

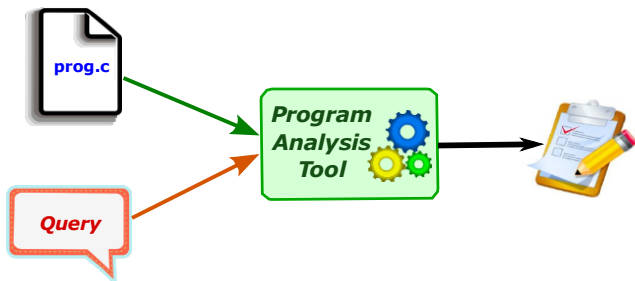
Seminar 14

(not required for the final exam)

**Please discuss a short
Introduction in program
analysis**

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

- **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 vs. 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

- + reasons about all executions
- less precise



Dynamic

- + more precise
- results limited to observed executions

Testing /Formal Verification

A very crude dichotomy:

Testing	Formal Verification
Correct with respect to the <i>set of test inputs</i> , and reference system	Correct with respect to <i>all inputs</i> , with respect to a <i>formal specification</i>
Easy to perform	Decidability problems, Computational problems,
Dynamic	Static

Static Program 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

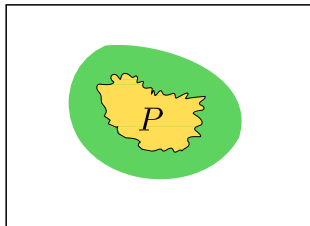
Sound, but incomplete: May say program is unsafe even though it is safe

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 Analyses

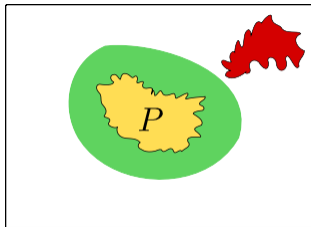
Key idea: Overapproximate (i.e., abstract) program behavior



How to design Sound Static Analyses

Key idea: Overapproximate (i.e., abstract) program behavior

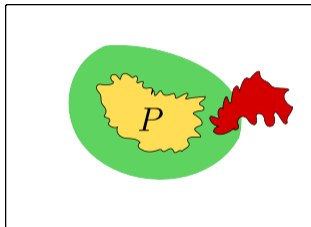
- Bad states outside over-approximation
⇒ Program safe



How to design Sound Static Analyses

Key idea: Overapproximate (i.e., abstract) program behavior

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



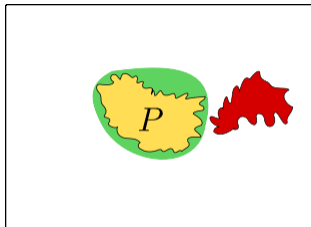
How to design Sound Static Analyses

Key idea: Overapproximate (i.e., abstract) program behavior

- 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	Useful 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 create Sound Abstractions

- 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.)

First Step: 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: $\{x \mid x \in \mathbb{Z} \wedge x > 0\}$
 - zero: $\{0\}$
 - neg: $\{x \mid x \in \mathbb{Z} \wedge x < 0\}$
 - non-neg: $\{x \mid x \in \mathbb{Z} \wedge x \geq 0\}$

First Step: 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: $\{x \mid x \in \mathbb{Z} \wedge x > 0\}$
 - zero: $\{0\}$
 - neg: $\{x \mid x \in \mathbb{Z} \wedge x < 0\}$
 - non-neg: $\{x \mid x \in \mathbb{Z} \wedge x \geq 0\}$

In addition, every abstract domain contains:

- \top (top): "Don't know", represents any value
- \perp (bottom): Represents empty-set

