Iterative Dataflow Analysis

- Many dataflow analyses have the same structure.
- They "interpret" the statements in program, collecting information as they proceed.
- Ideally, we would like to do "perfect interpretation," collecting exact information about how the program executes.
- We can't because we don't have the program inputs at compile time, and we aren't going to make our compiler run the program to completion during compilation–that just takes too long (why the hell do you think we are compiling the program in the first place?)

Iterative Dataflow Analysis

- Instead of interpreting the program exactly, we approximate the program behavior as we execute.
- This is called *Abstract Interpretation* and it is often a useful way of understanding different program analyses.
- Analysis via abstract interpretation is often defined by:
  - A *transfer function* — $f(n)$ — that simulates/approximates execution of instruction n on its inputs
  - A *joining operator* — $\vee$ — since you don't know which branch of an if statement (for example) will be taken at compile time, you need to interpret both and combine the results.
  - A *direction* — FORWARD or REVERSE — In the reverse direction, we are interpreting the program backwards.

Iterative Dataflow Analysis

To code up a particular analysis we need to take the following steps.

- First, we decide what sort of information we are interested in processing. This is going to determine the *transfer function* and the *joining operator*, as well as any *initial conditions* that need to be set up.
- Second, we decide on the appropriate direction for the analysis.
- In the forward direction, we:
  - Need to get the inputs from the previous instructions
  - Since we don't know exactly which instruction preceeded the current one, we use the join over all possible predecessors.
  - Once we have the input, we apply the transfer function, which generates an output.
  - Iterate the process.
  - Mathematically:

$$IN[n] = \vee_{p \in PRED[n]} OUT[p]$$
$$OUT[n] = f(n)$$

- In the backward direction, we:
  - Need get the outputs from the successor instructions.
  - Use the join since there are many successors.
  - Use the transfer function to get the inputs.
  - Iterate the process.
  - For reverse analyses:

$$OUT[n] = \vee_{s \in SUCC[n]} IN[s]$$

$$IN[n] = f(n)$$

Example: Live Variable Analysis

From last time, Live Variable Analysis:

- A register t is *live* on edge e if there exists a path from e to a use of t that does not go through a definition of t.
- $USE[n]$ - the set of registers that n uses.
- $DEF[n]$ - the set of registers that n defines.
- Transfer function $f(n) = USE[n] \cup (OUT[n] - DEF[n])$
- Join ($\vee$) is set union
- Direction: REVERSE

Live Variable Application 1: Register Allocation

Register Allocation:

1. Perform live variable analysis.
2. Build *interference graph*.
3. Color interference graph with real registers.

We'll be talking more about register allocation later, so I won't get into it now.

Live Variable Application 2: Dead Code Elimination

- Given statement s with a definition and no side-effects:

  `r1 = r2 + r3, r1 = M[r2], or r1 = r2`

  If r1 is *not* live at the end of s, then the s is *dead*

- Dead statements can be deleted.

- Some statements s do not define live variables, but cannot be consider dead because they have *side effects*.

  `r1 = call F, M[r1] = r2`

  Even if r1 is not live at the end of s, it is not dead.

Reaching Definition Analysis

Determines whether definition of register t directly affects use of t at some point in program.

Reaching Definition Definitions:

- Definition of d (of t) *reaches* statement u if a path of CFG edges exists from d to u that does not pass through a definition of t.

Analysis set-up:

- Label every temporary definition with a unique *definition id*
- Eg: $d : t = b + c$
- Eg: $d : t = M[b]$
- These instructions *generate* definition d
- These instructions also *kill* any previous definitions of register t.

Reaching Definition Analysis

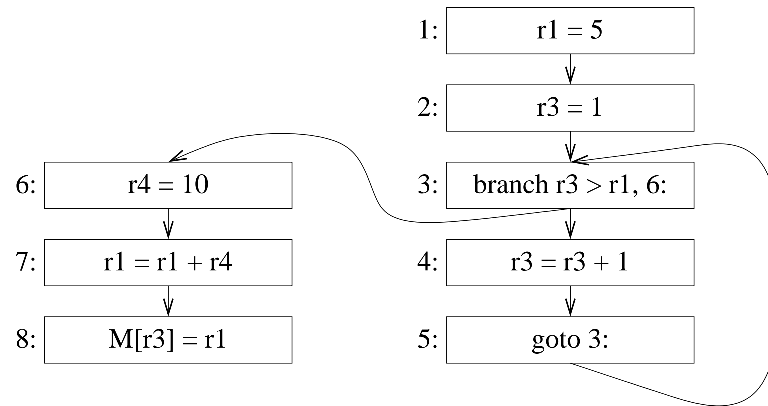
Reaching Definition Analysis Equation:

- $GEN[n]$ - the set of *definition id's* that n creates.
- $KILL[n]$ - the set of *definition id's* that n kills.
  - $defs(t)$ - set of all *definition id's* of register t.
- Transfer function $f(n) = GEN[n] \cup (IN[n] - KILL[n])$
- Join ($\vee$): Union
- Direction: FORWARD

$$IN[n] = \cup_{p \in PRED[n]} OUT[p]$$

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$$

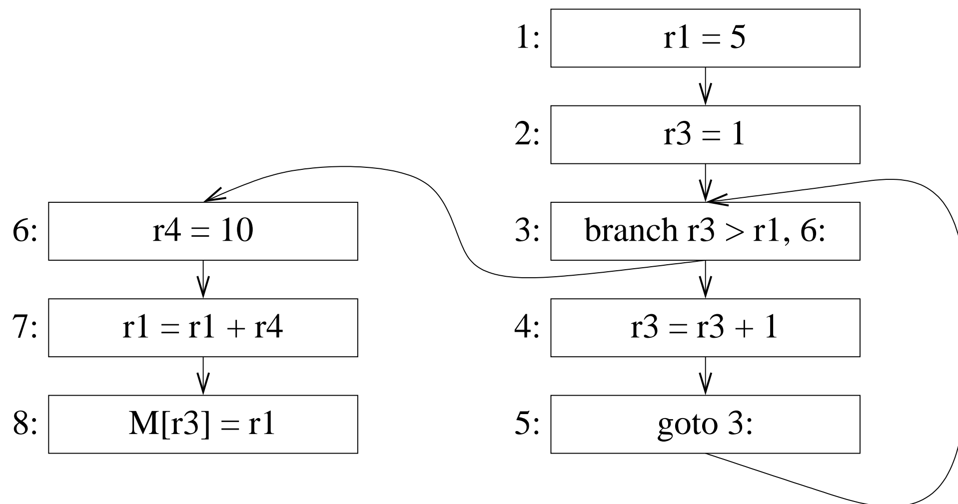
Reaching Definition Analysis Example



Node	GEN	$KILL$	IN	OUT	IN	OUT	IN	OUT
1								
2								
3								
4								
5								
6								
7								
8								

Reaching Definition Application 1: Constant Propagation

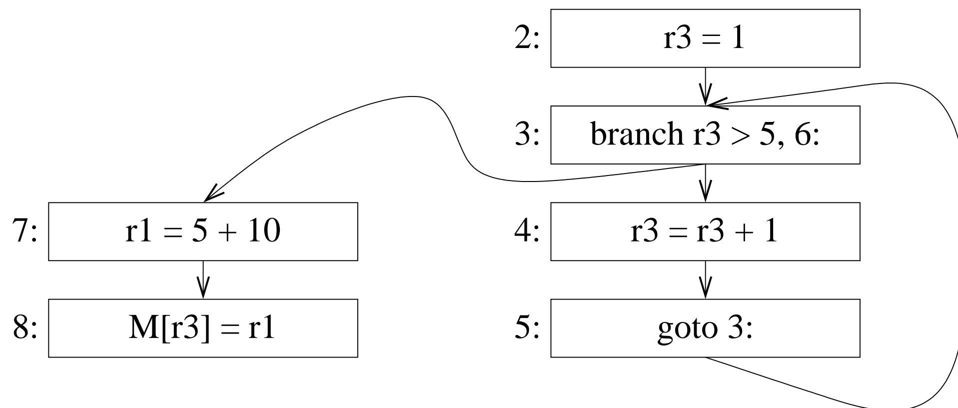
- Given Statement d: `a = c` where `c` is constant
- Given Statement u: `t = a op b`
- If statement d reach u and no other definition of `a` reaches u, then replace u by `t = c op b`.



Statements 1 and 6 are dead.

