

**Sorin Lerner, David Grove, and Craig Chambers**

**“Composing Dataflow Analyses and Transformations”**

**POPL 2002**

*Also starring:*

Cliff Click and Keith D. Cooper, “Combining Analyses, Combining Optimizations”, ACM Transactions on Programming Languages and Systems, March 1995

Craig Chambers, Jeffrey Dean, and David Grove, “Frameworks for Intra- and Interprocedural Dataflow Analysis”, University of Washington Computer Science Technical Report 96-11-02

*Summarized for 308-762 by John Jorgensen,  
19+107 November, 2001*

## Outline

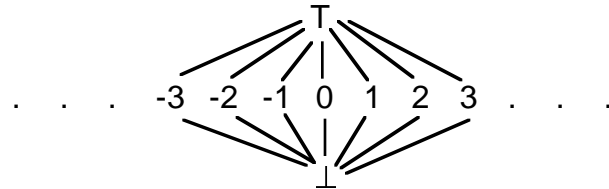
- Why combine analyses?
  - To improve precision.
  - To reduce compile time.
- Vortex framework: combine analyses while maintaining modularity
- Extended example combining three analyses
- Experimental results
- Applicability to Soot?

## Why combine analyses?

- Traditionally compilers include a series of independent dataflow analyses to discover properties which allow performance-improving transformations.
- What if the results of analysis A affect those of analysis B, and vice versa?
  - a phase ordering problem.
- Sometimes even iterating over the set of analyses cannot produce as precise a result as combining them.
- Iterating over a set of analyses probably takes longer than a combined analysis even if the result is no less precise.

## Example simple analysis: Constant Propagation

1. Approximating variables with constant values.



2. Variable  $v$  has constant value  $c$  at program point  $p$  if on all paths to  $p$ ,  $v$  has been defined as  $c$ .

3. Forwards analysis.

|         | $\perp$ | $c0$            | $T$ |
|---------|---------|-----------------|-----|
| $\perp$ | $\perp$ | $\perp$         | $T$ |
| $c1$    | $\perp$ | $(c0==c1)?c0:T$ | $T$ |
| $T$     | $T$     | $T$             | $T$ |

4. Confluence operator:

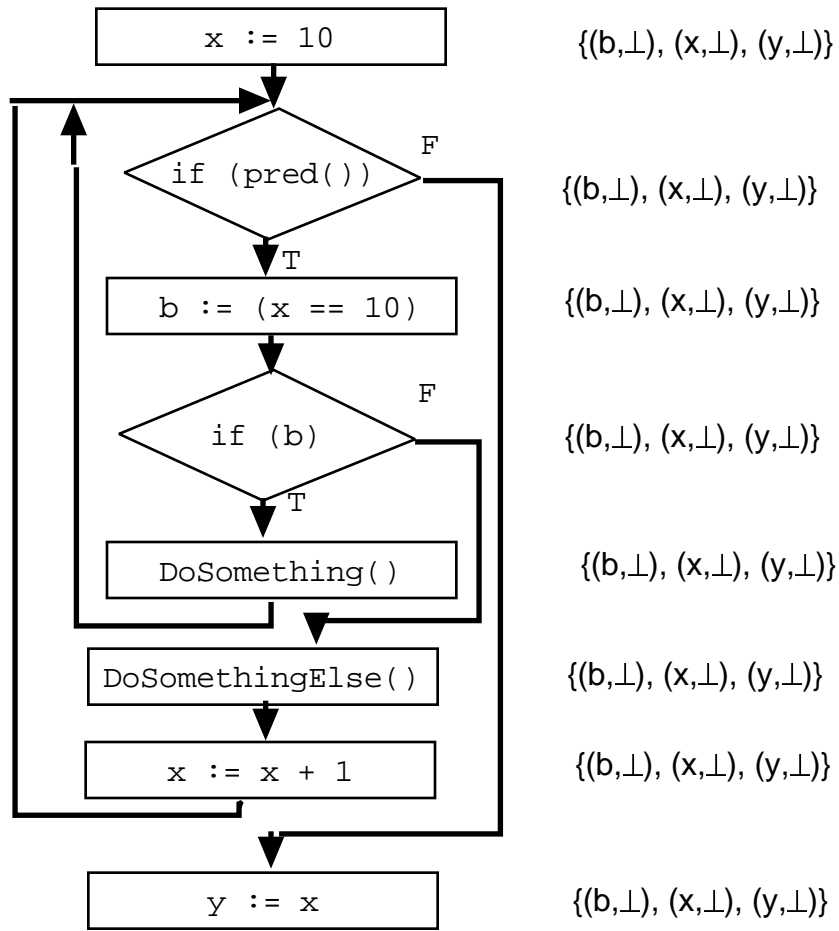
| op      | $\perp$ | $c0$                | $T$     |
|---------|---------|---------------------|---------|
| $\perp$ | $\perp$ | $\perp$             | $\perp$ |
| $c1$    | $\perp$ | $c0 \text{ op } c1$ | $T$     |
| $T$     | $\perp$ | $T$                 | $T$     |

5. Equation (for  $c2 := c0 \text{ op } c1$ ):

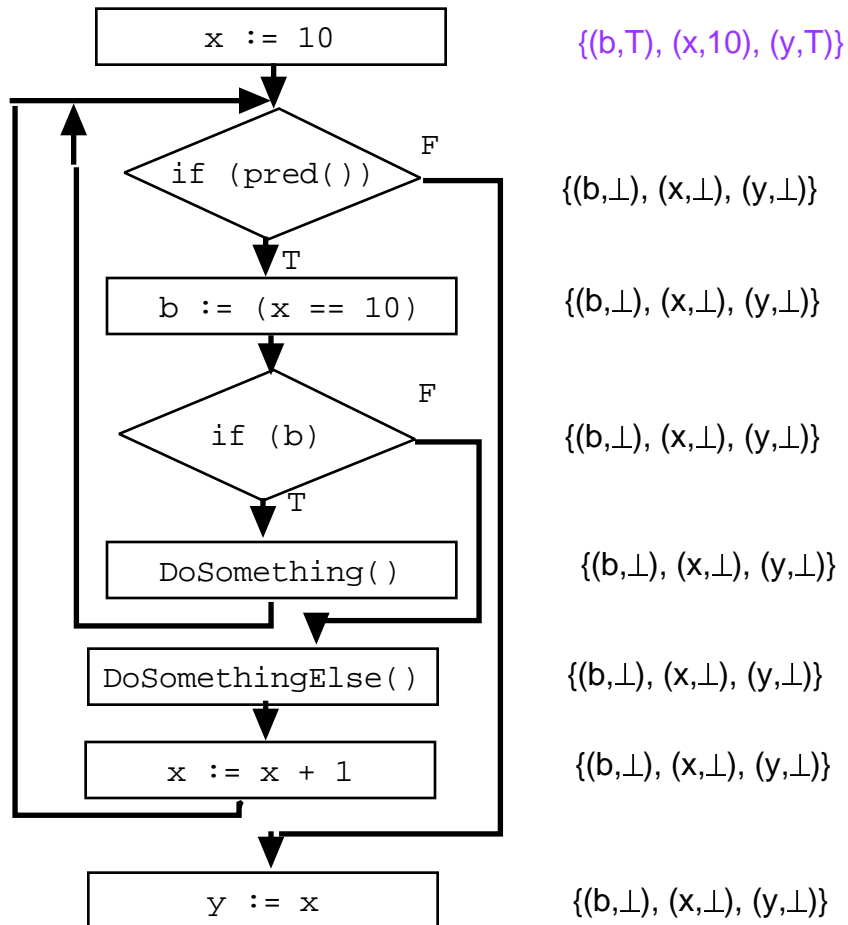
6. Initial conditions:

- $\text{out}(\text{start}) = \{(v0, T), (v1, T), \dots\}$
- $\text{out}(S_n) = \{(v0, \perp), (v1, \perp), \dots\}$

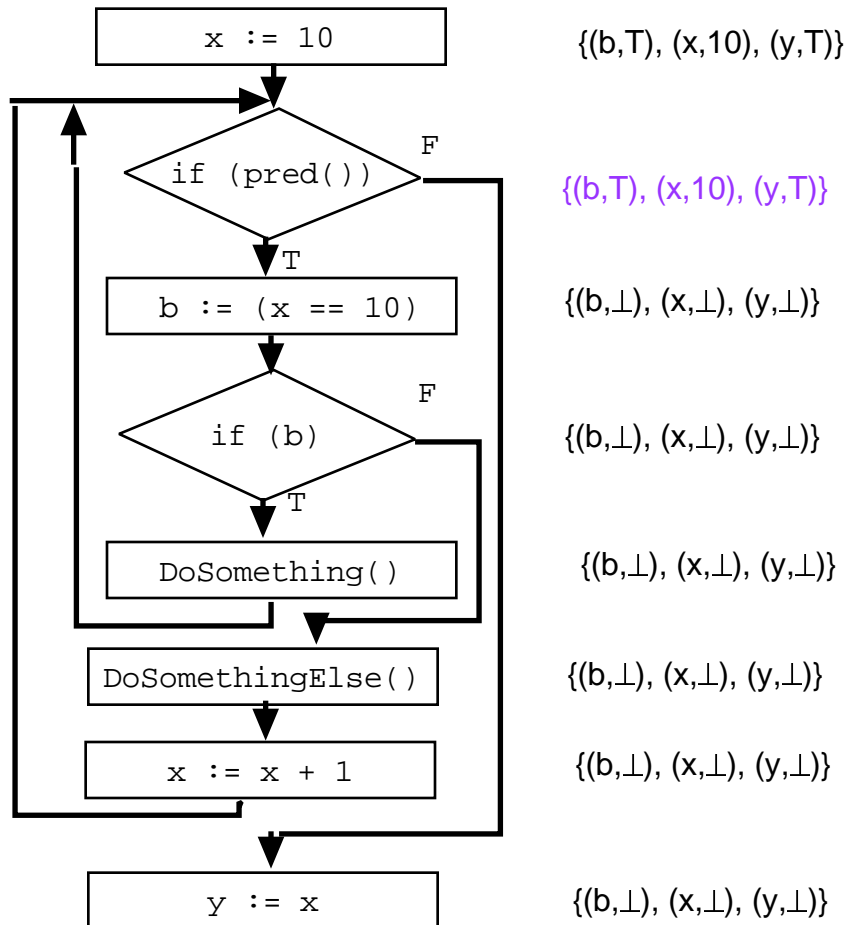
# Constant propagation example (0)



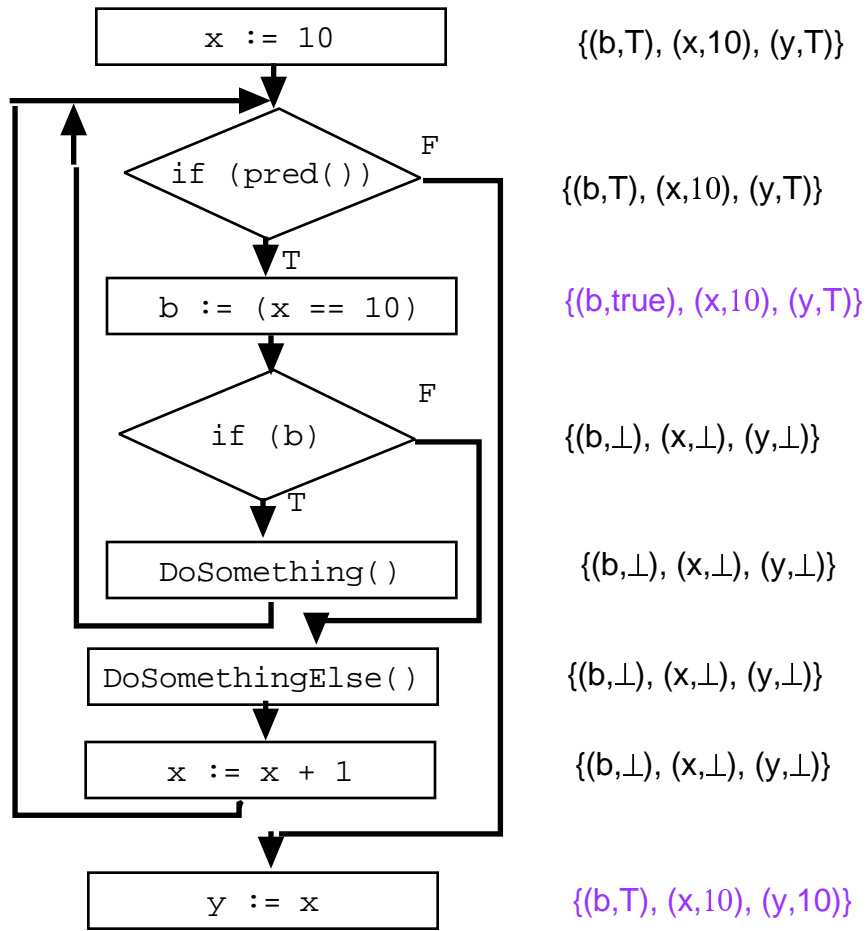
# Constant propagation example (1)



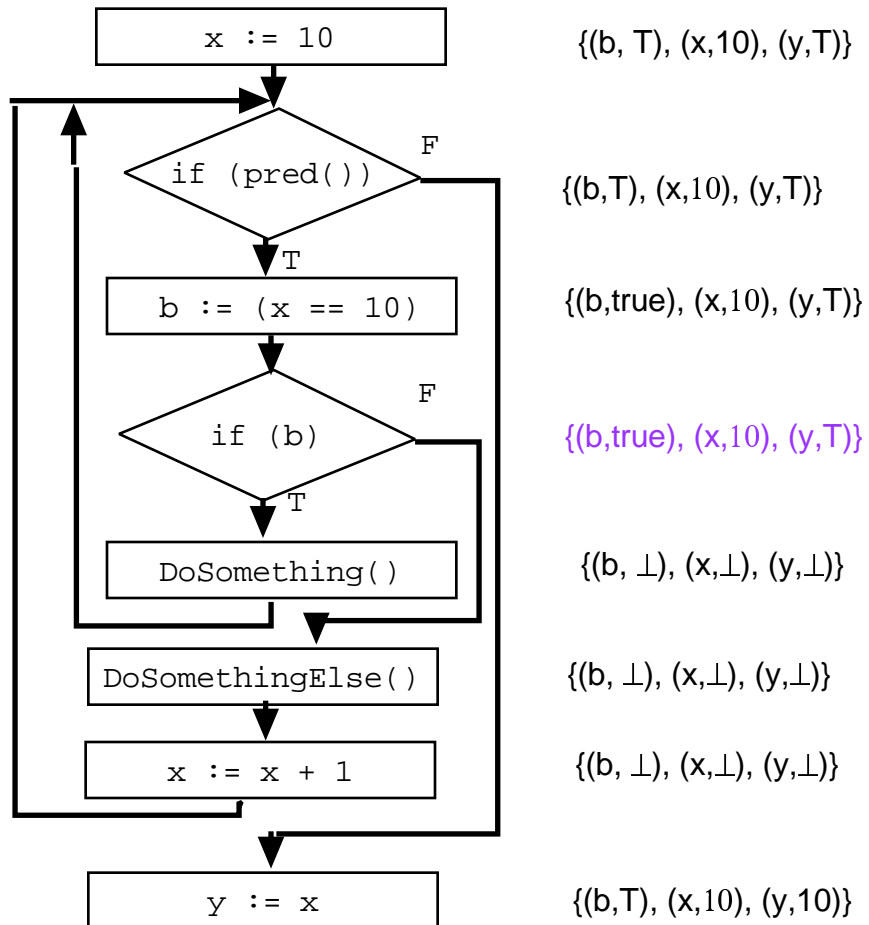
## Constant propagation example (2)



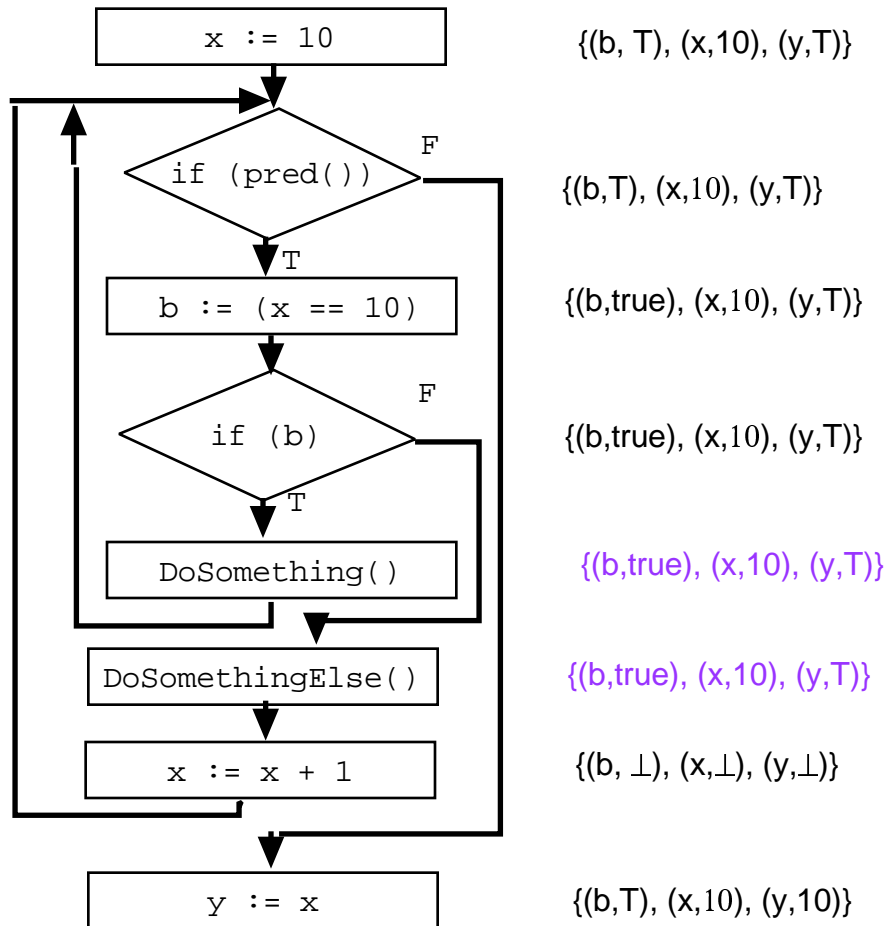
# Constant propagation example (3)