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

Second Step: Abstraction and Concretization Function

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

- $\alpha(\{ 2, 10, 0 \}) = \text{non-neg}$

- $\alpha(\{ 3, 99 \}) = \text{pos}$

- $\alpha(\{ -3, 2 \}) = \text{T}$

Second Step: Abstraction and Concretization Function

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

-

$$\alpha(\{2, 10, 0\}) = \text{non-neg}$$

-

$$\alpha(\{3, 99\}) = \text{pos}$$

-

- $\alpha(\{-3, 2\}) = \top$

Concretization function (γ) maps each abstract value to sets of concrete elements

$$\gamma(\text{pos}) = \{x \mid x \in \mathbb{Z} \wedge x > 0\}$$

Lattices and Abstract Domains

- Concretization function defines **partial order** on abstract values:

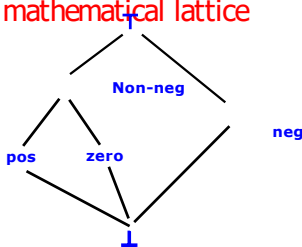
$$A_1 \leq A_2 \text{ iff } \gamma(A_1) \subseteq \gamma(A_2)$$

Lattices and Abstract Domains

- Concretization function defines **partial order** on abstract values:

$$A_1 \leq A_2 \text{ iff } \gamma(A_1) \subseteq \gamma(A_2)$$

- Furthermore, in an abstract domain, every pair of elements has a lub and glb \Rightarrow **mathematical lattice**

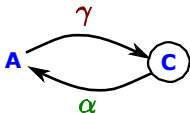


- Least upper bound of two elements is called their **join** – useful for reasoning about control flow in programs

Almost Inverses

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

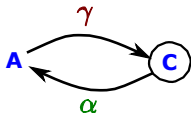
$$\alpha(\gamma(A)) = A$$



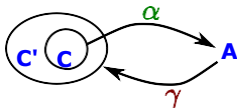
Almost Inverses

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

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



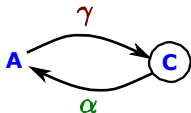
$$C \subseteq \gamma(\alpha(C))$$



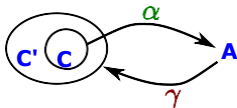
Almost Inverses

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

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



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



- This is called a **Galois insertion** and captures the soundness of the abstraction

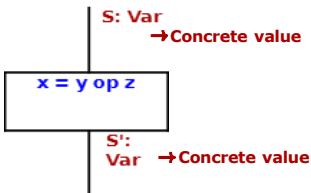
Step 3: Abstract Semantics

- Given abstract domain, α, γ , need to define **abstract transformers (i.e., semantics)** for each statement
- Describes how statements affect our abstraction

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

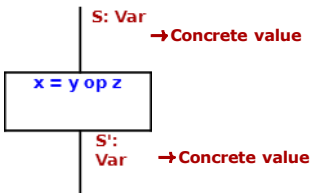
Operational Semantics



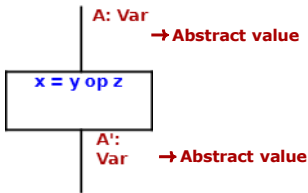
Step 3: Abstract Semantics

- 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



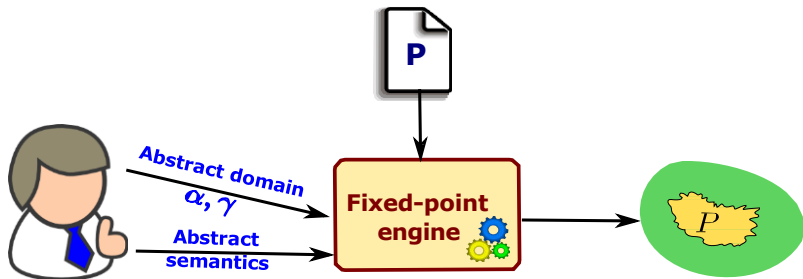
Back to Our Example

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

	pos	neg	zero	non-neg	\top	\perp
pos	pos	\top	pos	pos	\top	\perp
neg	\top	neg	neg	\top	\top	\perp
zero	pos	neg	zero	non-neg	\top	\perp
non-neg	pos	\top	non-neg	non-neg	\top	\perp
\top	\top	\top	\top	\top	\top	\perp
\perp	\perp	\perp	\perp	\perp	\perp	\perp

