Global Optimizations

Martin Kellogg

Course Announcements

Course Announcements

- As many of you have noticed, there was an issue with the PA3 autograder that prevented anyone from passing tests 16+
  - This issue has been fixed as of this morning
  - Discuss with class: do y'all need another extension on PA3? I am open to giving one.
    - First rule of compilers: semantics >>> optimization

Course Announcements

- As many of you have noticed, there was an issue with the PA3 autograder that prevented anyone from passing tests 16+
  - This issue has been fixed as of this morning
  - Discuss with class: do y'all need another extension on PA3? I am open to giving one.
    - First rule of compilers: semantics >>> optimization
- I will be out of town on Wednesday 4/30 (for ICSE)
  - I've therefore rearranged the calendar a bit
  - My PhD student Erfan Arvan will give a lecture on exceptions

Agenda

- Super-local Value Numbering
- Other regional optimizations
- Intro to global optimizations
- Dataflow analysis basics

Review: Local Value Numbering

Review: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block

Review: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - LVN uses a table to map **names, constants, and expressions** to ***value numbers***

Review: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - LVN uses a table to map **names, constants, and expressions** to ***value numbers***
    - each value number has a 1:1 relationship with the value that it represents

Review: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - LVN uses a table to map **names, constants, and expressions** to ***value numbers***
    - each value number has a 1:1 relationship with the value that it represents
  - LVN table keys are constructed from the value numbers of subexpressions

Review: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - LVN uses a table to map **names, constants, and expressions** to ***value numbers***
    - each value number has a 1:1 relationship with the value that it represents
  - LVN table keys are constructed from the value numbers of subexpressions
- LVN allows us to trivially identify **redundant computations**

Review: Local Value Numbering

- ***Local Value Numbering*** (***LVN***) is a classic algorithm for finding and eliminating redundant operations in a basic block
  - LVN uses a table to map **names, constants, and expressions** to ***value numbers***
    - each value number has a 1:1 relationship with the value that it represents
  - LVN table keys are constructed from the value numbers of subexpressions
- LVN allows us to trivially identify **redundant computations**
  - it's straightforward to extend it to other local optimizations, like constant folding

Review: Regional Optimizations

Review: Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together

Review: Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy

Review: Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target

Review: Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target
- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks

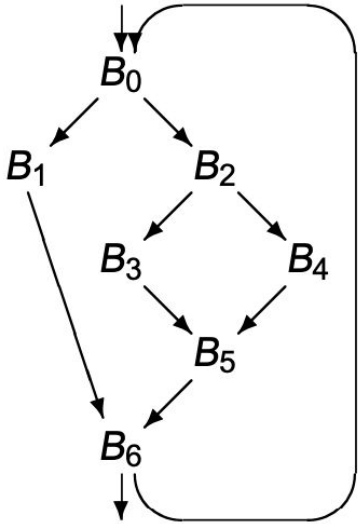
Review: Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target
- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks
  - Most local optimizations can operate on EBBs with small modifications

Review: Regional Optimizations

- A ***regional*** optimization considers one or more logically-related basic blocks together
  - These blocks are not required to form a whole procedure
    - that's a "global" optimization; the boundary is fuzzy
  - However, usually they're "related" in some way: for example, the whole body of a loop may be an optimization target
- Regional optimizations usually work on an ***extended basic block*** ("***EBB***"): a small control-flow graph of basic blocks
  - Most local optimizations can operate on EBBs with small modifications
- Formally, an EBB is a **maximal** collection of basic blocks with unique entry and exit blocks

Extended Basic Blocks: Example



B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_2$

B_1: $p_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_6$

B_2: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_3, B_4$

B_3: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$
$\rightarrow B_5$

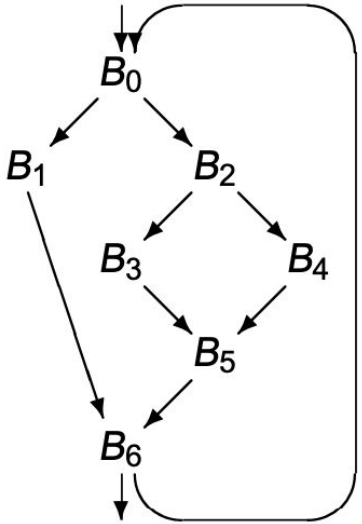
B_4: $e_1 \leftarrow a_0 + 17$
$t_0 \leftarrow c_0 + d_0$
$u_1 \leftarrow e_1 + f_0$
$\rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$u_2 \leftarrow \phi(u_0, u_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow c_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_6$

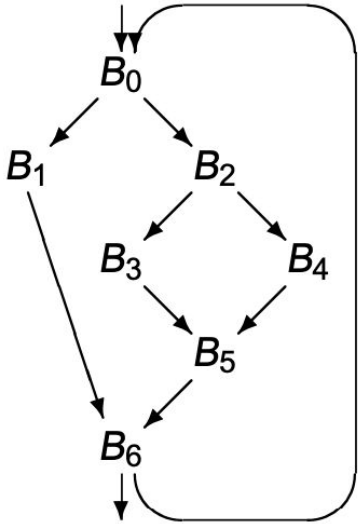
B_6: $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$

(don’t worry about the details)

Super-local Value Numbering

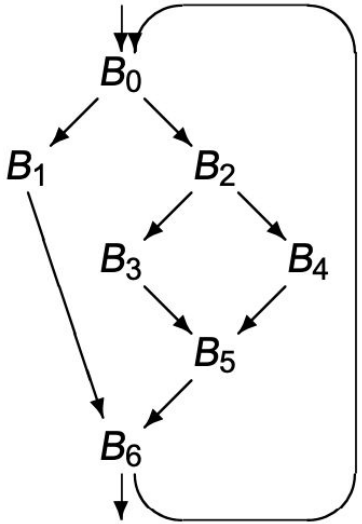


Super-local Value Numbering



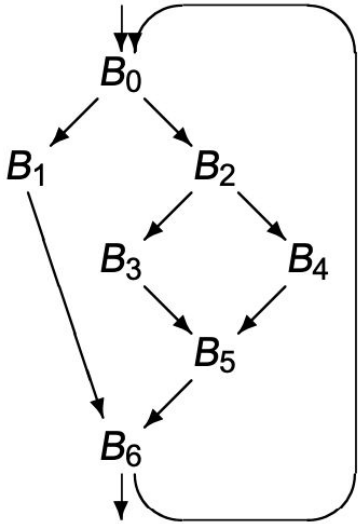
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB

Super-local Value Numbering



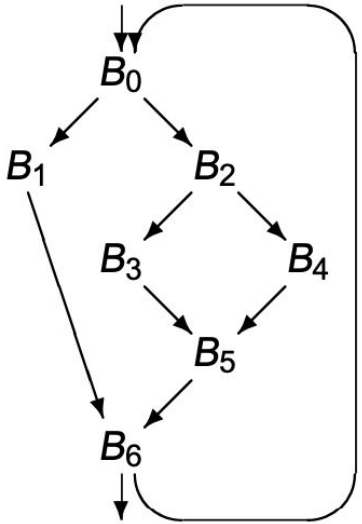
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**

Super-local Value Numbering



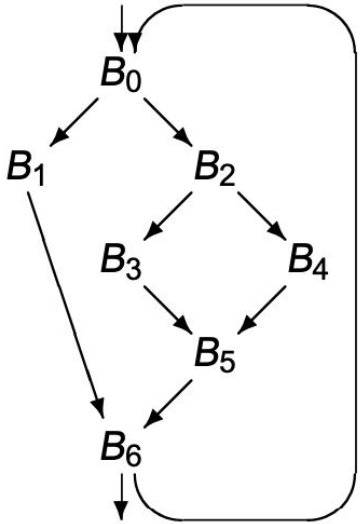
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**
  - That is, treat each path as if it were a block!
    - after all, no branches in a single path…

Super-local Value Numbering



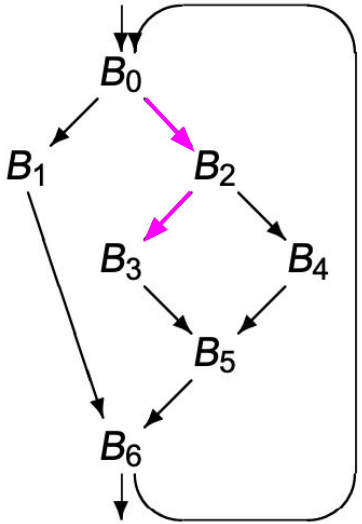
- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**
  - That is, treat each path as if it were a block!
    - after all, no branches in a single path…
- Blocks with single predecessor can keep the hashtable from the last block

Super-local Value Numbering



- To extend LVN to more than one basic block, we need to reason about **all possible paths** through the EBB
- In theory, we can consider each path **independently**
  - That is, treat each path as if it were a block!
    - after all, no branches in a single path…
- Blocks with single predecessor can keep the hashtable from the last block
- Any block with multiple predecessors, such as B_5, can use a fresh hashtable

S-LVN: Example



consider the path B_0, B_2, B_3

B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$(a_0 > b_0) \rightarrow B_1, B_2$

B_1: $p_0 \leftarrow c_0 + d_0$
$r_0 \leftarrow c_0 + d_0$
$\rightarrow B_6$

B_2: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$(a_0 > b_0) \rightarrow B_3, B_4$

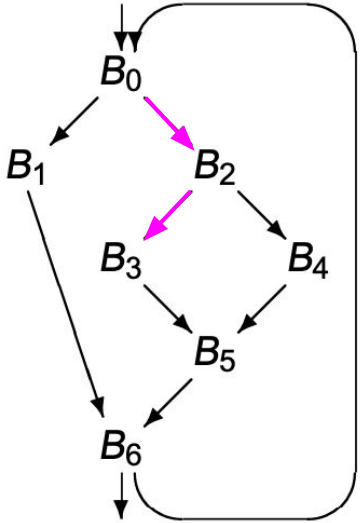
B_3: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$
$\rightarrow B_5$

B_4: $e_1 \leftarrow a_0 + 17$
$t_0 \leftarrow c_0 + d_0$
$u_1 \leftarrow e_1 + f_0$
$\rightarrow B_5$

B_5: $e_2 \leftarrow \phi(e_0, e_1)$
$u_2 \leftarrow \phi(u_0, u_1)$
$v_0 \leftarrow a_0 + b_0$
$w_0 \leftarrow c_0 + d_0$
$x_0 \leftarrow e_2 + f_0$
$\rightarrow B_6$

B_6: $r_2 \leftarrow \phi(r_0, r_1)$
$y_0 \leftarrow a_0 + b_0$
$z_0 \leftarrow c_0 + d_0$