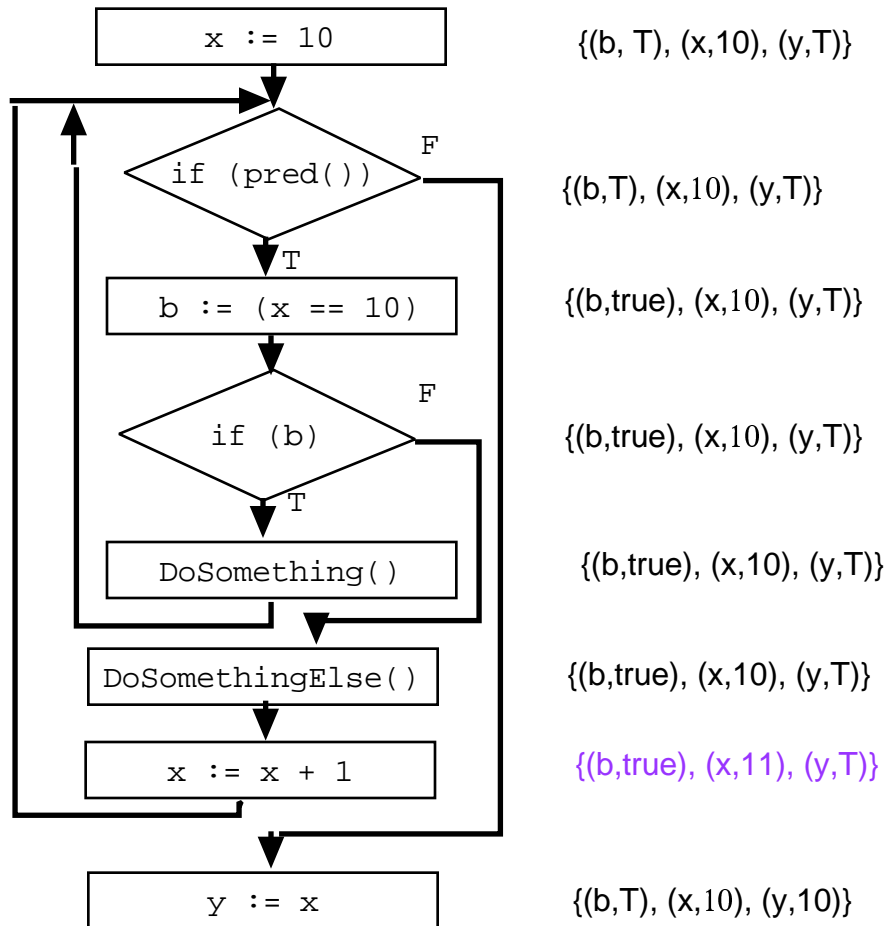
## Constant propagation example (4)



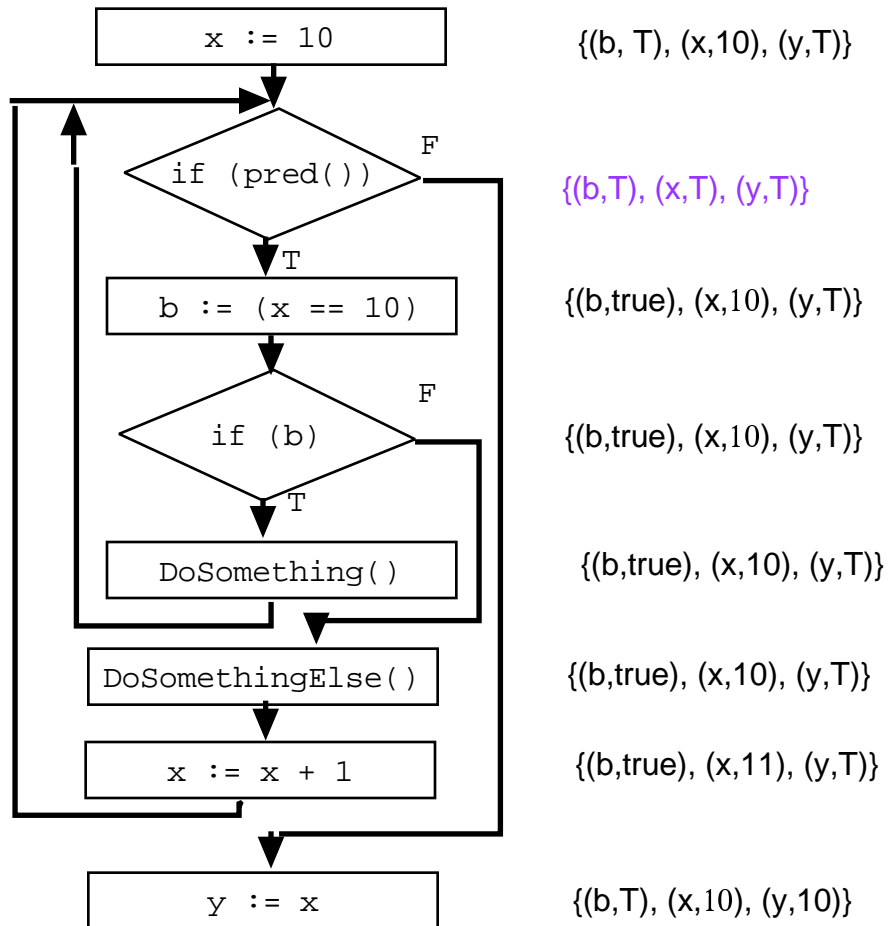
# Constant propagation example (5)



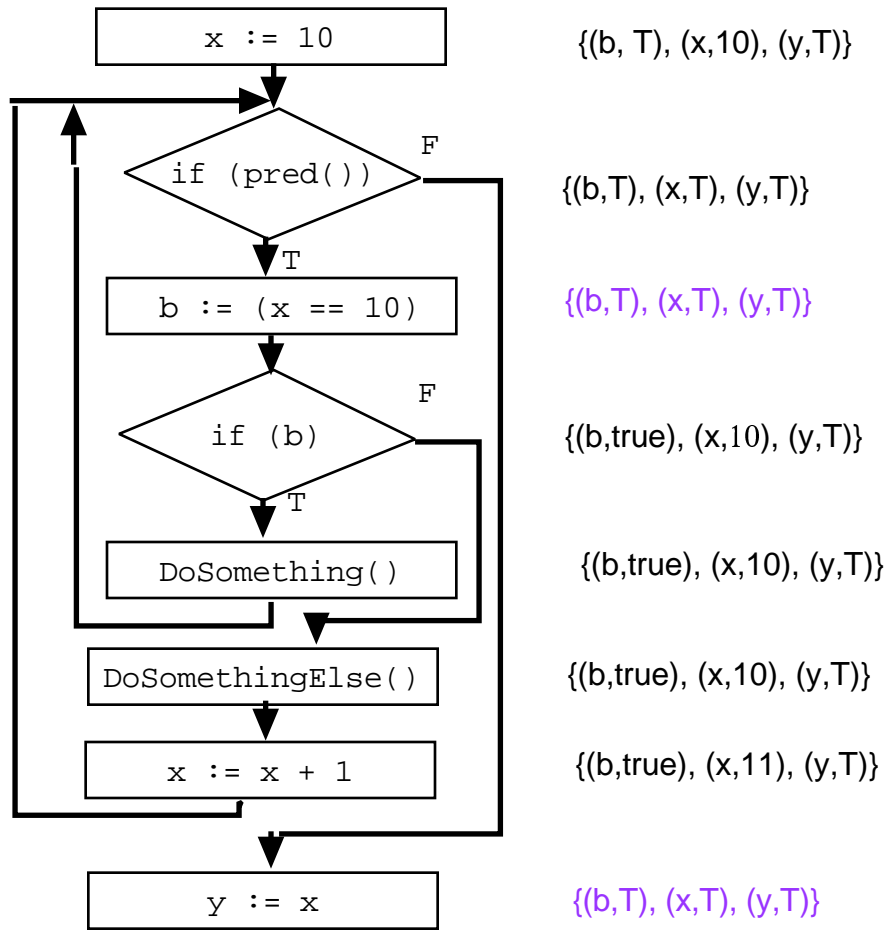
# Constant propagation example (6)



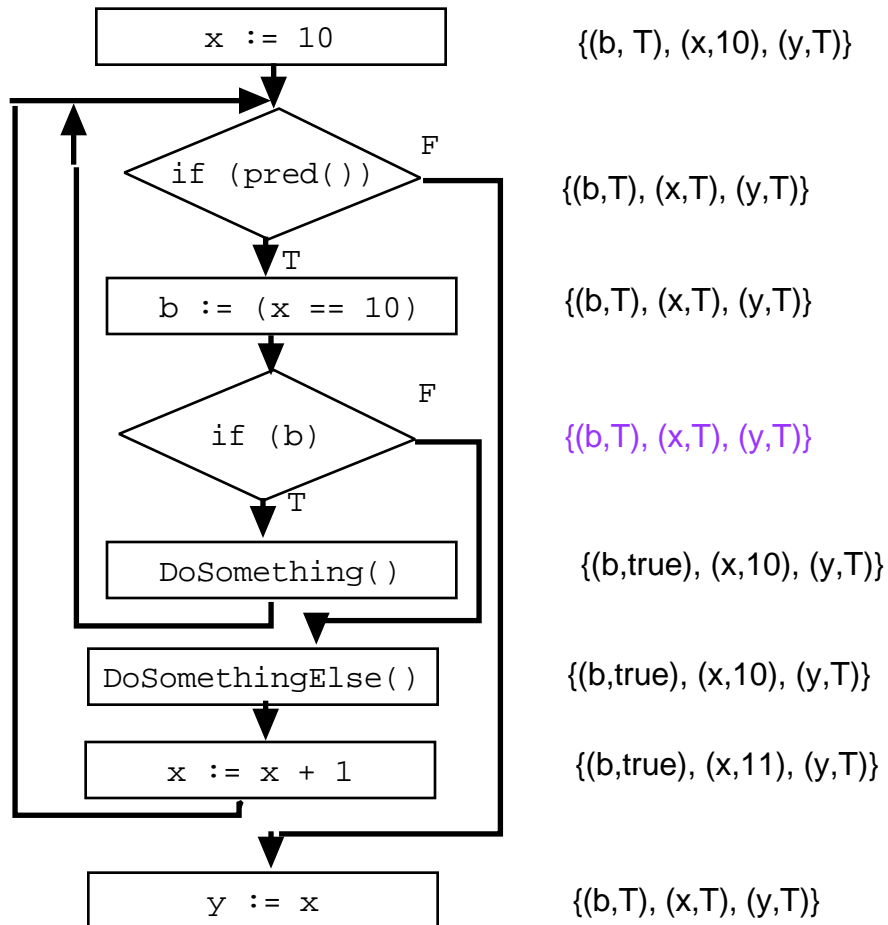
# Constant propagation example (7)



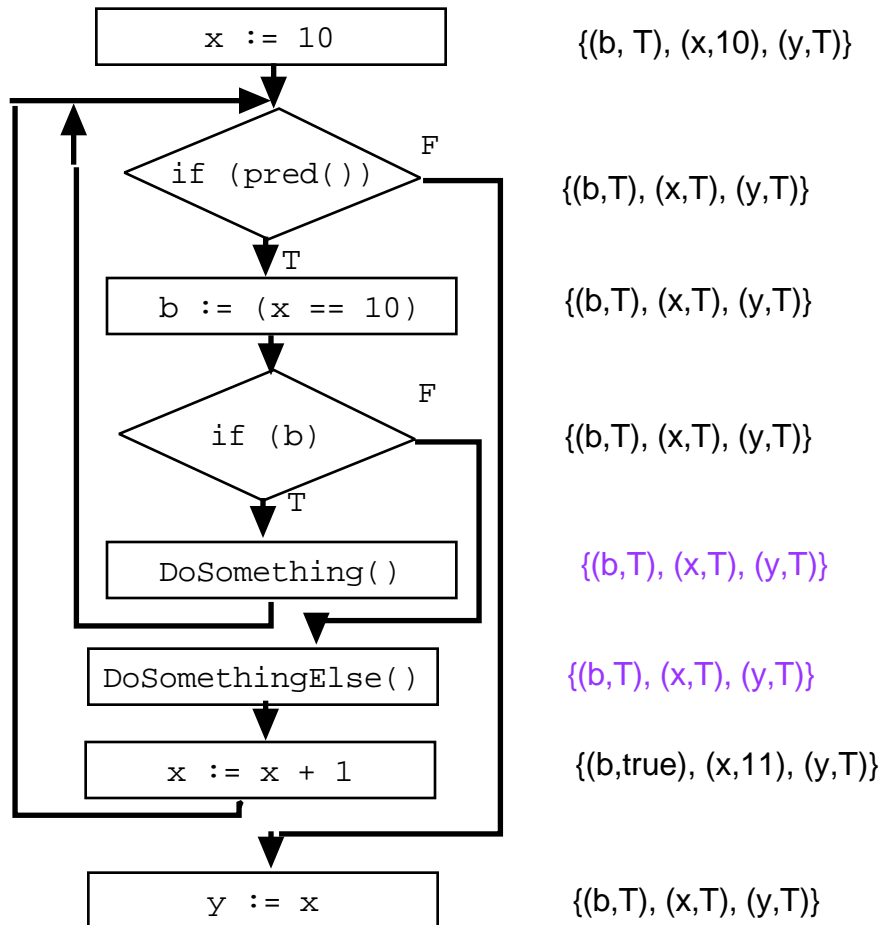
# Constant propagation example (8)



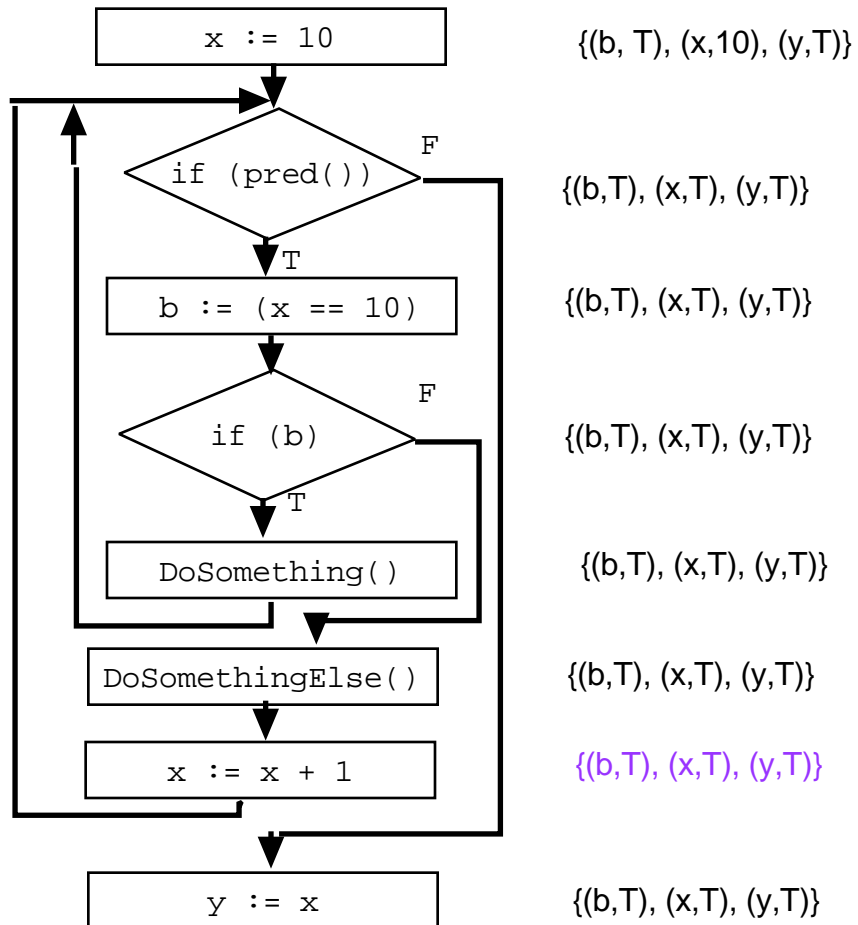
# Constant propagation example (9)