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

Putting It All Together

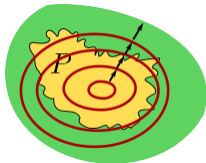


Fixed-point Computations

- **Fixed-point computation:** Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an equilibrium

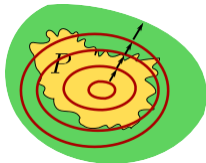
Fixed-point Computations

- **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 underapproximation and grow the approximation until it stops growing



Fixed-point Computations

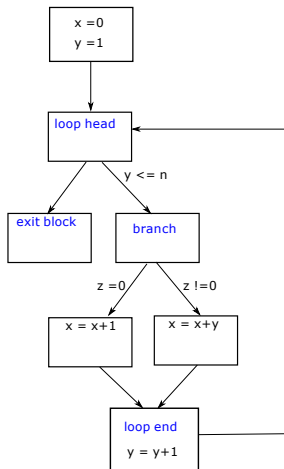
- **Fixed-point computation:** Repeated symbolic execution of the program using abstract semantics until our approximation of the program reaches an equilibrium
- **Least fixed-point:** Start with underapproximation and grow the approximation until it stops growing



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

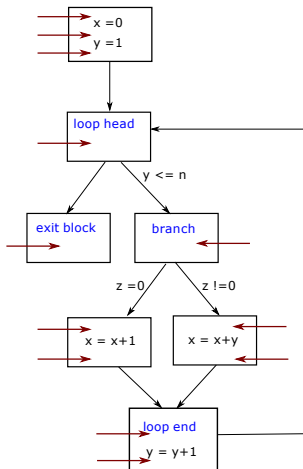
Performing Least Fixed Point Computation

- Represent program as a control-flow graph



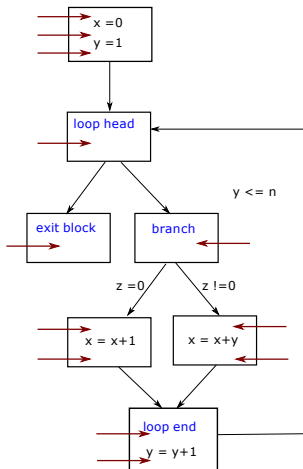
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point



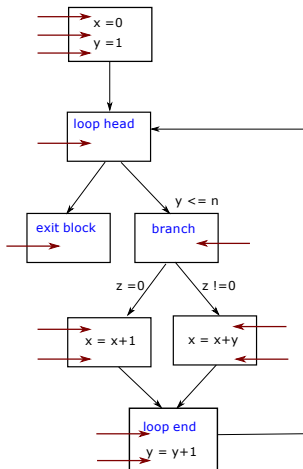
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to \perp



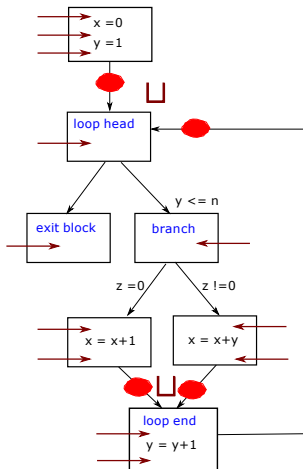
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to \perp
- Repeat until no abstract state changes at any program point:



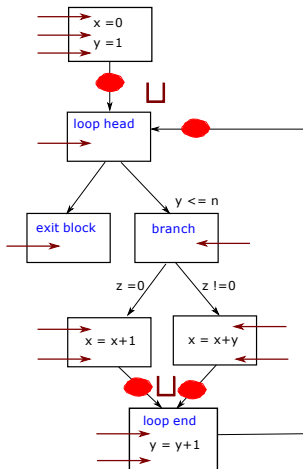
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to \perp
- 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



