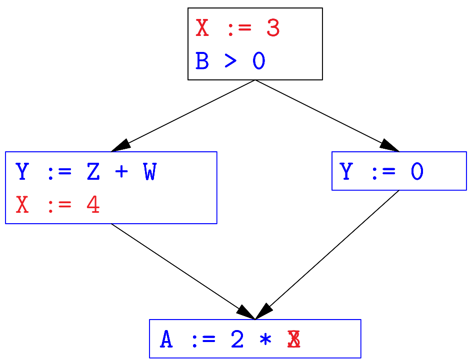
Lecture 37: Global Optimization

[Adapted from notes by R. Bodik and G. Necula]

Topics

- *Global optimization* refers to program optimizations that encompass multiple basic blocks in a function.
- (I have used the term *galactic optimization* to refer to going beyond function boundaries, but it hasn't caught on; we call it just *interprocedural optimization*.)
- Since we can't use the usual assumptions about basic blocks, global optimization requires *global flow analysis* to see where values can come from and get used.
- The overall question is: When can local optimizations (from the last lecture) be applied across multiple basic blocks?

A Simple Example: Copy Propagation



- Without other assignments to X, it is valid to treat the red parts as if they were in the same basic block.
- But as soon as *one* other block on the path to the bottom block assigns to X, we can no longer do so.
- It is correct to apply copy propagation to a variable x from an assignment statement A: x := ... to a given use of x in statement B only if the last assignment to x in every path from to B is A.

Issues

- This correctness condition is not trivial to check
- "All paths" includes paths around loops and through branches of conditionals
- Checking the condition requires global analysis: an analysis of the entire control-flow graph for one method body.
- This is typical for optimizations that depend on some property P at a particular point in program execution.
- Indeed, property P is typically undecidable, so program optimization is all about making *conservative* (but not cowardly) approximations of P.

Undecidability of Program Properties

- Rice's "theorem:" Most interesting dynamic properties of a program are undecidable. E.g.,
  - Does the program halt on all (some) inputs? (Halting Problem)
  - Is the result of a function F always positive? (Consider

```
def F(x):
    H(x)
    return 1
```

  Result is positive iff H halts.)
- Syntactic properties are typically decidable (e.g., "How many occurrences of x are there?").
- Theorem does not apply in absence of loops

Conservative Program Analyses

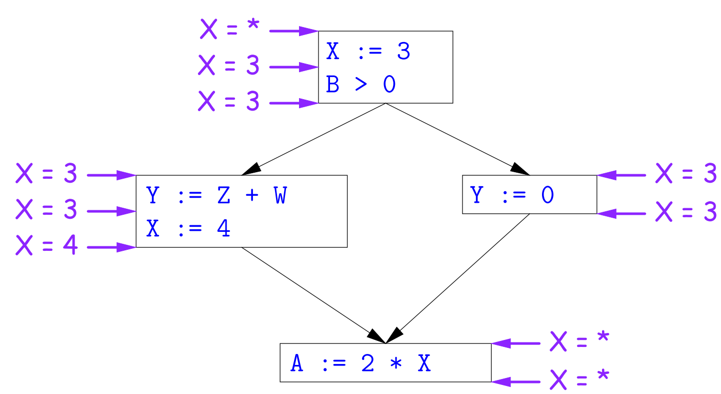
- If a certain optimization requires P to be true, then
  - If we know that P is definitely true, we can apply the optimization
  - If we don't know whether P is true, we simply don't do the optimization. Since optimizations are not supposed to change the meaning of a program, this is safe.
- In other words, in analyzing a program for properties like P, it is *always correct* (albeit non-optimal) to say "don't know."
- The trick is to say it as seldom as possible.
- *Global dataflow analysis* is a standard technique for solving problems with these characteristics.

Example: Global Constant Propagation

- *Global constant propagation* is just the restriction of copy propagation to constants.
- In this example, we'll consider doing it for a single variable (X).
- At every program point (i.e., before or after any instruction), we associate one of the following values with X

Value	Interpretation
#	(aka *bottom*) No value has reached here (yet)
c	(For c a constant) X definitely has the value c.
*	(aka *top*) Don't know what, if any, constant value X has.

