CS 292C Computer-Aided Reasoning for Software

Lecture 14-15: Abstract Interpretation

Yu Feng

Fall 2019

Slides based on lectures by Isil Dillig

Outline of the lectures

- Basic concepts of abstract interpretation
    - Abstract domain
    - Abstract semantics
    - Lattice theory
    - Galois connection
- Fixed-point computation (next lecture)

One of the worst bugs in history

Ariane V self-destructing, 1996

```
x = computation_for_Ariane();
y = (short int) x ;
```

One of the worst bugs in history

Ariane V self-destructing, 1996

```
x = computation_for_Ariane();
y = (short int) x ;
```

Integer overflow

Airbus A380

Prove absence of bugs by Astree (2002–) (Cousot et al.).

http://www.astree.ens.fr/

Why is difficult to find bugs?

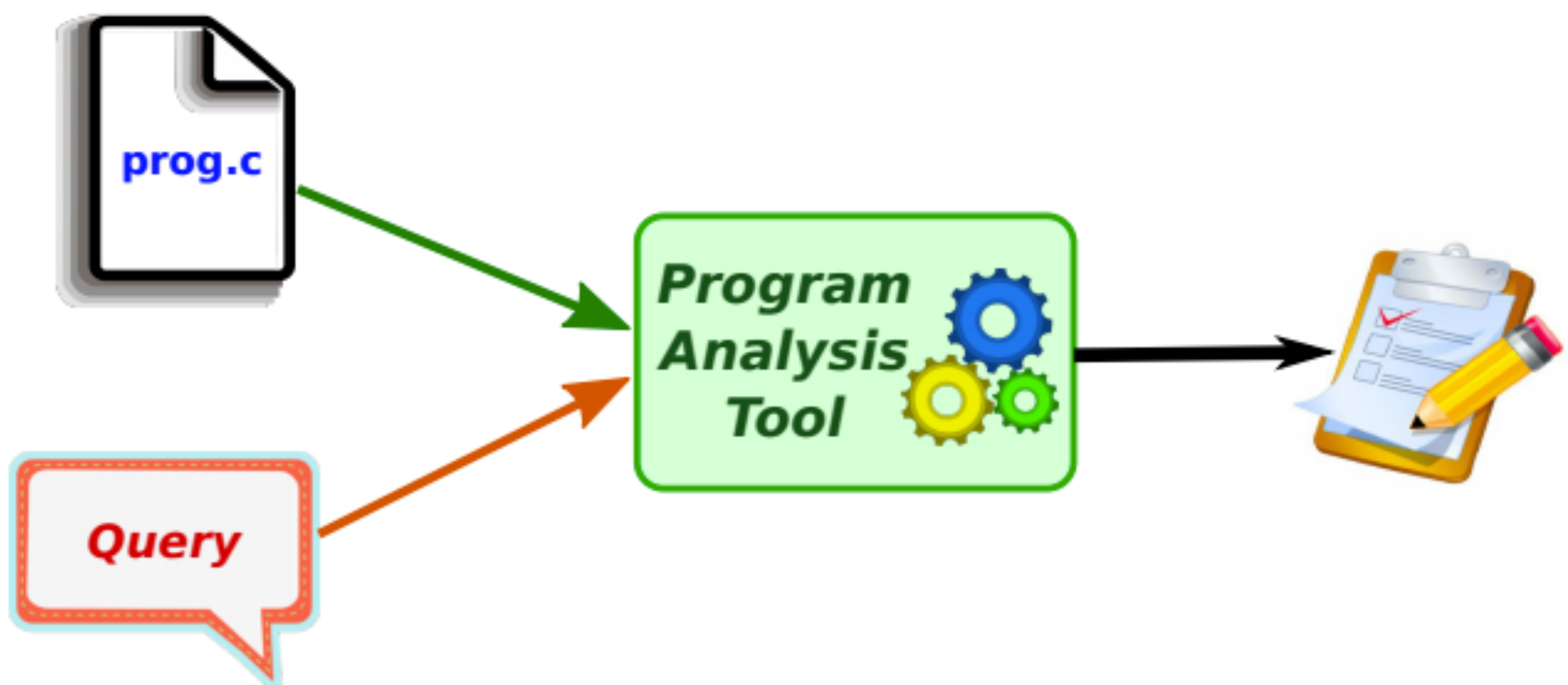
State explosion:

- We cannot represent the **concrete state space** X. Four 32-bit variables: 2^{128} states.
- Too large for explicit-state model-checking (need to memorize all states in memory)
- Also large for bounded model-checking (using clever structures e.g. BDDs).

What is program analysis

- Very broad topic, but generally speaking, **automated** analysis of program behavior

- Program analysis is about developing algorithms and tools that can analyze **other programs**



Applications of program analysis

Applications of program analysis

- **Bug finding**. e.g., expose as many assertion failures as possible

Applications of program analysis

- **Bug finding**. e.g., expose as many assertion failures as possible
- **Security**. e.g., does an app leak private user data?

Applications of program analysis

- **Bug finding**. e.g., expose as many assertion failures as possible
- **Security**. e.g., does an app leak private user data?
- **Verification**. e.g., does the program always behave according to its specification?

Applications of program analysis

- **Bug finding**. e.g., expose as many assertion failures as possible
- **Security**. e.g., does an app leak private user data?
- **Verification**. e.g., does the program always behave according to its specification?
- **Compiler optimizations**. e.g., which variables should be kept in registers for fastest memory access?

Applications of program analysis

- **Bug finding**. e.g., expose as many assertion failures as possible
- **Security**. e.g., does an app leak private user data?
- **Verification**. e.g., does the program always behave according to its specification?
- **Compiler optimizations**. e.g., which variables should be kept in registers for fastest memory access?
- **Automatic parallelization**. e.g., is it safe to execute different loop iterations on parallel?

Dynamic v.s. static program analysis

- Two flavors of program analysis:
  - Dynamic analysis: Analyzes program while it is running
  - Static analysis: Analyzes source code of the program

Static

+reason about all paths
- less precise

Dynamic

+more precise
- limited to finite executions

Static analysis

Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”

Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”

- But this question is **undecidable**!

Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”
- But this question is **undecidable**!
- This means static analyses are either:

Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”
- But this question is **undecidable**!
- This means static analyses are either:
    - **Unsound**: May say program is safe even though it is unsafe

Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”
- But this question is **undecidable**!
- This means static analyses are either:
  - **Unsound**: May say program is safe even though it is unsafe
  - **Sound, but incomplete**: May say program is unsafe even though it is safe

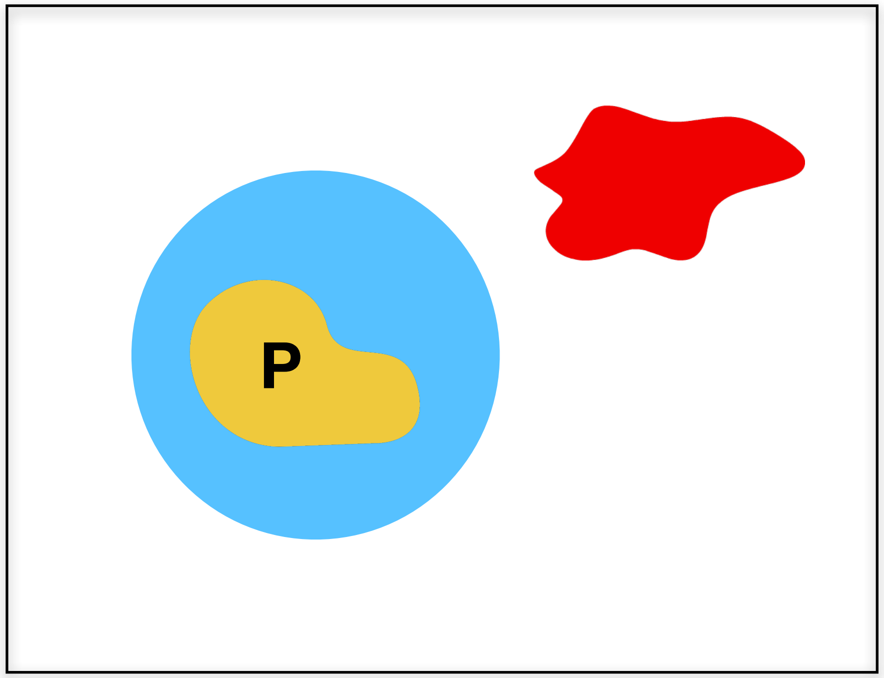
Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”
- But this question is **undecidable**!
- This means static analyses are either:
  - **Unsound**: May say program is safe even though it is unsafe
  - **Sound, but incomplete**: May say program is unsafe even though it is safe
  - **Non-terminating**: Always gives correct answer when it terminates, but may run forever

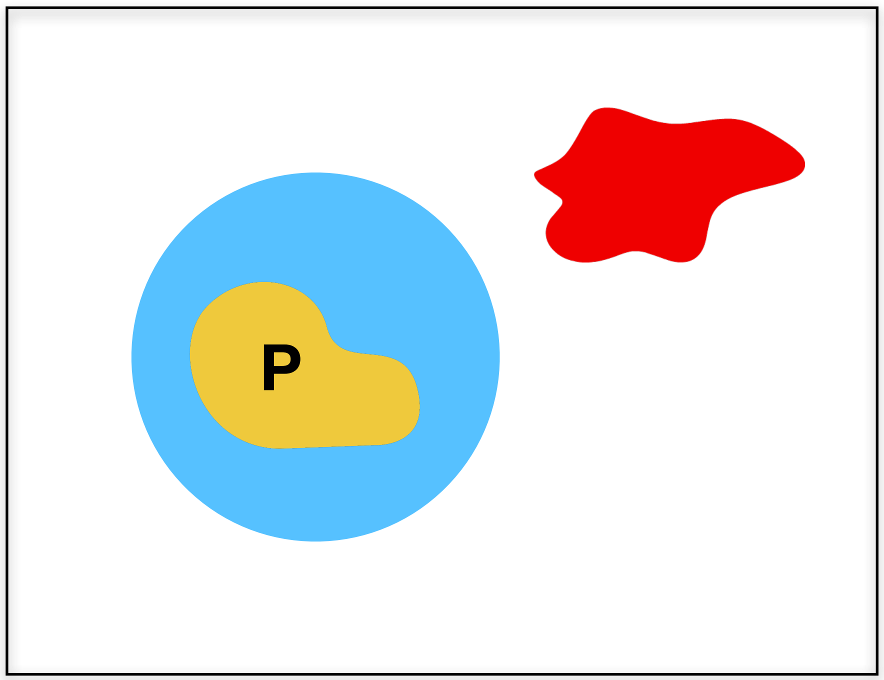
Static analysis

- Typical static analysis question: “Given source code of program P and desired property Q, does P exhibit Q in **all possible executions**?”
- But this question is **undecidable**!
- This means static analyses are either:
  - **Unsound**: May say program is safe even though it is unsafe
  - **Sound, but incomplete**: May say program is unsafe even though it is safe
  - **Non-terminating**: Always gives correct answer when it terminates, but may run forever
- Many static analysis techniques are sound but incomplete.

