A survey of techniques for precise program slicing

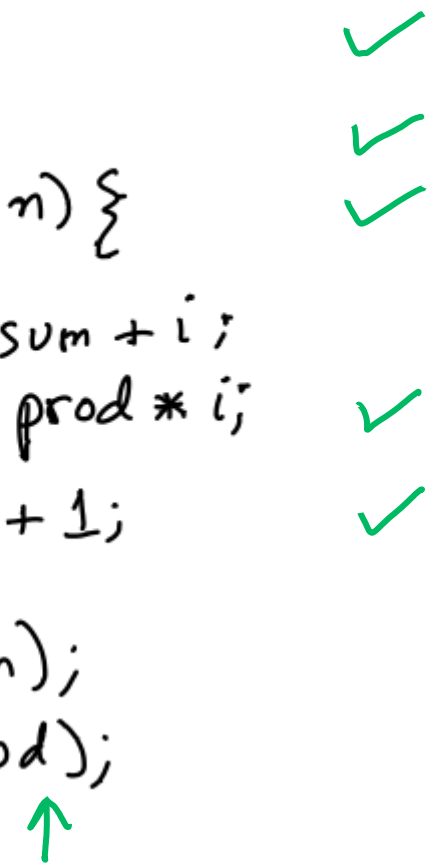
Komondoor V. Raghavan
Indian Institute of Science, Bangalore

The problem of program slicing

- Given a program *P*, and a statement *c* (the criterion), identify statements and conditionals in the program that are relevant to the variables that occur in *c*
  - A conditional is relevant if modifying the conditional could disturb the values of the variables in *c* from what's expected (on any input)
  - A statement is relevant if modifying its rhs could disturb the values of the variables at *c*
- Intuitively, a slice is a projection of *P* that's behaviorally equivalent to *P* wrt what's observable at *c*

An example

```
sum = 0;
prod = 1;
i = 1;
while (i ≤ n) {
    sum = sum + i;
    prod = prod * i;
    i = i + 1;
}
print (sum);
print (prod);
```

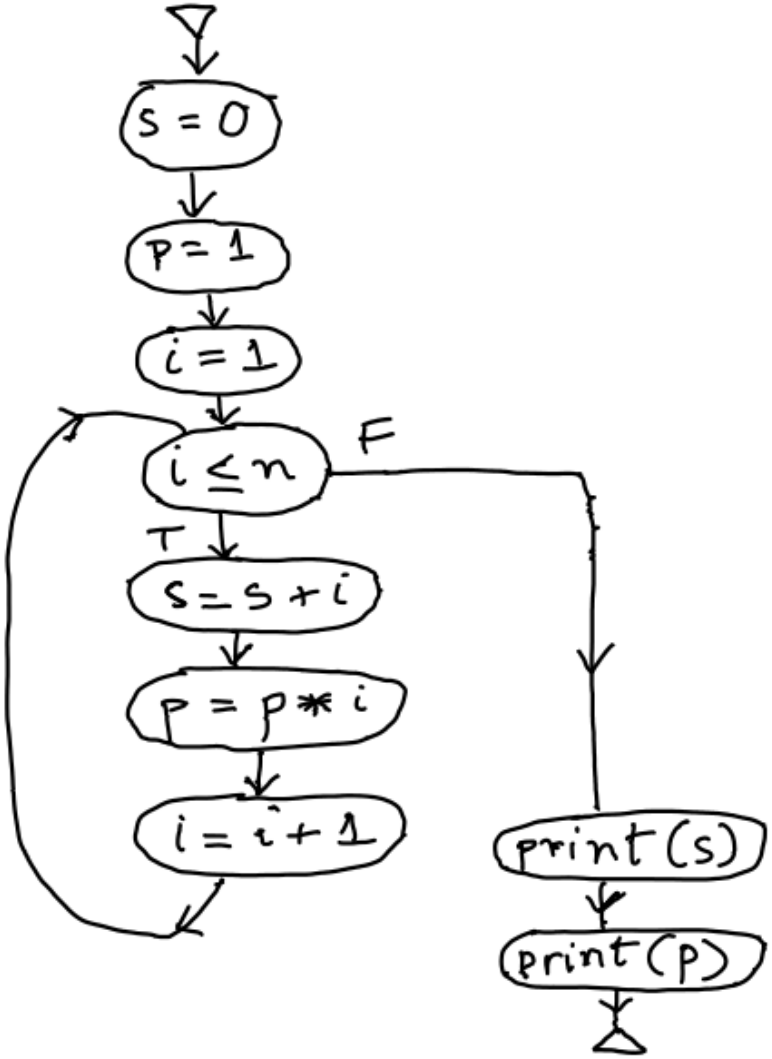


Applications of slicing

- Software understanding tools
- Software maintenance tools
  - Clone detection
  - Merging back different variants of a program
  - Decomposition of monolithic programs into coherent functionalities (e.g., sum-product example)
  - Recovering independent threads from sequential program
- Compilers and verification tools
  - Improves scalability, by identifying portion of program that's relevant to a property that needs to be checked

Control flow graph

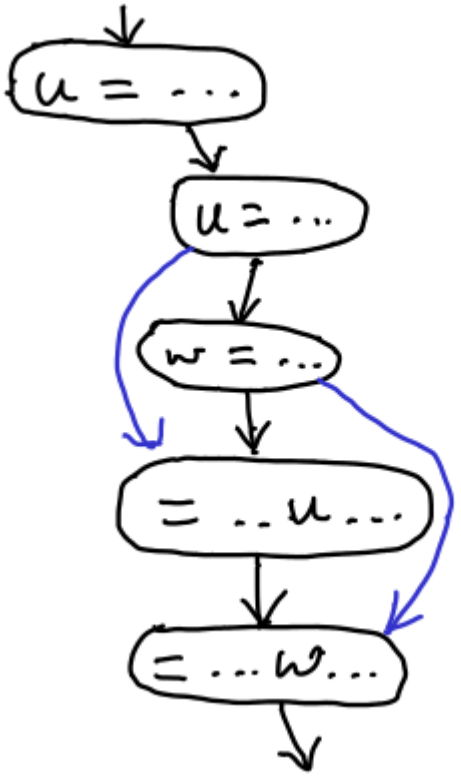
```
sum = 0;
prod = 1;
i = 1;
while (i ≤ n) {
    sum = sum + i;
    prod = prod * i;
    i = i + 1;
}
print (sum);
print (prod);
```



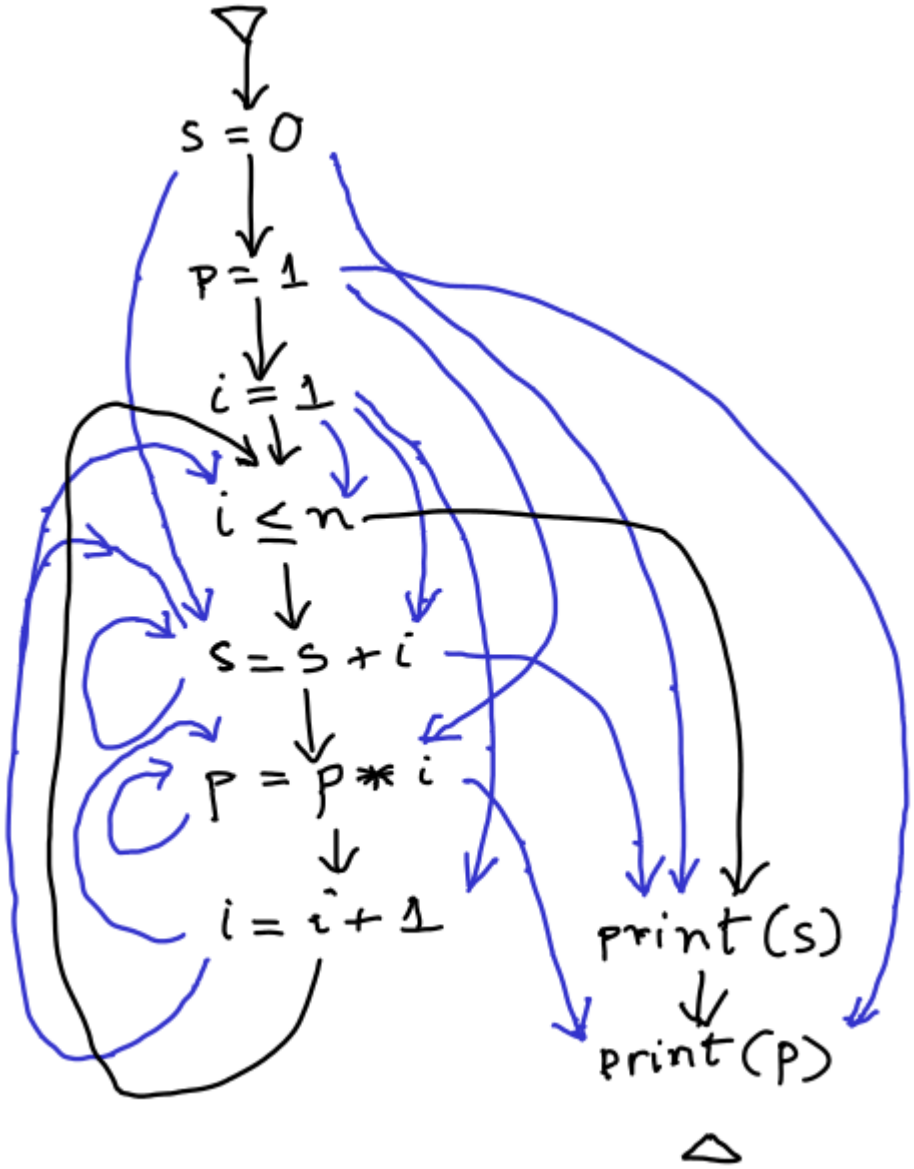
Flow dependence relation

s1 $\longrightarrow$ *s2* if

- *s1* defines a variable *v*
- *s2* uses *v*
- there is a control-flow path from *s1* to *s2* along which no other statement defines *v*