S-LVN: Example

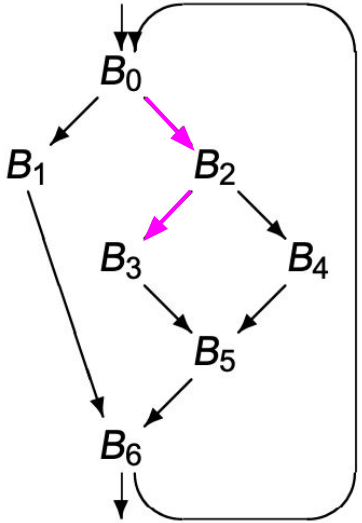


consider the path B_0, B_2, B_3

combine into a single logical block

B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$

S-LVN: Example



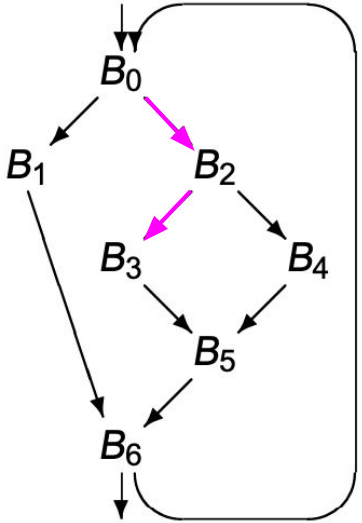
consider the path B_0, B_2, B_3

combine into a single logical block

B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
$q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
$e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
$u_0 \leftarrow e_0 + f_0$

LVN can discover this redundancy on its own…

S-LVN: Example



consider the path B_0, B_2, B_3

combine into a single logical block

B_0: $m_0 \leftarrow a_0 + b_0$
$n_0 \leftarrow a_0 + b_0$
B_1: $q_0 \leftarrow a_0 + b_0$
$r_1 \leftarrow c_0 + d_0$
B_2: $e_0 \leftarrow b_0 + 18$
$s_0 \leftarrow a_0 + b_0$
B_3: $u_0 \leftarrow e_0 + f_0$

17
d_0
f_0
$,e_1)$
$,u_1)$
b_0
d_0
f_0
$,r_1)$
b_0
d_0

...but we need S-LVN to discover these two!

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work
  - insight: paths share **common prefixes**
    - e.g., (B_0, B_2, B_3) and (B_0, B_2, B_4) share prefix (B_0, B_2)

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work
  - insight: paths share **common prefixes**
    - e.g., (B_0, B_2, B_3) and (B_0, B_2, B_4) share prefix (B_0, B_2)
  - the compiler can **cache** the results for common prefixes and reuse them when analyzing related paths

S-LVN: Practicality

- Unfortunately, analyzing each path separately isn't feasible, because paths **grow exponentially** in the number of branches
  - this is called the ***path explosion problem***
    - it impacts a number of important static analyses that work at path granularity (most famously symbolic execution)
- Regional optimizations can capitalize on the **tree structure** of an EBB, though, to avoid redoing too much work
  - insight: paths share **common prefixes**
    - ... and (B_0, B_2, B_4) share prefix (B_0, B_2)
    - ...ne the results for common prefixes and ...alyzing related paths

For more details on this algorithm, see the book.

Other Regional Optimizations

- Loop unrolling
- Code motion
- Loop induction variable elimination

Loop Unrolling

Loop Unrolling

- To ***unroll*** a loop, replicate the loop's body and adjust the logic that controls the number of iterations performed

Loop Unrolling

- To ***unroll*** a loop, replicate the loop's body and adjust the logic that controls the number of iterations performed
- Direct benefits:
  - reduce number of branches (they're expensive)
  - enable reuse of certain computations (e.g., outer loop indices)
  - improve spatial locality, especially for array accesses

Loop Unrolling

- To ***unroll*** a loop, replicate the loop's body and adjust the logic that controls the number of iterations performed
- Direct benefits:
  - reduce number of branches (they're expensive)
  - enable reuse of certain computations (e.g., outer loop indices)
  - improve spatial locality, especially for array accesses
- Loop unrolling changes the **ratio** of arithmetic to memory operations in the loop

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
  - unrolling increases the number of operations in the loop body, which might enable other optimizations
    - e.g., more operations in the loop body might unlock interesting instruction schedules

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
  - unrolling increases the number of operations in the loop body, which might enable other optimizations
    - e.g., more operations in the loop body might unlock interesting instruction schedules
  - unrolling can enable multi-word instructions (i.e., SIMD)
    - ***SIMD*** = "single instruction, multiple data"

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
  - unrolling increases the number of operations in the loop body, which might enable other optimizations
    - e.g., more operations in the loop body might unlock interesting instruction schedules
  - unrolling can enable multi-word instructions (i.e., SIMD)
    - ***SIMD*** = "single instruction, multiple data"
  - unrolled loop may use more registers, and if it causes a spill the unrolling is almost certainly not worth it

Loop Unrolling: Indirect Effects

- Loop unrolling has a number of **indirect effects**, both positive and potentially negative:
  - it increases program size. If this causes the instruction cache to overflow, it's not worthwhile to unroll the loop.
  - unrolling increases the number of operations in the loop body, which might enable other optimizations
    - e.g., more operations in the loop body might unlock
    - uctions (i.e., SIMD)
    - e data"
    - and if it causes a spill
    - the unrolling is almost certainly not worth it

Whether or not to unroll a loop often depends on these factors, so there is **no one-size-fits-all algorithm** for deciding whether to unroll

Code Motion

- Goal: move **loop-invariant calculations** out of loops

Code Motion

- Goal: move **loop-invariant calculations** out of loops
- Example:

```
for (i = 0; i < 10; i++) {
 a[i] = a[i] + b[j];
 z = z + 10000;
}
```

Code Motion

- Goal: move **loop-invariant calculations** out of loops
- Example:

```
for (i = 0; i < 10; i++) {
 a[i] = a[i] + b[j];
 z = z + 10000;
}
```

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i++) {
 a[i] = a[i] + t1;
 z = z + t2;
}
```

Code Motion

- Goal: move **loop-invariant calculations** out of loops
- Example:

```
for (i = 0; i < 10; i++) {
 a[i] = a[i] + b[j];
 z = z + 10000;
}
```

```
t1 = b[j];
t2 = 10000;
for (i = 0; i < 10; i++) {
 a[i] = a[i] + t1;
 z = z + t2;
}
```

- Benefit: avoids **redundant computation** each time around the loop

Loop Induction Variable Elimination

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**

> ***Strength reduction*** means replacing expensive operations with equivalent but less expensive operations
>
> - e.g., `x*2` -> `x+x`

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction variable***
  - incremented each time around loop
  - offsets & pointers calculated from it
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop
  - then do loop-invariant code motion

Loop Induction Variable Elimination

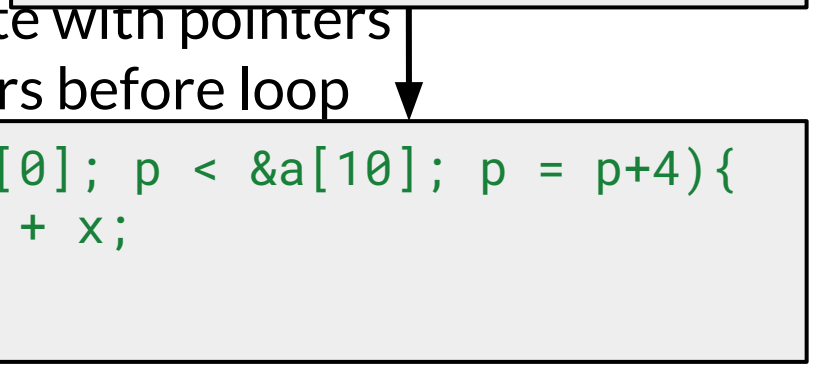
- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction vari***
  - incremented each time around
  - offsets & pointers calculated fr
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offsets/pointers each time around loop
  - no expensive scaling in loop
  - then do loop-invariant code motion

```
for (i = 0; i < 10; i++){
    a[i] = a[i] + x;
}
```

Loop Induction Variable Elimination

- Common special case of loop-based **strength reduction**
- For-loop index is the ***induction vari***
  - incremented each time around
  - offsets & pointers calculated fr
- If used only to index arrays, rewrite with pointers
  - compute initial offsets/pointers before loop
  - increment offset
  - no expensive sca
  - then do loop-inv