How to design sound static analysis

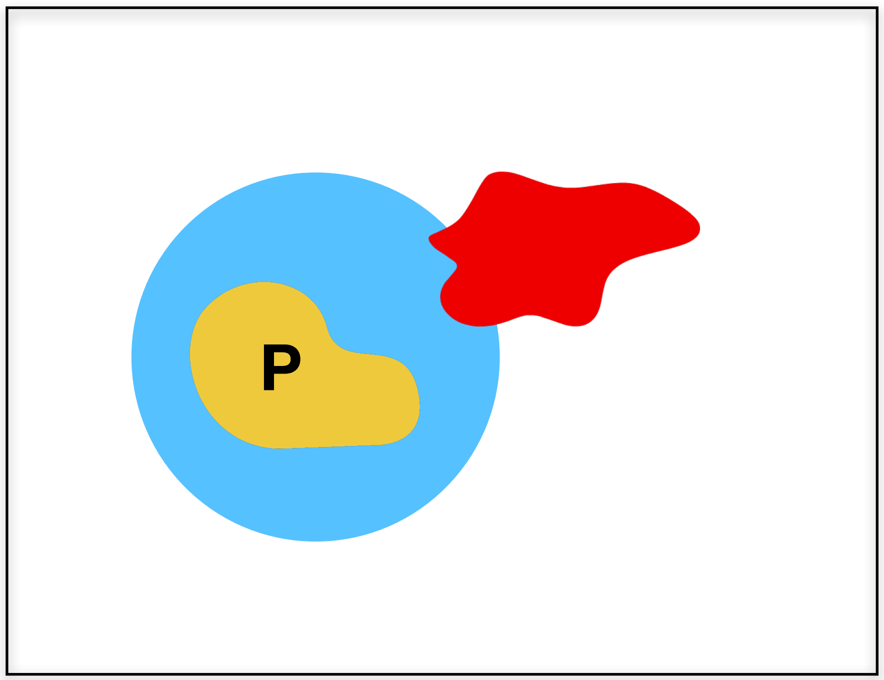


How to design sound static analysis



- Bad states outside over-approximation
  $\Rightarrow$ Program safe

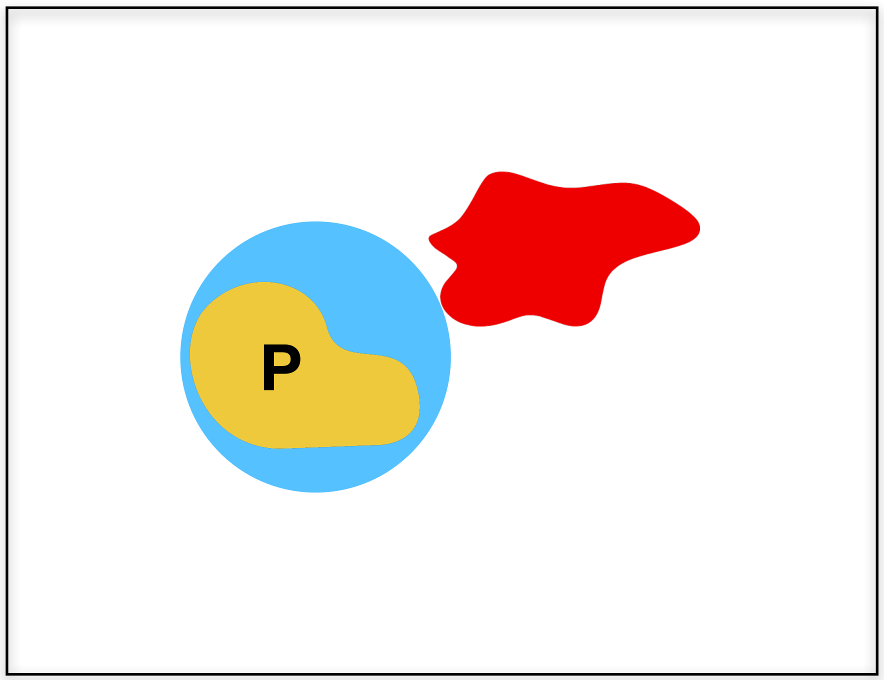
How to design sound static analysis



- Bad states outside over-approximation
  ⇒ Program safe

- Bad states inside over-approximation, but outside P ⇒ false alarm

How to design sound static analysis



- Bad states outside over-approximation
  ⇒ Program safe
- Bad states inside over-approximation, but outside P ⇒ false alarm
- **Goal**: Construct abstractions that are **precise** enough (i.e., few false alarms) and that **scale** to real programs

Examples of abstractions

There is no “one size fits all” abstraction

- What information is useful depends on what you want to prove about the program!

Examples of abstractions

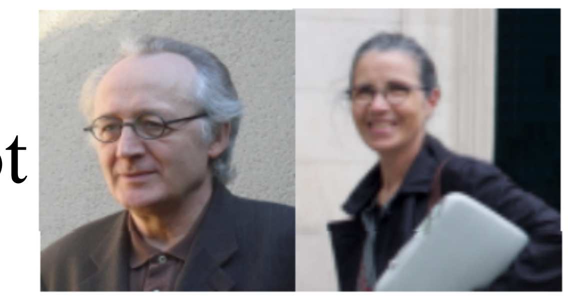
There is no "one size fits all" abstraction

- What information is useful depends on what you want to prove about the program!

Application	Possible abstraction
No division-by-zero errors	zero vs. non-zero
Data structure verification	list, tree, graph, . . .
No out-of-bounds array accesses	ranges of integer variables

How to design sound abstracts

- Useful theory for understanding how to design sound static analyses is **abstract interpretation**
  - Seminal '77 paper by Patrick & Radhia Cousot



- Not a specific analysis, but rather a framework for designing sound-by-construction static analyses
- Let's look at an example: A static analysis that tracks the sign of each integer variable (e.g., positive, non-negative, zero etc.)

Step 1: Design an abstract domain

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
    - pos: $\{x \mid x \in Z \land x > 0\}$

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
  - pos: $\{x \mid x \in Z \wedge x > 0\}$
  - zero: $\{0\}$

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
  - pos: $\{x \mid x \in Z \wedge x > 0\}$
  - zero: $\{0\}$
  - neg: $\{x \mid x \in Z \wedge x < 0\}$

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
  - pos: $\{x \mid x \in Z \wedge x > 0\}$
  - zero: $\{0\}$
  - neg: $\{x \mid x \in Z \wedge x < 0\}$
  - non-neg: $\{x \mid x \in Z \wedge x \geq 0\}$

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
  - pos: $\{x \mid x \in Z \wedge x > 0\}$
  - zero: $\{0\}$
  - neg: $\{x \mid x \in Z \wedge x < 0\}$
  - non-neg: $\{x \mid x \in Z \wedge x \geq 0\}$
- In addition, every abstract domain contains:

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
  - pos: $\{x \mid x \in Z \wedge x > 0\}$
  - zero: $\{0\}$
  - neg: $\{x \mid x \in Z \wedge x < 0\}$
  - non-neg: $\{x \mid x \in Z \wedge x \geq 0\}$
- In addition, every abstract domain contains:
  - $\top$ (top): "Don't know", represents any value

Step 1: Design an abstract domain

- An abstract domain is just a set of abstract values we want to track in our analysis
- For our example, let's fix the following abstract domain:
  - pos: $\{x \mid x \in Z \land x > 0\}$
  - zero: $\{0\}$
  - neg: $\{x \mid x \in Z \land x < 0\}$
  - non-neg: $\{x \mid x \in Z \land x \geq 0\}$
- In addition, every abstract domain contains:
  - $\top$ (top): "Don't know", represents any value
  - $\bot$ (bottom): Represents empty-set

Step 2: Abstraction and concretization functions

Step 2: Abstraction and concretization functions

- Abstraction function (α) maps sets of concrete elements to the most precise value in the abstract domain

Step 2: Abstraction and concretization functions

- Abstraction function (α) maps sets of concrete elements to the most precise value in the abstract domain
    - $\alpha(\{2, 10, 0\})$ = non-neg

Step 2: Abstraction and concretization functions

- Abstraction function (α) maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\})$ = non-neg
  - $\alpha(\{3, 99\})$ = pos

Step 2: Abstraction and concretization functions

- Abstraction function (α) maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\})$ = non-neg
  - $\alpha(\{3, 99\})$ = pos
  - $\alpha(\{-3, 2\}) = \top$

Step 2: Abstraction and concretization functions

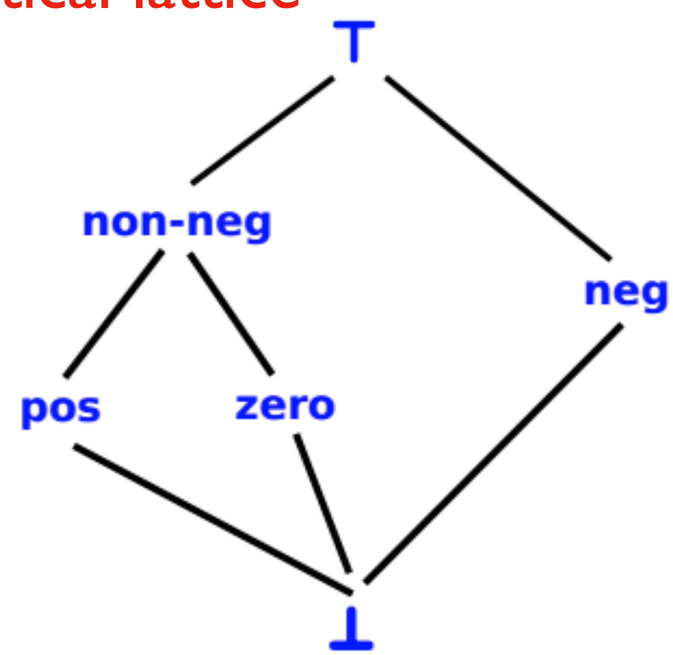
- Abstraction function (α) maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\})$ = non-neg
  - $\alpha(\{3, 99\})$ = pos
  - $\alpha(\{-3, 2\}) = \top$
- Concretization function (γ) maps each abstract value to sets of concrete elements

Step 2: Abstraction and concretization functions

- Abstraction function (α) maps sets of concrete elements to the most precise value in the abstract domain
  - $\alpha(\{2, 10, 0\})$ = non-neg
  - $\alpha(\{3, 99\})$ = pos
  - $\alpha(\{-3, 2\}) = \top$
- Concretization function (γ) maps each abstract value to sets of concrete elements
  - $\gamma(\text{pos}) = \{x \mid x \in Z \wedge x > 0\}$

Lattices and abstract domains

- Concretization function defines partial order on abstract values: $A_1 \leq A_2$ iff $\gamma(A_1) \subseteq \gamma(A_2)$
- Furthermore, in an abstract domain, every pair of elements has a lub and glb $\Rightarrow$ mathematical lattice

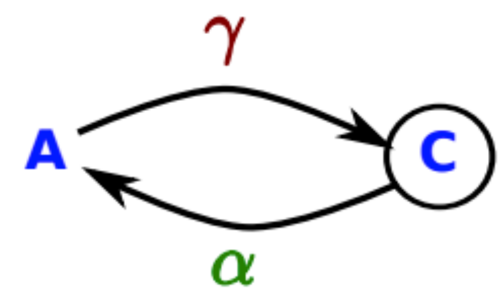


- Least upper bound of two elements is called their **join** – useful for reasoning about control flow in programs

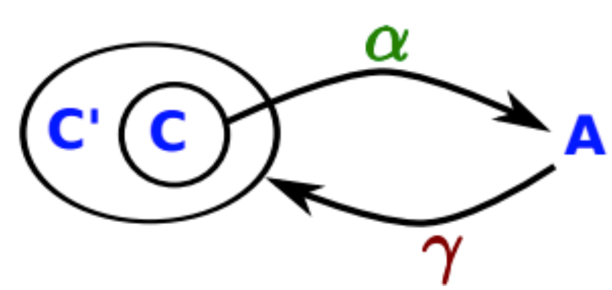
Galois connection

- Important property of the abstraction and concretization function is that they are almost inverses via Galois connection:

$$\alpha(\gamma(A)) = A$$



$$C \subseteq \gamma(\alpha(C))$$

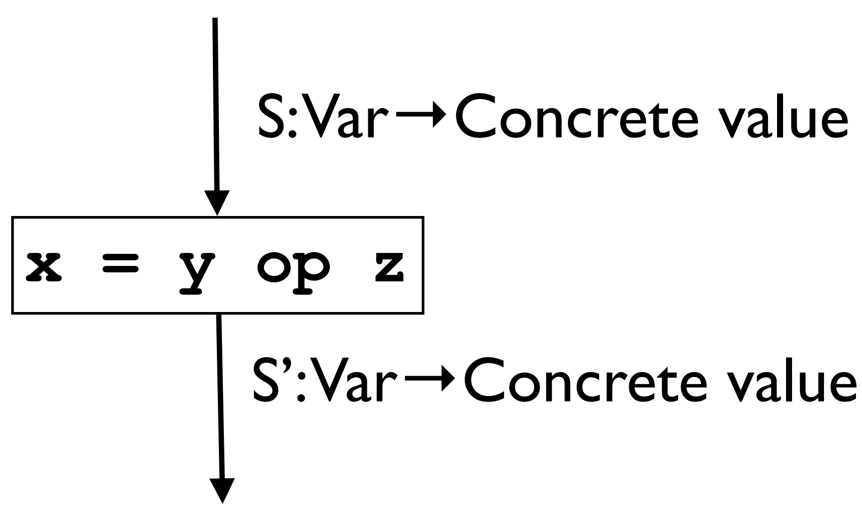


- This is called a Galois connection that captures the soundness of the abstraction