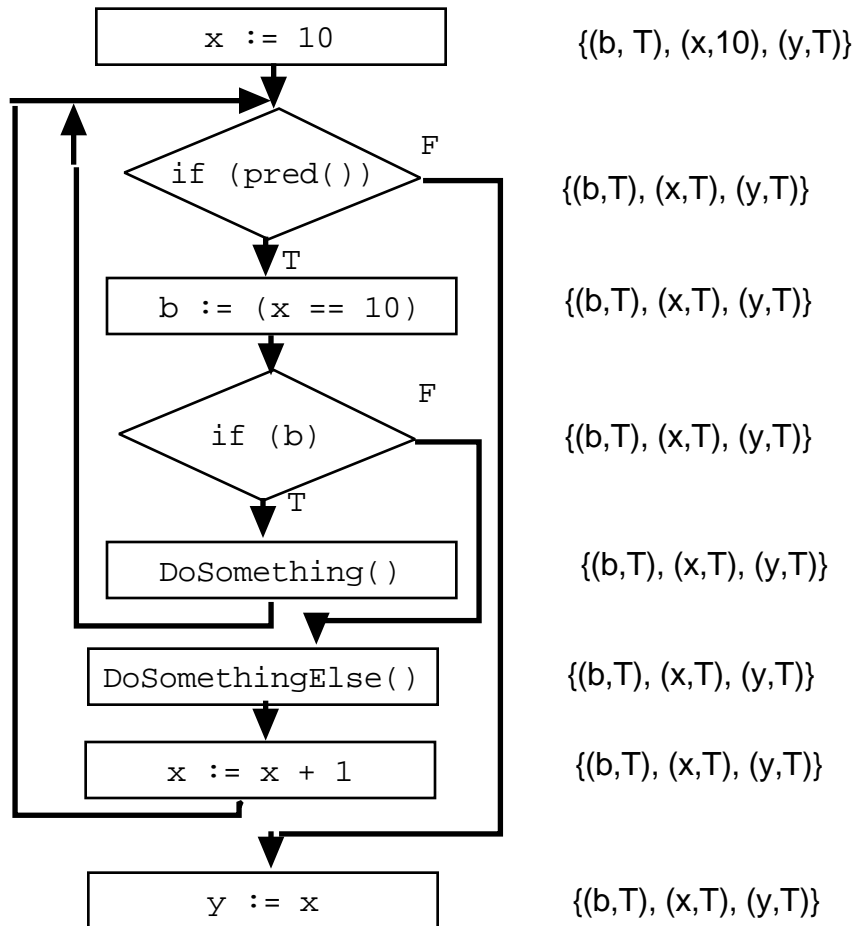
# Constant propagation example (10)



# Constant propagation example (11)



# Constant propagation example (12)



## Example simple analysis: Reachability

1. Approximating whether each statement is reachable

$$\begin{array}{c} R \\ | \\ U \end{array}$$

2. Statement  $s$  is reachable if some path from the start node to  $s$  may be executed.

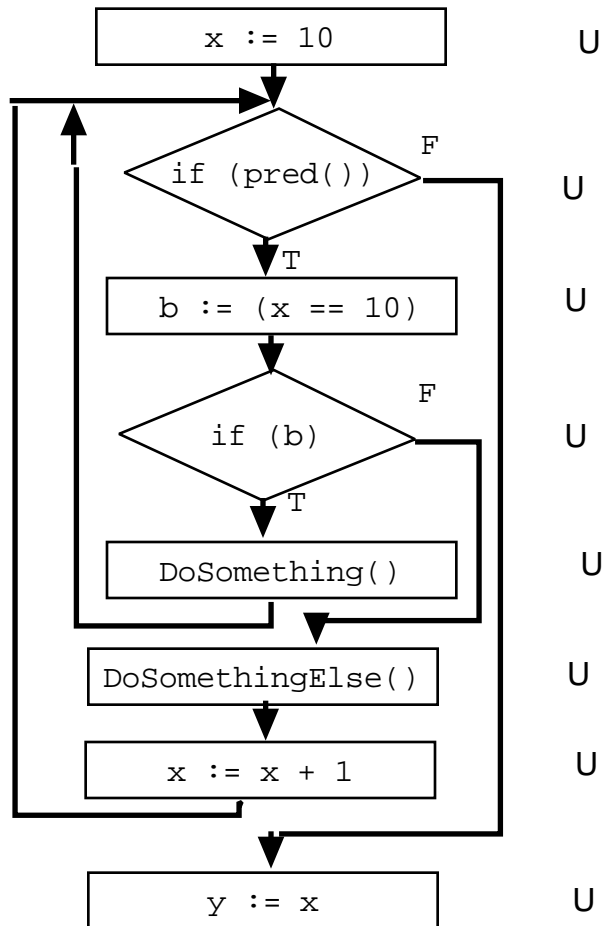
3. Forwards analysis.

4. Confluence operator:  $\bigwedge_{p \in \text{pred}(s)} \left\{ \begin{array}{l} U \text{ if } p \text{ is } U \\ U \text{ if } p \text{ is if false then} \\ U \text{ if } p \text{ is if true else} \\ R \text{ otherwise} \end{array} \right.$

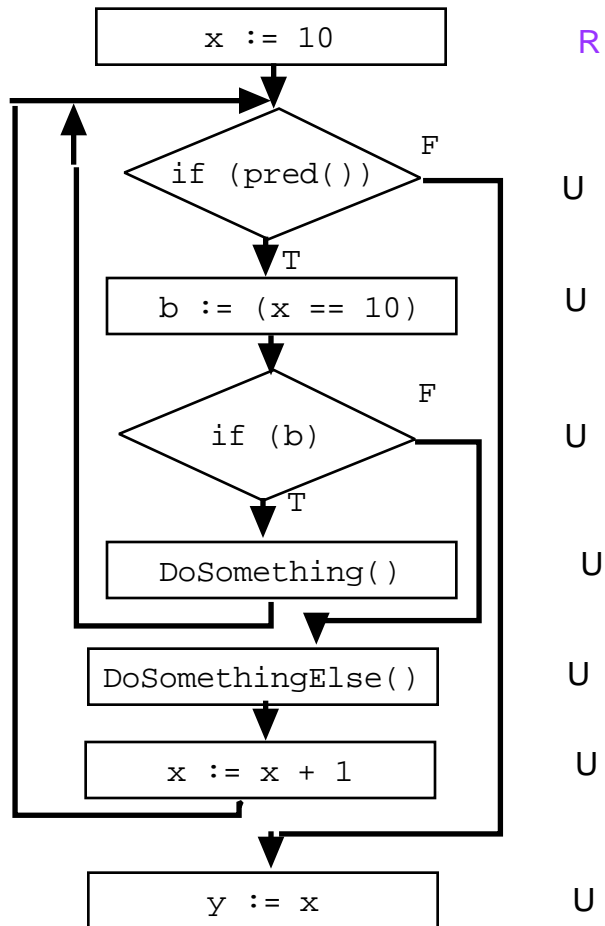
5. Equations: incorporated in confluence operator

6. Initial conditions: Start  $R$ , all others  $U$

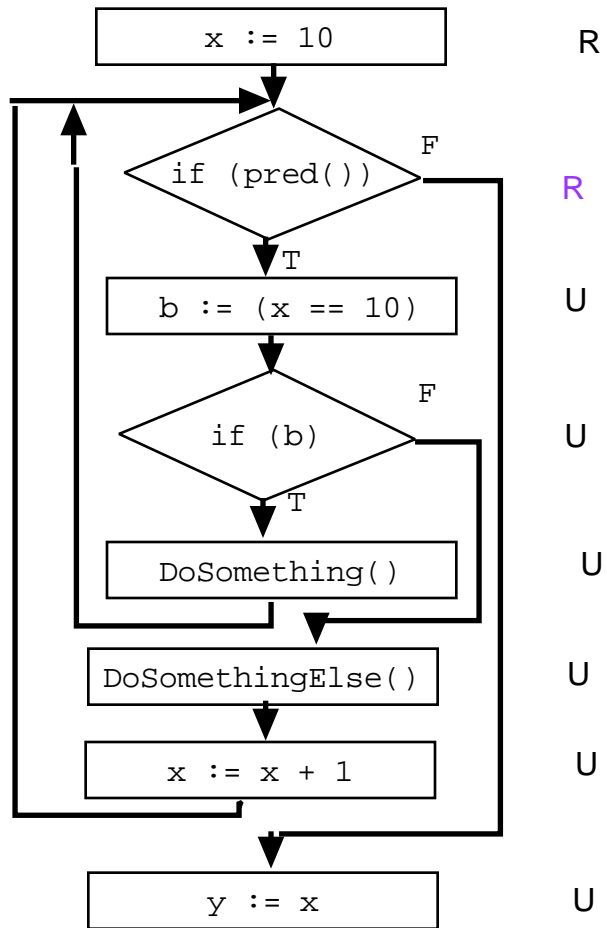
# Reachability example (0)



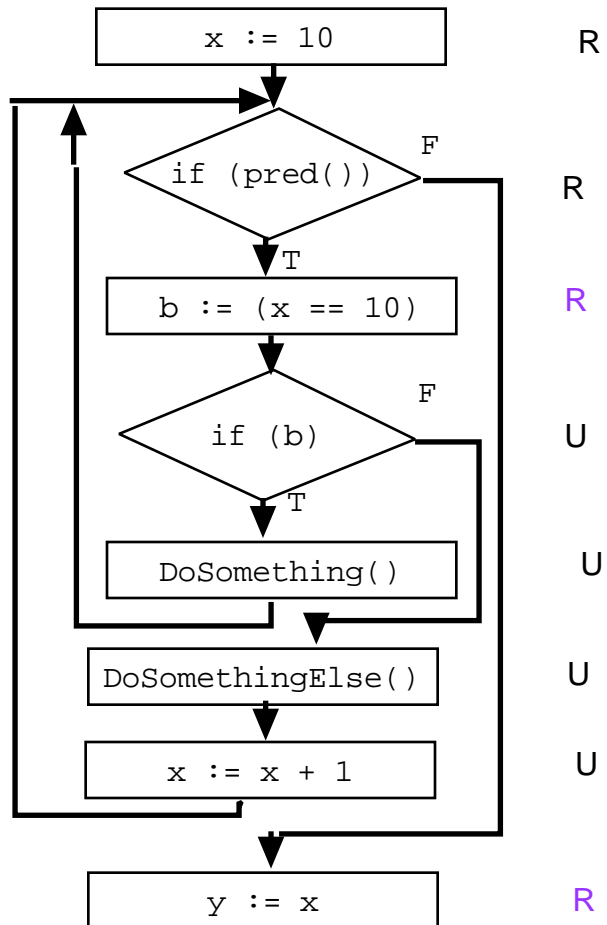
# Reachability example (1)



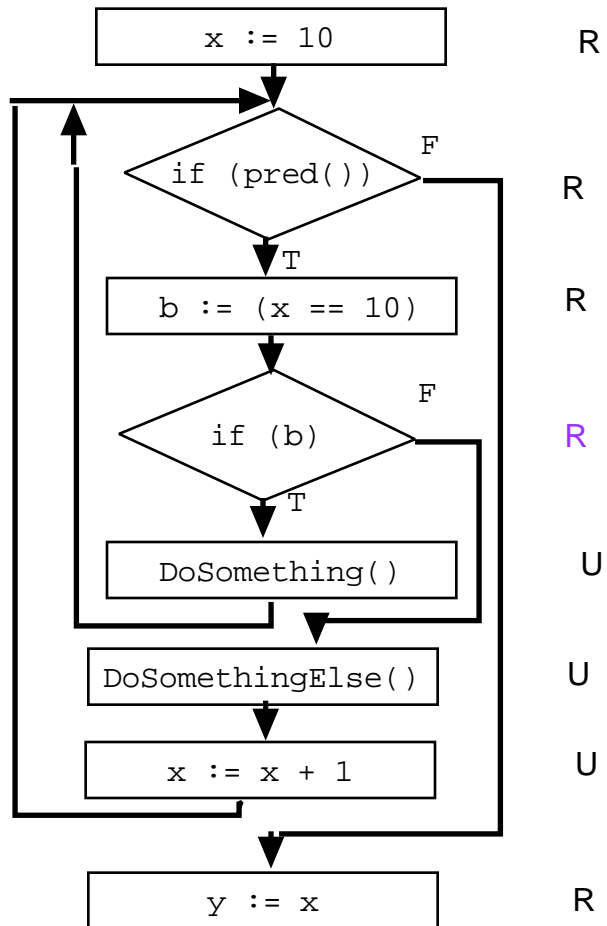
## Reachability example (2)



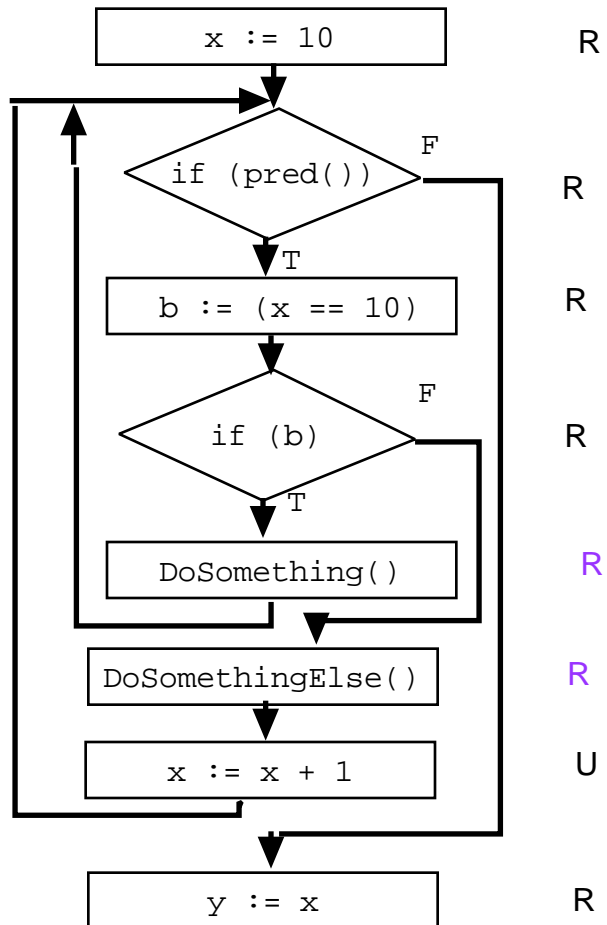
## Reachability example (3)



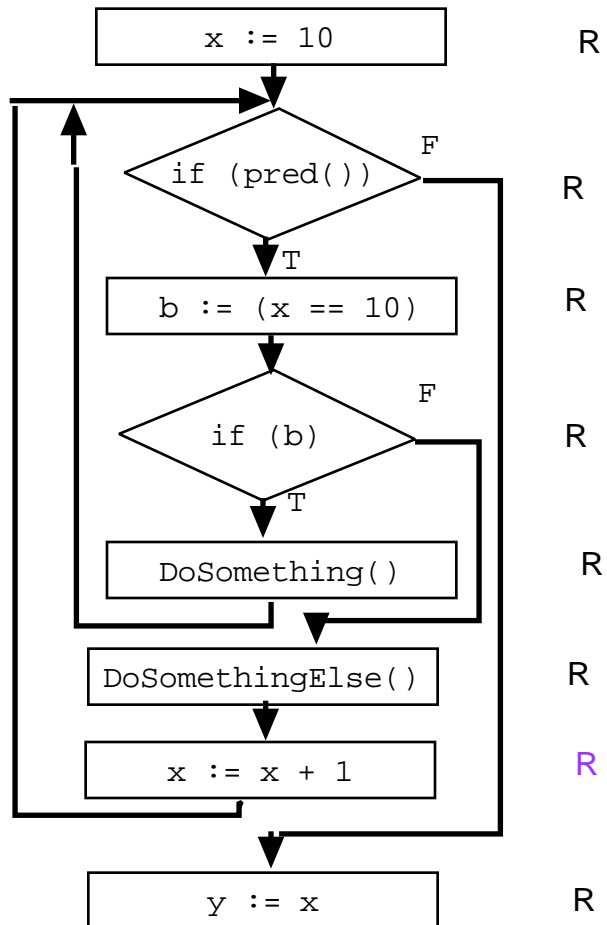
## Reachability example (4)



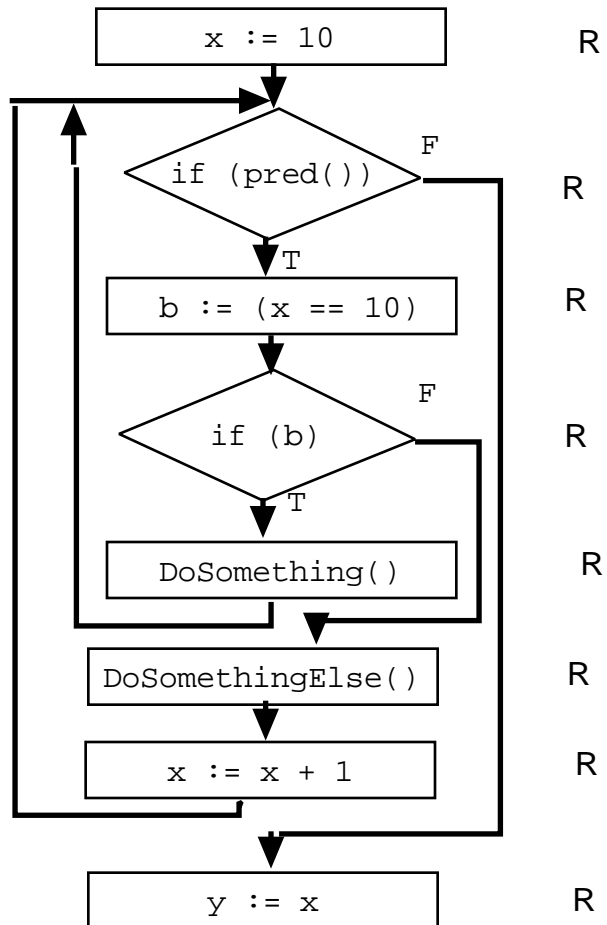
## Reachability example (5)