```
for (i = 0; i < 10; i++){
    a[i] = a[i] + x;
}
```



```
for (p = &a[0]; p < &a[10]; p = p+4){
    *p = *p + x;
}
```

Summary: Regional Optimizations

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality
  - This means that they are inherently **more risky** than their local cousins

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality
  - This means that they are inherently **more risky** than their local cousins
- Nearly all local optimizations can be extended to work at the regional level

Summary: Regional Optimizations

- Regional optimizations offer **more opportunities** than local optimizations
- Generally operate on **extended basic blocks** with one entry/exit
  - commonly, the **body of a loop**
- Benefits of a regional optimization often depend on **indirect effects**, such as spatial locality
  - This means that they are inherently **more risky** than their local cousins
- Nearly all local optimizations can be extended to work at the regional level
  - Which you want to use is up to you!

Trivia Break: CS + Pop Culture

This American has been a professor of both cognitive science and computer scientist, though his PhD (from the University of Oregon) is actually in Physics. His dissertation work was one of the early examples of modern scientific data visualization. However, he is most famous not for his academic work but for his nonfiction writing: his 2007 book *I Am a Strange Loop* won the Los Angeles Times Book Prize for Science and Technology, and his 1979 book *Gödel, Escher, Bach: An Eternal Golden Braid* won the Pulitzer Prize for general nonfiction and a National Book Award (at that time called The American Book Award) for Science.

Trivia Break: Technology and Business

This Seattle-based ecommerce company's first order from a non-employee was a copy of Douglas Hofstadter's *Fluid Concepts*, on April 3, 1995. The company survived the dot-com crash in the early 2000s and went on to become one of the world's largest companies, with a market capitalization of more than 1.8 trillion USD. Internally, its corporate culture is focused on the concept of a "flywheel" between different parts of the business: that is, that growth in one part should encourage growth in other parts, creating positive feedback loops.

“Global” Optimizations

“Global” Optimizations

- A ***global optimization*** changes an entire method (consisting of multiple basic blocks).

“Global” Optimizations

- A ***global optimization*** changes an entire method (consisting of multiple basic blocks).
- We must be **conservative** and only apply global optimizations when they **preserve the original semantics**.

“Global” Optimizations

- A ***global optimization*** changes an entire method (consisting of multiple basic blocks).
- We must be **conservative** and only apply global optimizations when they **preserve the original semantics**.
- We use ***global dataflow analyses*** to determine if it is OK to apply an optimization.

“Global” Optimizations

- A ***global optimization*** changes an entire method (consisting of multiple basic blocks).
- We must be **conservative** and only apply global optimizations when they **preserve the original semantics**.
- We use ***global dataflow analyses*** to determine if it is OK to apply an optimization.
    - These analyses have a lot in common with abstract interpretation, which we covered earlier in the course

“Global” Optimizations

- A ***global optimization*** changes an entire method (consisting of multiple basic blocks).
- We must be **conservative** and only apply global optimizations when they **preserve the original semantics**.
- We use ***global dataflow analyses*** to determine if it is OK to apply an optimization.
