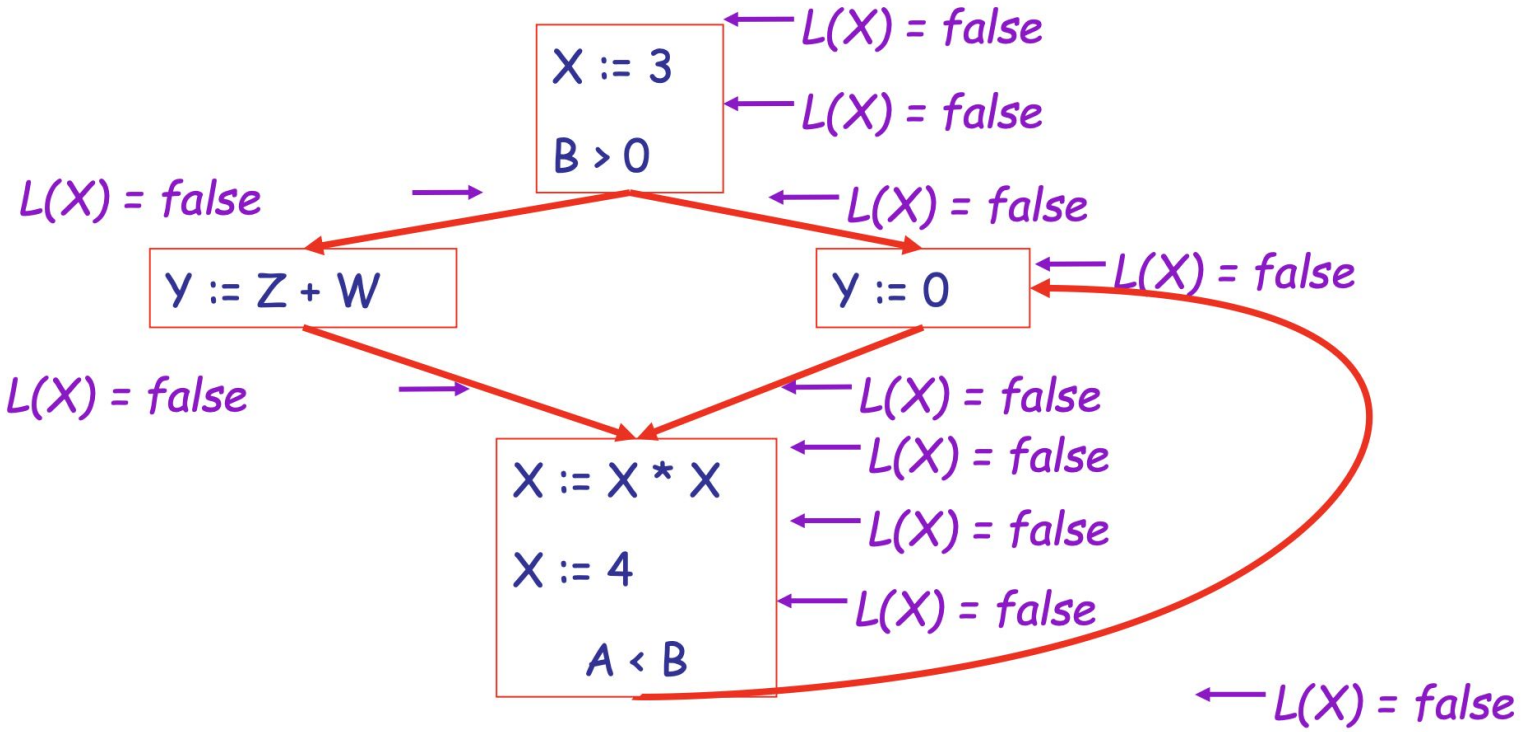
$L_{out}(x, p) = \bigvee \{ L_{in}(x, s) \mid s \text{ is a successor of } p \}$

Liveness Analysis: Algorithm

- Let all L_(...) = false initially
- Repeat until all statements s satisfy rules 1-4 :
  - Pick s where one of rules 1-4 does not hold and update using the appropriate rule

