Compiler Techniques

Lecture 11: Dataflow Analysis (Forward)

Tianwei Zhang

Outline

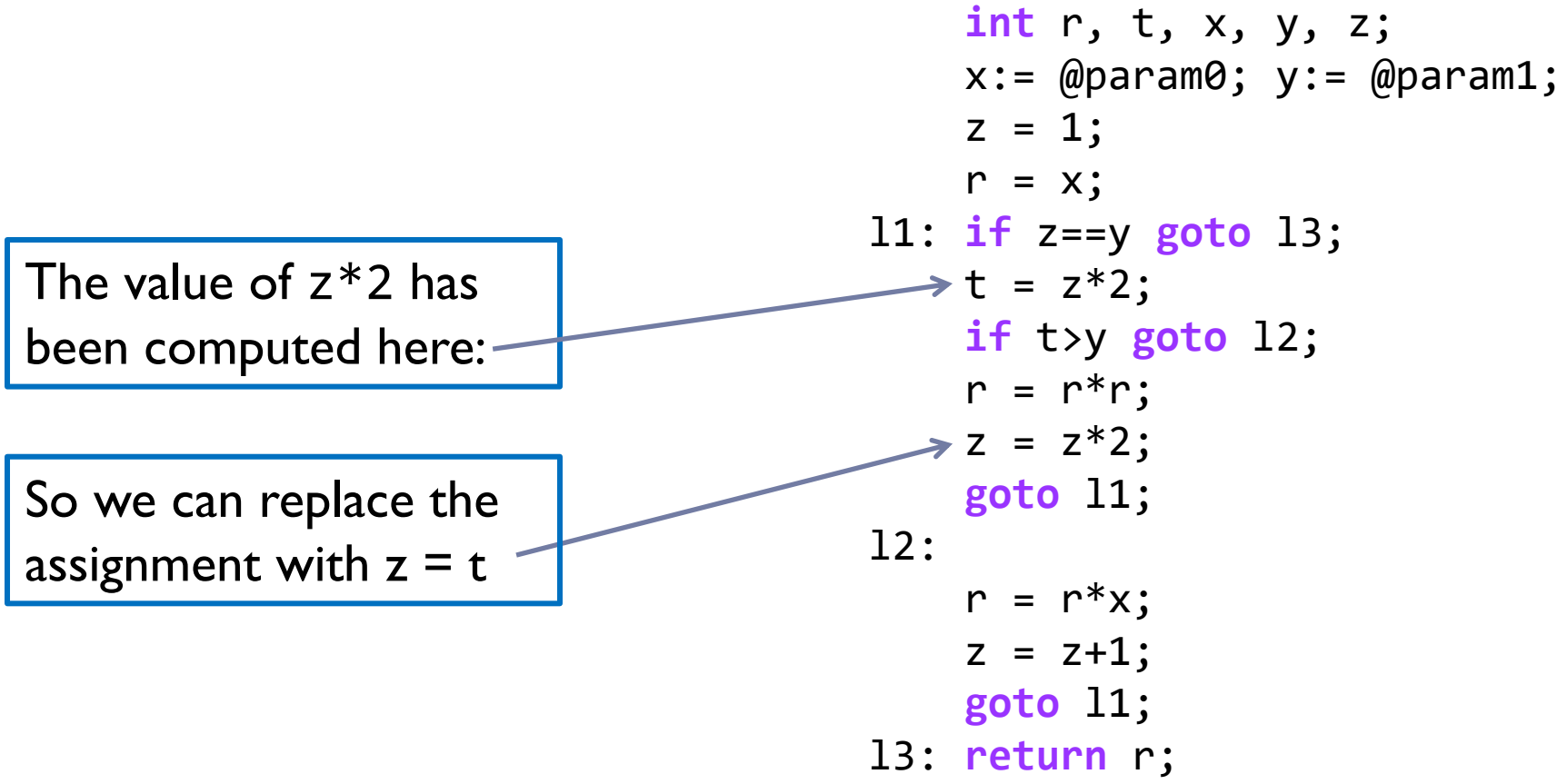
- **Available Expression Analysis**
- **Reaching Definition Analysis**
- **Summary of Dataflow Analysis**

Outline

- **Available Expression Analysis**
- **Reaching Definition Analysis**
- **Summary of Dataflow Analysis**

Common Subexpression Elimination

- Identify expressions that have been computed before, and reuse previously computed result



Correctness Conditions

- Two conditions need to be held in order to eliminate a common subexpression:
  1. The expression must have been computed previously on *every* possible execution path, not just on one
  2. None of the variables involved in computing the expression may have been updated in the meantime

```
    if z > 0 goto l
    x = y + z
l:  r = y + z
```

```
x = y + z
y = y + 1
r = y + z
```

Available Expressions

- These conditions can be formularized as the concept of an available expression in terms of the CGF
- An expression e is available at a point *p* if
  1. e is computed on every path from ENTRY to *p*, and
  2. no variable in e is overwritten between the computation of e and *p*
- Flow sets:
  - $in_A(n)$: the set of available expressions before *n*
  - $out_A(n)$: the set of available expression after *n*
- Goal of available expression analysis: compute $in_A(n)$ and $out_A(n)$ for every CFG node *n*
  - The set of possible expressions is infinite, but we only need to consider expressions that actually occur in the method (which is a finite set)
  - For simplicity, we only consider expressions computed from local variables using arithmetic operators

Transfer Functions

- If we already know $in_A(n)$, it is easy to compute $out_A(n)$:
  - An expression e is available after *n* if

    (1) Node *n* computes e, or

    (2) e is available before *n* and *n* does not write to any variable in e
- More denotations:
  - vars(e): the set of (local) variables in an expression e
  - comp(n): the set of expressions computed by n
  - def(*n*): the set of variables node *n* writes to
- **Transfer function** for $out_A(n)$:

$$out_A(n) = in_A(n) \setminus \{e \mid vars(e) \cap def(n) \neq \varnothing\} \cup comp\ (n)$$

- This is forward flow analysis.

