

# Dataflow analysis

---

# Dataflow analysis: what is it?

---

- A common framework for expressing algorithms that compute information about a program
- Why is such a framework useful?

# Dataflow analysis: what is it?

---

- A common framework for expressing algorithms that compute information about a program
- Why is such a framework useful?
- Provides a common language, which makes it easier to:
  - communicate your analysis to others
  - compare analyses
  - adapt techniques from one analysis to another
  - reuse implementations (eg: dataflow analysis frameworks)

# Data flow analysis

---

- Goal :
  - collect information about how a procedure manipulates its data
- This information is used in various optimizations
  - For example, knowledge about what expressions are available at some point helps in common subexpression elimination.
- IMPORTANT!
  - Data flow analysis should never tell us that a transformation is safe when in fact it is not.
  - It is better to not perform a **valid** optimization that to perform one that changes the function of the program.

# Data flow analysis

---

- IMPORTANT!
  - Data flow analysis should never tell us that a transformation is safe when in fact it is not.
  - When doing data flow analysis we must be
    - Conservative
      - Do not consider information that may not preserve the behavior of the program
    - Aggressive
      - Try to collect information that is as exact as possible, so we can get the greatest benefit from our optimizations.

# Global Iterative Data Flow Analysis

---

- Global:
  - Performed on the control flow graph
  - Goal = to collect information at the **beginning** and **end** of each basic block
- Iterative:
  - Construct data flow equations that describe how information flows through each basic block and solve them by iteratively converging on a solution.

# Global Iterative Data Flow Analysis

---

- Components of data flow equations
  - Sets containing collected information
    - **In** (or **entry**) set: information coming into the BB from outside (following flow of data)
    - **gen** set: information generated/collected within the BB
    - **kill** set: information that, due to action within the BB, will affect what has been collected outside the BB
    - **out** (or **exit**) set: information leaving the BB
  - Functions (operations on these sets)
    - **Transfer functions** describe how information changes as it flows through a basic block
    - **Meet functions** describe how information from multiple paths is combined.

# Global Iterative Data Flow Analysis

- Algorithm sketch
  - Typically, a bit vector is used to store the information.
    - For example, in reaching definitions, each bit position corresponds to one definition.
  - We use an iterative fixed-point algorithm.
  - Depending on the nature of the problem we are solving, we may need to traverse each basic block in a forward (top-down) or backward direction.
    - The order in which we "visit" each BB is not important in terms of algorithm correctness, but is important in terms of efficiency.
  - In & Out sets should be initialized in a conservative and aggressive way.

```
Initialize gen and kill sets
Initialize in or out sets (depending on "direction")
while there are no changes in in and out sets {
    for each BB {
        apply meet function
        apply transfer function
    }
}
```

# Typical problems

---

- Reaching definitions
  - For each use of a variable, find all definitions that reach it.
- Upward exposed uses
  - For each definition of a variable, find all uses that it reaches.
- Live variables
  - For a point  $p$  and a variable  $v$ , determine whether  $v$  is live at  $p$ .
- Available expressions
  - Find all expressions whose value is available at some point  $p$ .
- Very Busy expressions
  - Find all expressions whose value will be used in all the next paths

# Reaching definitions

---

- Determine which definitions of a variable may reach each use of the variable.
  - For each use, list the definitions that reach it. This is also called a **ud-chain**.
  - In global data flow analysis, we collect such information at the endpoints of a basic block, but we can do additional local analysis within each block.
- Uses of reaching definitions :
  - constant propagation
    - we need to know that all the definitions that reach a variable assign it to the same constant
  - copy propagation
    - we need to know whether a particular copy statement is the only definition that reaches a use.
  - code motion
    - we need to know whether a computation is loop-invariant