Example of Result of Constant Propagation



Using Analysis Results

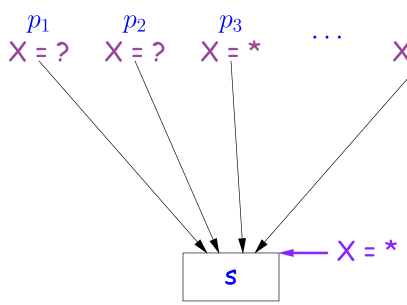
- Given global constant information, it is easy to perform the optimization:
  - If the point immediately before a statement using x tells us that x = c, then replace x with c.
  - Otherwise, leave it alone (the conservative option).
- But how do we compute these properties x = ...?

Transfer Functions

- **Basic Idea:** Express the analysis of a complicated program as a combination of simple rules relating the change in information between adjacent statements
- That is, we *"push"* or *transfer* information from one statement to the next.
- For each statement s, we end up with information about the value of x immediately before and after s:

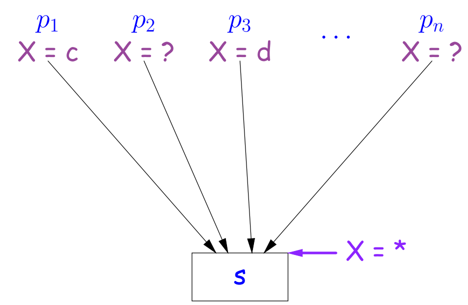
  Cin(X,s) = value of x before s
  Cout(X,s) = value of x after s
- Here, the "values of x" we use come from an *abstract domain*, containing the values we care about—#, $*$, k—values computed *statically* by our analysis.
- For the constant propagation problem, we'll compute Cout from Cin, and we'll get Cin from the Couts of predecessor statements, Cout(X, p_1),...,Cout(X,p_n).

Constant Propagation: Rule 1



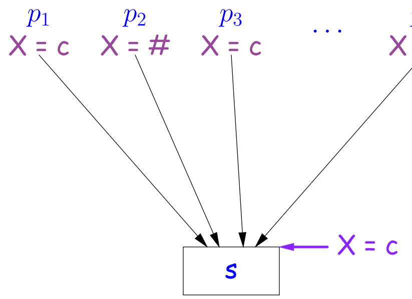
If Cout(X, p_i) = $*$ for some i, then Cin(X, s) = $*$

Constant Propagation: Rule 2



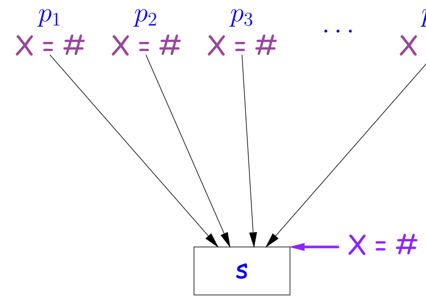
If Cout(X, p_i) = c and Cout(X, p_j) = d with constants $c \neq d$, then Cin(X, s) = $*$

Constant Propagation: Rule 3



If Cout(X, p_i) = c for some i and
Cout(X, p_j) = c or Cout(X, p_j) = # for all j,
then Cin(X, s) = c

Constant Propagation: Rule 4

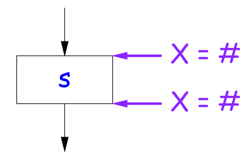


If Cout(X, p_j) = # for all j, then Cin(X, s) = #

Constant Propagation: Computing Cout

- Rules 1–4 relate the *out* of one statement to the *in* of the successor statements, thus propagating information *forward* across CFG edges.
- Now we need *local* rules relating the *in* and *out* of a single statement to propagate information across statements.

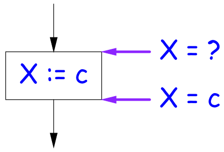
Constant Propagation: Rule 5



Cout(X, s) = # if Cin(X, s) = #

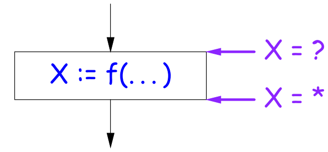
The value '#' means "so far, no value of X gets here, because the we don't (yet) know that this statement ever gets executed."

