Analysis and Transformation

- Analysis:
  - Control Flow Analysis
  - Dataflow Analysis
- Transformation:
  - Register Allocation
  - Optimization
    - Copy propagation
    - Constant folding
    - Common subexpression elimination

Iterative Dataflow Analysis Framework

- These dataflow analyses are all very similar $\rightarrow$ define a framework.
- Specify:
  - The property you are interested in tracking. (liveness, reaching definitions, available expressions, etc.)
  - A *transfer function* - how to interpret the action of instructions with respect to the property
  - A *confluence operator* - $\vee$. (how to combine information from multiple predecessors or successors)
  - A *direction* - FORWARD or REVERSE.
  - The *initial conditions* for the analysis. (no variables considered live, no expressions considered available, etc.)
- For forward analyses:

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

$$OUT[n] = f(A, B, IN)$$

- For reverse analyses:

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

$$IN[n] = f\,(A, B, OUT)$$

Examples

Liveness:

- 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.
- Register t is *live-in* at CFG node n if t is live on any in-edge of n.
- Register t is *live-out* at CFG node n if t is live on any out-edge of n.

Assuming sets $USE[n]$ and $DEF[n]$, what directions should we do liveness and what are the dataflow equations for liveness?

Examples

Liveness:

- 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.
- Register t is *live-in* at CFG node n if t is live on any in-edge of n.
- Register t is *live-out* at CFG node n if t is live on any out-edge of n.

Assuming sets $USE[n]$ and $DEF[n]$, what directions should we do liveness and what are the dataflow equations for liveness?

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

$$IN[n] = USE[n] \cup (OUT[n] - DEF[n])$$

Reverse.

Reaching Definition Analysis

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

- 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.

Assuming sets $GEN[n]$ and $KILL[n]$, what directions should we do reaching definition analysis and what are the dataflow equations for reaching definition analysis?

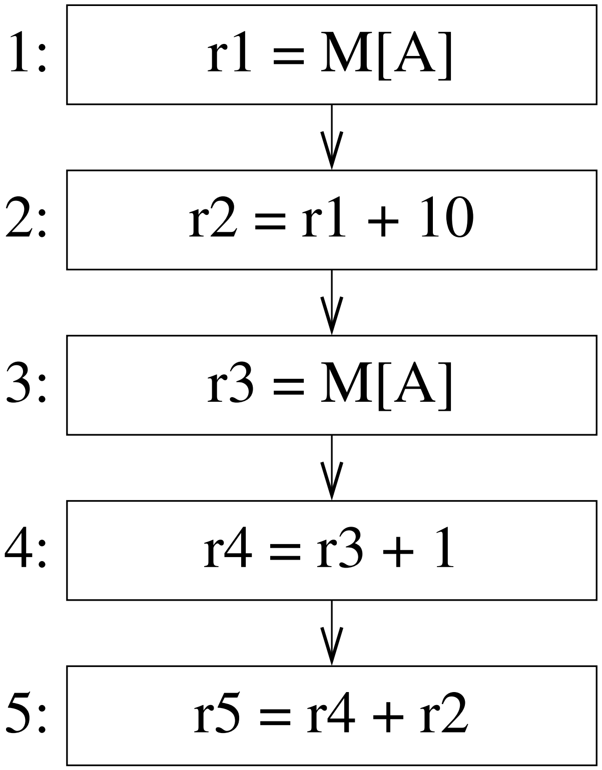
$$IN[n] = \cup_{p \in PRED[n]} OUT[p]$$

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

Forward.

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

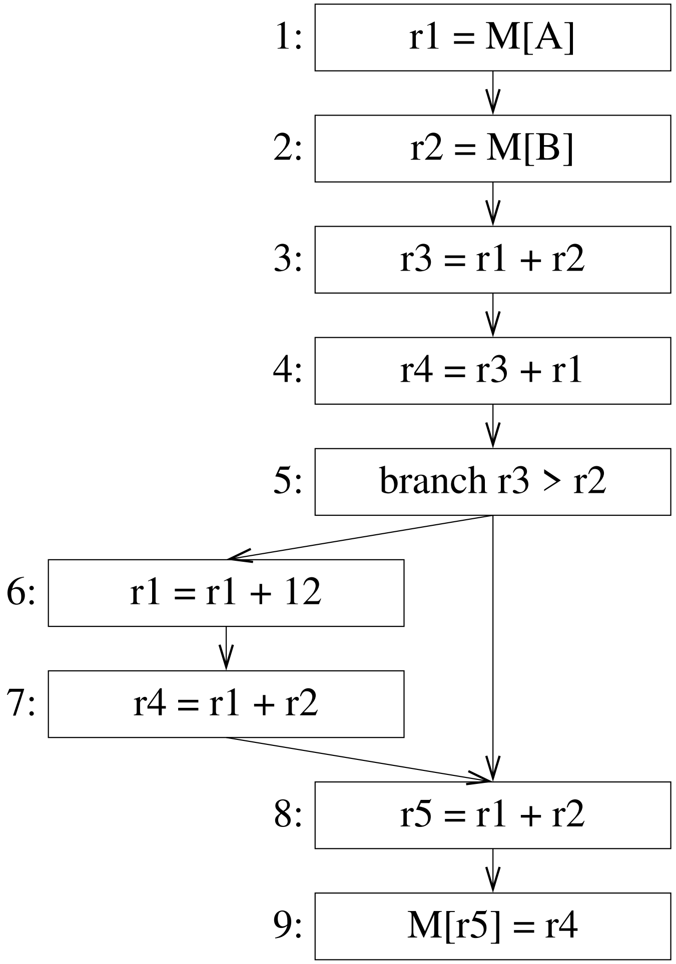
Available Expression Analysis:

- $exp(t)$ - set of all expressions containing t.
- Set definition ($A[n]$): $GEN[n]$ - the set of all expressions generated by n.
- Set definition ($B[n]$): $KILL[n]$ - the set of all expressions that n kills - $exp(n)$.
- Confluence operator ($\vee$): $\cap$
  - Use of $\cup$, required initialization of IN and OUT sets to $\emptyset$.
  - Use of $\cap$, requires initialization of IN and OUT sets to U (except for IN of entry node).
- Direction: FORWARD

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

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

Available Expression Analysis Example

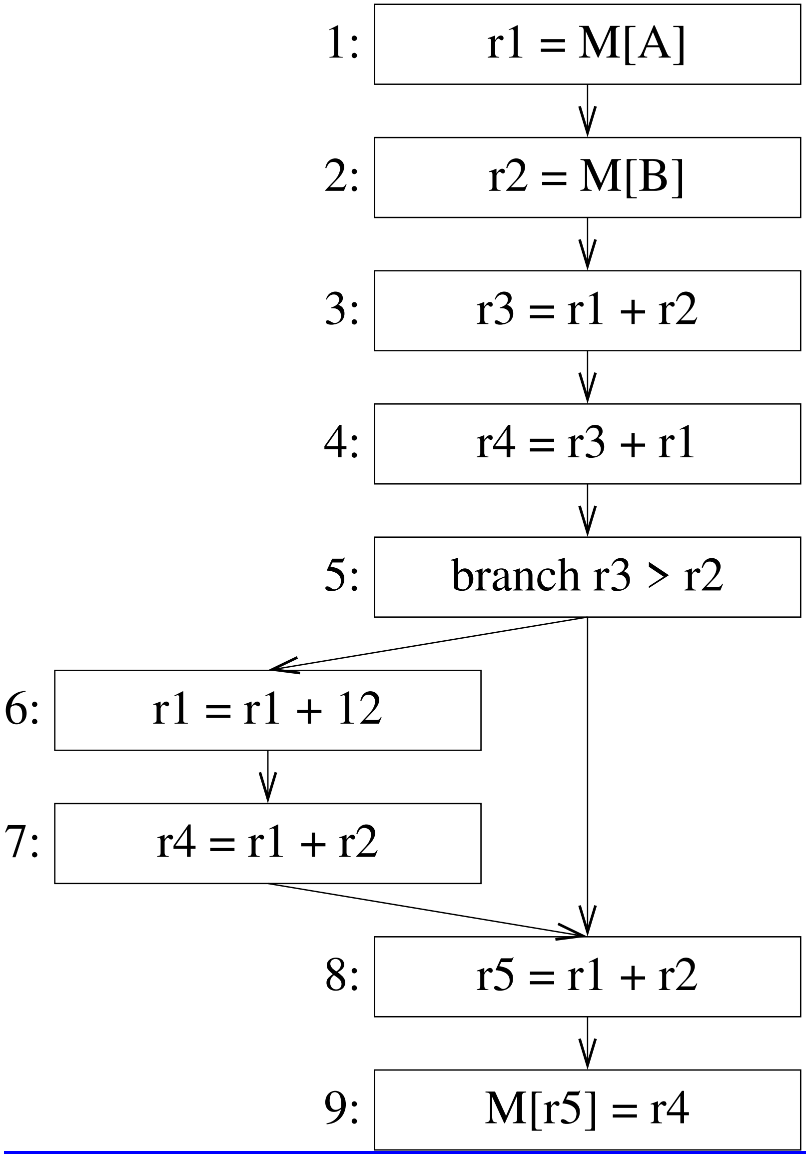


Common Subexpression Elimination

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



Common Subexpression Elimination - Example

`r1 + r2` **in node 8 is a common subexpression.**