# Something obvious

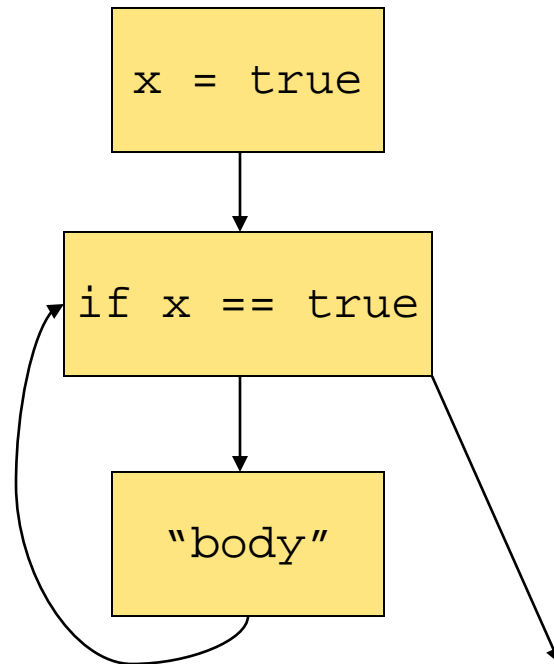
---

```
boolean x = true;
while (x) {
    . . . // no change to x
}
```

- Doesn't terminate.
- **Proof:** only assignment to **x** is at top, so **x** is always true.

# As a Control Flow Graph

---



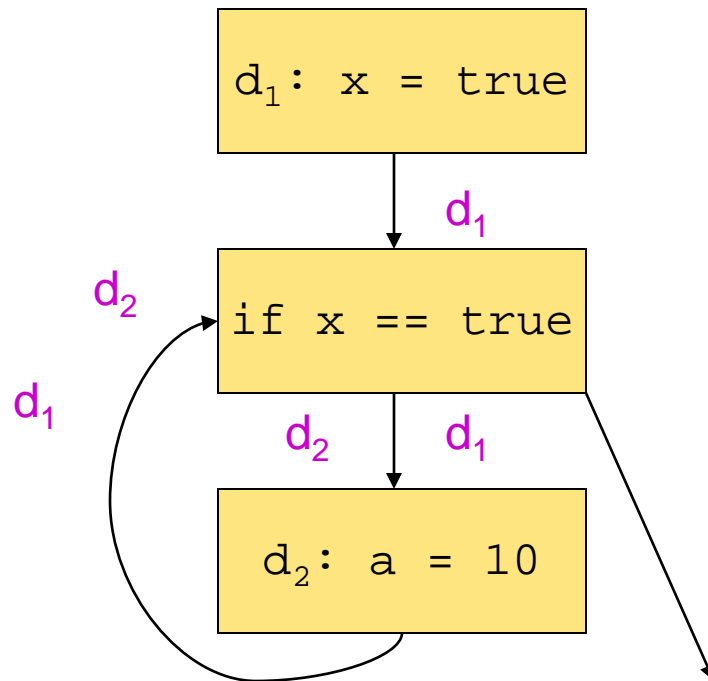
# Formulation: Reaching Definitions

---

- Each place some variable  $x$  is assigned is a *definition*.
- **Ask**: for this use of  $x$ , where could  $x$  last have been defined?
- **In our example**: only at `x=true`.

# Example: Reaching Definitions

---



# Clincher

---

- Since `at x == true`,  $d_1$  is the only definition of `x` that reaches, it must be that `x` is true at that point.
- The conditional is not really a conditional and can be replaced by a branch.