Transfer Functions

- How do we get $in_A(n)$? There are two cases:
  - Node n is the Entry node. Then no expression is available before a method:

$$in_A(\text{Entry}) = \varnothing$$

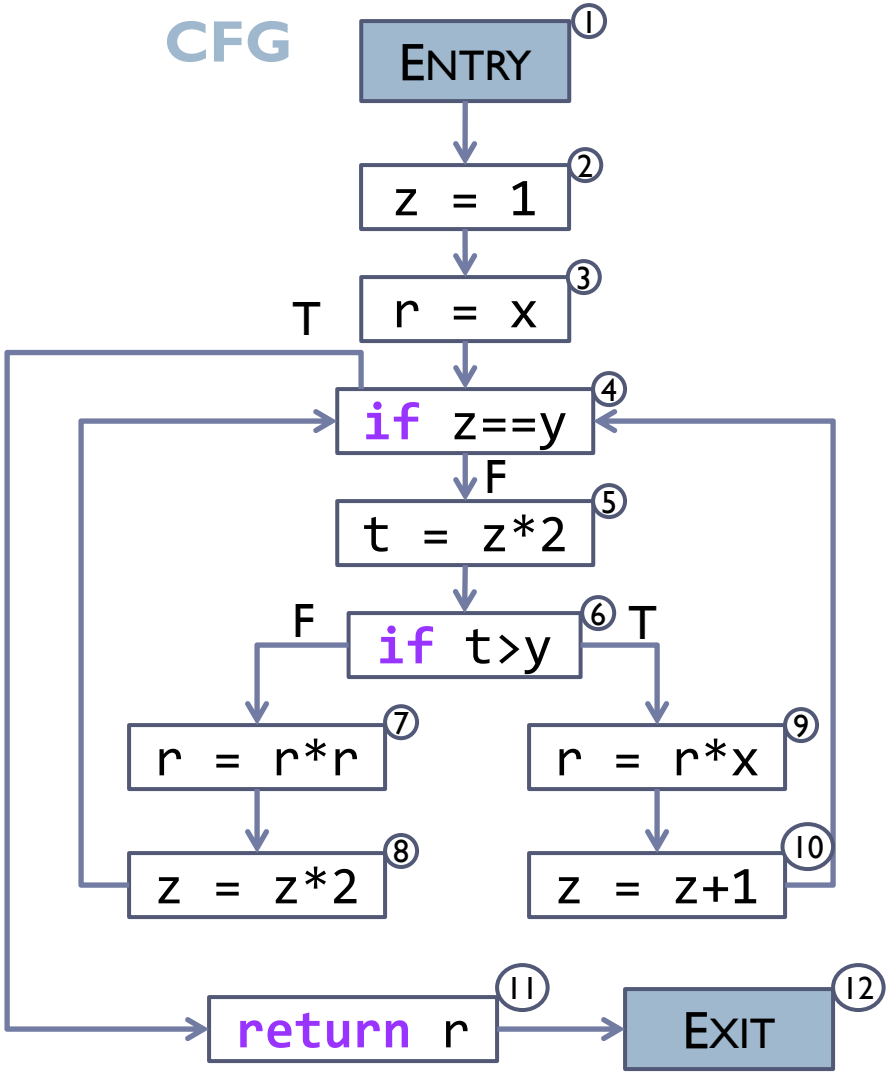
  - Node *n* has at least one predecessor node.
    - pred(*n*): the set of predecessor nodes of *n*.
    - An expression *e* is available before *n* if it is available after every predecessor of n

$$in_A(n) = \cap \{ out_A(m) \mid m \in pred(n) \}$$

- This is **must analysis,** as intersection is used to combine results from predecessor nodes.