## Reachability example (6)

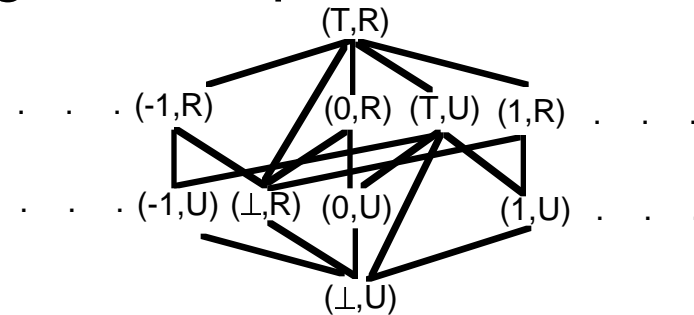


## Reachability example (7)



# Combining constant propagation with reachability

1. Approximating ordered pairs of what we approximated before:

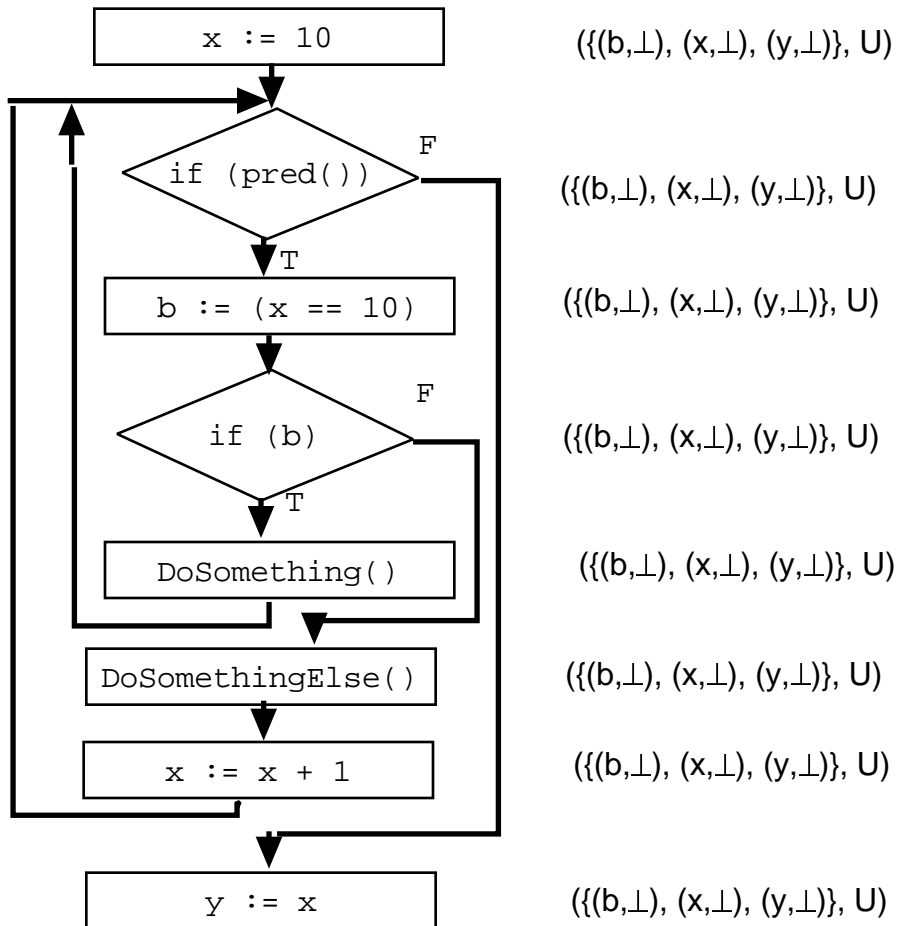


- 4,5. Confluence operator and equations for (c,r):

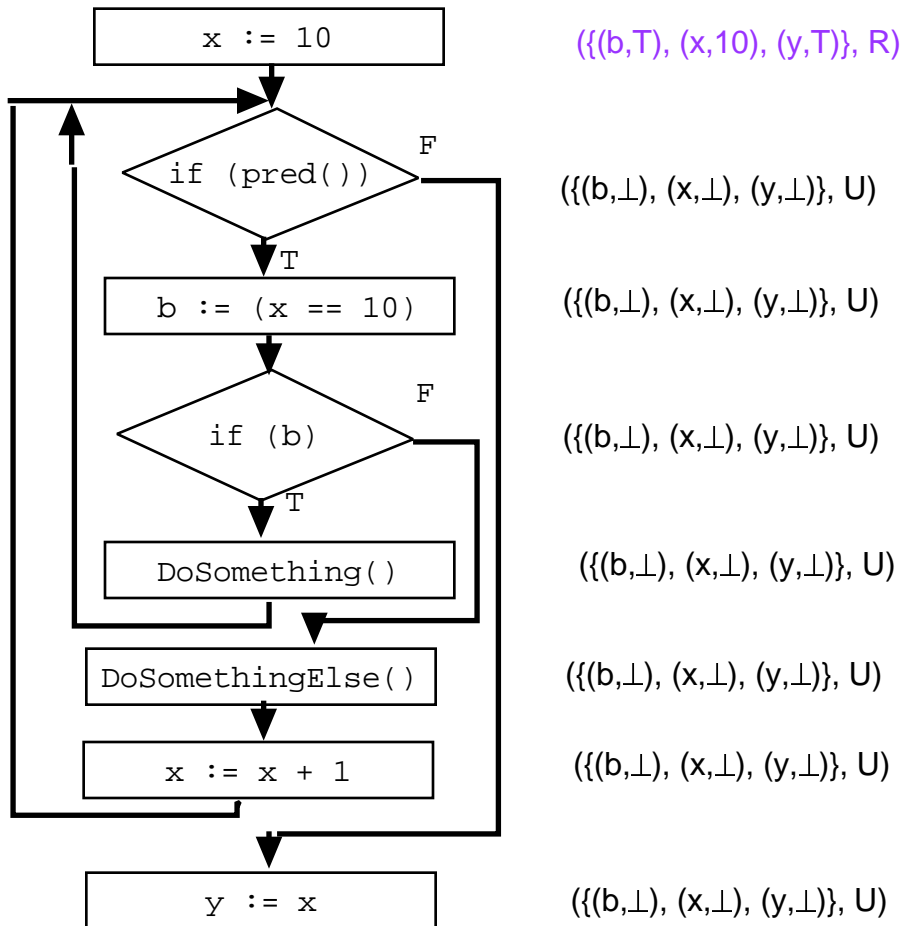
$$c = \begin{cases} \text{simple } c \text{ if } s \text{ is } R \\ \perp \text{ if } s \text{ is } U \end{cases}$$

$$r = \bigwedge_{p \in \text{pred}(s)} \begin{cases} U \text{ if } p \text{ is } U \\ U \text{ if } p \text{ is if } b \text{ then } \wedge \sigma(b) = \text{false} \\ U \text{ if } p \text{ is if } b \text{ else } \wedge \sigma(b) = \text{true} \\ R \text{ otherwise} \end{cases}$$

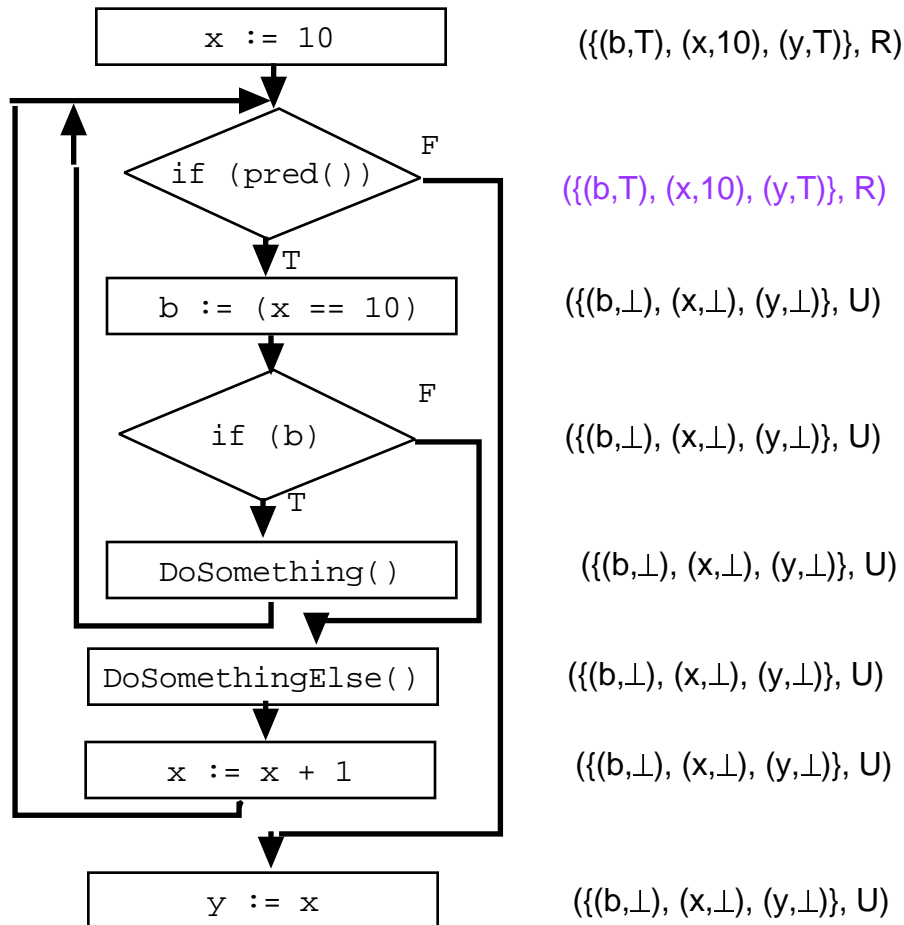
# Combined example (0)



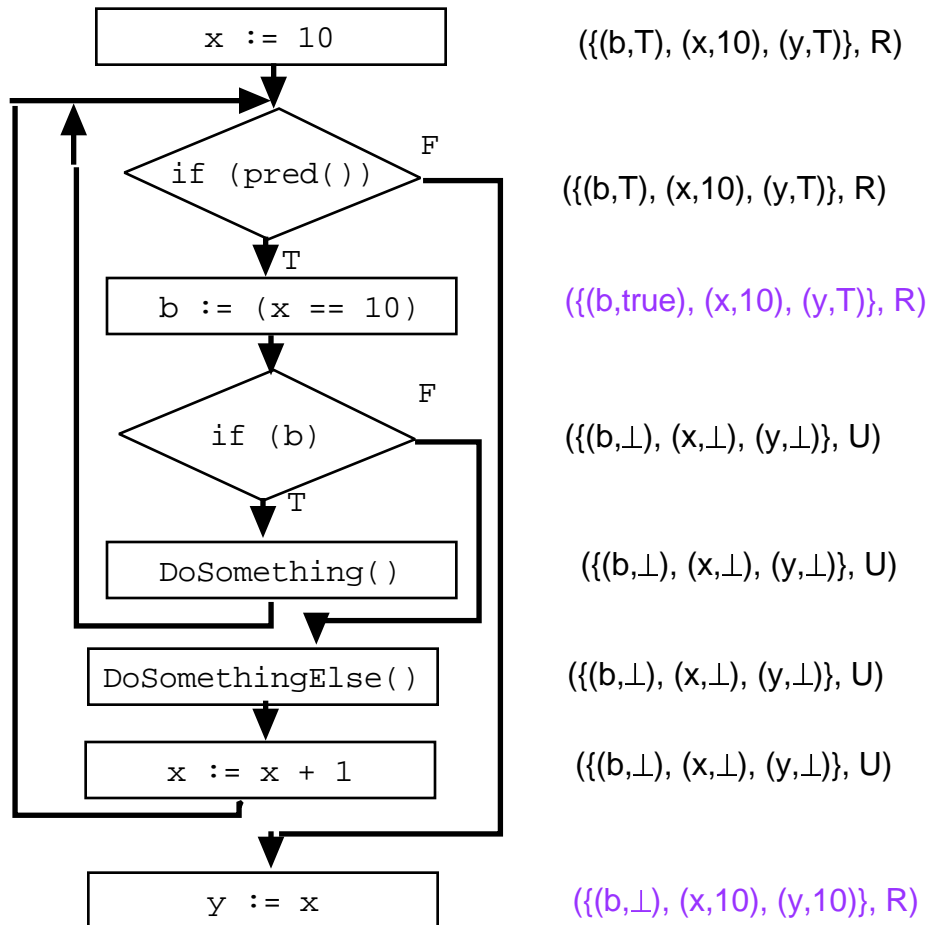
# Combined example (1)



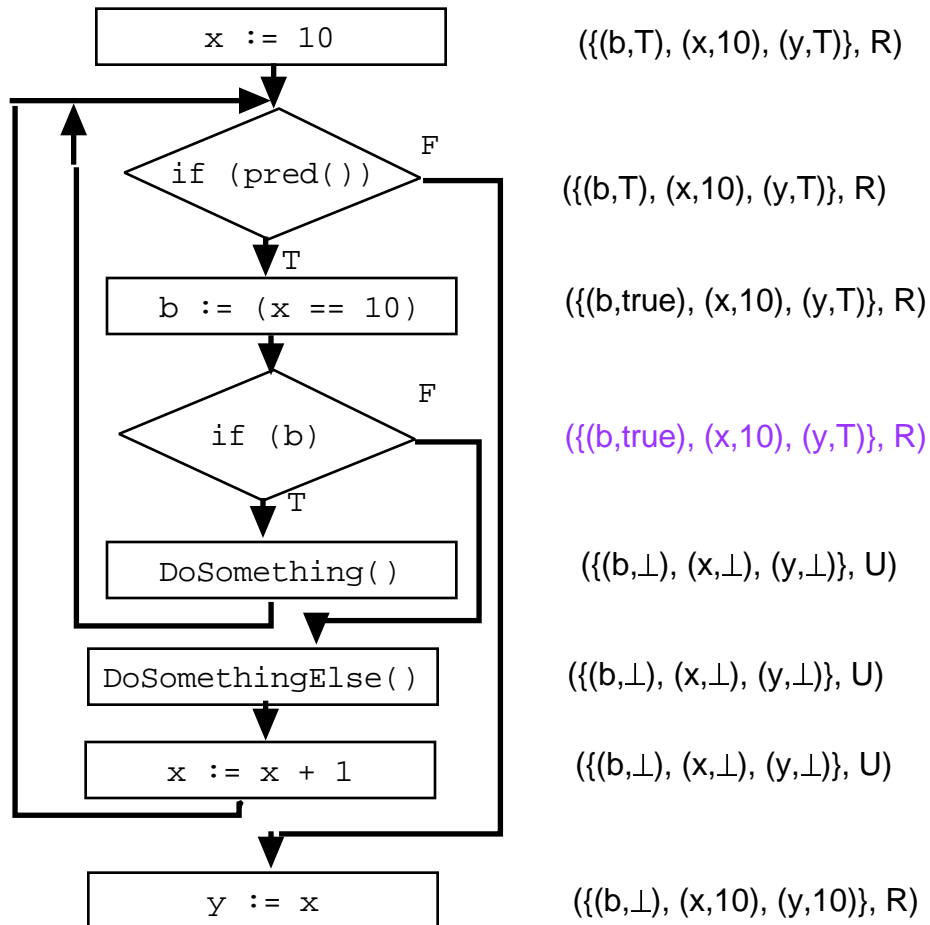
## Combined example (2)



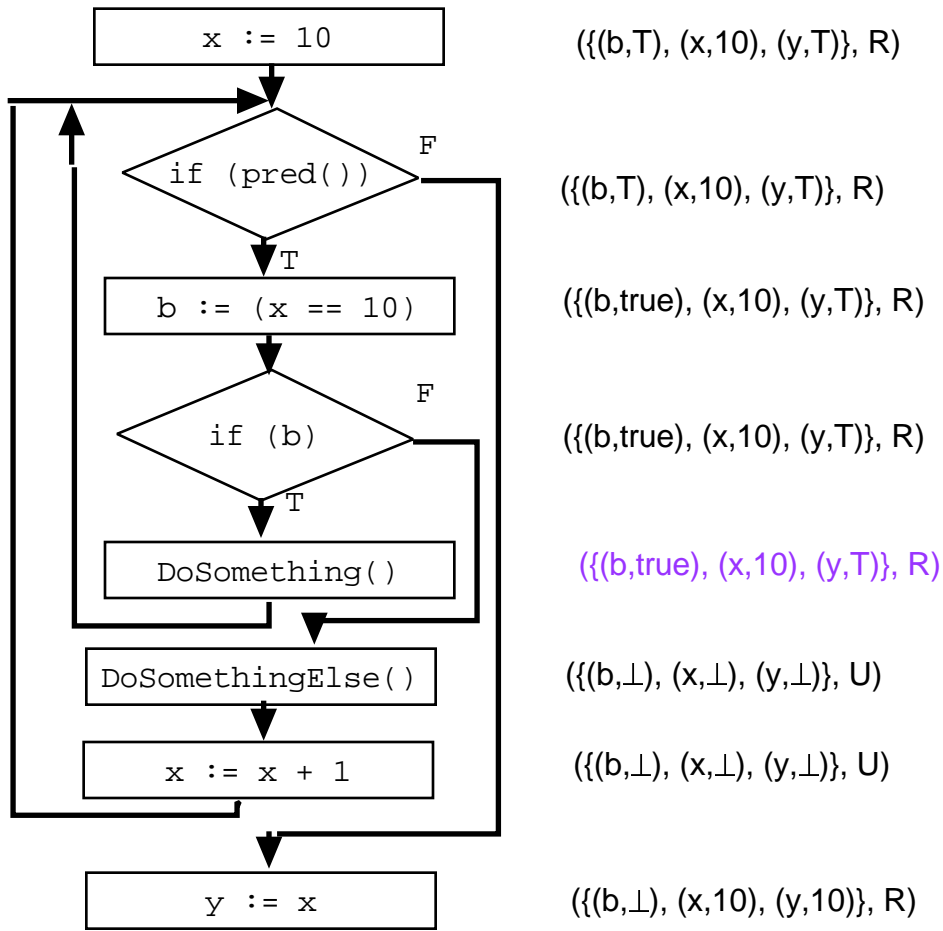
## Combined example (3)