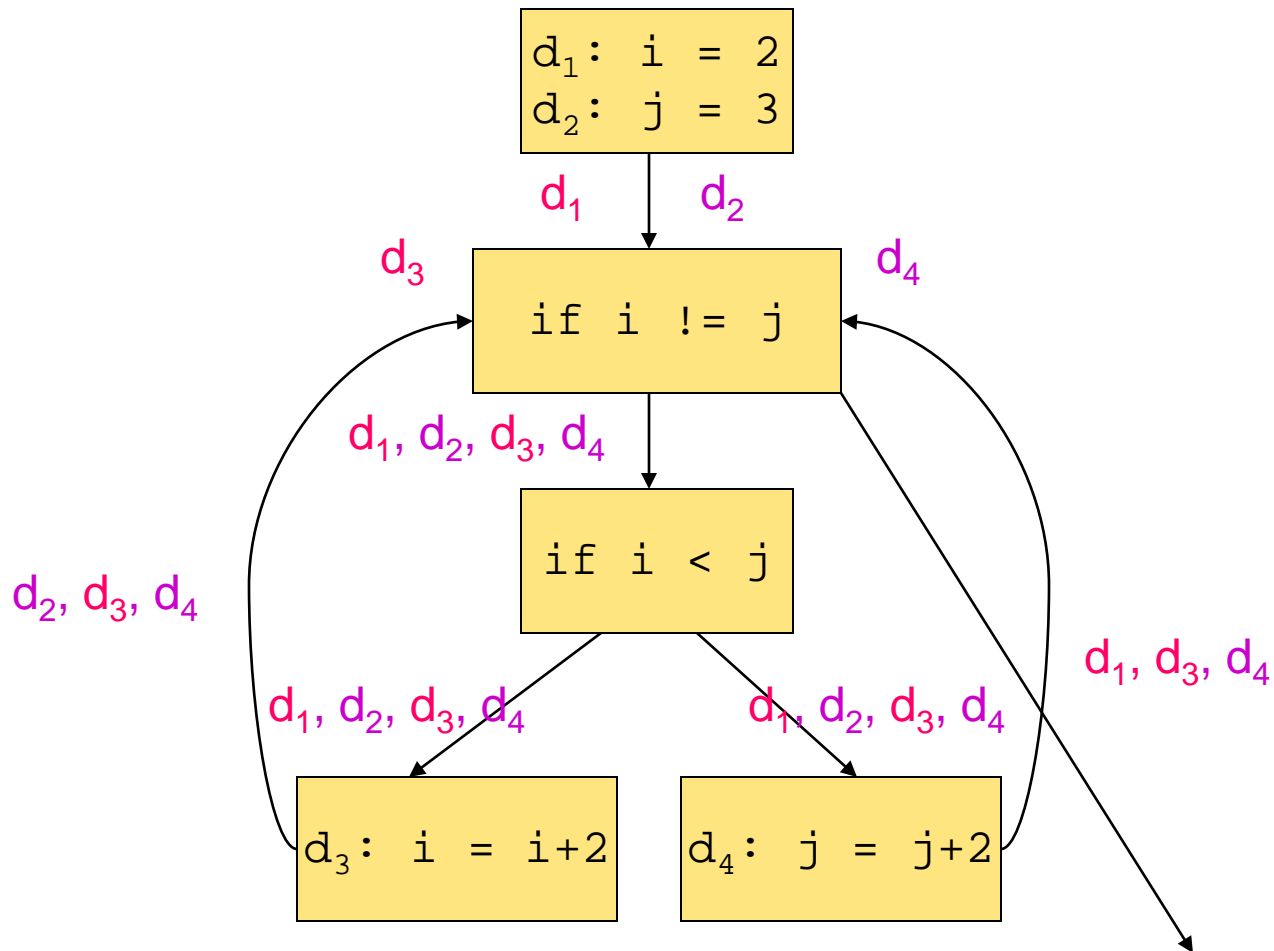
# Not Always That Easy

---

```
int i = 2; int j = 3;
while (i != j) {
    if (i < j) i += 2;
    else j += 2;
}
```

- We'll develop techniques for this problem, but later ...

# The Control Flow Graph



# DFA is Sufficient Only

---

- In this example,  $i$  can be defined in two places, and  $j$  in two places.
- No obvious way to discover that  $i \neq j$  is always true.
- But OK, because reaching definitions is sufficient to catch most opportunities for *constant folding* (replacement of a variable by its only possible value).

# Be Conservative!

---

- (Code optimization only)
- It's **OK** to discover a subset of the opportunities to make some code-improving transformation.
- It's **not OK** to think you have an opportunity that you don't really have.