Constant Propagation: Rule 6



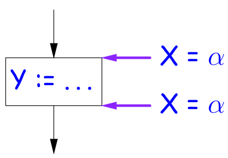
Cout(X, X := c) = c if c is a constant and ? is not #.

Constant Propagation: Rule 7



Cout(X, X := f(. . .)) = ∗ for any function call, if ? is not #.

Constant Propagation: Rule 8

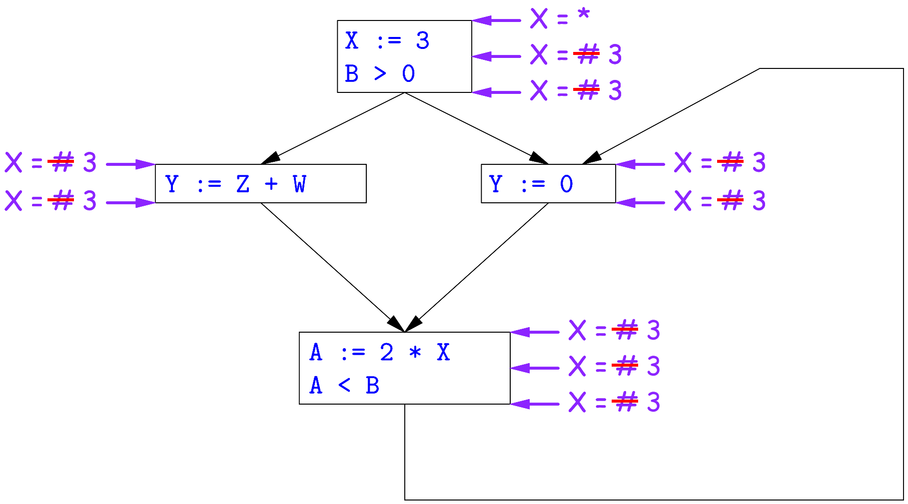


Cout(X, Y := . . .) = Cin(X, Y := . . .) if X and Y are different variables.

Propagation Algorithm

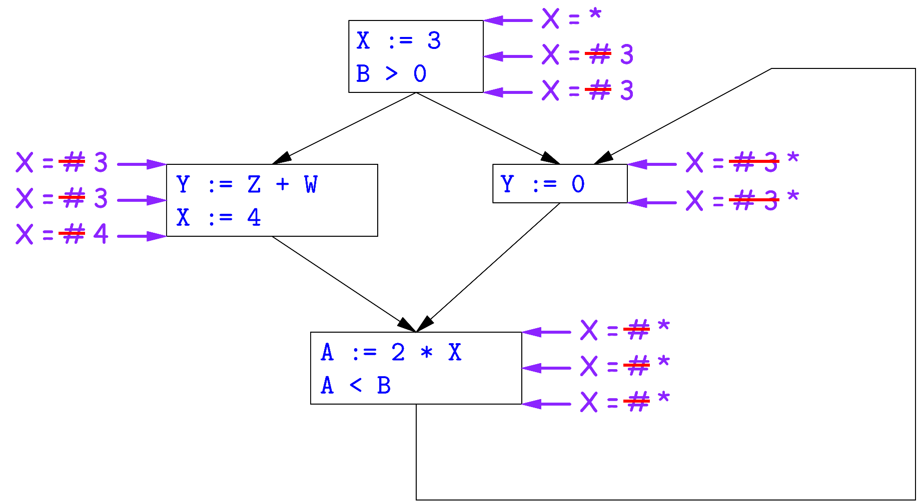
- To use these rules, we employ a standard technique: *iteration to a fixed point*:
- Mark all points in the program with *current approximations* of the variable(s) of interest (X in our examples).
- Set the initial approximations to X = * for the program entry point and X = # everywhere else.
- Repeatedly apply rules 1–8 every place they are applicable until nothing changes—until the program is at a *fixed point* with respect to all the transfer rules.
- We can be clever about this, keeping a list of all nodes any of whose predecessors' *Cout* values have changed since the last rule application.