Available Expressions Example

CFG



Transfer Functions

$out_A(1) = in_A(1)$

$out_A(2) = in_A(2) \setminus \{ z*2, z+1 \}$

$out_A(3) = in_A(3) \setminus \{ r*r, r*x \}$

$out_A(4) = in_A(4)$

$out_A(5) = in_A(5) \cup \{ z*2 \}$

$out_A(6) = in_A(6)$

$out_A(7) = in_A(7) \setminus \{ r*r, r*x \} \cup \{\}$

$out_A(8) = in_A(8) \setminus \{ z*2, z+1 \} \cup \{\}$

$out_A(9) = in_A(9) \setminus \{ r*r, r*x \} \cup \{\}$

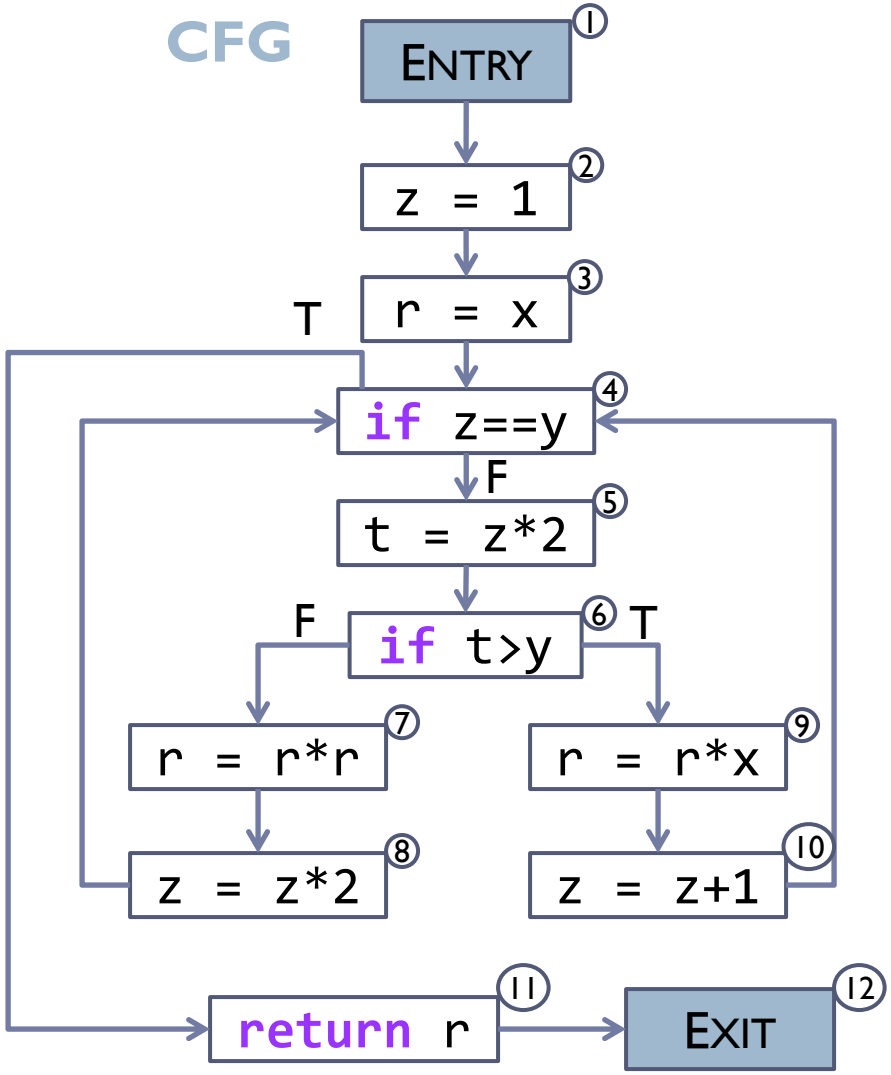
$out_A(10) = in_A(10) \setminus \{ z*2, z+1 \} \cup \{\}$

$out_A(11) = in_A(11)$

$out_A(12) = in_A(12)$

r*r should not be included in comp(7) as the value of r has been changed immediately by the statement in node 7. The same can be said for node 8, 9, and 10.

Available Expressions Example



Transfer Functions

$in_A(1) = \varnothing$

$in_A(2) = out_A(1)$

$in_A(3) = out_A(2)$

$in_A(4) = out_A(3) \cap out_A(8) \cap out_A(10)$

$in_A(5) = out_A(4)$

$in_A(6) = out_A(5)$

$in_A(7) = out_A(6)$

$in_A(8) = out_A(7)$

$in_A(9) = out_A(6)$

$in_A(10) = out_A(9)$

$in_A(11) = out_A(4)$

$in_A(12) = out_A(11)$

Method 1: Iterative Solution

- The equation system can be solved by iteration:
  1. Set $out_A(n) = in_A(n) = U$ for all nodes *n where U* is the set of *all* expressions in the method, since we use intersection to compute $in_A(n)$. Exception: $out_A(\text{ENTRY}) = in_A(\text{ENTRY}) = \varnothing$
  2. For every node *n*, recompute $in_A(n)$ based on the values we have computed in the previous iteration and then recompute $out_A(n)$ from $in_A(n)$
  3. Keep doing step 2 until the values do not change any further

Simplifying Transfer Equations

- To make it easier to solve the equations, we substitute away $in_A(n)$ equations, so we only have to solve for $out_A(n)$

$$
\begin{aligned}
out_A(1) &= in_A(1)\\
out_A(2) &= in_A(2) \setminus \{ z*2, z+1 \}\\
out_A(3) &= in_A(3) \setminus \{ r*r, r*x \}\\
out_A(4) &= in_A(4)\\
out_A(5) &= in_A(5) \cup \{ z*2 \}\\
out_A(6) &= in_A(6)\\
out_A(7) &= in_A(7) \setminus \{ r*r, r*x \} \cup \{\}\\
out_A(8) &= in_A(8) \setminus \{ z*2, z+1 \} \cup \{\}\\
out_A(9) &= in_A(9) \setminus \{ r*r, r*x \} \cup \{\}\\
out_A(10) &= in_A(10) \setminus \{ z*2, z+1 \} \cup \{\}\\
out_A(11) &= in_A(11)\\
out_A(12) &= in_A(12)
\end{aligned}
$$

$$
\begin{aligned}
in_A(1) &= \varnothing\\
in_A(2) &= out_A(1)\\
in_A(3) &= out_A(2)\\
in_A(4) &= out_A(3) \cap out_A(8) \cap out_A(10)\\
in_A(5) &= out_A(4)\\
in_A(6) &= out_A(5)\\
in_A(7) &= out_A(6)\\
in_A(8) &= out_A(7)\\
in_A(9) &= out_A(6)\\
in_A(10) &= out_A(9)\\
in_A(11) &= out_A(4)\\
in_A(12) &= out_A(11)
\end{aligned}
$$

$$
\begin{aligned}
out_A(1) &= \varnothing\\
out_A(2) &= out_A(1) \setminus \{ z*2, z+1 \}\\
out_A(3) &= out_A(2) \setminus \{ r*r, r*x \}\\
out_A(4) &= out_A(3) \cap out_A(8) \cap out_A(10)\\
out_A(5) &= out_A(4) \cup \{ z*2 \}\\
out_A(6) &= out_A(5)\\
out_A(7) &= out_A(6) \setminus \{ r*r, r*x \}\\
out_A(8) &= out_A(7) \setminus \{ z*2, z+1\}\\
out_A(9) &= out_A(6) \setminus \{ r*r, r*x \}\\
out_A(10) &= out_A(9) \setminus \{ z*2, z+1\}\\
out_A(11) &= out_A(4)\\
out_A(12) &= out_A(11)
\end{aligned}
$$

Method 2: Worklist Algorithm (Forward)

- $out_A(n)$ can only change if $in_A(n)$ changes, and $in_A(n)$ changes only if $out_A(m)$ changes for some predecessor m of n
- We can avoid unnecessary recomputation by keeping a *worklist* of nodes for which $in_A(n)$ has changed:

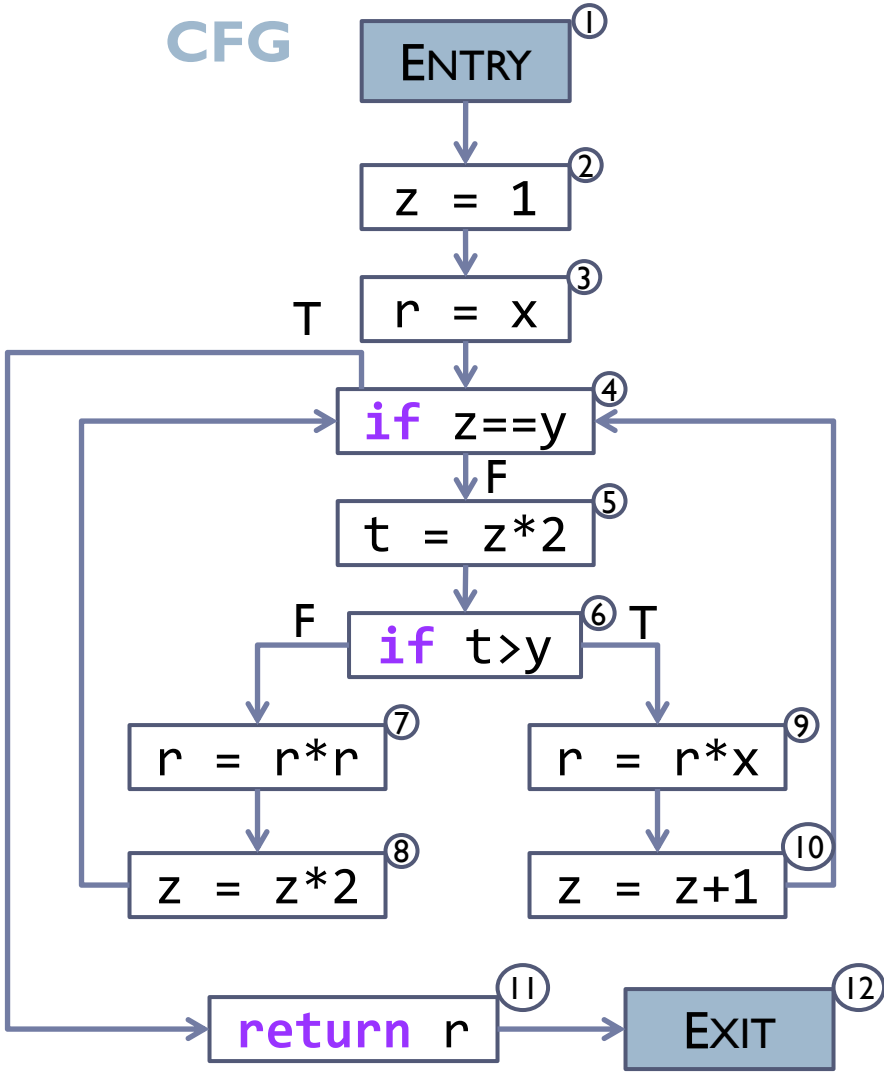
```
worklist = [ all nodes ]
    while worklist != empty do
        m = removeFirst(worklist)
        recompute out_A(m)
        if out_A(m) has changed then
            for each successor n of m
                compute in_A(n)
                if in_A(n) has changed then
                    put n into worklist (if not already in worklist)
```

Worklist Algorithm Example (Forward)

- Available expression analysis using the worklist algorithm.
  - We initialize the worklist to contain all nodes in the CFG to ensure that each node is evaluated at least once
  - These nodes can be in an arbitrary order. For forward flow analysis, we sort these nodes as the normal order in the worklist, to reduce number of iterations.
  - For all nodes, $out_A(n)$ and $in_A(n)$ are initialized to U = {z*2, r*r, r*x, z+1}, the set of all expressions in the method, except $in_A(1) = \varnothing$.

Example

Worklist	$out_A(m)$ &	$in_A(n)$
1, …, 12	$out_A(1) = \varnothing$	$in_A(2) = \varnothing$
2, …, 12	$out_A(2) = \varnothing$	$in_A(3) = \varnothing$
3, …, 12	$out_A(3) = \varnothing$	$in_A(4) = \varnothing$
4, …, 12	$out_A(4) = \varnothing$	$in_A(5) = \varnothing$
		$in_A(11) = \varnothing$
5, …, 12	$out_A(5) = \{z*2\}$	$in_A(6) = \{z*2\}$
6, …, 12	$out_A(6) = \{z*2\}$	$in_A(7) = \{z*2\}$
		$in_A(9) = \{z*2\}$
7, …, 12	$out_A(7) = \{z*2\}$	$in_A(8) = \{z*2\}$
8, …, 12	$out_A(8) = \varnothing$	$in_A(4) = \varnothing$
		(no change)
9, …, 12	$out_A(9) = \{z*2\}$	$in_A(10) = \{z*2\}$
10, …, 12	$out_A(10) = \varnothing$	$in_A(4) = \varnothing$
		(no change)
11, 12	$out_A(11) = \varnothing$	$in_A(12) = \varnothing$
12	$out_A(12) = \varnothing$	