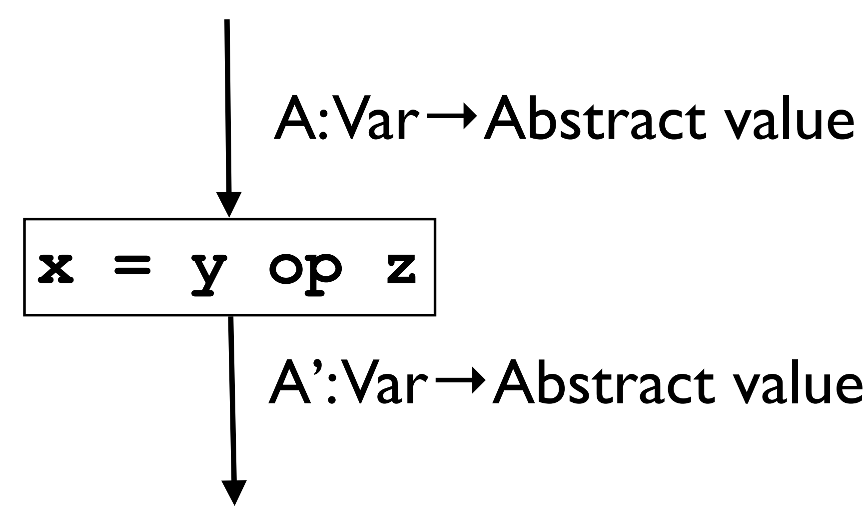
Step 3: Abstract semantics

- Given abstract domain, α, γ, need to define abstract transformers (i.e., semantics) for each statement
  - Describes how statements affect our abstraction
  - Abstract counter-part of operational semantics rules

Operational semantics



Abstract semantics



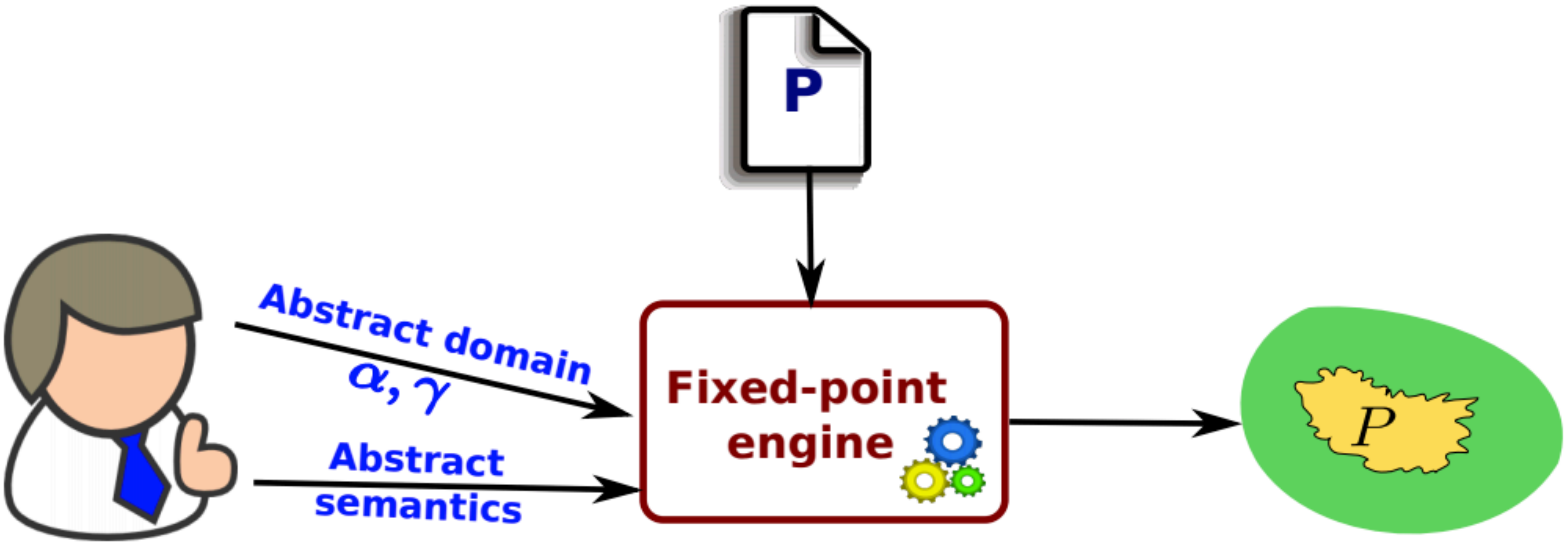
Back to our running example

- For our sign analysis, we can define abstract transformer for x = y + z as follows:

	pos	neg	zero	non-neg	$\top$	$\bot$
pos	pos	$\top$	pos	pos	$\top$	$\bot$
neg	$\top$	neg	neg	$\top$	$\top$	$\bot$
zero	pos	neg	zero	non-neg	$\top$	$\bot$
non-neg	pos	$\top$	non-neg	non-neg	$\top$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

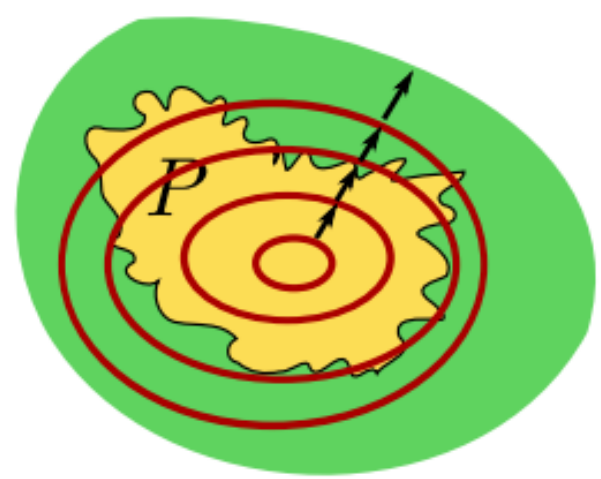
- To ensure soundness of static analysis, must prove that abstract semantics faithfully models concrete semantics

Put it all together



Fixed-point computation

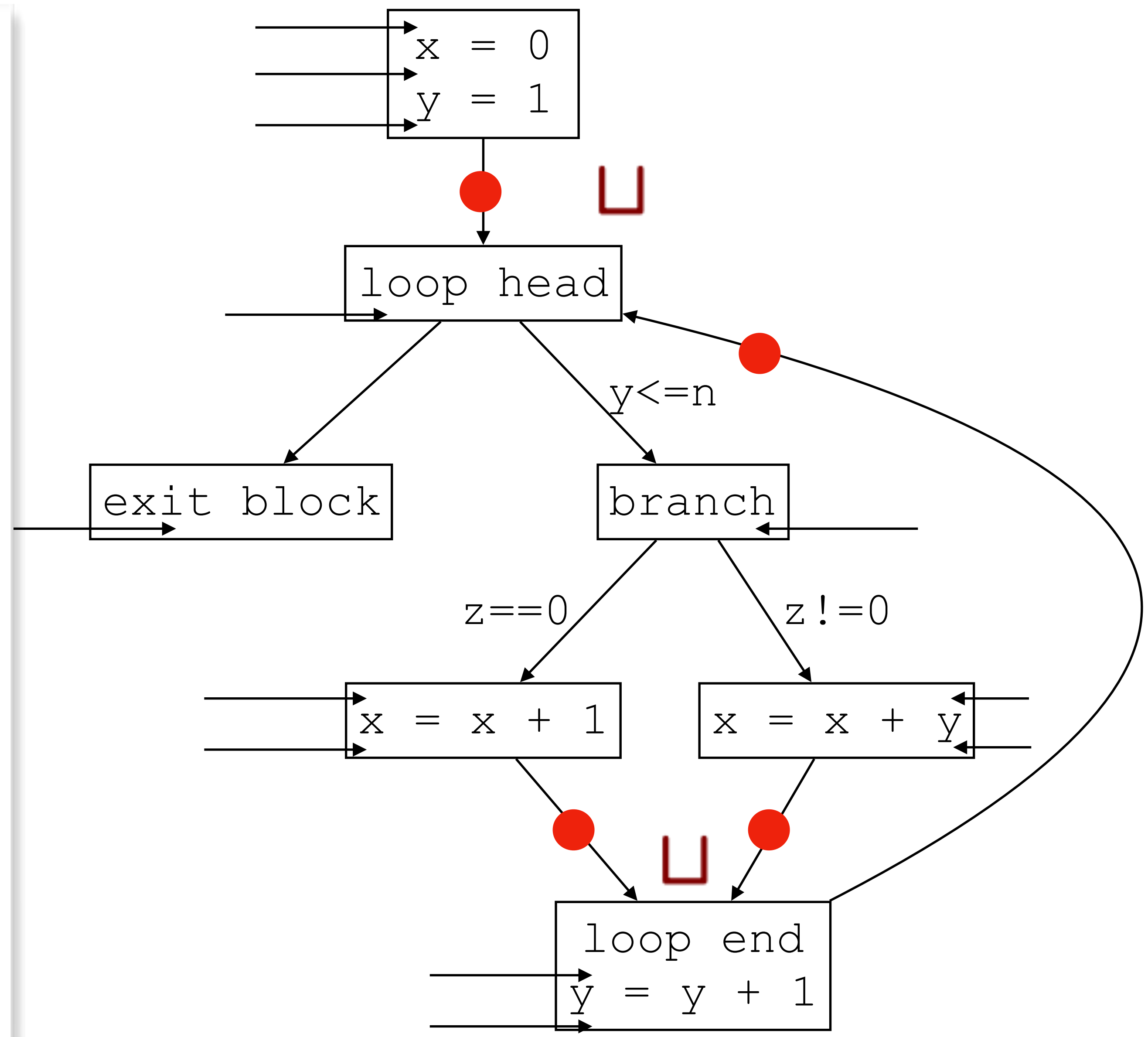
- **Fixed-point computation**: Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an equilibrium
- **Least fixed-point**: Start with under-approximation and grow the approximation until it stops growing



- Assuming correctness of your abstract semantics, the least fixed point is an **over-approximation** of the program!