An Example of the Algorithm



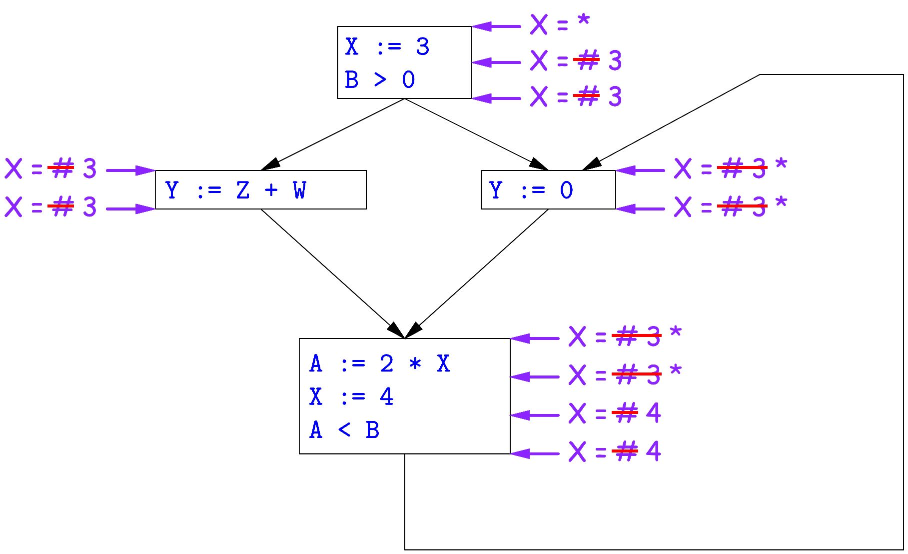
So we *can* replace X with 3 in the bottom block.

Another Example of the Propagation Algorithm



Here, we *cannot* replace X in two of the basic blocks.

A Third Example



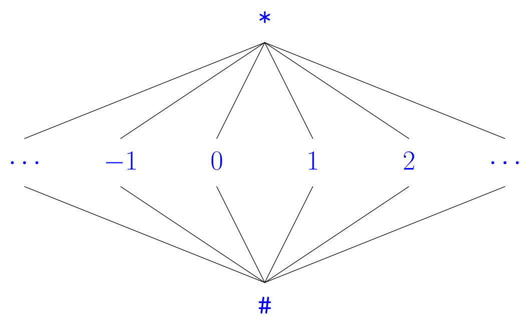
Likewise, we *cannot* replace X.

Comments

- The examples used a depth-first approach to considering possible places to apply the rules, starting from the entry point.
- In fact, the order in which one looks at statements is irrelevant. We could have changed the Cout values after the assignments to X first, for example.
- The # value is necessary to avoid deciding on a final value too soon. In effect, it allows us to tentatively propogate constant values through before finding out what happens in paths we haven't looked at yet.

Ordering the Abstract Domain

- We can simplify the presentation of the analysis by ordering the values $\# < c < *$.
- Or pictorially, with lower meaning less than,



- ...a mathematical structure known as a *lattice*.
- With this, our rule for computing Cin is simply a *least upper bound:*

  Cin(x, s) = lub { Cout(x, p) such that p is a predecessor of s }.