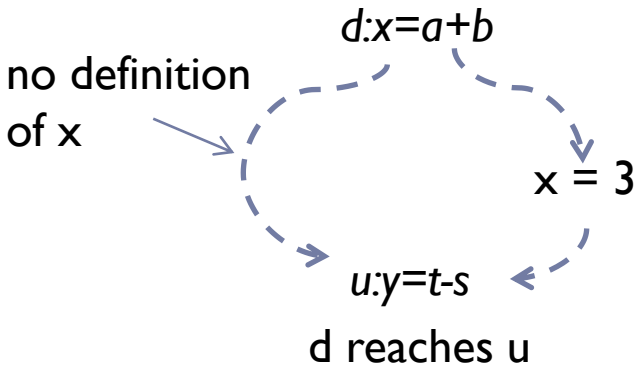


Outline

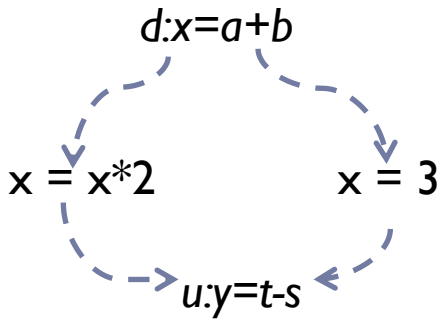
- Available Expression Analysis
- **Reaching Definition Analysis**
- Summary of Dataflow Analysis

Reaching Definition

- Every assignment is a definition
- Definition of a variable *x* at a program point *d* reaches point *u* if there exists a path from *d* to *u* such that *x* is not re-defined on this path.
- Problem statement:
  - For each point in the program, determine if each definition in the program reaches the point



d reaches u

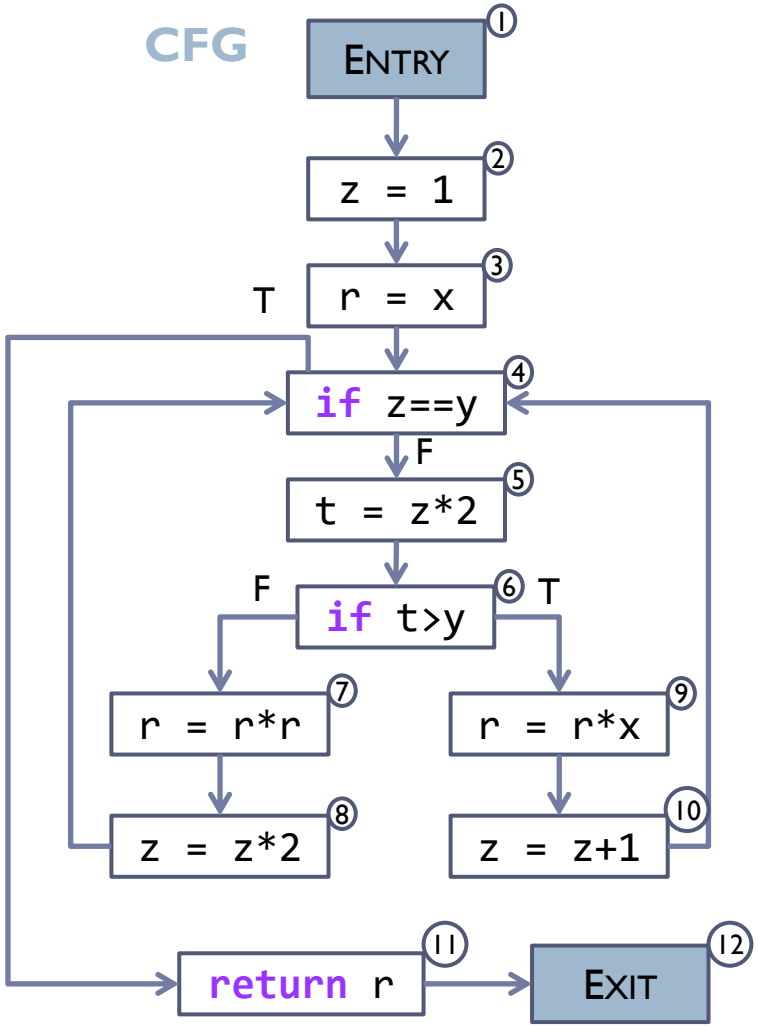


d cannot reach u

Reaching Definition

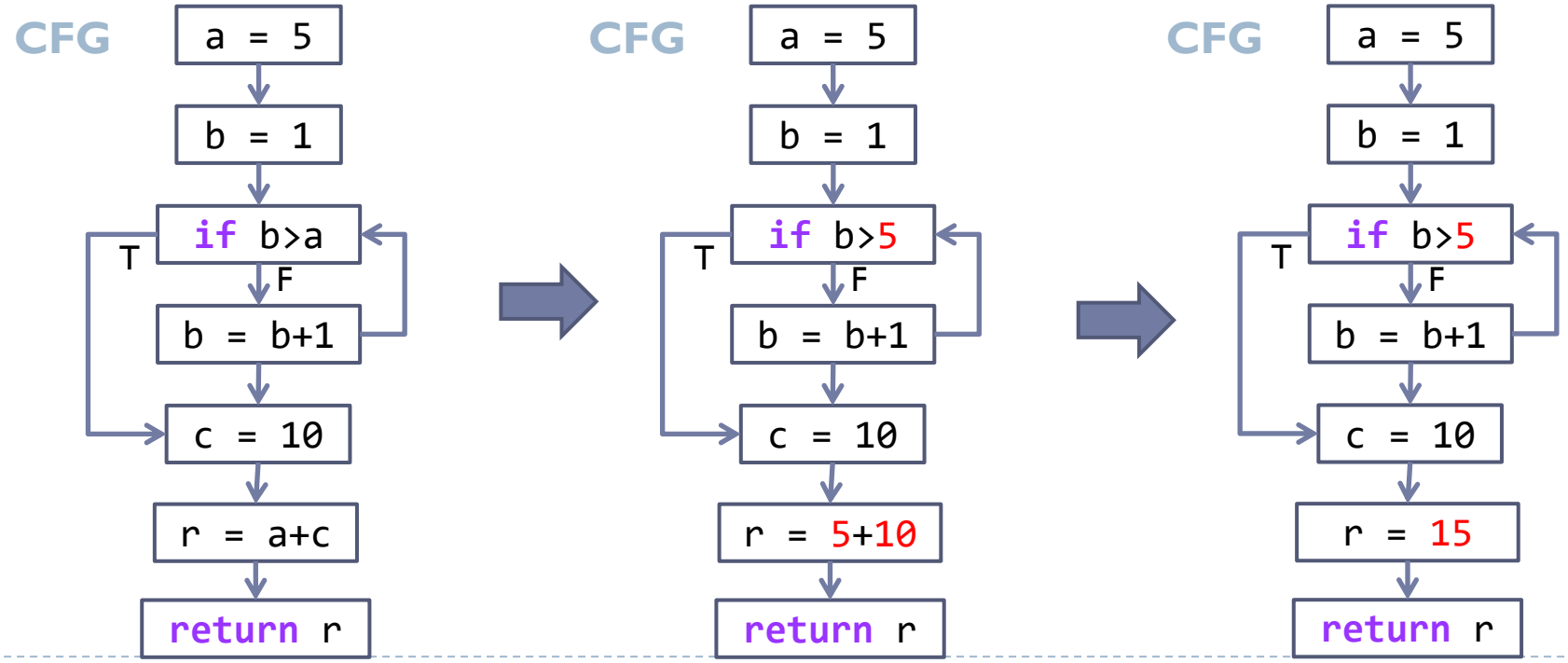
- d_2 can reach: {3,4,5,6,7,9,11}
- d_3 can reach: {4,5,6,11}
- d_5 can reach: {6,7,8,9,10,4,11}
- d_7 can reach: {8,4,5,6,11}
- d_8 can reach: {4,5,6,7,9,11}
- d_9 can reach: {10,4,5,6,11}
- d_{10} can reach: {4,5,6,7,9,11}