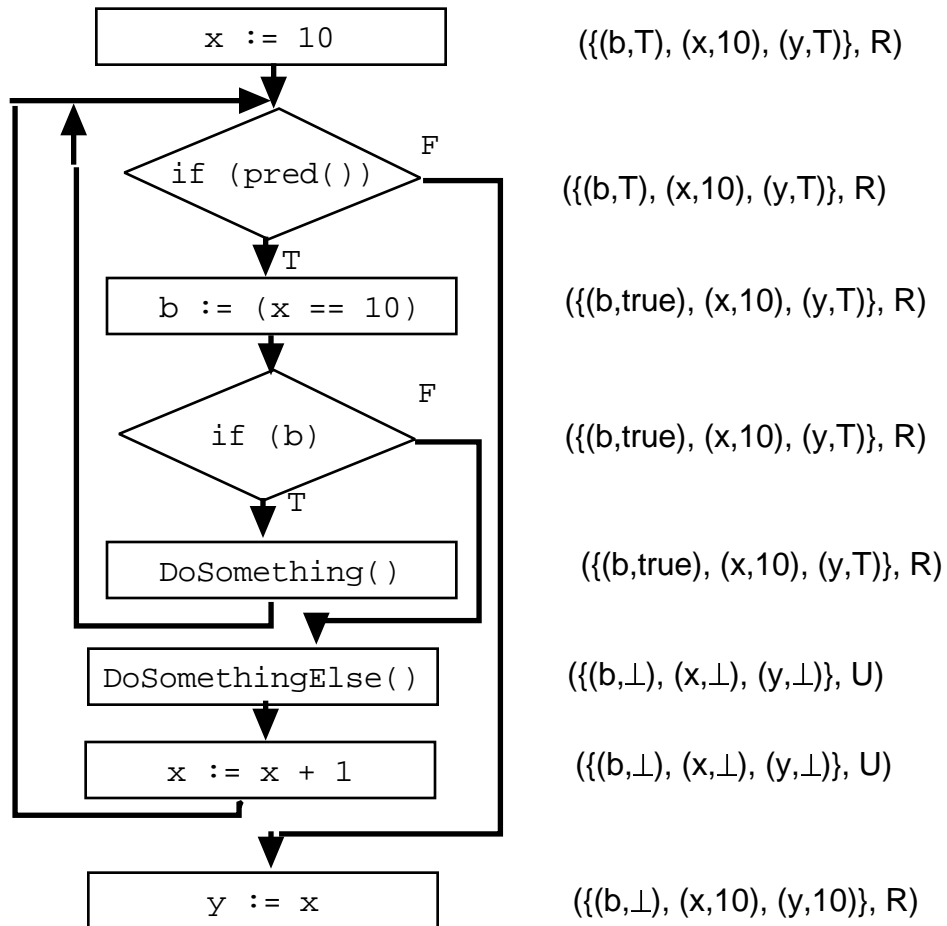
## Combined example (4)



# Combined example (5)



## Combined example (6)



## **Vortex framework: combines modular analyses**

- Combined constant propagation and reachability analysis provides more precise results, but requires explicit specification of the “superanalysis”.
- Prefer to compose separately-specified analyses without having to provide the glue manually.
  - BUT, theory of abstract interpretation shows that combined analysis domains provide better results only with the addition of new flow functions [Cousot&Cousot,1979].
- Lerner et. al.’s answer: let different analyses communicate by tentatively replacing nodes in the control flow graph while it is being analyzed.

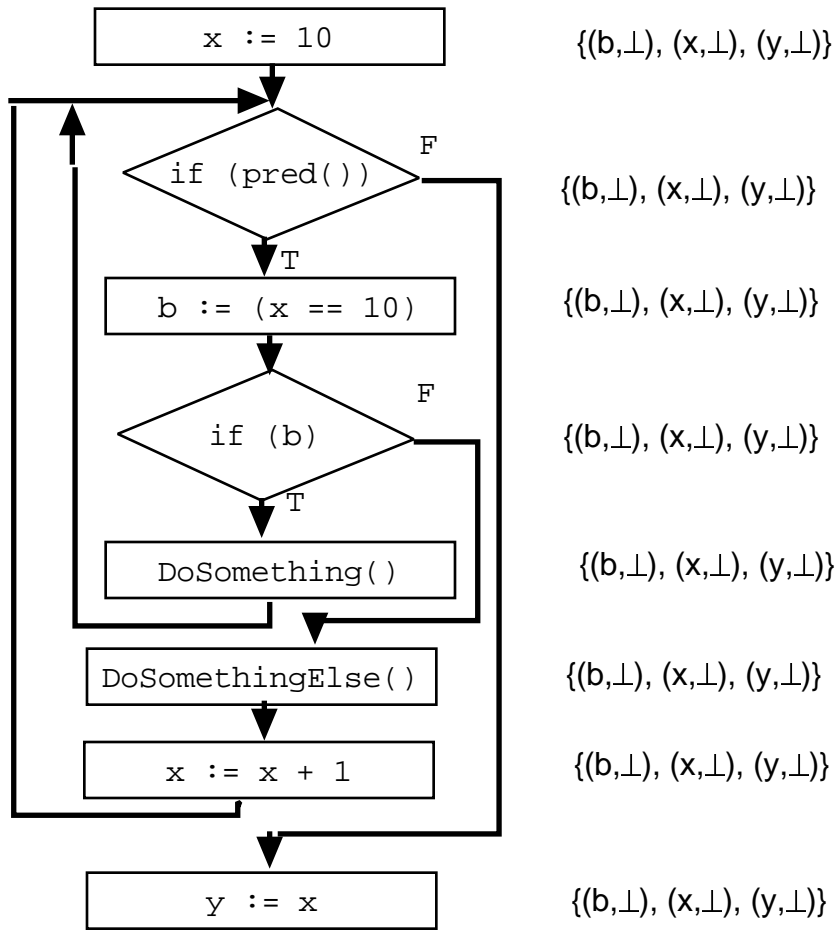
## Flow functions with replacement graphs

- Flow functions ( $f: Node \times D^* \rightarrow D^* \cup Graph$ ) return either:
  - dataflow values to propagate to the next node, or
  - a subgraph that replaces the current node during the computation of dataflow values to propagate:
    - Perform iterative flow analysis on the subgraph, initialized with the node's input values (with recursive replacement possible).
    - When fixed point reached for subgraph, use results as the node's output values.
    - Discard the replacement graph (well, cache it, because ...).
- When fixed point is reached for the entire cfg, use the last set of replacement graphs as the overall analysis's transformation.

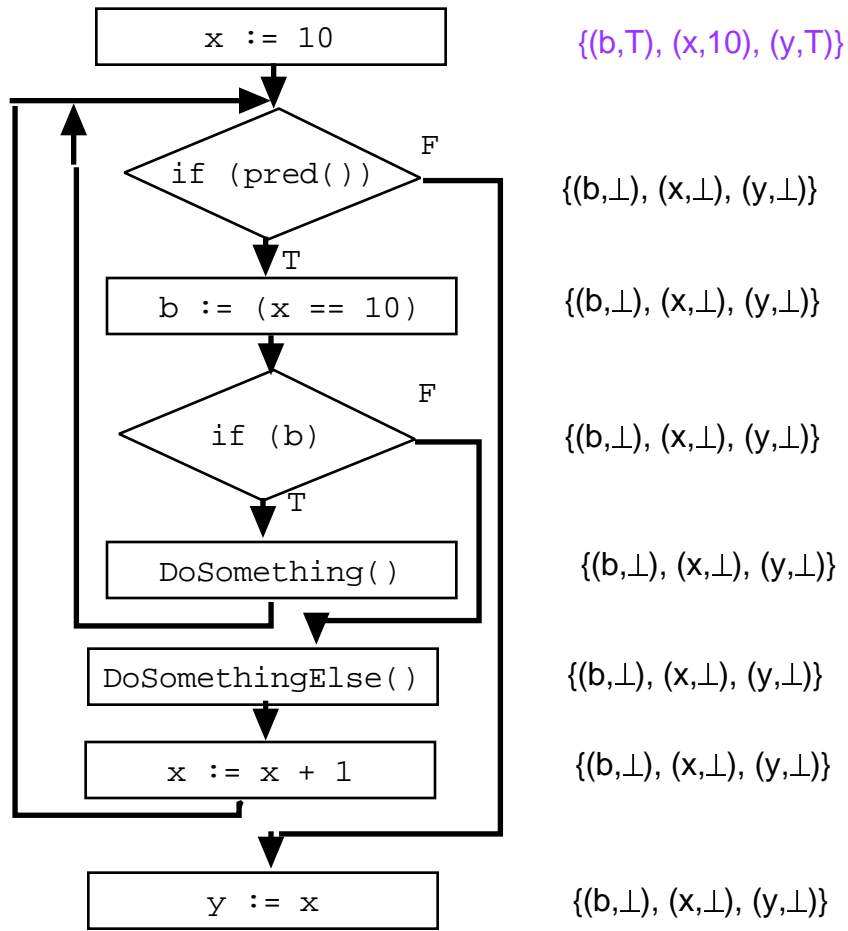
## Flow functions for constant propagation/folding

- $F\langle x := k \rangle \sigma = PROPAGATE(\sigma[x \mapsto k])$
- $F\langle x := y \text{ op } z \rangle \sigma = \text{let } t := \sigma[y] \text{ op } \sigma[z]$ 
  - if *constant*( $t$ ) then
  - $REPLACE(\langle x := t \rangle)$
  - else
  - $PROPAGATE(\sigma[x \mapsto t])$
- $F\langle \text{if } b \text{ goto } L1 \text{ else goto } L2 \rangle \sigma =$ 
  - if *constant*( $\sigma[b]$ ) then
  - if  $\sigma[b] == \text{true}$  then
  - $REPLACE(\langle \text{goto } L1 \rangle)$
  - else
  - $REPLACE(\langle \text{goto } L2 \rangle)$
  - else
  - $PROPAGATE(\sigma)$

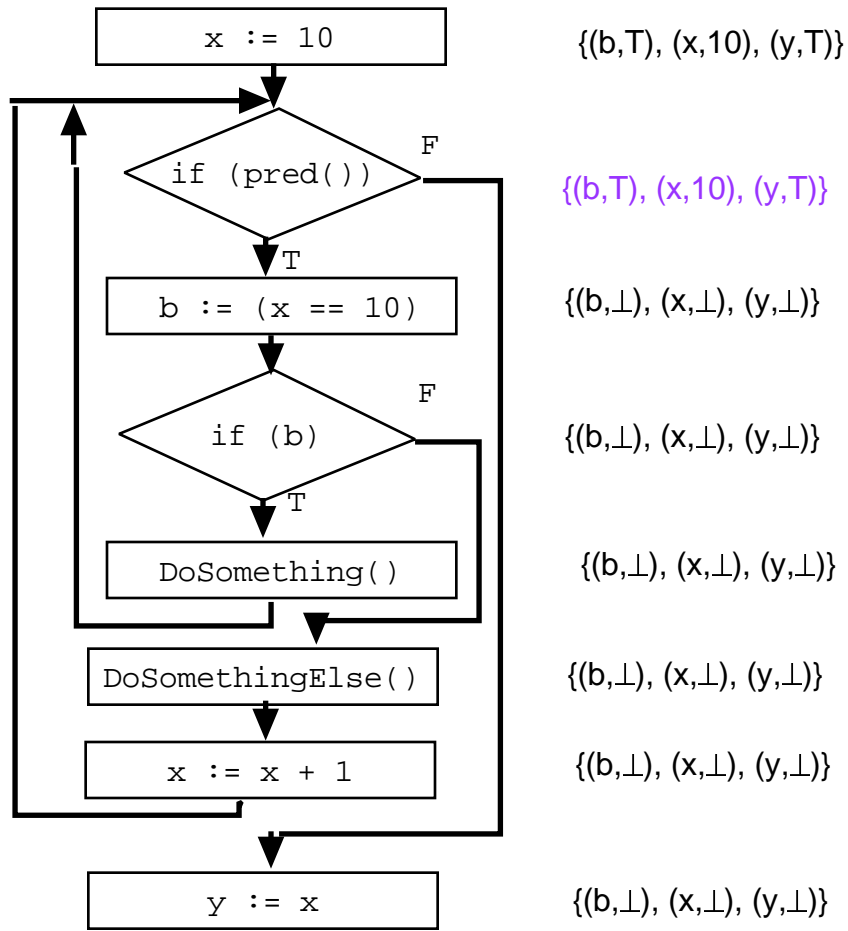
# Example using graph replacement (0)



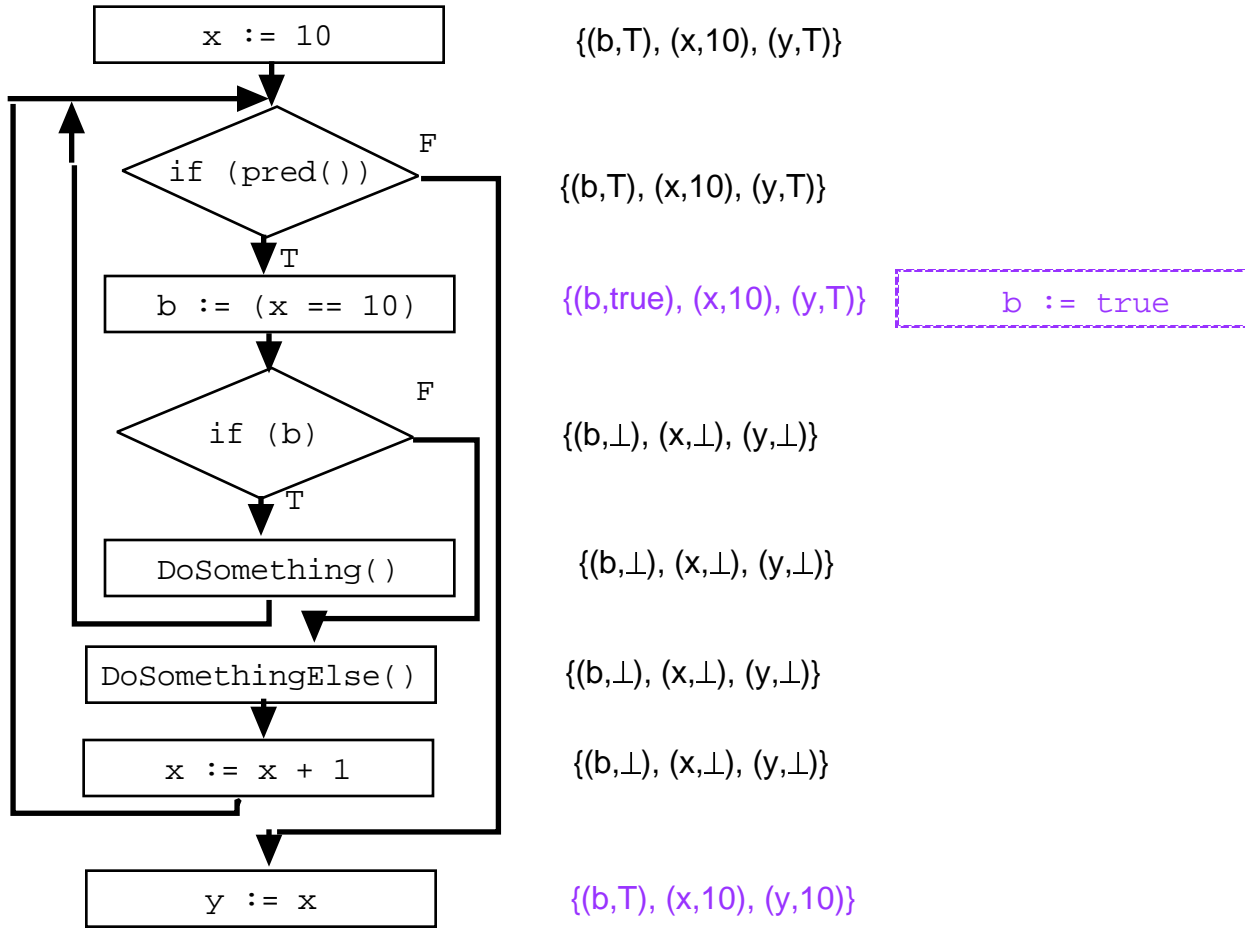
# Example using graph replacement (1)