  - These analyses have a lot in common with abstract interpretation, which we covered earlier in the course
  - Like an abstract interpretation, flow analyses are built out of **simple transfer functions** and can work forwards or backwards.

“Global” Optimizations (continued)

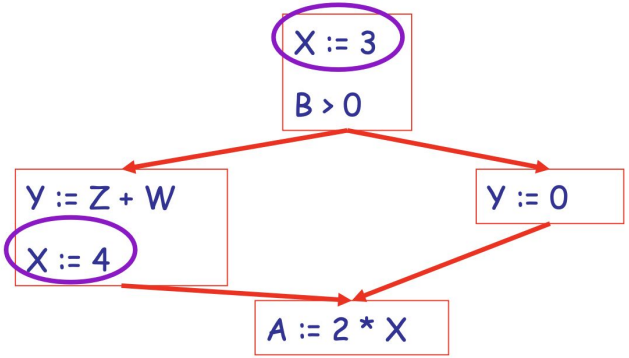
- We want to apply **the same kinds of optimizations** at the global level that we do at the local and regional levels
  - constant folding, DCE, etc.

“Global” Optimizations (continued)

- 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.
- How would we **know** it is OK to globally propagate constants?

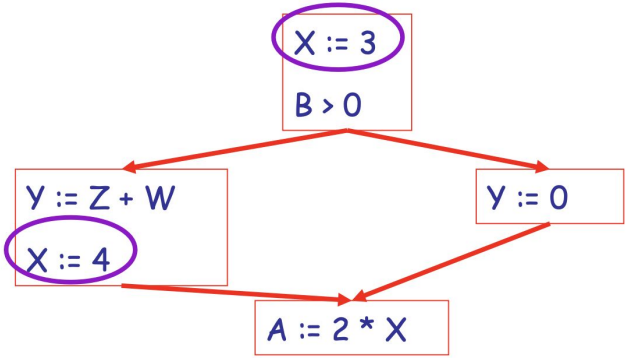
“Global” Optimizations (continued)

- 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.
- How would we **know** it is OK to globally propagate constants?
  - e.g., in the CFG to the left, it’s **not safe** to constant fold X!



“Global” Optimizations (continued)

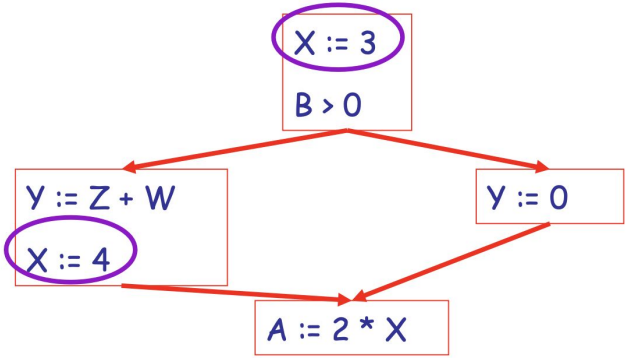
- We want to apply **the same kinds of optimizations** at the global level that we do at the local and regional levels
  - constant folding, DCE, etc.
- How would we **know** it is OK to globally propagate constants?
  - e.g., in the CFG to the left, it’s **not safe** to constant fold X!



- To replace a use of *x* by a constant *k* we must know this ***correctness condition***:

“Global” Optimizations (continued)

- 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.
- How would we **know** it is OK to globally propagate constants?
  - e.g., in the CFG to the left, it’s **not safe** to constant fold X!

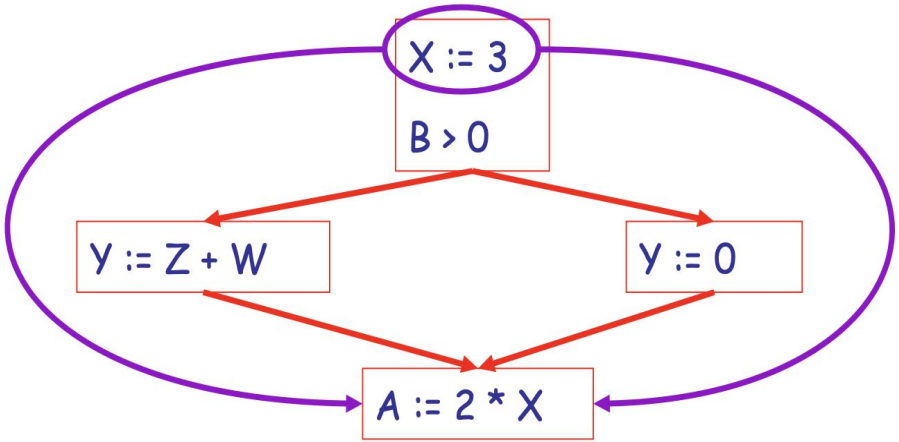


- To replace a use of *x* by a constant *k* we must know this ***correctness condition***:

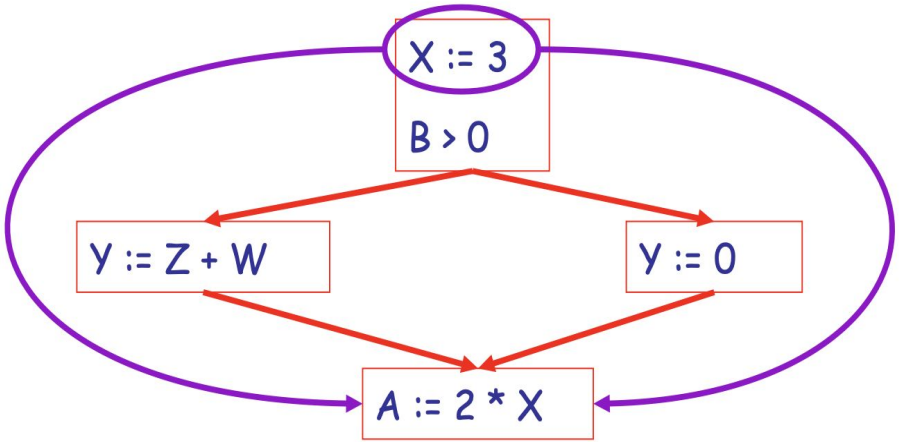
  On every path to the use of x, the last assignment to x is x := k

“Global” Constant Folding: Examples

“Global” Constant Folding: Examples

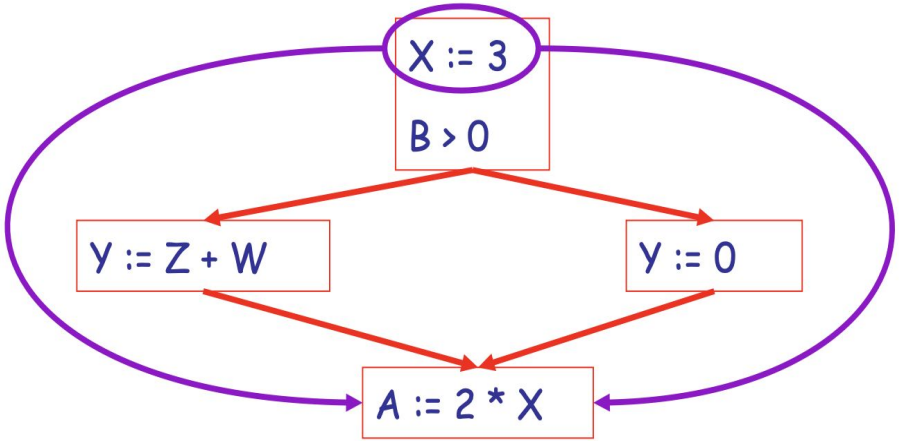


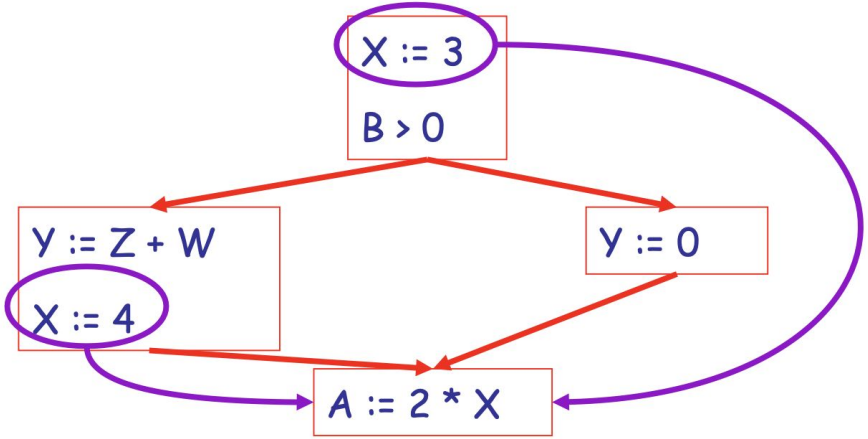
“Global” Constant Folding: Examples



ok to constant fold:
all paths have X := 3

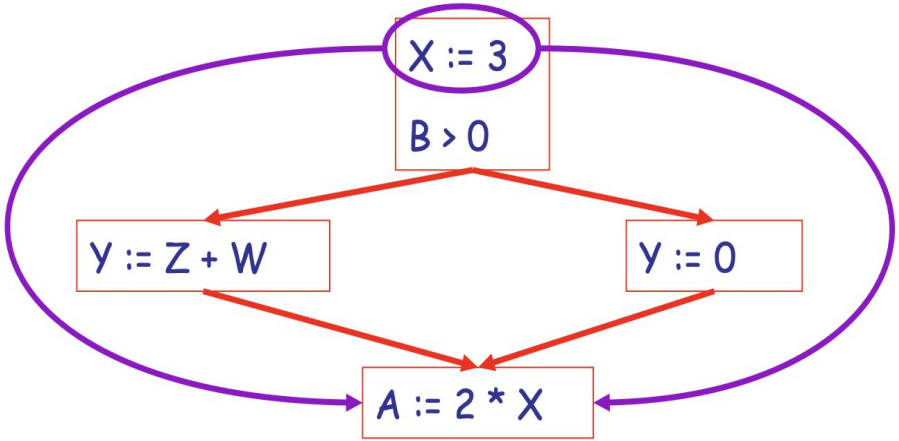
“Global” Constant Folding: Examples



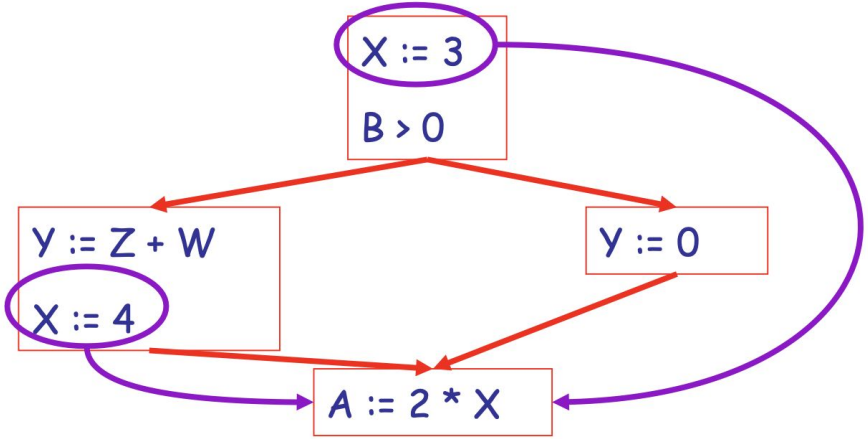


ok to constant fold:
all paths have X := 3

“Global” Constant Folding: Examples



ok to constant fold:
all paths have X := 3



not ok to constant fold:
one path has X := 3, the other has X := 4

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check
- “**All paths**” includes paths around loops and through branches of conditionals

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check
- “**All paths**” includes paths around loops and through branches of conditionals
  - May have an arbitrary number of jumps out of the procedure, unlike regional optimization

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check
- “**All paths**” includes paths around loops and through branches of conditionals
  - May have an arbitrary number of jumps out of the procedure, unlike regional optimization
- Checking such a condition requires **global analysis**

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check
- “**All paths**” includes paths around loops and through branches of conditionals
  - May have an arbitrary number of jumps out of the procedure, unlike regional optimization
- Checking such a condition requires **global analysis**
  - “Global” = an analysis of the **entire control-flow graph** for one method body

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check
- “**All paths**” includes paths around loops and through branches of conditionals
  - May have an arbitrary number of jumps out of the procedure, unlike regional optimization
- Checking such a condition requires **global analysis**
  - “Global” = an analysis of the **entire control-flow graph** for one method body