Termination

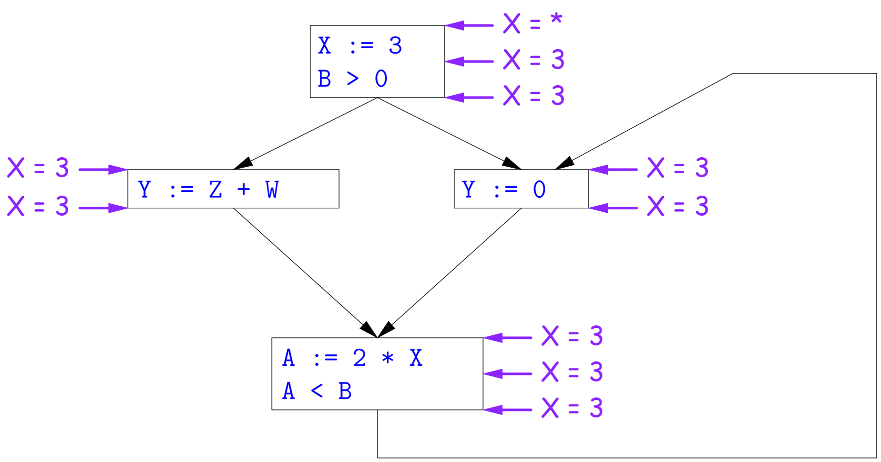
- Simply saying "repeat until nothing changes" doesn't guarantee that eventually nothing changes.
- But the use of lub explains why the algorithm terminates:
  - Values start as # and only increase
  - By the structure of the lattice, therefore, each value can only change twice.
- Thus the algorithm is linear in program size. The number of steps

  = $2\times$ Number of Cin and Cout values computed

  = $4\times$ Number of program statements.

Liveness Analysis

Once constants have been globally propagated, we would like to eliminate dead code



After constant propagation, `X := 3` is dead code (assuming this is the entire CFG)

Terminology: Live and Dead

- In the program

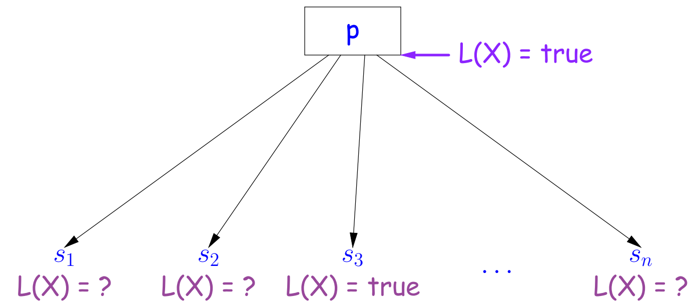
  ```
  X := 3;  /*(1)*/  X = 4;  /*(2)*/  Y := X  /*(3)*/
  ```

- the variable `X` is *dead* (never used) at point (1), *live* at point (2), and may or may not be live at point (3), depending on the rest of the program.
- More generally, 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'`; and
  - That path has no intervening assignment to `x`
- A statement `x := ...` is dead code (and may be deleted) if `x` is dead after the assignment.

Computing Liveness

- We can express liveness as a function of information transferred between adjacent statements, just as in copy propagation
- Liveness is simpler than constant propagation, since it is a boolean property (true or false).
- That is, the lattice has two values, with `false<true`.
- It also differs in that liveness depends on what comes *after* a statement, not before—we propagate information *backwards* through the flow graph, from Lout (liveness information at the end of a statement) to Lin.

Liveness Rule 1

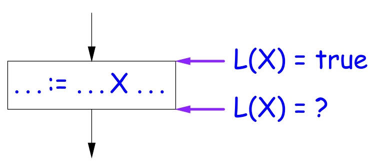


- So

  Lout(x, p) = lub { Lin(x, s) such that s is a predecessor of p }.

- Here, least upper bound (lub) is the same as "or".

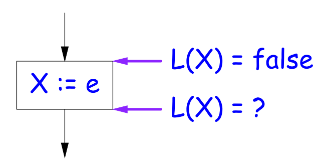
Liveness Rule 2



Lout(X, s) = true if `s` uses the previous value of `X`.

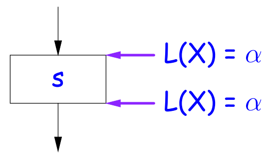
- The same rule applies to any other statement that uses the value of `X`, such as tests (e.g., `X < 0`).

Liveness Rule 3



Lout(X, X := e) = false if `e` does not use the previous value of `X`.