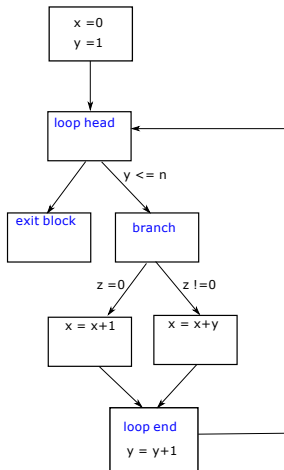
Performing Least Fixed Point Computation

- Represent program as a **control-flow graph**
- Want to compute abstract values at every program point
- Initialize all abstract states to \perp
- 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



An Example

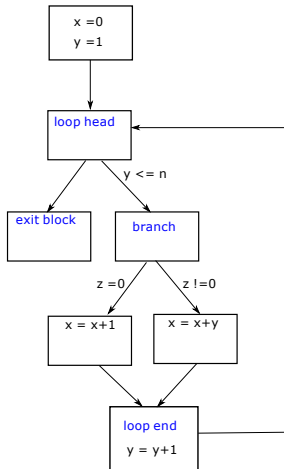
```
x = 0;  
y = 0;  
  
while(y <= n)  
{  
  if (z == 0)  
  { x = x+1;  
  }  
  else {  
    x = x + y;  
  }  
  y = y+1  
}
```



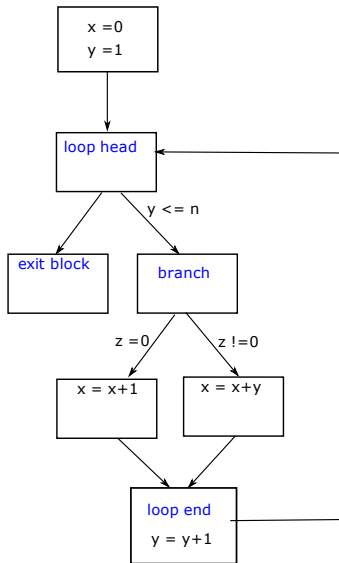
An Example

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

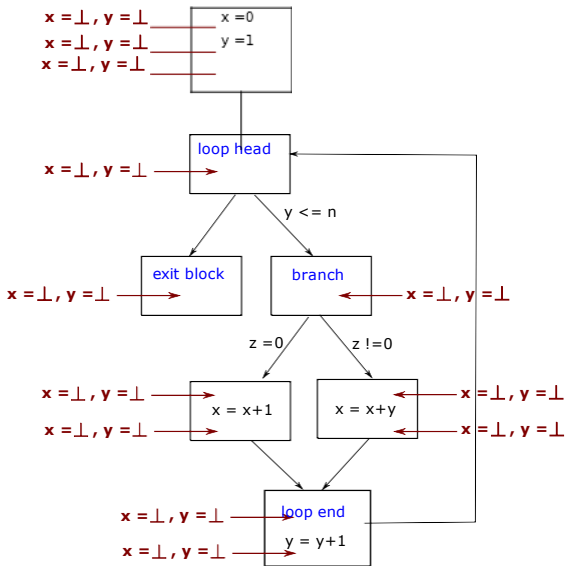
*Is x always
non-negative
inside the loop?*