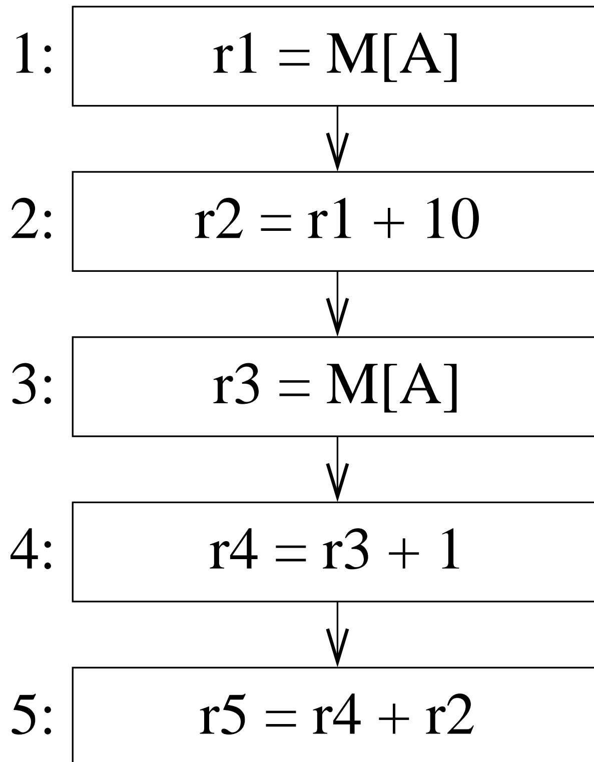
Constant Folding

- Given Statement d: `t = a op b`
- If `a` and `b` are constant, compute `c` as `a op b`, replace d by `t = c`



Common Subexpression Elimination

If `x op y` is computed multiple times, *common subexpression elimination* (CSE) attempts to eliminate some of the duplicate computations.



Need to track expression propagation → available expression analysis

Definitions

Definitions:

- Expression `x op y` is *available* at CFG node n if, on every path from CFG entry node to n, `x op y` is computed at least once, and neither `x` nor `y` are defined since last occurrence of `x op y` on path.
- Can compute set of expressions available at each statement using system of dataflow equations.
- Statement `r1 = M[r2]`:
  - *generates* expression `M[r2]`.
  - *kills* all expressions containing `r1`.
- Statement `r1 = r2 + r3`:
  - *generates* expression `r2 + r3`.
  - *kills* all expressions containing `r1`.

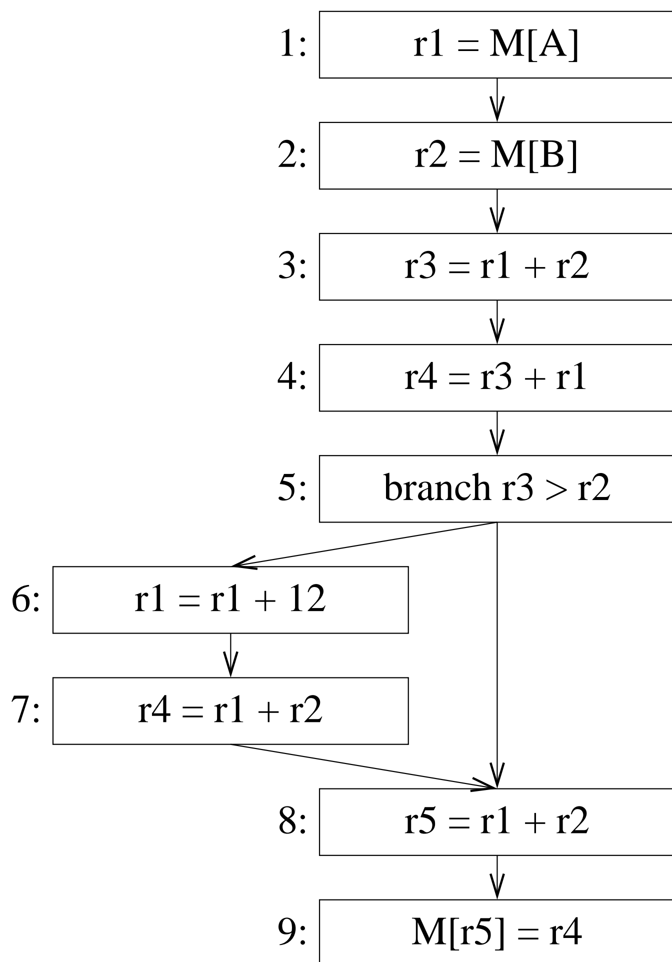
Available Expression Analysis

Available Expression Analysis:

- $exp(t)$ - set of all expressions containing t.
- $GEN[n]$ - the set of all expressions generated by n.
- $KILL[n]$ - the set of all expressions that n kills - $exp(n)$.
- Transfer function $f(n) = GEN[n] \cup (IN[n] - KILL[n])$
- Join operator ($\vee$): set intersection
  - Use of union as join, required initialization of IN and OUT sets to $\emptyset$.
  - Use of intersection, requires initialization of IN and OUT sets to U (except for IN of entry node).
- Direction: FORWARD

$$IN[n] = \cap_{p \in PRED[n]} OUT[p]$$

$$OUT[n] = GEN[n] \cup (IN[n] - KILL[n])$$



Node	GEN	$KILL$	IN	OUT
1	M[A]	r1+r2, r1+12, r3+r1	-	U
2	M[B]	r1+r2	U	U
3	r1+r2	r3+r1	U	U
4	r3+r1		U	U
5			U	U
6		r1+r2, r3+r1, r1+12	U	U
7	r1+r2		U	U
8	r1+r2	M[r5]	U	U
9		M[A], M[B], M[r5]	U	U

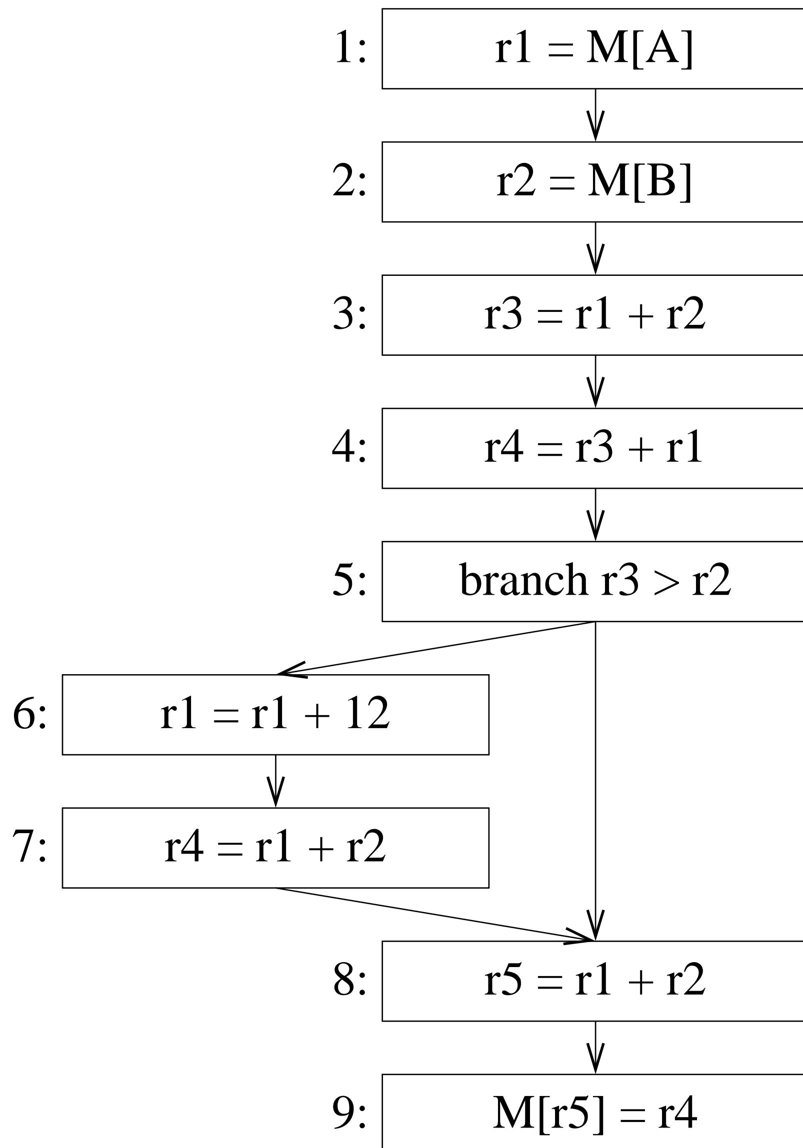
Node	GEN	$KILL$	IN	OUT
1	1	378, 6, 4	-	U
2	2	378	U	U
3	378	4	U	U
4	4		U	U
5			U	U
6		378, 4, 6	U	U
7	378		U	U
8	378	9	U	U
9		1, 2, 9	U	U

Common Subexpression Elimination

Given statement s: `t = x op y`:
If expression `x op y` is available at beginning of node s then:

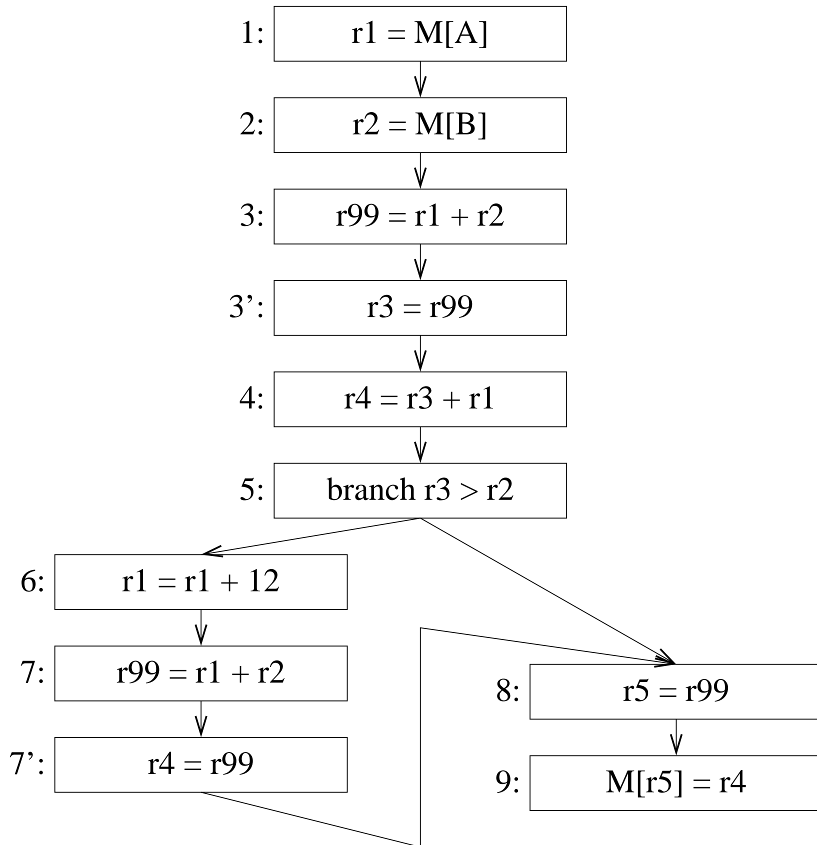
1. starting from node s, traverse CFG edges backwards to find last occurrence of `x op y` on each path from entry node to s.
2. create new temporary `w`.
3. for each statement s': `v = x op y` found in (1), replace s' by:
   ```
   w = x op y
   v = w
   ```
4. replace statement s by: `t = w`

Common Subexpression Elimination - Example



Copy Propagation

- Given statement d: `a = z` (`a` and `z` are both register temps) $\rightarrow$ d is a copy statement.
- Given statement u: `t = a op b`.
- If d reaches u, no other definition of `a` reaches u, and no definition of `z` exists on any path from d to u, then replace u by: `t = z op b`.


1:
r1 = M[A]
2:
r2 = M[B]
3:
r99 = r1 + r2
3':
r3 = r99
4:
r4 = r3 + r1
5:
branch r3 > r2
6:
r1 = r1 + 12
7:
r99 = r1 + r2
7':
r4 = r99
8:
r5 = r99
9:
M[r5] = r4

Sets

- Sets have been used in all the dataflow and control flow analyses presented.
- There are at least 3 representations which can be used:
  - Bit-Arrays:
    - Each *potential* member is stored in a bit of some array.
    - Insertion, Member is $O(1)$.
    - Assuming set size of N and word size of W - Union (OR) and Intersection (AND) is $O(N/W)$.
  - Sorted Lists/Trees:
    - Each member is stored in a list element.
    - Insertion, Member, Union, Intersection is $O(size)$. (Insertion, Member is $O(\log_2 size)$ in trees.)
    - Better for sparse sets than bit-arrays.
  - Hybrids: - Trees with bit-arrays
    - Use Tree to hold elements containing bit-arrays.
    - Union, Intersection is $O(size/W)$. Insertion, Member is $O(\log_2 size/W)$.

Basic Block Level Analysis

- To improve performance of dataflow, process at basic block level.
  - Represent the entire basic block by a single *super-instruction* which has any number of destinations and sources.
  - Run dataflow at basic block level.
  - Expand result to the instruction level.

The Phase Ordering Problem

- One of the tricky problems engineering compilers is how to order optimizations: *the phase ordering problem*.
- Hard because optimizations interact:
  - Constant folding creates opportunities for constant propagation.
  - Constant propagation creates opportunities for constant folding.
  - Both create opportunities for dead code elimination and common subexpression elimination.
  - Other optimizations such as function inlining, create further opportunities for almost all other optimizations
- Exam problem: Should Optimization X precede optimization Y? Explain.
- Often times, compilers apply apply a series of optimizations several times.