- ***Dataflow analysis*** is a common global analysis

“Global” Optimizations: Discussion

- This correctness condition is **not trivial** to check
- “**All paths**” includes paths around loops and through branches of conditionals
  - May have an arbitrary number of jumps out of the procedure, unlike regional optimization
- Checking such a condition requires **global analysis**
  - “Global” = an analysis of the **entire control-flow graph** for one method body
- ***Dataflow analysis*** is a common global analysis
  - called “dataflow” analysis because it propagates information about how data moves through the control-flow graph

Common Traits of Dataflow Analyses

Common Traits of Dataflow Analyses

- Proving most global optimizations safe depends on knowing a property P at a particular point in program execution
  - i.e., **for all** executions, is P true at this point?

Common Traits of Dataflow Analyses

- Proving most global optimizations safe depends on knowing a property P at a particular point in program execution
  - i.e., **for all** executions, is P true at this point?
- Proving P at any specific program point typically requires knowledge of the entire method body

Common Traits of Dataflow Analyses

- Proving most global optimizations safe depends on knowing a property P at a particular point in program execution
  - i.e., **for all** executions, is P true at this point?
- Proving P at any specific program point typically requires knowledge of the entire method body
- Property P is typically **undecidable**
  - Why?

Common Traits of Dataflow Analyses

- Proving most global optimizations safe depends on knowing a property P at a particular point in program execution
  - i.e., **for all** executions, is P true at this point?
- Proving P at any specific program point typically requires knowledge of the entire method body
- Property P is typically **undecidable**
  - Why?
    - Simple consequence of **Rice's Theorem**

Review: Undecidability of Program Properties

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:

> **"interesting"** in this context means "not trivial", i.e., not uniformly true or false for all programs

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.
    - Take function H and find out if it halts by testing function `F(x) = { H(x); return 1; }` to see if it has a positive result

Review: Undecidability of Program Properties

- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program halt on all (some) inputs?
    - This is called the **halting problem**
  - Is the result of a function F always positive?
    - *Assume* we can answer this question precisely
    - Oops: We can now solve the halting problem.
    - Take function H and find out if it halts by testing function `F(x) = { H(x); return 1; }` to see if it has a positive result
    - Contradiction!

Review: Undecidability of Program Properties

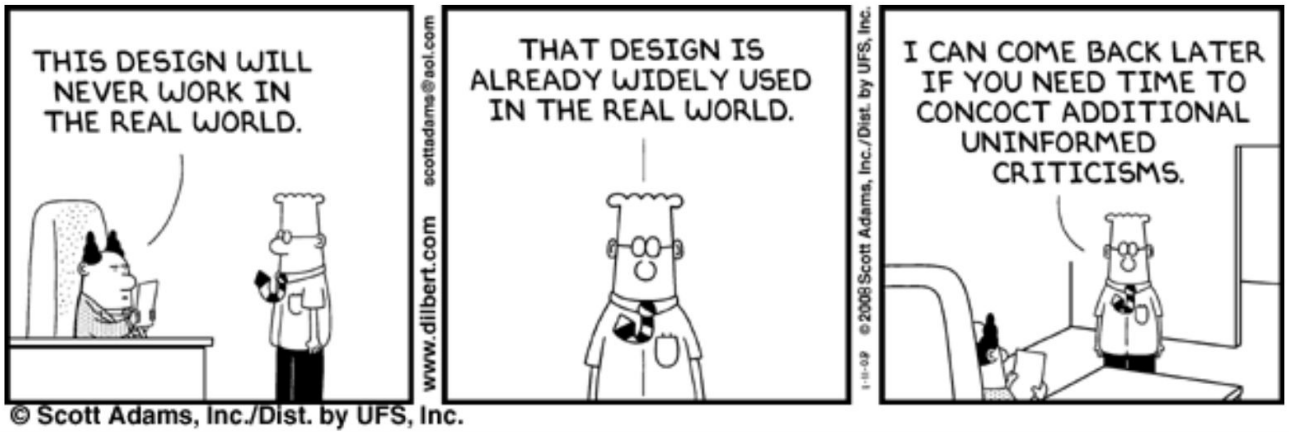
- ***Rice's Theorem***: All interesting dynamic properties of a program are undecidable:
  - Does the program hal
    - This is called the h
  - Is the result of a funct
    - *Assume* we can ans
    - Oops: We can now
    - Take function H an
      `F(x) = { H(x); return 1; }` to see if it has a positive result
    - Contradiction!

Rice's theorem caveats:

- only applies to **semantic** properties (syntactic properties are decidable)
- "programs" only includes programs **with loops**

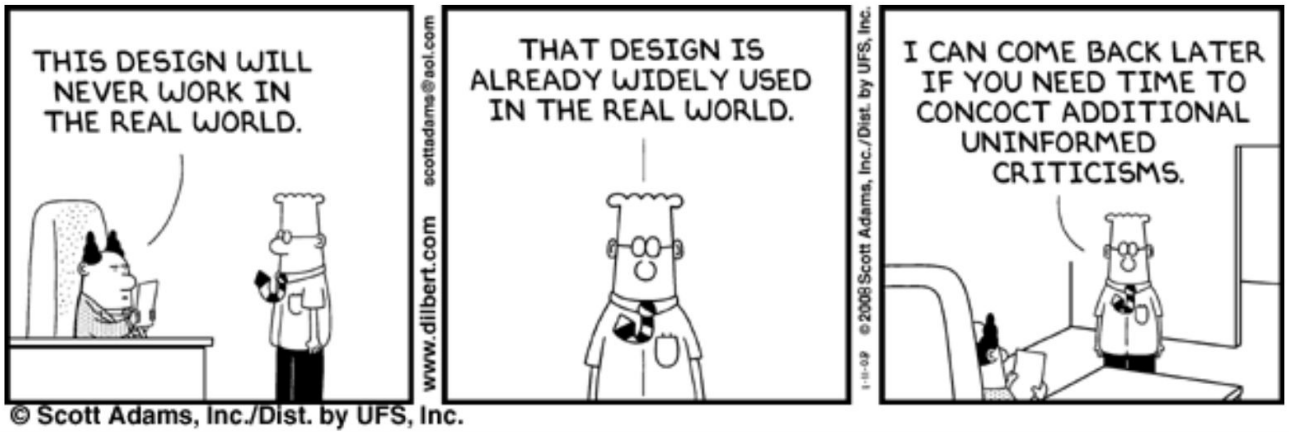
Back To Optimization

- Recall that to replace a use of x by a constant k we must know that:
  On every path to the use of x, the last assignment to x is x := k



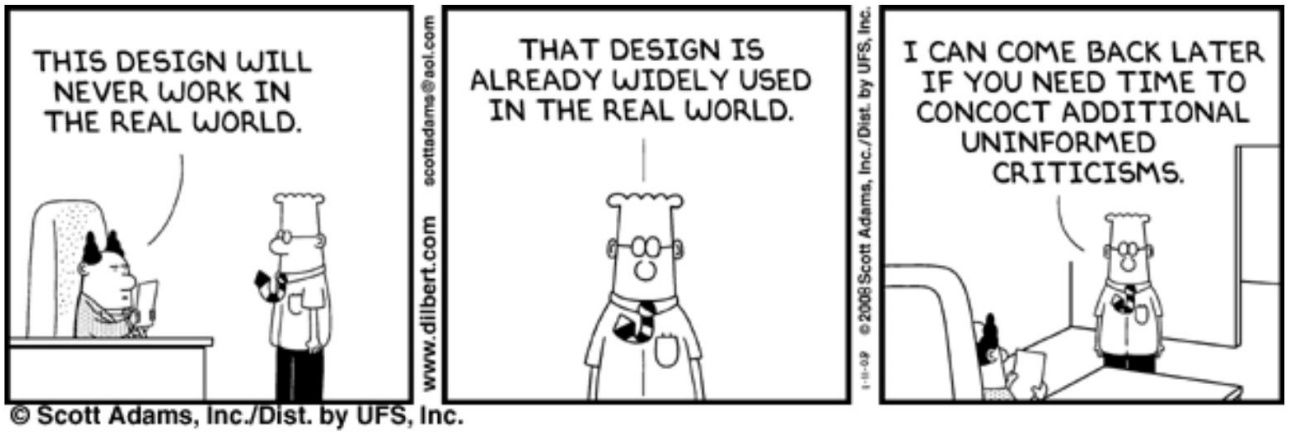
Back To Optimization

- Recall that to replace a use of x by a constant k we must know that:
  On every path to the use of x, the last assignment to x is x := k
- This correctness condition is **hard** to check (undecidable)



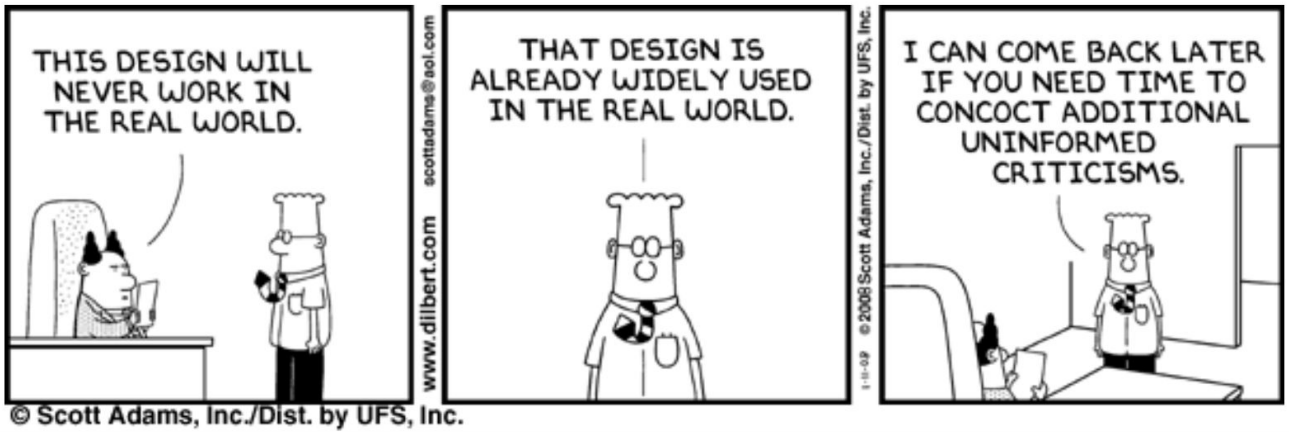
Back To Optimization

- Recall that to replace a use of x by a constant k we must know that:
  On every path to the use of x, the last assignment to x is x := k
- This correctness condition is **hard** to check (undecidable)
- Checking it requires an analysis of a whole method body



Back To Optimization

- Recall that to replace a use of x by a constant k we must know that:
  On every path to the use of x, the last assignment to x is x := k
- This correctness condition is **hard** to check (undecidable)
- Checking it requires an analysis of a whole method body
- We said that was **impossible**, right?



Conservative Program Analysis

- Because our analysis must run on a computer, we need the analysis itself to be **decidable**

Conservative Program Analysis

- Because our analysis must run on a computer, we need the analysis itself to be **decidable**
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(

Conservative Program Analysis

- Because our analysis must run on a computer, we need the analysis itself to be **decidable**
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(
- **Solution**: when in doubt, allow the analysis to answer "I don't know"

Conservative Program Analysis

- Because our analysis must run on a computer, we need the analysis itself to be **decidable**
- But, because of Rice's Theorem, we know that finding the right answer all the time is undecidable :(
- **Solution**: when in doubt, allow the analysis to answer "I don't know"
    - this is called ***conservative*** analysis

Conservative Program Analysis

- It's **always correct** to say "I don't know"

Conservative Program Analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible

Conservative Program Analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible
- A ***sound*** program analysis has no false negatives

Conservative Program Analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible
- A ***sound*** program analysis has no false negatives
  - always answers "I don't know" if the property of interest ***might*** **not be true**

Conservative Program Analysis

- It's **always correct** to say "I don't know"
  - **key challenge** in program analysis: say "I don't know" as rarely as possible
- A ***sound*** program analysis has no false negatives
  - always answers "I don't know" if the property of interest ***might*** **not be true**
- Our dataflow analyses for enabling optimizations will be sound and conservative
  - i.e., we will **not optimize** when they say "I don't know"

Example Analysis: Global Constant Folding

- **Global constant folding** is one example of an optimization that requires global dataflow analysis

Example Analysis: Global Constant Folding

- **Global constant folding** is one example of an optimization that requires global dataflow analysis
- Global constant folding can be performed at any point where the **correctness condition** holds

Example Analysis: Global Constant Folding

- **Global constant folding** is one example of an optimization that requires global dataflow analysis
- Global constant folding can be performed at any point where the **correctness condition** holds

Correctness condition for global constant folding:
On every path to the use of x, the last assignment to x is x := k

Example Analysis: Global Constant Folding

- **Global constant folding** is one example of an optimization that requires global dataflow analysis
- Global constant folding can be performed at any point where the **correctness condition** holds
- Let's consider the case of computing the correctness condition at all program points for a single variable **X**