- Node 3: $\{d_2\}$
- Node 4: $\{d_2, d_3, d_5, d_7, d_8, d_9, d_{10}\}$
- Node 5: $\{d_2, d_3, d_7, d_8, d_9, d_{10}\}$
- ……



Application: Constant Propagation and Folding

- Given a node *n*: `t = a op b`:
  - If definition e: `a = c` (where `c` is a constant) reaches *n* and no other definition of `a` reaches *n*, then replace *n* with e: `t = c op b`
  - If `a` and `b` are constant, compute `v` as `a op b` and replace *n* as `t = v`



Reaching Definition Analysis

- A definition *e* (at node *m*) can reach node *n* if
  1. There exists a path *p* from *m* to *n*, and
  2. The variable of *e* is not redefined along this path
- Flow sets:
  - $in_R(n)$: the set of definitions that can reach before *n*
  - $out_R(n)$: the set of definitions that can reach after *n*
- Goal of reaching definition analysis: compute $in_R(n)$ and $out_R(n)$ for every CFG node *n*
  - We only need to consider definitions that actually occur in the method

Transfer Functions

- If we already know $in_R(n)$, it is easy to compute $out_R(n)$:
  - A definition e can reach after *n* if

    (1) Node *n* gives e, or

    (2) e reaches before *n* and *n* does not re-define the variable in e
- More denotations:
  - gen(n): the set of definitions given by *n*
  - kill(*n*): the set of definitions whose variables are written by node *n*
- **Transfer function** for $in_R(n)$:

$$out_R(n) = in_R(n) \setminus kill(n) \cup gen\ (n)$$

- This is forward flow analysis.

Transfer Functions

- How do we get $in_R(n)$? There are two cases:
  - Node n is the Entry node. Then no definition can reach before a method:

$$in_R(Entry) = \varnothing$$

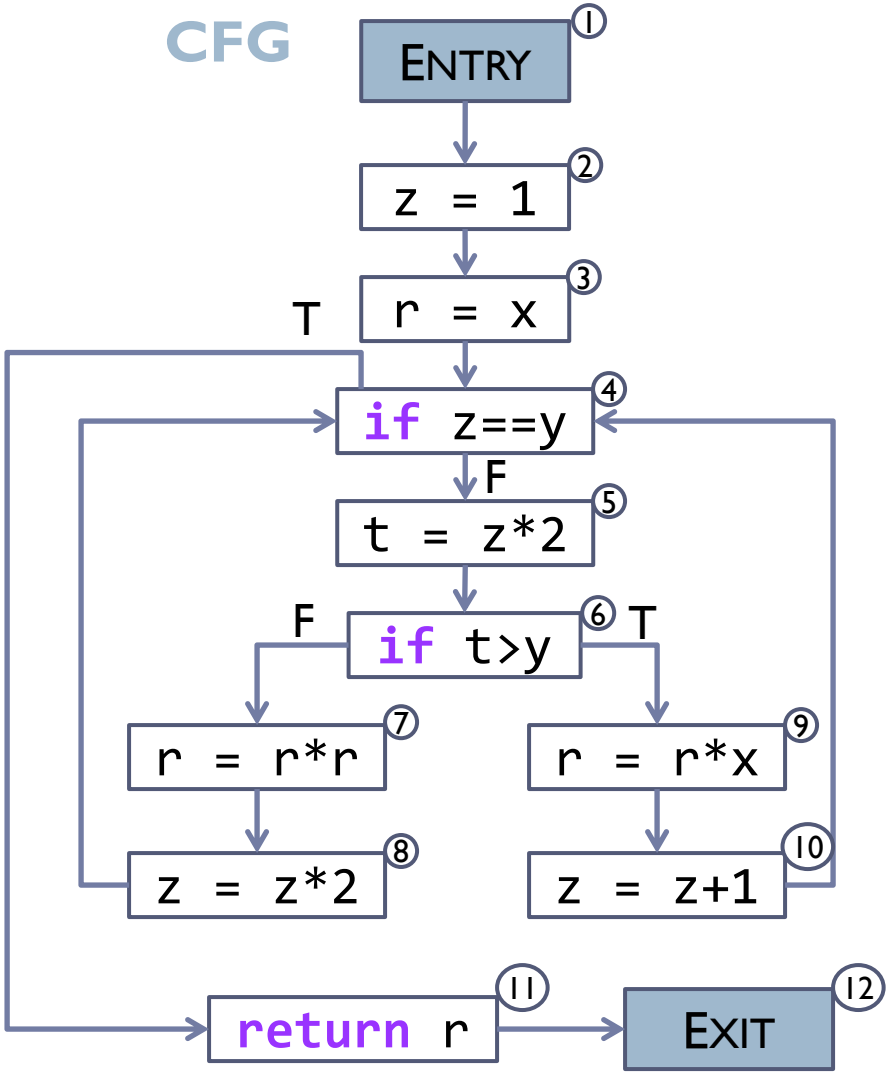
  - Node *n* has at least one predecessor node.
    - pred(*n*): the set of predecessor nodes of *n*.
    - A definition e can reach before *n* if it can reach any predecessor of n