Flow dependences

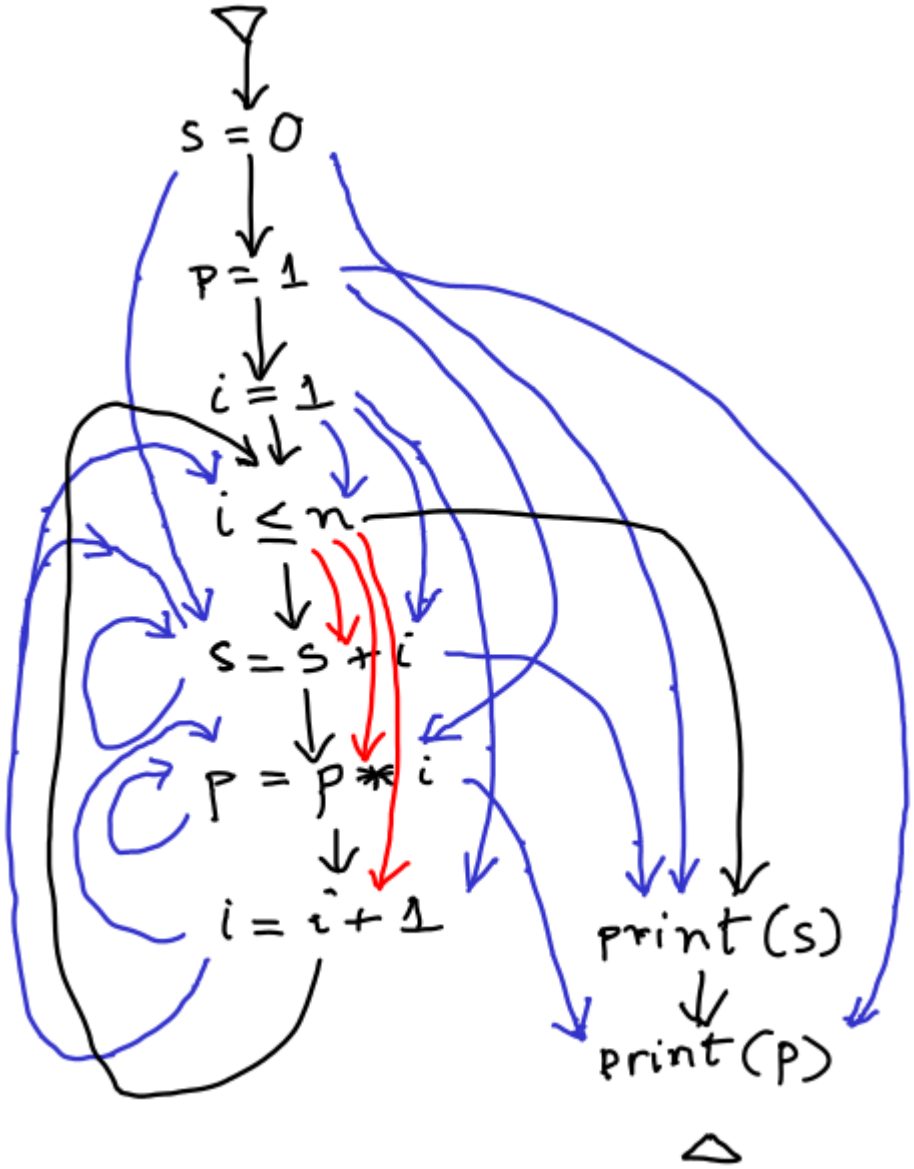


Control dependence relation

s1 $\longrightarrow$ *s2* if

- *s1* is a conditional
- *s2* is definitely reachable along one branch out of *s1*
- there is a path along the other branch along which *s2* is not reached

Flow + control dependences



Basic slicing technique

1. From *P*, construct flow dependence relation *F* and control dependence relation *C*
2. Obtain reflexive-transitive closure *R* of ($F \cup C$)
3. Slice = { *s* | <*s, c*> in *R* }, where *c* is given criterion

Illustration of slicing

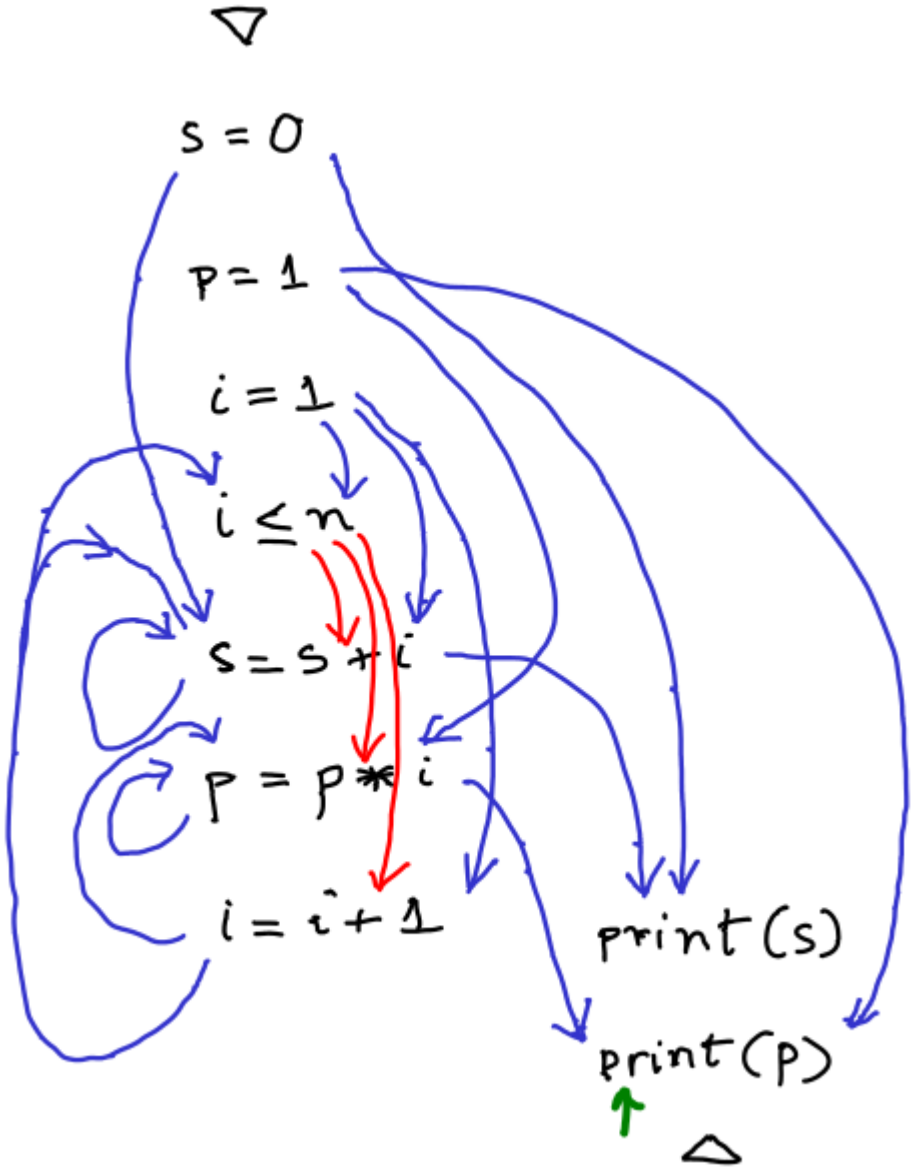
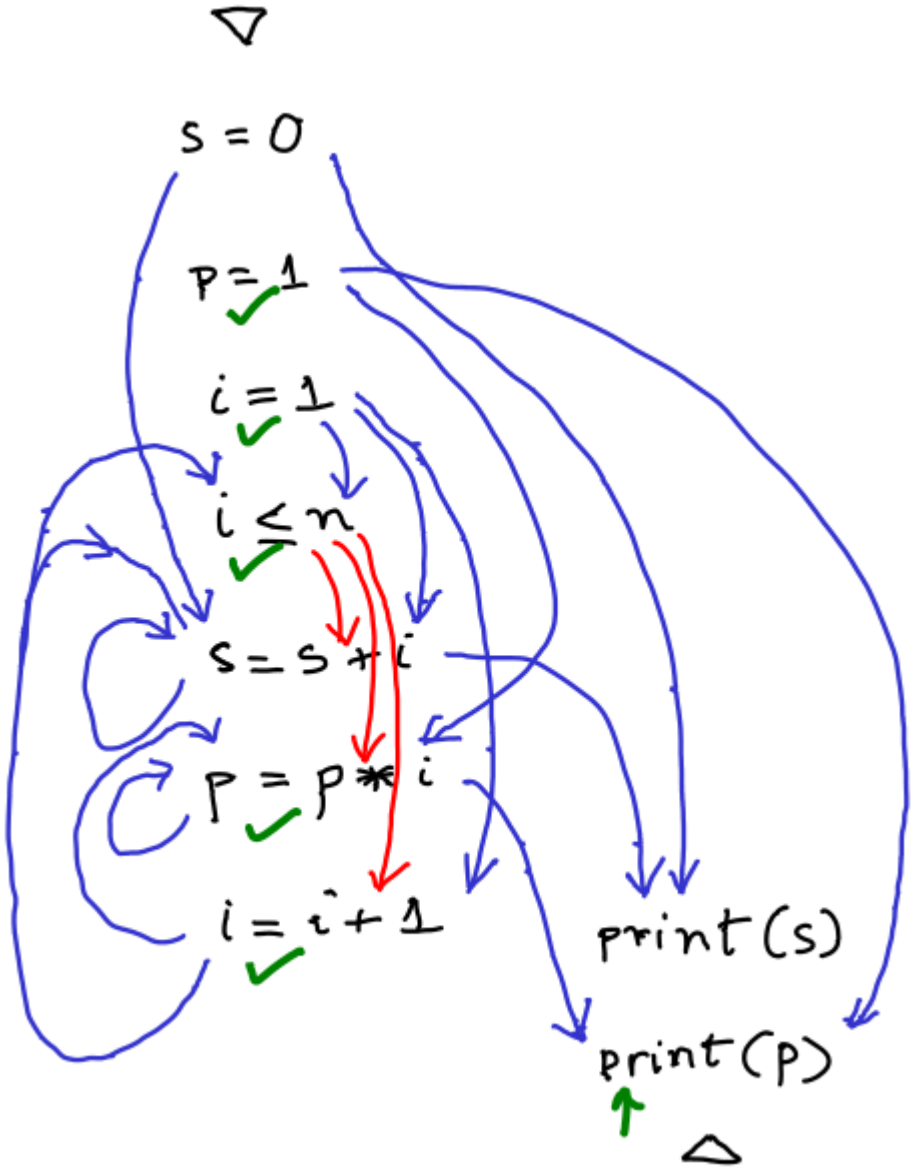
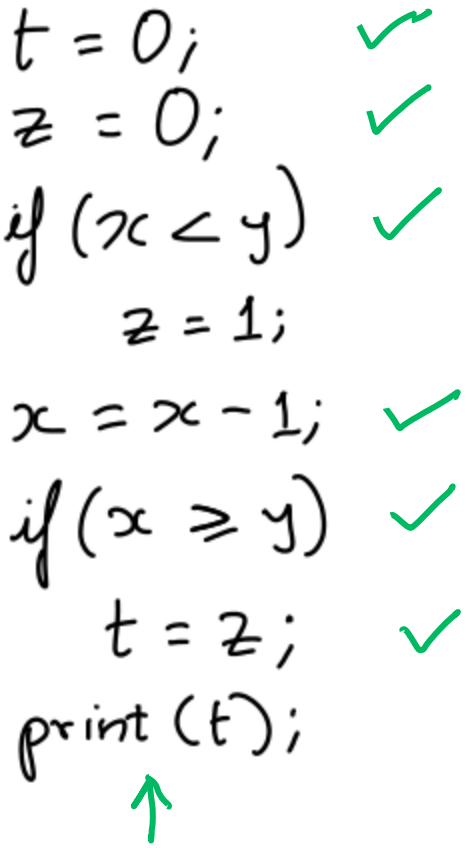