# Example: Be Conservative

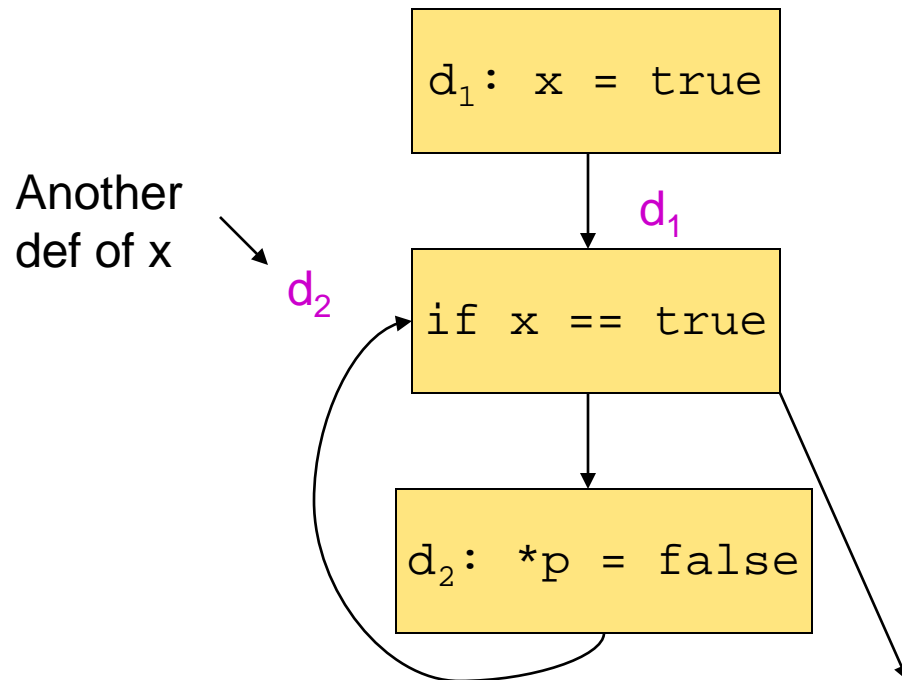
---

```
boolean x = true;
while (x) {
    . . . *p = false; . . .
}
```

- Is it possible that **p** points to **x**?

# As a Control Flow Graph

---



# Possible Resolution

---

- Just as data-flow analysis of “reaching definitions” can tell what definitions of  $x$  might reach a point, another DFA can eliminate cases where  $p$  definitely does not point to  $x$ .
- **Example:** the only definition of  $p$  is  $p = \&y$  and there is no possibility that  $y$  is an alias of  $x$ .

# Formalization:

## Reaching definitions Analysis

---

- A definition  $D$  reaches a point  $p$  if there is a path from  $D$  to  $p$  along which  $D$  is not killed.
- A definition  $D$  of a variable  $x$  is killed when there is a redefinition of  $x$ .
- How can we represent the set of definitions reaching a point?

# Reaching definitions

---

- What is safe?
  - To assume that a definition reaches a point even if it turns out not to.
  - The computed set of definitions reaching a point  $p$  will be a **superset** of the actual set of definitions reaching  $p$
  - It's a “possible”, not a “definite” property
  - Goal : make the set of reaching definitions as small as possible (i.e. as close to the actual set as possible)

# Reaching definitions

---

- How are the **gen** and **kill** sets defined?
  - **gen**[B] = {definitions that appear in B and reach the end of B}
  - **kill**[B] = {all definitions that never reach the end of B}
- What is the direction of the analysis?
  - forward
  - **out**[B] = **gen**[B]  $\cup$  (**in**[B] - **kill**[B])

# Reaching definitions

---

- What is the **confluence** operator?
  - union
  - $\mathbf{in}[P] = \cup \mathbf{out}[Q]$ , over the predecessors Q of P
- How do we initialize?
  - start small
    - Why? Because we want the resulting set to be as small as possible
  - for each block B initialize  $\mathbf{out}[B] = \mathbf{gen}[B]$