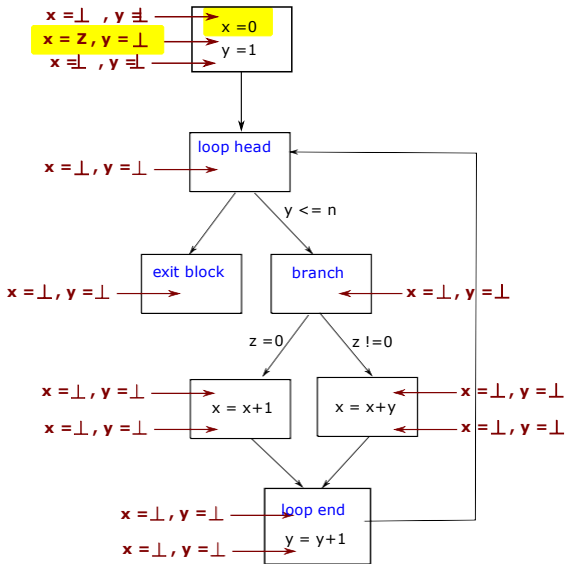
Fixed-Point Computation



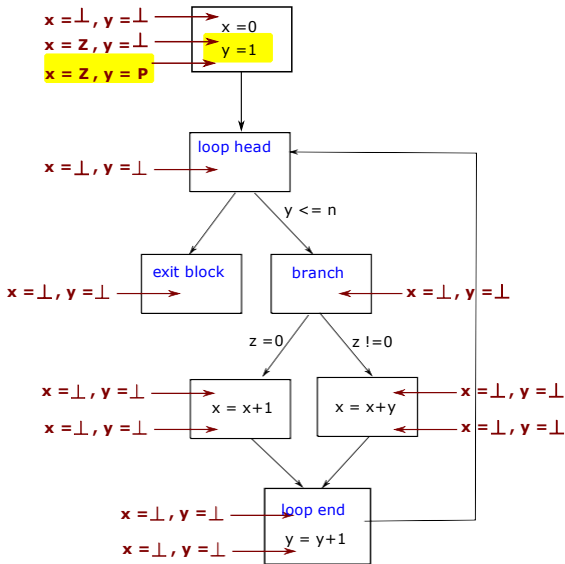
Fixed-Point Computation



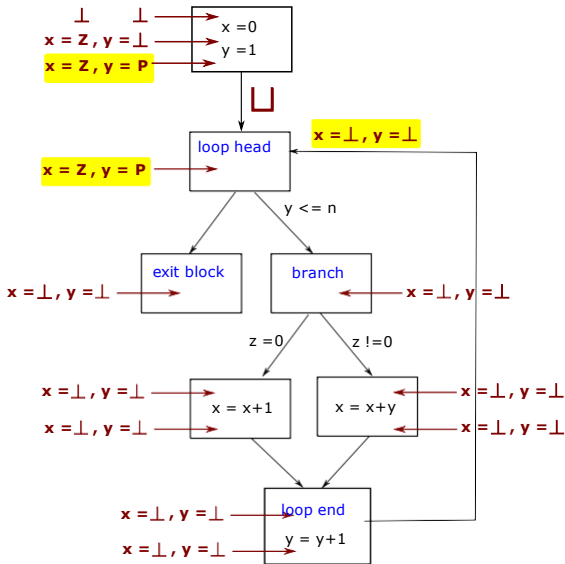
Fixed-Point Computation



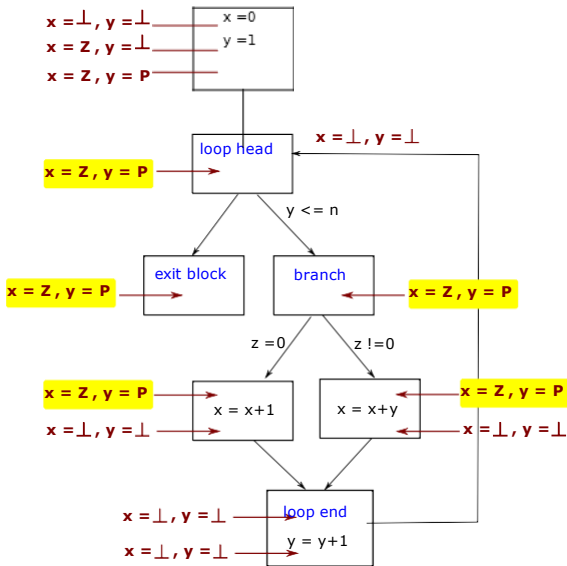
Fixed-Point Computation



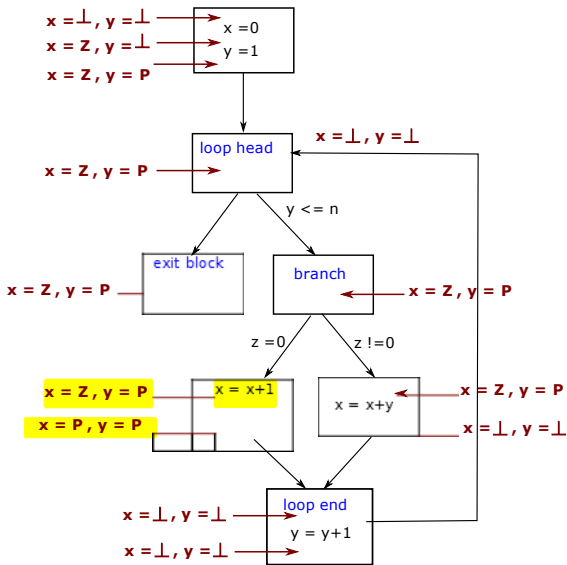
Fixed-Point Computation



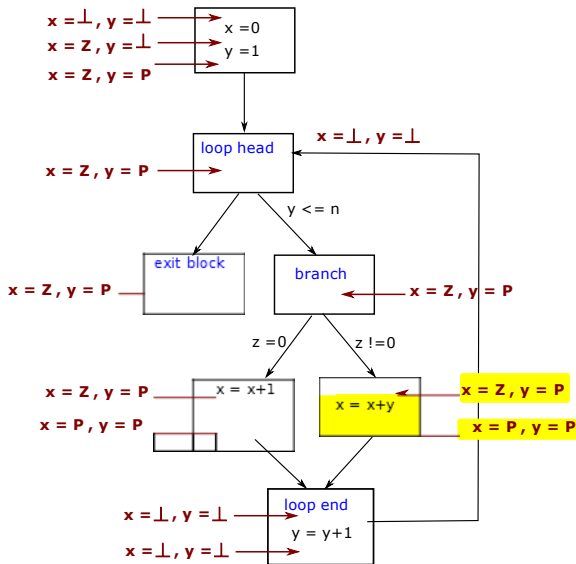
Fixed-Point Computation



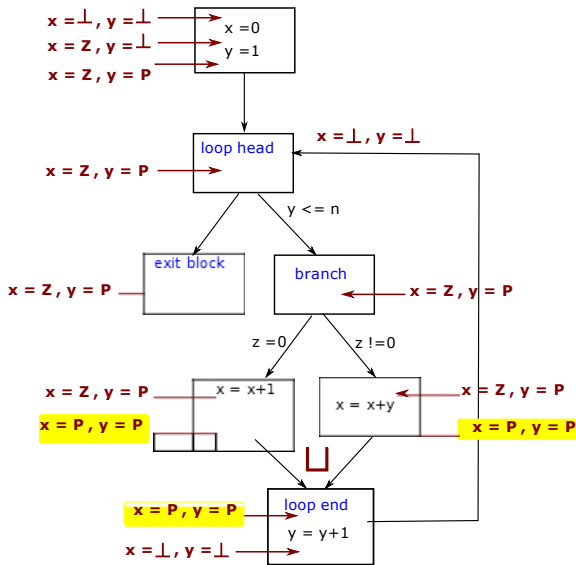
Fixed-Point Computation