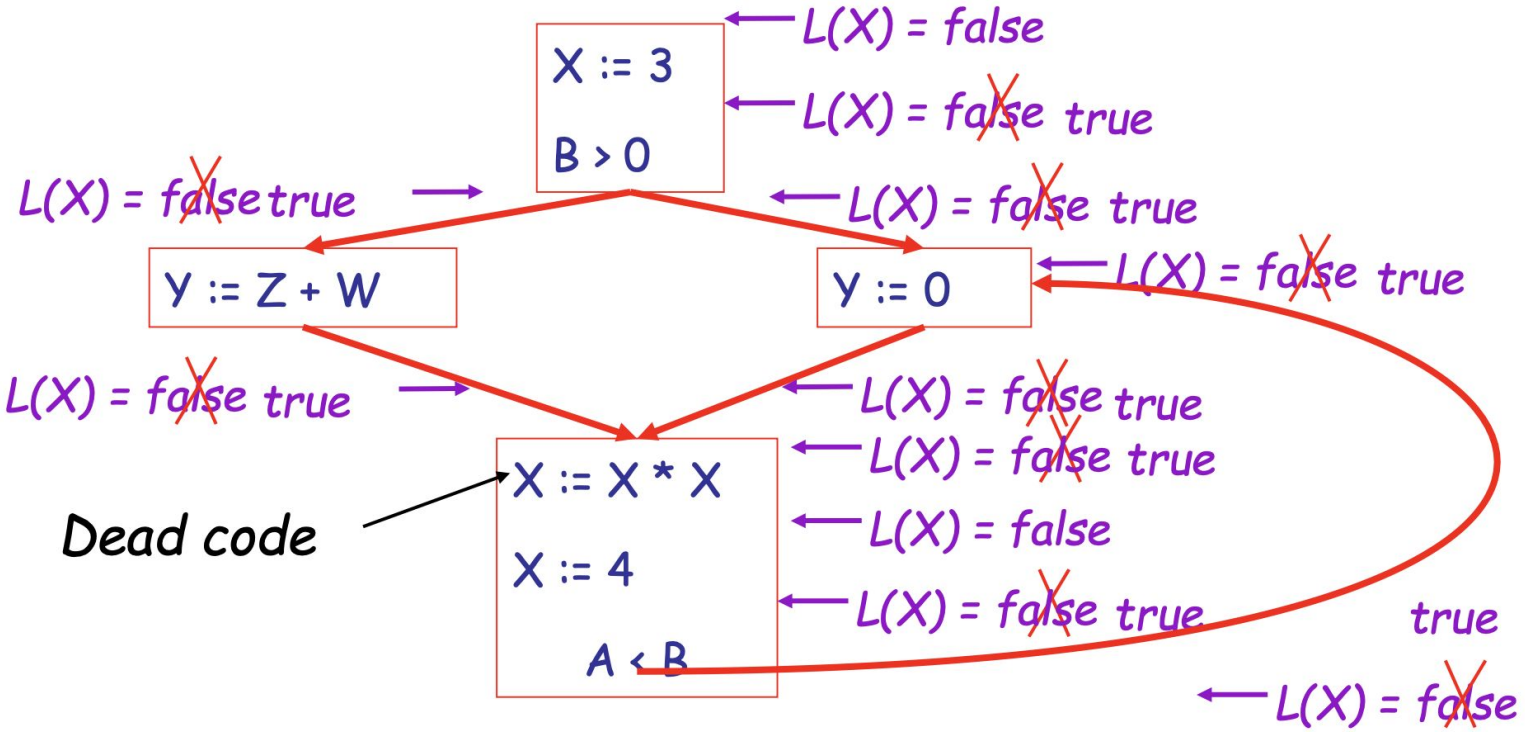
(on the whiteboard)

Liveness Analysis: Example



(on the whiteboard)

Liveness Analysis: Example



Liveness Analysis: Dataflow Equations

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:
  - *IN*(b) = set of facts that are true on entry to b

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:
  - *IN*(b) = set of facts that are true on entry to b
  - *OUT*(b) = set of facts that are true on exit from b