Illustration of slicing

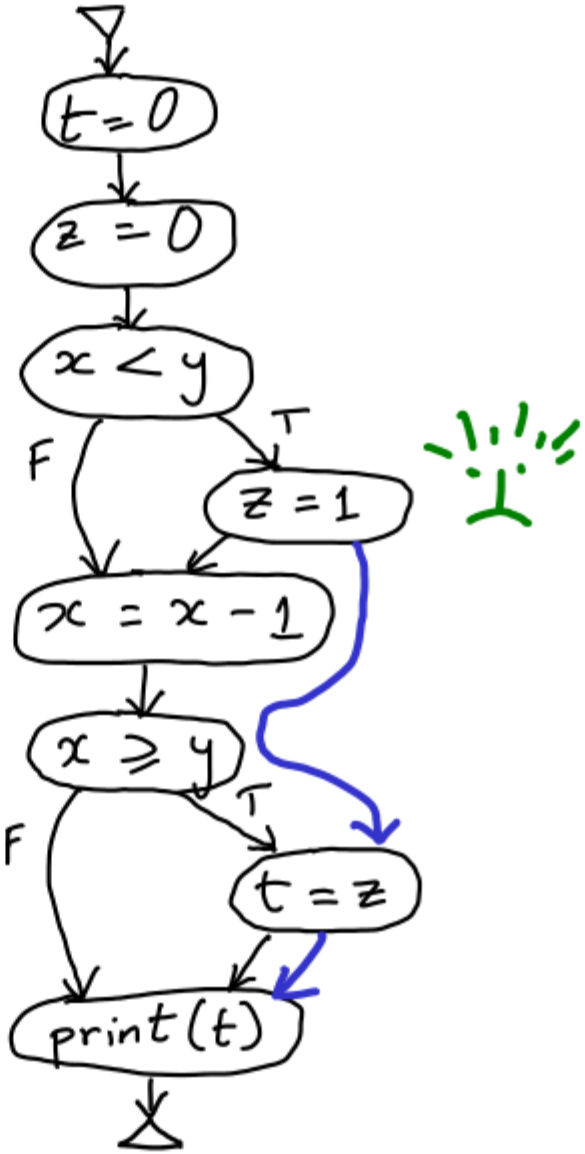


A more complex example



Basic technique yields imprecise slice

```
t = 0;
z = 0;
if (x < y)
    z = 1;
x = x - 1;
if (x ≥ y)
    t = z;
print (t);
```

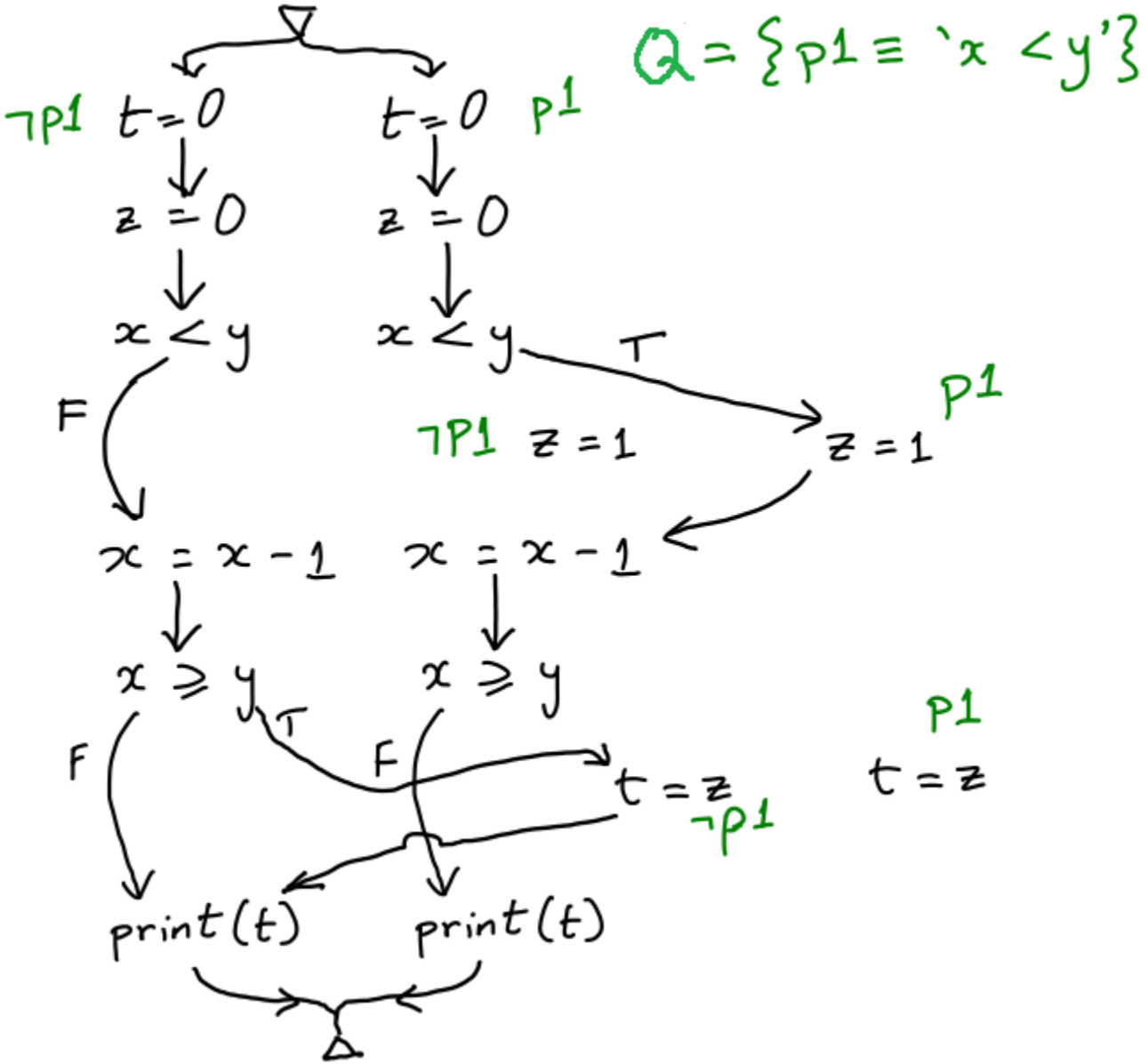


Need to rule out infeasible paths

[Hong et al., '95] achieve this by code duplication

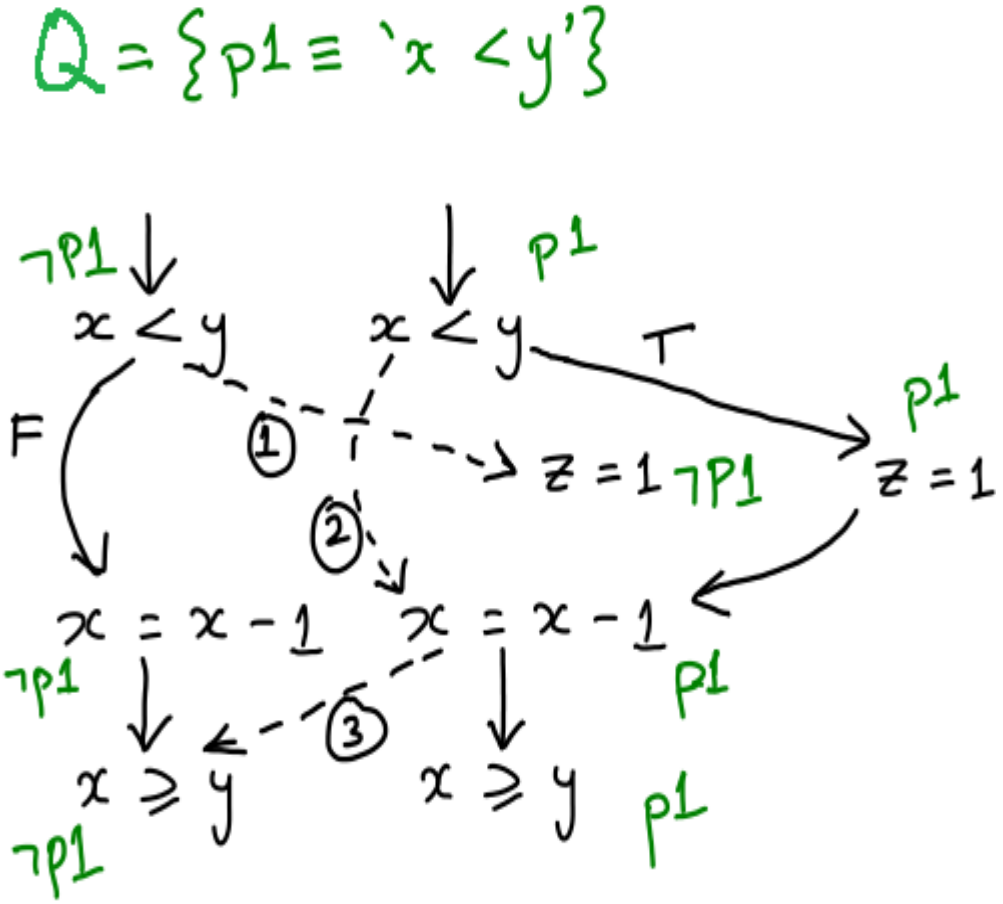
- Take a set of predicates Q (on program variables) as input
- Make up to $2^{|Q|}$ copies of each statement, one for each combination of predicate evaluations
- Identify feasible paths in this "exploded" flow graph
- Then, apply usual slicing technique on this exploded graph

Exploded flow graph



Adding edges in exploded flow graph

- Edge (1) not present because in state ¬p1 x < y cannot True
- Edge (2) not present for similar reason
- Edge (3) not present because:
  Program in state p1 remains in same state after executing "x = x – 1"