Example Analysis: Global Constant Folding

- **Global constant folding** is one example of an optimization that requires global dataflow analysis
- Global constant folding can be performed at any point where the **correctness condition** holds
- Let's consider the case of computing the correctness condition at all program points for a single variable **X**
  - We can easily extend this to other variables later

Example Analysis: Global Constant Folding

- **Global constant folding** is one example of an optimization that requires global dataflow analysis
- Global constant folding can be performed at any point where the **correctness condition** holds
- Let's consider the case of computing the correctness condition at all program points for a single variable **X**
  - We can easily extend this to other variables later
  - Keep in mind what you know about **abstract interpretation** as we go through this, and look for the similarities
    - there are many!

Global Constant Folding: Abstraction

- To make the problem precise, we associate one of the following **abstract values** with X at every program point:

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”

(Look familiar?)

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”

(Look familiar?)

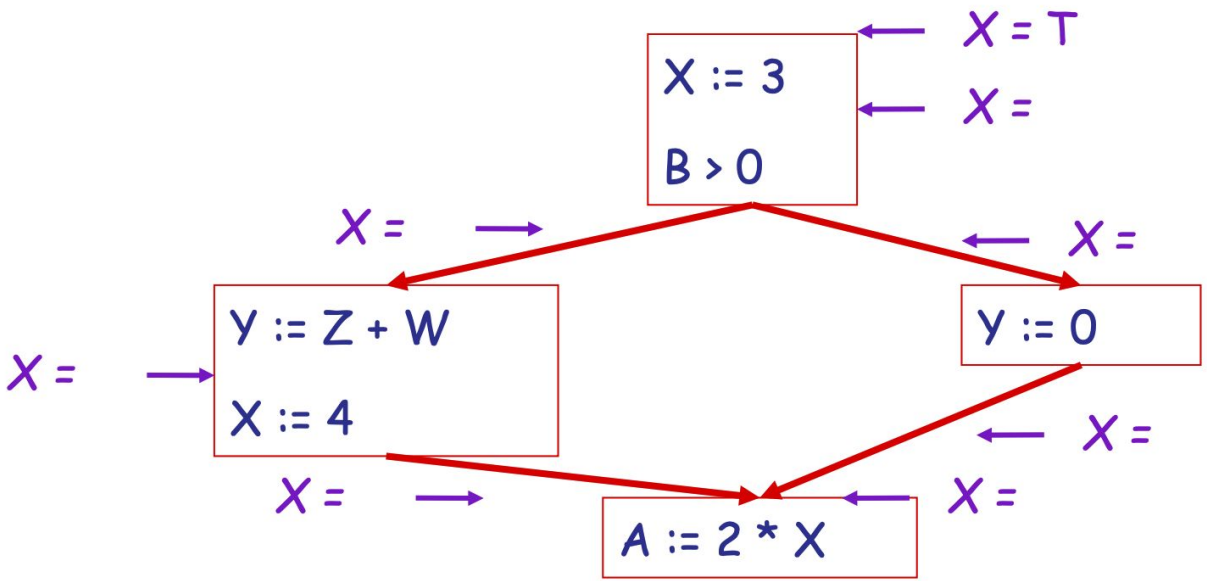
Global Constant Folding: Abstraction

- 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”

(Look familiar?)

Global Constant Folding: Formalized

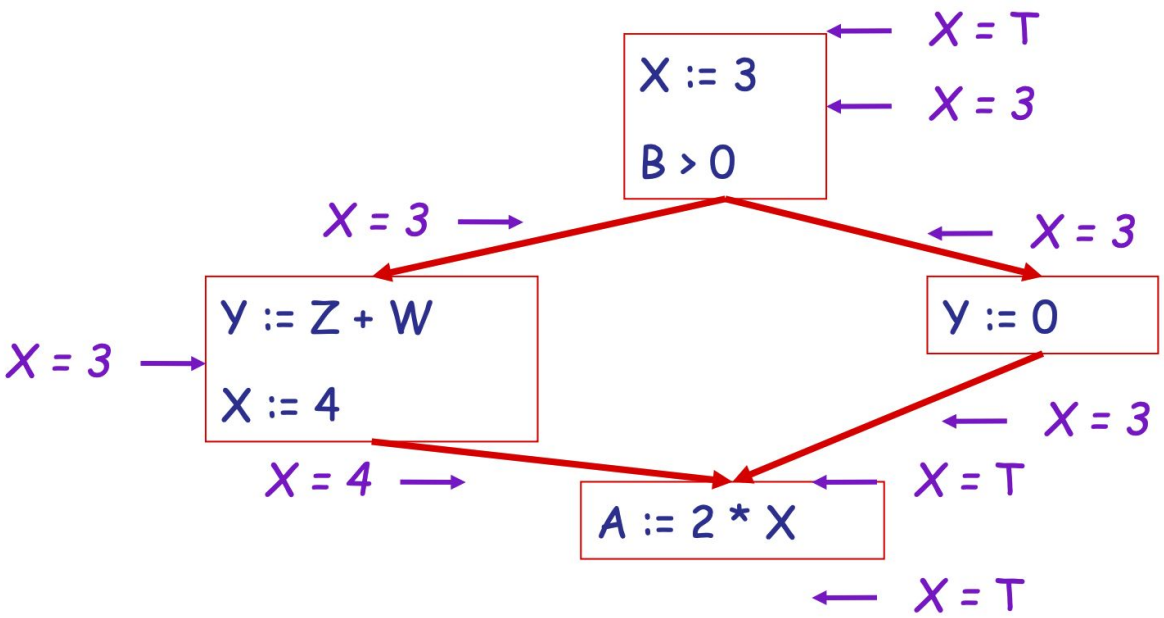
Get out a piece of paper. Fill in these blanks:



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

Using the Information

Using the Information

- Given global constant information, it is easy to decide whether or not to perform the optimization

Using the Information

- Given global constant information, it is easy to decide whether or not to perform the optimization
  - Simply inspect the *x* = *?* associated with a statement using *x*

Using the Information

- Given global constant information, it is easy to decide whether or not to perform the optimization
  - Simply inspect the *x* = *?* associated with a statement using *x*
  - If x is a constant at that point, replace that use of x by the constant!

Using the Information

- Given global constant information, it is easy to decide whether or not to perform the optimization
  - Simply inspect the *x* = *?* associated with a statement using *x*
  - If x is a constant at that point, replace that use of x by the constant!

- But how can an **algorithm** compute *x* = *?*

Key Idea

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

Explanation:

Key Idea

Explanation:

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

Key Idea

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

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

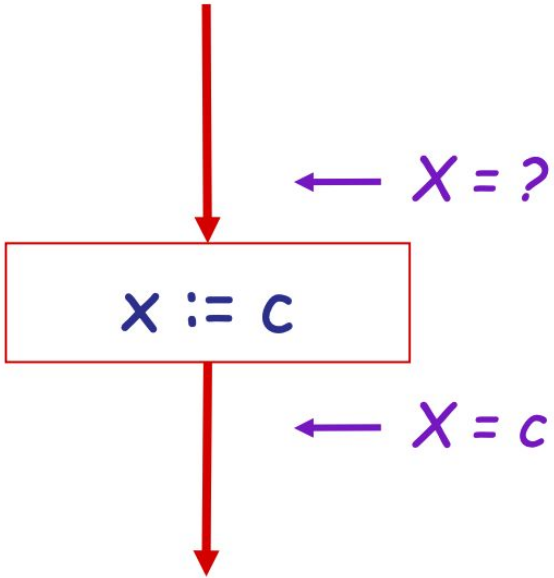
Key Idea

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

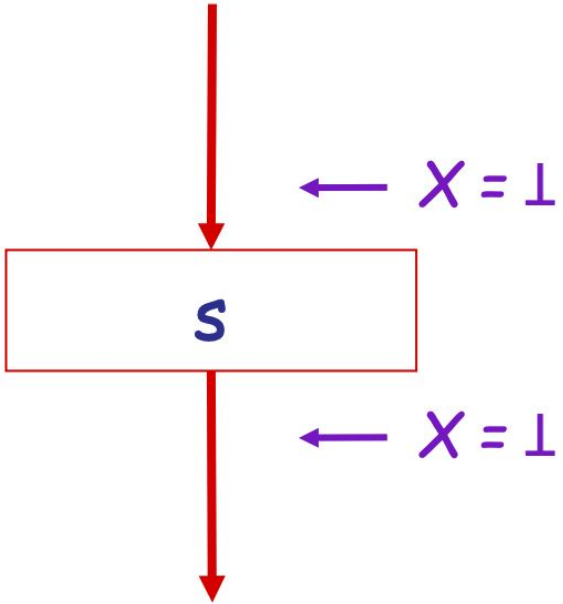


$C_{out}(x, x := c) = c$ if c is a constant

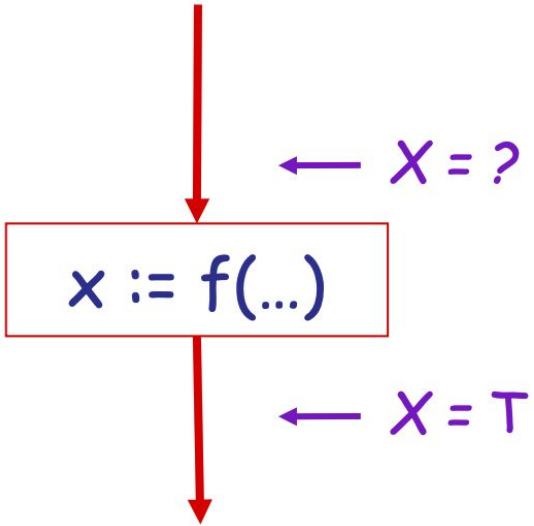
Transfer functions: rule 2

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

Recall bottom = “unreachable code”

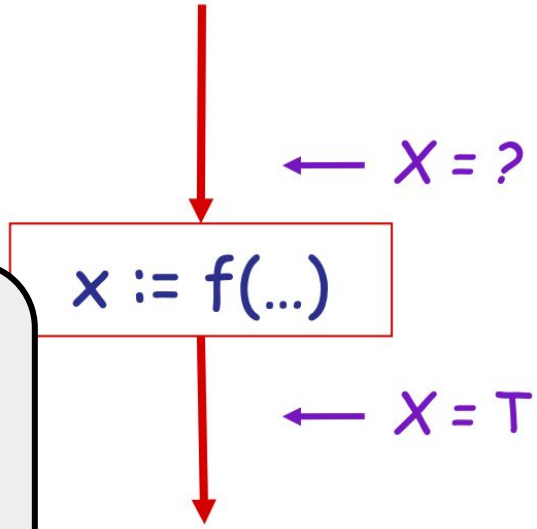


Transfer functions: rule 3



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

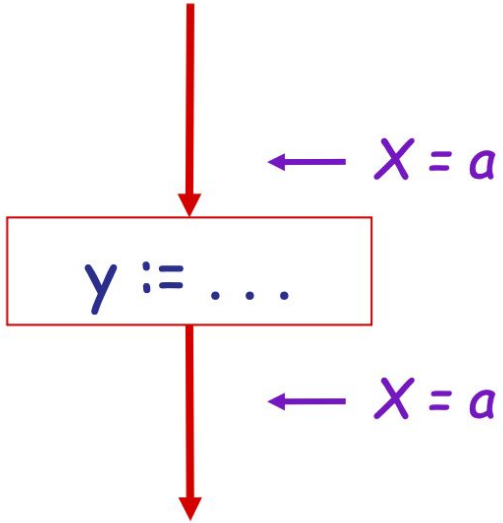
Transfer functions: rule 3



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

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

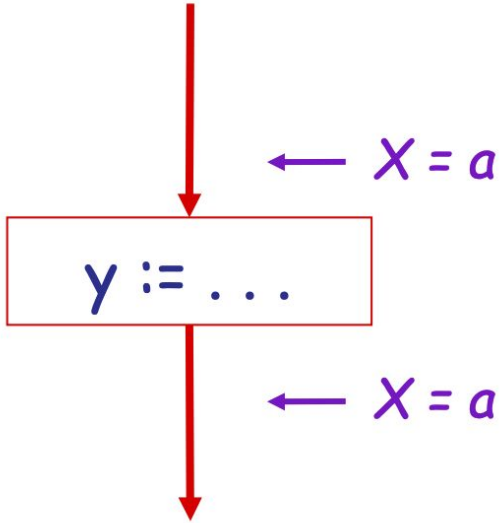
Transfer functions: rule 4



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

Transfer functions: rule 4

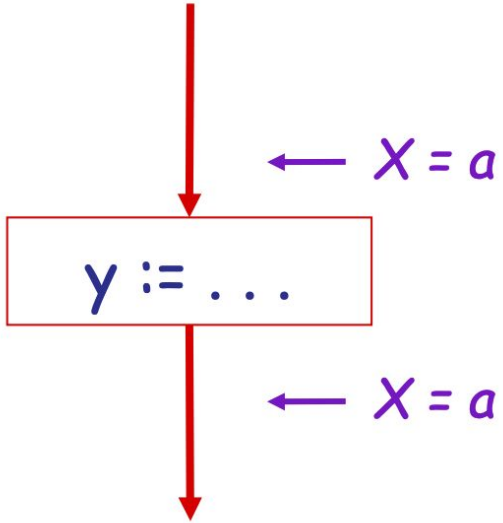
How hard is it to check if x ≠ y on all executions?