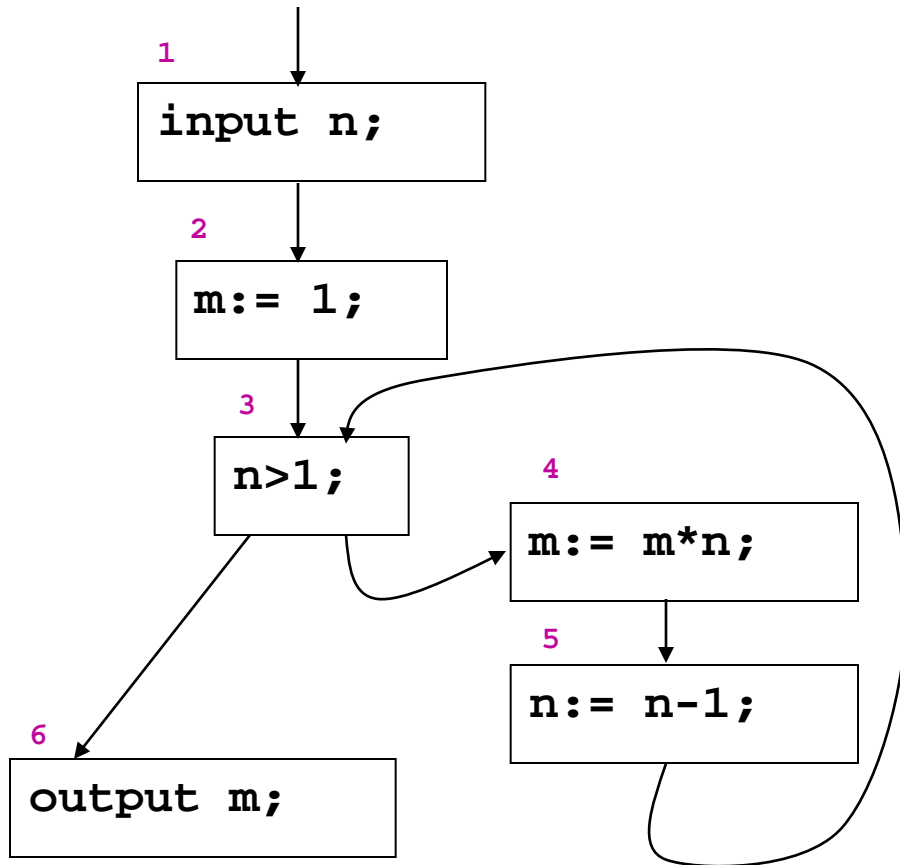
# Formal specification

---

- The reaching Definition Analysis is specified by the following equations:
- For each program point,

$$RD_{in}(p) = \begin{cases} \perp & \text{if } p \text{ is the initial point in the control graph} \\ \bigcup \{ RD_{out}(q) \mid \text{there is an arrow from } q \text{ to } p \} & \end{cases}$$

$$RD_{out}(p) = gen_{RD}(p) \cup (RD_{in}(p) \setminus kill_{RD}(p))$$



- $RD_{in}(1) = \{(n, ?), (m, ?)\}$   
 $RD_{out}(1) = \{(n, ?), (m, ?)\}$
- $RD_{in}(2) = \{(n, ?), (m, ?)\}$   
 $RD_{out}(2) = \{(n, ?), (m, 2)\}$
- $RD_{in}(3) = RD_{out}(2) \cup RD_{out}(5)$   
 $= \{(n, ?), (n, 5), (m, 2), (m, 4)\}$   
 $RD_{out}(3) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$
- $RD_{in}(4) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$   
 $RD_{out}(4) = \{(n, ?), (n, 5), (m, 4)\}$
- $RD_{in}(5) = \{(n, ?), (n, 5), (m, 4)\}$   
 $RD_{out}(5) = \{(n, 5), (m, 4)\}$
- $RD_{in}(6) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$   
 $RD_{out}(6) = \{(n, ?), (n, 5), (m, 2), (m, 4)\}$