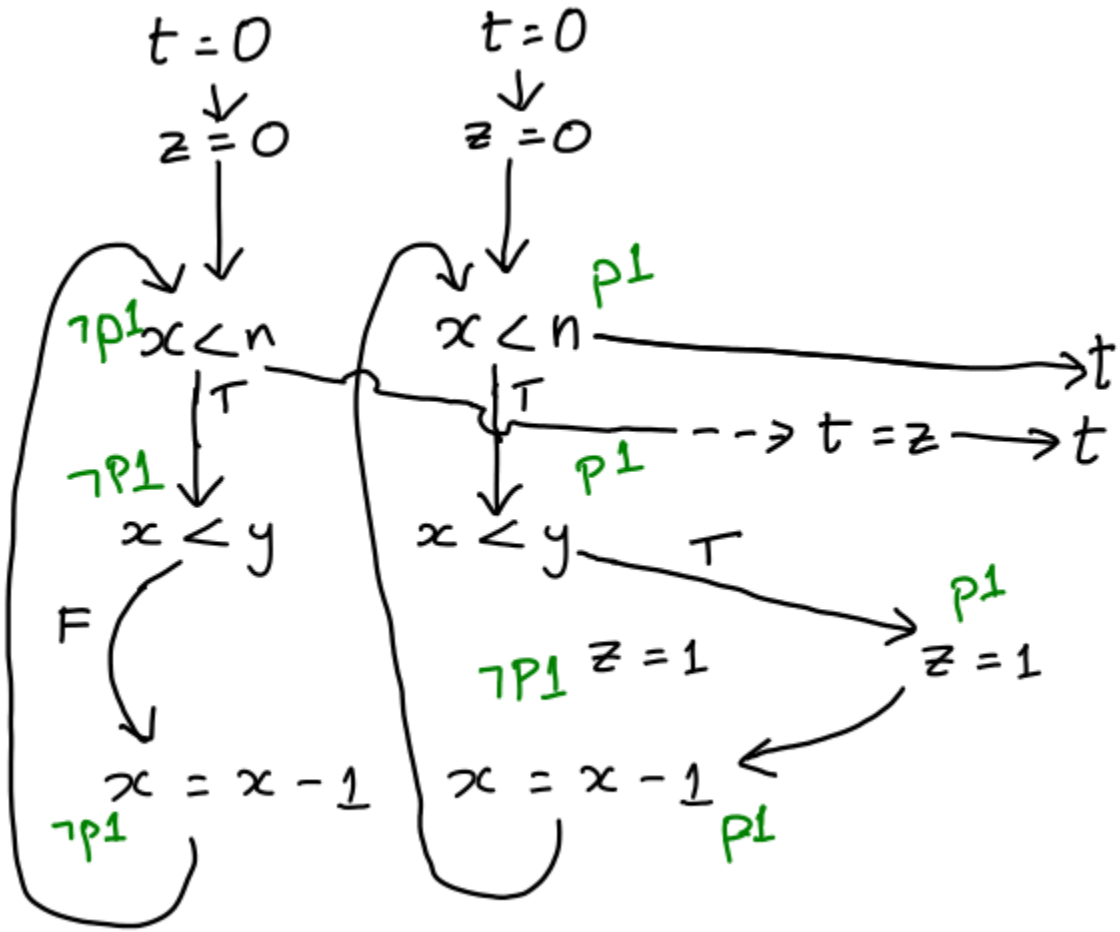


Loops

```
t = 0;
z = 0;
while (x < n)
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```

Loops

```
t = 0;
z = 0;
while (x < n) {
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```

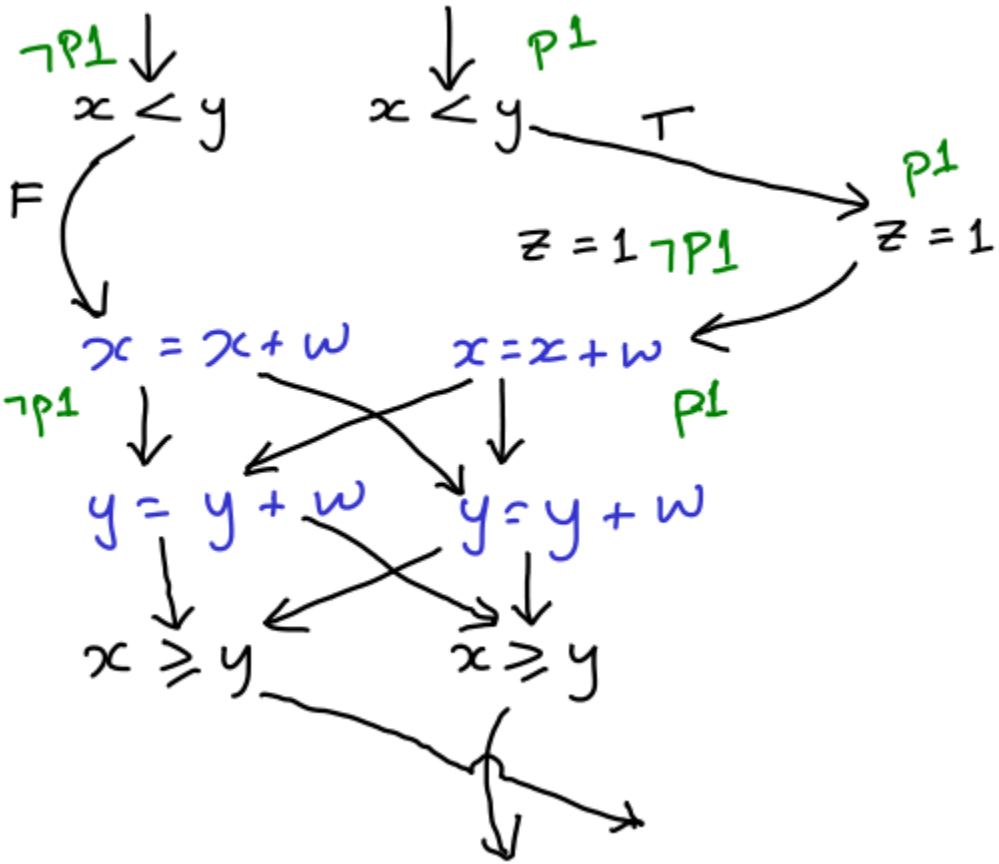


Precision is closely linked to given partitioning

```
t = 0;
z = 0;
if (x < y)
    z = 1;
x = x + w;
y = y + w;
if (x ≥ y)
    t = z;
print (t);
```

Precision is closely linked to given partitioning ☹

```
t = 0;
z = 0;
if (x < y)
    z = 1;
x = x + w;
y = y + w;
if (x ≥ y)
    t = z;
print (t);
```



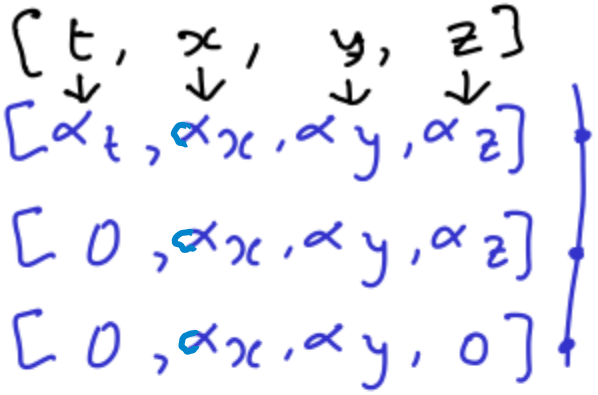
Summary of Hong et al.

- Obtains more precise slices than standard slicing, by excluding certain infeasible paths
- Handles loops cleanly
- Precision is linked to given partitioning *Q*
  - Partitioning needs to be selected carefully, based on statements in program
  - In general, a bigger *Q* gives better precision (at the expense of slicing time)
  - Other work exists to infer suitable *Q* automatically from program by iterative refinement
    - However, in the context of verification, not slicing

An approach based on symbolic execution [Jaffar et al., '12]

- Explodes control-flow graph by symbolically executing all possible paths in the program
- Does not require *Q* as input
- Basic idea
  - During execution, at each point
    - Have a symbolic store, which tracks current values of variables as expressions on program's initial parameters
    - Have path constraint, which is a predicate on the initial parameters that needs to hold for path *p* to be feasible
  - If p is $s1 \rightsquigarrow sn$, and $sn \rightarrow sp$ and $sn \rightarrow sq$, split execution into two paths $s1 \rightsquigarrow sp$ and $s1 \rightsquigarrow sq$.