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

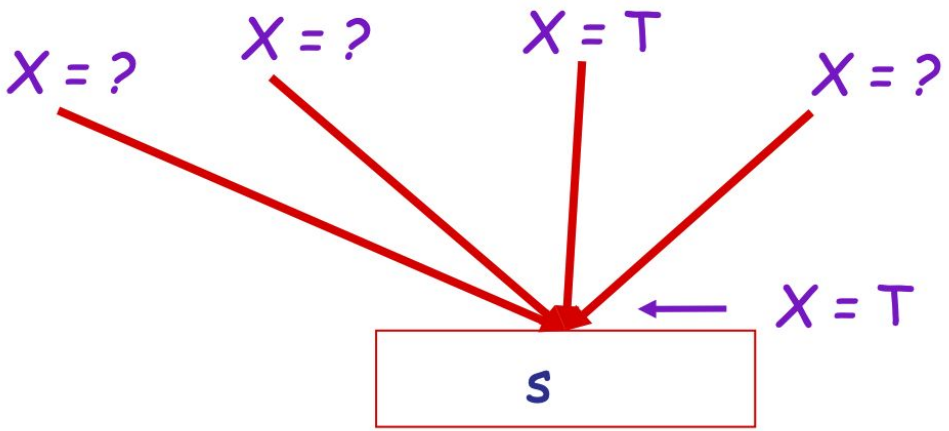
Propagation between Statements

- 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

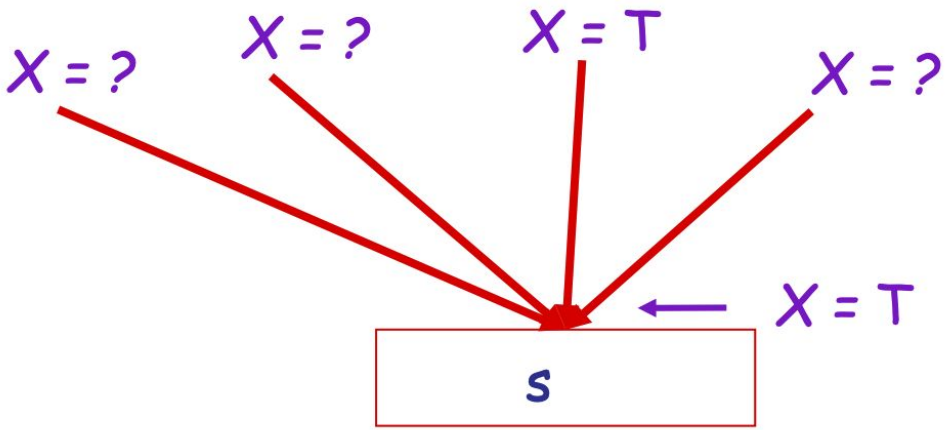
Transfer functions: rule 5



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

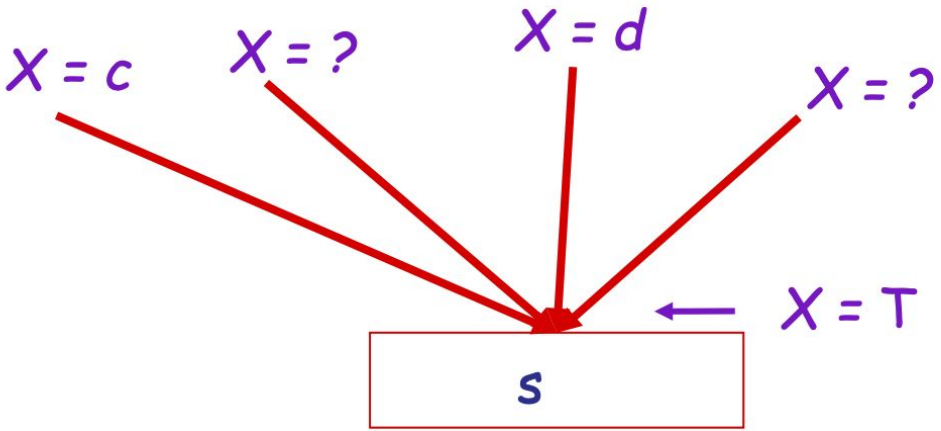
Transfer functions: rule 5

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



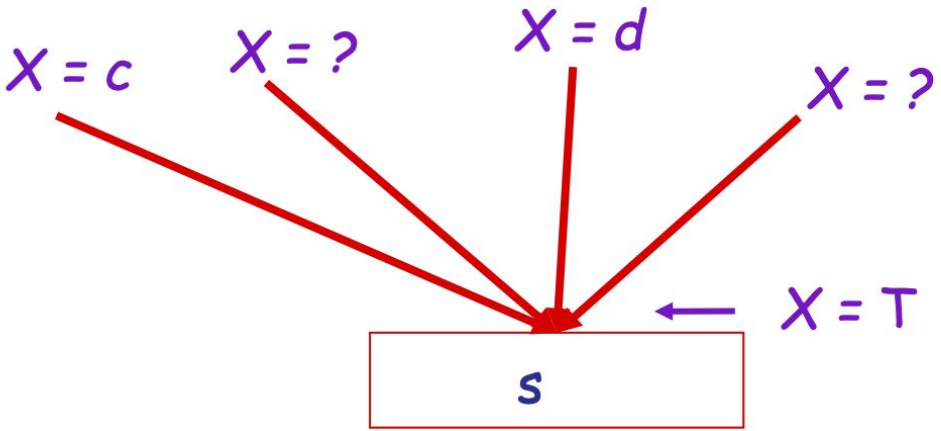
if $C_{out}(x, p_i) = T$ for some i, then $C_{in}(x, s) = T$

Transfer functions: rule 6



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

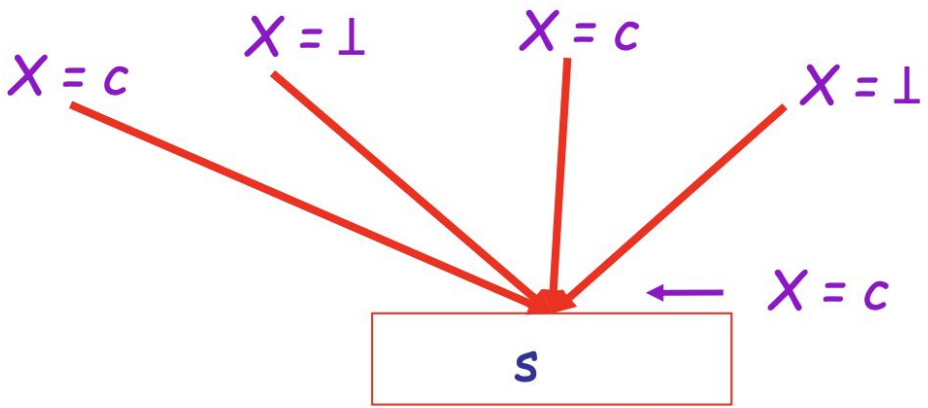
Transfer functions: rule 6



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 $C_{in}(x, s) = T$

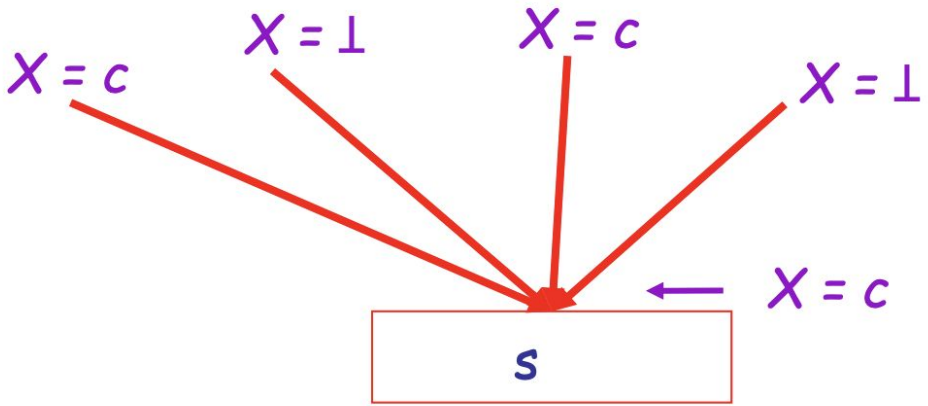
Transfer functions: rule 7



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

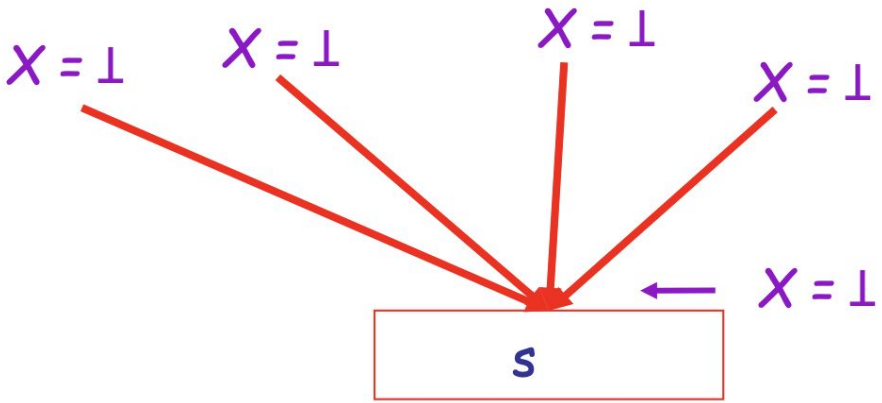
Transfer functions: rule 7

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

The Dataflow Analysis Algorithm

The Dataflow Analysis Algorithm

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

A static 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 static 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 static 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 static analysis alg

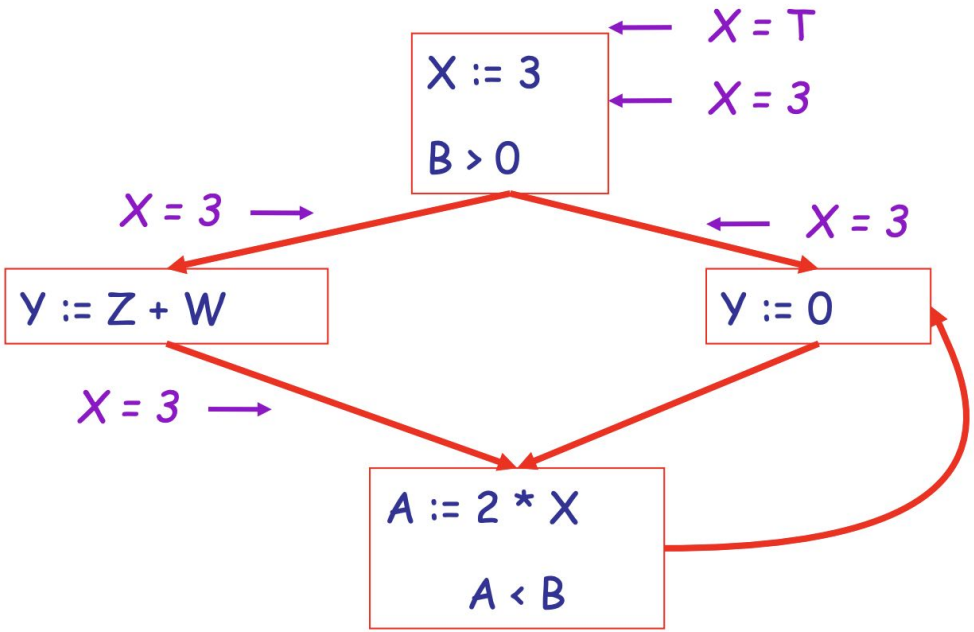
- For every **entry point** *e* to
  - why top? Top models
    inputs to the procedu
- Set $C_{in}(x, s) = C_{out}(x, s) =$ bo
- **Repeat** until all points satisfy rules 1-8:
  - Pick s not satisfying rules 1-8 and update using the appropriate rule

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?

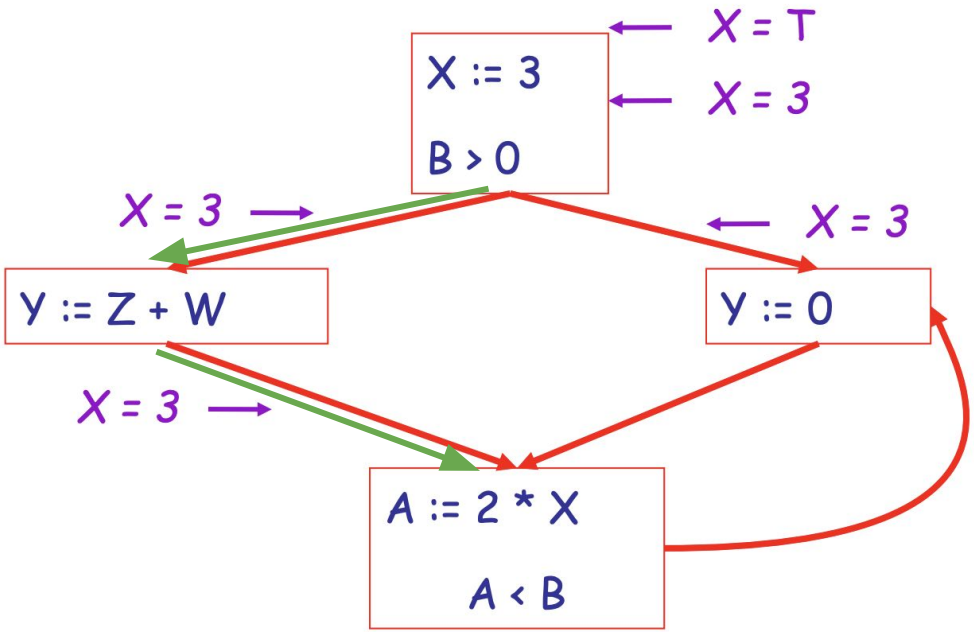
Why do we need bottom?

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



Why do we need bottom?

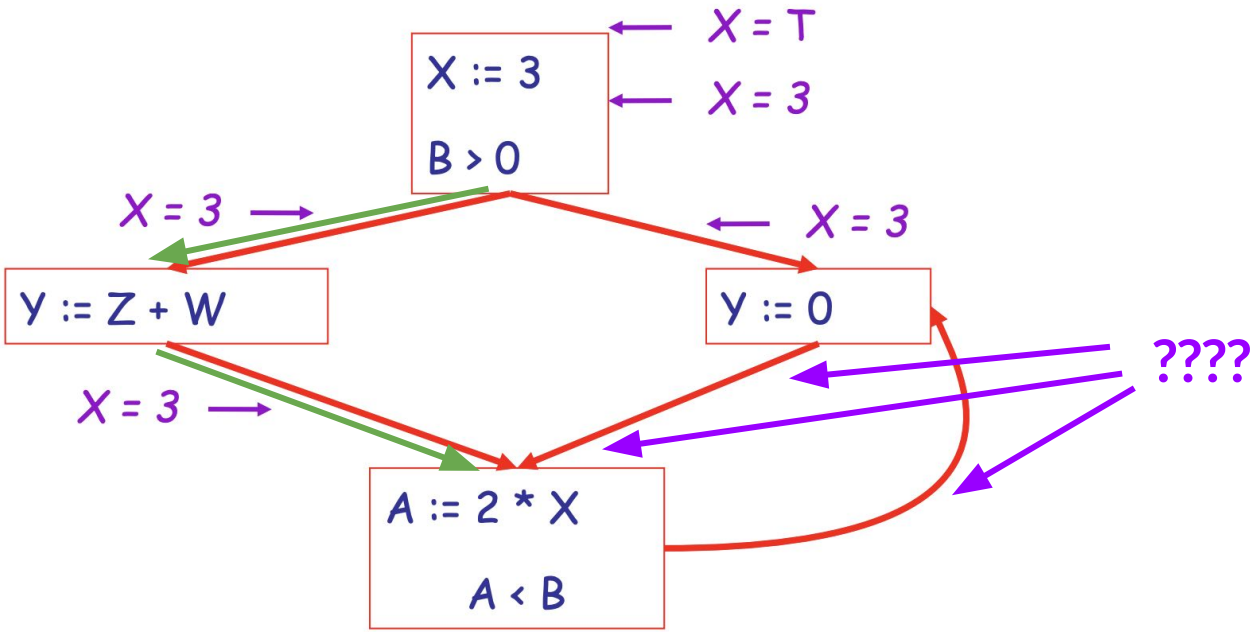
- To understand why we need to set non-entry points to bottom initially, consider a program with a loop:



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

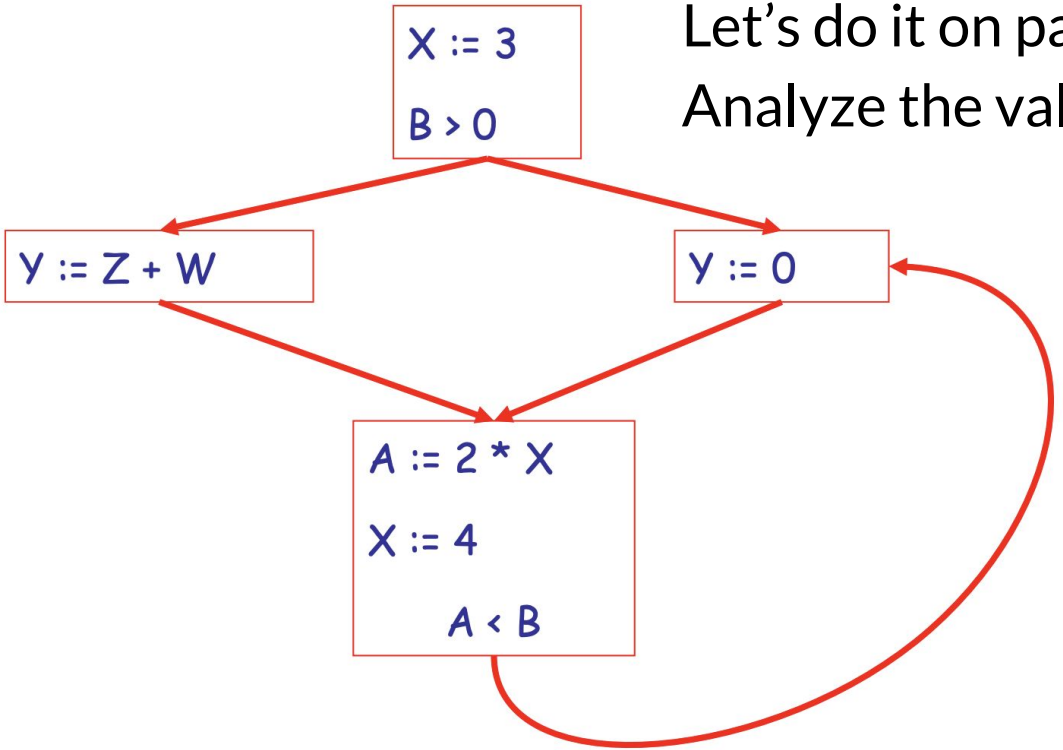
Why do we need bottom?

- 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?

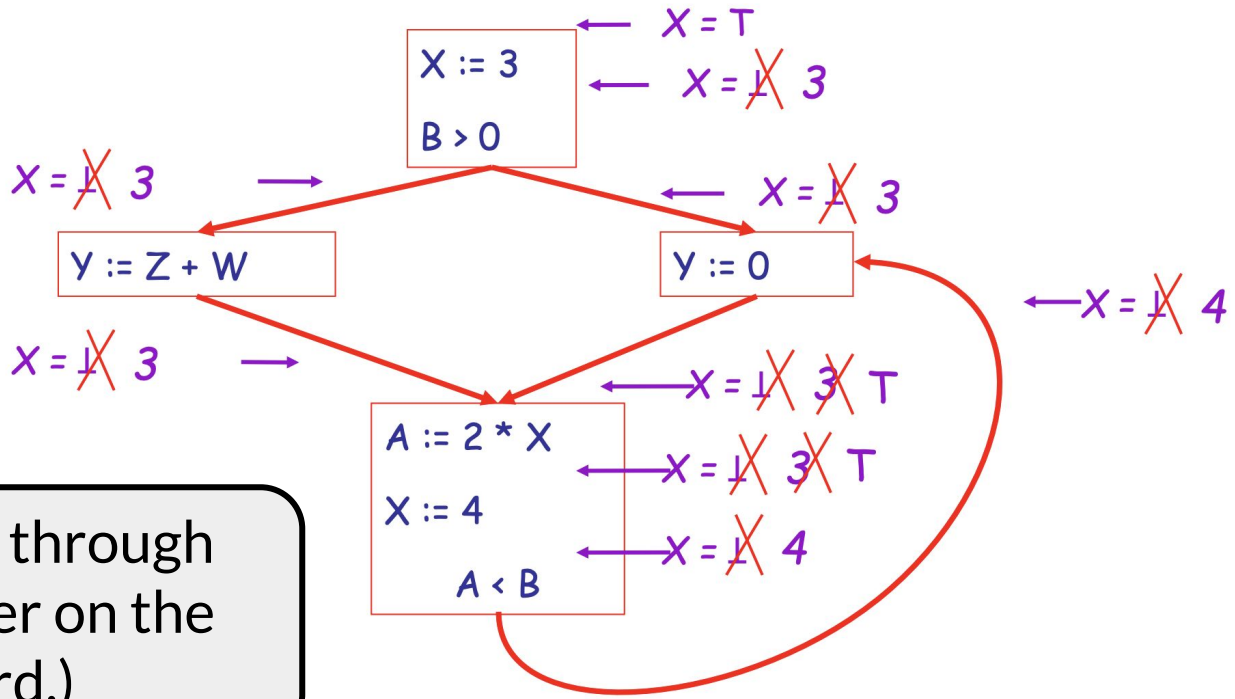
- To understand why we need to set non-entry points to bottom initially, consider a program with a loop.
- Because of **cycles**, all points must have values at all times during the analysis
- Intuitively, assigning some initial value allows the analysis to **break cycles**
- The initial value bottom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

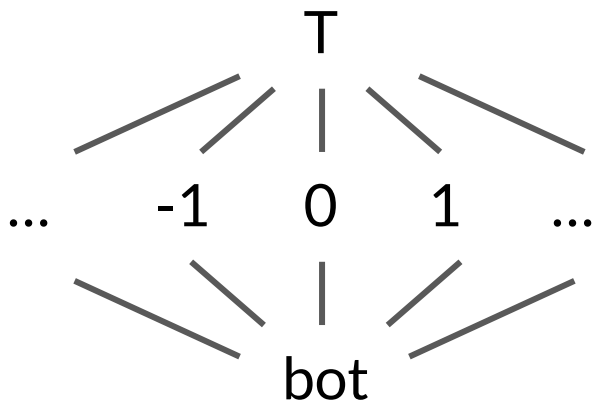
- You may have observed that there is a natural ***order*** to the different abstract values in our dataflow analysis
  - (Most) locations start as bottom
  - Locations whose current value is bottom might become *c* or T
  - Locations whose current value is *c* might become T
    - but never go back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

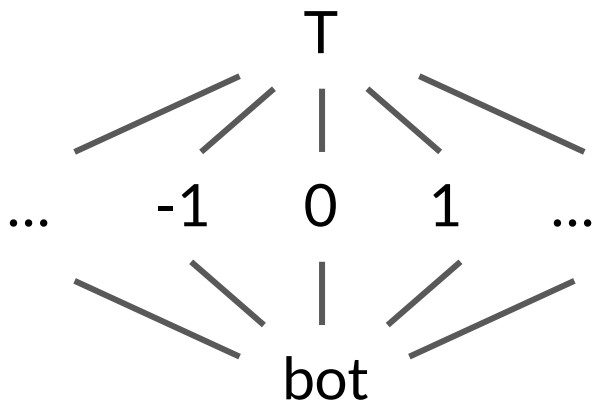
Lattices & Orderings

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 nullness 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)

Next Time

- More dataflow analysis
  - “liveness analysis” for dead code elimination
- Inter-procedural analyses
  - Inlining, tail call optimization, and more

Course Announcements

- As many of you have noticed, there was an issue with the PA3 autograder that prevented anyone from passing tests 16+
  - This issue has been fixed as of this morning
  - Discuss with class: do y'all need another extension on PA3? I am open to giving one.
    - First rule of compilers: semantics >>> optimization
- I will be out of town on Wednesday 4/30 (for ICSE)
  - I've therefore rearranged the calendar a bit
  - My PhD student Erfan Arvan will give a lecture on exceptions