# Algorithm

---

- **Input:** Control Graph Diagram
- **Output :** RD
- **Steps:**
  - step 1 (inicialization):
    - $RD_{in}(p)$  is the emptyset for each p
    - $RD_{in}(1) = \tau = \{(x,?) \mid x \text{ is a program variable}\}$

---

- Step 2 (iteration)

- Flag = TRUE;

- while Flag

- Flag = FALSE;

- for each program point p

- new =  $U\{f(RD, q) \mid (q, p) \text{ is an edge of the graph}\}$

- if  $RD_{in}(p) \neq new$

- Flag = TRUE;

- $RD_{in}(p) = new;$

where  $f(RD, q) = gen_{RD}(q) \cup (RD_{in}(q) \setminus kill_{RD}(q))$

# Example

```
[ input n; ]1  
[ m := 1; ]2  
  [ while n > 1 do ]3  
    [ m := m * n; ]4  
    [ n := n - 1; ]5  
[ output m; ]6
```

- $RD_{in}(1) = \{(n,?), (m,?)\}$
- $RD_{in}(2) = \{(n,?), (m,?)\}$
- $RD_{in}(3) = \{(n,?), (n,5), (m,2), (m,4)\}$
- $RD_{in}(4) = \{(n,?), (n,5), (m,4)\}$
- $RD_{in}(5) = \{(n,5), (m,4)\}$
- $RD_{in}(6) = \{(n,?), (n,5), (m,2), (m,4)\}$

# Using Reaching Definition analysis for Global Constant Folding

---

# Constant Folding

---

- By using the Reaching Definitions Analysis, we can now formally define the rules for global constant folding optimizations.
- If  $P$  is a program, we denote by  $RD$  the minimal solution of the Reaching Definition Analysis for  $P$ .
- A statement  $S$  in  $P$  can be transformed in a more optimized statement, by applying one of the rules below, and we'll use the notation:

$$RD \vdash S \triangleright S'$$

# Rule 1

---

$$1. \quad RD \vdash [x := a]^v \triangleright [x := a[y \rightarrow n]]^v$$

$$\text{if } y \in FV(a) \quad \wedge \quad (y, ?) \notin RD_{\text{entry}}(v) \quad \wedge \\ \text{for every } (z, \mu) \in RD_{\text{entry}}(v): (z=y \Leftrightarrow [\dots]^\mu \hat{=} [y:=n]^\mu)$$

The rule says that a variable can be substituted by a constant value if the Reaching Definition Analysis ensures that this is the only value that the variable can hold.

$a[y \rightarrow n]$  means that in the expression  $a$ , variable  $y$  is substituted by value  $n$

$FV(a)$  denotes the set of free variables in the expression  $a$ .

# Rule 2

---

2.  $RD \vdash [x := a]^v \triangleright [x := n]^v$

if  $FV(a) = \emptyset \wedge a \notin \text{Num} \wedge$  the value of  $a$  is  $n$

- The rule says that an expression can be evaluated at compile time if it contains no free variables.

# Composition rules

---

$$3. \quad RD \vdash S_1 \triangleright S'_1 \Leftrightarrow$$

$$RD \vdash S_1 ; S_2 \triangleright S'_1 ; S_2$$

$$4. \quad RD \vdash S_2 \triangleright S'_2 \Leftrightarrow$$

$$RD \vdash S_1 ; S_2 \triangleright S_1 ; S'_2$$

- These rules say that the transformation of a sub-statement (here a sequential statement) can be extended to the whole statement.