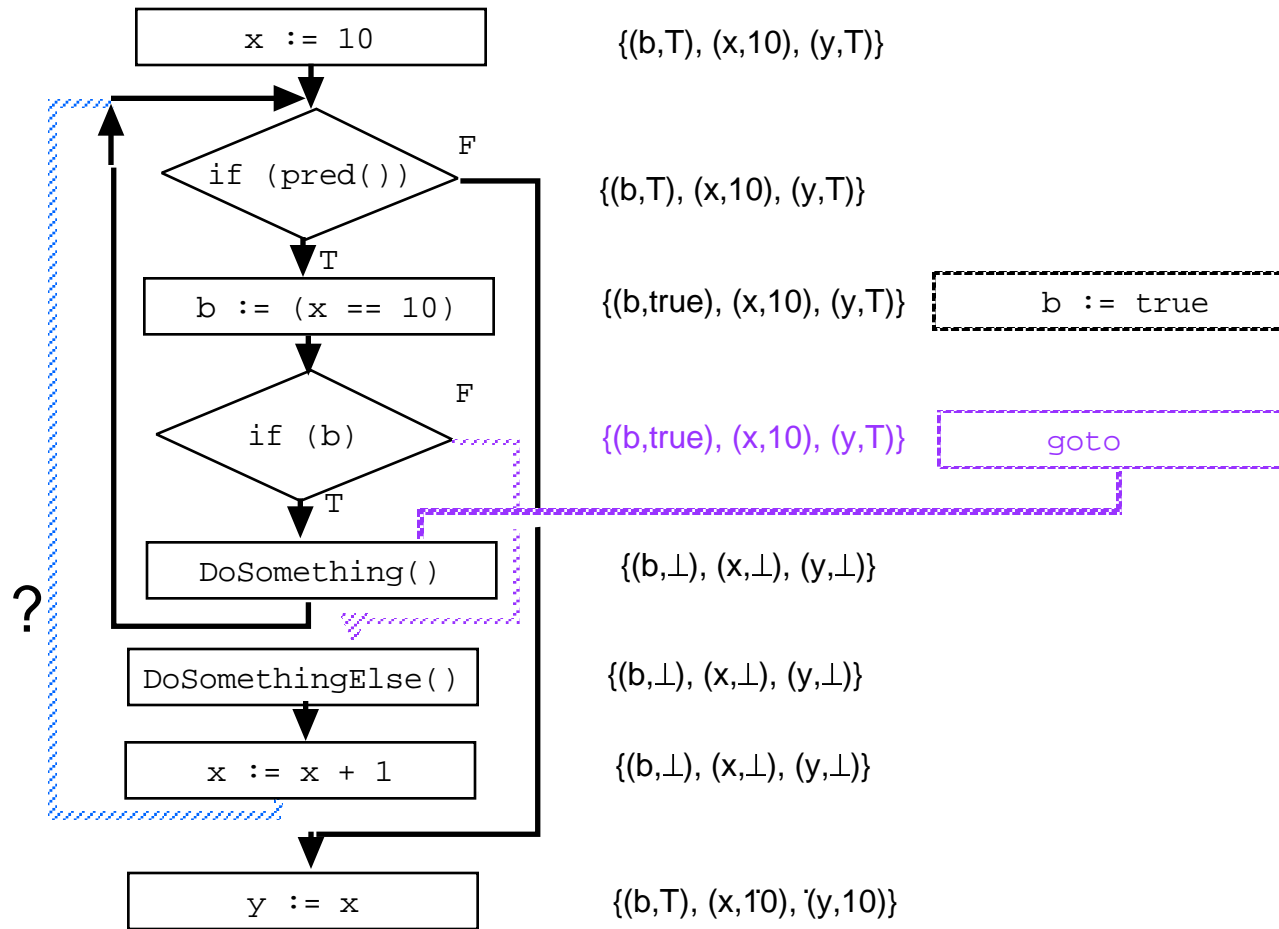
## Example using graph replacement (2)



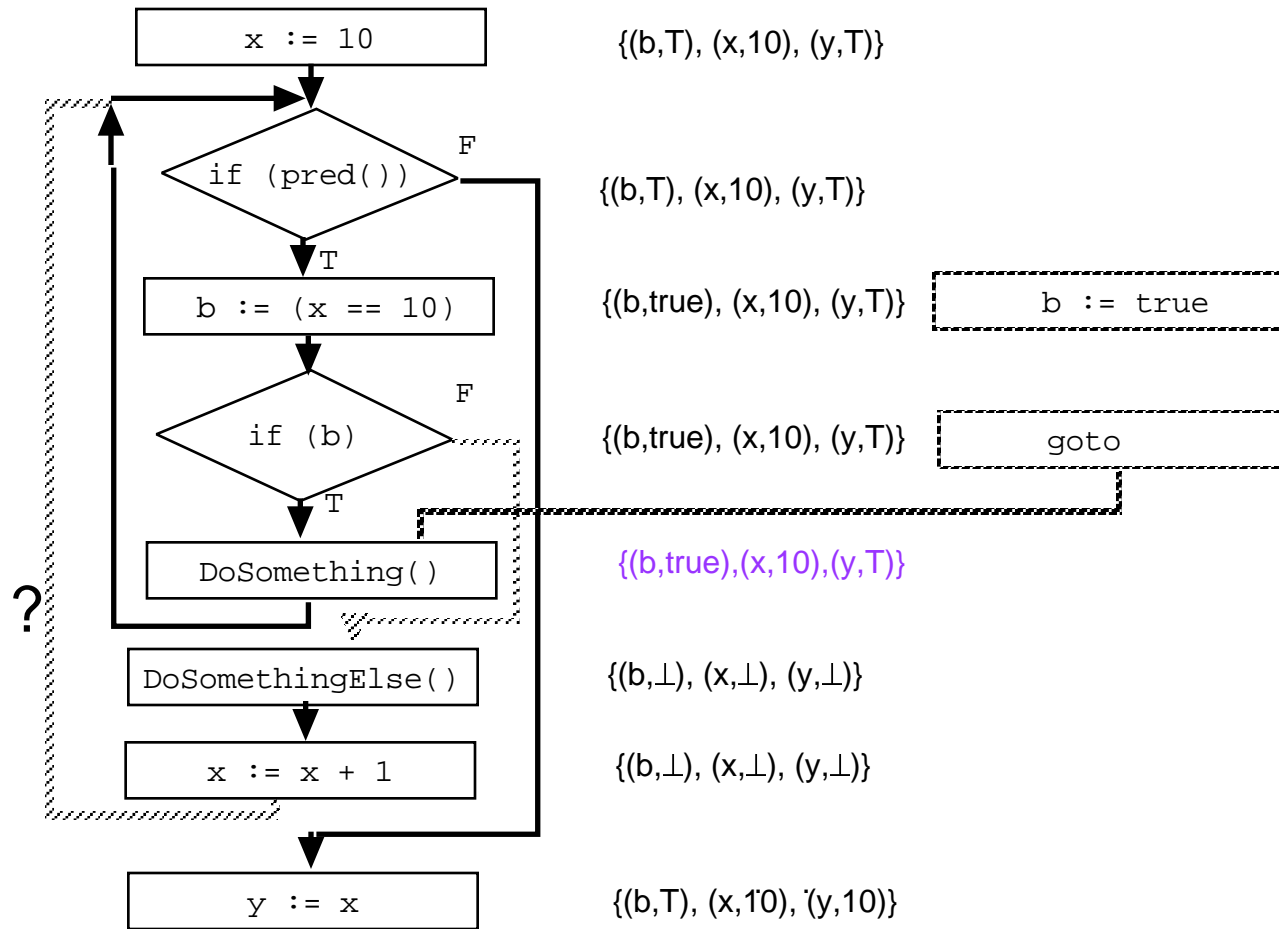
# Example using graph replacement (3)



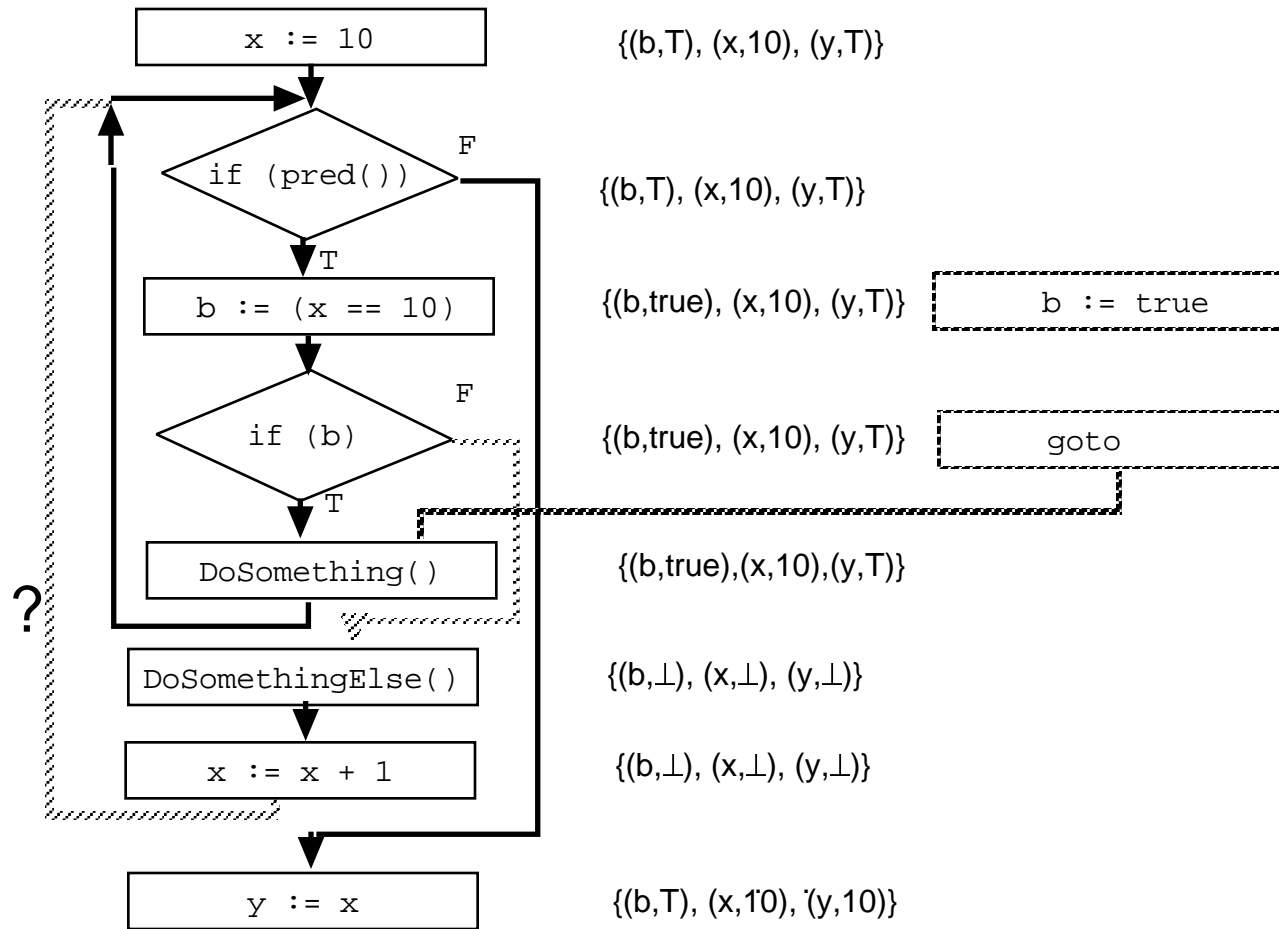
# Example using graph replacement (4)



# Example using graph replacement (5)

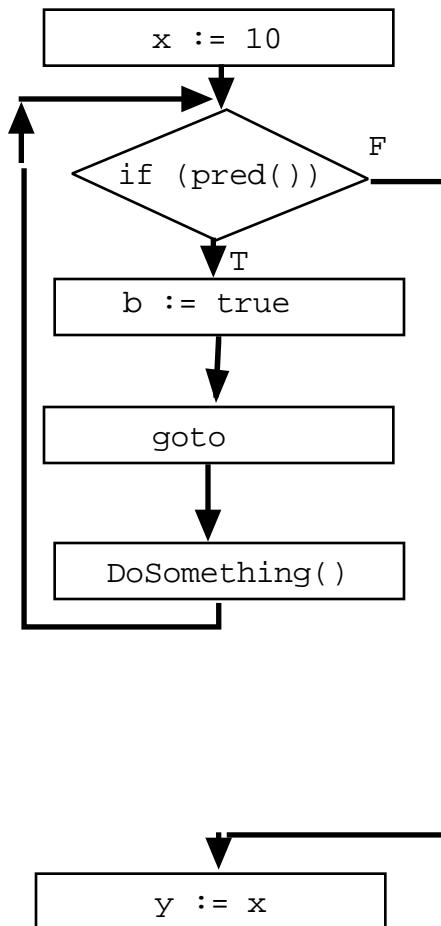


# Example using graph replacement (6)



## Example using graph replacement (7)

- After the fixed point is reached, commit transformations:



## Composing (?) analyses

- Vortex framework provides a `ComposedAnalysisInfo` class which is a tuple of dataflow facts from component analyses.
- When a cfg node is visited by a composed analysis:
  - each of the combined flow functions is computed for the node.
  - If all of them return *PROPAGATE* values, those are combined into the tuple to flow to successor nodes,
  - otherwise, the local analysis is rerun, with the first *REPLACE* value replacing the node.

## Another analysis to combine: class analysis

- $F\langle x := \text{new } C \rangle \sigma = \text{PROPAGATE}(\sigma[x \mapsto \{C\}])$

$$F\langle x := y.m(z_1, \dots, z_n) \rangle \sigma =$$

let  $methods = \bigcup_{c \in \sigma[y]} \text{method\_lookup}(c, m)$  in

if  $methods = \{F\}$  then

- $\text{REPLACE}(\langle x := F(y, z_1, \dots, z_n) \rangle)$

else

$$\text{PROPAGATE}(\sigma[x \mapsto T])$$

## Another analysis to combine: class analysis (cont)

$F\langle x := y \text{ instanceof } C \rangle \sigma =$

if  $\sigma[y] \subseteq \text{subclasses}(C)$  then

*REPLACE*( $\langle x := \text{true} \rangle$ )

- else if  $\sigma[y] \cap \text{subclasses}(C) == \emptyset$  then

*REPLACE*( $\langle x := \text{false} \rangle$ )

else

*PROPAGATE*( $\sigma[x \mapsto \text{Bool}]$ )

- Presumably confluence operator returns the closest common ancestor.

## Another analysis to combine: inlining

$F\langle x := F(y, y_1, \dots, y_n) \rangle \sigma =$

if `should_inline(F)` then

let  $G = \text{body}(F)$  in

- let  $G' = \text{subst\_formals}(G, x, y_1, \dots, y_n)$  in  
*REPLACE*( $\langle G' \rangle$ )

else

*PROPAGATE*( $\sigma$ )

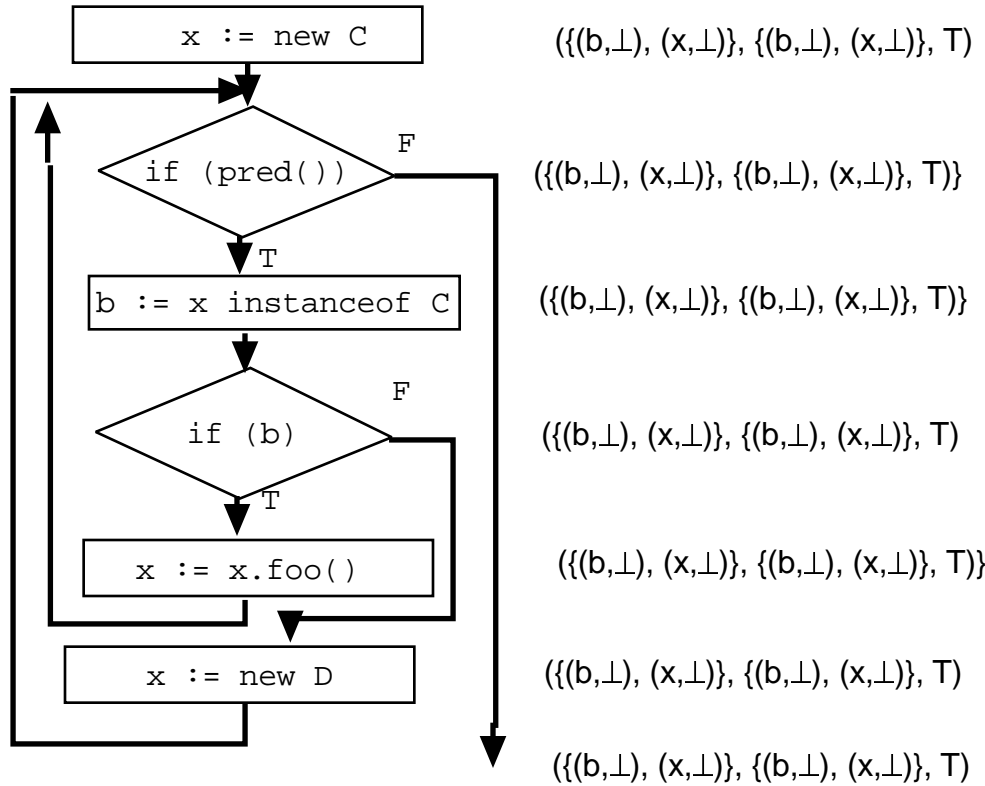
- Inlining is a pure transformation, its applicability does not depend on dataflow facts.