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:
  - *IN*(b) = set of facts that are true on entry to b
  - *OUT*(b) = set of facts that are true on exit from b
  - *GEN*(b) = set of facts created and not killed during b

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:
  - *IN*(b) = set of facts that are true on entry to b
  - *OUT*(b) = set of facts that are true on exit from b
  - *GEN*(b) = set of facts created and not killed during b
  - *KILL*(b) = set of facts killed during b

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:
  - *IN*(b) = set of facts that are true on entry to b
  - *OUT*(b) = set of facts that are true on exit from b
  - *GEN*(b) = set of facts created and not killed during b
  - *KILL*(b) = set of facts killed during b
- These sets are related by the equation:

$$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

Liveness Analysis: Dataflow Equations

- Alternate formalism: all dataflow analyses involve **four sets of facts** about each basic block b:
  - *IN*(b) = set of facts that
  - *OUT*(b) = set of facts th
  - *GEN*(b) = set of facts cr
  - *KILL*(b) = set of facts ki
- These sets are related by the equation:

$$OUT(b) = GEN(b) \cup (IN(b) - KILL(b))$$

Alternative algorithm for dataflow analysis: solve this ***dataflow equation*** iteratively for all blocks until all sets reach fixpoint

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) =
  - *OUT*(b) =
  - *GEN*(b) =
  - *KILL*(b) =

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) = live variables on entry to b
  - *OUT*(b) =
  - *GEN*(b) =
  - *KILL*(b) =

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) = live variables on entry to b
  - *OUT*(b) = live variables on exit from b
  - *GEN*(b) =
  - *KILL*(b) =

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) = live variables on entry to b
  - *OUT*(b) = live variables on exit from b
  - *GEN*(b) = variables ***used by*** b
  - *KILL*(b) =

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) = live variables on entry to b
  - *OUT*(b) = live variables on exit from b
  - *GEN*(b) = variables ***used by*** b
  - *KILL*(b) = variables ***defined by*** b

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) = live variables on entry to b
  - *OUT*(b) = live variables on exit from b
  - *GEN*(b) = variables ***used by*** b
  - *KILL*(b) = variables ***defined by*** b
- Why does GEN = “used variables” and KILL = “defined variables”?

Liveness Analysis: Dataflow Equations

- To use this framework to model liveness, we need to define these four sets:
  - *IN*(b) = live variables on entry to b
  - *OUT*(b) = live variables on exit from b
  - *GEN*(b) = variables ***used by*** b
  - *KILL*(b) = variables ***defined by*** b
- Why does GEN = “used variables” and KILL = “defined variables”?
  - a variable **becomes live** because of a use
  - a variable is **no longer live** because of a definition

Dataflow Equations vs Transfer Functions

- Either way of thinking about dataflow analysis is **equally valid**

Dataflow Equations vs Transfer Functions

- Either way of thinking about dataflow analysis is **equally valid**
  - I prefer transfer functions, but you might prefer dataflow equations

Dataflow Equations vs Transfer Functions

- Either way of thinking about dataflow analysis is **equally valid**
  - I prefer transfer functions, but you might prefer dataflow equations
  - We can easily prove that anything that we can express with one is can be expressed with the other

Dataflow Equations vs Transfer Functions

- Either way of thinking about dataflow analysis is **equally valid**
  - I prefer transfer functions, but you might prefer dataflow equations
  - We can easily prove that anything that we can express with one is can be expressed with the other
- There is a much more in-depth treatment of dataflow equations (including how to use them for liveness analysis) in *EaC* chapter 9.
  - If you're confused, **start there**

Liveness Analysis: Other Uses

- Liveness analysis is useful for many things in a compiler, not just dead code elimination

Liveness Analysis: Other Uses

- Liveness analysis is useful for many things in a compiler, not just dead code elimination
- Some examples of other uses:

Liveness Analysis: Other Uses

- Liveness analysis is useful for many things in a compiler, not just dead code elimination
- Some examples of other uses:
  - **Register allocation**: only live variables need a register