Illustration of symbolic execution



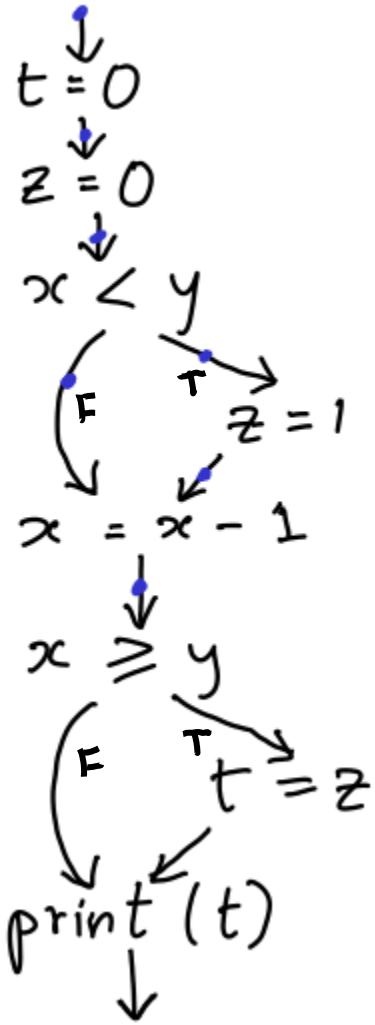
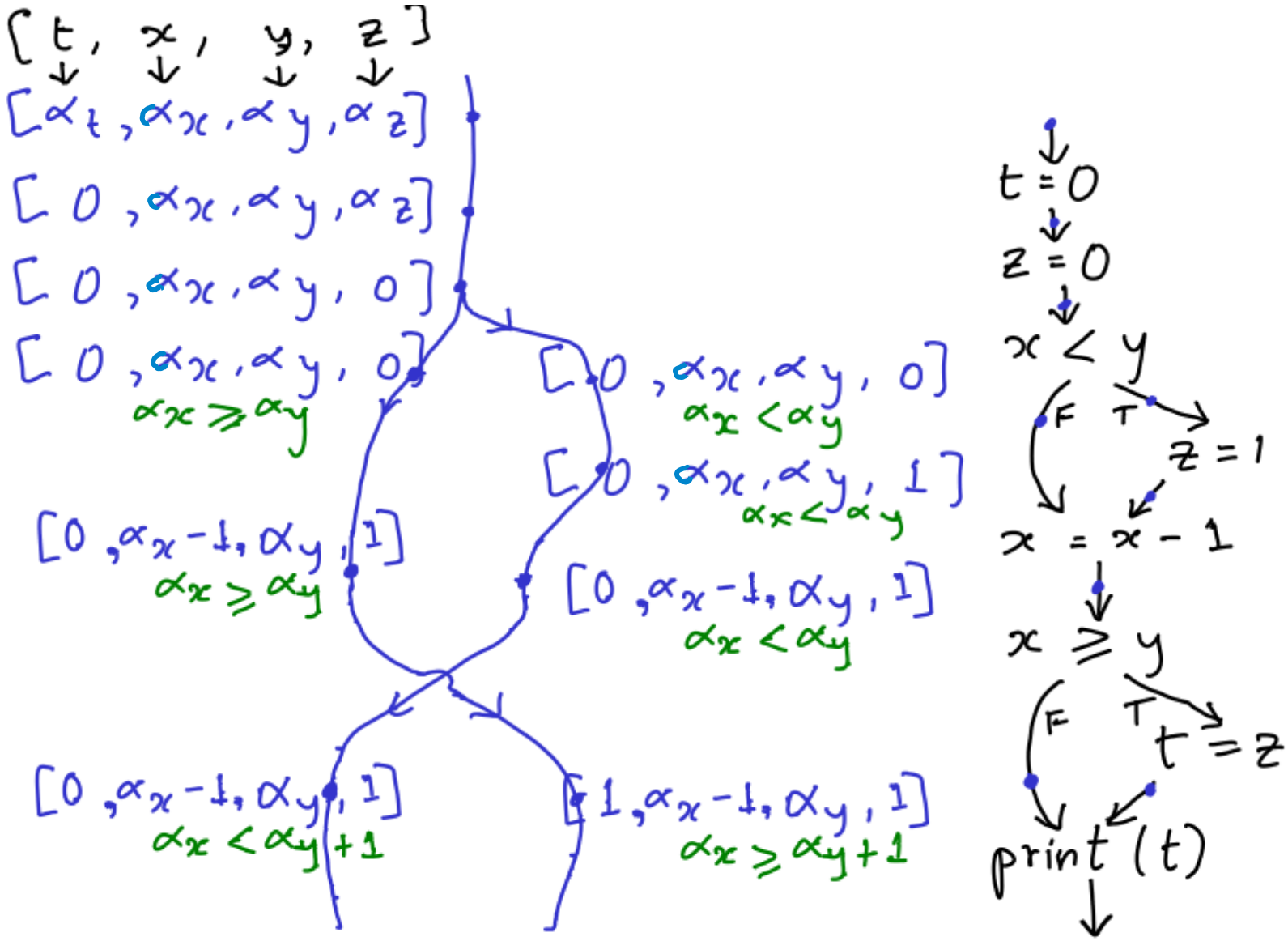
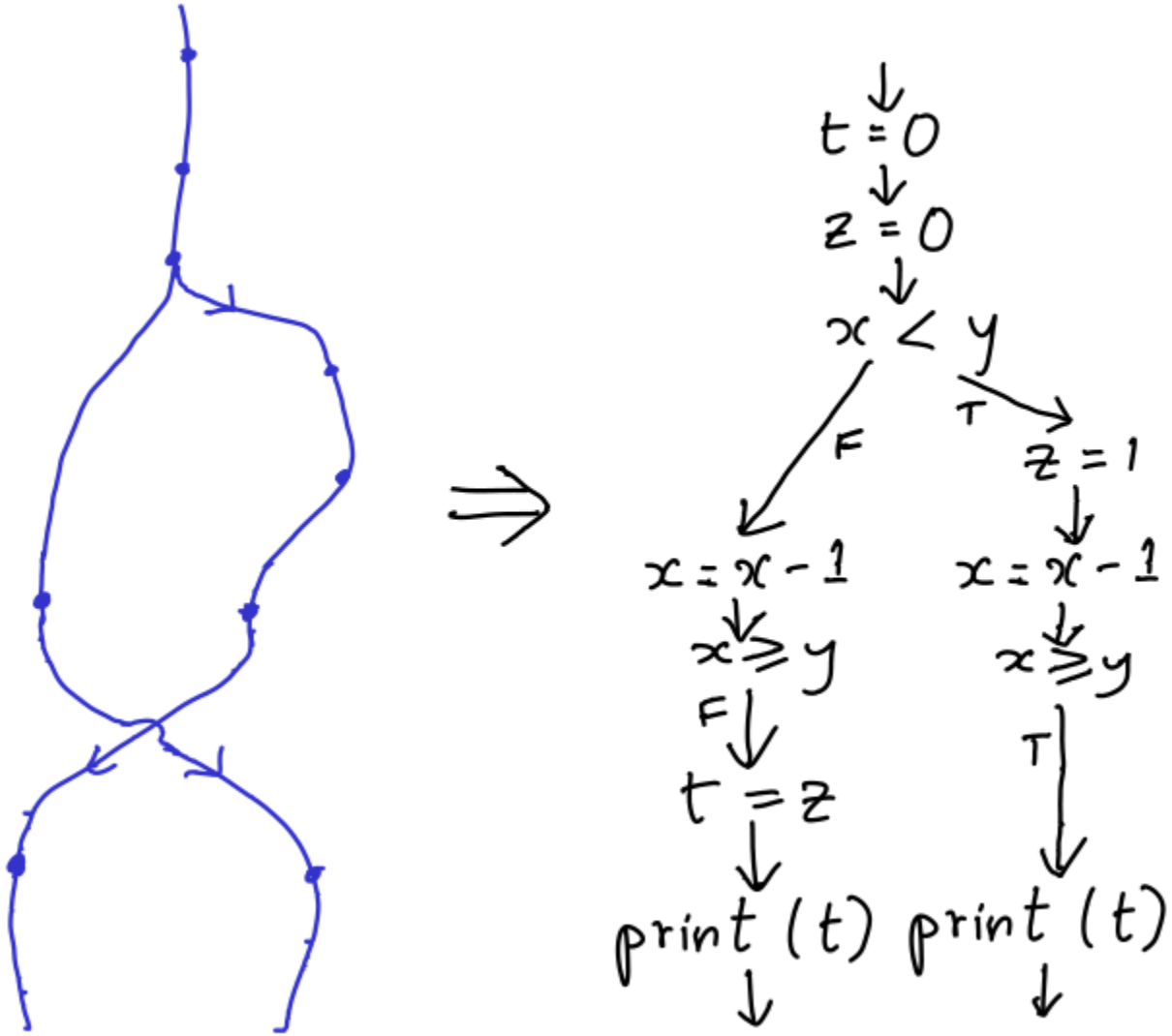


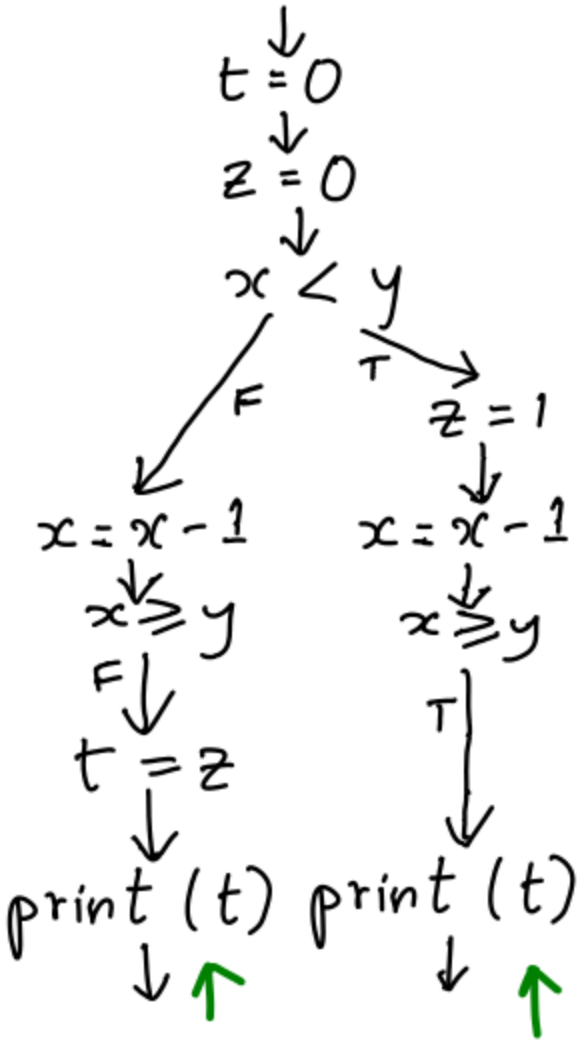
Illustration of symbolic execution



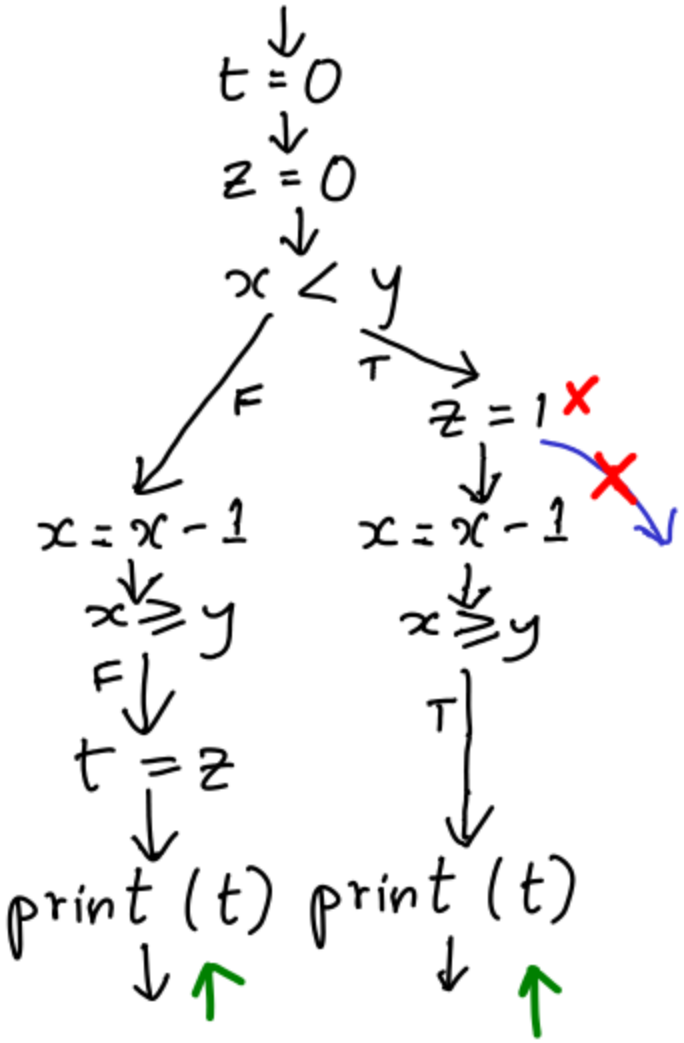
Symbolic paths ⟶ exploded flow graph



Now, perform standard slicing



Now, perform standard slicing



So what do we have …

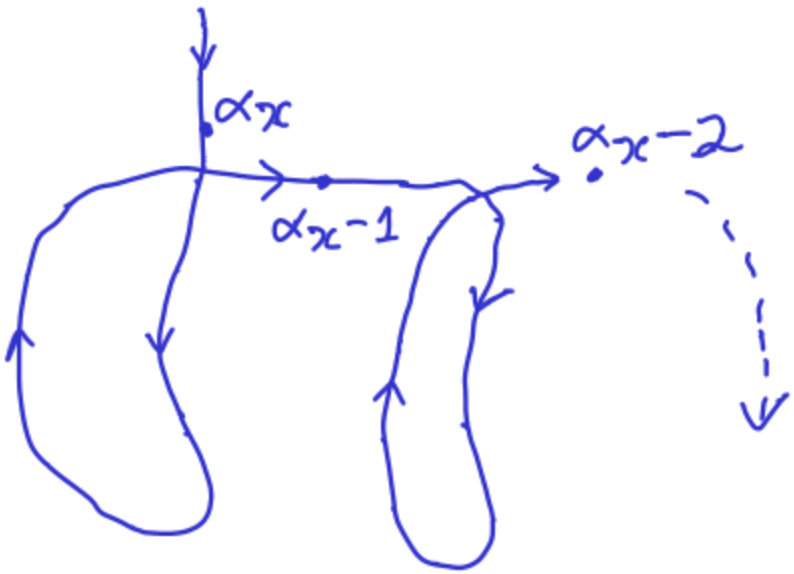
- Fully automated. Does not need partitioning *Q*.
- Precise even on examples like the complex one seen earlier (involving x = x + w; y = y + w;)
- However, problem with loops