Liveness Rule 4

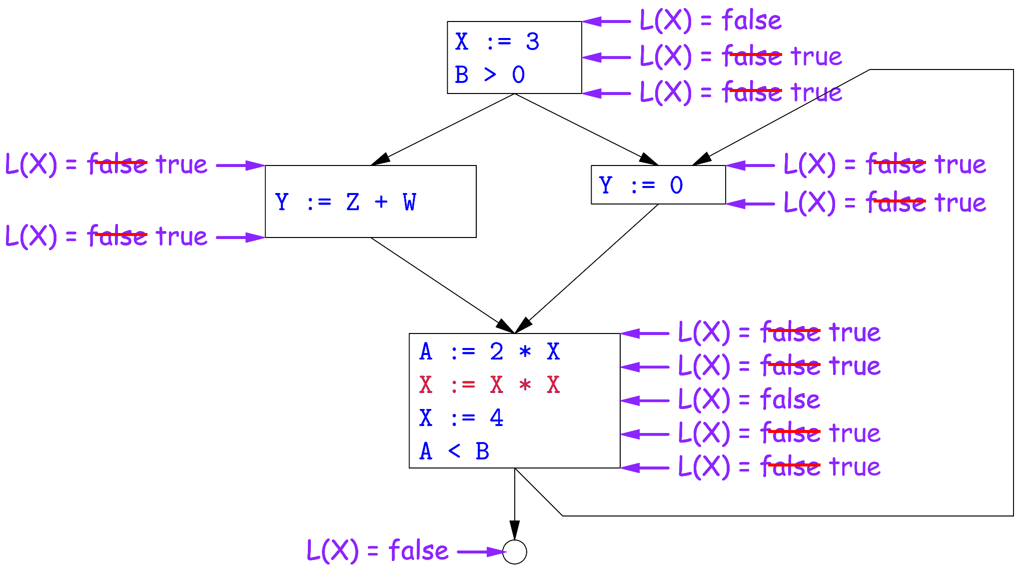


Lout(X, s) = Lin(X, s) if s does not mention X.

Propagation Algorithm for Liveness

- Initially, let all Lin and Lout values be false.
- Set Lout value at the program exit to true iff x is going to be used elsewhere (e.g., if it is global and we are analyzing only one procedure).
- As before, repeatedly pick s where one of 1–4 does not hold and update using the appropriate rule, until there are no more violations.
- When we're done, we can eliminate assignments to X if X is dead at the point after the assignment.

Example of Liveness Computation



Termination

- As before, a value can only change a bounded number of times: the bound being 1 in this case.
- Termination is guaranteed
- Once the analysis is computed, it is simple to eliminate dead code, but having done so, we must recompute the liveness information.

SSA and Global Analysis

- For local optimizations, the single static assignment (SSA) form was useful.
- But applying it to a full CFG is requires a trick.
- E.g., how do we avoid two assignments to the temporary holding x after this conditional?

```
if a > b:
    x = a
else:
    x = b
# where is x at this point?
```

- Answer: a small kludge known as ϕ "functions"
- Turn the previous example into this:

```
if a > b:
    x1 = a
else:
    x2 = b
x3 = φ(x1, x2)
```


ϕ Functions

- An artificial device to allow SSA notation in CFGs.
- In a basic block, each variable is associated with one definition,
- ϕ functions in effect associate each variable with a set of possible definitions.
- In general, one tries to introduce them in strategic places so as to minimize the total number of ϕs.
- Although this device increases number of assignments in IL, register allocation can remove many by assigning related IL registers to the same real register.
- Their use enables us to extend such optimizations as CSE elimination in basic blocks to *Global CSE Elimination.*
- With SSA form, easy to tell (conservatively) if two IL assignments compute the same value: just see if they have the same right-hand side. The same variables indicate the same values.

Summary

- We've seen two kinds of analysis:
  - Constant propagation is a *forward analysis:* information is pushed from inputs to outputs.
  - Liveness is a *backwards analysis*: information is pushed from outputs back towards inputs.
- But both make use of essentially the same algorithm.
- Numerous other analyses fall into these categories, and allow us to use a similar formulation:
  - An abstract domain (abstract relative to actual values);
  - Local rules relating information between consecutive program points around a single statement; and
  - Lattice operations like least upper bound (or *join*) or greatest lower bound (or *meet*) to relate inputs and outputs of adjoining statements.