# Composition rules

---

5.  $RD \vdash S_1 \triangleright S'_1 \Leftrightarrow$

$RD \vdash \text{if } [b]^v \text{ then } S_1 \text{ else } S_2 \triangleright$

$\text{if } [b]^v \text{ then } S'_1 \text{ else } S_2$

6.  $RD \vdash S_2 \triangleright S'_2 \Leftrightarrow$

$RD \vdash \text{if } [b]^v \text{ then } S_1 \text{ else } S_2 \triangleright$

$\text{if } [b]^v \text{ then } S_1 \text{ else } S'_2$

7.  $RD \vdash S \triangleright S' \Leftrightarrow$

$RD \vdash \text{while } [b]^v \text{ do } S \triangleright$

$\text{while } [b]^v \text{ do } S'$

# Example

---

- Consider the program:  
     $[x:=10]^1; [y:=x+10]^2; [z:=y+10]^3;$
- The minimal solution of the Reaching Definition Analysis is:
- $RD_{in}(1) = \{(x, ?), (y, ?), (z, ?)\}$   
     $RD_{in}(2) = \{(x, 1), (y, ?), (z, ?)\}$   
     $RD_{in}(3) = \{(x, 1), (y, 2), (z, ?)\}$

- Using RD, we may start applying the rules above:

- $RD \vdash [x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$   
 $\triangleright [x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$

- Here we apply Rule 1, with  $a=(x+10)$   
 $RD_{in}(2) = \{(x,1),(y,?),(z,?)\}$

$$RD \vdash [y := a]^2 \triangleright [y := a[x \rightarrow 10]]^2$$

if  $x \in FV(a) \wedge (x,?) \notin RD_{in}(2) \wedge$   
 for every  $(z,\mu) \in RD_{in}(2): (z=x \Rightarrow [ \dots ]^\mu \text{ is } [x:=10]^\mu)$

- $RD \vdash [x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$ 
  - ▷  $[x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$
  - ▷  $[x:=10]^1; [y:=20]^2; [z:=y+10]^3$

- Here we apply Rule 2, with expression  $a=(10+10)$

$$RD \vdash [y := a]^2 \triangleright [y := n]^2$$

if  $FV(a)=\emptyset \wedge a \notin \text{Num} \wedge$  the value of expression  $a$  is  $n$

- $RD \vdash [x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$ 
  - ▷  $[x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$
  - ▷  $[x:=10]^1; [y:=20]^2; [z:=y+10]^3$
  - ▷  $[x:=10]^1; [y:=20]^2; [z:=20+10]^3$

- Here we apply again Rule 1, with  $a=(y+10)$

$$RD \vdash [z := a]^3 \triangleright [z := a[y \rightarrow 20]]^3$$

if  $y \in FV(a) \wedge (y, ?) \notin RD_{in}(3) \wedge$   
 for every  $(w, \mu) \in RD_{in}(3): (w=y \Leftrightarrow [ \dots ]^\mu \text{ is } [y:=20]^\mu)$

- $\text{RD} \vdash [x:=10]^1; [y:=x+10]^2; [z:=y+10]^3$ 
  - ▷  $[x:=10]^1; [y:=10+10]^2; [z:=y+10]^3$
  - ▷  $[x:=10]^1; [y:=20]^2; [z:=y+10]^3$
  - ▷  $[x:=10]^1; [y:=20]^2; [z:=20+10]^3$
  - ▷  $[x:=10]^1; [y:=20]^2; [z:=30]^3$

- Here we apply again Rule 2 with  $a=(20+10)$

$$\text{RD} \vdash [z := a]^3 \triangleright [z := n]^3$$

if  $\text{FV}(a)=\emptyset \wedge a \notin \text{Num} \wedge$  the value of expression  $a$  is  $n$

- 
- The example above show how to get a sequence of transformations

$$RD \models S_1 \triangleright S_2 \triangleright S_3 \triangleright \dots \triangleright S_k$$

- Theoretically, once computed  $S_2$  we should re-execute a reaching Definition Analysis to the new program.
- However, if  $RD$  is a solution of the Reaching Def. Analysis for  $S_i$  and  $RD \models S_i \triangleright S_{i+1}$ , then it is easy to see that  $RD$  is also a solution of the Reaching Def. Analysis for  $S_{i+1}$ .  
In fact the transformation applies to elements that do not affect at all the Reaching Def. Analysis.