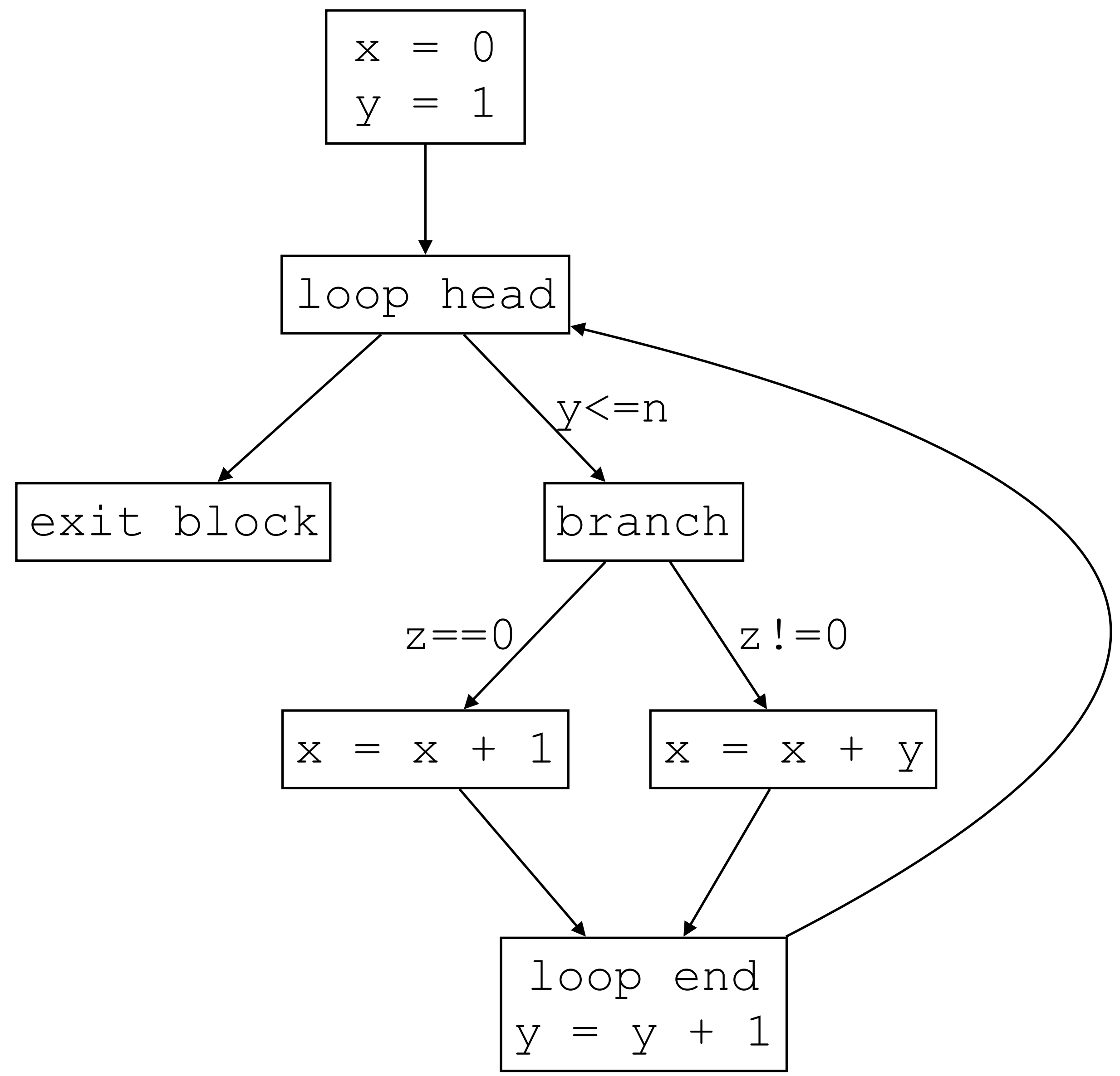
Least fixed-point computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to ⊥
- Repeat until no abstract state changes at any program point:
  - Compute abstract state on entry to a basic block B by taking the join of B's predecessors
  - Symbolically execute each basic block using abstract semantics



Fixed-point computation

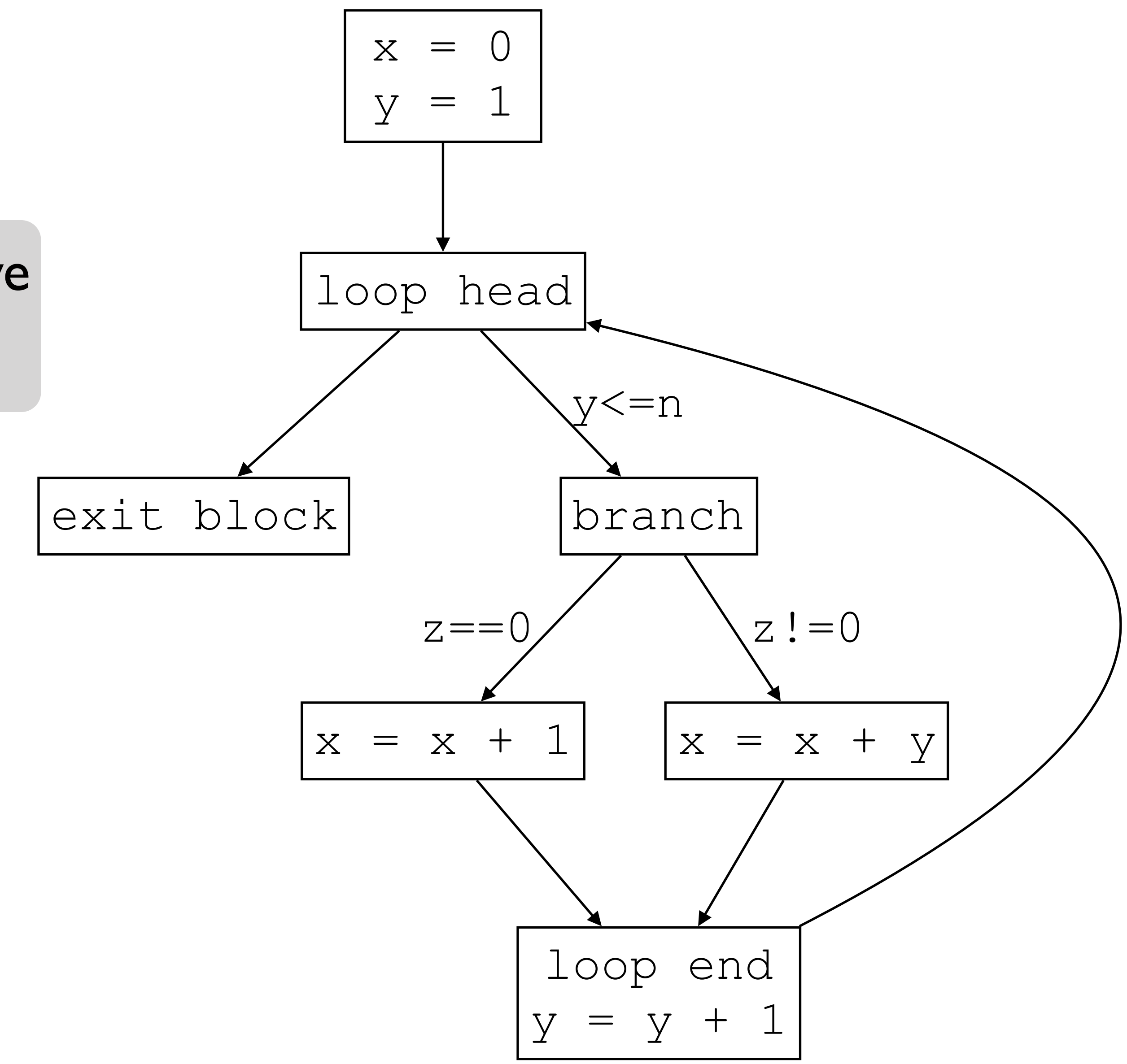
```
x = 0;
y = 1;
while(y <= n) {
 if (z == 0) {
   x = x + 1;
 } else {
   x = foo(x,y);
 }
 y = y + 1;
}
```



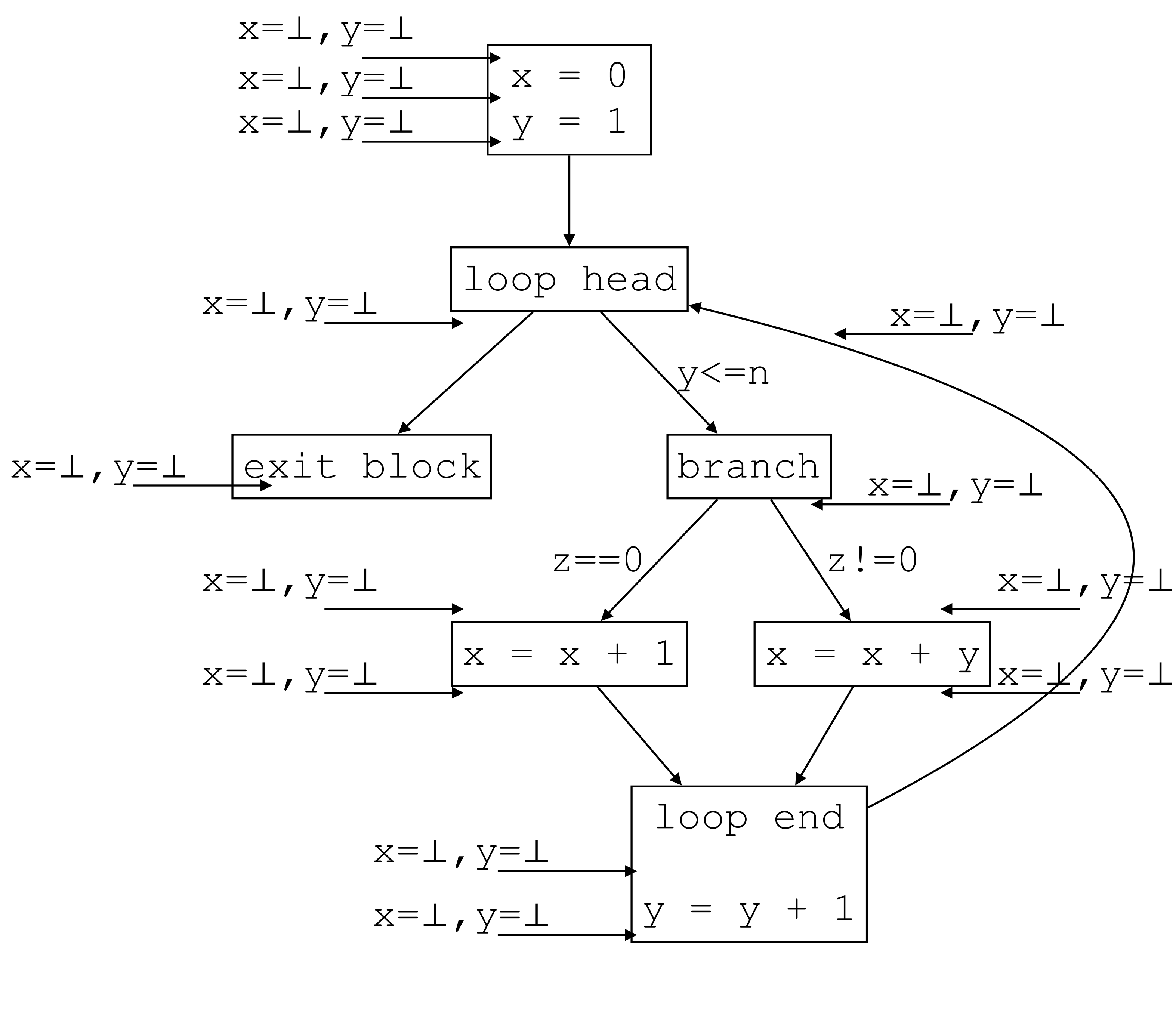
Fixed-point computation

```
x = 0;
y = 1;
while(y <= n) {
 if (z == 0) {
  x = x + 1;
 } else {
  x = foo(x,y);
 }
 y = y + 1;
}
```

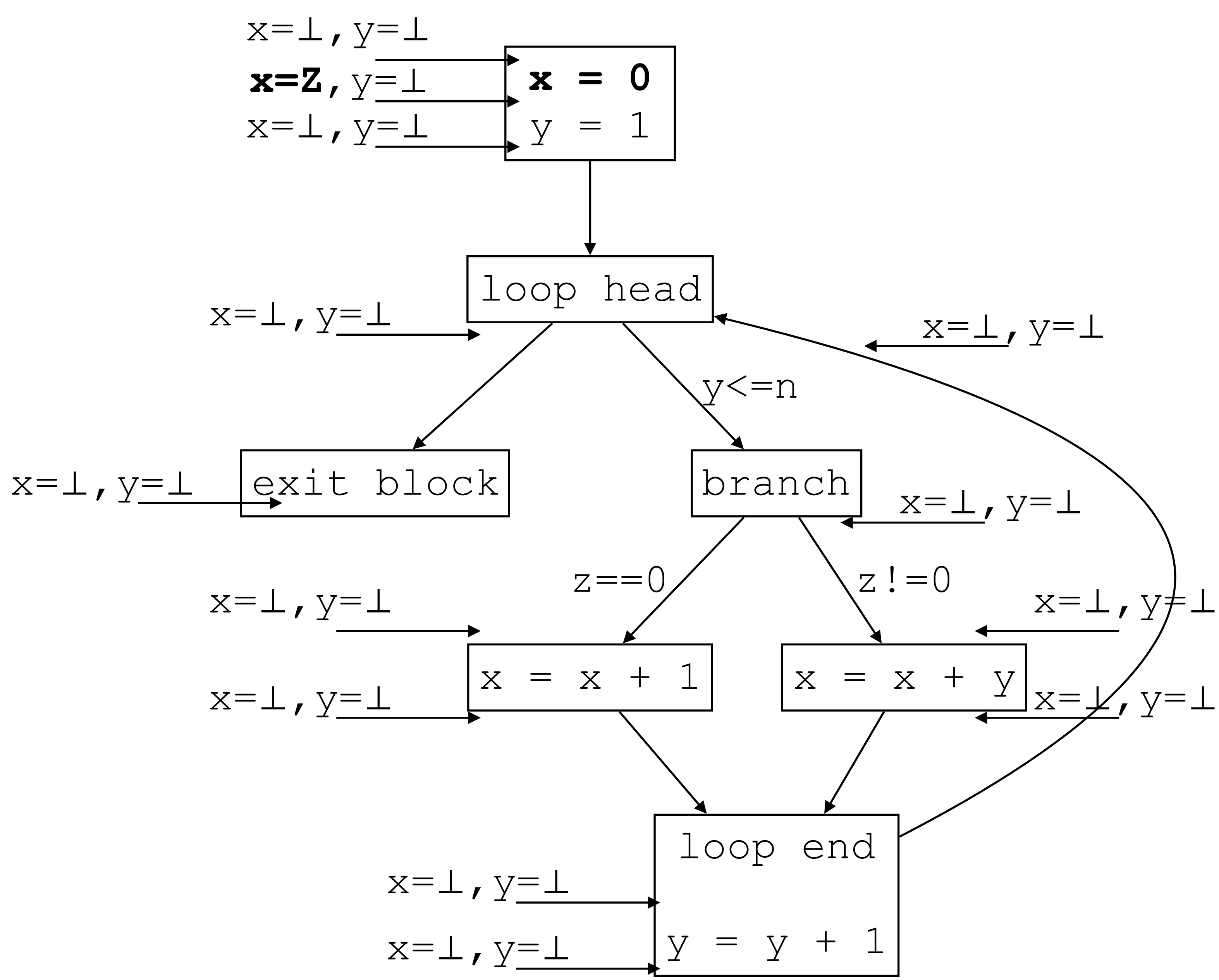
Is x always non-negative inside the loop?