Liveness Analysis: Other Uses

- Liveness analysis is useful for many things in a compiler, not just dead code elimination
- Some examples of other uses:
  - **Register allocation**: only live variables need a register
  - Detecting uses of **uninitialized variables**: if live at declaration (before initialization) then it might be used uninitialized

Liveness Analysis: Other Uses

- Liveness analysis is useful for many things in a compiler, not just dead code elimination
- Some examples of other uses:
  - **Register allocation**: only live variables need a register
  - Detecting uses of **uninitialized variables**: if live at declaration (before initialization) then it might be used uninitialized
  - Improve **SSA construction**: only need Φ-functions for variables that are live in a block (later)

Liveness Analysis: Summary

Liveness Analysis: Summary

- A variable is ***live*** at a program point iff there is any path from that program point to a use of the variable along which the variable is not redefined

Liveness Analysis: Summary

- A variable is ***live*** at a program point iff there is any path from that program point to a use of the variable along which the variable is not redefined
- **Two ways** to do liveness analysis (equally-valid!):
  - transfer functions
  - dataflow equations

Liveness Analysis: Summary

- A variable is ***live*** at a program point iff there is any path from that program point to a use of the variable along which the variable is not redefined
- **Two ways** to do liveness analysis (equally-valid!):
  - transfer functions
  - dataflow equations
- Liveness analysis has lots of uses:
  - DCE, register allocation, detecting uses of uninitialized variables, etc.

Interprocedural Optimizations

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today
- However, procedure calls also introduce **significant overhead**
  - pre-call/post-return bookkeeping, prologue/epilogue, jump

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today
- However, procedure calls also introduce **significant overhead**
  - pre-call/post-return bookkeeping, prologue/epilogue, jump
- Calls are also **hard to reason about** in global optimizations
  - compiler doesn't know what will happen inside the call

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize each procedure **independently** using global analyses like those we've discussed today
- However, procedure calls also introduce **significant overhead**
  - pre-call/post-return bookkeeping, prologue/epilogue, jump
- Calls are also **hard to reason about** in global optimizations
  - compiler doesn't know what will happen inside the call
- These downsides of procedure calls motivate ***interprocedural*** (or "***whole-program***") optimizations that span procedure boundaries

Interprocedural Optimizations

- Dividing the program up into procedures give one big benefit: **separate compilation**
  - we can also optimize e
    global analyses like tho
- However, procedure calls
  - pre-call/post-return bo
- Calls are also **hard to reason about** in global optimizations
  - compiler doesn't know what will happen inside the call
- These downsides of procedure calls motivate ***interprocedural*** (or "***whole-program***") optimizations that span procedure boundaries

Today we will take a brief look at two interprocedural optimizations:

- inlining
- tail call optimization

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:
  - the actual procedure body is only one instruction

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:
  - the actual procedure body is only one instruction
  - the prologue and epilogue dominate, we may have to spill registers at call sites, etc.

Interprocedural Optimizations: Inlining

- Consider the following procedure:

```
int add(int x, int y):
   return x + y
```

- Imagine generating code for a call to this procedure:
  - the actual procedure body is only one instruction
  - the prologue and epilogue dominate, we may have to spill registers at call sites, etc.
- Key idea of ***inlining***: for such a procedure call, replace the call with the procedure's body

Inlining: Benefits & Risks

- Inlining is useful when:

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:
  - increases **code size** (may overflow instruction cache)

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedure to be inlined is much shorter than the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:
  - increases **code size** (may overflow instruction cache)
  - increases **register pressure**

Inlining: Benefits & Risks

- Inlining is useful when:
  - the body of the procedu
    the prologue/epilogue
  - inlining enables **specialization** (e.g., arguments are constants)
  - inlining enables **other optimizations** (e.g., part of the body is dead at this particular call site)
- Inlining also has risks:
  - increases **code size** (may overflow instruction cache)
  - increases **register pressure**

Note similar benefits and risks to **loop unrolling**. Deciding whether to inline is similarly complicated!

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?
  - Is the callee procedure **significantly smaller** than the calling procedure?