The problem with loops

```
t = 0;
z = 0;
while (x < n) {
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```

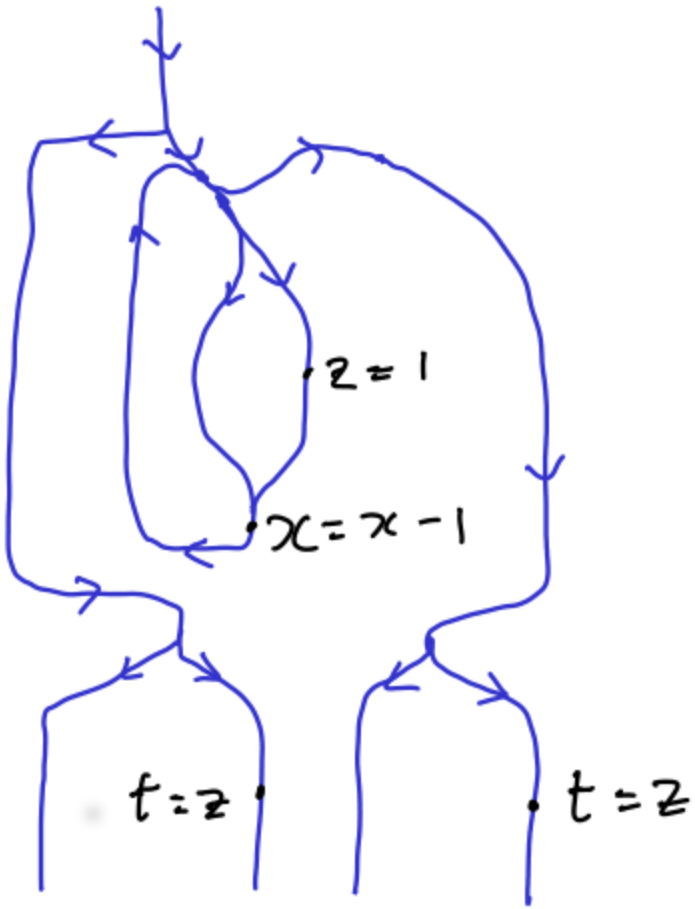
The problem with loops

```
t = 0;
z = 0;
while (x < n) {
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```



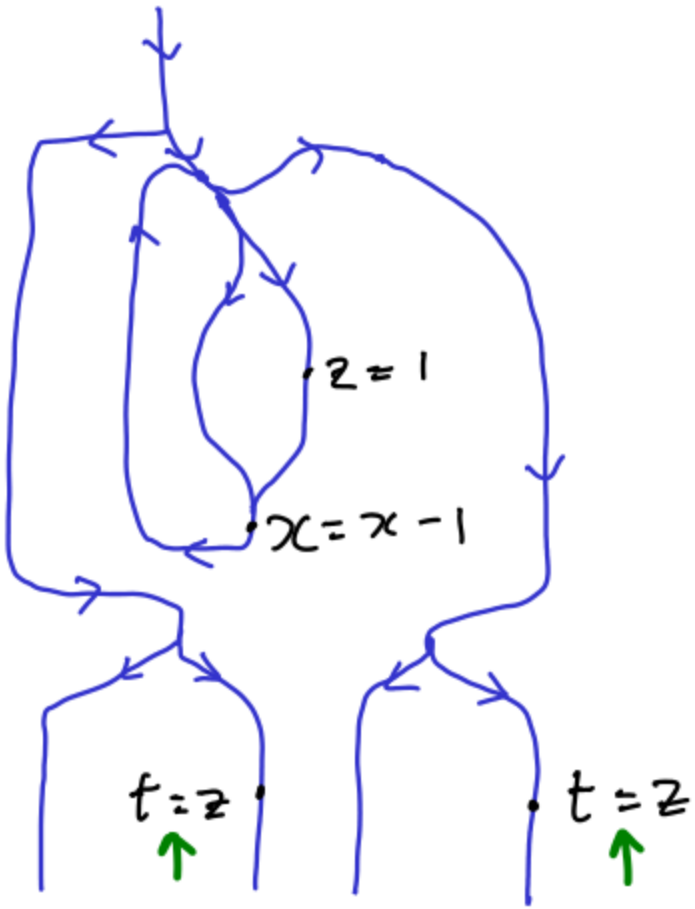
The exploded flow graph

```
t = 0;
z = 0;
while (x < n) {
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```



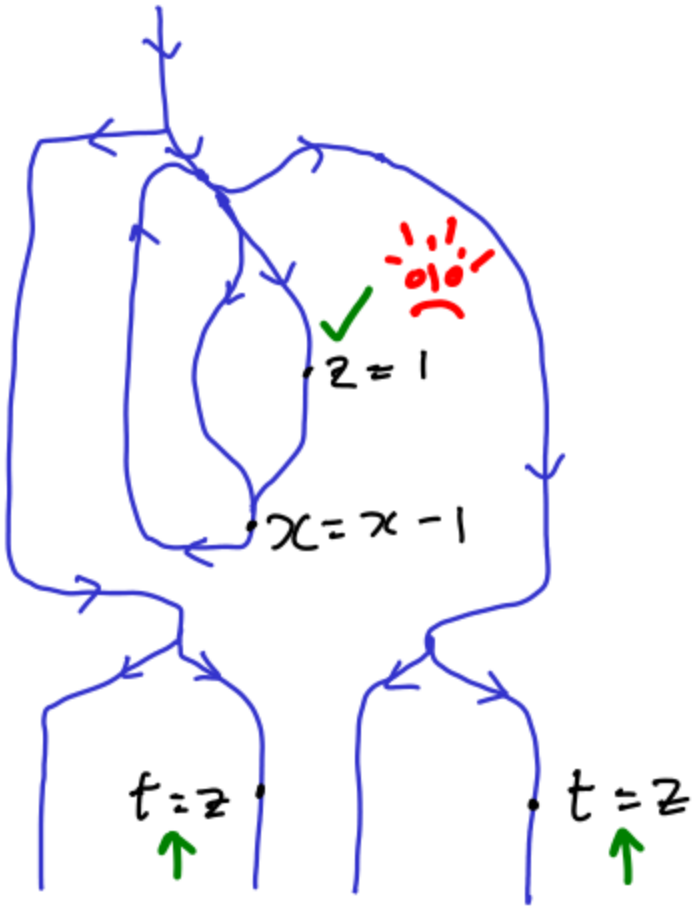
Slicing

```
t = 0;
z = 0;
while (x < n) {
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```



Imprecise slicing ☹

```
t = 0;
z = 0;
while (x < n) {
    if (x < y)
        z = 1;
    x = x - 1;
}
if (x ≥ y)
    t = z;
print (t);
```



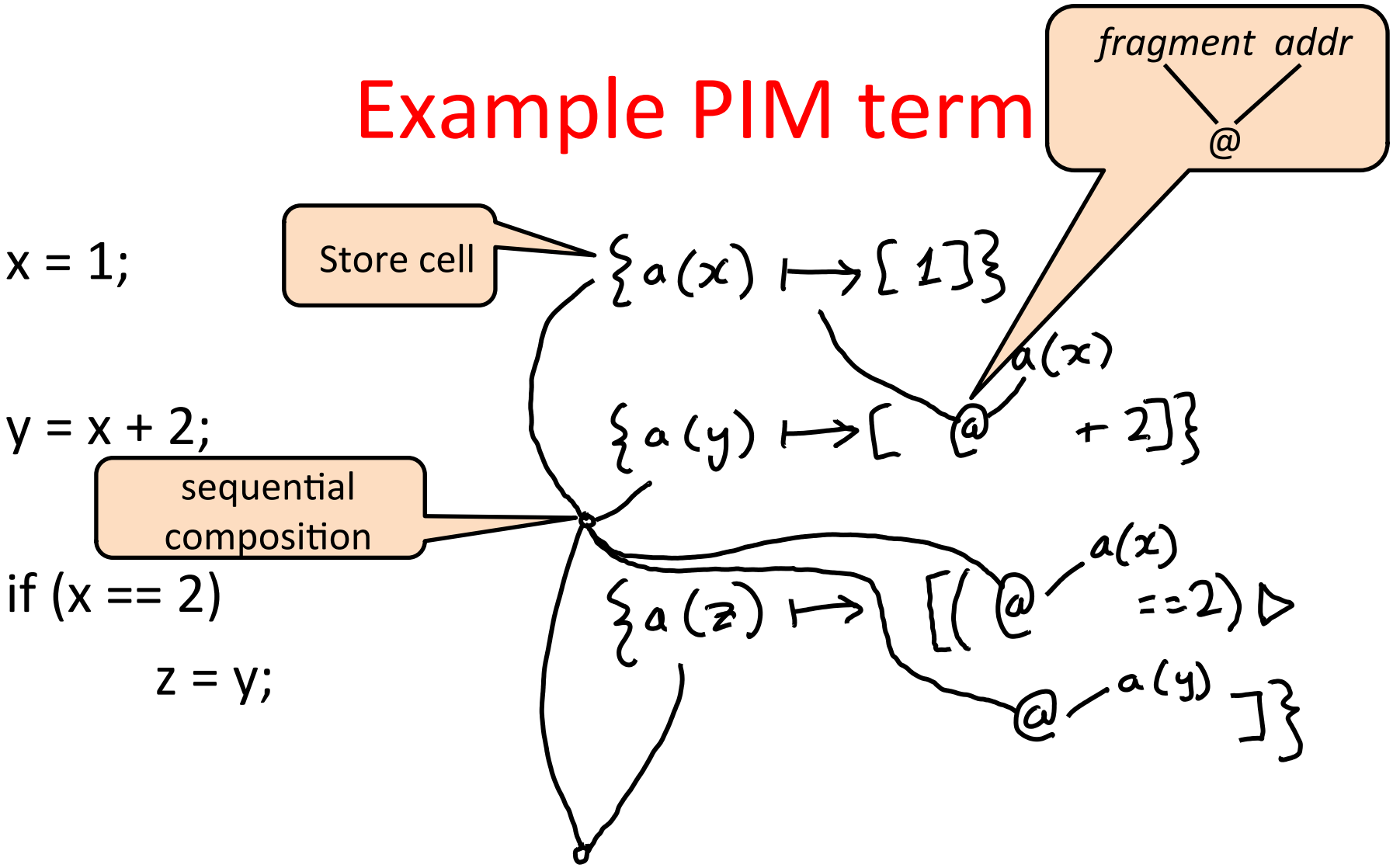
Our approach [Komondoor '13]

- Objectives
  - Fully precise in loop-free fragments, without relying on user-provided partitioning
  - Use user-provided partitioning only when "crossing" loop iterations
  - Handle programs that access and manipulate linked data structures