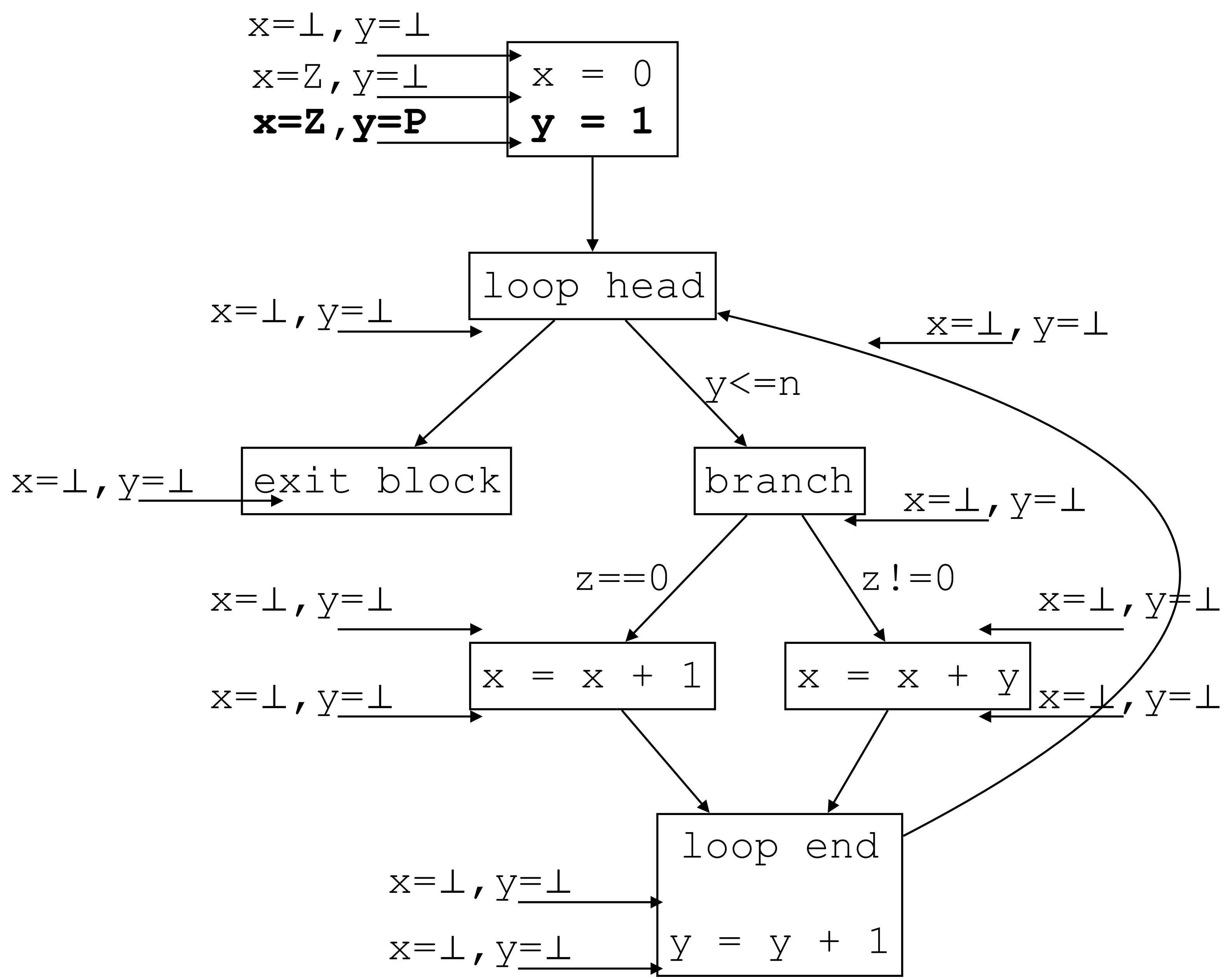
Fixed-point computation



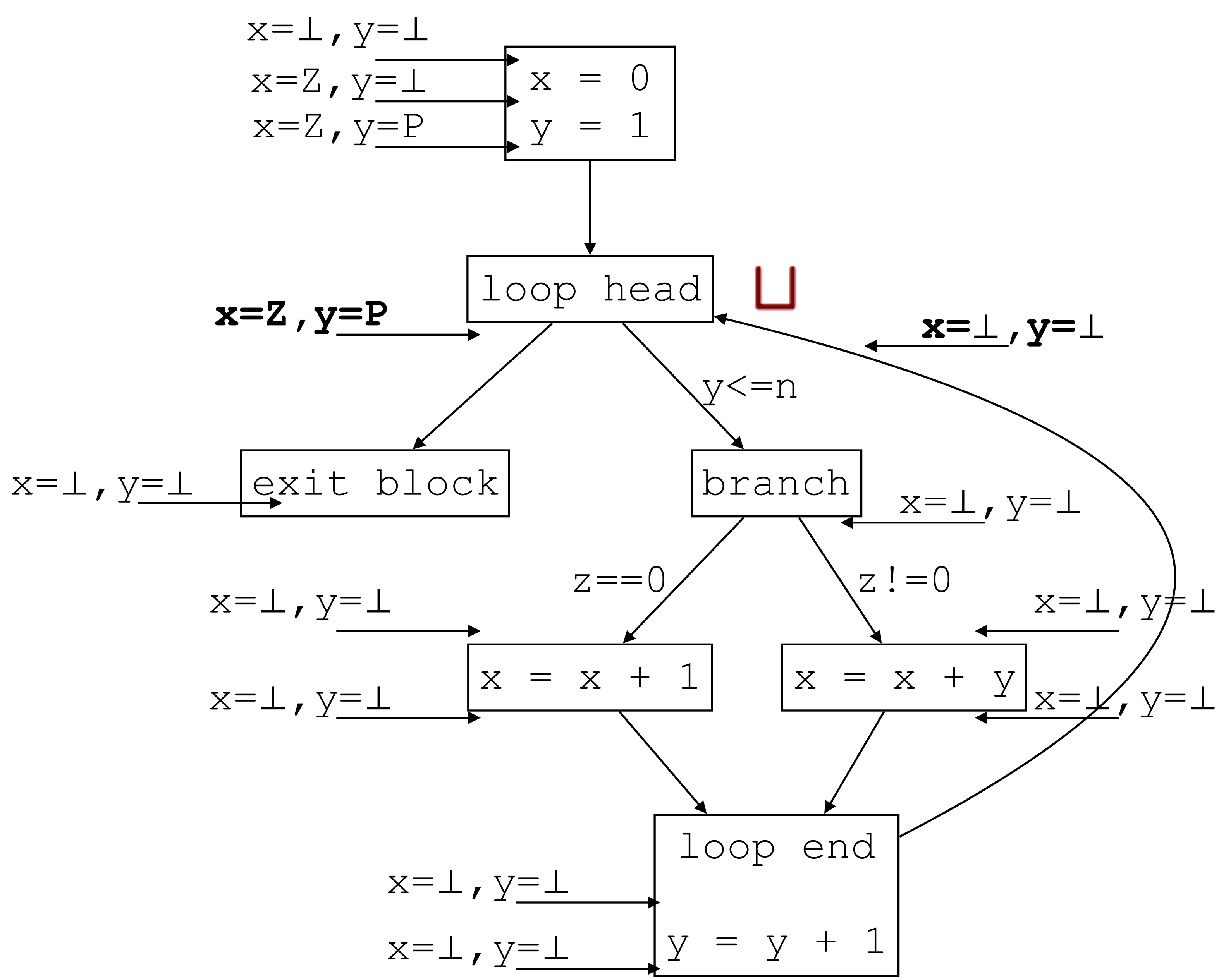
Fixed-point computation



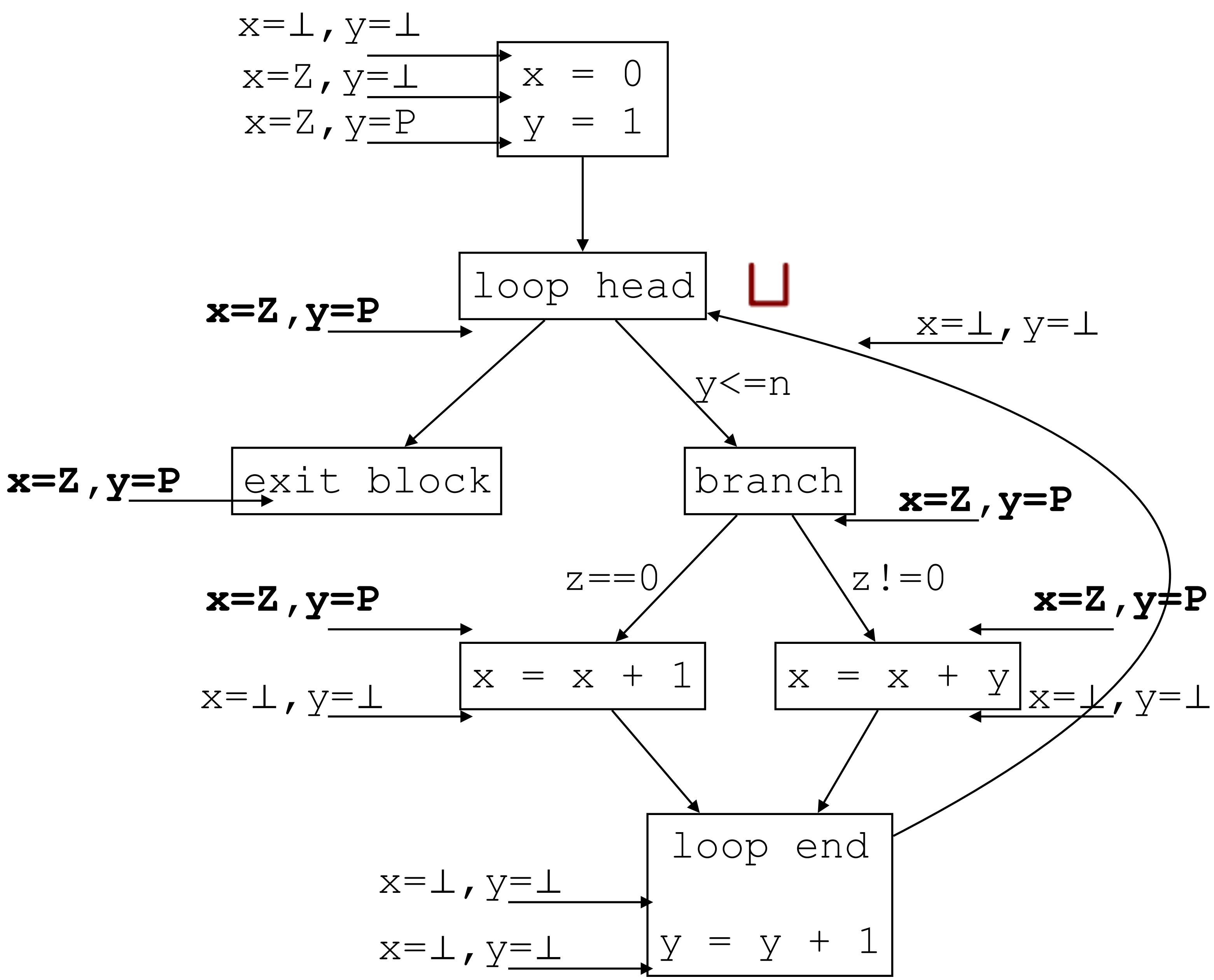
Fixed-point computation



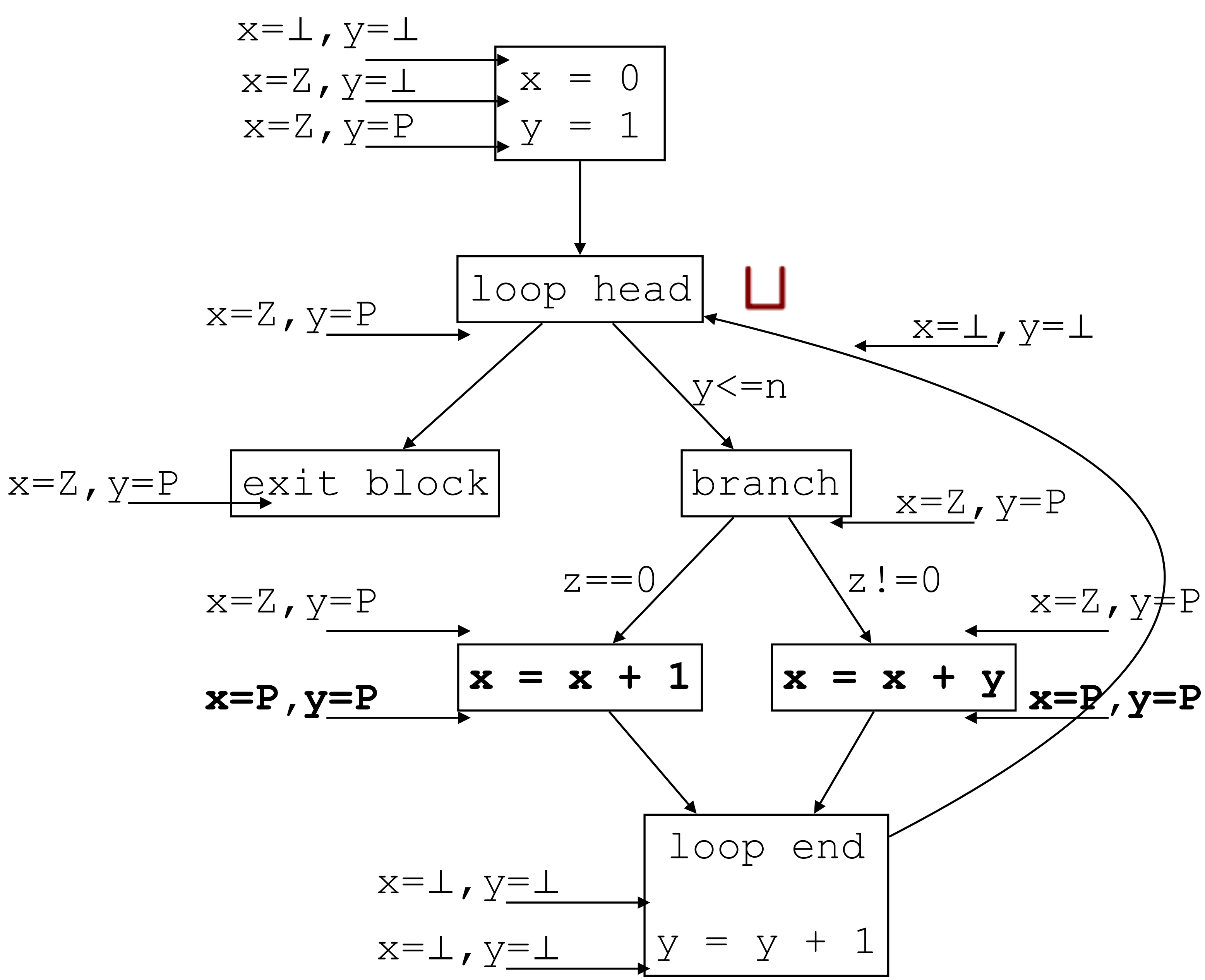
Fixed-point computation



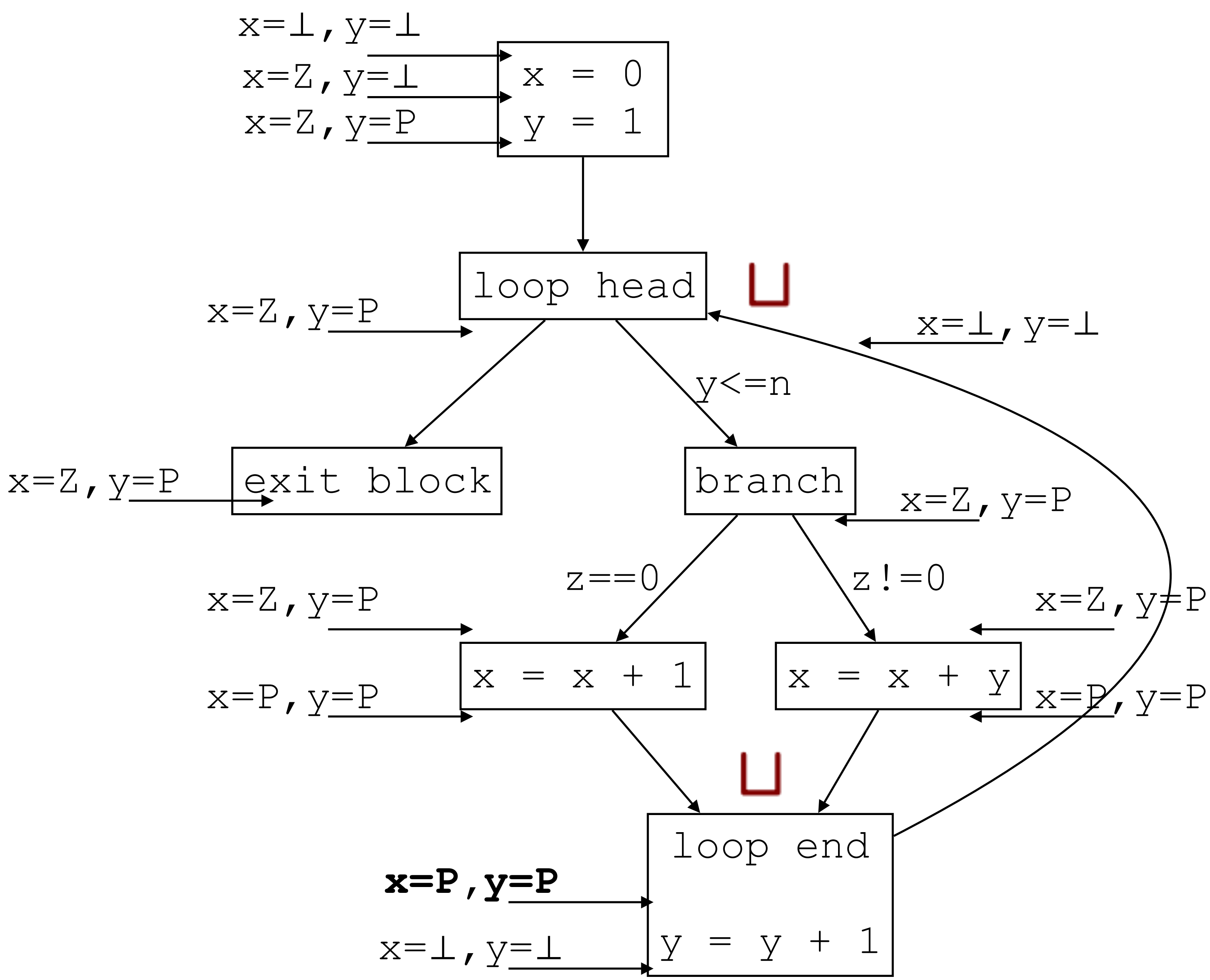
Fixed-point computation



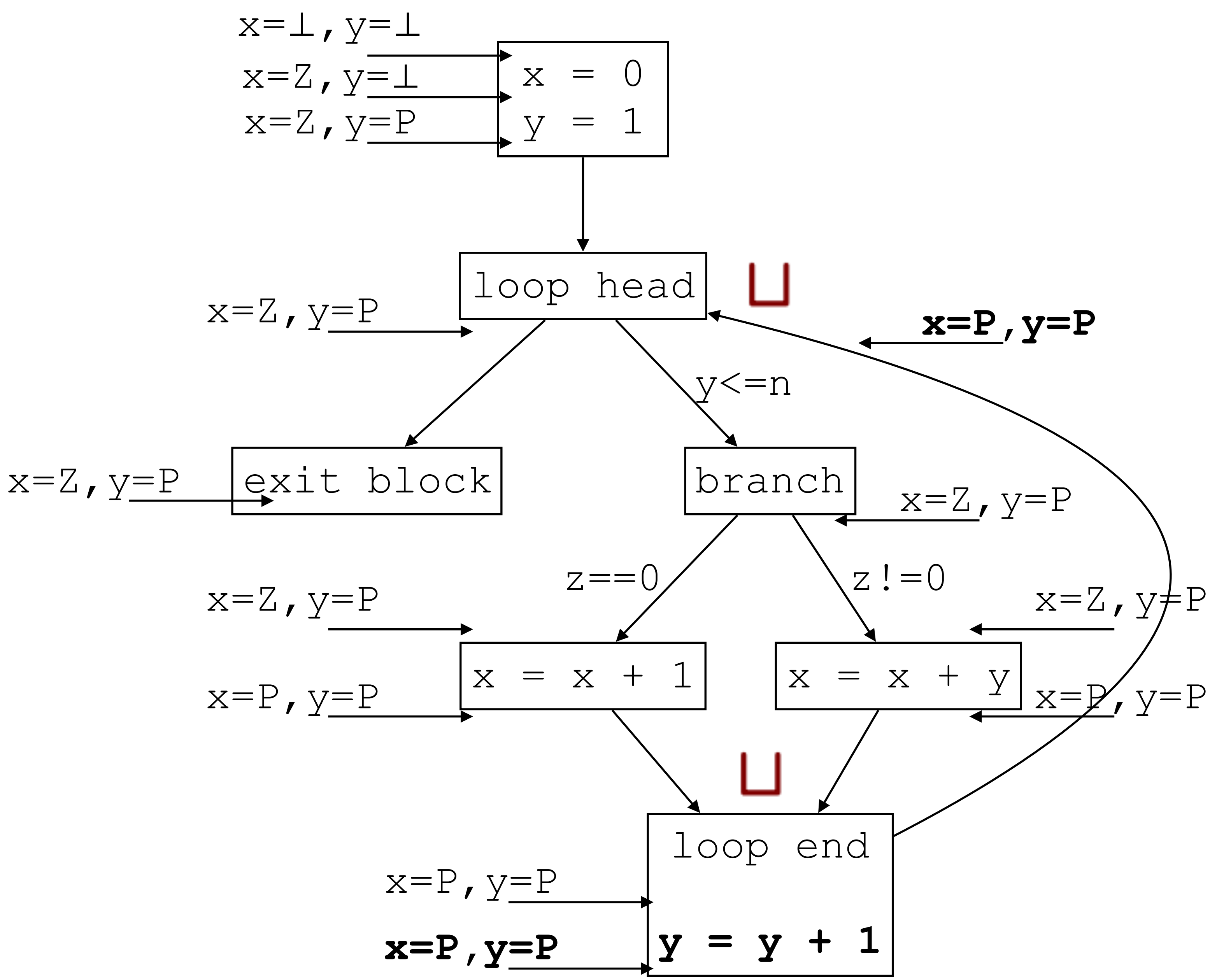
Fixed-point computation



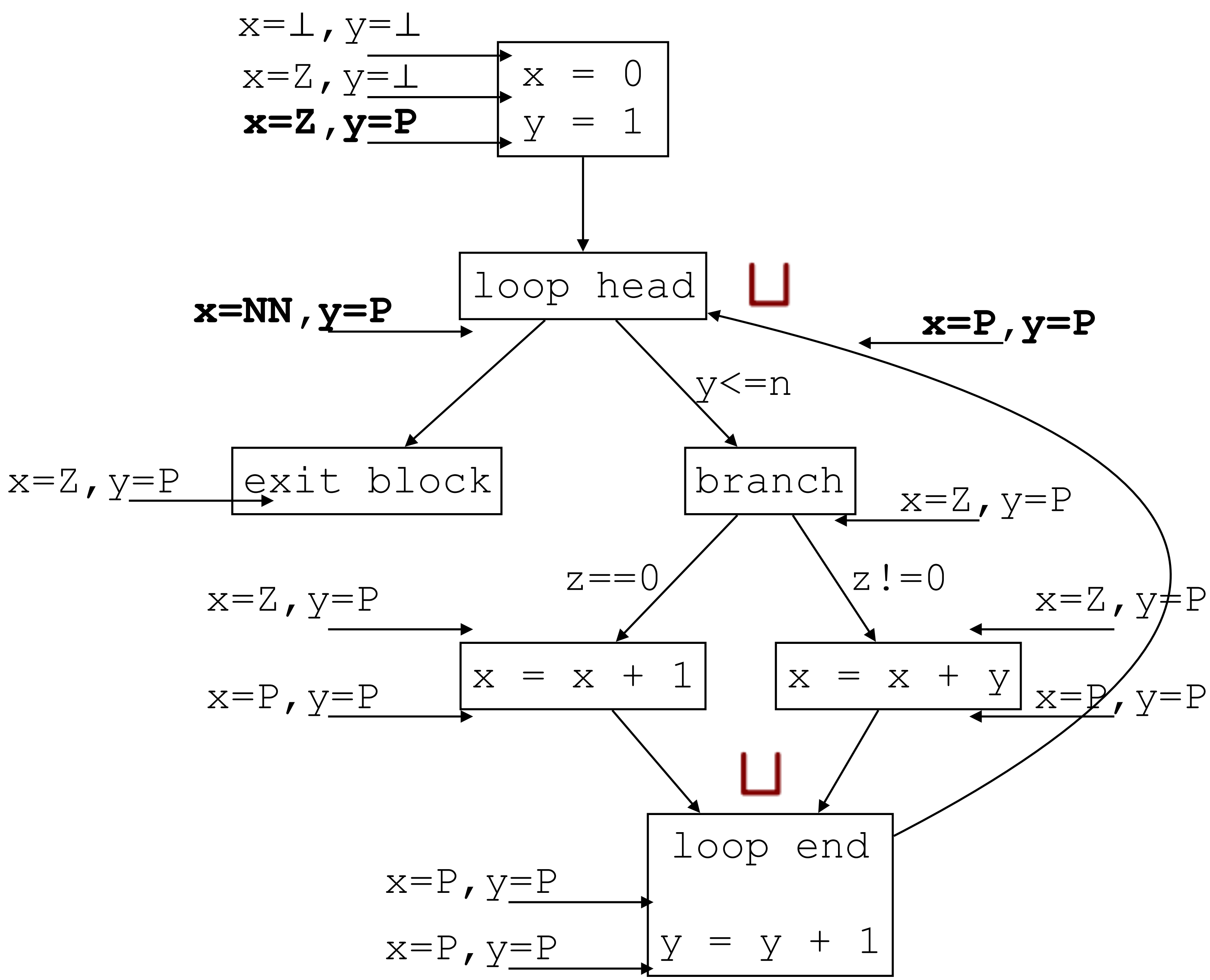
Fixed-point computation



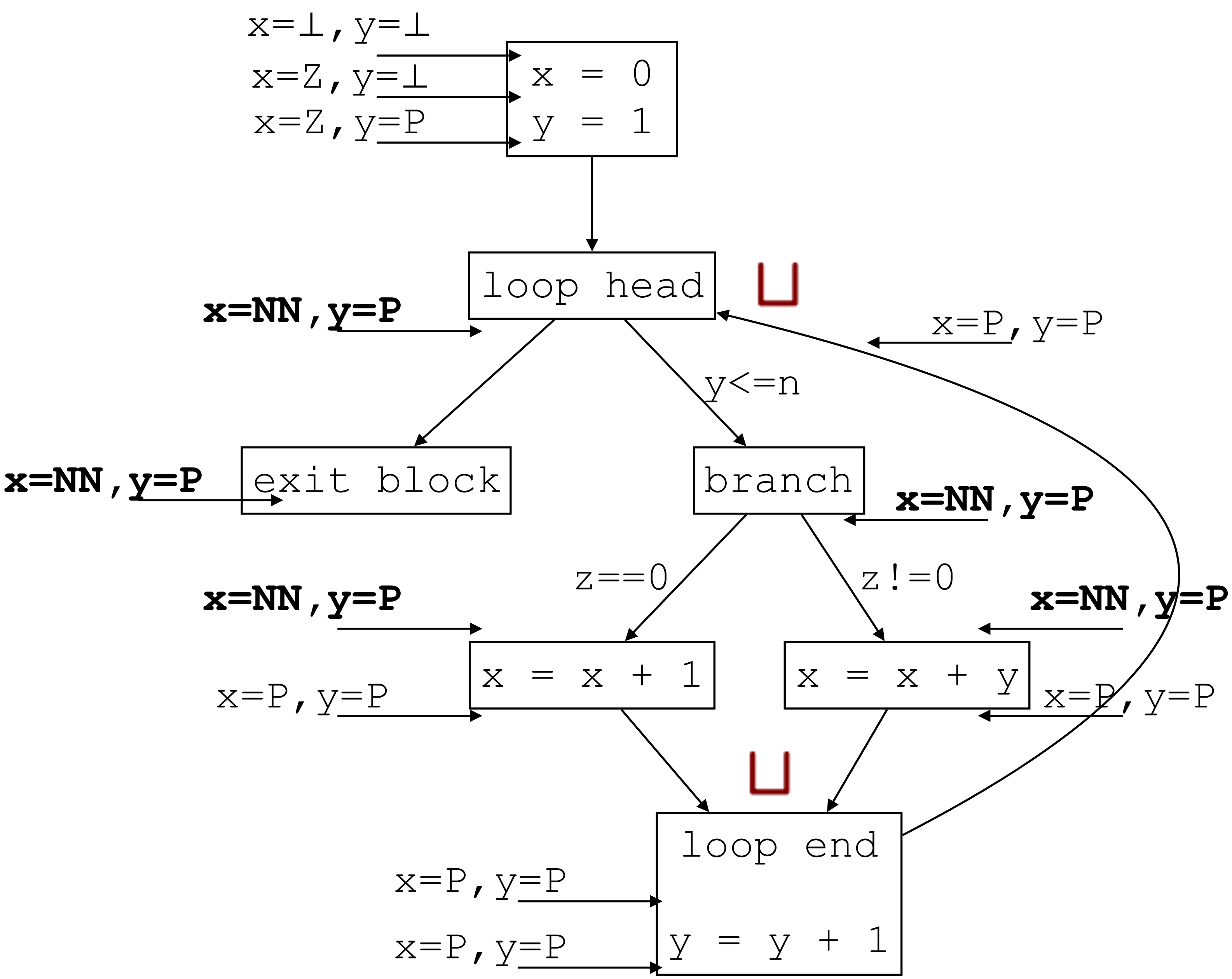
Fixed-point computation



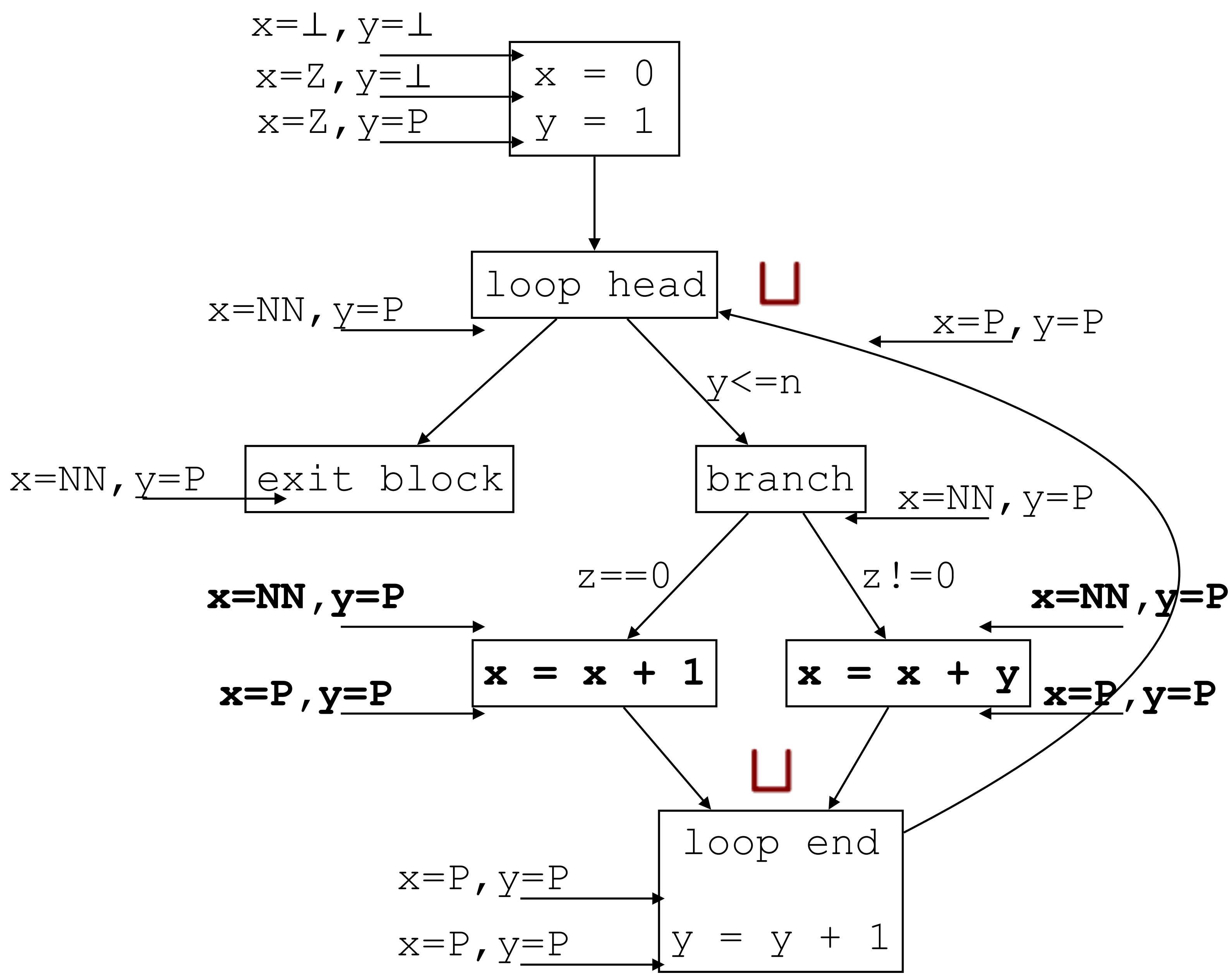
Fixed-point computation