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?
  - Is the callee procedure **significantly smaller** than the calling procedure?
  - Static ***call count***: the number of distinct sites that call the procedure.
    - Any procedure called just once is a good inlining candidate.

Inlining: Common Heuristics

- In practice, production compilers will use **heuristics** to decide when/if to inline, such as:
  - Is this procedure a **leaf** in the call graph?
    - That is, does it not call any other procedures itself?
  - Is the callee procedure **significantly smaller** than the calling procedure?
  - Static ***call count***: the number of distinct sites that call the procedure.
    - Any procedure called just once is a good inlining candidate.
  - **Profile data**, such as fraction of execution time (if available)

Interprocedural Optimizations: Tail Calls

Interprocedural Optimizations: Tail Calls

- Consider a procedure that calls another procedure and then immediately returns, like this example:

```
int foo(...):
    ...
    return bar(...)
```

- What will happen as `bar` returns?

Interprocedural Optimizations: Tail Calls

- Consider a procedure that calls another procedure and then immediately returns, like this example:

```
int foo(...):
    ...
    return bar(...)
```

- What will happen as `bar` returns?
  - We will execute the epilogues of `foo` and `bar` **in sequence**, with no intervening instructions

Interprocedural Optimizations: Tail Calls

- Consider a procedure that calls another procedure and then immediately returns, like this example:

```
int foo(...):
    ...
    return bar(...)
```

- What will happen as `bar` returns?
  - We will execute the epilogues of `foo` and `bar` **in sequence**, with no intervening instructions
    - Including many **redundant operations** (e.g., resetting `%rsp`)

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue
- This optimization is most important for ***tail-recursive*** procedures that call **themselves** as the last operation in their body
  - e.g., imagine a naive Fibonacci implementation

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue
- This optimization is most important for ***tail-recursive*** procedures that call **themselves** as the last operation in their body
  - e.g., imagine a naive Fibonacci implementation
  - Tail-call elimination often reduces asymptotic stack space requirements from linear to constant for tail-recursive calls

Interprocedural Optimizations: Tail Calls

- ***Tail-call elimination*** is an interprocedural optimization that allows a function called as the last instruction in a procedure to return to the calling procedure's caller directly
  - This eliminates redundant operations in the epilogue
- This optimization is most important for ***tail-recursive*** procedures that call **themselves** as the last operation in their body
  - e.g., imagine a naive Fibonacci implementation
  - Tail-call elimination often reduces asymptotic stack space requirements from linear to constant for tail-recursive calls
- **Functional languages** practically require tail-call elimination

Interprocedural Optimizations: Issues

Interprocedural Optimizations: Issues

- The biggest difficulty in interprocedural optimization is maintaining support for **separate compilation**
  - Traditional "***compilation unit***" is a procedure or file

Interprocedural Optimizations: Issues

- The biggest difficulty in interprocedural optimization is maintaining support for **separate compilation**
  - Traditional "***compilation unit***" is a procedure or file
- If all of the code is definitely available, it suffices to **track dependencies** between procedures from an optimization perspective, and then re-optimize whenever a dependent procedure changes

Interprocedural Optimizations: Issues

- The biggest difficulty in interprocedural optimization is maintaining support for **separate compilation**
  - Traditional "***compilation unit***" is a procedure or file
- If all of the code is definitely available, it suffices to **track dependencies** between procedures from an optimization perspective, and then re-optimize whenever a dependent procedure changes
- Alternatively, we can defer interprocedural optimization until ***link time***, when a ***linker*** combines the object files from each compilation unit into a single executable. We'll talk more about this later.

Course Announcements

- PA3 deadline is now Monday, April 14 AoE
  - I have also moved the PA4c1 deadline to 4/28
- PA4 leaderboard will go live (with your PA3 submissions so far) Sometime Soon™
  - recall that PA4 is due on the day of the final exam
    - which means I cannot grant extensions on it!
- There was a bug in the reference compiler's implementation of the `in_string()` builtin
  - new version of Cool (v1.40) released
  - I will not test the difference between the two, so it's okay to continue to use v1.39