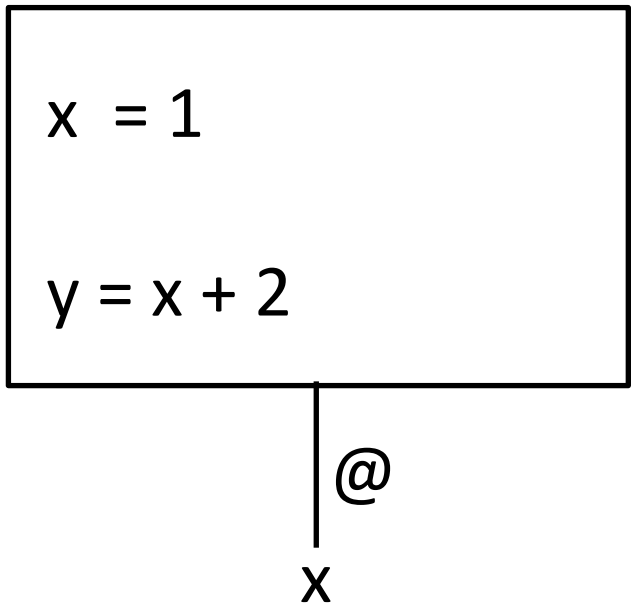
We use PIM

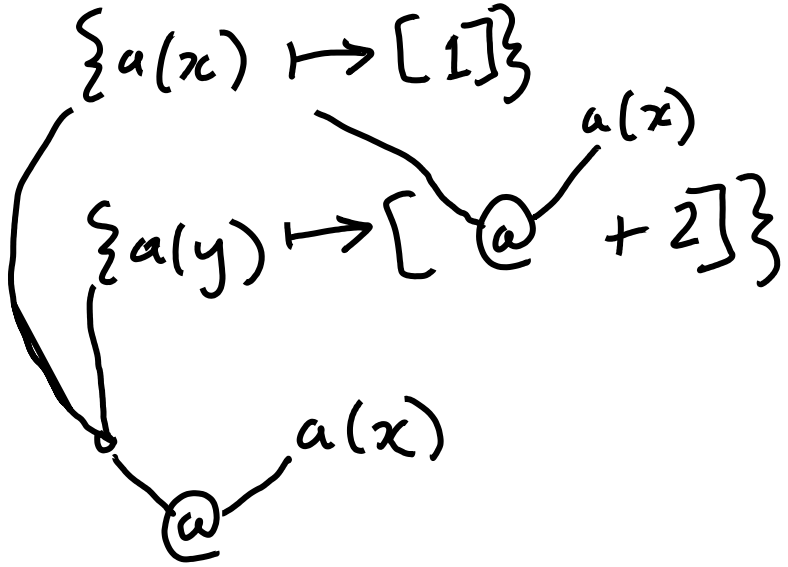
- What is PIM?
  - A graph/term representation for C programs
  - An equational logic and rewrite system on terms
    - Embodies the full concrete operational semantics of C
- Applications
  - Precise constrained slicing
  - Partial evaluation

Example PIM term

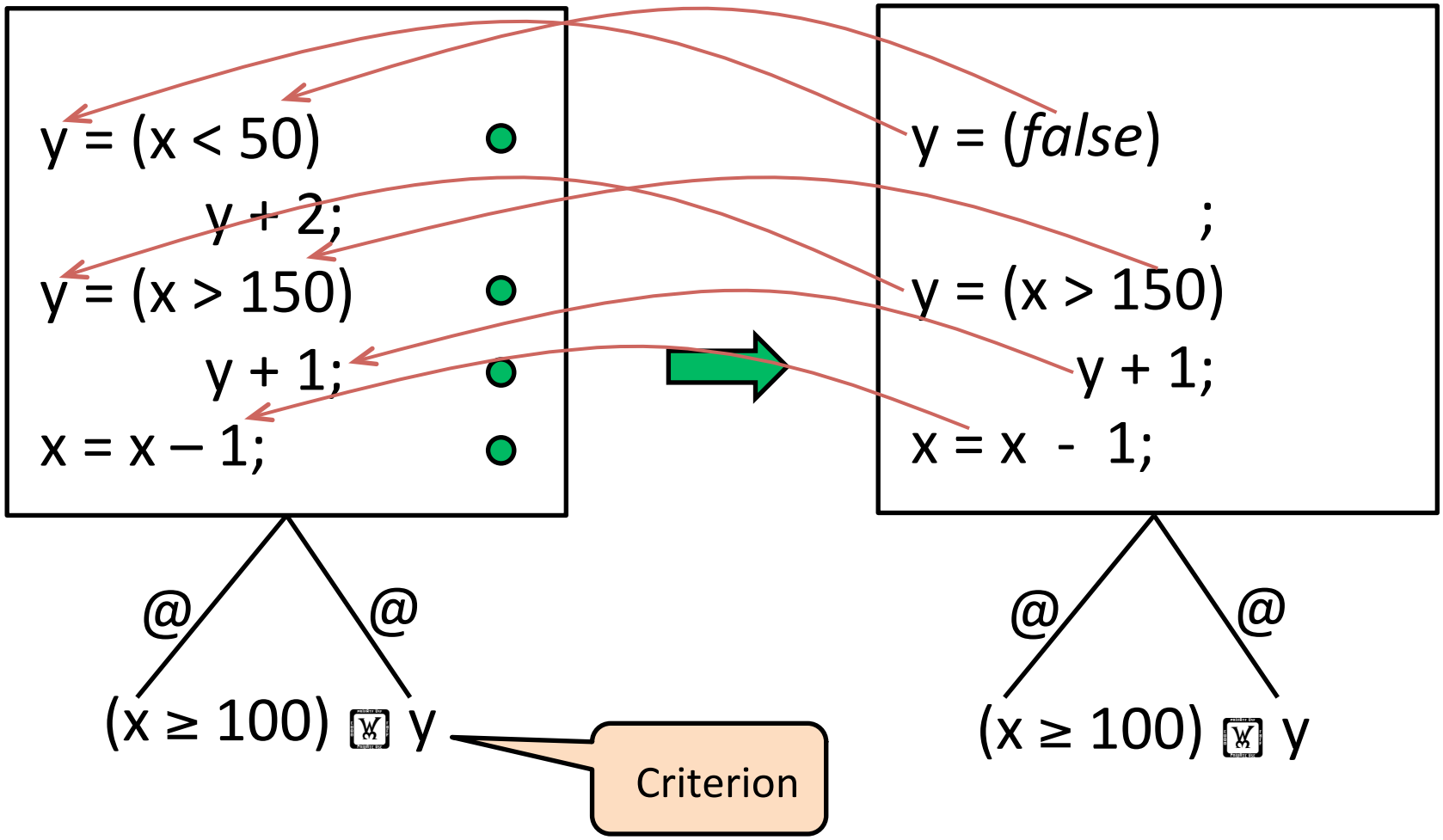


Our notation





Slicing via term simplification in PIM



Summary of PIM's approach

1. Convert the (program + criterion) into a store lookup
2. Rewrite/simplify the store lookup term
3. Identify subterms in the program on which simplified term is dependent
4. These terms constitute the slice

Fully precise in loop-free fragments. No partitioning required as input.

Slicing a loop

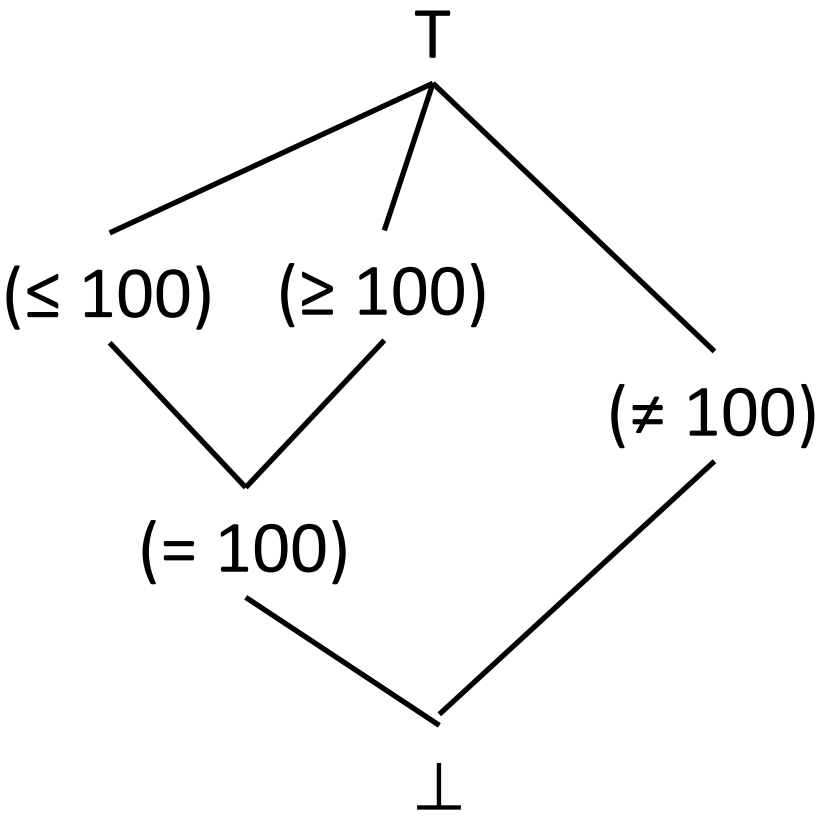
```
while (x > n) {
   y = (x < 50)
         y + 2;
   y = (x > 150)
         y + 1;
   x = x – 1;
}
```