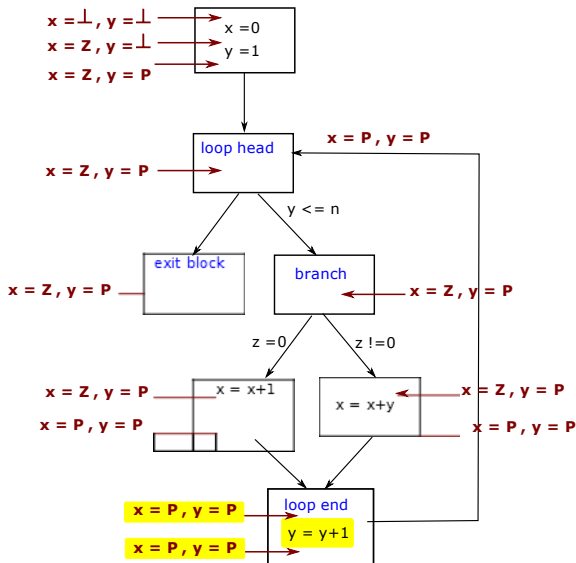
Fixed-Point Computation



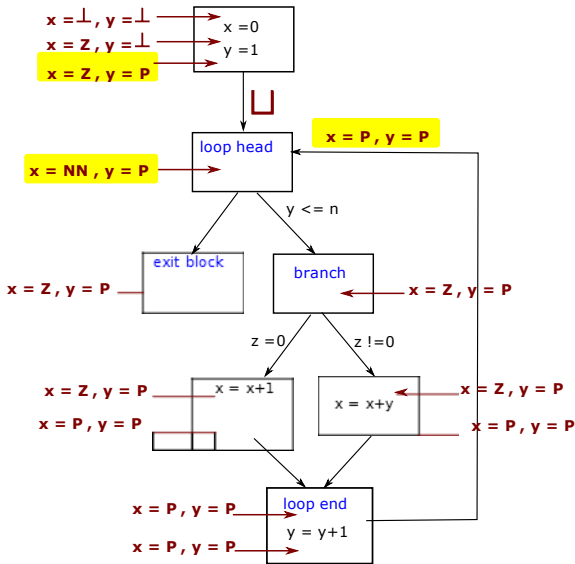
Fixed-Point Computation



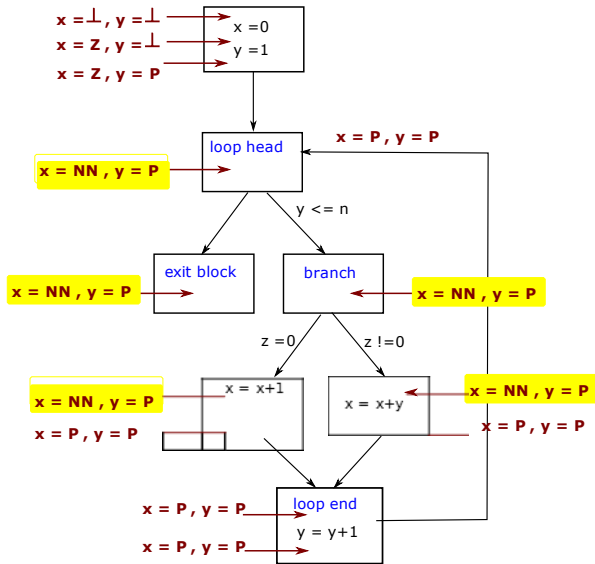
Fixed-Point Computation



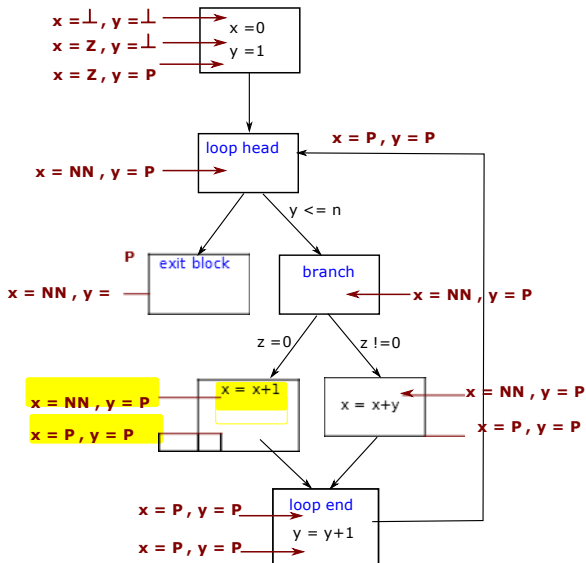
Fixed-Point Computation



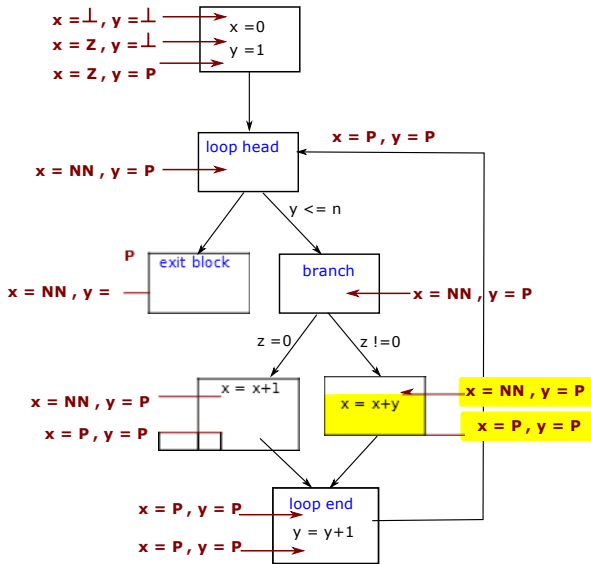
Fixed-Point Computation



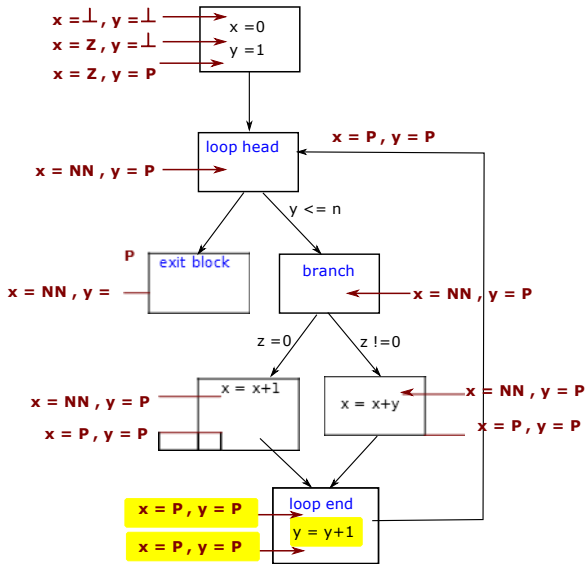
Fixed-Point Computation



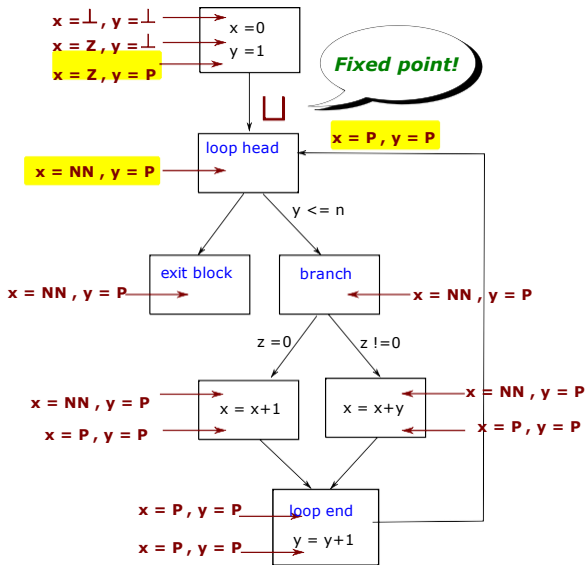
Fixed-Point Computation



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?

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**

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

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.

Lessons To Take Away

- Considered only one static analysis approach, but illustrates two key ideas underlying program analysis:
 - **Abstraction:** Only reason about certain properties of interest

Lessons To Take Away

- 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 overapproximation of the program

Lessons To Take Away

- 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 overapproximation 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

Lessons To Take Away

- 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 overapproximation 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

Challenges and Open Problems

Many open problems

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