## Combined analysis example program

```
class A {
    A foo() {return new A;}
}
class C extends A {
    A foo() {return this;}
}
class D extends A {
}

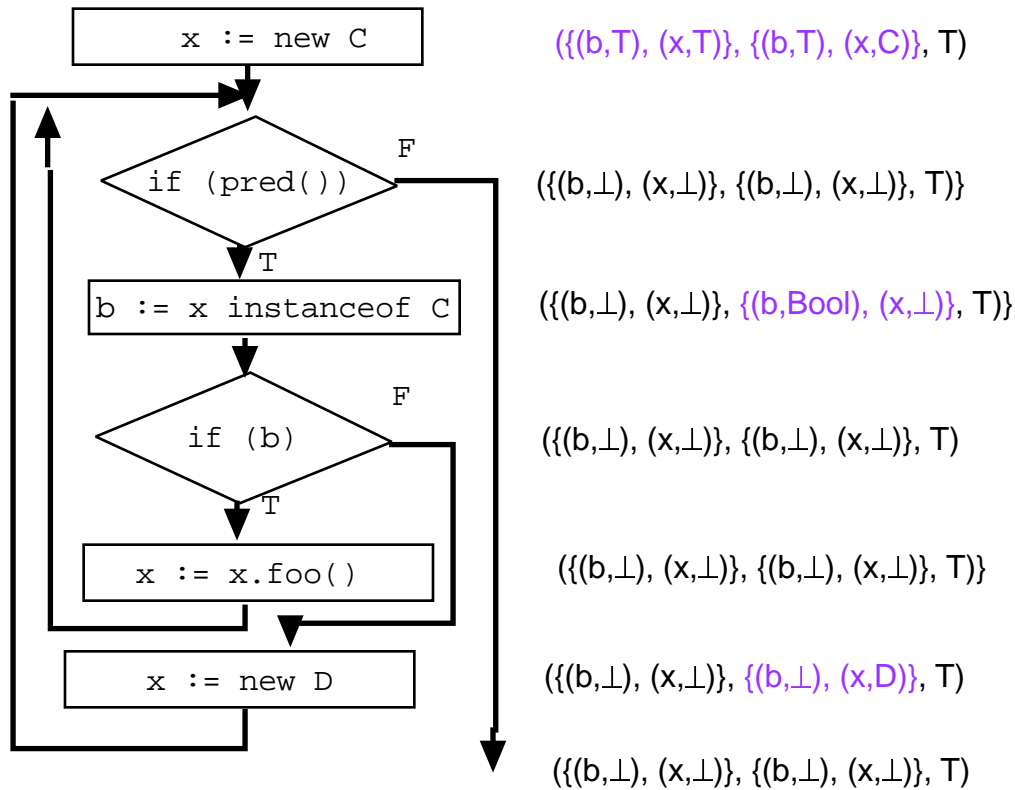
A x = new C;
while (pred()) {
    Bool b = x instanceof C;
    if (b) {
        x = x.foo();
    } else {
        x := new D;
    }
}
```

# Combined analysis example (0)



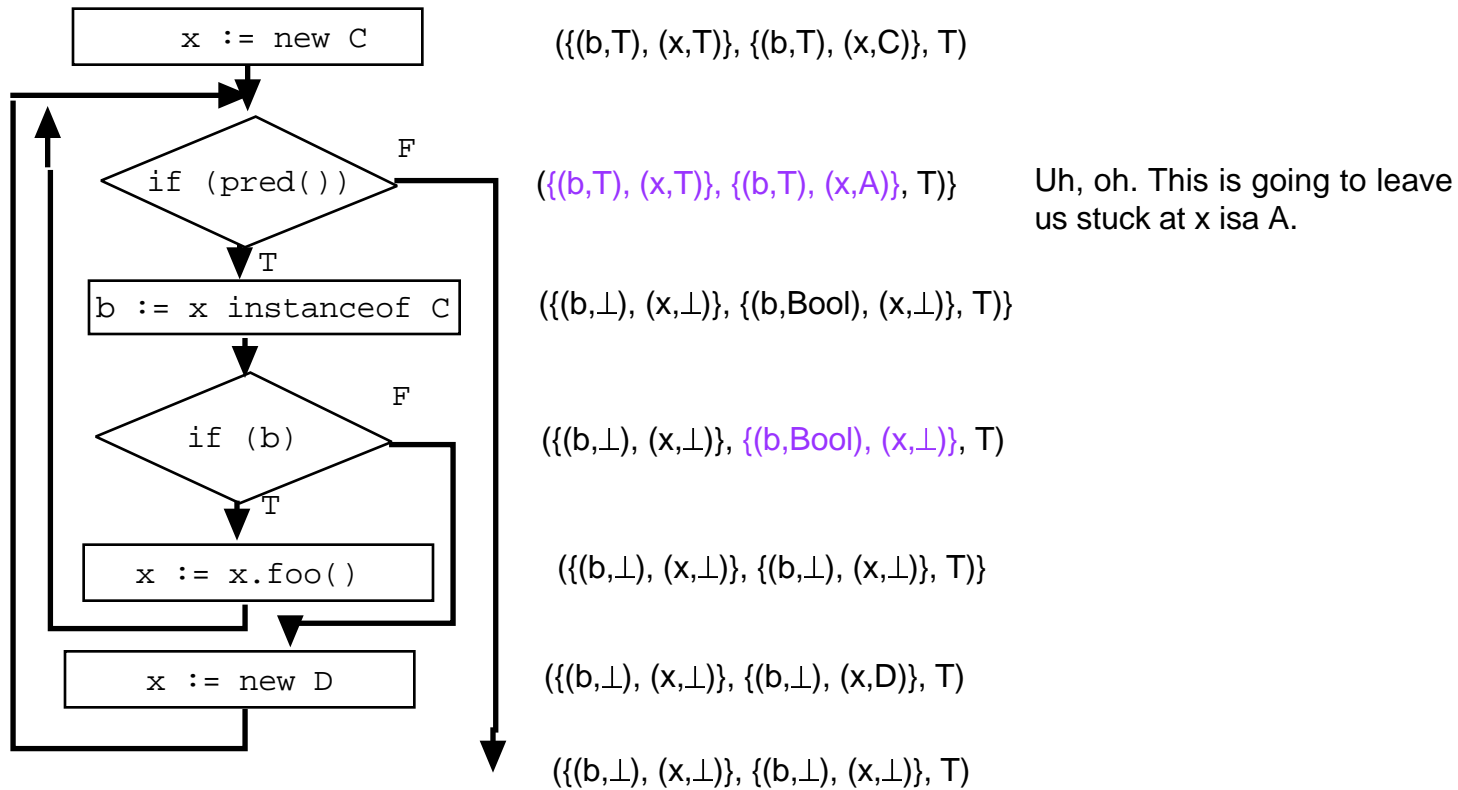
- Dataflow values are triples:  
     *(constant prop, class analysis, inlining)*

# Combined analysis example (1)



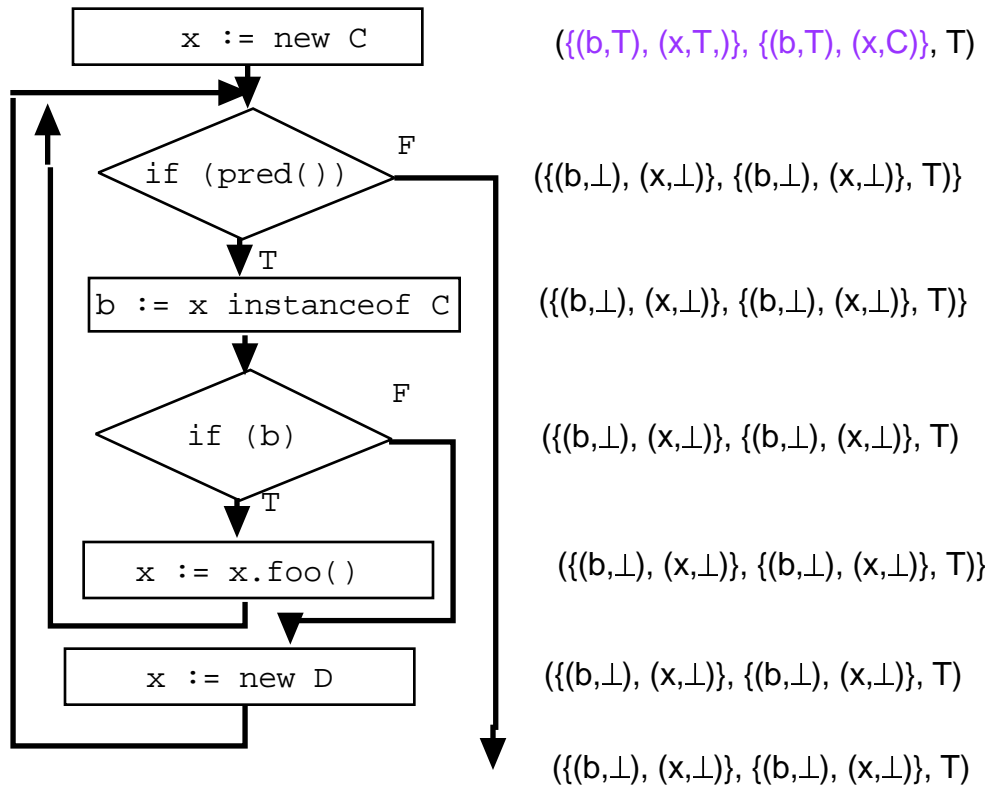
(The class flow function actually says  $(x, T)$ , but assuming that does not apply when input is  $\perp$ )

## Combined analysis example (2)



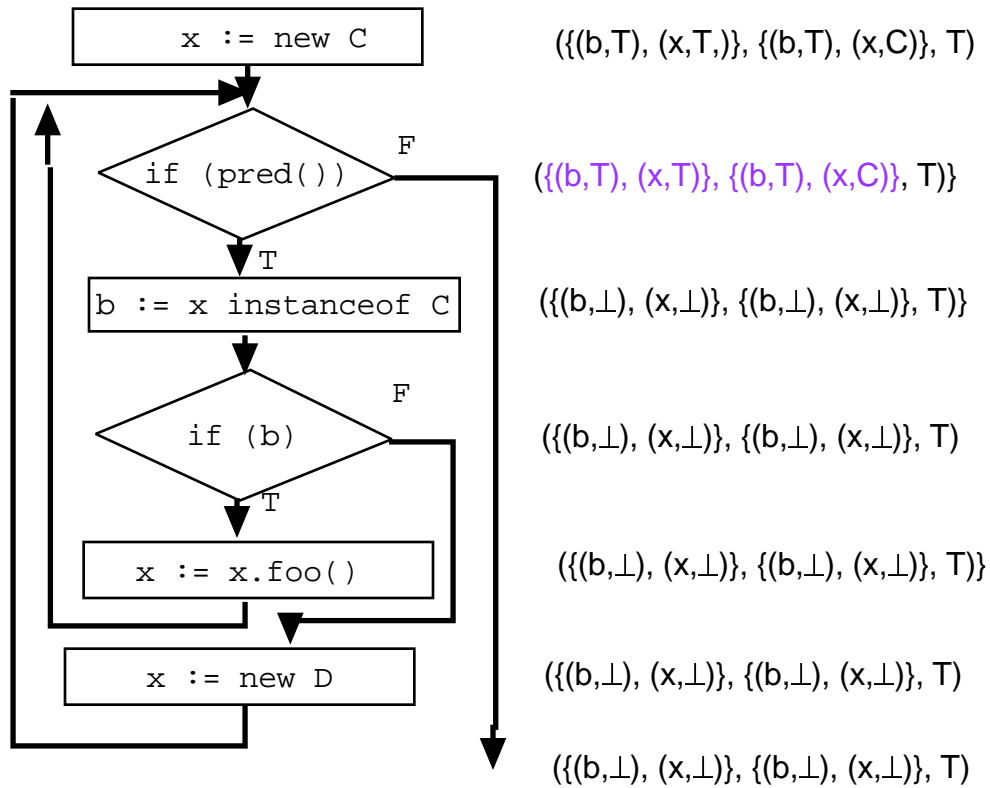
- So a naive dataflow implementation won't let "x isa C" flow around the loop.

# Combined analysis example, worklist (1)

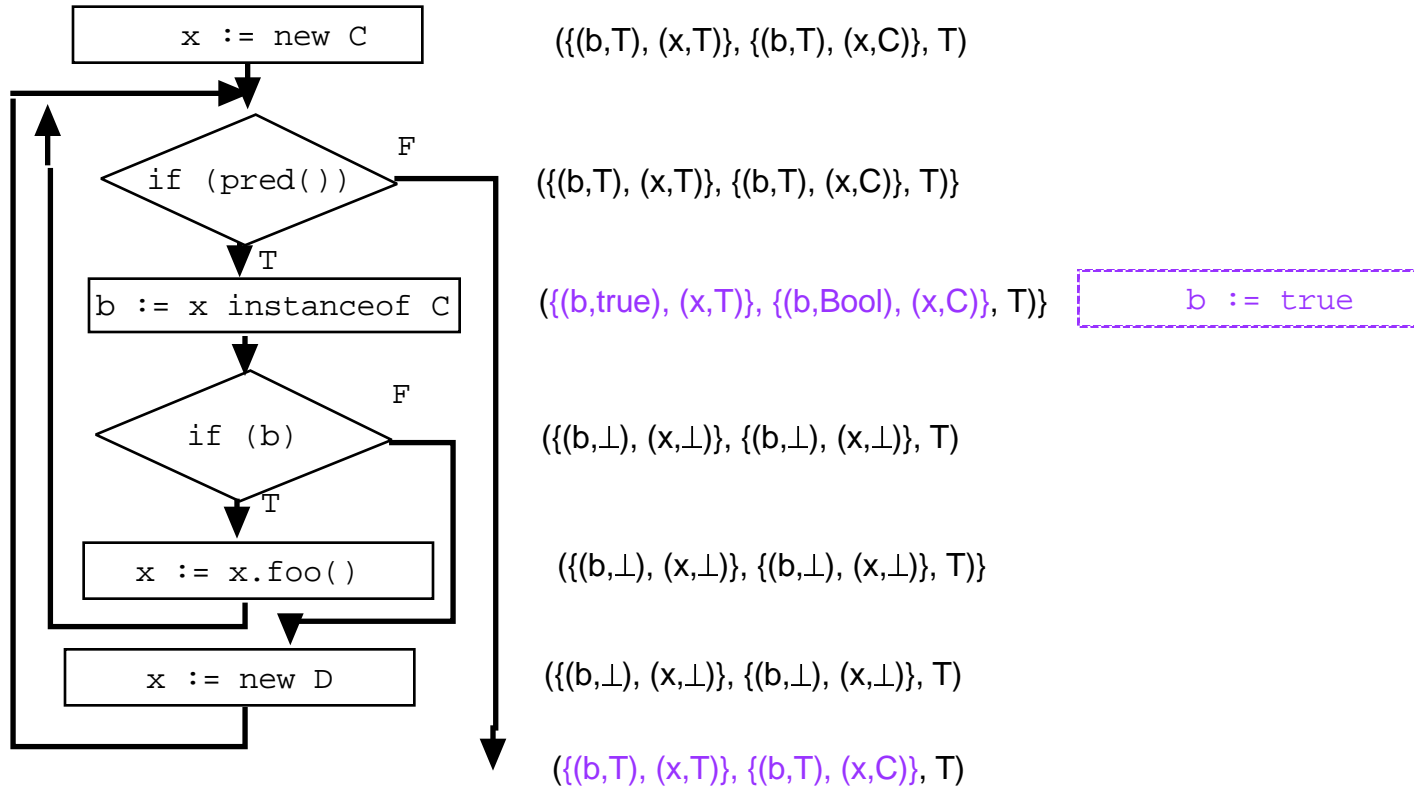


- (Changing only the successors of changed nodes.)