$$in_R(n) = \cup \{ out_R(m) \mid m \in pred(n) \}$$

- This is **may analysis,** as union is used to combine results from predecessor nodes.

Reaching Definition Example

CFG



Transfer Functions

$out_A(1) = in_A(1)$

$out_A(2) = in_A(2) \setminus \{d_8, d_{10}\} \cup \{d_2\}$

$out_A(3) = in_A(3) \setminus \{d_7, d_9\} \cup \{d_3\}$

$out_A(4) = in_A(4)$

$out_A(5) = in_A(5) \setminus \{\} \cup \{d_5\}$

$out_A(6) = in_A(6)$

$out_A(7) = in_A(7) \setminus \{d_3, d_9\} \cup \{d_7\}$

$out_A(8) = in_A(8) \setminus \{d_2, d_{10}\} \cup \{d_8\}$

$out_A(9) = in_A(9) \setminus \{d_3, d_7\} \cup \{d_9\}$

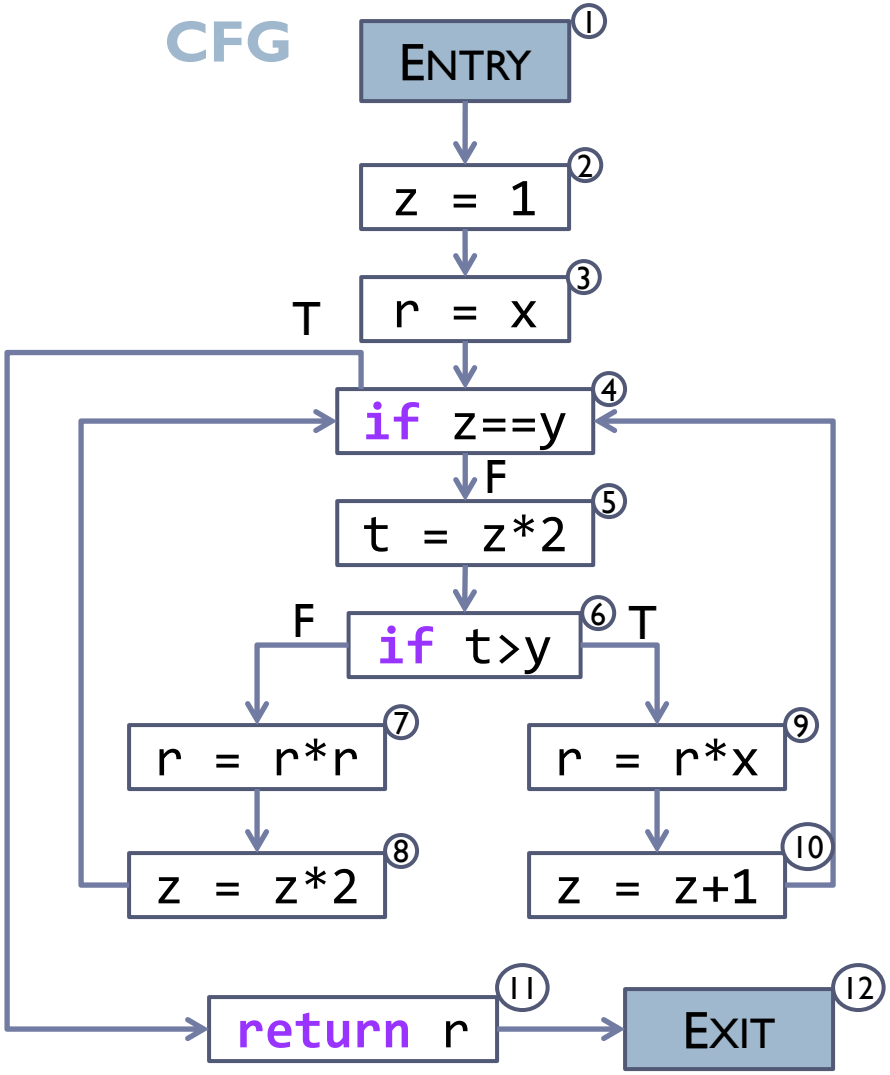
$out_A(10) = in_A(10) \setminus \{d_2, d_8\} \cup \{d_{10}\}$

$out_A(11) = in_A(11)$

$out_A(12) = in_A(12)$

Reaching Definition Example

CFG



Transfer Functions

$in_A(1) = \varnothing$

$in_A(2) = out_A(1)$

$in_A(3) = out_A(2)$

$in_A(4) = out_A(3) \cup out_A(8) \cup out_A(10)$

$in_A(5) = out_A(4)$

$in_A(6) = out_A(5)$

$in_A(7) = out_A(6)$

$in_A(8) = out_A(7)$

$in_A(9) = out_A(6)$

$in_A(10) = out_A(9)$

$in_A(11) = out_A(4)$

$in_A(12) = out_A(11)$

Solving Transfer Functions

- The transfer functions can be solved in the same way as available expression analysis, using iterative or worklist algorithms.