PIM does not terminate while computing precise slice

(x = 100) ⊠ y

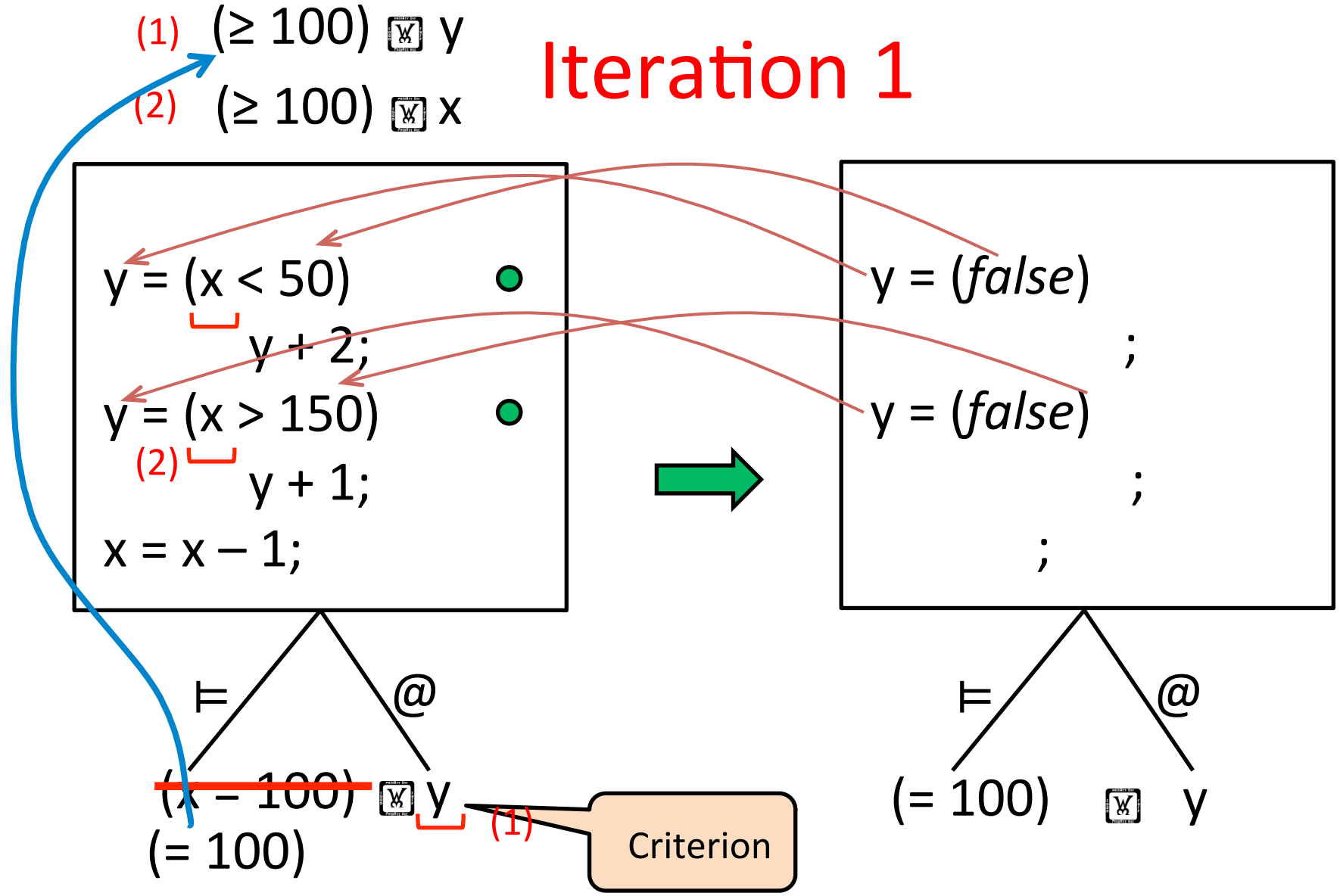
Criterion

Abstract lattice for given example



(Tracks only value of x)

Iteration 1

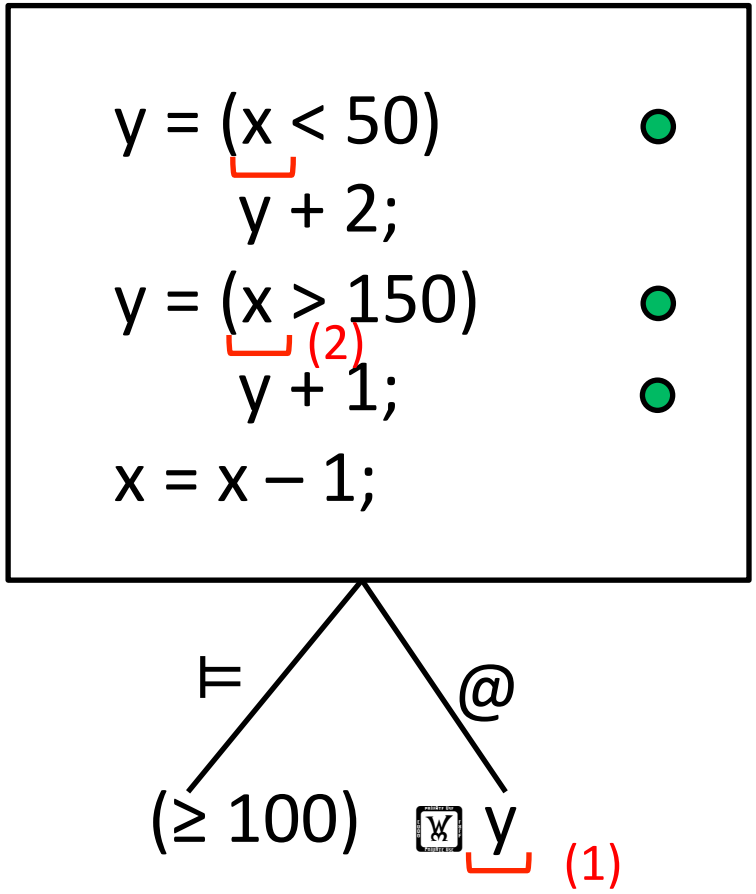


abstract weakest pre-condition

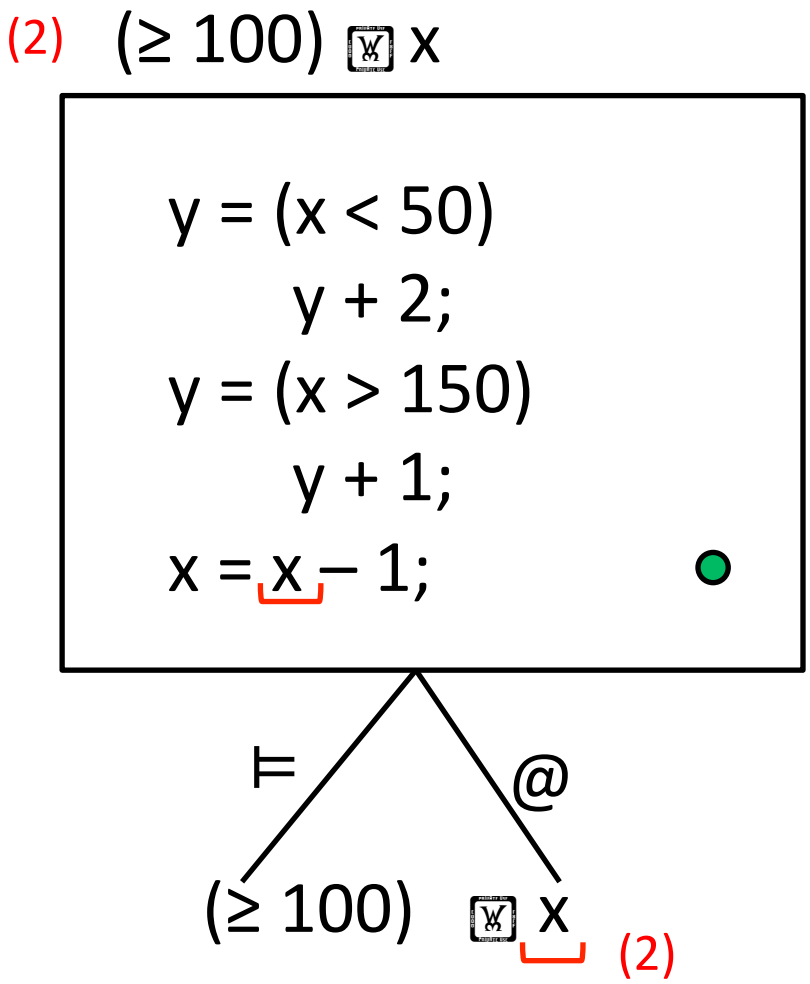
Iteration 2

(1) (≥ 100) ⋔ y

(2) (≥ 100) ⋔ x



Iteration 3



Final slice

```
while (x > n) {
   y = (x < 50)
         y + 2;
   y = (x > 150)
         y + 1;
   x = x – 1;
}
```

Our approach, at each iteration

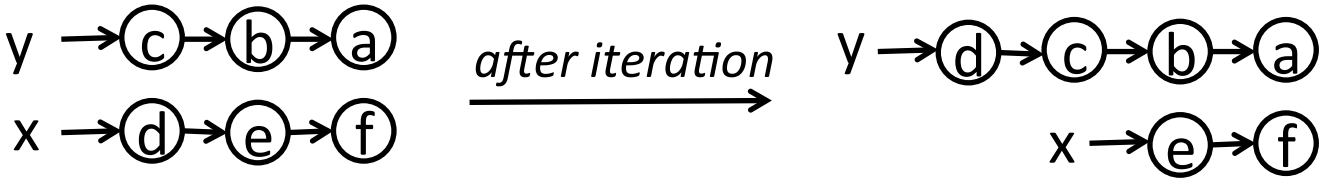
- Use abstract predicates, of the form `*s* ⊨ *l*', where *s* is a fragment and *l* is an element of a user-provided abstract lattice *L*
- Convert concrete guards in criteria to abstract guards at the beginning of each iteration
- Rewrite term using extended PIM rewrite rules
- Then, use dependences to obtain the slice

Ensuring termination

- If given lattice is finite
  - Assuming no heap, finite number of addresses.
  - Therefore, there is a bound on total number of possible abstract-guarded criteria.

- If lattice is finite-height
  - Whenever we generate a new criterion $c \equiv l$  v,
    If we had previously generated a criterion l'  v
    then modify c to $(l \sqcup l')$  v.
  - This also bounds the total number of possible criteria.

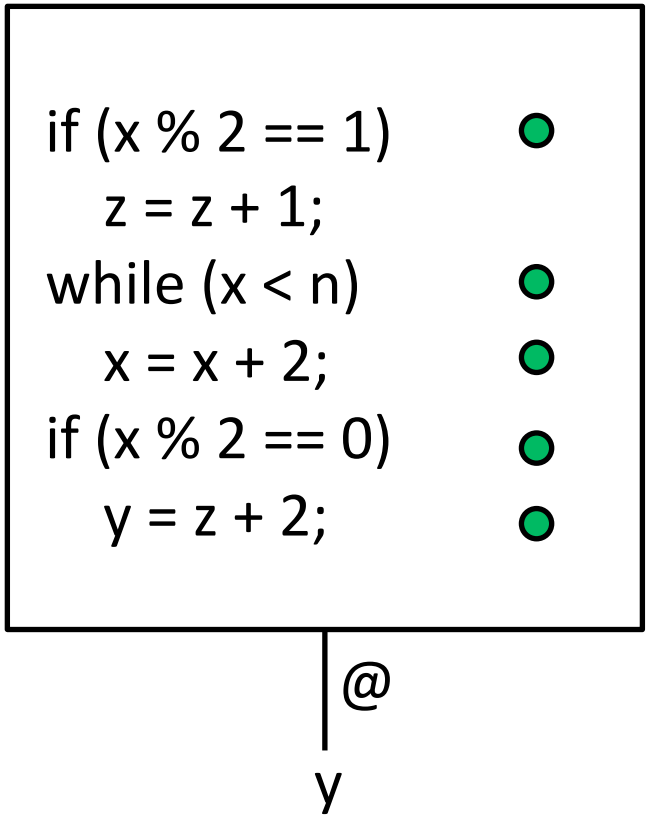
Example




```
// x  points to a singly-linked
// list
y = null;
while (x.d != k) {
   t = y;
   y = x;
   x = x.next;
   y.next = t;
}
```

@x

Another example



Summary of our approach

- Fully precise slicing in loop-free fragments
- Slicing of loops: Precision linked to user-provided lattice
- We address loops that traverse heap structures
- Support partial evaluation also
- Technical contribution
  - Integrate abstract interpretation with term rewriting
  - May be useful in other applications where term rewriting is used