# Combined analysis example, worklist (2)

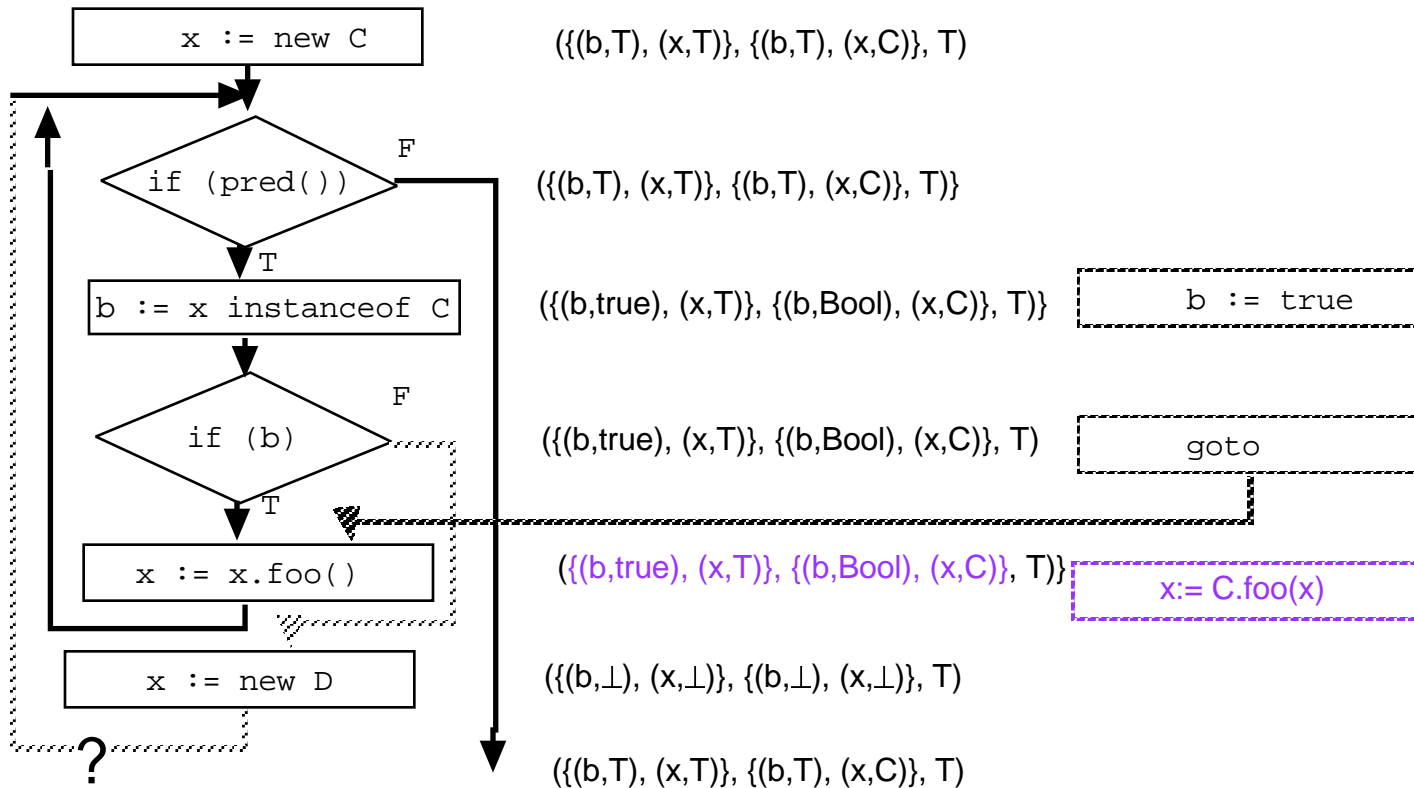


# Combined analysis example, worklist (3)

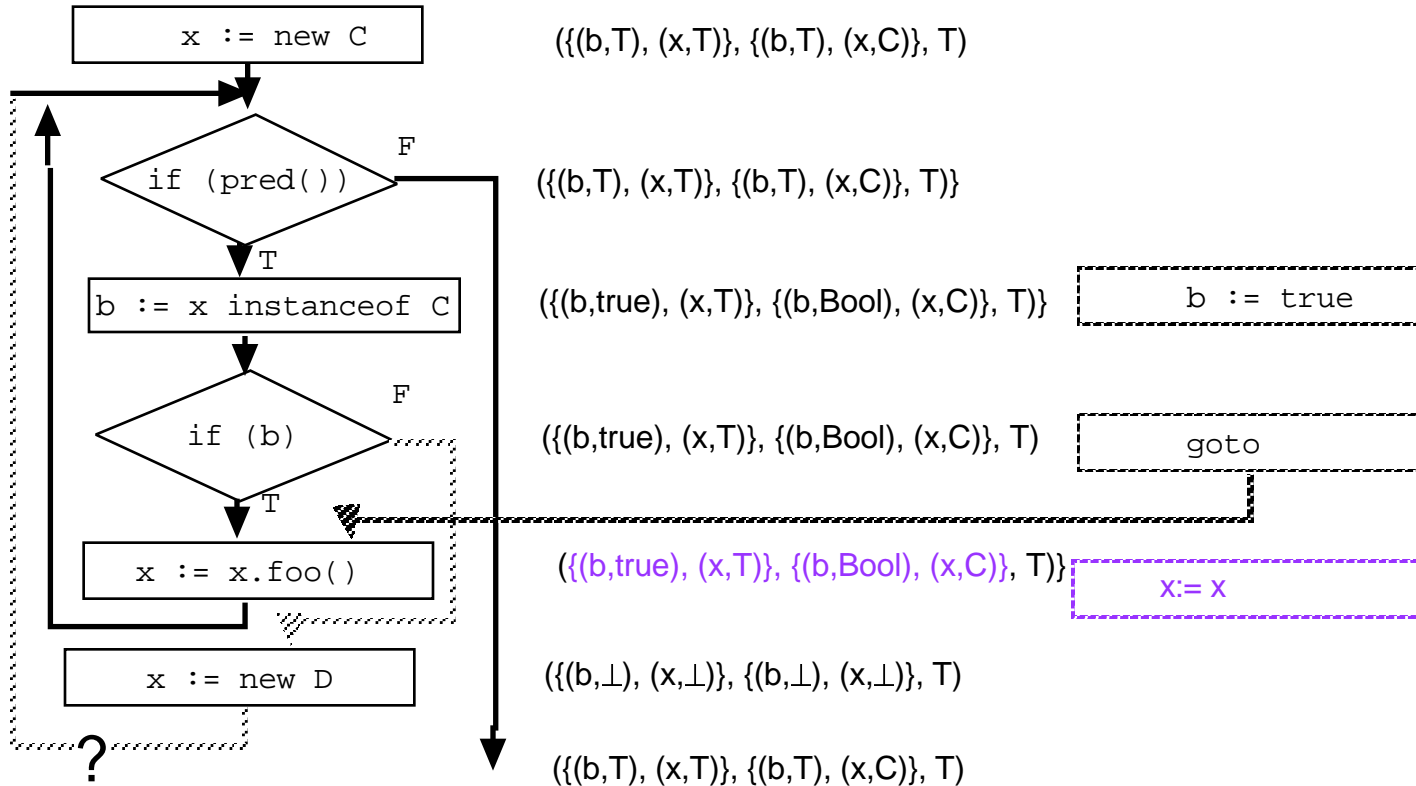




# Combined analysis example, worklist (5)



# Combined analysis example, worklist (6)

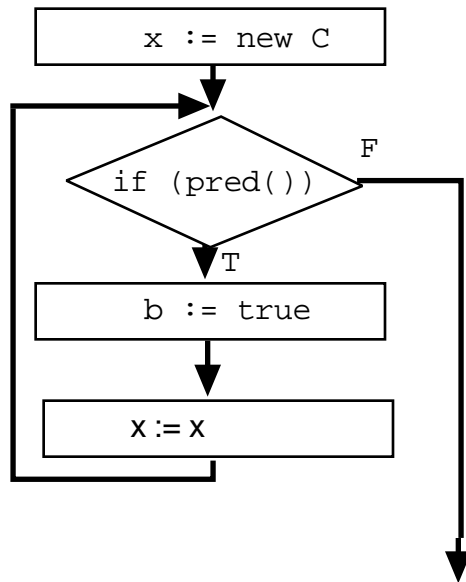


# Combined analysis example, worklist (7)



## Combined analysis example, worklist (8)

- After the fixed point is reached, commit transformations:



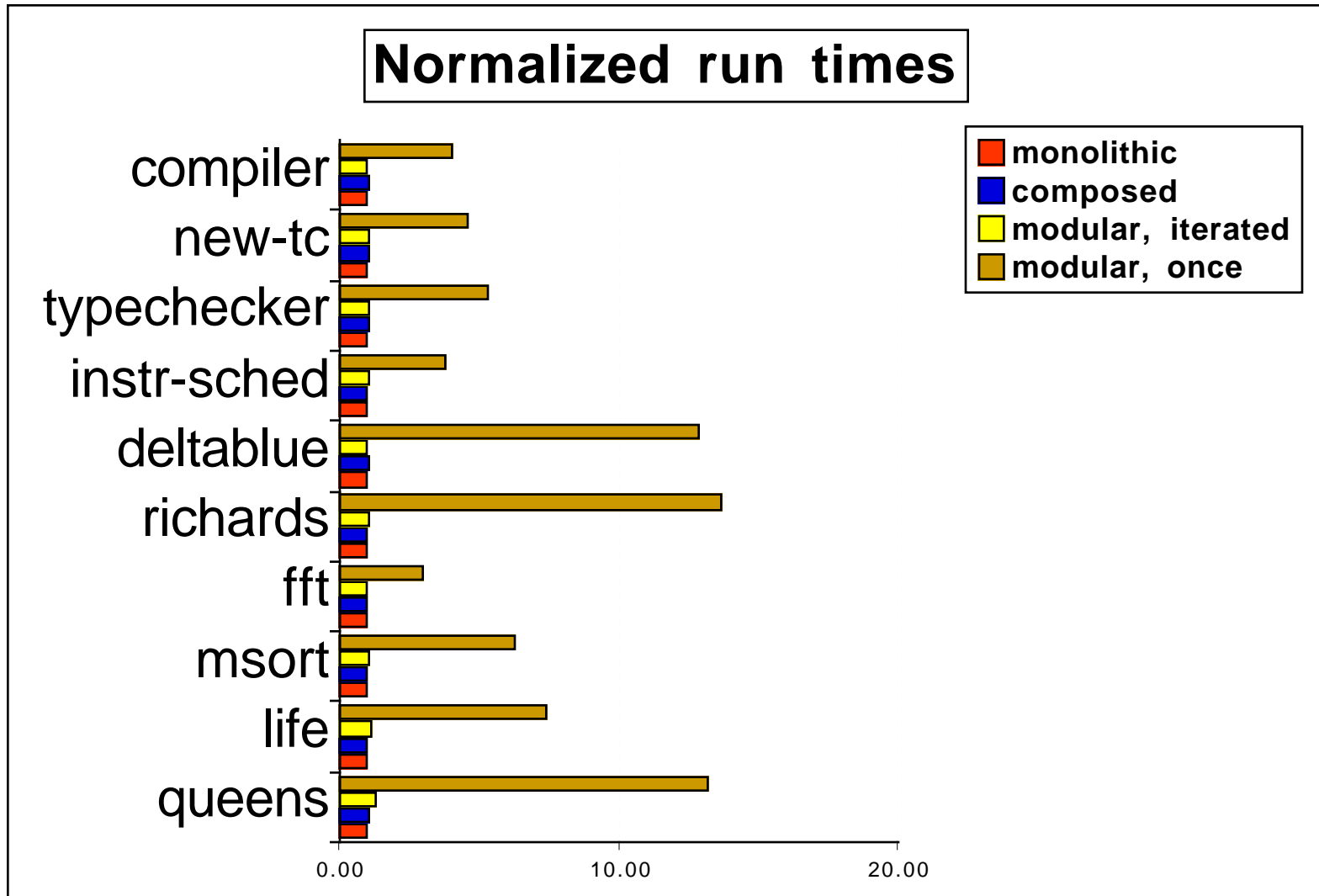
## Are the graph replacements really local?

- Replacing branches with gotos changes edges in the surrounding graph.
  - The Vortex framework removes dead code on the fly (changing more edges). Lerner et. al. say this is not necessary.
  - The proofs of soundness perform induction on the graph replacements. Is that sound when the replacements are not pure substructures?

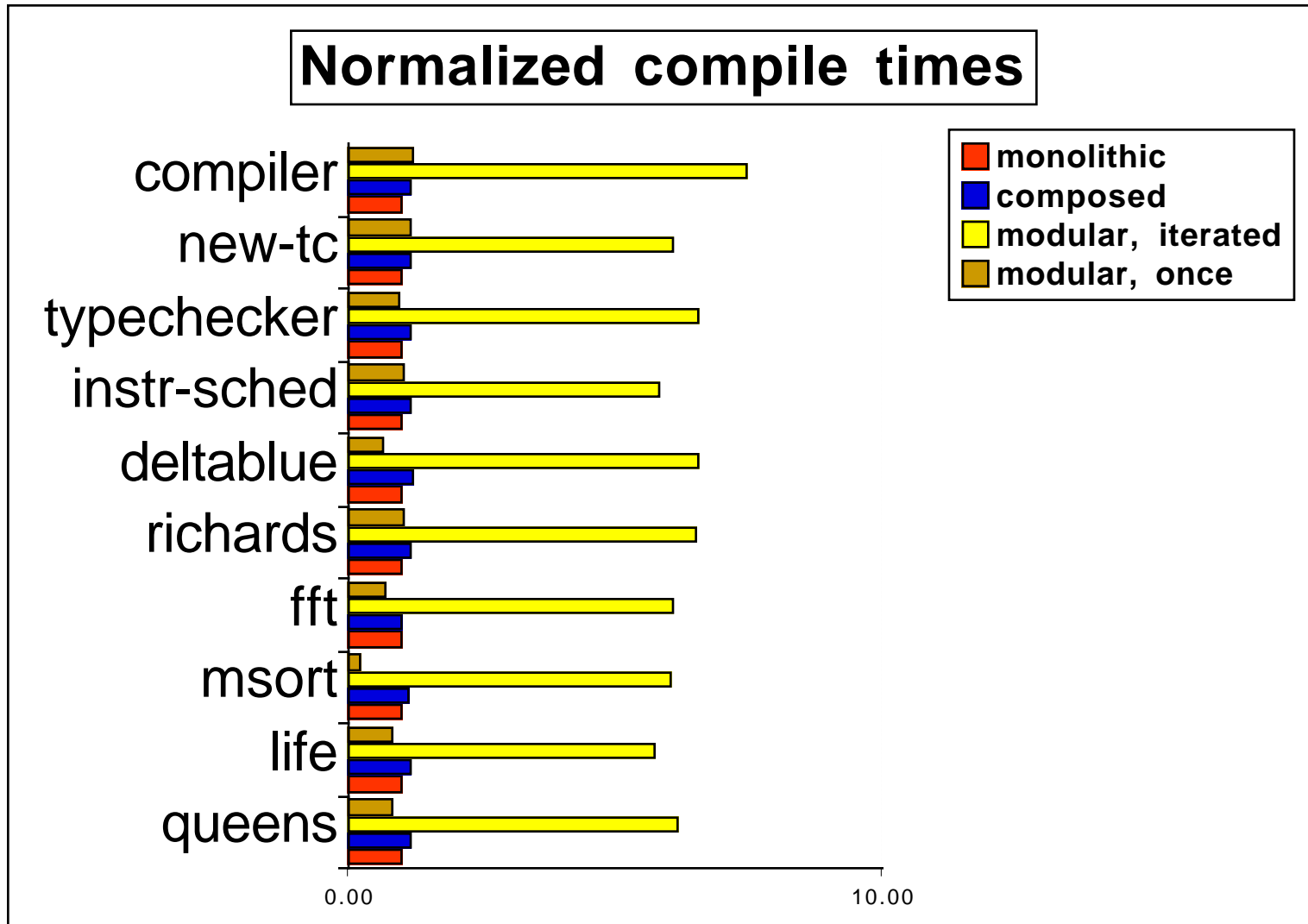
## Experimental Results —benchmarks

| <b>Benchmark</b> | <b>Number of lines</b> |
|------------------|------------------------|
| queens           | 50                     |
| life             | 80                     |
| msort            | 110                    |
| fft              | 150                    |
| richards         | 400                    |
| deltablue        | 650                    |
| instr-sched      | 2400                   |
| typechecker      | 20000                  |
| new-tc           | 23500                  |
| compiler         | 50000                  |

# Experimental Results



# Experimental Results



## Applicability to Soot

- Restrictions on combining via graph replacement:
  - Combined analyses must flow in same direction, use the same intermediate representation.
  - Replacement graph replaces only one node in cfg: how do you deal with transformations which reorder the cfg?
  - How do you communicate with a pure analysis which performs no transformations?
- Workarounds used or suggested in Vortex:
  - Analyses can “snoop” the dataflow values of other analyses (but lose modularity).
  - Replacement graphs may “tunnel” their input dataflow values from edges in the surrounding graph, rather than using the replaced node’s input edges.

## **Transform classes in Soot's packs**

(not all associated with a dataflow analysis)

- ArrayBoundsChecker
- CommonSubexpressionEliminator
- ConditionalBranchFolder
- ConstantPropagatorAndFolder
- CopyPropagator
- DeadAssignmentEliminator
- NullPointerChecker
- PartialRedundancyEliminator
- ProfilingGenerator
- RectangularArrayFinder
- StaticInliner
- StaticMethodBinder
- UnconditionalBranchFolder
- UnreachableCodeEliminator
- UnusedLocalEliminator

## Flow Analysis classes in Soot:

- Backward:
  - IsolatednessAnalysis (PRE)
  - GlobalAnticipatabilityAnalysis (PRE)
  - ArrayIndexLivenessAnalysis
  - SimpleLiveLocalsAnalysis
- Forward:
  - EarliestnessAnalysis (PRE)
  - DelayednessAnalysis (PRE)
  - LocalDefsFlowAnalysis
  - FastAvailableExpressionsAnalysis
  - SlowAvailableExpressionsAnalysis

## Clients of SimpleLocalDefs in Soot:

Best candidates for composition: already using a common dataflow analysis:

- AggregatorBody
- ClassFieldAnalysis
- ConstantPropagatorAndFolder
- ConstructorFolder
- CopyPropagator
- DeadAssignmentEliminator
- FastAvailableExpressionsAnalysis (also forward analysis)
- JasminClass
- LoadStoreOptimizer (can replace 2 statements at a time)
- LocalSplitter
- StaticMethodBinder
- TypeResolver