- Initialization:
  - $out_R(n) = in_R(n) = \varnothing$

Outline

- Available Expression Analysis
- Reaching Definition Analysis
- **Summary of Dataflow Analysis**

Categorization of Dataflow Analysis

- Forward analysis: the set after the node depends on the set before the node
- Backward analysis: the set before the node depends on the set after the node
- Must analysis: intersection is used to combine results from predecessor/successor nodes.
- May analysis: union is used to combine results from predecessor/successor nodes.

	Forward	Backward
Must	**Available Expression**	**Very Busy Expression**
May	**Reaching Definition**	**Liveness**

Optimization Purposes

- Liveness:
  - Analyzing variables. Used for register allocation.
- Very busy expression:
  - Analyzing expressions. Used for code hoisting.
- Available expression
  - Analyzing expressions. Used for common subexpression elimination.
- Reaching definition:
  - Analyzing definitions. Used for constant propagation and folding

	Forward	Backward
Must	**Available Expression**	**Very Busy Expression**
May	**Reaching Definition**	**Liveness**

Transfer Functions

- Two flow sets in(*n*) and out(*n*) for before and after every node *n* in the CFG
  - Liveness: set of variables
  - Very busy expression: set of expressions
  - Available expression: set of expressions
  - Reaching definition: set of definitions
- For every node, there is a function that computes either in(*n*) from out(*n*) (backward analysis), or out(*n*) from in(*n*) (forward analysis)
- There is also another function that computes either out(*n*) from in(*m*) for the successors *m* of *n* (backward analysis), or in(*n*) from out(*m*) for the predecessors *m* of *n* (forward analysis)

Worklist Algorithm

Backward analysis

```
worklist = [ all nodes ]
   while worklist != empty do
      m = removeFirst(worklist)
      recompute in(m)
      if in(m) has changed then
         for each predecessor n of m
            compute out(n)
            if out(n) has changed then
               put n into worklist (if not
                     already in worklist)
```

Forward analysis

```
worklist = [ all nodes ]
   while worklist != empty do
      m = removeFirst(worklist)
      recompute out(m)
      if out(m) has changed then
         for each successor n of m
            compute in(n)
            if in(n) has changed then
               put n into worklist (if not
                     already in worklist)
```

Worklist Algorithm Initialization

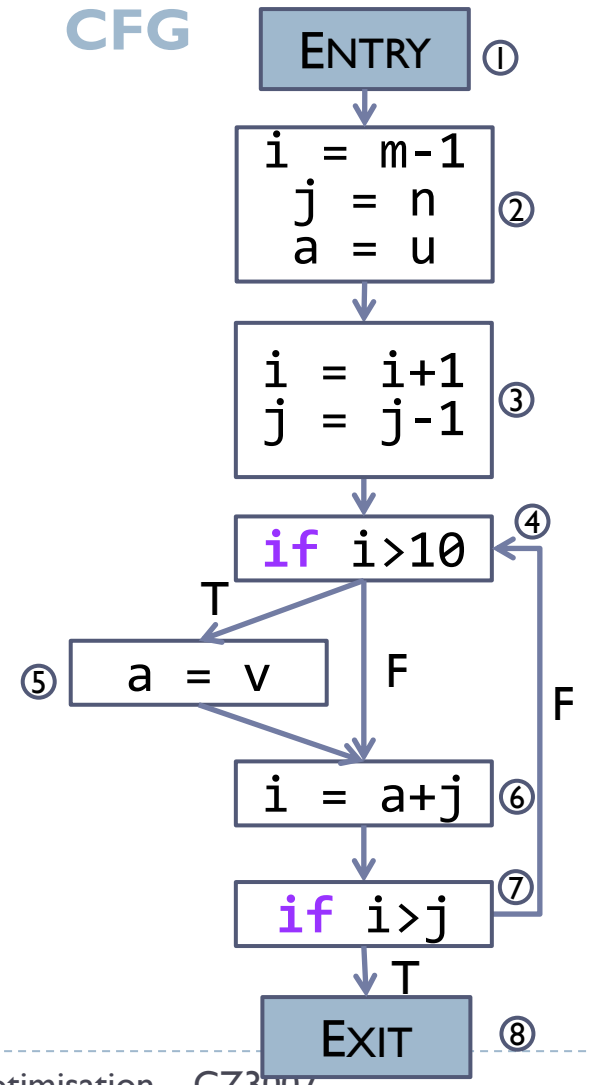
- For all nodes, if the join operator is union, then out(n) = in(n) = ∅, otherwise U
  - Exception: in(ENTRY) (forward analysis) or out(EXIT) (backward analysis) which are defined by the analysis
- We initialise the worklist to contain all nodes in the CFG to ensure that each node is evaluated at least once
- The order in which the worklist is initialized is important:
  - As far as possible the nodes should be ordered so that a node is only evaluated after all its predecessors or successors have updated their values.
  - Generally, we initialize the worklist in a normal order for forward analysis, and reversed order for backward analysis. But this can be difficult when the CFG has cycles

For a New Dataflow Analysis Problem

- Direction: forward or backward?
- Two sets of transfer function
  - May or must?
  - Sets of definition
- Initial values
  - Special case for ENTRY or EXIT node

Control Flow Graph with Basic Blocks

- In these two lectures, we build control flow graph with each node as one instruction.
- Actually a node can be a basic block, which consists of a sequence of instructions without branches
- Data flow analysis methods can be performed in the same way. The definition of each set (out, in, use, def, comp, etc.) needs consider all the instructions in the same block
- Analysis at the basic block level can improve the performance



Conflicts between Optimisations

- Sometimes different optimisations may be in conflict with each other
  - e.g., common subexpression elimination increases the live range of local variables (*i.e.*, the number of nodes where they are live)
  - This makes register allocation harder, so the performance gain of avoiding recomputation may be lost because more memory accesses are needed
- Sometimes the order of optimisations may also affect the final results.
- Deciding which optimisations to apply and when to apply them is difficult, and there is no general recipe