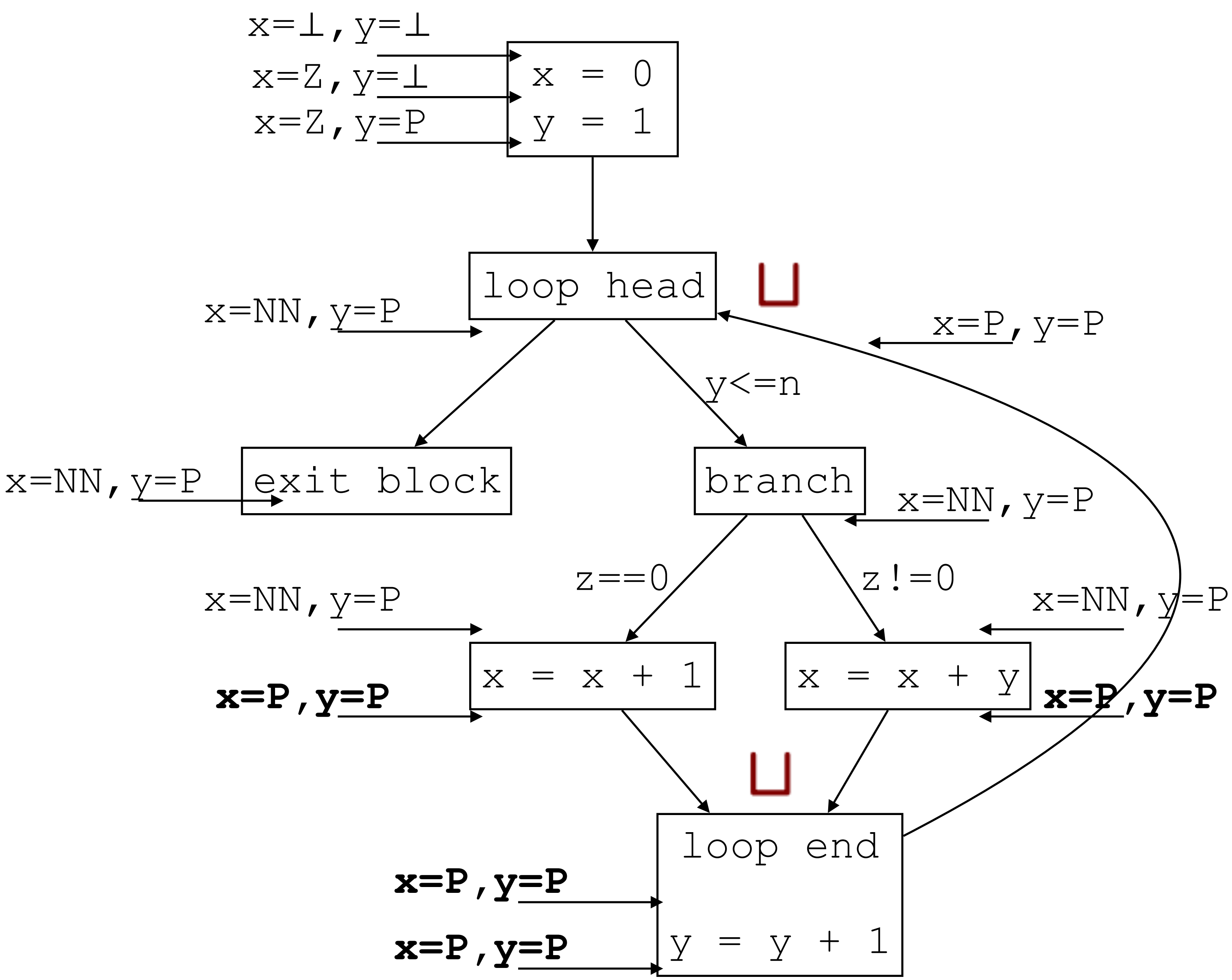
Fixed-point computation



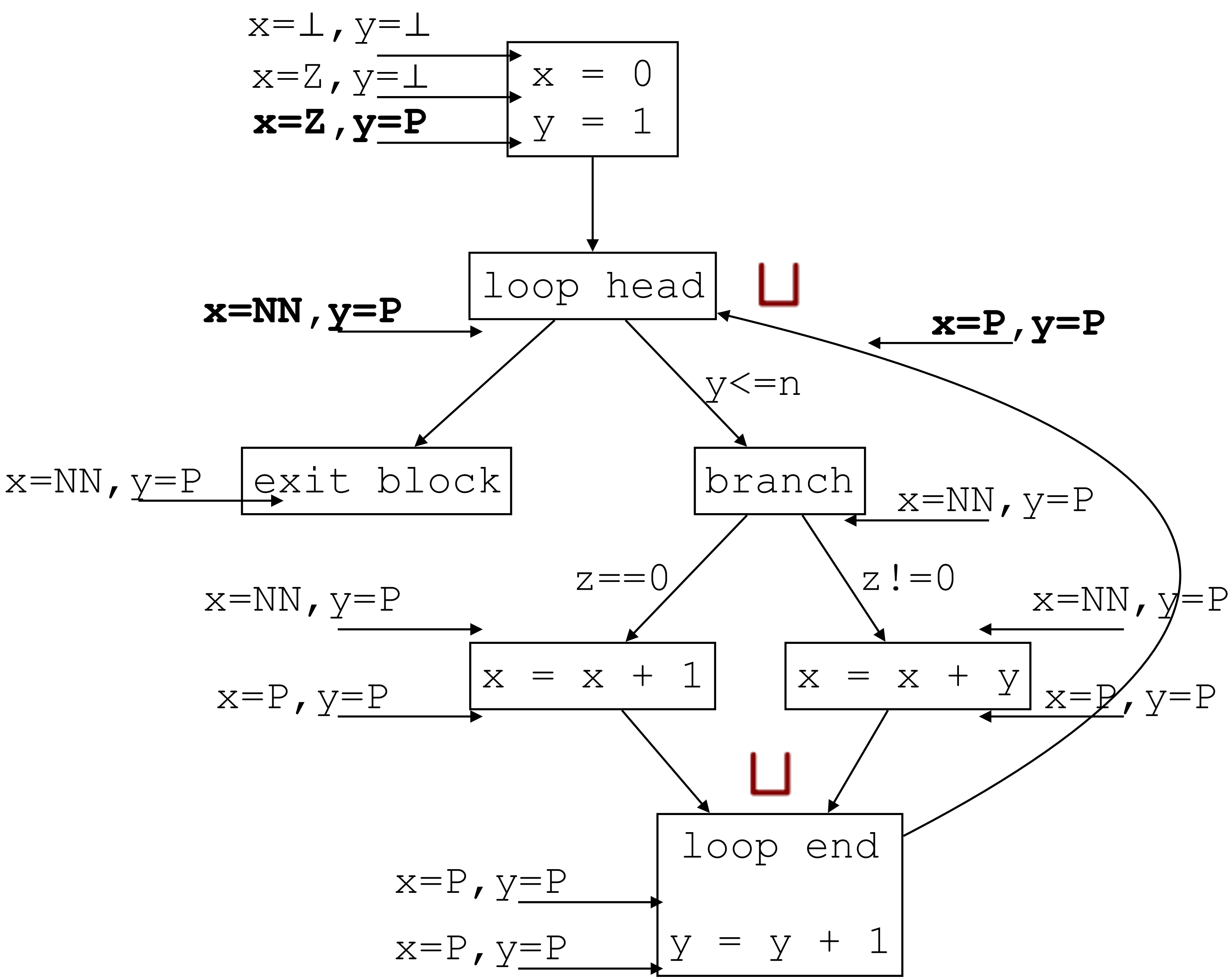
Fixed-point computation



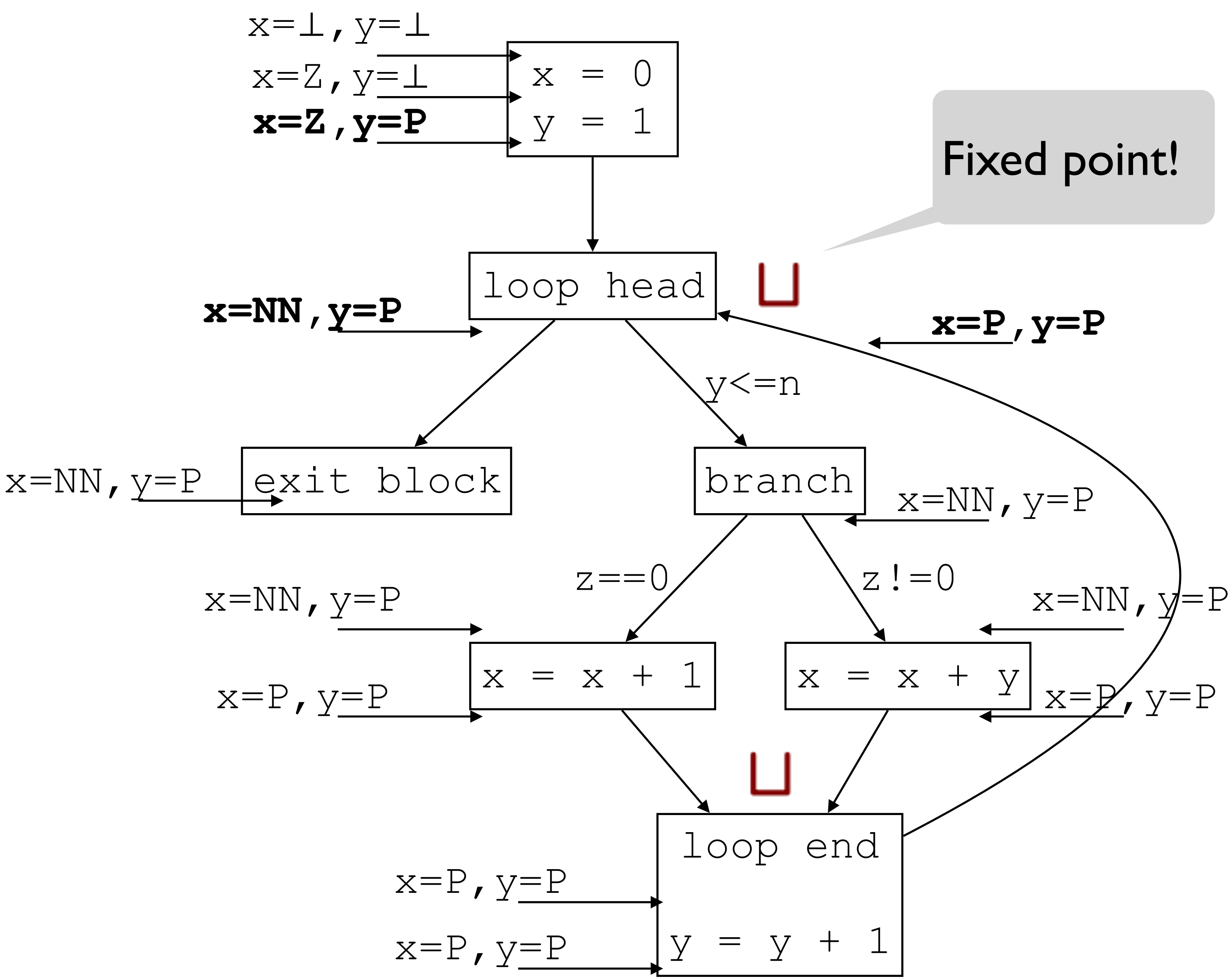
Fixed-point computation



Fixed-point computation



Fixed-point computation



Termination of fixed point computation

- In this example, we quickly reached least fixed point – but does this computation always terminate?
  - Yes, assuming abstract domain forms complete lattice
  - This means every subset of elements (including infinite subsets) have a LUB
- Unfortunately, many interesting domains do not have this property, so we need **widening operators** for convergence.

Take-away lessons

Take-away lessons

- Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:
  - Abstraction: Only reason about certain properties of interest
  - Fixed-point computation: Allows us to obtain sound over-approximation of the program

Take-away lessons

- Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:
  - Abstraction: Only reason about certain properties of interest
  - Fixed-point computation: Allows us to obtain sound over-approximation of the program
- But many static analyses also differ in several ways:
  - Flow (in)sensitivity: Some analyses only compute facts for the whole program, not for every program point
  - Path sensitivity: More precise analyses compute different facts for different program paths
  - Analysis direction: Forwards vs. backwards

Challenges and open problems

Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
- Dealing with open programs
- Modular program analysis
- ...

Challenges and open problems

Many open problems in program analysis

- Precise and scalable heap reasoning
- Concurrency
- Dealing with open programs
- Modular program analysis
- ...

Exciting area with lots of interesting topics to work on!

Thank you!

Showtime on Wednesday!

https://ucsb-plse.slack.com/